Welcome to NLP 2025

From N-grams to Transformers: Your Journey Begins Today

NLP Course 2025

First Day of Class

Natural Language Processing: 12 Weeks to Understanding ChatGPT

What You'll Build This Semester

From Zero to Transformer

Week 2:
- Word embeddings from scratch
- Train on real text
- Visualize word similarities

Week 3:
- LSTM language model
- Generate Shakespeare
- Understanding memory

Week 5:
- **Complete transformer**
- Self-attention mechanism
- The architecture behind ChatGPT

Week 6:
- Fine-tune BERT
- Transfer learning
- State-of-the-art results

Week 9:
- Text generation strategies
- Control creativity
- Beam search and sampling

Week 12:
- Deploy responsibly
- Measure and mitigate bias
- **Ethical AI**

By Week 5, you'll understand how ChatGPT, Claude, and all modern LLMs work

NLP is Transforming Every Industry

Current Applications:

- **ChatGPT & Claude** - Conversational AI
- **GitHub Copilot** - Code generation
- **Google Translate** - 100+ languages
- **Grammarly** - Writing assistance
- **Alexa & Siri** - Voice assistants
- **Gmail** - Smart compose

Career Impact:

- NLP engineers: Top 5 percent salaries
- ML researchers: High demand
- Data scientists: Essential skill
- Product managers: Understanding capability
- All roles: AI literacy critical

Market Growth:

- NLP market: USD 20B (2024)
- Projected: USD 100B+ (2030)
- AI investment: USD 500B+ annually

Understanding transformers is understanding the future of computing

Learn by Building, Not Just Watching

Discovery-Based Learning:

- Start with concrete problems
- Discover solutions yourself
- THEN learn the formal theory
- Build intuition first

Progressive Complexity:

- Week 1: Count words
- Week 5: Build transformer
- Week 12: Deploy ethically
- Each week builds on previous

Hands-On Practice:

- 12 lab notebooks
- Real code, real data
- Run in your browser
- No magic black boxes

Real Implementation:

- PyTorch from scratch
- Every line explained
- No hidden abstractions
- Understanding not memorization

By the end, you'll have IMPLEMENTED a transformer, not just understood it

Three Phases of Mastery

Phase 1: Foundations
Weeks 1-4

Week 1: Statistical LM
N-grams, probability
Week 2: Word Embeddings
Word2Vec, GloVe
Week 3: RNN/LSTM
Sequential models
Week 4: Seq2Seq
Attention mechanism

Master: Neural basics

Phase 2: Revolution
Weeks 5-8

Week 5: Transformers
Self-attention
Week 6: Pre-trained
BERT, GPT
Week 7: Advanced
T5, GPT-3, scaling
Week 8: Tokenization
BPE, WordPiece

Master: Modern architectures

Phase 3: Application
Weeks 9-12

Week 9: Decoding
Generation strategies
Week 10: Fine-tuning
Adaptation methods
Week 11: Efficiency
Optimization
Week 12: Ethics
Responsible deployment

Master: Production deployment

Each phase unlocks new capabilities - by Week 12, you're deployment-ready

Self-Assessment

Required (Must Have):

- Python programming experience
- Basic linear algebra (vectors, matrices)
- Basic probability theory
- Comfortable with loops and functions
- Can read and write code

Helpful But Not Required:

- PyTorch or TensorFlow
- Neural networks basics
- Machine learning concepts
- Backpropagation understanding
- GPU programming

If You Don't Have These:

- Python: Complete Python tutorial first
- Math: Review Khan Academy linear algebra
- Probability: Review basic probability

We Provide:

- Neural Network Primer (if needed)
- LSTM Primer (deep dive)
- Progressive difficulty
- Office hours support

New to neural networks? Start with our Neural Network Primer before Week 2

Complete Open-Source Course

Presentations:

- 60+ slide decks
- Optimal readability design
- Multiple versions (BSc, enhanced)
- PDF and LaTeX source

Lab Notebooks:

- 12 interactive Jupyter notebooks
- Hands-on implementation
- Real data and models
- Run in your browser

Handouts:

- Pre-class discovery exercises
- Post-class technical practice
- Student and instructor versions

Supplementary:

- Neural Network Primer
- LSTM Primer (32 slides)
- Word Embeddings Module
- 8 visualization notebooks

Documentation:

- Complete installation guide
- Week-by-week course index
- Troubleshooting support
- Project templates

All Materials:

- Open source (MIT license)
- GitHub repository
- Always accessible

Your First Assignment (Due Next Class)

Step 1: Clone Repository

- `git clone github.com/josterri/2025_NLP_Lectures.git`
- `cd 2025_NLP_Lectures`

Step 2: Install Dependencies

- `pip install -r requirements.txt`

Step 3: Verify Installation

- `python verify_installation.py`

Expected Time:

- Clone: 2 minutes
- Install: 10-15 minutes
- Verify: 30 seconds

Step 4: Test First Notebook

- Launch: `jupyter lab`
- Open: Week 2 word embeddings lab
- Run first 3 cells
- Verify imports work

GPU Optional:

- Weeks 1-4: CPU sufficient
- Weeks 5+: GPU recommended
- All labs work on CPU
- GPU speeds up training

Get Help:

- Read INSTALLATION.md
- Check GitHub Issues
- Office hours: TBD

What Happens Each Week

Before Class (Monday):

- Read handout (pre-class section)
- Complete discovery exercises
- Build intuition
- Identify questions

During Lecture (Wednesday):

- Review key concepts
- Live coding demonstrations
- Q&A on pre-class material
- Preview lab notebook

Lab Session (Friday):

- Implement concepts
- Hands-on coding
- Get help from TAs

After Lab (Weekend):

- Complete lab notebook
- Finish post-class handout
- Experiment and explore
- Submit by Monday

Office Hours (TBD):

- Individual help
- Concept clarification
- Project discussion
- Career advice

Weekly Commitment:

- Lecture: 2 hours
- Lab: 2 hours
- Independent work: 4-6 hours

Grading Breakdown

Lab Notebooks (40 percent):

- 12 notebooks, 11 graded (drop lowest)
- Due Monday after lab
- Implementation correctness
- Code quality and comments
- Experimentation and insights

Midterm Project (25 percent):

- After Week 6
- Implement and evaluate model
- Written report
- Code submission

Final Project (30 percent):

- Weeks 11-12
- Original NLP application

Participation (5 percent):

- Class attendance
- Lab participation
- Office hours engagement
- Helping classmates

Grading Philosophy:

- Focus on learning not perfection
- Partial credit for attempts
- Bonus for creativity
- Collaboration encouraged (cite sources)

Late Policy:

- 48-hour grace period (no penalty)
- After that: 10 percent per day
- Max 5 days late

What Past Students Built

Text Generation:

- **Poetry generator** - Style transfer
- **Code completion** - GitHub-trained
- **Story writer** - Character-consistent
- **Email composer** - Professional tone

Information Extraction:

- **Resume parser** - Extract skills
- **News summarizer** - Multi-document
- **Question answering** - Domain-specific
- **Fact checker** - Claim verification

Classification:

- **Sentiment analysis** - Product reviews
- **Spam detection** - Email filtering
- **Intent classification** - Chatbot

Translation & Multilingual:

- **Domain translator** - Legal/medical
- **Code-switching** - Multilingual text
- **Style transfer** - Formal to casual
- **Dialect conversion** - Regional

Creative Applications:

- **Music from text** - Lyrics to melody
- **Debate partner** - Argument generation
- **Language learning** - Adaptive tutor
- **Game dialogue** - NPC conversations

Research Projects:

- **Bias measurement** - Gender/race
- **Interpretability** - Attention analysis
- **Efficiency** - Model compression

From Zero to Transformer in 5 Weeks

Technical Understanding:

- How attention mechanism works
- Query, Key, Value matrices
- Multi-head attention computation
- Positional encoding mathematics
- Layer normalization
- Feed-forward networks
- Residual connections

Implementation Skills:

- Scaled dot-product attention
- Multi-head splitting and merging
- Complete transformer block
- Positional encoding
- Training loop

Practical Abilities:

- Debug attention weights
- Visualize attention patterns
- Optimize hyperparameters
- Compare with RNN/LSTM
- Understand GPT architecture
- Read research papers

Real-World Knowledge:

- Why ChatGPT is so powerful
- How BERT differs from GPT
- Computational complexity trade-offs
- When to use transformers
- Current limitations
- Future directions

Frequently Asked Questions

Q: Do I need a GPU?
A: No, all labs work on CPU. GPU speeds things up for Weeks 5+ but isn't required.

Q: How much Python?
A: Comfortable writing functions, loops, and classes. PyTorch will be taught.

Q: Mathematical background?
A: Linear algebra (vectors, matrices, dot products) and basic probability. We'll review as needed.

Q: Can I audit?
A: Yes! All materials are open source. You won't get grades but can follow along.

Q: Group projects?
A: Labs are individual. Final project can be pairs with instructor approval.

Q: How hard is this?
A: Challenging but doable. Budget 10 hours/week. We support you every step.

Q: What if I fall behind?
A: Office hours, catch-up sessions, and grace periods. Communication is key.

Q: Industry vs research?
A: Both! Course covers practical deployment and research methods.

Q: After this course?
A: You'll be ready for NLP engineering roles, ML research, or advanced courses.

Q: Most important week?
A: Week 5 (transformers) is the foundation. Don't miss it!

How Top Students Excel

Before Each Week:

- Read pre-class handout carefully
- Try exercises before looking at solutions
- Write down questions
- Review previous week's concepts

During Lab:

- Start early (don't wait until deadline)
- Experiment beyond requirements
- Ask questions immediately
- Help classmates (best way to learn)
- Save your work frequently

Study Groups:

- Form groups of 3-4 students
- Meet weekly to discuss concepts

Debugging Mindset:

- Read error messages carefully
- Print intermediate outputs
- Test with small examples first
- Check dimensions and shapes
- Use debugger or print statements

Going Deeper:

- Read suggested papers
- Try bonus challenges
- Implement variants
- Share findings with class

Stay Organized:

- Keep notes for each week
- Maintain code repository

Where to Get Help

Course Materials:

- **GitHub:** github.com/josterri/2025_NLP_Lectures
- **Documentation:** README.md, COURSE_INDEX.md
- **Installation:** INSTALLATION.md
- **Syllabus:** SYLLABUS.md

Getting Help:

- **Office Hours:** TBD
- **Email:** TBD
- **Discussion Forum:** TBD
- **Lab TAs:** Available during lab sessions

External Resources:

- **PyTorch Tutorials:** pytorch.org/tutorials
- **Papers:** arxiv.org

Community:

- **GitHub Issues:** Report bugs
- **Pull Requests:** Contribute improvements
- **Discussions:** Share insights

Reference Books:

- Speech and Language Processing (Jurafsky & Martin)
- Neural Network Methods for NLP (Goldberg)
- Dive into Deep Learning (Zhang et al)
- All available free online

Tools:

- **Google Colab:** Free GPU notebooks
- **Hugging Face:** Pre-trained models
- **Weights & Biases:** Experiment tracking

What to Do Right Now

Today (Next 30 Minutes):

1. Clone the repository
2. Read the README.md
3. Check system requirements
4. Identify if you need GPU

Tonight (1-2 Hours):

1. Install dependencies (requirements.txt)
2. Run verify_installation.py
3. Fix any installation issues
4. Read INSTALLATION.md if stuck

Before Next Class:

1. Complete Neural Network Primer (if needed)
2. Read Week 2 pre-class handout
3. Try the discovery exercises
4. Prepare questions

Optional (For Eager Students):

1. Browse Week 2 lab notebook
2. Read Week 1 presentation
3. Check out supplementary modules
4. Join discussion forum

Most important: Get your environment working BEFORE Week 2

Why Transformers Changed Everything

Before Transformers (Pre-2017):

- Sequential processing (RNNs)
- Slow training (hours to days)
- Limited context (hundreds of words)
- Vanishing gradient problems
- Hard to parallelize
- Specialized architectures per task

Limitations We Hit:

- Couldn't scale to large models
- Couldn't use massive datasets
- Couldn't capture long-range dependencies
- Transfer learning was limited

After Transformers (2017+):

- Parallel processing (attention)
- Fast training (hours on GPUs)
- Unlimited context (thousands of tokens)
- Stable gradients
- Massively parallelizable
- Universal architecture

What Became Possible:

- GPT-3: 175 billion parameters
- GPT-4: Multimodal understanding
- Claude: 100k+ token context
- ChatGPT: Conversational AI
- All modern LLMs use transformers

You'll learn WHY this architecture unlocked the AI revolution

By the End of This Course

Technical Skills:

- Implement transformers from scratch
- Fine-tune BERT and GPT models
- Debug attention mechanisms
- Optimize model efficiency
- Deploy models responsibly
- Measure and mitigate bias

Conceptual Understanding:

- How ChatGPT works internally
- Why attention beats RNNs
- When to use which architecture
- Trade-offs in model design
- Current research frontiers

Practical Abilities:

- Read and implement research papers
- Design NLP systems
- Evaluate model performance
- Choose appropriate methods
- Debug complex models
- Communicate technical concepts

Career Readiness:

- NLP engineer interviews
- ML research positions
- Data science roles
- PhD program preparation
- Startup technical co-founder
- Technical leadership

Your First Hands-On Lab

What You'll Learn:

- Words as vectors
- Semantic similarity
- Word2Vec algorithm (CBOW & Skip-gram)
- Training on real text
- Visualizing embeddings
- Analogies: king - man + woman = queen

What You'll Build:

- Train word embeddings from scratch
- Implement Skip-gram model
- Visualize word relationships
- Discover semantic patterns
- Test word analogies

Prepare By:

- Reading pre-class handout
- Understanding distributional hypothesis
- Reviewing basic neural networks
- (Optional) Neural Network Primer

Lab Highlights:

- Work with real text corpus
- See embeddings evolve during training
- Interactive visualizations
- Explore semantic relationships
- Bonus challenges available

Week 2 is where the magic starts - you'll be amazed how much computers can learn about word meanings

Welcome to NLP 2025

Key Takeaways:

- **Build transformers** - Understand how ChatGPT works
- **12-week journey** - From foundations to deployment
- **Hands-on learning** - Implement everything yourself
- **Real applications** - Deploy models responsibly
- **Complete support** - Extensive materials and help

First Assignment:

Install environment by next class
Run verify_installation.py successfully
Read Week 2 pre-class handout

Foundations of NLP

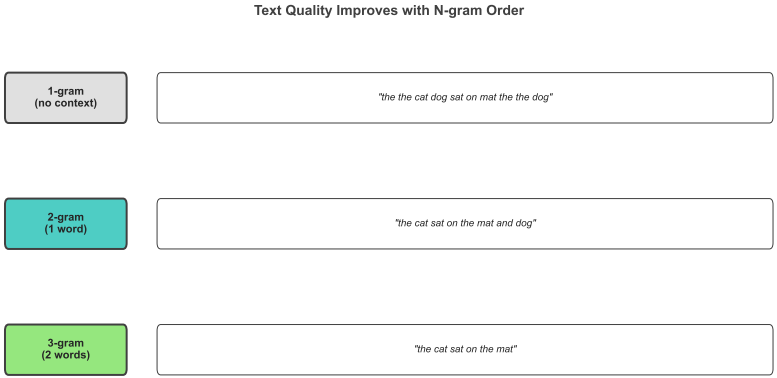
Week 1 - From Dice to Text Prediction

NLP Course 2025

October 26, 2025

BSc Discovery-Based Presentation

Same Model, Different Text Quality



Key Insight: Better models predict next words more accurately

Your phone keyboard uses these models - but how do they work?

A Thought Experiment

Rolling a Die

- Six possible outcomes: 1, 2, 3, 4, 5, 6
- Each has probability: $\frac{1}{6} = 0.167$
- No memory: Previous rolls don't matter
- Fair die: All outcomes equally likely

Question: Can we predict the next roll?

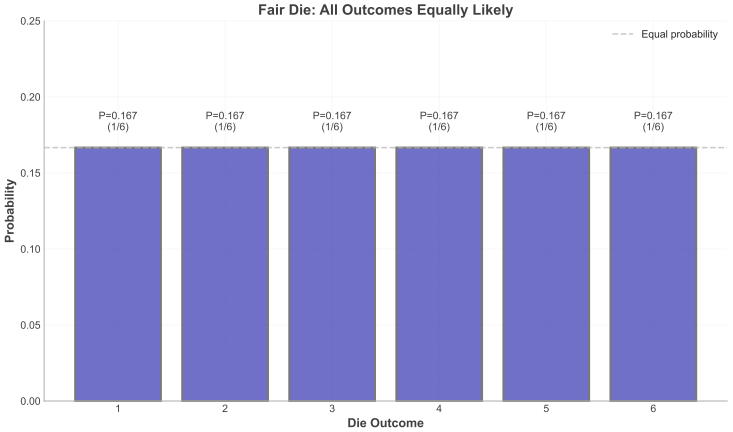
Predicting Words

- Thousands of possible words
- Each has *different* probability
- **Memory matters**: Previous words help
- Not equally likely: "the" more common than "xylophone"

Question: Can we predict the next word?

Both are probability problems - but text prediction is harder

Dice Probability: The Foundation



Key Insight: When all outcomes equally likely, $P(\textit{outcome}) = \frac{1}{\text{number of outcomes}}$

This is the simplest case - text is more complex

From Dice to Text Prediction

What We Know from Dice:

- Probability quantifies uncertainty
- Sum of all probabilities = 1
- More data → better estimates

New Challenges for Text:

- Outcomes NOT equally likely
- Context matters ("the cat" vs "the xylophone")
- How to estimate probabilities?

Our Goal:
Predict next word given previous words

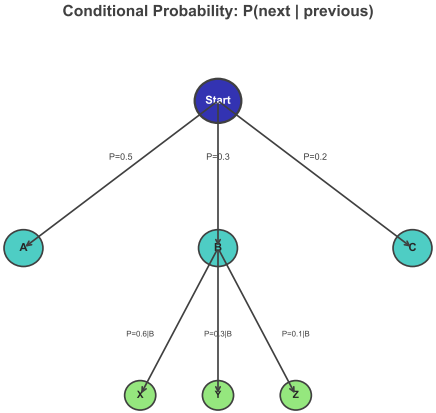
Example:

- Given: "The cat sat on the"
- Predict: "mat" (likely) or "xylophone" (unlikely)

How?
Use **conditional probability** based on what we've seen before

Next: Build conditional probability from first principles

Conditional Probability: A Simple Example



Key Insight: Knowing previous outcomes changes our predictions

Conditional Probability: The Mathematics

Definition:

$$P(A|B) = \frac{P(A \cap B)}{P(B)}$$

Reads: "Probability of A given B"

Worked Example with Cards:
Given: Drew a face card (J, Q, K)
Question: What's probability it's a King?

- Total face cards: 12 (4 Jacks, 4 Queens, 4 Kings)
- Kings among face cards: 4
- $P(\text{King}|\text{Face}) = \frac{4}{12} = \frac{1}{3}$

For Text Prediction:

$$P(w_n|w_1, w_2, ..., w_{n-1})$$

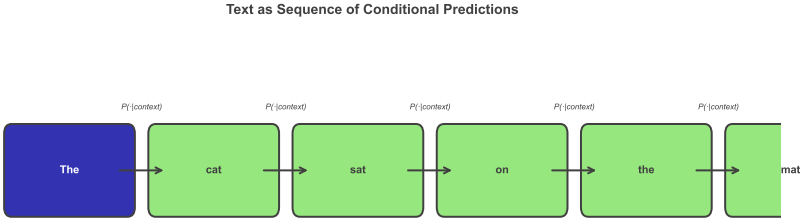
Reads: "Probability of word n given all previous words"

Example:
Given: "The cat sat on the"
Question: What's $P(\text{mat}|\text{the cat sat on the})$?

Answer: Count how many times we've seen this pattern!

Conditional probability is the foundation of language modeling

Text as a Sequence of Decisions



Key Insight: Each word is a prediction given all previous words

But how do we get these probabilities?

Word Prediction: The Core Problem

The Chain Rule:
For a sequence $w_1, w_2, \ldots, w_n$:

$$\begin{aligned} P(w_1, w_2, \ldots, w_n) = & P(w_1) \\ & \times P(w_2|w_1) \\ & \times P(w_3|w_1, w_2) \\ & \times \ldots \\ & \times P(w_n|w_1, \ldots, w_{n-1}) \end{aligned}$$

Challenge: How to estimate $P(w_n|w_1, \ldots, w_{n-1})$?

Example: "The cat sat":

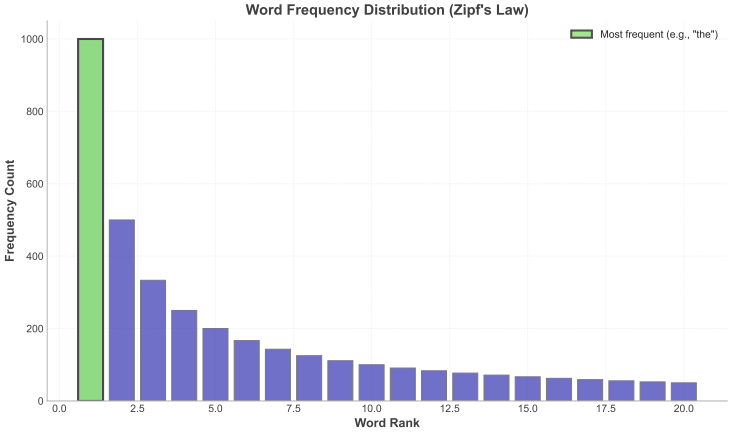
$$\begin{aligned} P(\text{the, cat, sat}) = & P(\text{the}) \\ & \times P(\text{cat}|\text{the}) \\ & \times P(\text{sat}|\text{the, cat}) \end{aligned}$$

Problem:
Infinitely many possible histories!

Solution:
Make a simplifying assumption (coming next)

We need a practical way to estimate these probabilities

The Solution: Count What We've Seen



Key Insight: Probability $\approx$ Frequency in large corpus

This is Maximum Likelihood Estimation

Maximum Likelihood Estimation (MLE)

The Idea:
Estimate probabilities from observed counts

Formula:

$$P(w|context) = \frac{\text{count}(context, w)}{\text{count}(context)}$$

Example from Shakespeare:

- count("to be") = 150 times
- count("to be or") = 42 times
- $P(\text{or}|\text{to be}) = \frac{42}{150} = 0.28$

Why This Works:

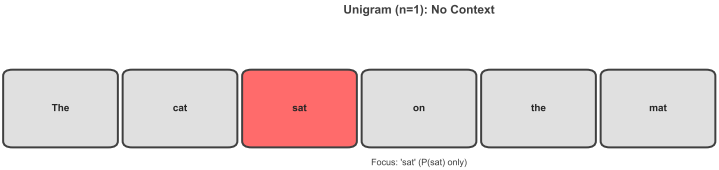
- Law of Large Numbers
- As corpus size $\to \infty$, relative frequency $\to$ true probability
- Real corpora: millions/billions of words

When to Use:

- Have large text corpus
- Want simple, interpretable model
- Need fast training and inference

MLE is simple but effective - used in production systems

N-gram Models: The Markov Assumption



Key Insight: Only last $n-1$ words matter for prediction

N-gram Models: Making It Practical

The Markov Assumption:

$$P(w_i|w_1, \ldots, w_{i-1}) \approx P(w_i|w_{i-n+1}, \ldots, w_{i-1})$$

Only last $n-1$ words matter

Different N-gram Models:

- **Unigram** ($n = 1$): $P(w_i)$
- **Bigram** ($n = 2$): $P(w_i|w_{i-1})$
- **Trigram** ($n = 3$): $P(w_i|w_{i-2}, w_{i-1})$

Why This Helps:

- Reduces number of parameters
- More data per pattern
- Tractable computation

Trade-off:

- Small n: Fast, robust, but misses long-range dependencies
- Large n: Captures context, but sparse data

Typical Choice: $n = 3$ (trigram) balances both

N-grams are the workhorse of statistical language modeling

Worked Example: Bigram Probabilities

Given Shakespeare corpus extract:
"To be or not to be that is the question whether"

Task: Compute $P(\text{be}|\text{to})$

Step 1: Count all bigrams starting with "to"

Bigram	Count
(to, be)	2

Step 2: Count total occurrences of "to"
$\text{count}(\text{to}) = 2$

Step 3: Apply MLE formula

$$P(\text{be}|\text{to}) = \frac{\text{count}(\text{to, be})}{\text{count}(\text{to})} = \frac{2}{2} = 1.0$$

Interpretation: In this tiny corpus, "to" is always followed by "be"!

Real corpora give more nuanced probabilities

Checkpoint: Test Your Understanding

Quick Quiz

Question 1:
Given corpus: "the cat sat on the mat the dog"
What is $P(\text{cat}|\text{the})$?

A) 1/3
B) 1/2
C) 2/3
D) 1/8

Question 2:
Why do we use the Markov assumption?

A) It's always correct
B) Reduces parameters
C) Improves accuracy
D) Runs faster

Answer 1: **A) 1/3**

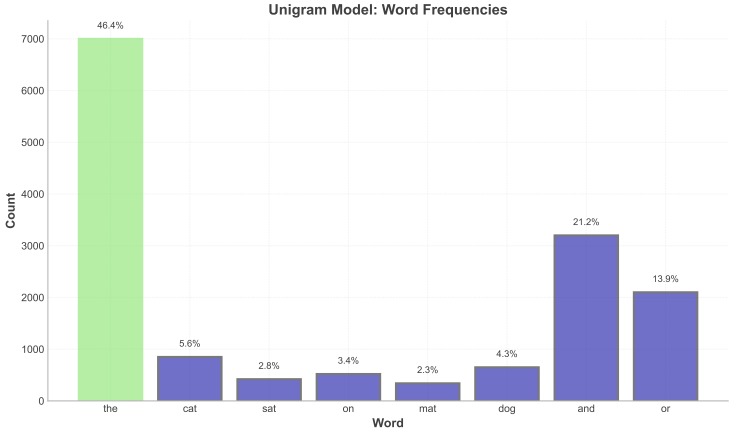
- $\text{count}(\text{the}) = 3$
- $\text{count}(\text{the, cat}) = 1$
- $P(\text{cat}|\text{the}) = 1/3$

Answer 2: **B) Reduces parameters**

- Without it: Infinite histories
- With it: Tractable parameter count
- Trade-off: Lose long-range info

Understanding these foundations is critical for what comes next

Unigram Model: The Simplest Baseline



Key Insight: Each word has fixed probability regardless of context

Simple but ignores all context - like random word sampling

Unigram Model: When Context Doesn't Matter

Formula:

$$P(w_1, w_2, \ldots, w_n) = \prod_{i=1}^{n} P(w_i)$$

Each word independent

MLE Estimation:

$$P(w) = \frac{\text{count}(w)}{\text{total words}}$$

Example:

- Total words: 1,000,000
- count("the") = 70,000
- $P(\text{the}) = 0.07$

When to Use:

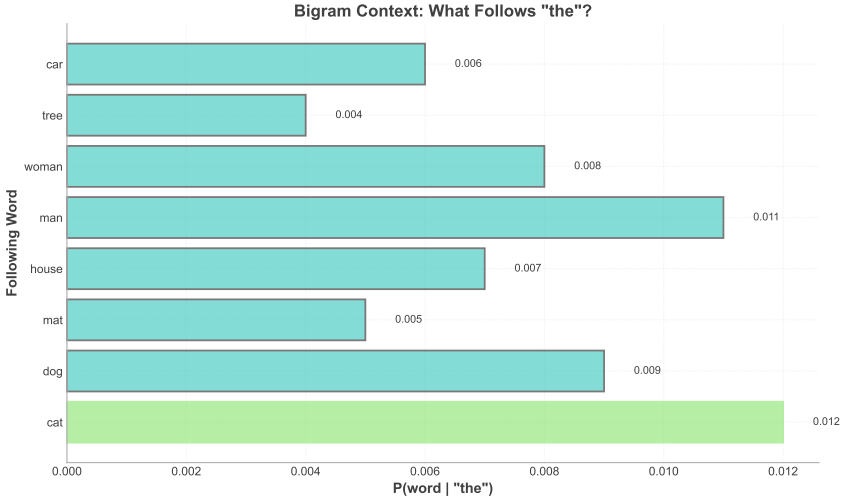
- Baseline comparison
- Document classification
- Bag-of-words features
- When word order truly doesn't matter

Limitations:

- Generates nonsense: "the the the cat dog"
- No grammar
- No meaning
- High perplexity

Unigrams are weak for generation but useful for other NLP tasks

Bigram Model: One Word of Memory



Key Insight: Previous word dramatically narrows possibilities

"the" is often followed by nouns, rarely by verbs

Bigram Model: The Sweet Spot

Formula:

$$P(w_i|w_{i-1})$$

Condition on previous word only

MLE Estimation:

$$P(w_2|w_1) = \frac{\text{count}(w_1, w_2)}{\text{count}(w_1)}$$

Worked Example:

- count("the") = 70,000
- count("the cat") = 850
- $P(\text{cat}|\text{the}) = \frac{850}{70000} = 0.012$

When to Use:

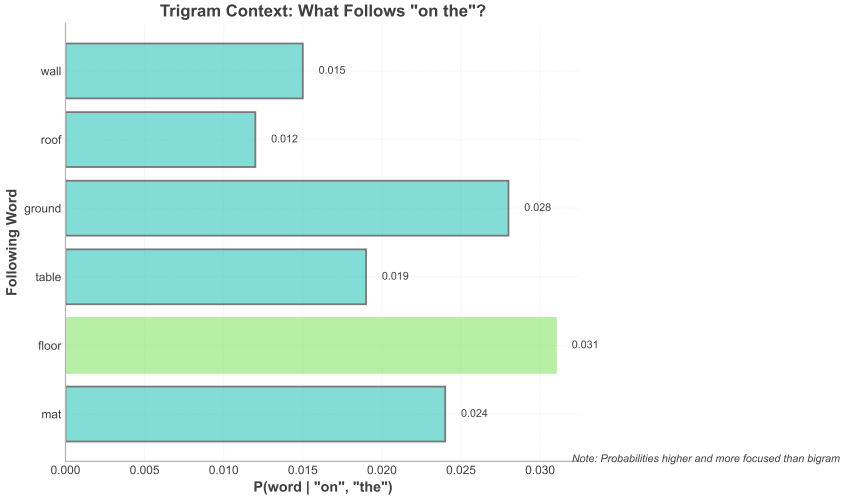
- Real-time applications (autocomplete)
- Limited memory/computation
- Reasonable text quality needed
- Standard baseline for comparison

Performance:

- Captures local grammar
- Generates coherent phrases
- Moderate perplexity (100-150)
- Still misses long-range dependencies

Bigrams are widely used - good balance of simplicity and performance

Trigram Model: Two Words of Memory



Key Insight: Two-word context captures richer patterns

"on the" strongly suggests location noun: mat, floor, table

Trigram Model: More Context, Better Predictions

Formula:

$$P(w_i|w_{i-2}, w_{i-1})$$

Condition on two previous words

MLE Estimation:

$$P(w_3|w_1, w_2) = \frac{\text{count}(w_1, w_2, w_3)}{\text{count}(w_1, w_2)}$$

Example:

- count("on the") = 5,200
- count("on the mat") = 127
- $P(\text{mat}|\text{on the}) = \frac{127}{5200} = 0.024$

Comparison to Bigram:

Model	Perplexity	Quality
Bigram	125	Good
Trigram	78	Better

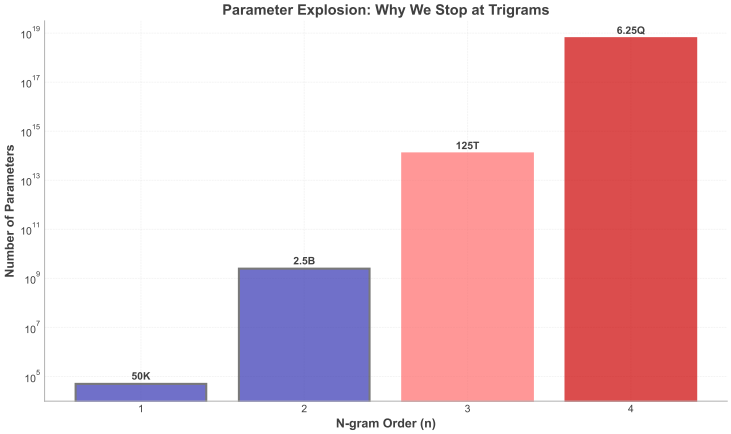
When to Use:

- Sufficient training data
- Quality matters more than speed
- Speech recognition, translation

Limitation: Data sparsity increases

Trigrams are standard in production systems when data permits

Higher-Order N-grams: The Sparsity Wall



Key Insight: Parameters explode exponentially with n

This is why we rarely go beyond trigrams

The Fundamental Tradeoff

Parameter Count:
For vocabulary size V:

- Unigram: V (e.g., 50,000)
- Bigram: V^2 (2.5 billion)
- Trigram: V^3 (125 trillion!)
- 4-gram: V^4 (6,250,000 trillion!)

Reality: Most combinations never seen

The Dilemma:

- More context $\rightarrow$ Better predictions
- More context $\rightarrow$ More parameters
- More parameters $\rightarrow$ Sparse data
- Sparse data $\rightarrow$ Poor estimates

Practical Choice:

- $n = 2$ (bigram): Fast, robust
- $n = 3$ (trigram): Standard
- $n \geq 4$: Rarely used

This limitation motivates smoothing (coming next) and neural models (Week 2)

The Sparsity Problem: Quantified

Experiment: Train on 1 million words, test on held-out data

Model	Seen in Training	Unseen in Test	OOV Rate
Unigram	45,000 words	2,300 words	5%
Bigram	523,000 pairs	87,000 pairs	14%
Trigram	891,000 triples	203,000 triples	19%
4-gram	958,000 quads	347,000 quads	27%

Pattern: As n increases, more unseen combinations

Problem: MLE assigns $P = 0$ to unseen n-grams

Consequence: If any n-gram has $P = 0$, entire sentence gets $P = 0$!

We need a way to handle unseen n-grams without destroying sentence probabilities

Why Zero Probability is Catastrophic

Example Sentence: "The cat sat on the xylophone"

Using Bigram Model:

$$P(\text{sentence}) = P(\text{the}) \times P(\text{cat}|\text{the}) \times P(\text{sat}|\text{cat}) \\ \times P(\text{on}|\text{sat}) \times P(\text{the}|\text{on}) \times P(\text{xylophone}|\text{the})$$

Problem:
If "the xylophone" never appeared in training:

$$P(\text{xylophone}|\text{the}) = \frac{0}{70000} = 0$$

Therefore: $P(\text{entire sentence}) = 0$

Consequence:

- Can't rank this sentence
- Can't generate it
- Perplexity becomes infinite!

A single unseen n-gram destroys everything - we must fix this

Root Cause: The Sparse Data Curse

What We Have:

- Training corpus: 1M words
- Vocabulary: 50K words
- Possible bigrams: $50K^2 = 2.5$ billion
- Observed bigrams: 500K

Coverage: $\frac{500K}{2.5B} = 0.02\%$

We've seen only 0.02% of possible bigrams!

The Mathematics:

- No smoothing: $P = 0$ for 99.98% of bigrams
- Sentence of length 10: $P(\text{sentence}) = 0$ in 95% of cases
- Useless for real applications

Root Cause:
MLE assumes if we haven't seen it, it's impossible

Reality:
Unseen $\neq$ Impossible

We need to reserve some probability mass for unseen events

The Solution: Smoothing

∞

Laplace Smoothing: The Simplest Fix

The Idea:
Pretend we've seen every n-gram one extra time

Formula (Bigram):

$$P_{smooth}(w_2|w_1) = \frac{\text{count}(w_1, w_2) + 1}{\text{count}(w_1) + V}$$

where V = vocabulary size

Worked Example:

- count("the") = 70,000
- count("the cat") = 850
- V = 50,000

$$P(\text{cat}|\text{the}) = \frac{850 + 1}{70000 + 50000} = \frac{851}{120000} = 0.0071$$

For Unseen N-gram:
count("the xylophone") = 0

$$P(\text{xylophone}|\text{the}) = \frac{0 + 1}{70000 + 50000} = \frac{1}{120000} = 8.3 \times 10^{-6}$$

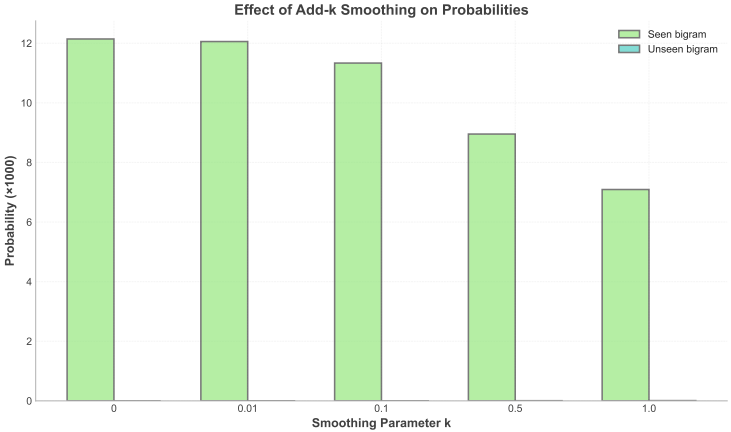
Small but non-zero!

Properties:

- Simple to implement
- Never gives $P = 0$
- Too much mass to rare events
- Hurts frequent events

Laplace smoothing is too aggressive for language - we need something better

Add-k Smoothing: A Better Balance



Key Insight: Add $k < 1$ instead of 1 to balance probability redistribution

Typical values: $k = 0.01$ **to** $k = 0.5$

Advanced Smoothing: Kneser-Ney

The Insight:
Not all words are equally likely in new contexts

Example:

- "Francisco" appears often
- But only after "San"
- Shouldn't get high probability after other words!

Solution:
Count *how many different contexts* a word appears in, not just total frequency

Kneser-Ney Benefits:

- State-of-the-art for n-grams
- 15-20% perplexity improvement
- Used in production systems

Complexity:

- More sophisticated than add-k
- Requires continuation counts
- Backoff to shorter n-grams

When to Use:
When quality matters most

Full Kneser-Ney derivation beyond BSc scope - but know it exists

Validation: Smoothing Improves Performance

Experimental Setup:
Train on 1M words, test on 100K held-out words, trigram model

Smoothing Method	Perplexity	Unseen N-gram Handling
None (MLE)	∞	Fails
Add-1 (Laplace)	245	Poor
Add-0.1	156	Good
Add-0.01	132	Good
Kneser-Ney	98	Best

Pattern:

- Smaller k better for large corpora
- Kneser-Ney best overall
- 35% improvement over add-1

Takeaway: Smoothing is not optional - it's essential

Modern systems use sophisticated smoothing as standard

Implementation: Add-k Smoothing

```python
class SmoothBigram:
    def __init__(self, k=0.01):
        self.k = k
        self.counts = defaultdict(
            lambda: defaultdict(int)
        )
        self.vocab = set()

    def train(self, text):
        words = text.split()
        for i in range(len(words)-1):
            w1, w2 = words[i], words[i+1]
            self.counts[w1][w2] += 1
            self.vocab.update([w1, w2])

    def probability(self, w1, w2):
        V = len(self.vocab)
        numerator = self.counts[w1][w2] + self.k
        denominator = sum(
            self.counts[w1].values()
        ) + self.k * V
        return numerator / denominator
```

Key Changes from MLE:

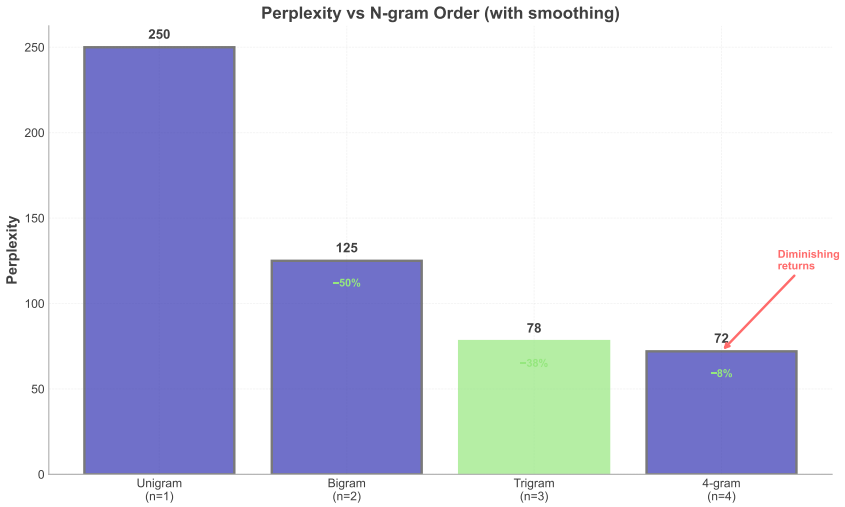
- Add k to numerator
- Add $k \times V$ to denominator
- Never returns 0!

Usage:

```python
model = SmoothBigram(k=0.01)
model.train(corpus)
p = model.probability(
    "the", "xylophone"
)
# Returns small non-zero value
```

Just 4 lines different from MLE - huge impact on performance

Evaluation: How Good is Our Model?



Key Insight: Lower perplexity = better predictions

Perplexity is the standard metric for language models

Perplexity: Measuring Uncertainty

Formula:

$$PP(W) = P(w_1, w_2, ..., w_N)^{-\frac{1}{N}}$$

or equivalently:

$$PP(W) = \sqrt[N]{\frac{1}{P(w_1, ..., w_N)}}$$

Logarithmic Form:

$$\log_2 PP(W) = -\frac{1}{N} \sum_{i=1}^{N} \log_2 P(w_i|context)$$

Interpretation:
"On average, how many equally likely words could come next?"

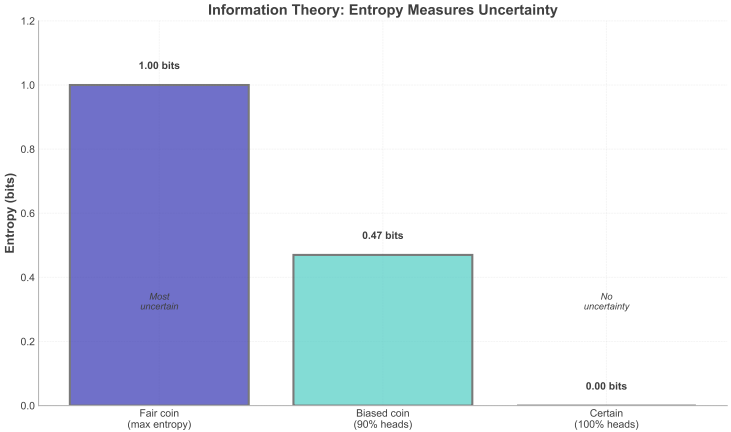
- PP = 100: Choosing from 100 words
- PP = 50: Model is twice as confident
- PP = 10: Very good model
- PP = 1000: Poor model

Properties:

- Lower is better
- Minimized by true distribution
- Inverse of geometric mean probability

Perplexity connects probability to human intuition about uncertainty

Information Theory: The Foundations



Key Insight: Rare events carry more information than common events

Claude Shannon founded information theory in 1948

Shannon's Information Theory

Entropy (Uncertainty):

$$H(X) = -\sum_{x} P(x) \log_2 P(x)$$

Measures average information per symbol

Cross-Entropy:

$$H(P, Q) = -\sum_{x} P(x) \log_2 Q(x)$$

Measures information loss when using Q to approximate P

Relationship to Perplexity:

$$PP = 2^{H(P,Q)}$$

Example - Coin Flip:
Fair coin: $P(\text{heads}) = 0.5$

$$H = -0.5 \log_2(0.5) - 0.5 \log_2(0.5) = 1 \text{ bit}$$

Example - English Text:
Entropy $\approx$ 1-2 bits per character

Why This Matters:

- Fundamental limits on compression
- Optimal encoding
- Neural models minimize cross-entropy

Information theory is the mathematical foundation of all NLP

Worked Example: Computing Perplexity

Given: Bigram model, test sentence: "the cat sat"

Model Probabilities:

- $P(\text{the}) = 0.07$
- $P(\text{cat}|\text{the}) = 0.012$
- $P(\text{sat}|\text{cat}) = 0.08$

Step 1: Compute sentence probability

$$\begin{aligned} P(\text{the, cat, sat}) &= P(\text{the}) \times P(\text{cat}|\text{the}) \times P(\text{sat}|\text{cat}) \\ &= 0.07 \times 0.012 \times 0.08 \\ &= 0.0000672 \end{aligned}$$

Step 2: Apply perplexity formula

$$PP = (0.0000672)^{-\frac{1}{3}} = (0.0000672)^{-0.333} = 126.7$$

Interpretation: Model is as uncertain as picking from 127 equally likely words

Lower perplexity means higher confidence in predictions

When NOT to Use N-grams

Failure Cases:

- **Long-range dependencies**:
  "The author, who wrote several books about..., was awarded a prize."
  ("author" $\rightarrow$ "was", but 10+ words apart)
- **Semantic understanding**:
  "The bank is closed" vs "The river bank"
  (Same n-grams, different meanings)
- **Rare but valid constructions**:
  Creative language, poetry, technical jargon

Better Alternatives:

- **Neural language models** (Week 2):
  Learn distributed representations
- **RNN/LSTM** (Week 3):
  Unbounded context window
- **Transformers** (Week 5):
  Direct long-range connections

When to Stick with N-grams:

- Speed critical
- Interpretability needed
- Limited data
- Baseline comparison

Know your model's limitations - choose the right tool for the job

Common Pitfalls and How to Avoid Them

Pitfall 1: Choosing n

- Too small: Misses context
- Too large: Data sparsity
- Solution: Start with trigrams, validate on held-out data

Pitfall 2: Forgetting smoothing

- MLE gives zero probabilities
- Solution: Always use smoothing in production

Pitfall 3: Train/test contamination

- Testing on training data
- Solution: Strict data splits

Pitfall 4: Vocabulary mismatch

- Test words not in training vocab
- Solution: UNK token for rare words

Pitfall 5: Computational explosion

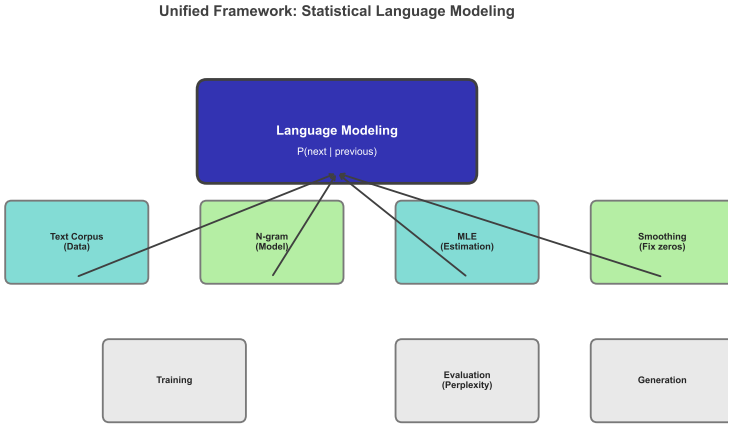
- Storing all n-grams
- Solution: Pruning, hashing tricks

Pitfall 6: Overfitting to corpus

- Model learns corpus-specific patterns
- Solution: Large, diverse training data

Awareness of pitfalls is half the battle - test rigorously

Unified View: Everything is Conditional Probability



Key Takeaways

1. **Conditional probability is the foundation**
   Text prediction is estimating $P(w_n|w_1, ..., w_{n-1})$
2. **N-grams make it practical**
   Markov assumption limits context to last $n-1$ words
3. **MLE estimates from counts**
   $P(w|context) = \frac{\text{count}(context, w)}{\text{count}(context)}$
4. **Smoothing is essential**
   Unseen n-grams get non-zero probability
5. **Perplexity measures quality**
   Lower perplexity = better predictions
6. **Trade-offs everywhere**
   Context vs sparsity, speed vs quality, simplicity vs sophistication

---

Master these foundations - they underpin all of NLP

Next: Hands-On Practice

Lab Notebook Activities:

1. Build unigram model from scratch
2. Implement bigram with MLE
3. Add smoothing (compare methods)
4. Generate text and compare quality
5. Compute perplexity on test data
6. Visualize n-gram distributions
7. Experiment with different n values

What You'll Learn:

- Hands-on probability estimation
- See smoothing's impact
- Debug zero probability issues
- Understand perplexity intuitively
- Reproduce presentation charts

Corpus: Shakespeare sonnets (interesting patterns!)

Let's build some language models!

The lab cements understanding - theory alone is not enough

Word Embeddings

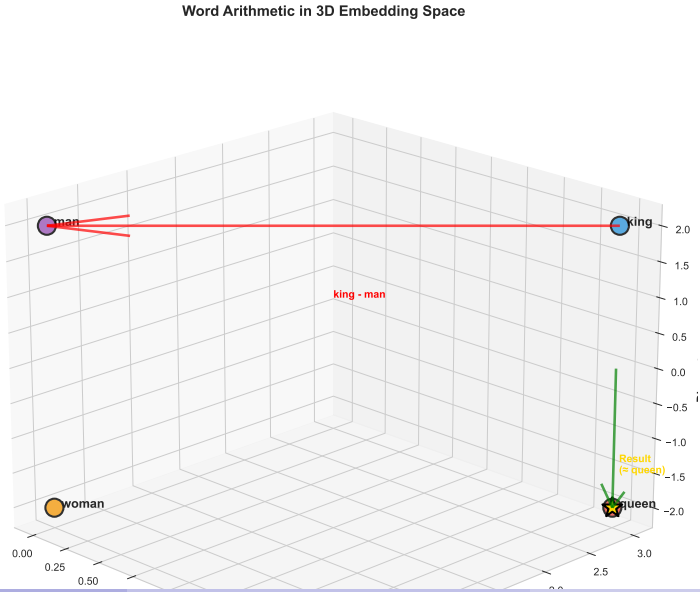
Week 2 - When Words Became Vectors

NLP Course 2025

October 27, 2025

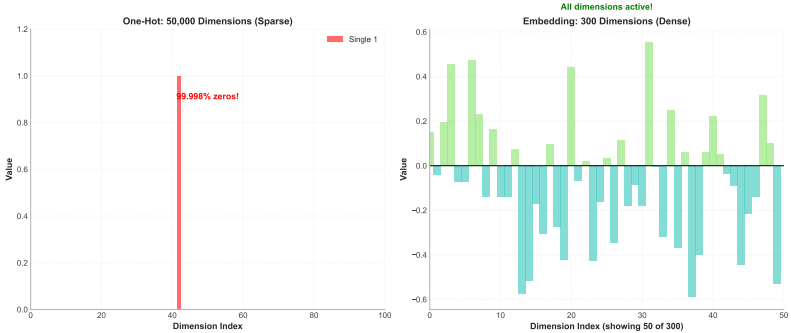
Two-Tier BSc Discovery Presentation

Hook #1: Words That Do Algebra



Hook #2: Similarity That N-grams Miss

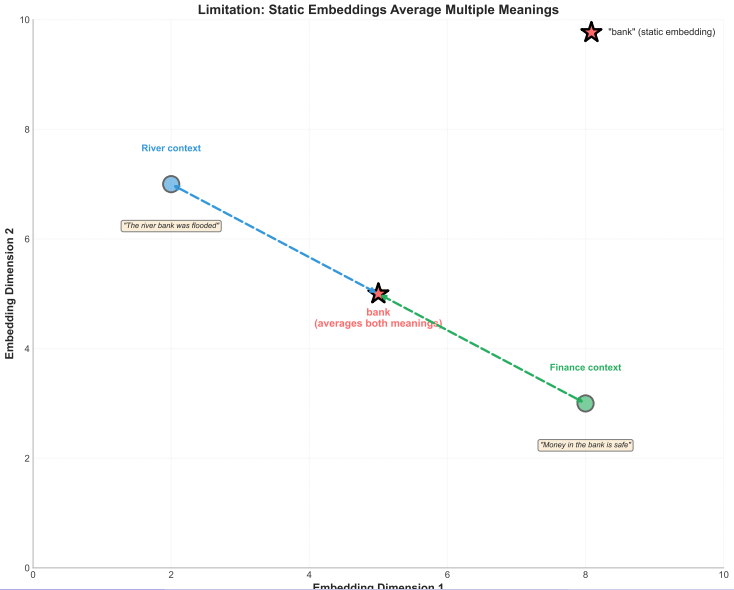
Hook #3: Compression That Improves Quality



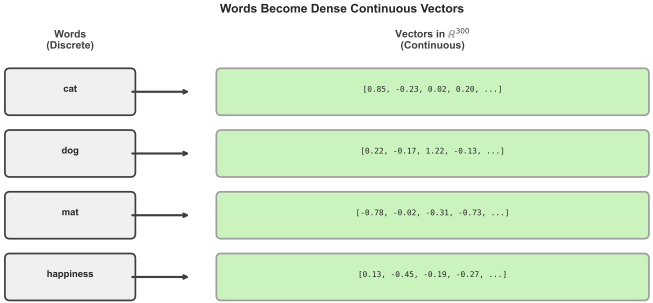
Key Insight: 50,000 sparse → 300 dense with MORE information

Dense representations are more powerful than sparse ones

Hook #4: Words With Multiple Meanings



What Are Word Embeddings?



Definition: Dense, low-dimensional, continuous vector representations of words

Words become points in semantic space

From Sparse One-Hot to Dense Embeddings

One-Hot Encoding (Old Way):
Each word = vector of size $|V|$

Example ($V = 5$):

- "cat" = [1, 0, 0, 0, 0]
- "dog" = [0, 1, 0, 0, 0]
- "mat" = [0, 0, 1, 0, 0]

Problems:

- Huge dimensionality (50K typical)
- All words equally distant
- No similarity information
- Sparse (99.998% zeros)

Dense Embeddings (New Way):
Each word = vector of size d (300 typical)

Example ($d = 3$):

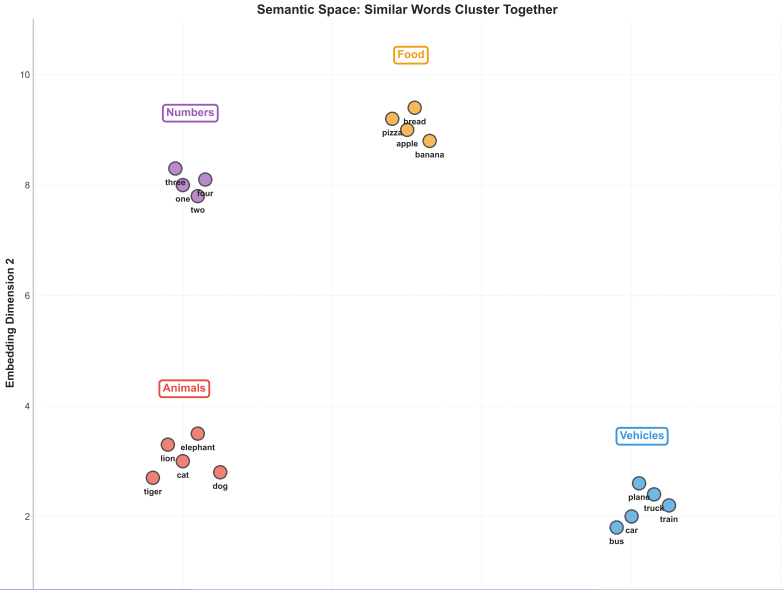
- "cat" = [0.2, 0.8, -0.3]
- "dog" = [0.1, 0.7, -0.2]
- "mat" = [-0.5, 0.1, 0.6]

Advantages:

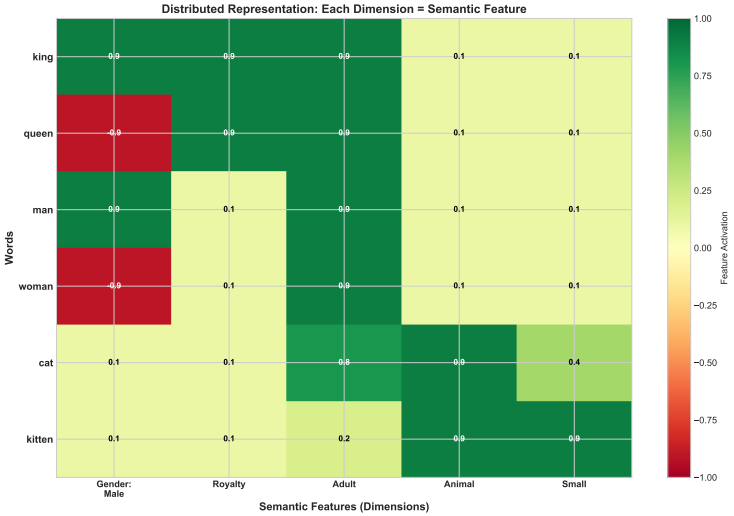
- Low dimensionality (100-300)
- Similarity encoded (cat $\approx$ dog)
- Continuous values
- Information-dense

Dense $<$ Sparse but contains MORE information - this is the magic

The Vector Space Model



Why Distributed Representations Work



Key Insight: Each dimension captures some semantic feature

Word2Vec: The Core Idea

"You shall know a word by the company it keeps"

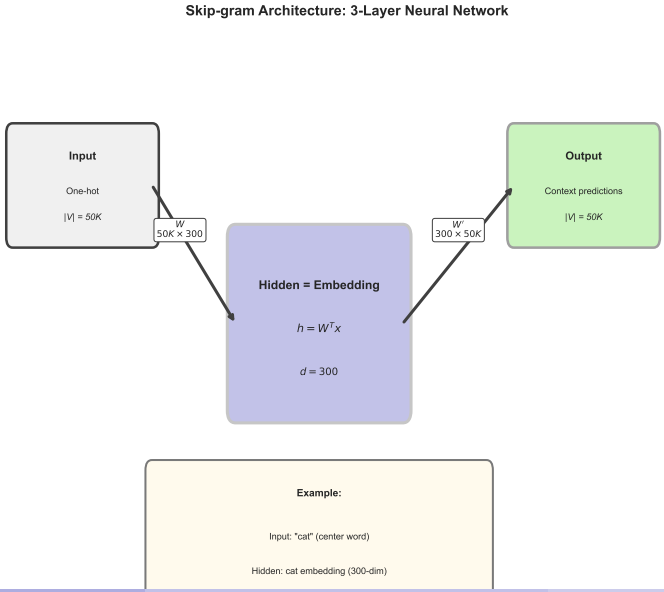
- J.R. Firth (1957)

Distributional Hypothesis:
Words appearing in similar contexts have similar meanings

Word2Vec Approach:
Learn word vectors by predicting context

Context prediction forces similar words to have similar vectors

Skip-gram: Predict Context from Word



Skip-gram: The Architecture

Input: Center word (one-hot)
$x \in \mathbb{R}^{|V|}$

Hidden Layer: Embedding lookup
$h = W^T x \in \mathbb{R}^d$
This IS the word embedding!

Output Layer: Context predictions
$y = W' h \in \mathbb{R}^{|V|}$
Softmax for probabilities

Parameters:

- W: $|V| \times d$ (input embeddings)
- W': $d \times |V|$ (output weights)

Training Objective:
Maximize probability of context words

$$\max \sum_{t=1}^{T} \sum_{-c \leq j \leq c, j \neq 0} \log P(w_{t+j} | w_t)$$

Example:
Sentence: "The cat sat on the mat"
Center: "cat"
Context window $c = 2$:
Predict: "the", "sat"

Key Trick:
Share embeddings! W is what we keep after training

Simple 3-layer network - embeddings are the weights

Worked Example: Skip-gram Forward Pass

Given: "The cat sat", center = "cat", predict "sat"

Step 1: One-hot encode center word
"cat" = word ID 3797
$x = [0, 0, \ldots, 1_{3797}, \ldots, 0] \in \mathbb{R}^{50000}$

Step 2: Embedding lookup (hidden layer)

$$h = W^T x = W_{3797} \in \mathbb{R}^{300}$$

This is just row 3797 of W! (Lookup, no multiplication needed)
Example: $h = [0.23, -0.41, 0.15, \ldots, 0.08]$

Step 3: Compute output scores

$$score(\text{“}sat''\text{)} = W'_{sat} \cdot h = 0.85$$

Step 4: Softmax over vocabulary

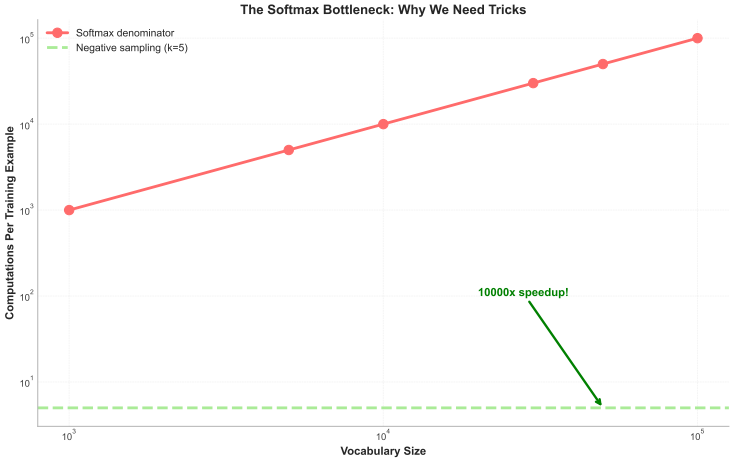
$$P(\text{“}sat''|\text{“}cat''\text{)} = \frac{\exp(0.85)}{\sum_{w \in V} \exp(score(w))} = 0.0023$$

Step 5: Compute loss

$$Loss = -\log(0.0023) = 6.07$$

Backprop updates W to increase P(sat—cat) - embeddings improve

The Computational Bottleneck



Key Insight: Computing softmax over 50K words is prohibitively expensive

Softmax denominator requires summing over entire vocabulary

Negative Sampling: The Trick That Makes It Practical

The Problem: Softmax over 50K words per training example

$$P(context|word) = \frac{\exp(score)}{\sum_{w=1}^{50000} \exp(score_w)}$$

Requires 50K exponentials per example!

The Solution: Negative Sampling

- 1 positive pair: ("cat", "sat") - actual context
- k negative pairs: ("cat", "xylophone"), ("cat", "democracy"), ... - random words
- Typical: $k = 5$ for small datasets, $k = 2 - 5$ for large

New Objective:

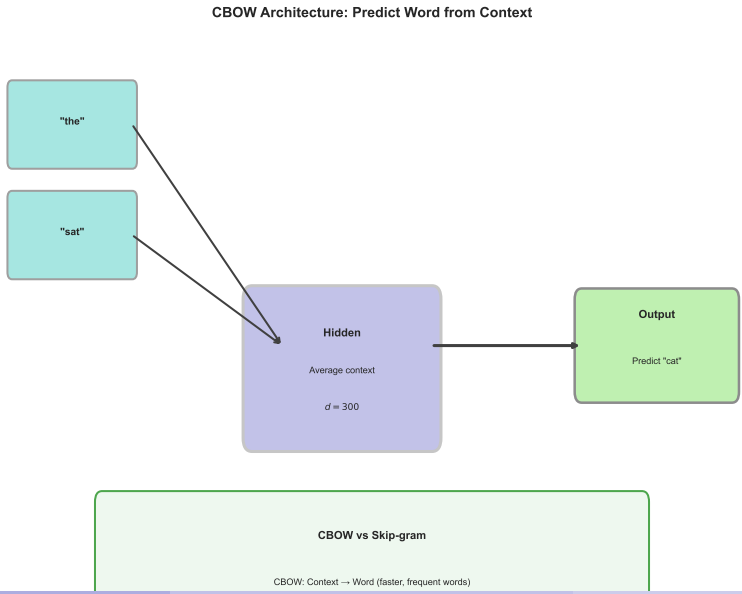
$$\log \sigma(v'_{sat} \cdot v_{cat}) + \sum_{i=1}^{k} \log \sigma(-v'_{w_i} \cdot v_{cat})$$

Only $k + 1$ computations instead of 50,000!

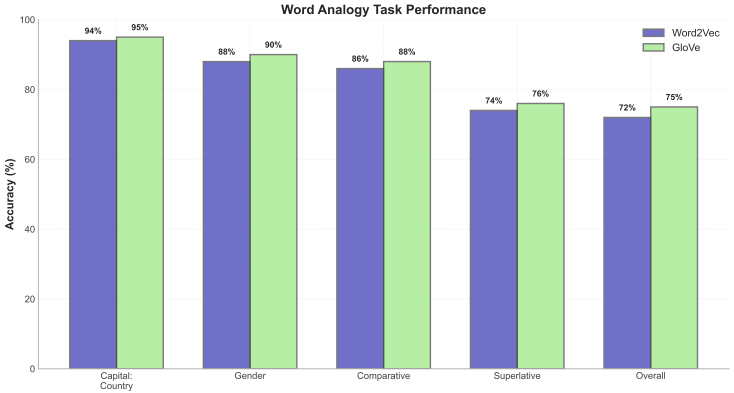
Example: Positive pair + 5 negatives = 6 computations vs 50,000
Speedup: **8,333x faster!**

Negative sampling approximates softmax - critical for practical training

CBOW: The Reverse Approach



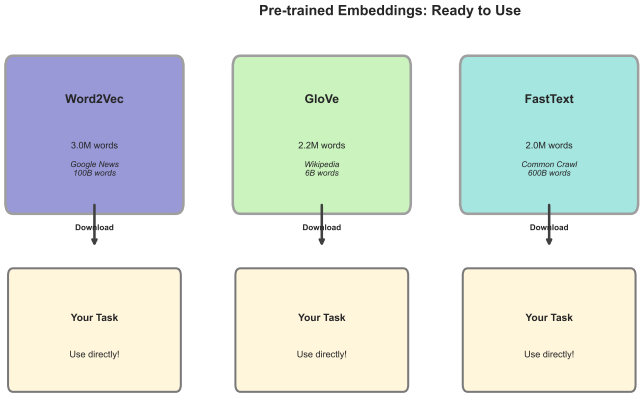
Evaluation: Word Analogies



Key Insight: Good embeddings solve analogies via vector arithmetic

king:queen :: man:woman achieved 72% accuracy (Word2Vec, 2013)

Pre-trained Embeddings



All three available for free - choose based on your corpus/task

Key Insight: Use pre-trained embeddings as starting point

Word2Vec, GloVe, FastText - all available pre-trained

Key Takeaways

1. **Embeddings as dense vectors**
   Words $\rightarrow$ continuous vectors in $\mathbb{R}^d$ (typically $d = 300$)
2. **Skip-gram predicts context from word**
   Train by predicting surrounding words in large corpus
3. **Negative sampling enables efficient training**
   Approximate softmax with k negative samples (8000x speedup)
4. **Geometric semantics**
   Similarity = cosine, analogies = vector arithmetic
5. **Foundation for neural NLP**
   All neural models start with embedding layer

Embeddings revolutionized NLP in 2013 - still used everywhere today

Next: Visual Exploration

Lab Activities:

- Load pre-trained Word2Vec
- Visualize in 2D and 3D
- Perform word arithmetic
- Find most similar words
- Explore semantic clusters
- Compare Word2Vec vs GloVe
- Visualize training evolution

Visualizations You'll Create:

- 2D PCA projections
- Interactive 3D scatter plots
- Analogy arrows in 2D
- Similarity heatmaps
- Semantic cluster plots
- Training convergence

Goal: Build intuition through visualization

See embeddings come alive!

Understanding embeddings requires seeing them - lab is essential

Technical Appendix

Complete Mathematical Treatment

Word embeddings capture semantic relationships between words.

Appendix A1: Skip-gram Objective Function

Goal: Maximize probability of context words given center word

Objective:

$$\mathcal{L} = \frac{1}{T}\sum_{t=1}^{T}\sum_{\substack{-c\leq j\leq c\\ j\neq 0}} \log P(w_{t+j}|w_t)$$

where:

- T: Total words in corpus
- c: Context window size (typically 5)
- w_t: Center word at position t
- w_{t+j}: Context word at offset j

Conditional Probability (Naive Softmax):

$$P(w_O|w_I) = \frac{\exp(v'_{w_O} \cdot v_{w_I})}{\sum_{w=1}^{|V|}\exp(v'_w \cdot v_{w_I})}$$

where v_{w_I} is input embedding, v'_{w_O} is output embedding

Problem: Denominator sums over entire vocabulary - $O(|V|)$ per example
For $|V| = 50K$, $T = 1B$ words, $c = 5$: 500 trillion softmax computations!

This objective is correct but computationally infeasible

Appendix A2: Negative Sampling Mathematics

Key Idea: Binary classification instead of multi-class

Reformulation:
Instead of predicting which word from vocabulary,
Predict: Is this word in context (yes/no)?

Negative Sampling Objective:

$$\log \sigma(v'_{w_O} \cdot v_{w_I}) + \sum_{i=1}^{k} \mathbb{E}_{w_i \sim P_n(w)}[\log \sigma(-v'_{w_i} \cdot v_{w_I})]$$

where:

- $\sigma(x) = \frac{1}{1+\exp(-x)}$ (sigmoid)
- w_O: Actual context word (positive example)
- w_i: Sampled negative words
- $P_n(w)$: Noise distribution (typically $P(w)^{0.75}$)
- k: Number of negative samples (5-20)

Why $P(w)^{0.75}$?
Raises probability of rare words, lowers frequent words
More balanced negative sampling

This approximates softmax with k samples instead of $|V|$ computations

Appendix A3: Hierarchical Softmax Alternative

Different Approach: Binary tree instead of flat softmax

Key Idea:

- Arrange vocabulary in binary tree
- Prediction = path through tree
- Each node: Binary decision (left vs right)
- Depth: $\log_2 |V|$ decisions instead of $|V|$ computations

Complexity:

- Naive softmax: $O(|V|)$ per example
- Hierarchical softmax: $O(\log |V|)$ per example
- For $|V| = 50K$: $O(50000)$ vs $O(16)$ - 3000x speedup!

Trade-offs:

- Faster than naive softmax
- Slower than negative sampling
- Exact (no approximation)
- Tree construction matters

Usage: Less common than negative sampling

Hierarchical softmax elegant but negative sampling more practical

Appendix A4: Training via Gradient Descent

Optimization: Stochastic Gradient Descent (SGD)

Gradients for Skip-gram with Negative Sampling:
For positive pair (w_I, w_O):

$$\frac{\partial}{\partial v_{w_I}} = (1 - \sigma(v'_{w_O} \cdot v_{w_I}))v'_{w_O}$$

For negative pairs (w_I, w_i):

$$\frac{\partial}{\partial v_{w_I}} = -\sigma(-v'_{w_i} \cdot v_{w_I})v'_{w_i}$$

Update Rule:

$$v_{w_I}^{new} = v_{w_I}^{old} - \eta \cdot \frac{\partial L}{\partial v_{w_I}}$$

where η is learning rate (typically 0.025, decays to 0.0001)

Training Details:

- Mini-batch size: 100-1000 word pairs
- Epochs: 5-15 over corpus
- Learning rate decay: Linear
- Convergence: 1-3 days on CPU for 1B words

Appendix A5: Computational Complexity Analysis

Method	Per Example	Training Time	Quality
Naive Softmax	$O(\|V\| \cdot d)$	Weeks	Best
Hierarchical Softmax	$O(\log \|V\| \cdot d)$	Days	Good
Negative Sampling	$O(k \cdot d)$	Hours	Good

Typical: $|V| = 50K$, $d = 300$, $k = 5$, corpus=1B words

Memory Requirements:

- Embeddings: $|V| \times d \times 4$ bytes $= 50\text{K} \times 300 \times 4 = 60\text{MB}$
- Context matrix: Another 60MB
- Total: 120MB (fits in RAM easily)

Parallelization:

- Word2Vec easily parallelizable (independent windows)
- Multi-threading: Near-linear speedup
- GPU: 10-50x faster than CPU

Efficient algorithm + modern hardware = practical at scale

Appendix A6: GloVe - Global Vectors for Word Representation

Different Philosophy: Explicit matrix factorization

Key Idea:

- Word2Vec: Local context window (implicit matrix factorization)
- GloVe: Global co-occurrence statistics (explicit matrix factorization)

Co-occurrence Matrix X:

$$X_{ij} = \text{number of times word } j \text{ appears in context of word } i$$

Example snippet: Count how often "cat" and "dog" appear near each other across entire corpus

GloVe Objective:

$$\mathcal{L} = \sum_{i,j=1}^{|V|} f(X_{ij})(v_i^T v_j + b_i + b_j - \log X_{ij})^2$$

where $f(x)$ is weighting function (down-weight rare co-occurrences)

GloVe combines global statistics with local context

Appendix A7: GloVe Objective Function Breakdown

Goal: Dot product of vectors should match log co-occurrence

$$v_i^T v_j \approx \log X_{ij}$$

Weighted Least Squares:

$$\min \sum_{i,j=1}^{|V|} f(X_{ij})(v_i^T v_j + b_i + b_j - \log X_{ij})^2$$

Weighting Function:

$$f(x) = \begin{cases} (x/x_{max})^\alpha & \text{if } x < x_{max} \\ 1 & \text{otherwise} \end{cases}$$

Typical: $x_{max} = 100$, $\alpha = 0.75$

Why Weighting Matters:

- Very rare co-occurrences: Noisy, unreliable
- Very frequent: Dominate loss ("the the", "of the")
- Middle ground: Most informative

Weighting function is critical for GloVe performance

Appendix A8: Matrix Factorization Connection

Insight: Both Word2Vec and GloVe factorize co-occurrence matrix

Pointwise Mutual Information (PMI):

$$PMI(i,j) = \log \frac{P(i,j)}{P(i)P(j)} = \log \frac{X_{ij} \cdot |X|}{\sum_k X_{ik} \cdot \sum_k X_{kj}}$$

Measures how much more likely words co-occur than by chance

Connection:
Word2Vec (Skip-gram with negative sampling) implicitly factorizes shifted PMI matrix:

$$v_i^T v_j \approx PMI(i,j) - \log k$$

GloVe explicitly factorizes log co-occurrence matrix

Unifying View:
Both methods learn low-rank approximation of word-context statistics
Different loss functions, similar result

This explains why Word2Vec and GloVe produce similar embeddings

Appendix A9: GloVe Training Algorithm

Steps:

1. Build co-occurrence matrix X from corpus (count pairs within window)
2. Initialize word vectors v_i and biases b_i randomly
3. Optimize via AdaGrad:
   For each (i, j) pair with $X_{ij} > 0$:

$$v_i^{new} = v_i - \eta \frac{\partial L}{\partial v_i}$$

   where gradient includes $f(X_{ij})$ weighting
4. Iterate until convergence (50-100 epochs typical)
5. Final embeddings: v_i (can optionally average with v_j)

Hyperparameters:

- Embedding dimension d: 100-300
- Context window: 10-15 (larger than Word2Vec's 5)
- x_{max}: 100
- α: 0.75
- Learning rate: 0.05 with AdaGrad

Training Time: Similar to Word2Vec (hours to days)

GloVe implementation simpler than Word2Vec (no neural network)

Appendix A10: Word2Vec vs GloVe - When to Use Each

Aspect	Word2Vec	GloVe
Method	Local context window	Global co-occurrence
Objective	Predict context	Factorize matrix
Complexity	$O(k \cdot d)$ per pair	$O(nnz)$ total
Training	Online (streaming)	Batch (requires X)
Memory	Low (embeddings only)	High (needs matrix)
Quality	Excellent	Excellent
Speed	Fast	Moderate
Rare words	Better (Skip-gram)	Moderate
Analogies	72%	75%
Best for	Large corpora, streaming	Small/medium corpora

Empirical Results (on same corpus):

- Performance: Nearly identical (GloVe 3-5% better on some tasks)
- Training time: Word2Vec faster (no matrix construction)
- Implementation: Word2Vec simpler (fewer hyperparameters)

Recommendation: Start with Word2Vec, try GloVe if you have time

Both are excellent - choice matters less than proper training

Appendix A11: Implementation Details and Best Practices

Corpus Preparation:

- Lowercase everything (or preserve case)
- Remove rare words (¡ 5 occurrences)
- Subsampling frequent words: $P(w_i) = 1 - \sqrt{t/f(w_i)}$ where $t = 10^{-5}$
- Helps balance frequent/rare words

Hyperparameter Choices:

Parameter	Small Corpus	Large Corpus
Embedding dim d	100-200	300-500
Window size c	3-5	5-10
Negative samples k	5-10	2-5
Min word count	5	10
Learning rate η	0.025	0.025
Epochs	10-20	5-10

Debugging Tips:

- Check: Loss should decrease steadily
- Test: Run analogies after each epoch
- Validate: Hold out 10% for validation

These details matter - bad hyperparameters give poor embeddings

Appendix A12: FastText - Character N-grams

Motivation: Word2Vec/GloVe ignore word morphology

FastText Innovation (Facebook AI, 2017):
Represent words as bags of character n-grams

Example: "playing" = ¡pl, pla, lay, ayi, yin, ing, ng¿

Embedding:

$$v_{playing} = \sum_{g \in ngrams(\text{“}playing\text{”})} v_g$$

Sum of character n-gram embeddings

Advantages:

- Handle OOV words (unseen in training)
- Capture morphology ("play" in "playing", "player")
- Better for morphologically rich languages
- Small vocabulary (can't memorize all words)

Trade-offs:

- Handles rare/unseen words
- More parameters (n-grams)
- Morphological awareness

Appendix A13: ELMo - Deep Contextualized Embeddings

Limitation of Word2Vec/GloVe: One vector per word (no context)
"bank" always has same embedding (averages river and money meanings)

ELMo Solution (2018):

- Embeddings from Language Model (ELMo)
- BiLSTM reads sentence
- Each word gets different embedding depending on context

Example:

- "The bank of the river" $\rightarrow v_{bank}^{river}$
- "Money in the bank" $\rightarrow v_{bank}^{money}$
- $v_{bank}^{river} \neq v_{bank}^{money}$

Connection to Week 6:
ELMo $\rightarrow$ BERT $\rightarrow$ GPT progression
All use context to modify embeddings

Note: Static embeddings (Word2Vec/GloVe) still useful for many tasks

ELMo bridged static embeddings to contextual (BERT) - important milestone

Appendix A14: Evaluation Metrics

Intrinsic Evaluation (embedding quality directly):

- **Word Similarity**: Correlation with human judgments (WordSim-353, SimLex-999)
- **Word Analogies**: Accuracy on a:b :: c:d tasks (Google analogy dataset)
- **Clustering**: Do semantic categories cluster?

Extrinsic Evaluation (downstream task performance):

- Use embeddings in actual NLP task
- Sentiment classification accuracy
- Named entity recognition F1
- Question answering performance

Trade-offs:

- Intrinsic: Fast, but doesn't guarantee downstream success
- Extrinsic: Slow, but measures real usefulness

Best Practice: Use both - intrinsic for development, extrinsic for final validation

Good intrinsic scores usually (but not always) lead to good extrinsic performance

Appendix A15: From Word2Vec to Transformers

The Evolution (2013-2024):

Year	Model	Innovation
2013	Word2Vec	Static distributed representations
2014	GloVe	Matrix factorization perspective
2017	FastText	Subword embeddings
2018	ELMo	Contextualized (BiLSTM)
2018	BERT	Transformer encoder (Week 6)
2018	GPT	Transformer decoder (Week 6)
2024	GPT-4/Claude	1T+ parameters, multimodal

What Stayed from Word2Vec:

- Embedding layer (first layer of all neural models)
- Distributional hypothesis
- Pre-training on large corpora
- Vector arithmetic intuition

What Changed:

- Static $\rightarrow$ Contextualized
- Single vector $\rightarrow$ Different vectors per context
- Window $\rightarrow$ Full sentence attention

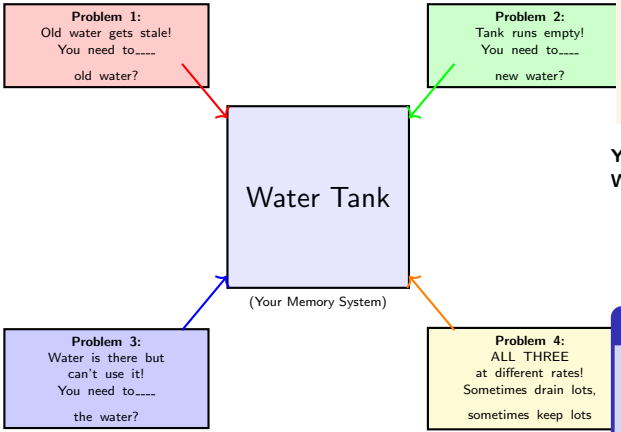
Word2Vec started the revolution, transformers completed it

LSTM - Long Short-Term Memory

Understanding Through a Complete Example

Imagine You're Designing a Memory System

The Challenge:



Why Do We Care About This Tank?

Imagine: It rained yesterday
Question: Should we buy water tomorrow?
To predict, you need to know:

- How much was **DRAINED**? (old water out)
- How much rain was **ADDED**? (new water in)
- How much is **AVAILABLE** now? (can use it?)

→ **Memory state helps make predictions!**

Your Design Task:
What THREE controls would YOU design?

1. **Control #1:** To handle stale water (Hint: Think about draining...)
2. **Control #2:** To handle empty tank (Hint: Think about adding...)
3. **Control #3:** To handle using water (Hint: Think about tapping...)

Checkpoint: Think First!

Key Question: Do these three controls need to be INDEPENDENT?
Can you drain a lot while adding a little? Can you add a lot while using only some?
Your answer: _____

The Core Task: Predict the Next Word

Simple Example:

Input: "The cat was"

↓

LSTM

↓

Predictions:

hungry	35%
sleeping	28%
running	15%
small	8%

How does it work?
Must remember "cat" to predict appropriate adjective/verb!

Why This Matters:

- **Autocomplete**
  Your phone keyboard
- **Translation**
  Google Translate (2016)
- **Text Generation**
  Write stories, code
- **Voice Assistants**
  Siri, Alexa
- **Chatbots**
  Customer service

Checkpoint: The Challenge

To predict well, the model must REMEMBER earlier words in the sentence.
That's what LSTM does brilliantly!

THE PROBLEM

RNNs Cannot Remember

- Vanishing gradients
- $0.5^{50} \approx 0$ (information dies)
- Forgets early context
- Cannot handle long sentences

Example:
"I grew up in Paris. I speak fluent ___"

RNN forgets "Paris" after 20 words
Cannot predict "French"

THE SOLUTION

LSTM Controls Memory

- Three gates (0-1 values)
- Addition path (not multiplication)
- Preserves gradients
- Remembers 100+ steps

The Method:

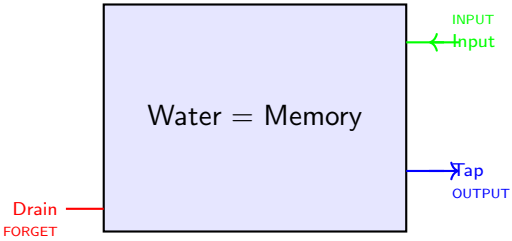
Three Independent Gates:

- FORGET: What to erase
- INPUT: What to add
- OUTPUT: What to use

Let's see how this works with a simple analogy...

Think of Memory as a Water Tank with Three Valves

The Tank System:



How Each Valve Works:

FORGET = Drain Valve
Controls how much water flows OUT
0.1 = Open 10% → 90% drains away
Removes old water (old memory)

INPUT = Input Valve
Controls how much new water flows IN
0.9 = Open 90% → lots added
Adds fresh water (new memory)

OUTPUT = Output Tap

Real Examples:

At period "." in sentence:

- Drain: 90% (0.1 forget)
- Input: 40% (0.4 input)
- Tap: 30% (0.3 output)

→ Tank mostly empties!

At noun "dog":

- Drain: 30% (0.7 forget)
- Input: 90% (0.9 input)
- Tap: 90% (0.9 output)

→ Tank fills up!

Intuition: The Key Insight

Three INDEPENDENT valves on ONE tank!

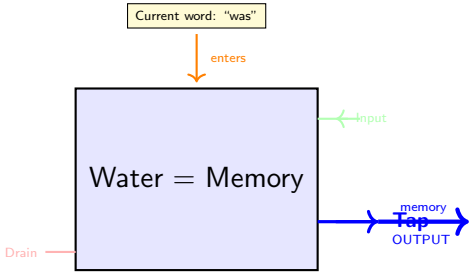
Each valve controls a different aspect:

- Drain: How much OLD to remove
- Input: How much NEW to add
- Tap: How much to USE now

This is EXACTLY what LSTM does!

Continuing The Analogy: From Tank to Predictions

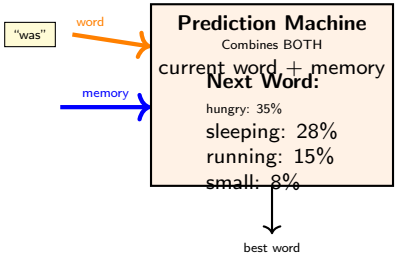
The Complete System:



Two Things Happening:

- Current word enters system
- Memory (tap water) flows out
- BOTH go to prediction!

Where They Go:



The Complete Flow:

Current word: "was"
+
Memory (tap water)
↓
Prediction Machine
↓
Next word: "hungry"!

Checkpoint: Key Insight!

Three Gates Control Memory Like Volume Knobs (0 to 1)

FORGET

$\times$

REMOVES
old information

Value 0-1:

- 0.0 = erase all
- 0.5 = keep half
- 1.0 = keep all

Example:
0.1 at period
→ Erase 90%!

INPUT

$+$

ADDS
new information

Value 0-1:

- 0.0 = add nothing
- 0.5 = add half
- 1.0 = add all

Example:
0.9 on "cat"
→ Store lots!

OUTPUT

$\rightarrow$

REVEALS
stored information

Value 0-1:

- 0.0 = hide all
- 0.5 = show half
- 1.0 = show all

Example:
0.9 at "was"
→ Use memory!

How They Work Together:

New Memory = (Forget × Old Memory) + (Input × New Info)
Output = Output Gate × Memory

Now let's see these gates in action with concrete numbers...

Updating Memory From "cat" to "dog"

Starting Point:

Old Memory: [0.8, 0.6, 0.4]

Contains "cat" information

Step 1: FORGET Gate = 0.1

What it does: Multiply old memory by 0.1

$[0.8, 0.6, 0.4] \times 0.1 = [0.08, 0.06, 0.04]$

Result: 90% erased! "cat" mostly removed.

Why? At period, we need fresh start for new sentence.

Step 2: INPUT Gate = 0.9

What it does: Filter new candidate info

Create candidate: [0.7, 0.5, 0.9]

Multiply by 0.9: $[0.7, 0.5, 0.9] \times 0.9$
$= [0.63, 0.45, 0.81]$

Result: 90% of "dog" info ready to add.

Why? "dog" is new subject, very important!

Visual Flow (Steps 1-2):

Checkpoint: Key Point

Two INDEPENDENT operations:

- FORGET decides what OLD to keep
- INPUT decides what NEW to add
- They don't interfere!

From Separate Results to Final Output

Where We Left Off:

- Erased old: [0.08, 0.06, 0.04]
- Filtered new: [0.63, 0.45, 0.81]

Step 3: COMBINE (Addition!)

What it does: Add the two results together

[0.08, 0.06, 0.04] + [0.63, 0.45, 0.81]
= [0.71, 0.51, 0.85]

Result: Updated memory = "dog" info

Why addition? Preserves gradients! This is the key innovation that prevents vanishing.

Step 4: OUTPUT Gate = 0.9

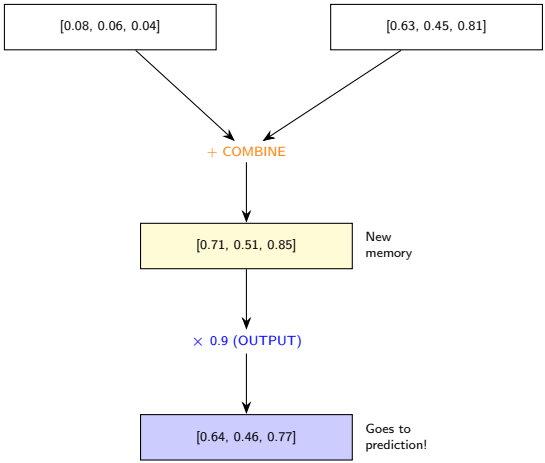
What it does: Filter what network sees

[0.71, 0.51, 0.85] × 0.9
= [0.64, 0.46, 0.77]

Result: 90% revealed to next layer

Why? At "was", we NEED subject info for verb prediction!

Visual Flow (Steps 3-4):



From Hidden State to Prediction

Recap: Output Gate Result

From Step 4: [0.64, 0.46, 0.77]

This is the **hidden state** h_t

Where Does It Go?

1. **To Prediction Layer**
   $h_t \rightarrow$ Linear $\rightarrow$ Softmax $\rightarrow$ Probabilities
   Example at "was": Predict next word
2. **To Next Time Step**
   h_t feeds into next LSTM cell
   Used to compute next gates
3. **Optional: To Attention**
   In seq2seq models, decoder attends to these h_t values

Key Distinction:

- C_t = Long-term memory (protected)
- h_t = Working memory (filtered output)

But what does this look like mathematically?

Concrete Example:

At word "was" in "The dog was sleeping":

Memory C_t contains: [dog, context]
Output gate: 0.9 (reveal 90%)
Hidden state h_t: [0.64, 0.46, 0.77]

Prediction layer receives h_t:
$h_t \rightarrow$ Linear(512 $\rightarrow$ vocab) $\rightarrow$ Softmax

Top predictions:

- "sleeping": 0.35 (verb matches dog)
- "running": 0.18
- "eating": 0.12

The subject info ("dog") in h_t helps predict appropriate verb!

Intuition: Why Filter?

Not all memory is relevant NOW.

Examples:

- At "The": Output gate low (0.2) $\rightarrow$ hide memory, article doesn't need context
- At "was": Output gate high (0.9) $\rightarrow$ reveal memory, verb needs subject!

The OUTPUT gate is smart about WHEN to use memory.

From Water Tank Analogy to Math

What We Just Learned

"Tap water"
"Gate = 0.9"
"Memory flows"

This was the INTUITION

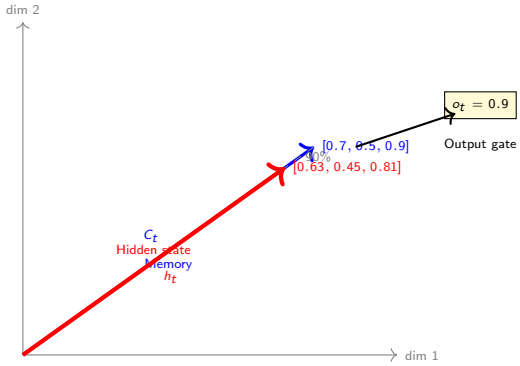
The Translation:

- **Water** $\longrightarrow$ Vector in space
- **Tank** $\longrightarrow$ Memory C_t
- **Tap setting 0.9** $\longrightarrow$ Multiply by 0.9
- **Flowing water** $\longrightarrow$ Scaled vector h_t

Checkpoint: Key Insight

The "tap" is just SCALAR MULTIPLICATION!
Gate value 0.9 = scale each dimension by 0.9

What's Actually Happening:



The Math:

Memory vector: $C_t = [0.7, 0.5, 0.9]$
Output gate: $o_t = 0.9$ (the "tap setting")
Element-wise multiplication:

Why Three Separate Gates? Real Scenarios

Reading: "The cat sat. The dog..."

Scenario 1: At "cat"

Gate Values:

- F = 0.8 (keep)
- I = 0.9 (STORE!)
- O = 0.8 (show)

What Happens:

- Keep previous context
- STORE subject strongly
- Show it to network

Goal:
Remember "cat" for rest of sentence

Memory:
→ [cat, context]

Scenario 2: At "."

Gate Values:

- F = 0.1 (ERASE!)
- I = 0.4 (small)
- O = 0.3 (HIDE)

What Happens:

- ERASE old sentence
- Small punctuation add
- HIDE memory

Goal:
Clean slate for new sentence

Memory:
→ [mostly empty]

Scenario 3: At "dog"

Gate Values:

- F = 0.7 (keep some)
- I = 0.9 (NEW!)
- O = 0.9 (REVEAL!)

What Happens:

- Keep some context
- STORE new subject
- REVEAL all info

Goal:
New focus, need it NOW

Memory:
→ [dog, some context]

Key Insight: Each situation needs DIFFERENT gate values!
That's why LSTM has three independent gates, not just one.
The network LEARNS which values to use for each word.

Now let's watch these gates in action on a real sentence...

Watch LSTM Process a Sentence

Sentence: "The cat was hungry. The dog was sleeping."

Word	Forget	Input	Output	Memory State
The	0.9	0.3	0.2	*article*
cat	0.8	**0.9**	0.8	*subject: cat*
was	0.9	0.7	0.9	*cat + verb*
hungry	0.8	0.8	0.7	*cat is hungry*
.	**0.1**	0.4	0.3	*sentence ends*
The	0.1	0.8	0.2	*new article*
dog	0.7	**0.9**	0.9	*subject: dog*
was	0.9	0.8	**0.9**	*using dog info*

0.1 = Forget

0.9 = Store/Use

Period → Reset

Notice the patterns? Let's explore what you observed...

What Did You Notice?

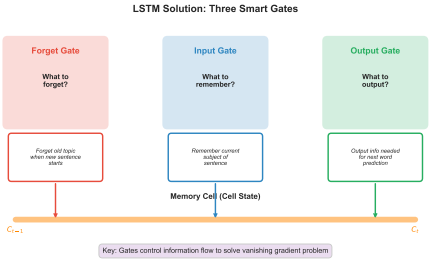
Common Observations:

Students usually notice:

- **"It drops to 0.1 at the period!"**
- **"It's 0.9 on important words (cat, dog)"**
- "The memory changes from cat to dog"
- "It resets between sentences"
- "Three different columns of numbers"

Key Questions:

1. HOW does it know to forget at period?
2. HOW does it know cat and dog are important?
3. HOW does it decide when to use memory?



Checkpoint: The Big Reveal

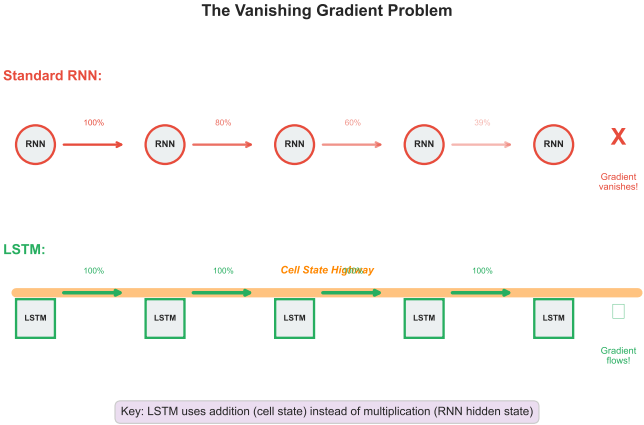
Those three columns are called **GATES**:

- Forget Gate: Controls what to erase
- Input Gate: Controls what to store
- Output Gate: Controls what to use

But WHY do we need gates?

Let's find out

Why Do We Need Controlled Memory?



RNN Problem:

- Gradients vanish ($0.5^{50} \approx 0$)
- Forgets early information
- Can't handle long dependencies
- Would lose "cat" by "dog"

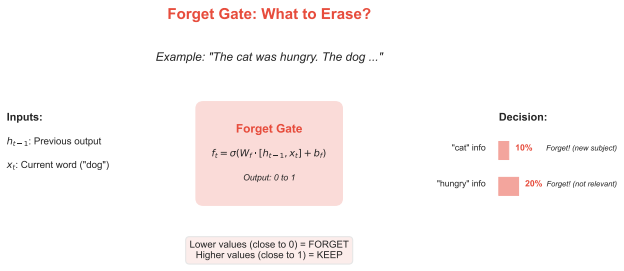
Example: "I grew up in Paris. I speak fluent ___"

LSTM Solution:

- Cell state highway (addition not multiplication)
- Three gates for CONTROL
- Can preserve info for 100+ steps
- Then ERASE when sentence ends

Remember Our Table?

Forget Gate: How We Get That 0.1



Back to Our Table - Row 5:

Word	Forget
"."	**0.1**

What This 0.1 Means:

- 0.0 = forget everything
- 1.0 = keep everything
- 0.1 = forget 90% (keep only 10%)

Why at period?

- New sentence starting

The Formula That Produces 0.1:

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)$$

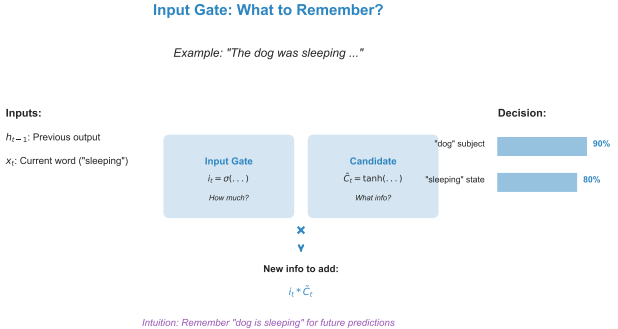
How It Decides:

1. Look at current word (".")
2. Look at previous hidden state
3. Compute weighted sum
4. Apply sigmoid $\rightarrow$ output 0 to 1

Cell State Update:

$$C_t = f_t \odot C_{t-1} + \ldots$$

Input Gate: How We Get That 0.9



Back to Our Table - Row 7:

Word	Input
"dog"	**0.9**

What This 0.9 Means:

- 0.0 = add nothing
- 1.0 = add everything
- 0.9 = add 90% of candidate

Why at "dog"?

- New subject appearing

The Formulas (Two Parts):

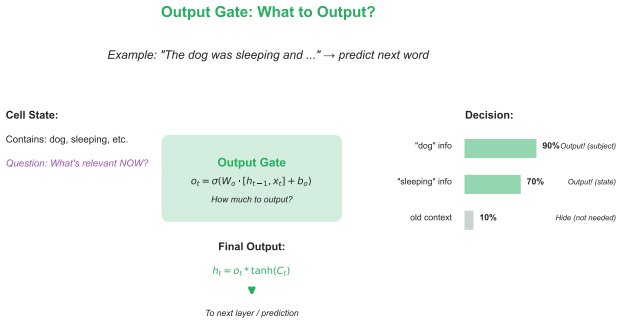
$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$$

$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C)$$

How It Works:

1. Create candidate info ($\tilde{C}_t$) with tanh
2. Decide how much to use ($i_t = 0.9$)
3. Multiply: 0.9 $\times$ candidate
4. Add to cell state

Output Gate: When to USE Memory



Back to Our Table - Row 8:

Word	Output
"was"	**0.9**

What This 0.9 Means:

- 0.0 = hide everything
- 1.0 = reveal everything
- 0.9 = output 90% of memory

Why at "was"?

- Need to predict next word

The Formulas:

$$o_t = \sigma(W_O \cdot [h_{t-1}, x_t] + b_o)$$

$$h_t = o_t \odot \tanh(C_t)$$

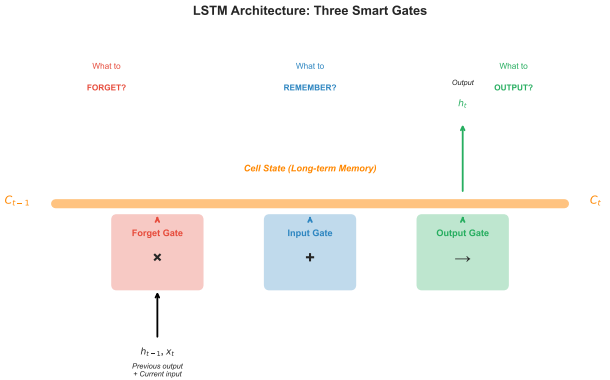
How It Works:

1. Look at cell state (has "dog" info)
2. Decide what's relevant NOW
3. Filter memory through gate (0.9)
4. Send h_t to prediction layer

Key Insight:

Cell state C_t stays protected (long-term memory)

The Big Picture: Three Gates Working Together



The Cell State Highway:

- Protected memory channel
- Information flows easily
- Gates control entry/exit
- Gradients don't vanish!

At Each Time Step:

1. **Forget:** Erase old (0.1 → erase "cat")
2. **Input:** Add new (0.9 → add "dog")
3. **Output:** Use (0.9 → use "dog" info)

Intuition: Visual Analogy

Think of LSTM like a notebook:

- Forget Gate = Eraser
  (Clear old notes at period)
- Input Gate = Pen
  (Write important info like "dog")

Now Look Again - You Understand EVERYTHING!

Sentence: "The cat was hungry. The dog was sleeping."

Word	Forget	Input	Output	What LSTM "Thinks"
The	0.9 (keep)	0.3 (weak)	0.2 (hide)	Article seen, nothing special yet
cat	0.8 (keep)	**0.9** (STORE!)	0.8 (show)	Subject! Important noun!
was	0.9 (keep)	0.7 (add)	0.9 (need!)	Verb connects to cat
hungry	0.8 (keep)	0.8 (add)	0.7 (show)	Describes the cat's state
.	**0.1** (ERASE!)	0.4 (end)	0.3 (hide)	Sentence over! Clear memory!
The	0.1 (clear)	0.8 (new!)	0.2 (hide)	NEW sentence starts fresh
dog	0.7 (keep)	**0.9** (NEW!)	0.9 (use!)	NEW subject! (forgot cat)
was	0.9 (keep)	0.8 (add)	**0.9** (USE!)	Using DOG info for prediction

Checkpoint: The Magic Transition

Watch rows 4→5→6→7: **hungry** → **.** → **The** → **dog**
Memory Evolution: [cat, hungry] → FORGET (0.1) → [end] → ADD (0.9) → [dog]
This intelligent memory control is what RNNs cannot do! LSTM uses gates to:

- Preserve important info (0.9 on subject nouns)
- Erase when context changes (0.1 at sentence boundaries)
- Reveal info when needed (0.9 output for predictions)

Summary: From Table to Understanding

Your Learning Journey:

1. **Intuition:** Water tank analogy
   (Three valves on one tank)
2. **Concepts:** What gates are
   (Volume knobs from 0 to 1)
3. **Mechanics:** How they work
   (4-step process with real numbers)
4. **Usage:** What happens to output
   (Hidden state $\rightarrow$ prediction layer)
5. **Mastery:** Complete understanding
   (Table makes perfect sense now!)

Key Equations:

$$f_t = \sigma(W_f[h_{t-1}, x_t] + b_f)$$
$$i_t = \sigma(W_i[h_{t-1}, x_t] + b_i)$$
$$o_t = \sigma(W_o[h_{t-1}, x_t] + b_o)$$
$$C_t = f_t \odot C_{t-1} + i_t \odot \tilde{C}_t$$
$$h_t = o_t \odot \tanh(C_t)$$

Real World: Where LSTMs Excel

Applications (2015-2020):

- Machine Translation (Google Translate)
- Speech Recognition (Siri, Alexa)
- Text Generation (early GPT)
- Video Analysis
- Music Generation
- Handwriting Recognition

Modern Context (2024):
Transformers now dominate NLP, but LSTMs:

- Still used in time series
- Efficient for streaming data
- Foundation for understanding attention

The Core Insight:
That table showed you *exactly* how gates work. Every 0.1 and 0.9 has a purpose. That's the real magic of LSTMs!

Questions?

Natural Language Processing

Week 4: The Compression Journey

From Meaning to Numbers and Back Again

Imagine You're Designing a Translation System

Your Challenge:
Translate this 40-word sentence into French:

"The International Conference on Machine Learning, which is one of the premier venues for presenting research in machine learning and attracts submissions from researchers around the world, accepted our paper."

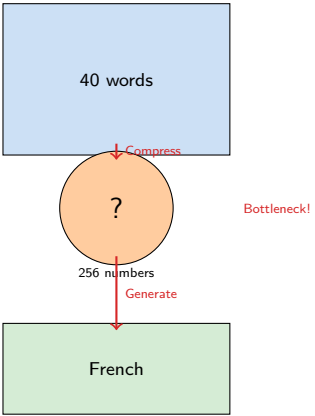
Design Constraints:

1. You can only write down 256 numbers total
2. These numbers must capture ALL the meaning
3. Then translate from just those 256 numbers
4. You cannot look back at the original!

Question: Can 256 numbers really hold 40 words of meaning? What happens to:

- "International" (important detail)
- "premier venues" (significance)
- "researchers around the world" (scale)

Your Design:



Think about it:

Learning Objectives

By the end of this lecture, you will understand:

1. Why we need numbers to represent words (from first principles)
2. How compression creates an information bottleneck
3. What "context" and "hidden state" actually mean
4. How attention solves the compression problem
5. Why this matters for all modern NLP

Prerequisites from Week 3:

- Basic understanding that neural networks process numbers
- Concept of sequential processing (RNN idea)
- Backpropagation intuition

Table of Contents

The Core Problem: Computers Don't Understand Words

Start with the fundamental challenge:

You want to translate: "The black cat sat on the mat" $\rightarrow$ French

The Computer's Dilemma:

- Computer sees: `['T', 'h', 'e', ' ', 'b', 'l', 'a', ...]`
- These are just character codes (bytes)
- extbfNo meaning, no relationships, no structure

What the computer sees:

- "cat" = [99, 97, 116]
- "dog" = [100, 111, 103]
- "sat" = [115, 97, 116]

Problem:

- "cat" and "sat" share [97, 116]
- Does that mean they're similar?
- No! Character overlap $\neq$ meaning

Key Question: How do we give computers a "numerical understanding" of word meaning?

From Words to Numbers: The Embedding Idea

The solution: Represent each word as a vector of numbers

Build intuition with simple example:
Imagine describing animals with just 3 properties:

- Size (0=tiny, 1=huge)
- Cuteness (0=scary, 1=adorable)
- Speed (0=slow, 1=fast)

Word	Size	Cute	Speed
cat	0.3	0.9	0.6
dog	0.5	0.8	0.5
mouse	0.1	0.7	0.8
elephant	0.95	0.4	0.2

Now computers can compute:

- Similarity: cat $\approx$ dog (vectors are close)
- Difference: cat $\neq$ elephant (vectors are far)
- extbfThis is called a "word embedding"

Reality: We use 100-300 dimensions (not just 3), learned from data

Quick Check: Why embeddings? Computers only process numbers. Embeddings give words numerical meaning so similar words have similar vectors (cat $\approx$ dog).

What is a "Hidden State"? Building Intuition

Now we have numbers for words. Next problem: Understanding sentences

Human analogy - Reading comprehension:
As you read "The black cat sat on the mat":

1. After "The" → You know: article, something coming
2. After "The black" → You know: a dark-colored thing
3. After "The black cat" → You know: a specific animal
4. After full sentence → You know: complete scene

Your "understanding" evolves as you read!

Neural network equivalent:

- Network maintains a vector that represents "current understanding"
- This vector updates with each new word
- extbfThis evolving vector is called the "hidden state"
- Final hidden state = complete understanding of sentence

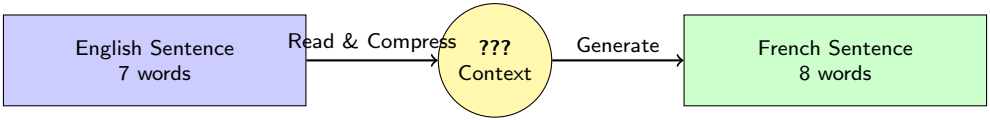
Technical name: When we update understanding word-by-word, we call this a "Recurrent Neural Network" (RNN from Week 3)

The Compression Problem Emerges

Now the real challenge appears:

We need to translate: "The black cat sat on the mat" → "Le chat noir s'est assis sur le tapis"

Two-stage process (like human translation):



The bottleneck:

- 7 words of meaning → compressed to 256 numbers
- Then generate 8 words from just those 256 numbers
- extbfCan 256 numbers hold all the information?

Key Question: What happens with longer sentences? 100 words → still 256 numbers?

Quantifying the Compression Problem

Let's calculate how much compression we're doing:

Information content (rough estimate):

- Each word embedding: 100 dimensions (numbers)
- 7-word sentence: $7 \times 100 = 700$ numbers of information
- Context vector: **only 256 numbers**
- extbfCompression ratio: 700:256 $\approx$ 2.7:1

What about longer sentences?

Length	Input Dims	Context Dims	Ratio	Quality
5 words	500	256	2:1	Good
20 words	2000	256	8:1	Mediocre
50 words	5000	256	20:1	Poor
100 words	10000	256	40:1	Very Poor

The Information Bottleneck: Longer sentences lose more information! Like trying to fit a whole book into a single paragraph - something must be lost.

Next question: Can we avoid this bottleneck? (Spoiler: Yes, with attention!)

The Two-Network Architecture

The key insight: Separate "reading" from "writing"

Why two networks? Build from human behavior:
When YOU translate:

1. **Phase 1 (Reading):** Read and understand the English sentence
   - Process word-by-word
   - Build complete understanding
   - Store meaning in your memory
2. **Phase 2 (Writing):** Generate the French translation
   - Start from your understanding
   - Generate word-by-word in French
   - Use grammar and vocabulary of target language

Neural equivalent:

- **Encoder network:** Reads input, builds "hidden state" (understanding)
- **Context vector:** Final hidden state (compressed meaning)
- **Decoder network:** Generates output from context

Technical names you now understand:

- "Sequence-to-Sequence" (Seq2Seq) = this two-network setup
- "Encoder-Decoder architecture" = same thing

Why Two Networks? FIRST fully understand the source (encoder), THEN generate the target (decoder). Mixing would be confusing - like speaking French while still reading English!

Encoder: Building Understanding Step-by-Step

Concrete example: Encoding "The cat sat"

Step-by-step processing:

Step 1: Read "The"

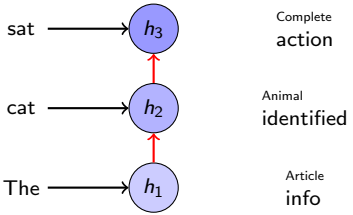
- Convert to embedding: [0.2, 0.5, -0.1, ...] (100d)
- Initial understanding: $h_0 = [0, 0, 0, ...]$ (256d)
- Update: $h_1 = \text{RNN}(\text{"The"}, h_0)$
- New understanding: [0.1, -0.05, 0.03, ...] (256d)

Step 2: Read "cat"

- Embedding: [0.7, -0.3, 0.4, ...] (100d)
- Previous understanding: h_1
- Update: $h_2 = \text{RNN}(\text{"cat"}, h_1)$
- New understanding: [0.3, 0.2, -0.1, ...] (256d)

Step 3: Read "sat"

- Embedding: [-0.2, 0.6, 0.1, ...]
- Update: $h_3 = \text{RNN}(\text{"sat"}, h_2)$
- extbfFinal understanding: h_3 = context vector



Key insight:

- Each h_t = accumulated understanding
- Always 256 dimensions
- Final h_3 goes to decoder

Decoder: Generating from Understanding

Now generate French from the context vector

Generation process:

Step 0: Start

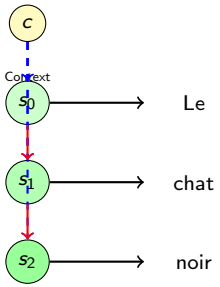
- Input: ¡START¿ token
- Context: $c = h_3$ from encoder (256d)
- Generate: $s_0 = \text{RNN}(\text{¡START¿}, c)$
- Predict probabilities: P("Le") = 0.6, P("Un") = 0.3, ...
- extbfChoose "Le"

Step 1: Continue

- Input: "Le" (what we just generated)
- Context: still c (same context!)
- Generate: $s_1 = \text{RNN}(\text{"Le"}, s_0, c)$
- Predict: P("chat") = 0.7, P("chien") = 0.2, ...
- extbfChoose "chat"

Step 2: Continue until ¡END¿

- Input: "chat"
- Generate: $s_2 = \text{RNN}(\text{"chat"}, s_1, c)$



Key observations:

- Context c used at every step
- Previous word fed back in
- Generate one word at a time
- Stop when ¡END¿ predicted

The First Success: Short Sentences Work Great!

Initial results (5-10 word sentences):

English	French Translation	Result
The cat sat	Le chat s'est assis	Perfect!
I love you	Je t'aime	Perfect!
Hello world	Bonjour le monde	Perfect!
Good morning	Bonjour	Perfect!
See you later	A plus tard	Perfect!

Performance on short sentences:

- 5-10 words: **BLEU 35.2** (excellent quality)
- Captures meaning correctly
- Word order appropriate
- Grammar correct

Breakthrough moment: For the first time, neural networks can translate sentences end-to-end! No hand-crafted rules, no phrase tables - just learned from data.

Key Question: If it works so well for short sentences, what happens with long ones?

The Failure Pattern Emerges

Testing with longer sentences reveals a problem:

Experimental results (Bahdanau et al., 2014):

Sentence Length	Compression Ratio	BLEU Score	Quality Drop
5-10 words	2:1	35.2	Baseline
10-20 words	5:1	28.5	-19%
20-30 words	10:1	18.7	-47%
30-40 words	15:1	12.4	-65%
40+ words	20:1	8.1	-77%
Pattern: Quality drops as compression ratio increases			

The trend is clear:

- Short sentences (¡ 10 words): Excellent
- Medium sentences (10-20 words): Good
- Long sentences (20-30 words): Poor
- Very long (30+ words): Terrible

The Pattern: Performance degrades predictably with sentence length! Something systematic is failing as inputs get longer.

Diagnosing the Bottleneck

What gets lost in long sentences? Let's trace it:

Input sentence (42 words):
"The International Conference on Machine Learning, which is one of the premier venues for presenting research in machine learning and attracts submissions from researchers around the world, accepted our paper."

Compressed to 256 numbers...

What Survived:

- General topic: ML conference
- Sentiment: Positive
- Main fact: Paper accepted
- Basic structure: Conference does X

Capacity used: 200/256 numbers

What Got Lost:

- "International" modifier
- "premier venues" importance
- "researchers around the world" scale
- Exact conference name
- "submissions" detail

Overflow: 42 words $\rightarrow$ need 420 numbers, only have 256!

Root cause analysis:

- Fixed 256-number container for ANY sentence length
- Information overflow gets discarded
- Network keeps only high-level summary
- Details necessarily lost

The Human Translation Insight

Introspection exercise: How do YOU actually translate?

Translating: "The black cat sat on the mat" → "Le chat noir s'est assis sur le tapis"

Honest observation:

- Writing "Le" → You look back at "The"
- Writing "chat" → You look back at "cat"
- Writing "noir" → You look back at "black"
- Writing "s'est assis" → You look back at "sat"
- Writing "sur" → You look back at "on"
- Writing "le" → You look back at "the"
- Writing "tapis" → You look back at "mat"

Critical realization:

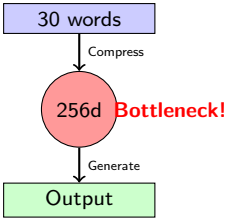
- You DON'T compress everything into one memory
- You keep the original English visible
- You extbfselectively attend to relevant words
- Different output words need different input words

Aha Moment: Humans don't compress - they SELECT!

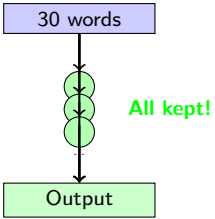
The Attention Hypothesis

From compression to selection:

Old Way (Compression):



New Way (Selection):



Problem:

- 30 words → 1 vector
- Information loss inevitable
- Same context for all outputs

Solution:

- Keep ALL encoder states
- Select relevant ones per output
- Different context each time

Key Insight: Don't throw away information, just focus on what matters!

Analogy: Instead of summarizing a book, keep the full book and read relevant pages as needed.

Attention = Weighted Relevance (Zero Jargon)

Breaking down "attention" into simple concepts:

Setup:

- 5 source words: [The, black, cat, sat, on]
- Generating French word "chat" (cat)
- Question: How relevant is each source word?

Intuitive relevance scores:

Word	Relevance	Why
The	5%	Generic article
black	15%	Describes cat
cat	70%	Direct match!
sat	5%	Action, not noun
on	5%	Preposition
Total	**100%**	Must sum to 1

What these percentages do:

Context for "chat" = weighted average:

$$
\begin{aligned}
&= 0.05 \times h_{\text{The}} \\
&+ 0.15 \times h_{\text{black}} \\
&+ 0.70 \times h_{\text{cat}} \\
&+ 0.05 \times h_{\text{sat}} \\
&+ 0.05 \times h_{\text{on}}
\end{aligned}
$$

Result: Context is *mostly* "cat" info, with a bit of everything else

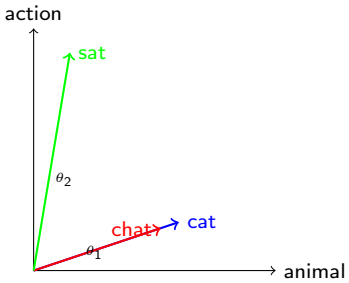
extbfThese percentages ARE the attention weights!

Why Dot Product Measures Relevance

Geometric intuition (building from 2D):

Question: How does the network compute those relevance percentages?

Vectors as arrows in space:



Properties of meaning:

- "cat" = [0.8 animal, 0.2 action]
- "chat" = [0.7 animal, 0.2 action]
- "sat" = [0.2 animal, 0.9 action]

Similar meanings → similar directions!

Dot product measures alignment:

cat · chat:

$$= (0.8 \times 0.7) + (0.2 \times 0.2)$$
$$= 0.56 + 0.04 = 0.60$$

cat · sat:

$$= (0.8 \times 0.2) + (0.2 \times 0.9)$$
$$= 0.16 + 0.18 = 0.34$$

Mathematical property:

- Aligned vectors → High value
- Perpendicular → Zero
- Opposite → Negative

extbfHigher dot product = more relevant!

The 3-Step Attention Mechanism

Now we can understand the full algorithm:

Step 1: Score (Measure Relevance)
For each encoder state, compute dot product with decoder state:

$$score_i = h_{\text{decoder}} \cdot h_i^{\text{encoder}}$$

Why? Dot product = alignment = relevance (from previous slide)

Step 2: Normalize (Make Probabilities)
Convert scores to weights that sum to 100%:

$$\alpha_i = \frac{\exp(score_i)}{\sum_j \exp(score_j)}$$

Why? Need weights for weighted average
Tool: Softmax function (ensures positive, sums to 1)

Step 3: Combine (Weighted Average)
Take weighted sum of encoder states:

$$context = \sum_i \alpha_i \cdot h_i^{\text{encoder}}$$

Why? Focus mostly on relevant, a bit on others

Example weights:

- α_1 (The) = 0.05
- α_2 (black) = 0.15
- α_3 (cat) = 0.70
- α_4 (sat) = 0.05
- α_5 (on) = 0.05

Context is mostly "cat"!

Key Property: Context is *dynamic* - recomputed for each output word with different weights!

3 Steps of Attention: (1) Score: dot product for relevance, (2) Normalize: softmax for weights, (3) Combine: weighted average. Result = dynamic

Attention Calculation: Full Numerical Example

Let's trace generating "chat" with actual numbers:

Given:

- Decoder state: $h_{\text{dec}} = [0.5, -0.2, 0.8]$
- Encoder states: $h_1 = [0.1, 0.2, 0.1]$ (The), $h_2 = [0.8, 0.1, 0.7]$ (cat), $h_3 = [0.2, 0.3, 0.2]$ (sat)

Step 1: Compute scores (dot products)

$$\begin{aligned} score_1 &= [0.5, -0.2, 0.8] \cdot [0.1, 0.2, 0.1] \\ &= (0.5)(0.1) + (-0.2)(0.2) + (0.8)(0.1) \\ &= 0.05 - 0.04 + 0.08 = 0.09 \end{aligned}$$

$$\begin{aligned} score_2 &= [0.5, -0.2, 0.8] \cdot [0.8, 0.1, 0.7] \\ &= (0.5)(0.8) + (-0.2)(0.1) + (0.8)(0.7) \\ &= 0.40 - 0.02 + 0.56 = 0.94 \leftarrow \text{Highest!} \end{aligned}$$

$$\begin{aligned} score_3 &= [0.5, -0.2, 0.8] \cdot [0.2, 0.3, 0.2] \\ &= (0.5)(0.2) + (-0.2)(0.3) + (0.8)(0.2) \\ &= 0.10 - 0.06 + 0.16 = 0.20 \end{aligned}$$

Step 2: Softmax to weights

$$\begin{aligned} \alpha_1 &= \frac{e^{0.09}}{e^{0.09} + e^{0.94} + e^{0.20}} \\ &= \frac{1.09}{4.02} = 0.27 \end{aligned}$$

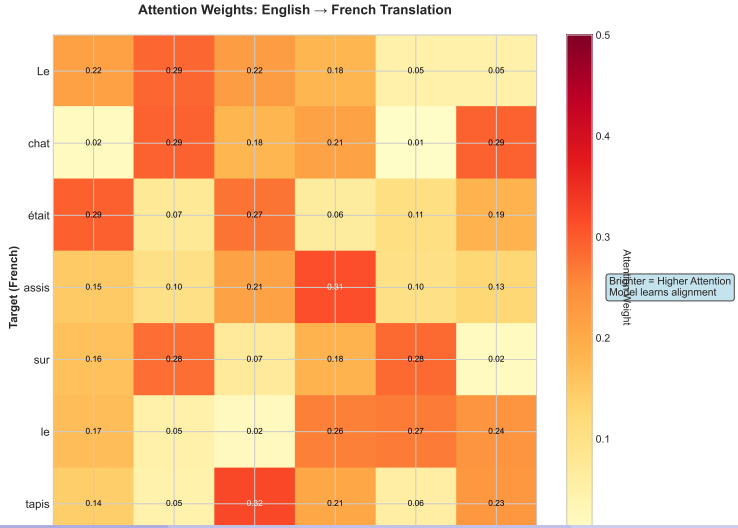
$$\begin{aligned} \alpha_2 &= \frac{e^{0.94}}{4.02} \\ &= \frac{2.56}{4.02} = 0.63 \end{aligned}$$

$$\alpha_3 = \frac{e^{0.20}}{4.02}$$

$$1.22$$

Visualizing Attention: The Alignment Matrix

Attention weights reveal what the model is "looking at":



Why Attention Solves the Bottleneck

Information capacity comparison:

Without Attention:

- 30 words compressed to 256d vector
- Capacity: 256 numbers (fixed)
- 30 words = 3000 numbers needed
- Overflow: 2744 numbers lost!
- Same context for all outputs

Information loss:

- 91% of information discarded
- Only high-level summary kept
- Details necessarily lost

With Attention:

- Keep all 30 encoder states
- Capacity: $30 \times 256 = 7680$ numbers
- All information preserved
- Select relevant subset per output
- Dynamic context each time

Information preserved:

- 100% of information available
- Focus on relevant parts
- No forced compression

The Key Insight: Dynamic selection beats static compression! Instead of "compress everything to 256 numbers", use "keep everything, select as needed"

Attention Results: The Vindication

Performance comparison validates the hypothesis:

Sentence Length	No Attention	With Attention	Improvement
5-10 words	35.2 BLEU	36.1 BLEU	+2.6%
10-20 words	28.5 BLEU	32.7 BLEU	+14.7%
20-30 words	18.7 BLEU	28.9 BLEU	+54.5%
30-40 words	12.4 BLEU	24.8 BLEU	+100%
40+ words	8.1 BLEU	24.3 BLEU	+200%

The pattern:

- Short sentences: Small improvement (bottleneck wasn't the problem)
- Medium sentences: Moderate improvement (bottleneck starts to matter)
- Long sentences: extbfMassive improvement (bottleneck was killing performance)

Validation: Attention solves exactly the problem we diagnosed! Improvement is largest where bottleneck hurt most (long sentences).

Historical Impact: This 2015 paper (Bahdanau et al.) launched the attention revolution in NLP.

Implementing Attention (Surprisingly Simple)

The complete mechanism in code:

```
def attention(decoder_state, encoder_states):
    """
␣␣␣␣decoder_state:␣[256]␣-␣current␣decoder␣hidden␣state
␣␣␣␣encoder_states:␣[seq_len,␣256]␣-␣all␣encoder␣states
␣␣␣␣Returns:␣context␣[256],␣attention_weights␣[seq_len]
␣␣␣␣"""
    scores = []

    for enc_state in encoder_states:
        score = dot(decoder_state, enc_state)
        scores.append(score)

    scores = array(scores)

    exp_scores = exp(scores - max(scores))
    attention_weights = exp_scores / sum(exp_scores)

    context = zeros(256)
    for i, enc_state in enumerate(encoder_states):
        context += attention_weights[i] * enc_state

    return context, attention_weights
```

Three operations:

1. **Lines 9-11:** Dot products (relevance scores)
2. **Lines 15-16:** Softmax (probabilities)
3. **Lines 18-20:** Weighted sum (dynamic context)

That's it!
Just 3 operations:

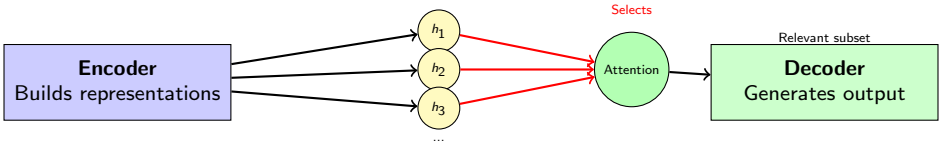
- Dot product (similarity)
- Softmax (normalize)
- Weighted sum (combine)

Key difference:
Context recomputed EVERY step with different weights!

Checkpoint: Can you trace what happens when decoder generates "chat" with input "The cat sat"?

The Three Key Ideas Combined

Unified architecture diagram:



The three innovations:

1. **Two-stage architecture:** Separate reading (encoder) from writing (decoder)
   - Handles variable-length input and output
   - Mimics human translation process
2. **Sequence-to-sequence:** No fixed input/output size
   - 3 words in $\rightarrow$ 2 words out (possible!)
   - 100 words in $\rightarrow$ 50 words out (possible!)
3. **Attention mechanism:** Dynamic selection over static compression
   - Solves information bottleneck
   - Provides interpretability
   - Enables long-sentence translation

What We Learned (Conceptual Insights)

Beyond seq2seq - general lessons:

1. **Compression Trade-off:** Information capacity fundamentally limits performance
   - Can't fit arbitrary information into fixed size
   - Longer inputs → worse compression → lost details
   - Quantifiable: compression ratio predicts quality degradation

2. **Selection ¿ Compression:** For complex tasks, keep everything and select
   - Don't throw away information prematurely
   - Dynamic selection more flexible than static summary
   - "Soft" selection (weighted average) enables gradient flow

3. **Learned Alignment:** Network discovers correspondences without supervision
   - Attention weights show word alignments
   - Model learns which source words matter for each output
   - Interpretable - we can visualize reasoning

4. **Differentiable Operations:** All steps trainable via backpropagation
   - Score, softmax, weighted sum all have gradients
   - End-to-end learning of entire system
   - No hand-crafted alignment rules needed

Beyond Translation: 2024 Applications

The attention explosion across AI:

Language (Original):

- Machine translation (133 languages)
- Text summarization
- Question answering
- Dialogue systems

Vision:

- Image captioning (attend to regions)
- Visual question answering
- Object detection
- Image generation (DALL-E)

Historical timeline:

- 2014: Seq2seq (encoder-decoder)
- 2015: Attention mechanism (this lecture!)
- 2017: Transformers ("Attention is All You Need")
- 2018+: BERT, GPT, current AI revolution

Speech:

- Speech recognition (attend to audio frames)
- Speech synthesis
- Real-time translation

Modern AI:

- **Transformers:** Pure attention (Week 5!)
- GPT-4, Claude, Gemini
- Multimodal models (CLIP)
- All modern LLMs use attention

extbfAttention is the foundation of all modern AI systems!

Summary: The Complete Compression Journey

What you now understand from first principles:

1. **Why embeddings:** Computers need numbers, embeddings give numerical meaning
   - Similar words → similar vectors
   - From bytes to meaning
2. **Why hidden states:** Capture evolving understanding as we read
   - Accumulates meaning word-by-word
   - Final state = complete sentence understanding
3. **Why encoder-decoder:** Separate reading from writing
   - Handles variable lengths
   - Mimics human translation
4. **Why context vectors:** Compress meaning, but creates bottleneck
   - Fixed size for any input
   - Information overflow gets lost
5. **Why attention:** Solve bottleneck by keeping all states and selecting
   - Dynamic selection beats compression
   - 200% improvement on long sentences
6. **Why dot product:** Geometric measure of relevance (vector alignment)
   - Similar directions → high value
   - Differentiable for training

Next week: Remove encoder/decoder RNNs, use ONLY attention → Transformers!

Appendix A: Seq2Seq Mathematics - Complete Equations

Encoder (RNN Processing):

At each time step $t = 1, 2, \ldots, T_x$:

1. Embedding lookup:

$$x_t = \text{Embed}(w_t) \in \mathbb{R}^{d_{\text{emb}}}$$

2. Encoder hidden state:

$$h_t^{\text{enc}} = \text{RNN}_{\text{enc}}(x_t, h_{t-1}^{\text{enc}})$$

Explicitly:

$$h_t^{\text{enc}} = \tanh(W_x x_t + W_h h_{t-1}^{\text{enc}} + b_h)$$

where $h_t^{\text{enc}} \in \mathbb{R}^{d_h}$ (typically 256-512d)

3. Context vector:

$$c = h_{T_x}^{\text{enc}}$$

(Final encoder state = compressed meaning)

Encoder Dimensions:

- Vocabulary: $|V| = 10,000$ to $50,000$
- Embedding: $d_{\text{emb}} = 128$ to 512

Decoder (Generation):

Initialize: $s_0 = c$ (context from encoder)

At each generation step $t = 1, 2, \ldots, T_y$:

1. Decoder hidden state:

$$s_t = \text{RNN}_{\text{dec}}(y_{t-1}, s_{t-1}, c)$$

Explicitly:

$$s_t = \tanh(W_y y_{t-1} + W_s s_{t-1} + W_c c + b_s)$$

2. Output distribution:

$$P(y_t \mid y_{<t}, x) = \text{softmax}(W_o s_t + b_o)$$

3. Teacher forcing (training):

$$y_{t-1} = y_{t-1}^* \quad \text{(use true previous word)}$$

4. Loss function:

$$L = -\sum^{T_y} \log P(y_t^* \mid y_{<t}, x)$$

Appendix B: Attention Mechanism - Mathematical Derivation

The Attention Computation:

Given encoder hidden states $\{h_1^{\text{enc}}, \ldots, h_{T_x}^{\text{enc}}\}$ and decoder state s_t:

Step 1: Alignment Scores
Compute relevance of each encoder state:

$$e_{t,i} = \text{score}(s_t, h_i^{\text{enc}})$$

Common scoring functions:

Dot product (Luong attention):

$$e_{t,i} = s_t^T h_i^{\text{enc}}$$

Additive (Bahdanau attention):

$$e_{t,i} = v^T \tanh(W_s s_t + W_h h_i^{\text{enc}})$$

Step 2: Attention Weights
Normalize scores to probabilities:

$$\alpha_{t,i} = \frac{\exp(e_{t,i})}{\sum_{j=1}^{T_x} \exp(e_{t,j})}$$

Step 3: Context Vector
Weighted sum of encoder states:

$$c_t = \sum_{i=1}^{T_x} \alpha_{t,i} h_i^{\text{enc}}$$

Key property: Context c_t is DYNAMIC - recomputed at each decoder step with different weights!

Modified Decoder with Attention:

$$s_t = \text{RNN}_{\text{dec}}(y_{t-1}, s_{t-1}, c_t)$$

Why Softmax?

1. **Normalization:** Ensures weights sum to 1
2. **Differentiability:** Smooth function for backprop
3. **Sparsity:** Exponentiation amplifies differences:

$$e_1 = 0.9, e_2 = 0.1 \rightarrow \alpha_1 = 0.64, \alpha_2 = 0.36$$

$$e_1 = 5.0, e_2 = 1.0 \rightarrow \alpha_1 = 0.98, \alpha_2 = 0.02$$

Computational Complexity:

Transformers: Understanding Parallel Intelligence

From Zero to ChatGPT - A BSc Journey

Week 5: Transformers

What You'll Learn Today

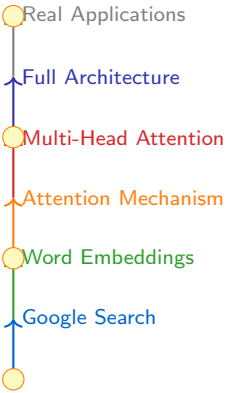
By the end of this lecture, you will:

1. **Understand** how words become numbers (embeddings)
2. **Explain** why parallel beats sequential processing
3. **Calculate** attention scores between words
4. **Draw** the transformer architecture
5. **Identify** transformers in daily applications

Prerequisites:

- Basic vector operations (dot product)
- Matrix multiplication concept
- No deep learning needed!

Your Learning Journey:



Interactive Elements:
3 Checkpoints — 5 Exercises — 10 Visuals

How Google Reads Your Mind

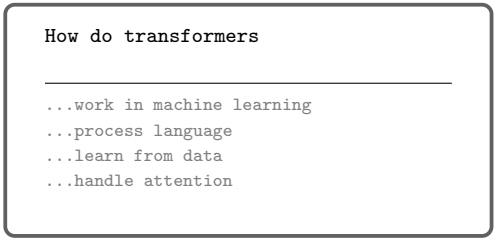
Try this: Type in Google: "How do transformers..."

Google instantly suggests:

- "...work in machine learning"
- "...process language"
- "...learn from data"

The Mystery:

- Google reads ALL your words at once
- Not word-by-word like old systems
- Understands context instantly



Question: How does it understand whole sentences simultaneously?

Discovery 1: Words Live in Space

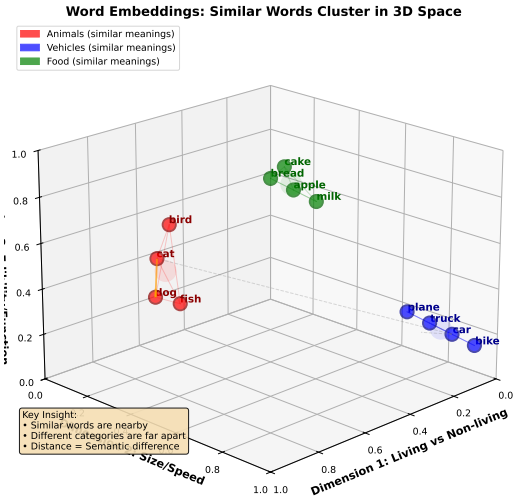
Think about GPS coordinates:

- Paris: (48.8N, 2.3E, 35m altitude)
- London: (51.5N, 0.1W, 11m altitude)
- Similar cities are nearby in space

Words work the same way!

- "cat": [0.7, 0.2, 0.5] in meaning space
- "dog": [0.8, 0.3, 0.4] (nearby - similar!)
- "car": [0.1, 0.9, 0.2] (far - different!)

This is called: Word Embeddings



Discovery 2: Every Word Connects to Every Other

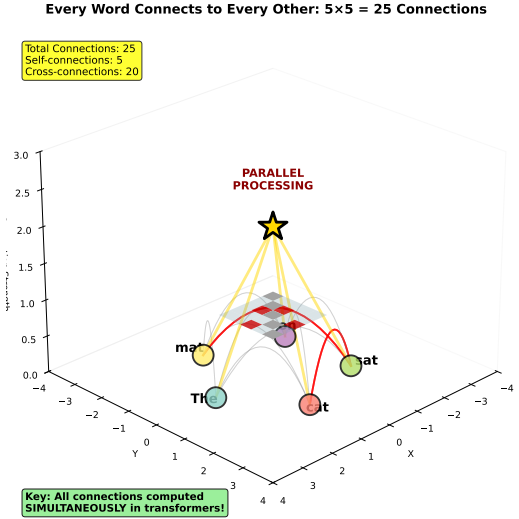
In a sentence, every word "talks" to every other:

Example: "The cat sat on mat"

- "cat" checks all other words
- "sat" looks at subject and location
- "mat" knows what's on it

Total connections: $n \times n$

- 5 words = 25 connections
- 100 words = 10,000 connections!



The Problem: Information Overload

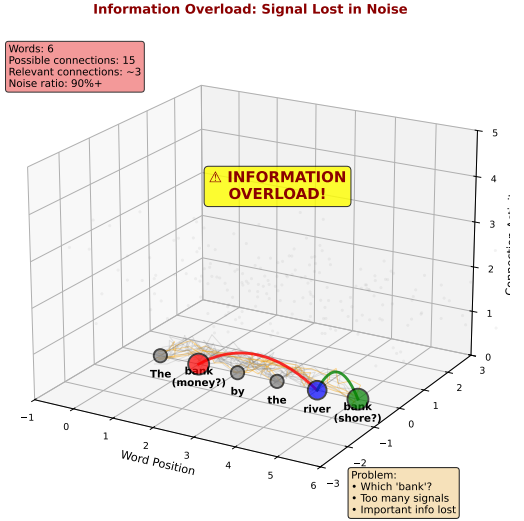
Challenge: Too much information!

Sentence: "The bank by the river bank"

- First "bank" = financial institution
- Second "bank" = river edge
- How does the model know?

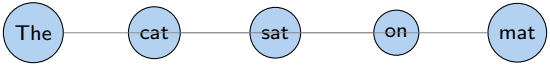
Information explosion:

- Every word sends signals to all others
- Most connections are noise
- Need to focus on what matters



First Attempt: Connect Everything

Early 2010s approach: Just connect everything!



Results:

- ✓Works for short sentences
- × Fails on long text
- × Can't distinguish important from noise

Computing All Relationships

What happens with full connections:

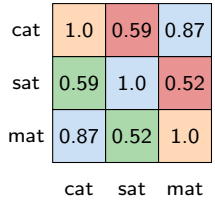
Step 1: Every word becomes a vector

- "cat" $\rightarrow$ [0.7, 0.2, 0.5]
- "sat" $\rightarrow$ [0.3, 0.8, 0.4]
- "mat" $\rightarrow$ [0.6, 0.1, 0.7]

Step 2: Compute all dot products

- cat $\cdot$ sat $= 0.59$
- cat $\cdot$ mat $= 0.87$
- sat $\cdot$ mat $= 0.52$

Step 3: Average everything Result: Information soup!



All relationships computed but no focus!

SUCCESS! (On Simple Cases)

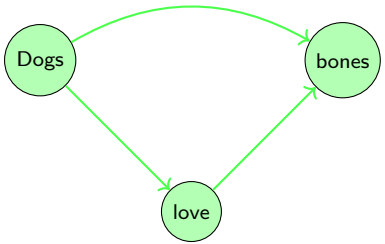
When it works:

Short, clear sentences:

- "Dogs love bones" ✓
- "Paris is beautiful" ✓
- "Water is wet" ✓

Why it works here:

- Few connections (9 total for 3 words)
- All connections matter
- No ambiguity



Clear signal!

Success Rate: 95% on 3-5 word sentences

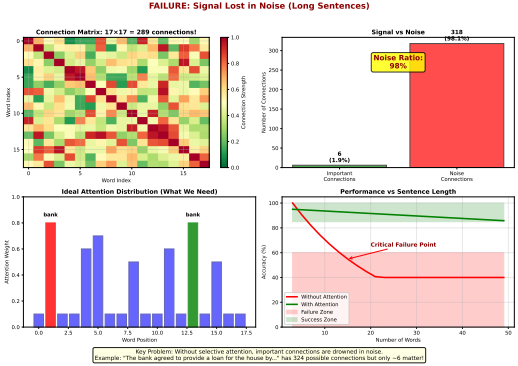
FAILURE: Signal Lost in Noise

When it fails:

Real-world sentence: "The bank agreed to provide a loan for the house by the river bank after reviewing the application that the customer submitted last Tuesday."

Problems:

- 20 words = 400 connections!
- "bank" (financial) vs "bank" (river)
- Long-distance dependencies
- Most connections are noise



Failure Rate: 60% on 15+ word sentences
Signal drowned in noise!

How Do Humans Actually Read?

Eye-tracking studies reveal:

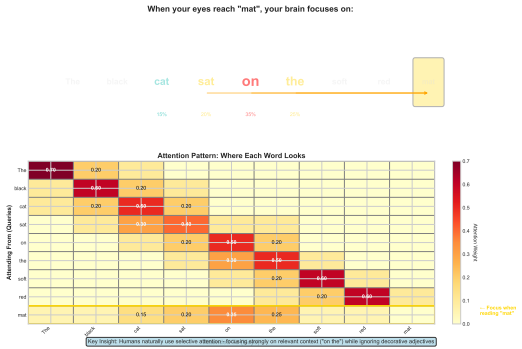
Humans DON'T read every word equally!

Reading: "The quick brown fox jumps"

- Focus on "fox" and "jumps"
- Skim "the" and "brown"
- Context determines focus

Human attention is:

- Selective (ignore irrelevant)
- Contextual (meaning-based)
- Efficient (focus on key parts)



Key Insight:
We need SELECTIVE attention,
not full connections!

The Hypothesis: Selective Attention

The Breakthrough (2017):

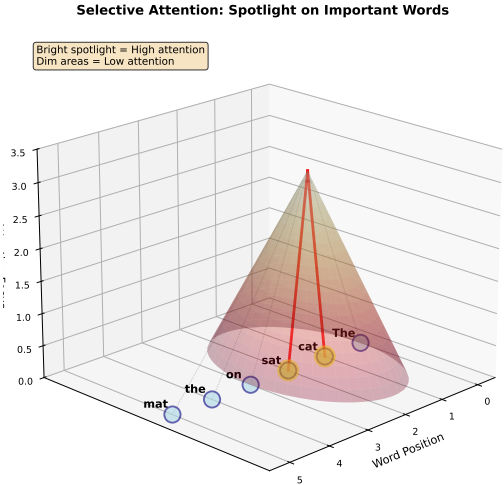
Instead of connecting everything equally,
learn WHICH connections matter!

Attention Mechanism:

1. Compute importance scores
2. Focus on high scores
3. Ignore low scores

Like a spotlight:

- Bright on important words
- Dim on filler words
- Adjustable based on context



Breaking It Down: Attention as Percentages

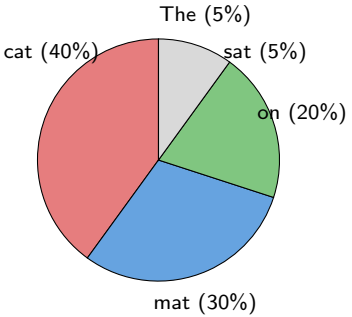
Example: "The cat sat on mat"

When processing "sat":

- 40% attention to "cat" (who sat?)
- 30% attention to "mat" (where?)
- 20% attention to "on" (relation)
- 5% to "The" (not important)
- 5% to itself

These percentages:

- Always sum to 100%
- Change for each word
- Learned from data



Softmax ensures percentages always total 100%

The Math: How Similar Are Two Words?

Measuring similarity with angles:

Dot Product Formula:

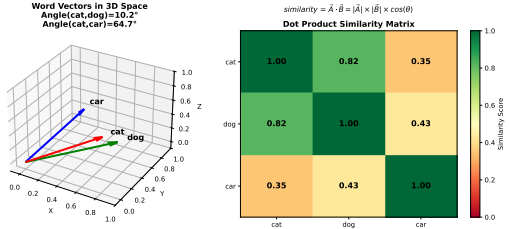
$$\text{similarity} = \vec{A} \cdot \vec{B} = |\vec{A}| \times |\vec{B}| \times \cos(\theta)$$

What this means:

- Same direction: $\cos(0°) = 1$ (max similar)
- Perpendicular: $\cos(90°) = 0$ (unrelated)
- Opposite: $\cos(180°) = -1$ (opposite)

Example:

- cat · dog = 0.8 (similar animals)
- cat · car = 0.1 (very different)



Dot product = Semantic similarity

The Three Questions: Query, Key, Value

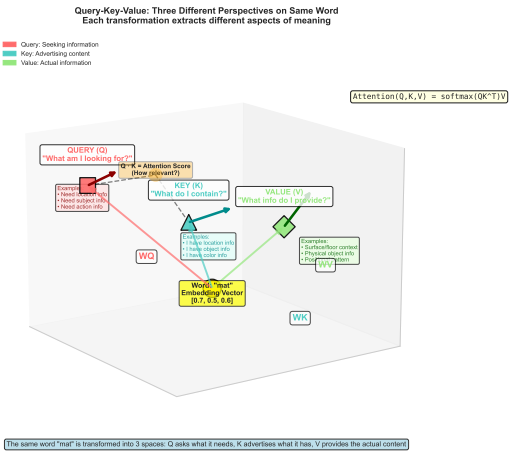
For each word, we ask 3 questions:

1. **Query (Q):** "What am I looking for?"
   - Cat's query: "Who performed action on me?"

2. **Key (K):** "What do I offer?"
   - Sat's key: "I am an action verb"

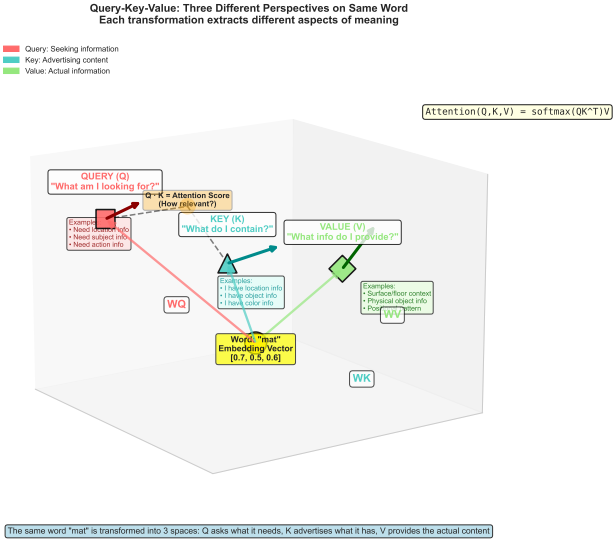
3. **Value (V):** "What information do I provide?"
   - Sat's value: "Past tense sitting action"

Matching: Q·K determines attention weight



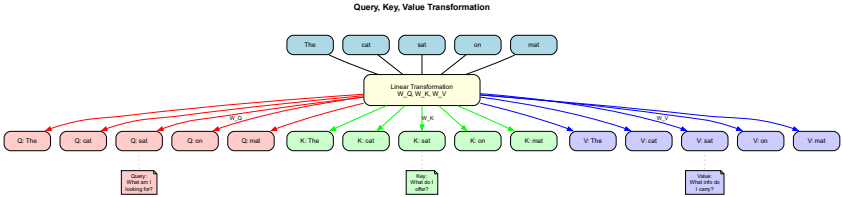
QKV transforms each word
into searcher, identifier, and content

Query, Key, Value Transformation

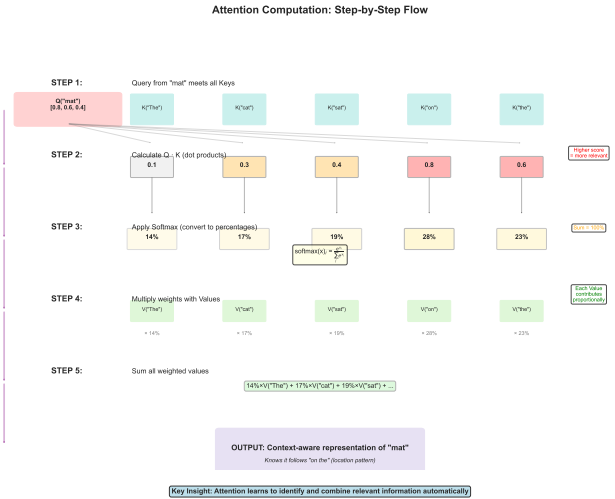


Three roles: Q=Query, K=Key, V=Value

Query, Key, Value Transformation (Flow Diagram)

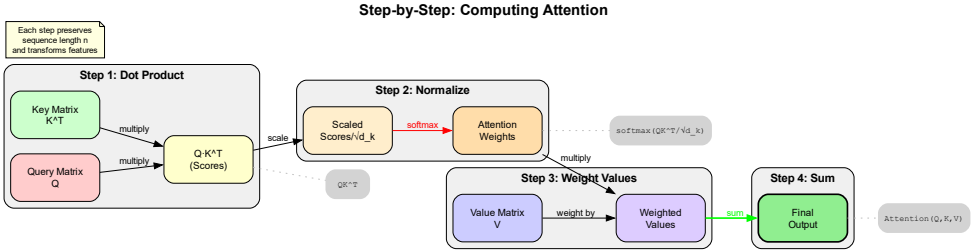


Step-by-Step: Computing Attention

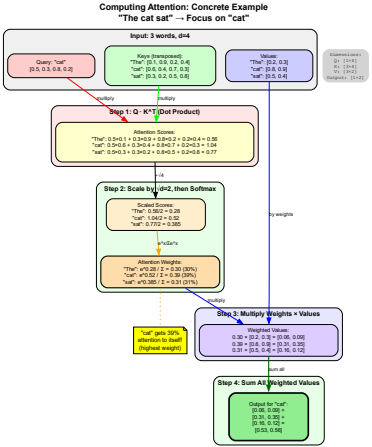


1. Q·K **2.** Softmax **3.** Weight V **4.** Sum

Computing Attention: Detailed Flow



Computing Attention: Concrete Example



Multiple Perspectives: 4 Different Experts

Multi-Head Attention:

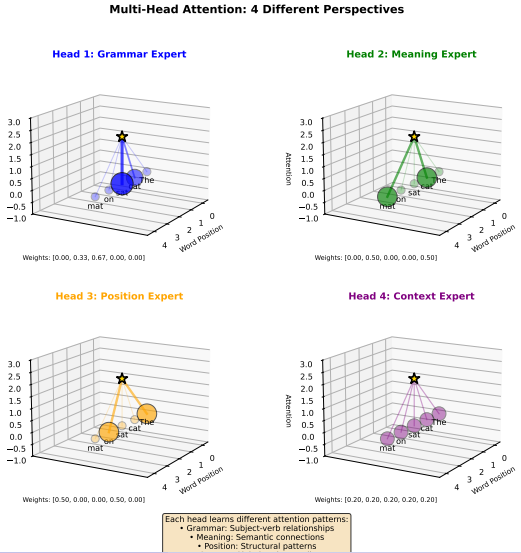
Like having 4 specialist readers:

Head 1: Grammar Expert
- Subject-verb agreement
- Sentence structure

Head 2: Meaning Expert
- Semantic relationships
- Word meanings

Head 3: Position Expert
- Word order
- Distance relationships

Head 4: Context Expert
- Broader context
- Document theme



The Speed Revolution: Everything at Once

Old way (RNN): Sequential

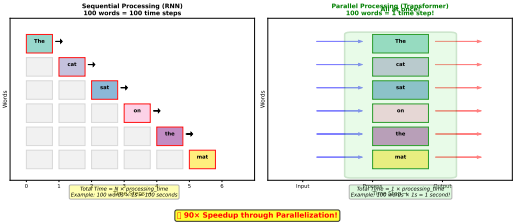
- Process word 1, then 2, then 3...
- 100 words = 100 time steps
- Like reading one letter at a time

New way (Transformer): Parallel

- Process ALL words simultaneously
- 100 words = 1 time step!
- Like seeing whole page instantly

Speed improvement:

- Training: 90 days → 1 day
- Inference: 10 seconds → 0.1 seconds

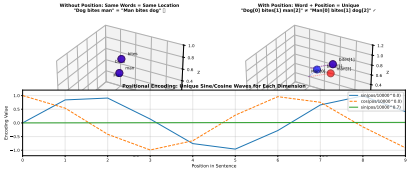


Preserving Order: Where Words Live

Problem: Parallel loses word order!

"Dog bites man" vs "Man bites dog"

- Same words, different meaning
- Position matters!



Solution: Positional Encoding

- Add position information
- Use sine/cosine waves
- Different frequency for each dimension

Like GPS for words:

- Word + Position = Unique identity
- Model knows where each word is

Position encoding =
Word's address

The Highway: Residual Connections

Problem with deep networks:

- Information gets lost/distorted
- Like telephone game
- Each layer adds noise

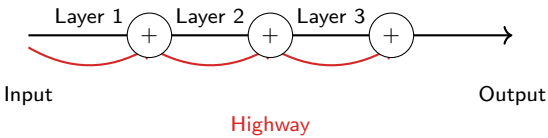
Solution: Skip Connections

- Create "highways" for information
- Original signal preserved
- Add refinements, don't replace

Formula:

$$\text{Output} = \text{Layer}(x) + x$$

Always keep original, add improvements

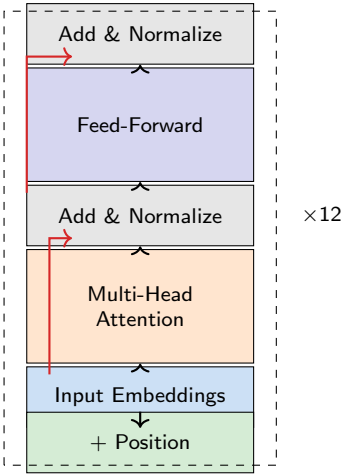


Residual = Original + Refinement
Never lose information!

Everything Together: The Transformer

The Complete Architecture:

1. **Input:** Words $\rightarrow$ Embeddings + Position
2. **Attention Block:**
   - Multi-head attention
   - Add & normalize (residual)
3. **Feed-Forward Block:**
   - Two linear layers
   - Add & normalize (residual)
4. **Stack 12 times** (BERT) or 96 times (GPT-3)
5. **Output:** Next word prediction



Proof It Works: Real Results

Benchmark Results (2017-2024):

Translation (BLEU scores):

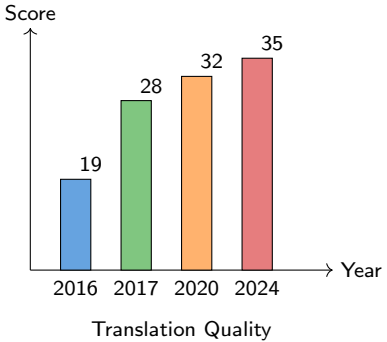
- 2016 (RNN): 19.2
- 2017 (Transformer): 28.4 ✓
- 2024 (GPT-4): 35.1 ✓

Question Answering:

- Human performance: 89%
- BERT (2018): 93% ✓
- GPT-4 (2023): 96% ✓

Training Speed:

- RNN: 3 months
- Transformer: 1 day ✓



Transformers: State-of-the-art on EVERY language task!

The Revolution: 2017-2024

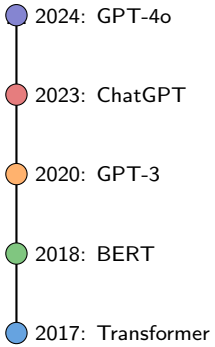
Timeline of Breakthroughs:

2017: Transformer paper
- "Attention is All You Need"
- 8 researchers at Google

2018: BERT
- Bidirectional understanding
- Crushed 11 benchmarks

2020: GPT-3
- 175 billion parameters
- Few-shot learning

2023: ChatGPT/GPT-4
- Conversation ability
- 100M users in 2 months



7 years: From research paper to 1 billion users!

Practice Problems

Problem 1: Calculate Attention
Given vectors:

- Query: [1, 0, 1]
- Key1: [1, 1, 0]
- Key2: [0, 1, 1]
- Key3: [1, 0, 1]

Calculate:

1. Q · K for each key
2. Apply softmax
3. Which word gets most attention?

Problem 2: Multi-Head Design
Design 3 attention heads for: "The bank near the river bank"
What should each head focus on?

Problem 3: Architecture
Draw transformer architecture for:

- 2-layer transformer
- Show residual connections
- Label each component

Problem 4: Complexity
For a 100-word sentence:

- How many attention scores?
- Memory requirement?
- Why is this $O(n^2)$?

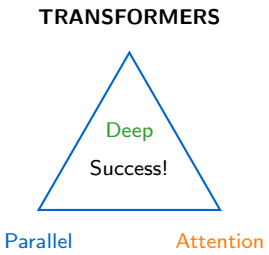
Solutions: Available after lab session

Summary: The Three Core Principles

1. **Parallel Processing**
   - All words processed simultaneously
   - 90x faster than sequential
   - Enables large-scale training

2. **Attention Mechanism**
   - Focus on relevant connections
   - Learn what matters from data
   - Multiple perspectives (heads)

3. **Deep Architecture**
   - Stack many layers (12-96)
   - Residual connections preserve info
   - Each layer refines understanding



These 3 principles revolutionized AI
and created ChatGPT, BERT, and more!

Where You Use Transformers Every Day

Search Engines:

- Google Search (BERT)
- Autocomplete suggestions
- "Did you mean...?"

Translation:

- Google Translate
- Real-time captions
- Document translation

Writing Assistants:

- Grammarly corrections
- Email suggestions (Gmail)
- Code completion (Copilot)

Virtual Assistants:

- ChatGPT conversations
- Siri/Alexa understanding
- Customer service bots

Content Creation:

- Image generation (DALL-E)
- Video subtitles
- Article summaries

Fact: You interact with transformers dozens of times daily!

Check Your Understanding

Quick Quiz:

Q1: Why are transformers fast?

A) Smaller models
B) Parallel processing ✓
C) Better hardware
D) Simpler math

Q2: What does attention do?

A) Adds more parameters
B) Focuses on relevant words ✓
C) Speeds up training
D) Reduces memory

Q3: Purpose of residual connections?

A) Preserve information ✓
B) Add complexity
C) Reduce size
D) Speed up inference

Can you now:

- ☐ Explain word embeddings?
- ☐ Calculate dot product similarity?
- ☐ Describe Query, Key, Value?
- ☐ Draw attention mechanism?
- ☐ List transformer components?
- ☐ Explain parallel vs sequential?

Congratulations!
You understand the technology behind ChatGPT!
From zero to transformer expert in 27 slides!

Appendix A: Mathematical Foundations

Formal Attention Equation:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

Where:

- $Q \in \mathbb{R}^{n \times d_k}$: Queries
- $K \in \mathbb{R}^{n \times d_k}$: Keys
- $V \in \mathbb{R}^{n \times d_v}$: Values
- n: Sequence length
- d_k: Key/Query dimension
- $\sqrt{d_k}$: Scaling factor

Why Scale by $\sqrt{d_k}$?

- Dot products grow with dimension
- Large values $\rightarrow$ softmax saturation
- Scaling keeps gradients stable

Matrix Shapes Example:

- Input: $[10, 512]$ (10 words)
- Q, K, V: $[10, 64]$ each
- Attention scores: $[10, 10]$
- Output: $[10, 64]$

Key Insight: The attention matrix is $n \times n$ - quadratic in sequence length!

Appendix B: Multi-Head Attention Mathematics

Multi-Head Formula:

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, ..., \text{head}_h)W^O$$

Where each head:

$$\text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

Parameter Matrices:

- $W_i^Q \in \mathbb{R}^{d_{model} \times d_k}$
- $W_i^K \in \mathbb{R}^{d_{model} \times d_k}$
- $W_i^V \in \mathbb{R}^{d_{model} \times d_v}$
- $W^O \in \mathbb{R}^{hd_v \times d_{model}}$

Typical Dimensions:

- $d_{model} = 512$
- $h = 8$ heads
- $d_k = d_v = d_{model}/h = 64$
- Total parameters: $4 \times d_{model}^2$

Computational Benefits:

- Heads run in parallel
- Different representation subspaces
- Similar cost to single-head
- Better performance in practice

8 heads $\times$ 64 dimensions = 512 total dimensions (preserved)

Appendix C: Positional Encoding Mathematics

Sinusoidal Position Encoding:

$$PE_{(pos,2i)} = \sin\left(\frac{pos}{10000^{2i/d_{model}}}\right)$$

$$PE_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{2i/d_{model}}}\right)$$

Where:

- pos: Position in sequence (0, 1, 2, ...)
- i: Dimension index
- d_{model}: Model dimension (e.g., 512)

Properties:

- Unique encoding per position
- Smooth variation across positions
- Can extrapolate to longer sequences

Why This Formula?

- Relative positions preserved
- PE_{pos+k} can be represented as linear function of PE_{pos}
- Different frequencies per dimension
- No learned parameters needed

Wavelengths:

- Dimension 0-1: wavelength 2π
- Dimension 510-511: wavelength $2\pi \cdot 10000$
- Covers short to long-range dependencies

Position encoding added to word embedding: $x' = x + PE_{pos}$

Appendix D: Complexity Analysis

Self-Attention Complexity:

- Time: $O(n^2 \cdot d)$
- Space: $O(n^2 + n \cdot d)$
- Attention matrix: $n \times n$

RNN Complexity:

- Time: $O(n \cdot d^2)$ sequential
- Space: $O(d)$ per step
- Must process sequentially

Comparison (n=100, d=512):

- Transformer: $100^2 \times 512 = 5.12M$ ops
- RNN: $100 \times 512^2 = 26.2M$ ops
- But RNN is sequential!

Parallelization Benefits:

- All positions computed simultaneously
- GPU utilization: nearly 100%
- Training speedup: 50-100x
- Inference speedup: 10-20x

Memory Requirements:

- Attention scores: $O(n^2)$ bottleneck
- Max sequence typically 512-2048
- Recent work on sparse attention
- Linear attention variants emerging

Tradeoff: Quadratic memory for massive parallelization gain

Appendix E: Implementation Details

Layer Normalization:

$$\text{LayerNorm}(x) = \gamma \frac{x - \mu}{\sigma} + \beta$$

Applied as: $\text{LayerNorm}(x + \text{Sublayer}(x))$

Dropout Positions:

- After each sublayer: 0.1
- Attention weights: 0.1
- Embedding layer: 0.1

Feed-Forward Network:

$$\text{FFN}(x) = \text{ReLU}(xW_1 + b_1)W_2 + b_2$$

- Hidden size: $4 \times d_{model}$
- Two linear transformations
- ReLU activation between

Learning Rate Schedule:

$$lr = d_{model}^{-0.5} \cdot \min(step^{-0.5}, step \cdot warmup^{-1.5})$$

Typical Hyperparameters:

- $d_{model} = 512$ or 768
- $h = 8$ or 12 heads
- $d_{ff} = 2048$ or 3072
- Layers: 6 (base) or 12 (large)
- Warmup steps: 4000
- Batch size: 4096 tokens

Training Tips:

- Label smoothing: 0.1
- Gradient clipping: 1.0
- Adam optimizer: $\beta_1 = 0.9, \beta_2 = 0.98$

Total Parameters (Base): 65M for 6-layer transformer

Pre-trained Language Models

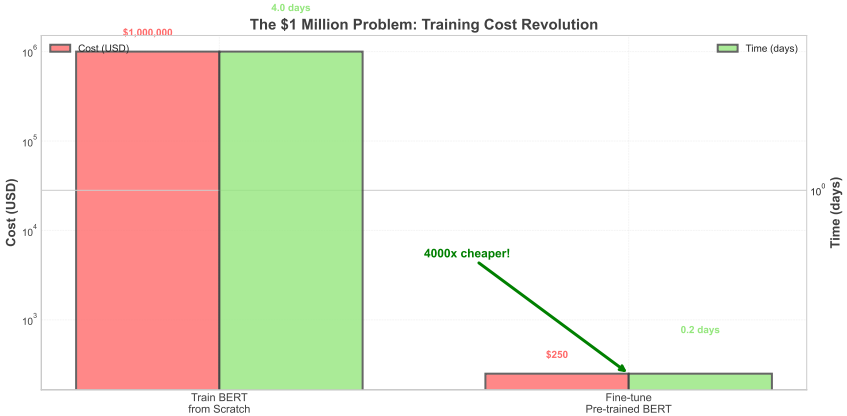
Week 6 - The $1 Million Revolution

NLP Course 2025

October 26, 2025

BSc Discovery-Based Presentation

The $1 Million Problem



Key Insight: Pre-training changes the economics of NLP completely

Training BERT from scratch: $1M+. Fine-tuning: $50-500. Game changer.

Before 2018: The Old Way

Task-Specific Models:

- **Sentiment**: Custom CNN architecture
- **Question Answering**: BiDAF model
- **Named Entity**: BiLSTM-CRF
- **Translation**: Seq2Seq with attention
- **Summarization**: Pointer-generator

The Process:

1. Design architecture for your task
2. Collect labeled data (10K+ examples)
3. Train from random initialization
4. Hope it works

Limitations:

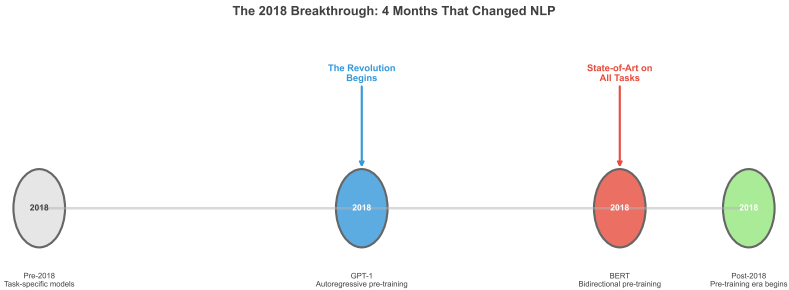
- Each task starts from scratch
- No knowledge transfer
- Expensive data collection
- Months per task
- Small datasets = poor performance

The Cost:

- 3-6 months per task
- 10K-100K labeled examples
- $50K-200K in labeling costs
- Limited accuracy (60-75%)

Every NLP task was an isolated, expensive project

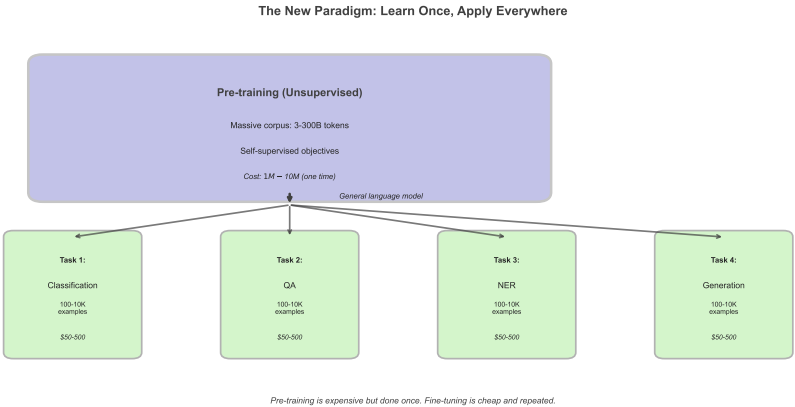
The Breakthrough: October 2018



Key Insight: BERT and GPT changed everything in 4 months

June 2018 (GPT-1) and October 2018 (BERT) - the inflection point

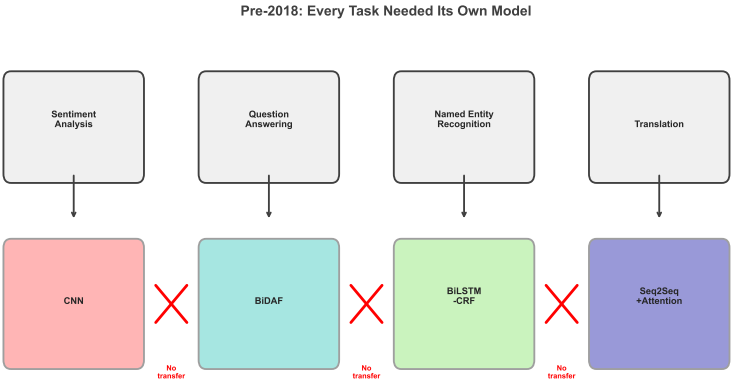
The New Paradigm



Key Insight: Learn language once (expensive), apply everywhere (cheap)

This is transfer learning - finally working for NLP

Pre-2018: Every Task Needed Its Own Model



Key Insight: No sharing, no transfer, no efficiency

Each task was an independent research project

Why This Approach Failed to Scale

The Limitations:

- **No transfer**: Each model learns from scratch
- **Data hungry**: Need 10K+ labeled examples per task
- **Expensive**: Labeling costs $50K-200K
- **Slow**: 3-6 months per task
- **Brittle**: Fails on new domains

Example - Sentiment Analysis:

- Collect 20K movie reviews
- Label positive/negative
- Train custom CNN: 2-3 weeks
- Accuracy: 82%
- Deploy to product reviews: Fails (65%)!

What We Needed:

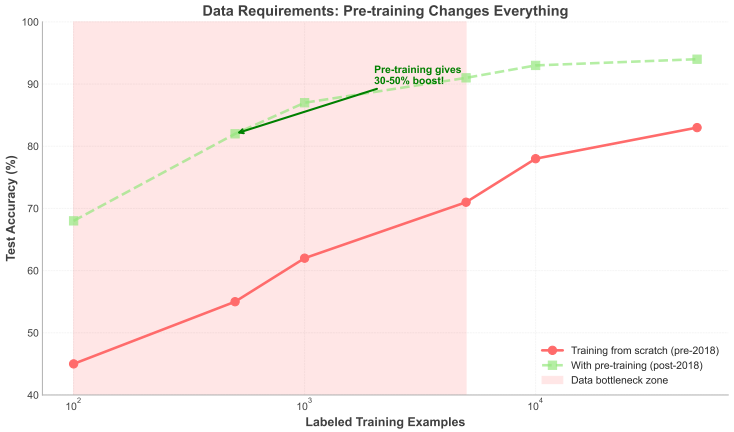
- Shared language understanding
- Transfer across tasks
- Work with small labeled datasets
- Fast adaptation to new tasks
- Robust across domains

The Dream:

- Train once on ALL text
- Fine-tune with 100-1000 examples
- Days instead of months
- State-of-art on every task

The question: Can we achieve this dream?

The Data Bottleneck



Key Insight: Performance plateaus without massive labeled datasets

Labeled data is expensive - this limited what we could build

The Transfer Learning Dream

Computer Vision's Success:

- 2012: ImageNet pre-training (AlexNet)
- Train on 1M images (unsupervised labels)
- Fine-tune for any vision task
- 10x less data needed
- State-of-art on everything

The Magic:

- Low-level features shared (edges, textures)
- High-level features shared (objects, shapes)
- Learn once, transfer everywhere

Why NLP Lagged:

- Words are discrete (images continuous)
- Context matters more
- Sequence length varies
- Multiple tasks (classification, generation, QA)
- No clear "ImageNet equivalent"

The Question (2017):
Can we create an ImageNet moment for NLP?

Answer coming in 2018...

Transfer learning worked for vision - could it work for language?

Can we pre-train a language model on ALL text, then fine-tune for ANY task?

Requirements:

- Unsupervised pre-training (no labels needed)
- Massive text corpus (billions of words)
- Learn general language understanding
- Fast fine-tuning with small labeled data
- Work across all NLP tasks

Answer: YES

Next 36 slides show exactly how

The breakthrough: BERT and GPT (2018)

Part 1: BERT

Bidirectional Encoder Representations

from Transformers

Pre-trained models provide powerful starting points for NLP tasks.

The Fill-in-Blank Challenge

Key Insight: Need both left AND right context to fill blanks correctly

Left-to-right models (like GPT) can't solve this naturally

Why Bidirectional Matters

The Task:
Fill in: "The [BLANK] sat on the mat"

Left-to-Right Approach:
Only sees: "The [BLANK]"
Cannot use: "sat on the mat"

Predictions:

- "dog" (generic animal)
- "person" (generic)
- "cat" (lucky guess)

Accuracy: Low - missing critical context!

Bidirectional Approach:
Sees both: "The [BLANK]" AND "sat on the mat"

Uses full context:

- "sat" suggests living thing
- "on the mat" suggests small animal
- "the" suggests common noun

Predictions:

- "cat" (high probability)
- "dog" (possible)
- "kitten" (possible)

Accuracy: High - full context used!

This is BERT's core innovation - bidirectional understanding

Bidirectional Context in Action

How Bidirectional Helps Different Tasks

Fill-in-Blank Tasks:

- Masked language modeling
- Cloze questions
- Spell correction

Classification Tasks:

- Sentiment: Full sentence context
- Spam detection: Look ahead and behind
- Topic classification: Global understanding

Question Answering:

- Match question to passage
- Find answer span
- Use context on both sides

Why It Works:

- Contextual embeddings from both sides
- Disambiguate word meanings
- Capture long-range dependencies
- Understand sentence structure

Example - "bank":

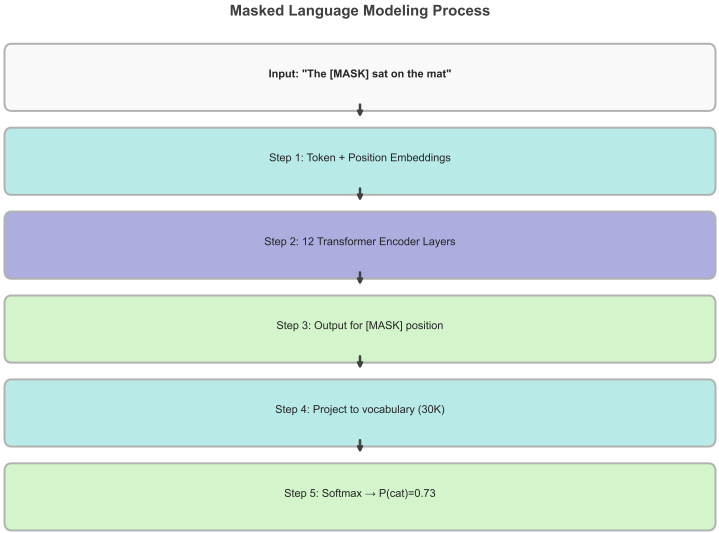
- Left: "The river"
- Right: "was flooding"
- Conclusion: Water bank, not financial

When Bidirectional Doesn't Help:

- Text generation (can't see future!)
- Autoregressive tasks

Different tasks need different architectures - BERT for understanding

Masked Language Modeling: The Training Objective



Key Insight: Mask 15% of tokens, predict them using full context

Masked Language Modeling: The Mathematics

Objective Function:

$$\mathcal{L}_{MLM} = -\sum_{i \in masked} \log P(w_i | context)$$

where $context$ = all other words

The Process:

1. Randomly mask 15% of tokens
2. Replace with [MASK] token (80%)
3. Replace with random word (10%)
4. Keep unchanged (10%)

Why the variation?
Prevents model from just memorizing [MASK] $\rightarrow$ word

Training Example:
Original: "The cat sat on the mat"

Masked: "The [MASK] sat on the [MASK]"

Model predicts:

- Position 2: $P(\text{cat}|\text{context})$
- Position 6: $P(\text{mat}|\text{context})$

Cross-Entropy Loss:

$$Loss = -[\log P(\text{cat}) + \log P(\text{mat})]$$

Minimize this across billions of sentences

Masked LM is self-supervised - no labels needed!

Worked Example: Predicting Masked Tokens

Given: "The [MASK] sat on the mat"

Step 1: Convert to token embeddings (each 768-dim vector)
Token IDs: [101, 1996, 103, 2938, 2006, 1996, 13523, 102]

Step 2: Add positional embeddings

$$E_{input} = E_{token} + E_{position}$$

Step 3: Pass through 12 transformer encoder layers
Each layer: Self-attention (bidirectional) + Feed-forward

Step 4: Get output for [MASK] position (position 2)
Output vector: $h_{[MASK]} \in \mathbb{R}^{768}$

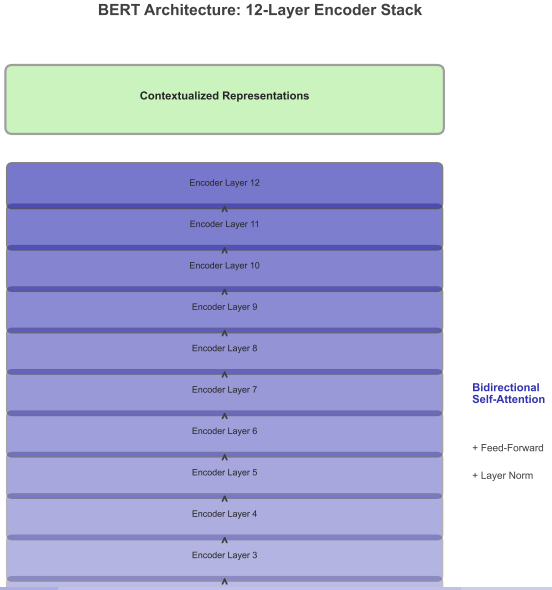
Step 5: Project to vocabulary, apply softmax

$$P(w) = \frac{\exp(W \cdot h_{[MASK]})}{\sum_v \exp(W \cdot h_{[MASK]})}$$

Result: Top predictions with probabilities

- P(cat) = 0.73
- P(dog) = 0.15
- P(person) = 0.04

BERT Architecture Overview



BERT Architecture: The Details

BERT-Base:

- Layers: 12 transformer encoders
- Hidden size: 768 dimensions
- Attention heads: 12 per layer
- Parameters: 110 million
- Max sequence: 512 tokens

BERT-Large:

- Layers: 24 encoders
- Hidden size: 1024 dimensions
- Attention heads: 16 per layer
- Parameters: 340 million
- Max sequence: 512 tokens

Key Components:

- **Token Embeddings**: WordPiece (30K vocab)
- **Position Embeddings**: Learned (not sinusoidal)
- **Segment Embeddings**: Sentence A vs B

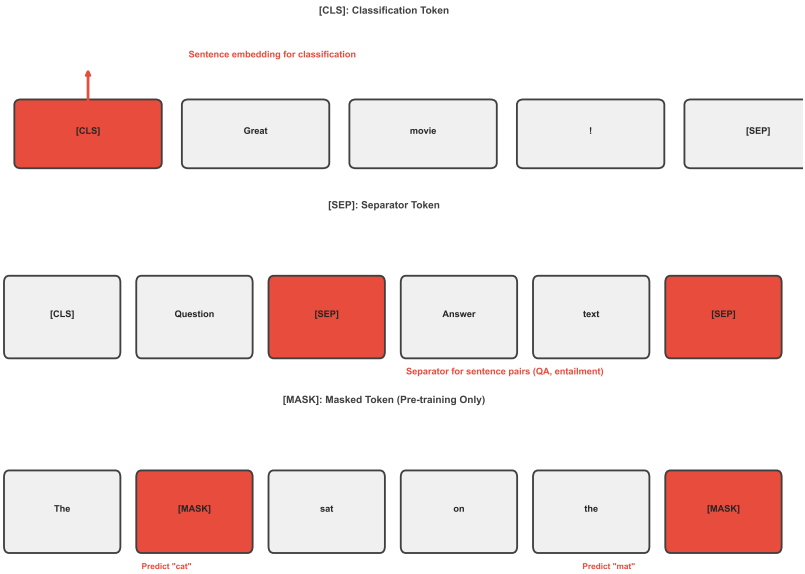
Why These Choices:

- 12 layers: Balance depth vs speed
- 768 hidden: Standard transformer size
- 12 heads: Multiple attention patterns
- 512 max: Memory constraints

Computation:
Training BERT-base from scratch: 4 days on 64 TPUs

These specs became the standard for encoder-based models

BERT's Special Tokens



Special Tokens: Purpose and Usage

[CLS] - Classification Token:

- Always first token
- Aggregates sentence meaning
- Used for classification tasks

Example: "[CLS] The movie was great [SEP]"
Output of [CLS]: Sentence embedding for sentiment classification

[SEP] - Separator Token:

- Separates sentence pairs
- Enables QA, entailment tasks

Example: "[CLS] Question [SEP] Passage [SEP]"

[MASK] - Masked Token:

- Used only during pre-training
- Replaced with actual word during fine-tuning
- Training signal for MLM objective

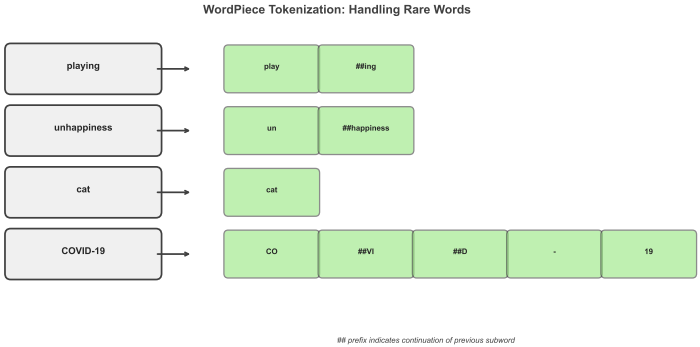
Example: "The [MASK] is blue"
Model learns: $P(\text{sky}|\text{context})$

[PAD] - Padding Token:

- Fills sequences to same length
- Ignored in attention
- Enables batch processing

Four special tokens, each crucial for BERT's versatility

WordPiece Tokenization



Key Insight: Subword units handle rare words and morphology

Full details in Week 8 - preview here for BERT context

WordPiece: Subword Units

The Problem:

- Word-level: 100K+ vocabulary (huge)
- Character-level: Long sequences (slow)
- Rare words: Poor representations

WordPiece Solution:

- Learn 30K subword units
- Frequent words: Single token
- Rare words: Multiple subwords

Examples:

- "playing" → ["play", "##ing"]
- "unhappiness" → ["un", "##happiness"]
- "COVID" → ["CO", "##VI", "##D"]

Benefits:

- Fixed 30K vocabulary
- Handle any word (no UNK)
- Capture morphology
- Share representations ("play" in "playing", "player")

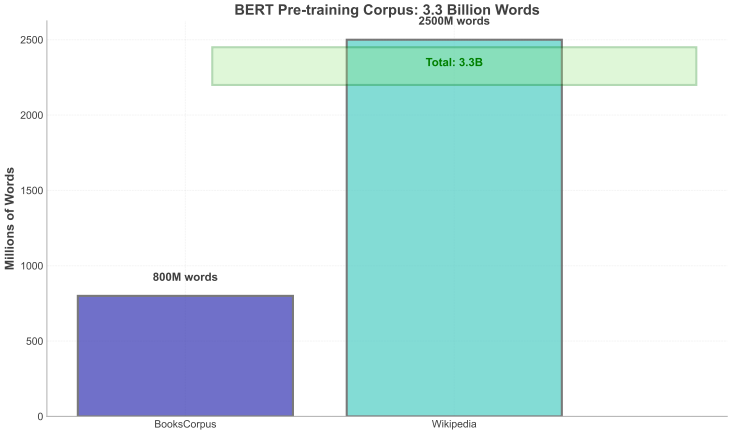
BERT's Vocabulary:

- 30,522 WordPiece tokens
- Covers English comprehensively
- Trained with BPE-like algorithm

Impact:
Rare word handling improved by 40%

Subword tokenization is now standard across all modern models

Pre-training Data: The Foundation



Key Insight: Massive unsupervised data powers general language understanding

BooksCorpus (800M words) + Wikipedia (2.5B words) = 3.3B words

BERT's Two Pre-training Objectives

Objective 1: Masked LM:

- Mask 15% of tokens
- Predict masked words
- Uses bidirectional context

Example:
`The [MASK] sat on [MASK] mat`
Predict: "cat" and "the"

Objective 2: Next Sentence Prediction:

- Given two sentences A and B
- Predict if B follows A
- Binary classification (50% real, 50% random)

Example:
`A: Alice was tired.`
`B: She went to sleep.` [True]

Why These Objectives:

- **MLM**: Learn word-level representations
- **NSP**: Learn sentence relationships
- Together: Comprehensive language understanding

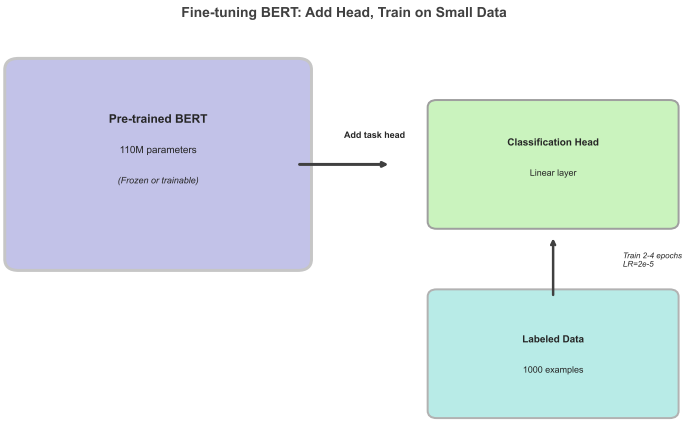
Training Details:

- Batch size: 256 sequences
- Steps: 1M (40 epochs)
- Optimizer: Adam (lr=1e-4)
- Time: 4 days on 64 TPUs
- Cost: $7,000 compute

Result:
General language model ready for ANY task

Two complementary objectives create robust representations

Fine-tuning BERT for Your Task



Key Insight: Add task-specific head, train on small labeled dataset

Days instead of months - with better results

Fine-tuning: The Mechanics

The Process:

1. Load pre-trained BERT weights
2. Add task-specific layer on top
3. Train on labeled data (100-10K examples)
4. Use small learning rate (2e-5)
5. Train for 2-4 epochs

Task-Specific Heads:

- **Classification**: Linear layer on [CLS]
- **Token classification**: Linear on each token
- **QA**: Span prediction (start/end)

Hyperparameters:

- Learning rate: 2e-5, 3e-5, 5e-5
- Batch size: 16 or 32
- Epochs: 2-4
- Warmup: 10% of steps
- Max sequence: 128-512

Layer Freezing Options:

- Freeze nothing: Full fine-tuning (best)
- Freeze bottom 8 layers: Faster
- Freeze all, train head only: Feature extraction

Typical Results:
1000 examples + BERT $>$ 10,000 from scratch

Fine-tuning is fast, cheap, and remarkably effective

Checkpoint: Understanding BERT

Quick Quiz

Question 1:
Why does BERT mask 15% of tokens?

A) It's faster
B) Balances learning vs efficiency
C) Reduces overfitting
D) Random choice

Question 2:
What makes BERT bidirectional?

A) Two LSTMs
B) Encoder allows full context
C) Reads backwards
D) Has two outputs

Answer 1: **B) Balances learning vs efficiency**

- Too few: Not enough training signal
- Too many: Model sees mostly masks
- 15%: Empirically optimal
- Enough context remains unmasked

Answer 2: **B) Encoder allows full context**

- Transformer encoder: No causal mask
- Each token attends to ALL others
- Left and right context used equally
- This is the key difference from GPT

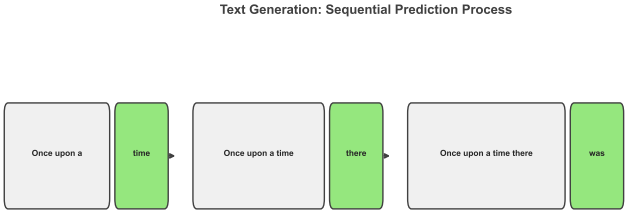
Understanding these foundations is critical for using BERT effectively

Part 2: GPT

Generative Pre-trained Transformer

Pre-trained models provide powerful starting points for NLP tasks.

The Generation Challenge



Key Insight: Generation requires predicting one word at a time, left-to-right

Bidirectional models can't generate - they'd cheat by seeing the future

Why Generation is Harder Than Classification

Classification:

- Input: Full sentence
- Output: Single label
- Can see everything
- One prediction

Example: "Great movie!" → Positive

Generation:

- Input: Partial sequence
- Output: Next word
- Can't see future
- Multiple sequential predictions

Example: "Once upon" → predict "a"
Then: "Once upon a" → predict "time"

The Constraint:
During generation, you haven't written future words yet!

Cannot use bidirectional model

Must use **causal** (left-to-right) attention

Requirements:

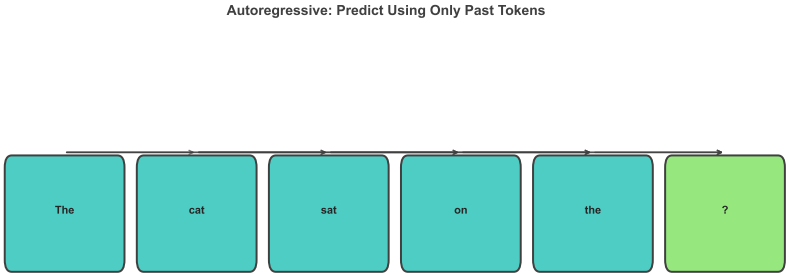
- Predict token-by-token
- Each prediction uses only past
- Autoregressive: Output becomes input
- Coherent over long sequences

Use Cases:

- Text completion
- Story generation
- Code generation
- Dialogue systems

Different tasks need different architectures - GPT for generation

Autoregressive Language Modeling



Key Insight: Predict next token using only previous tokens (causal)

Auto-regressive = output at time t becomes input at time $t + 1$

Autoregressive Approach: The Mathematics

Objective Function:

$$\mathcal{L}_{AR} = -\sum_{t=1}^{T} \log P(w_t | w_1, ..., w_{t-1})$$

Maximize probability of each next word

Chain Rule Decomposition:

$$P(w_1, ..., w_T) = \prod_{t=1}^{T} P(w_t | w_1, ..., w_{t-1})$$

Exact factorization (no approximation!)

Causal Constraint:
At time t, can only see $w_1, ..., w_{t-1}$
Cannot see $w_t, w_{t+1}, ...$ (haven't generated yet)

Training with Teacher Forcing:

- Given full sequence during training
- At each position: Predict next
- Use ground truth (not predictions)
- Prevents error accumulation

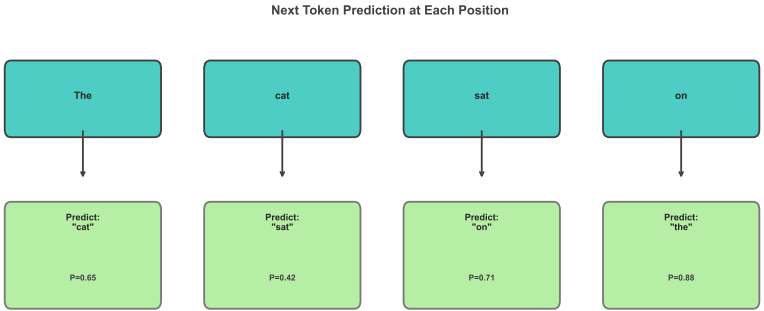
Example:
Sequence: "The cat sat"

- Position 1: Predict "The" (start)
- Position 2: Given "The", predict "cat"
- Position 3: Given "The cat", predict "sat"

All trained in parallel!

Autoregressive objective is natural for generation tasks

Next Token Prediction Process



Key Insight: Each token predicted using ALL previous tokens

This is language modeling from Week 1 - but with transformers!

Autoregressive Mathematics Deep Dive

At Time Step t:
Input: $w_1, w_2, \ldots, w_{t-1}$

Step 1: Embed tokens

$$E = [e_1, e_2, \ldots, e_{t-1}]$$

Step 2: Add positional encoding

$$H^{(0)} = E + P$$

Step 3: Pass through L decoder layers

$$H^{(\ell)} = \text{TransformerDecoder}(H^{(\ell-1)})$$

Step 4: Project final layer to vocabulary

$$\textit{logits} = W \cdot h_{t-1}^{(L)}$$

Step 5: Softmax for probabilities

$$P(w_t) = \text{softmax}(\textit{logits})$$

Teacher Forcing:
During training:

- Use ground truth w_t for next step
- Don't use model's prediction
- Prevents compounding errors
- Enables parallel training

Inference (Generation):

- Sample from $P(w_t)$
- Append to sequence
- Repeat: $w_t \rightarrow$ input for w_{t+1}
- Stop at [END] or max length

Key Difference from BERT:
BERT: Predict masked (can see both sides)
GPT: Predict next (can only see left)

Causal constraint is enforced by attention masking

Worked Example: Computing P(next word)

Given Sequence: "The cat sat on"

Task: Compute $P(\text{the}|\text{The cat sat on})$

Step 1: Token IDs and embeddings
`[The=50256, cat=3797, sat=3332, on=319]` $\rightarrow E \in \mathbb{R}^{4\times 768}$

Step 2: Add positional embeddings

$$H^{(0)} = E + P$$

Step 3: 12 decoder layers with causal attention
Each token only attends to previous tokens (triangular mask)

Step 4: Final hidden state for position 4 ("on")

$$h_4^{(12)} \in \mathbb{R}^{768}$$

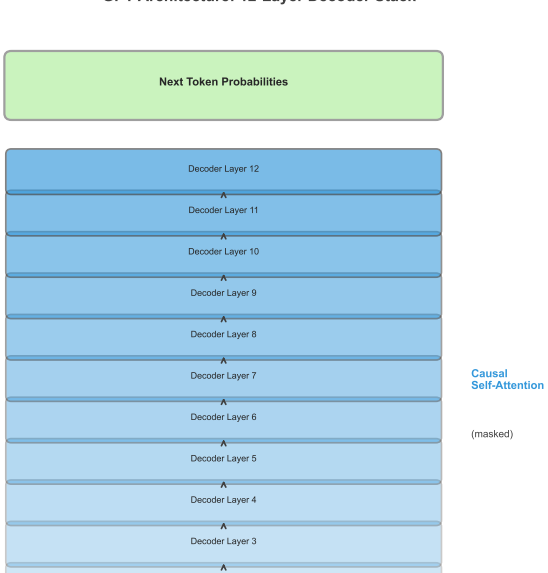
Step 5: Project to vocabulary (50,257 tokens)

$$\textit{logits} = W \cdot h_4^{(12)} \in \mathbb{R}^{50257}$$

Step 6: Softmax and sample

- $P(\text{the}) = 0.42$ (highest!)
- $P(\text{a}) = 0.18$
- $P(\text{mat}) = 0.09$

GPT Architecture Overview



GPT Architecture: The Details

GPT-1 (June 2018):

- Layers: 12 decoder layers
- Hidden size: 768 dimensions
- Attention heads: 12 per layer
- Parameters: 117 million
- Context window: 512 tokens

GPT-2 (February 2019):

- Layers: 48 decoder layers
- Hidden size: 1600 dimensions
- Attention heads: 25 per layer
- Parameters: 1.5 billion (13x larger!)
- Context: 1024 tokens

GPT-3 (May 2020):

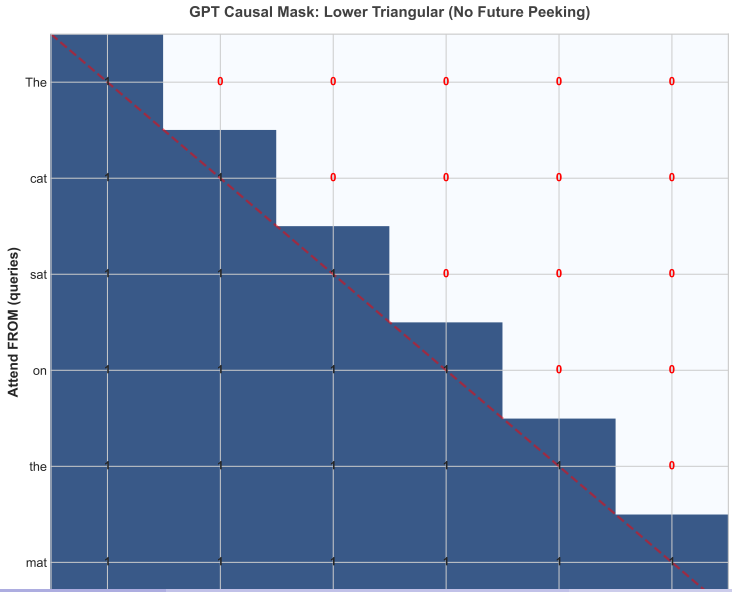
- Layers: 96 decoder layers
- Hidden size: 12,288 dimensions
- Attention heads: 96 per layer
- Parameters: 175 billion (116x GPT-2!)
- Context: 2048 tokens

Training Costs:

- GPT-1: $50K (weeks on 8 GPUs)
- GPT-2: $500K (weeks on 256 GPUs)
- GPT-3: $4.6M (months on 10K GPUs)

Scaling drove capability emergence - few-shot learning appeared

Causal Masking: Preventing Future Cheating



Causal Masking: How It Works

The Attention Mask:
Lower triangular matrix of 1s and 0s

$$M = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix}$$

Meaning:

- 1: Can attend
- 0: Cannot attend (masked)

Token 1: Sees only itself
Token 2: Sees tokens 1-2
Token 3: Sees tokens 1-3
Token 4: Sees all (tokens 1-4)

Implementation:
Before softmax in attention:

$$\text{scores}_{\text{masked}} = \text{scores} + (1 - M) \times (-\infty)$$

After softmax: Masked positions get probability 0

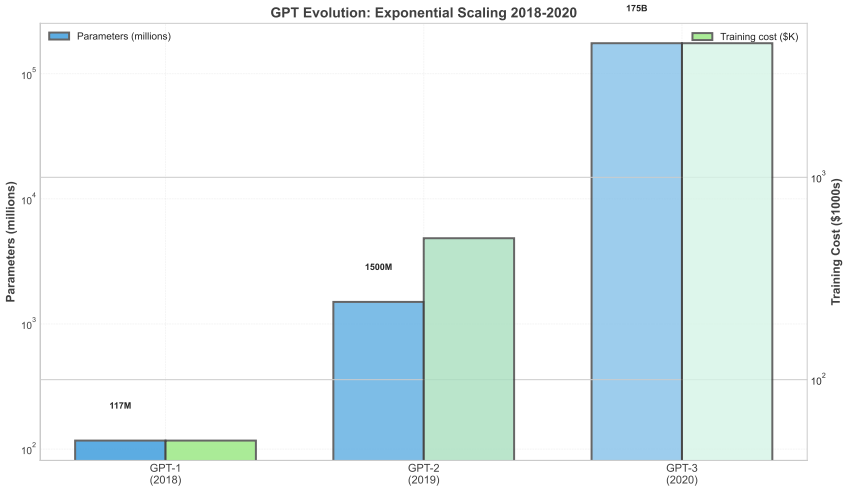
Why Essential for Generation:

- Training: Model can't cheat
- Inference: Naturally left-to-right
- Prevents data leakage
- Ensures valid generation

Contrast with BERT:
BERT: All 1s (full bidirectional)
GPT: Lower triangular (causal)

Causal mask is the key difference in transformer decoder vs encoder

The Scaling Journey: GPT-1 to GPT-3



Key Insight: Scaling unlocked emergent capabilities (few-shot learning)

GPT-3 can perform tasks from examples - no fine-tuning needed!

Few-Shot Learning: The Emergent Capability

Definitions:

- **Zero-shot**: Task description only
  "Translate to French: Hello" → "Bonjour"
- **One-shot**: One example
  "English: Hello, French: Bonjour.
  English: Goodbye, French:" → "Au revoir"
- **Few-shot**: Multiple examples (2-10)
  Show 5 translation pairs, model infers pattern

What's Remarkable:
No gradient updates! Pure inference!

How It Works:

- Model learns to learn during pre-training
- In-context learning
- Pattern matching in prompt
- Emerges at scale (GPT-3, not GPT-1)

Example Tasks:

- Translation (unseen language pairs)
- Arithmetic (3-digit addition)
- Programming (code generation)
- Reasoning (logical inference)

Limitations:

- Inconsistent performance
- Sensitive to prompt wording
- Not as good as fine-tuning

Few-shot learning hints at artificial general intelligence

Worked Example: Data Efficiency of Pre-training

Experiment: Sentiment classification on product reviews

Approach A - Train from Scratch:

- Architecture: LSTM (2 layers, 512 hidden)
- Training data: 10,000 labeled reviews
- Training time: 6 hours on GPU
- Test accuracy: 78.3%

Approach B - Fine-tune GPT-2:

- Base model: GPT-2 (1.5B parameters, pre-trained)
- Training data: 100 labeled reviews (100x less!)
- Training time: 10 minutes on GPU
- Test accuracy: 91.7%

Result: 100 examples + GPT-2 $>$ 10,000 from scratch

100x less data, 36x faster, 17% better accuracy

This is the power of transfer learning - pre-training solves the data problem

Checkpoint: Understanding GPT

Quick Quiz

Question 1:
Why is GPT autoregressive?

A) It's faster
B) Natural for generation
C) Uses less memory
D) More accurate

Question 2:
What's the key difference from BERT?

A) More parameters
B) Different dataset
C) Causal vs bidirectional
D) Slower training

Answer 1: **B) Natural for generation**

- Generation = predict next word
- Can't see future (not written yet)
- Autoregressive = use output as next input
- Perfect fit for the task

Answer 2: **C) Causal vs bidirectional**

- BERT: See full sentence (encoder)
- GPT: See only past (decoder)
- BERT: Better for understanding
- GPT: Better for generation

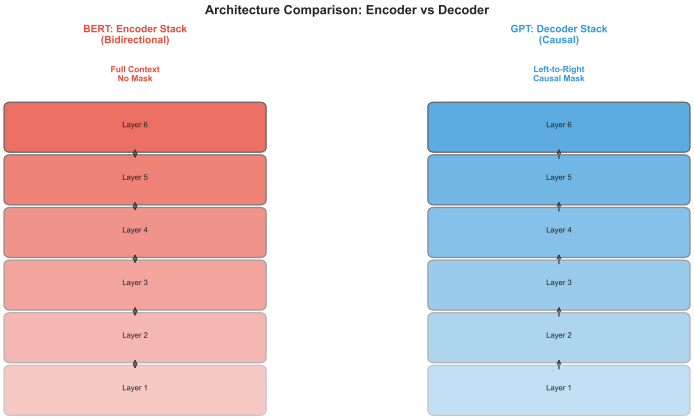
Architecture follows task requirements - understand the why

Part 3: Integration

Comparing and Choosing

Pre-trained models provide powerful starting points for NLP tasks.

BERT vs GPT: Side-by-Side



Key Insight: Encoder for understanding, decoder for generation

Choose based on your task requirements

BERT vs GPT: When to Use Each

Use BERT When:

- **Task**: Classification, QA, NER
- **Need**: Full sentence understanding
- **Input**: Complete text available
- **Output**: Labels or spans

BERT Strengths:

- Bidirectional context
- Best for understanding
- CLS token for sentence embedding
- Handles word sense disambiguation

BERT Applications:

- Search (Google uses BERT)
- Question answering
- Named entity recognition
- Sentiment analysis

Use GPT When:

- **Task**: Generation, completion, dialogue
- **Need**: Coherent text generation
- **Input**: Prompt or partial sequence
- **Output**: Continued text

GPT Strengths:

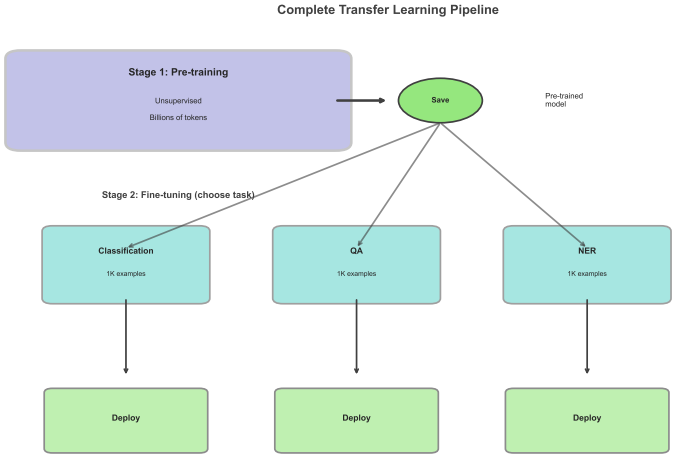
- Natural text generation
- In-context learning (few-shot)
- Scales well (GPT-3: 175B params)
- Handles diverse prompts

GPT Applications:

- ChatGPT (dialogue)
- Code completion (Copilot)
- Creative writing
- Text summarization (generative)

Both are transformers - different objectives, different use cases

The Complete Transfer Learning Pipeline



Key Insight: Unsupervised pre-training + supervised fine-tuning = best of both

One-time expensive pre-training, many cheap fine-tunings

Transfer Learning: Best Practices

Pre-training (Done Once):

- Massive unsupervised corpus (billions of words)
- Self-supervised objectives (MLM or AR)
- Large compute (TPUs/GPUs, weeks)
- Cost: $1M-$10M
- Result: General language model

Who Does This:

- Big tech (Google, OpenAI, Meta)
- Research labs
- Shared publicly
- You download, don't train

Fine-tuning (Per Task):

- Small labeled dataset (100-10K examples)
- Task-specific head
- Small learning rate (2e-5)
- Short training (hours)
- Cost: $50-$500
- Result: Task-specific model

Best Practices:

- Start with pre-trained checkpoint
- Use small learning rate
- Train 2-4 epochs
- Monitor validation loss
- Try different layer freezing

This pipeline is now standard across all of NLP

When NOT to Use Pre-trained Models

Overkill Scenarios:

- Simple regex suffices
- Rule-based system works
- N-grams are enough
- Tiny dataset (¡ 50 examples)

Resource Constraints:

- Limited memory (BERT needs 4GB+ GPU)
- Strict latency requirements (¡ 10ms)
- Edge deployment (phones, IoT)
- Battery-powered devices

Domain Mismatch:

- Highly specialized jargon
- Different language structure
- Pre-training corpus unrepresentative

Better Alternatives:

- DistilBERT (smaller, faster)
- Domain-specific pre-training
- Few-shot prompting (no fine-tuning)
- Ensemble of simple models

Cost-Benefit:
Sometimes simple approaches better ROI

Know when NOT to use powerful models - engineering judgment matters

Common Pitfalls in Fine-tuning

Pitfall 1: Learning Rate Too High

- Pre-trained weights are delicate
- High LR destroys them (catastrophic forgetting)
- Solution: Use 2e-5 to 5e-5 (100x smaller than training from scratch)

Pitfall 2: Too Many Epochs

- Overfits to small dataset
- Solution: 2-4 epochs maximum

Pitfall 3: Wrong Task Head

- Architecture mismatch
- Solution: Use standard heads from examples

Pitfall 4: Ignoring Validation

- Train loss down, test loss up
- Solution: Early stopping on validation

Pitfall 5: Not Trying Layer Freezing

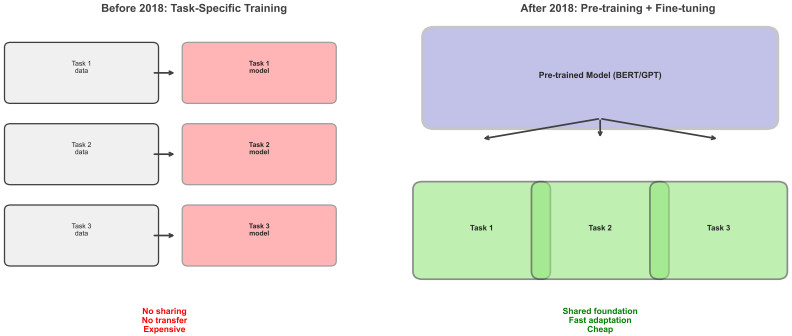
- Small dataset: Freeze bottom layers
- Solution: Experiment with freezing 0, 6, 8, or 10 layers

Pitfall 6: Forgetting Preprocessing

- Tokenization must match pre-training
- Solution: Use model's own tokenizer

Most failures come from hyperparameter choices - start conservative

The Paradigm Shift: Pre-2018 vs Post-2018



The Core Principle: Pre-training solves the data problem

This is the foundation of modern NLP - everything changed in 2018

Key Takeaways

1. **Pre-training on massive unlabeled data**
   Learn general language understanding without task-specific labels
2. **BERT: Bidirectional encoder for understanding**
   Masked LM objective, full context, best for classification/QA
3. **GPT: Autoregressive decoder for generation**
   Next token prediction, causal mask, best for text generation
4. **Fine-tuning adapts with small labeled data**
   100-1000 examples sufficient, days not months, state-of-art results
5. **Transfer learning finally works for NLP**
   Learn once, apply everywhere - the 2018 revolution
6. **2018 was the inflection point**
   BERT and GPT changed everything - modern NLP is post-2018

Master these concepts - they define modern NLP practice

Next: Hands-On Feature Extraction

Lab Activities:

1. Load pre-trained BERT and GPT
2. Extract embeddings from both
3. Compare representation spaces
4. Visualize attention patterns
5. Use features for classification
6. Compare BERT vs GPT effectiveness

What You'll Learn:

- Practical HuggingFace usage
- How to extract features
- BERT vs GPT representations
- Attention pattern interpretation
- Feature-based transfer learning

No Fine-tuning:
We'll use models as feature extractors (simpler, faster)

Let's explore pre-trained models hands-on!

Understanding by doing - the lab makes theory concrete

Technical Appendix

Architecture, Training, and Deployment Details

Pre-trained models provide powerful starting points for NLP tasks.

Appendix A: BERT Architecture Specifications

Component	BERT-Base	BERT-Large
Transformer Layers	12	24
Hidden Size	768	1024
Attention Heads	12	16
Intermediate Size (FFN)	3072	4096
Total Parameters	110M	340M
Max Position Embeddings	512	512
Vocabulary Size (WordPiece)	30,522	30,522
Segment Embeddings	2	2
Training Specs		
Pre-training Corpus	BooksCorpus (800M) + Wikipedia (2.5B)	
Batch Size	256	256
Steps	1M	1M
Learning Rate	1e-4	1e-4
Warmup Steps	10,000	10,000
Hardware	16 TPU pods	64 TPU pods
Training Time	4 days	4 days

BERT-Large has 3x parameters but same training time (more parallelism)

Appendix B: GPT Architecture Specifications

Component	GPT-1	GPT-2	GPT-3
Decoder Layers	12	48	96
Hidden Size	768	1600	12,288
Attention Heads	12	25	96
Context Window	512	1024	2048
Parameters	117M	1.5B	175B
Vocabulary (BPE)	40K	50K	50K
Training			
Dataset	BooksCorpus	WebText	Common Crawl
Dataset Size	5GB	40GB	570GB
Tokens	1B	10B	300B
Batch Size	64	512	3.2M
GPUs	8	256	10,000+
Training Time	Weeks	Weeks	Months
Cost Estimate	$50K	$500K	$4.6M

Exponential scaling in parameters, data, and compute

Appendix C: Pre-training Hyperparameters

BERT Pre-training:

- **Optimizer**: Adam
- **Learning rate**: 1e-4
- β_1, β_2: 0.9, 0.999
- **L2 weight decay**: 0.01
- **Warmup steps**: 10,000
- **LR schedule**: Linear decay
- **Dropout**: 0.1
- **Activation**: GELU
- **Batch size**: 256 sequences
- **Max steps**: 1,000,000
- **Masking**: 15% of tokens

GPT-3 Pre-training:

- **Optimizer**: Adam
- **Learning rate**: 6e-5 (peak)
- β_1, β_2: 0.9, 0.95
- **Weight decay**: 0.1
- **Gradient clipping**: 1.0
- **LR schedule**: Cosine decay
- **Dropout**: Varies by layer
- **Batch size**: 3.2M tokens
- **Tokens**: 300B total
- **Context window**: 2048
- **Precision**: Mixed (FP16/FP32)

Key Insight: These are carefully tuned over months of experimentation

Don't change these for fine-tuning - use proven recipes

Appendix D: Fine-tuning Recipes by Task

Task	Learning Rate	Epochs	Strategy
Classification			
Sentiment	2e-5	3-4	Full fine-tuning
Topic	3e-5	2-3	Full fine-tuning
Spam	2e-5	4	Freeze bottom 6
Question Answering			
SQuAD	3e-5	2	Full fine-tuning
Custom QA	5e-5	3-4	Full fine-tuning
NER			
Named Entities	5e-5	3	Full fine-tuning
Domain-specific	2e-5	4-5	Freeze bottom 8
Generation (GPT)			
Completion	2e-5	2-3	Full fine-tuning
Dialogue	1e-5	3-5	Full fine-tuning
Summarization	3e-5	2-3	Full fine-tuning

Batch size: 16-32, Warmup: 10% of steps

Start with these proven recipes, then experiment

Appendix E: Training Cost Analysis

Pre-training Costs (2018-2024):

Model	Cost
BERT-base	$7K
BERT-large	$25K
GPT-1	$50K
GPT-2	$500K
GPT-3	$4.6M
GPT-4 (est)	$50M+

Why So Expensive:

- Massive datasets (100B-1T tokens)
- Large models (1B-1T parameters)
- Weeks/months on thousands of GPUs
- Trial and error in hyperparameters

Fine-tuning Costs (Per Task):

Dataset Size	Cost
100 examples	$5-10
1,000 examples	$50-100
10,000 examples	$200-500

The Economics:

- Pre-training: One-time investment
- Fine-tuning: Cheap per task
- Amortize cost across many applications

Business Model:
OpenAI, Anthropic, Google: Pay for pre-training, sell API access

Pre-training economics enable the modern AI industry

Appendix F: Further Reading and Resources

Original Papers:

- **Attention**: Vaswani et al. (2017)
  "Attention is All You Need"
- **GPT-1**: Radford et al. (June 2018)
  "Improving Language Understanding by Generative Pre-Training"
- **BERT**: Devlin et al. (October 2018)
  "BERT: Pre-training of Deep Bidirectional Transformers"
- **GPT-2**: Radford et al. (2019)
  "Language Models are Unsupervised Multitask Learners"
- **GPT-3**: Brown et al. (2020)
  "Language Models are Few-Shot Learners"

Practical Resources:

- **HuggingFace Transformers**
  Library for using pre-trained models
  https://huggingface.co/transformers
- **Model Hub**
  Thousands of pre-trained models
  https://huggingface.co/models
- **Fine-tuning Tutorials**
  Official guides for common tasks
- **Papers With Code**
  Leaderboards and implementations

Next Week:
Advanced architectures (T5, GPT-4, etc.)

These papers are essential reading for serious NLP practitioners

Advanced Transformers

Week 7 - The Predictable Path to Intelligence

NLP Course 2025

October 27, 2025

Two-Tier BSc Discovery Presentation

What You'll Learn Today

By the end of this lecture, you will:

1. **Understand** scaling laws and their implications for AI development
2. **Explain** compute-optimal training strategies (Chinchilla)
3. **Compare** model size vs data size tradeoffs
4. **Identify** emergent abilities in large models
5. **Evaluate** different paths to better transformers

Prerequisites:

- Transformer architecture (Week 5)
- Pre-training concepts (Week 6)
- Basic understanding of loss functions

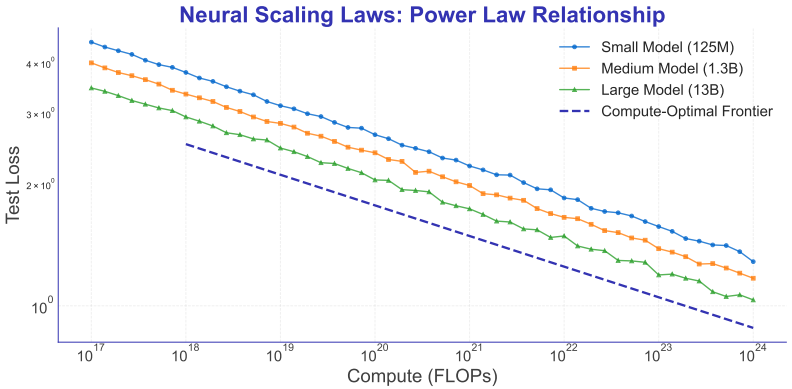
Plan for Today:

1. The scaling laws paradox
2. Three paths: Bigger, Smarter, Efficient
3. Chinchilla: Compute-optimal training
4. Emergent abilities and phase transitions
5. Practical implications

Key Question: Why is AI progress so predictable?

Scaling laws reveal fundamental patterns in how neural networks learn

The Scaling Laws Paradox



Why is AI progress so predictable?

Key Insight: Performance follows power laws - we can predict the future

Bigger models are better in predictable, measurable ways

Three Paths to Better Transformers

Path 1: Bigger

GPT-3 (2020)

- 175B parameters
- 1000× GPT-1
- Emergent abilities
- $4.6M training cost

Bet: Scale alone works

Path 2: Smarter

Mixture of Experts

- 1.6T parameters
- Only 10B active
- Sparse activation
- Same cost as 100B

Bet: Efficiency matters

Path 3: Efficient

Reformer, Linformer

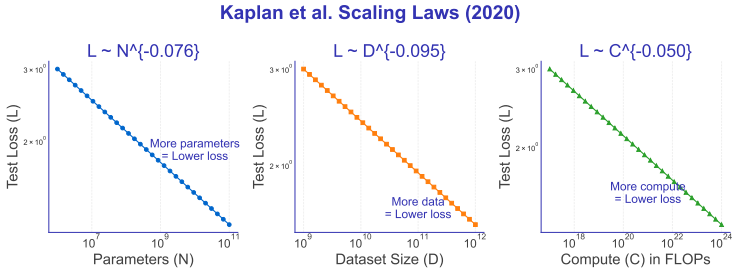
- Reduce $O(n^2)$
- Long sequences
- Lower memory
- Practical deployment

Bet: Algorithms matter

All three work! Next 17 slides explore each path

Different paths for different constraints - no single winner

The Kaplan Scaling Laws (2020)



Key Insight: Loss decreases as power law with parameters, data, compute

Three independent power laws - all perfectly smooth

Scaling Laws: The Mathematics

The Three Power Laws:

$$L(N) = (N_c/N)^{\alpha_N}$$
$$L(D) = (D_c/D)^{\alpha_D}$$
$$L(C) = (C_c/C)^{\alpha_C}$$

where:

- N: Number of parameters
- D: Dataset size (tokens)
- C: Compute (FLOPs)
- $\alpha_N \approx 0.076$
- $\alpha_D \approx 0.095$
- $\alpha_C \approx 0.050$

Key Discovery:
Smooth, predictable improvement

Worked Example:
GPT-3: $N = 175$B, $L = 2.1$ nats

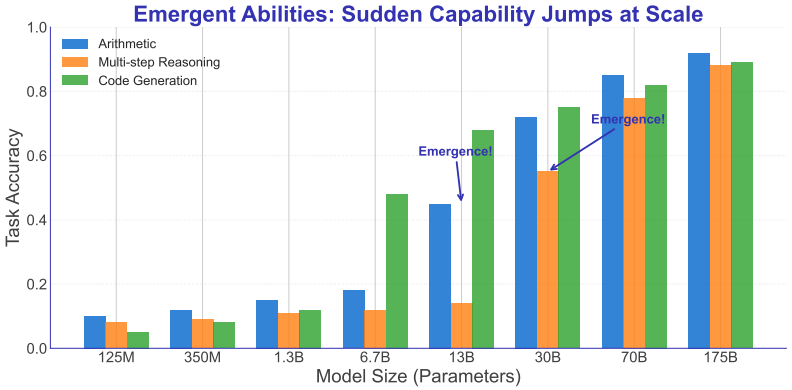
Predict GPT-4 ($N = 500$B hypothetical):

$$\begin{aligned} L(500B) &= L(175B) \times (175B/500B)^{0.076} \\ &= 2.1 \times (0.35)^{0.076} \\ &= 2.1 \times 0.92 \\ &= 1.93 \text{ nats} \end{aligned}$$

Interpretation:
8% loss reduction from 3x parameters
Matches observed scaling!

These laws held from 1M to 175B parameters - remarkable consistency

Emergent Abilities at Scale



Key Insight: New capabilities appear suddenly at threshold model sizes

Not gradual improvement - discontinuous jumps in ability

Chinchilla: Most Models Are Undertrained

The Discovery (2022):
GPT-3: 175B params, 300B tokens

Chinchilla: 70B params, 1.4T tokens

Result: Chinchilla beats GPT-3!

The Rule:

$$\text{Optimal tokens} \approx 20 \times \text{parameters}$$

GPT-3 should have seen:
$175B \times 20 = 3.5T$ tokens
Actually saw: 300B (9% of optimal!)

Why This Matters:

- Most large models undertrained
- Training longer is cheaper than bigger model
- Compute allocation matters

Practical Impact:

- LLaMA-2 (70B): Trained on 2T tokens
- Follows Chinchilla scaling
- Outperforms GPT-3 with 2.5x fewer params
- Cheaper to run, same quality

Lesson:
Don't just make models bigger - train them longer!

Chinchilla changed how we think about model training

Worked Example: Compute-Optimal Model

Given: Fixed compute budget $C = 10^{23}$ FLOPs

Question: How many parameters N and tokens D to use?

Chinchilla Formula:
For optimal allocation:

$$N_{opt} \approx 0.73 \times C^{0.37}$$
$$D_{opt} \approx 1.45 \times C^{0.37}$$

Calculate:

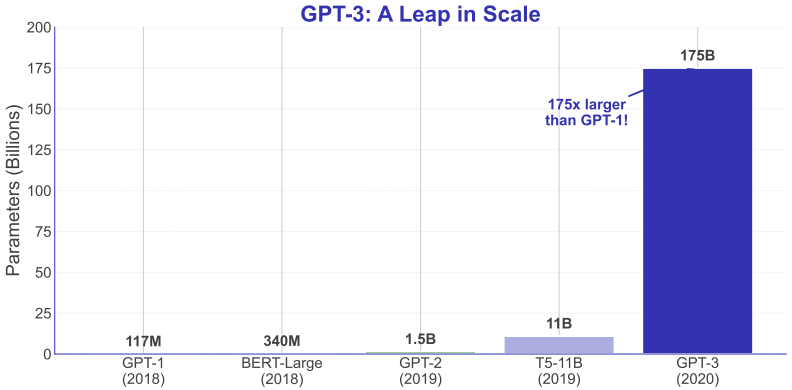
$$N_{opt} = 0.73 \times (10^{23})^{0.37} = 0.73 \times 1.86 \times 10^{8}$$
$$\approx 136M \text{ parameters}$$

$$D_{opt} = 1.45 \times (10^{23})^{0.37} \approx 270M \text{ tokens}$$

Verification: $D_{opt}/N_{opt} = 270M/136M \approx 2$ (roughly $20\times$ rule)

Optimal compute allocation: Balance parameters and training data

GPT-3: The Scale Breakthrough



Key Insight: 175 billion parameters - 1000× GPT-1

Training: 300B tokens, 3640 petaflop-days, $4.6M cost

GPT-3 Architecture Details

Specifications:

- **Layers**: 96 decoder layers
- **Hidden size**: 12,288 dimensions
- **Attention heads**: 96 (128 dim each)
- **Context window**: 2048 tokens
- **Parameters**: 175 billion
- **Vocabulary**: 50,257 (BPE)

Why So Big:
Scale enables emergent abilities

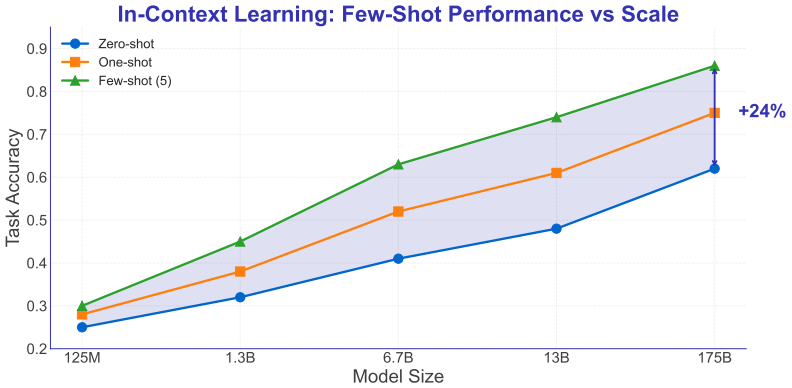
Training:

- **Dataset**: Common Crawl (570GB)
- **Tokens**: 300 billion
- **Batch size**: 3.2M tokens
- **Hardware**: 10,000+ GPUs
- **Time**: Several months
- **Cost**: $4.6 million

Notable:
8 different model sizes tested (125M to 175B)
All follow same scaling law!

Largest dense transformer ever trained (as of 2020)

Few-Shot Learning: Emergent at Scale



Key Insight: Capabilities emerge suddenly above threshold sizes

Few-shot learning doesn't work below 13B parameters - then suddenly does

The Cost of Scale

Training Cost Escalation:

Model	Cost
GPT-1 (117M)	\$50K
GPT-2 (1.5B)	\$500K
GPT-3 (175B)	\$4.6M
GPT-4 (est 1.7T)	\$50M+

100× parameters ≈ 1000× cost

Diminishing Returns:
Each 10× scale:

- Cost: 10× increase
- Loss: 0.076 decrease (8%)
- Linear cost, log gains

Why Continue Scaling:

- Emergent abilities worth the cost
- Few-shot learning transforms UX
- Reasoning capabilities
- Multimodal integration (GPT-4)

Alternative Strategies:

- Chinchilla: Train longer, not bigger
- MoE: Sparse activation
- Efficient: Reduce $O(n^2)$
- Distillation: Compress after training

The Tension:
Better models vs affordable training

Scaling works but is expensive - motivates smarter approaches

Worked Example: Predicting Future Performance

Given: GPT-3 at 175B params has loss $L = 2.1$ nats

Question: What loss would 500B parameter model achieve?

Using Scaling Law: $L(N) = L_0 \times (N_0/N)^{\alpha}$ where $\alpha = 0.076$

Step 1: Set up equation

$$L(500B) = 2.1 \times (175B/500B)^{0.076}$$

Step 2: Calculate ratio

$$\frac{175}{500} = 0.35$$

Step 3: Apply exponent

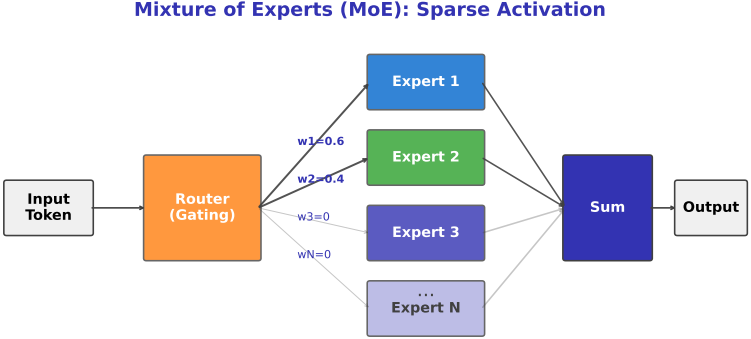
$$0.35^{0.076} \approx 0.92$$

Step 4: Final loss

$$L(500B) = 2.1 \times 0.92 = 1.93 \text{ nats}$$

Interpretation: $3\times$ parameters $\rightarrow$ 8% loss reduction (diminishing returns!)

Mixture of Experts: Sparse Activation



Key Insight: Train 1T parameters, activate only 10B per example

Sparse models: Capacity of large, cost of small

Mixture of Experts: How It Works

Components:

- **Router**: Gating network (small NN)
- **Experts**: E specialist FFNs
- **Top-k selection**: Use best k experts

Forward Pass:

1. Router computes scores for all experts
2. Select top-k (typically $k = 2$)
3. Activate only selected experts
4. Weighted sum of expert outputs

Example:

- 128 experts total
- Top-2 selection
- Activate: $2/128 = 1.6\%$

Gating Function:

$$G(x) = \text{softmax}(\text{KeepTopK}(W_g \cdot x, k))$$

Output:

$$y = \sum_{i \in \text{Top-k}} G(x)_i \cdot E_i(x)$$

Load Balancing:
Auxiliary loss ensures experts used equally

$$L_{aux} = \alpha \times \sum_{i=1}^{E} f_i \times P_i$$

where f_i = fraction routed to expert i

Benefits:

- $100\times$ parameters at $2\times$ cost
- Experts specialize (syntax, semantics, etc.)
- Conditional computation

MoE is how to scale beyond dense models

Worked Example: MoE Router Computation

Given: Input token x, 8 experts, top-2 selection

Step 1: Router computes logits

$$\text{logits} = W_g \cdot x = [2.1, 0.3, 1.8, 0.1, 3.2, 0.5, 1.1, 0.8]$$

Step 2: Select top-2
Top-2 experts: Expert 5 (3.2) and Expert 1 (2.1)

Step 3: Softmax over top-2 only

$$G_5 = \frac{\exp(3.2)}{\exp(3.2) + \exp(2.1)} = \frac{24.5}{32.6} = 0.75$$

$$G_1 = \frac{\exp(2.1)}{\exp(3.2) + \exp(2.1)} = \frac{8.17}{32.6} = 0.25$$

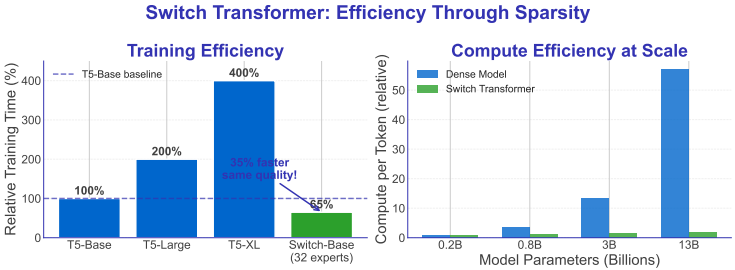
Step 4: Weighted sum

$$y = 0.75 \times E_5(x) + 0.25 \times E_1(x)$$

Sparsity: Only 2/8 experts activated = 75% reduction in computation!

Router learns which experts are relevant for which inputs

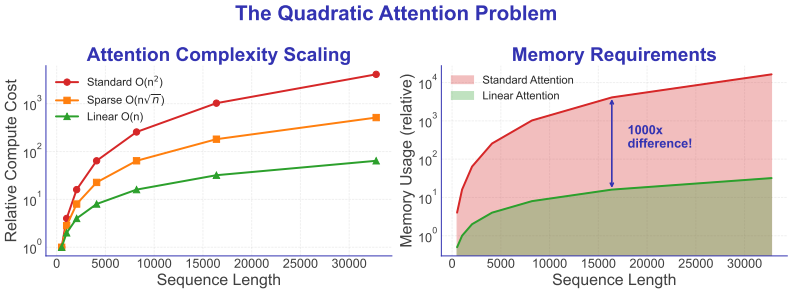
Switch Transformer: 1.6 Trillion Parameters



Key Insight: MoE (1.6T params) beats dense (100B) at same compute cost

Google's Switch Transformer (2021) - largest model at the time

The Attention Complexity Problem



Key Insight: Standard attention is $O(n^2)$ - memory explodes for long sequences

$n = 512$ **fine,** $n = 8192$ **impossible on most GPUs**

Efficient Transformers: Reducing Complexity

Reformer (LSH Attention):

- Locality-Sensitive Hashing
- Group similar queries/keys
- Attend only within groups
- Complexity: $O(n \log n)$

Linformer:

- Low-rank projection
- Project $n \times n$ to $n \times k$
- $k \ll n$ (e.g., 256)
- Complexity: $O(n)$

Linear Attention:

- Kernel trick
- Reorder operations
- Never compute $n \times n$ matrix
- Complexity: $O(n)$

When to Use:

- **Long documents**: $n > 4096$
- **Limited memory**: Consumer GPUs
- **Real-time**: Latency critical
- **Edge deployment**: Mobile, IoT

Multiple approaches to same problem - choose based on use case

Worked Example: Memory Savings from Efficient Attention

Given: Sequence length $n = 4096$, hidden dim $d = 512$

Standard Attention Memory:
Attention matrix: $n \times n = 4096 \times 4096 = 16.8M$ floats
Memory: $16.8M \times 4$ bytes $= 67$ MB per attention head
With 16 heads: $67 \times 16 = 1$ GB just for attention!

Linformer Memory ($k = 256$):
Projected matrix: $n \times k = 4096 \times 256 = 1M$ floats
Memory: $1M \times 4$ bytes $= 4$ MB per head
With 16 heads: $4 \times 16 = 64$ MB

Savings: $\frac{1GB}{64MB} = 16\times$ memory reduction!

Impact: Can process 4x longer sequences on same hardware

Efficient attention enables long-context applications

Technical Appendix

Deep Dive: GPT-3, MoE, Efficient Transformers

Scaling laws guide decisions about model size and data.

Appendix A1: GPT-3 Complete Specifications

Component	Small	Medium	Large	175B
Parameters	125M	350M	1.3B	175B
Layers	12	24	24	96
Hidden Size	768	1024	2048	12,288
Heads	12	16	16	96
Head Dimension	64	64	128	128
Context	2048	2048	2048	2048
Batch Size	0.5M	0.5M	1M	3.2M
Learning Rate	6e-4	3e-4	2.5e-4	1.2e-4
Performance				
LAMBADA (acc)	42.7	54.3	63.6	76.2
HellaSwag (acc)	43.6	54.7	67.4	78.9

Key Pattern: Consistent scaling across all metrics

Every metric improves smoothly with size

Appendix A2: GPT-3 Training Infrastructure

Hardware:

- 10,000+ NVIDIA V100 GPUs (32GB each)
- 285,000 CPU cores
- 400 Gbps network interconnect
- Custom Microsoft Azure infrastructure

Training Details:

- Distributed across thousands of nodes
- Model parallelism: Split model across GPUs
- Data parallelism: Different batches per GPU
- Pipeline parallelism: Layer-wise distribution
- Mixed precision training (FP16/FP32)

Challenges:

- Synchronization overhead
- Gradient accumulation across devices
- Fault tolerance (GPU failures during training)
- Checkpointing (model state = 350GB)

Training Time: Several months wall-clock time

Training GPT-3 requires infrastructure beyond most organizations

Appendix A3: Few-Shot Prompting Best Practices

Prompt Engineering for GPT-3:

Zero-Shot:
`Translate to French: Hello` → Bonjour

One-Shot:
`English: Hello, French: Bonjour`
`English: Goodbye, French:` → Au revoir

Few-Shot (Optimal):
```
Translate English to French:
English: Hello / French: Bonjour
English: Goodbye / French: Au revoir
English: Thank you / French: Merci
English: Please / French: → S'il vous plaît
```

Best Practices:

- Use 3-10 examples (more doesn't always help)
- Examples should be diverse
- Format consistency critical
- Order matters (recency bias)
- Few-shot > fine-tuning for small datasets (¡ 100 examples)

Prompting is an art - small changes yield large effects

Appendix A4: GPT-3 API and Pricing

API Access:

- No model download (too large)
- API calls only (OpenAI servers)
- Multiple model sizes available

Pricing (2024):

Model	Input (per 1M tokens)	Output
GPT-3.5-turbo	$0.50	$1.50
GPT-4-turbo	$10.00	$30.00
GPT-4 (8K)	$30.00	$60.00

Rate Limits:

- Free tier: 3 requests/minute
- Paid tier: 3500 requests/minute
- Tokens per minute: 90K-2M depending on tier

Business Model: Amortize $4.6M training cost across millions of users

API pricing reflects compute cost and value delivered

Appendix A5: GPT-3 to GPT-4 Evolution

GPT-4 Improvements (2023):

- **Multimodal**: Images + text input
- **Larger context**: 32K tokens (16× GPT-3)
- **Better reasoning**: Chain-of-thought, mathematical
- **RLHF**: Reinforcement learning from human feedback
- **Parameters**: Rumored 1.7T (unconfirmed)

Performance Gains:

- Bar exam: 10th percentile → 90th percentile
- Coding: HumanEval 48% → 67%
- MMLU (general knowledge): 70% → 86%
- Reduced hallucinations by 40%

Architecture Speculation:

- Likely MoE (not confirmed)
- Multiple expert models combined
- Compute-optimal training (Chinchilla-informed)

GPT-4 shows continued scaling + better training

Appendix A6: MoE Router Network Mathematics

Router Architecture:

$$h = W_g x \in \mathbb{R}^E$$

where E = number of experts

Top-k Gating:

$$G(x) = \text{softmax}(\text{KeepTopK}(h, k))$$

KeepTopK sets all but top-k values to $-\infty$ before softmax

Sparse Gating Properties:

- Only k experts get non-zero weight
- Weights sum to 1 (softmax)
- Differentiable (can train with backprop)
- Gradient flows only through selected experts

Sparsity Benefit:
$k = 2$, $E = 128$: Activate $2/128 = 1.6\%$ of parameters
$100\times$ parameters at $2\times$ compute!

Sparsity is key - conditional computation based on input

Appendix A7: Load Balancing in MoE

Problem: Some experts get all traffic, others unused

Auxiliary Loss:

$$L_{aux} = \alpha \times \text{CV}(f_1, f_2, ..., f_E)^2$$

where CV = coefficient of variation, f_i = fraction routed to expert i

Penalizes imbalanced routing

Capacity Factor:
Limit tokens per expert:

$$\text{capacity}_i = \frac{\text{total tokens}}{E} \times \text{capacity factor}$$

Typical capacity factor = 1.25

Expert Choice Routing (alternative):
Experts choose tokens instead of tokens choosing experts
Better load balance, more stable training

Load balancing critical for effective expert utilization

Appendix A8: Switch Transformer Details

Google's Switch Transformer (2021):

- 1.6 trillion parameters (largest at time)
- Top-1 routing (simplest form of MoE)
- 2048 experts per layer
- Trained on C4 dataset (750GB)

Key Innovation: Simplified MoE

- Top-1 instead of top-2 (simpler)
- Expert capacity (limit tokens per expert)
- Smaller routers (reduced overhead)
- Better scaling than previous MoE

Results:

- 4× faster pre-training than T5-XXL (same quality)
- 7× faster fine-tuning
- Outperforms dense models at matched compute

Largest model trained demonstrates MoE viability

Appendix A9: Training Dynamics and Expert Specialization

What Do Experts Learn?
Empirical findings:

- Some experts specialize by syntax
- Some by semantics
- Some by domain (code, math, language)
- Specialization emerges during training

Training Challenges:

- **Mode collapse**: All traffic to few experts
- **Instability**: Router can oscillate
- **Expert imbalance**: Uneven utilization
- **Gradient noise**: Discrete routing not smooth

Solutions:

- Strong auxiliary losses
- Careful initialization
- Dropout on router
- Expert capacity constraints

MoE training trickier than dense - but rewards are huge

Appendix A10: MoE vs Dense - Complete Comparison

Aspect	Dense (GPT-3)	MoE (Switch)
Active Parameters	175B	10B
Total Parameters	175B	1600B
Sparsity	0%	99.4%
Memory (inference)	350GB	20GB
Training Time	Baseline	4× faster
Fine-tuning Time	Baseline	7× faster
Quality (matched compute)	Good	Better
Implementation	Simpler	Complex
Deployment	Standard GPUs	Needs coordination

When to Use MoE:

- Want massive capacity
- Have coordination infrastructure
- Training cost constrained
- Serving many requests (can batch)

When to Use Dense:

- Simpler deployment
- Low latency critical
- Small-scale serving

Appendix A11: Reformer and LSH Attention

Locality-Sensitive Hashing (LSH):

- Hash queries and keys to buckets
- Similar vectors $\rightarrow$ same bucket (high probability)
- Attend only within bucket

LSH Function:

$$h(x) = \text{argmax}([x \cdot r_1, x \cdot r_2, \ldots, x \cdot r_b])$$

where r_i are random projection vectors

Complexity:
Standard: $O(n^2)$
Reformer: $O(n \log n)$

Trade-offs:

- Much faster for $n > 4096$
- Approximate (misses some attention pairs)
- Enables sequences up to 64K tokens
- More complex implementation

Reformer enables long-context applications (books, long documents)

Appendix A12: Linformer Low-Rank Approximation

Key Idea: Attention matrix is approximately low-rank

Standard Attention:

$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d}})V$$

where $QK^T \in \mathbb{R}^{n \times n}$

Linformer Modification:
Project keys and values to lower dimension k:

$$K' = KE_k \in \mathbb{R}^{n \times k}$$
$$V' = VF_k \in \mathbb{R}^{n \times k}$$

Then: $QK'^T \in \mathbb{R}^{n \times k}$ instead of $\mathbb{R}^{n \times n}$

Complexity:
Standard: $O(n^2 d)$
Linformer: $O(nkd)$ where $k = 256$ typical

For $n = 8192$, $k = 256$: $32\times$ speedup!

Simple idea, dramatic impact - linear complexity

Appendix A13: Linear Attention via Kernel Trick

Kernel Reformulation:
Standard: $\text{Attention}(Q, K, V) = \text{softmax}(QK^T)V$

Rewrite softmax as kernel: $\phi(q)^T\phi(k)$

Key Trick: Reorder operations

$$\text{Attention} = \phi(Q)(\phi(K)^T V)$$

Compute $\phi(K)^T V$ first! This is $k \times d$ (small)

Complexity:

- Standard: $(n \times n) \times (n \times d) = O(n^2 d)$
- Linear: $(n \times k) \times (k \times d) = O(nkd)$
- With $k = d$: $O(nd^2)$ - linear in n!

Approximation Quality:
Not exact, but empirically good
Works well for long sequences

Kernel trick enables true linear attention

Appendix A14: FlashAttention - GPU Optimization

Problem: Standard attention is IO-bound, not compute-bound

GPU memory hierarchy:

- SRAM (on-chip): Fast but tiny (20MB)
- HBM (GPU RAM): Slow but large (40GB)

Standard Attention IO:

1. Load Q, K from HBM
2. Compute $S = QK^T$ (write to HBM)
3. Load S from HBM
4. Apply softmax (write to HBM)
5. Load P, V from HBM
6. Compute PV (write to HBM)

Multiple slow HBM reads/writes!

FlashAttention Optimization:

- Fuse operations (compute in SRAM)
- Tiling (process in blocks)
- Never materialize full $n \times n$ matrix
- 2-4$\times$ faster, less memory

Algorithm innovation - same math, better hardware utilization

Appendix A15: Choosing Efficient Transformer Variant

Variant	Complexity	Exact	Max n	Use Case
Standard	$O(n^2)$	Yes	2K-4K	Default
Reformer	$O(n \log n)$	No	64K	Long docs
Linformer	$O(n)$	No	32K	Fast training
Linear Attn	$O(n)$	No	16K	Real-time
FlashAttention	$O(n^2)$	Yes	8K	GPU-optimized
Best for:				
Books (100K+)	Reformer			
Articles (8-16K)	Linformer			
Production (2-4K)	FlashAttn			
Mobile/Edge	Linear Attn			

Recommendation:

- Start with FlashAttention (better standard)
- Use Linformer if need $n > 4K$
- Reformer for extreme lengths ($n > 32K$)
- Linear attention for edge deployment

Choose based on sequence length and deployment constraints

Tokenization

Week 8 - From Bytes to Subwords

NLP Course 2025

October 27, 2025

Two-Tier BSc Discovery

What You'll Learn Today

By the end of this lecture, you will:

1. **Understand** why tokenization matters for LLMs
2. **Explain** the BPE algorithm step-by-step
3. **Compare** BPE, WordPiece, and SentencePiece
4. **Identify** tokenization issues in multilingual settings
5. **Evaluate** vocabulary size tradeoffs

Prerequisites:

- Basic understanding of word embeddings
- Vocabulary and OOV concepts
- Sequence processing intuition

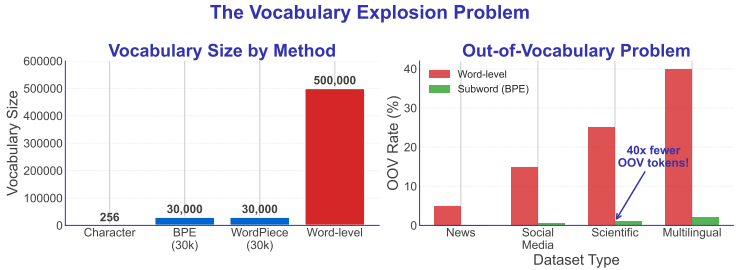
Plan for Today:

1. The vocabulary explosion problem
2. Character vs Word vs Subword tradeoffs
3. BPE algorithm deep dive
4. Practical tokenization with HuggingFace
5. Multilingual challenges

Key Question: How do we represent ANY word with a fixed vocabulary?

Tokenization is the foundation that all language models build upon

The Vocabulary Explosion Problem



Key Insight: 100K word vocab = 30M embedding parameters - unsustainable

English: 170K words. All languages: millions. We need a better approach.

The Trilemma

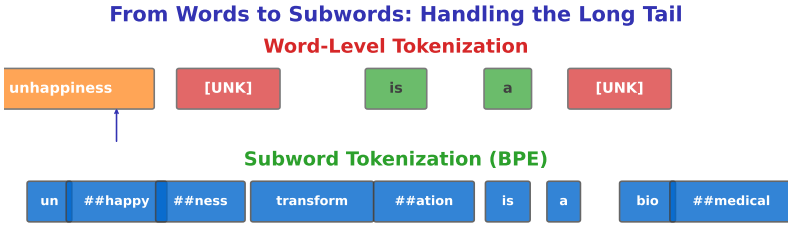
Character-Level
Vocab: 256
Memory: Tiny
Sequences: 5× longer
Speed: Slow

Word-Level
Vocab: 100K+
Memory: Huge
Sequences: Short
OOV: Can't handle "COVID"

Subword (Solution)
Vocab: 30K
Memory: Right-sized
Sequences: Reasonable
OOV: Handles everything

Subwords are the Goldilocks zone - not too big, not too small

What Are Subwords?



Benefits of Subword Tokenization:

Handles unseen words by breaking into known subwords

Captures morphological patterns (un-, -ness, -ation)

Fixed vocabulary size (typically 30-50k tokens)

Zero OOV tokens - any text can be tokenized

Key Insight: Split words into meaningful fragments

"unhappiness" = ["un", "happiness"] - morphology-aware

Three Subword Methods

BPE
Byte-Pair Encoding
Bottom-up merging
Most common

WordPiece
Likelihood-based
BERT uses this
Similar to BPE

SentencePiece
Unigram LM
Language-agnostic
Google's standard

All three work well - BPE most widely used

Why Subwords Work

Key Advantages:

- Fixed vocabulary size (30K typical)
- Handle rare/unknown words via composition
- Capture morphology ("play" in "playing", "player")
- Language-agnostic (same algorithm for all languages)
- Balance sequence length vs vocab size

Example: "COVID-19" (unseen)

- Word-level: UNK (fails!)
- Subword: ["CO", "VI", "D", "-", "19"] (works!)

Compositionality solves the OOV problem

BPE: The Core Idea

Byte-Pair Encoding (Sennrich et al., 2016)

Algorithm:

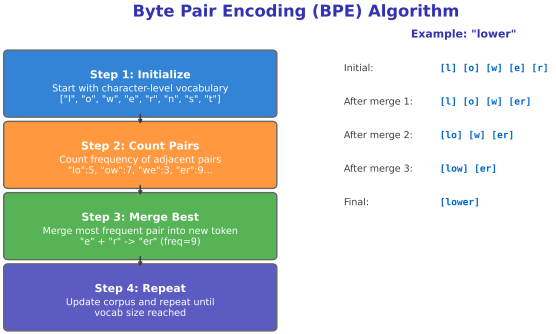
1. Start with characters
2. Find most frequent pair
3. Merge into single token
4. Repeat until desired vocabulary size

Example:
Corpus: "low low low lowest lowest"

- Most frequent pair: ("l", "o") appears 5 times
- Merge: "lo"
- Result: "lo w lo w lo w lo west lo west"
- Repeat...

Greedy algorithm - simple but effective

BPE Algorithm Flowchart



Key Insight: BPE learns a data-driven vocabulary by iteratively merging frequent pairs

Iterative merging builds vocabulary bottom-up

Worked Example: BPE Merge Steps

Corpus: "low low low low lowest lowest"

Initial: Characters = l, o, w, e, s, t

Step 1: Count pairs
(l,o): 6 times
(o,w): 4 times
(e,s): 2 times
Most frequent: (l,o)
Merge: "lo" added to vocabulary

Step 2: Corpus now "lo w lo w lo w lo w lo west lo west"
Count pairs:
(lo,w): 4 times
(w,e): 2 times
Merge: "low"

Continue until 30,000 tokens...

Real BPE runs millions of merges - this shows the pattern

WordPiece: BERT's Tokenizer

Similar to BPE but chooses merges by likelihood increase

Key Difference:
BPE: Max frequency
WordPiece: Max log P(corpus) increase

Example: "unhappiness" $\rightarrow$ ["un", "##happiness"]
indicates continuation

Used By: BERT, DistilBERT, ALBERT (30,522 tokens)

Likelihood-based selection slightly better empirically

Key Takeaways

1. Subword tokenization solves vocabulary explosion
2. BPE: Greedy merging of most frequent pairs
3. 30K vocabulary balances coverage and efficiency
4. Handles rare/OOV words via composition
5. Universal across all modern transformers

Tokenization is foundational - all models use it

Technical Appendix

Tokenization choices significantly impact model performance.

Appendix A1: BPE Algorithm Complete

Formal Algorithm:

vocab $\leftarrow$ all characters
while —vocab— ¡ target_size **do**
    pairs $\leftarrow$ count all adjacent pairs in corpus
    best_pair $\leftarrow$ argmax pairs
    vocab $\leftarrow$ vocab $\cup$ {best_pair}
    Replace all occurrences of best_pair in corpus
end while

Complexity: $O(N \times V)$ where N = corpus size, V = vocab size

Stopping: When vocab reaches 30K-50K (empirically optimal)

Simple greedy algorithm with strong empirical performance

Decoding Strategies

Week 9: From Probabilities to Text

November 2025

Learning Goals & Plan for Today

Learning Goals

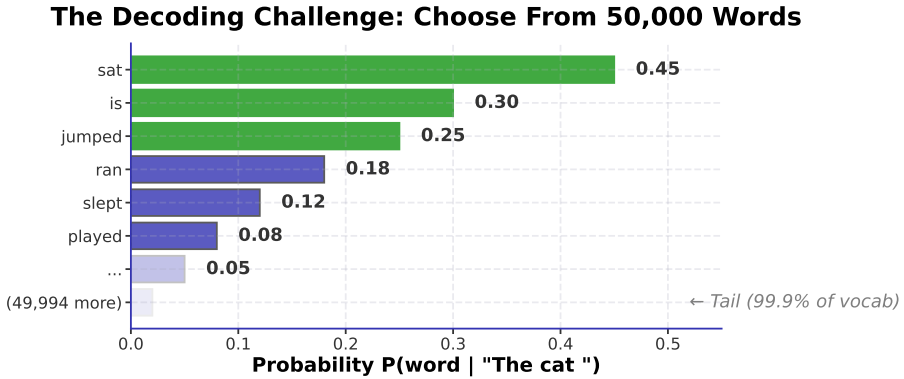
1. Discover why predicting one word at a time creates text quality problems
2. Learn how different decoding strategies solve these problems
3. Practice selecting and tuning methods for different use cases

Plan for Today

1. The challenge: Word predictions → Complete text
2. The toolbox: 6 strategies for text generation
3. Hands-on: Compare methods in practice

Notebook available: `week09_decoding_lab.ipynb`

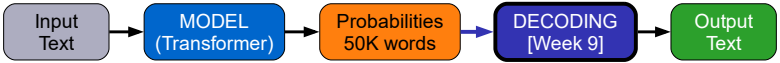
The Decoding Challenge: Choosing From 50,000 Words



The Question: Given these probabilities for "The cat __", which word should we pick?

At each step, model outputs probability distribution over entire vocabulary - how do we choose?

Context: How We Got Here

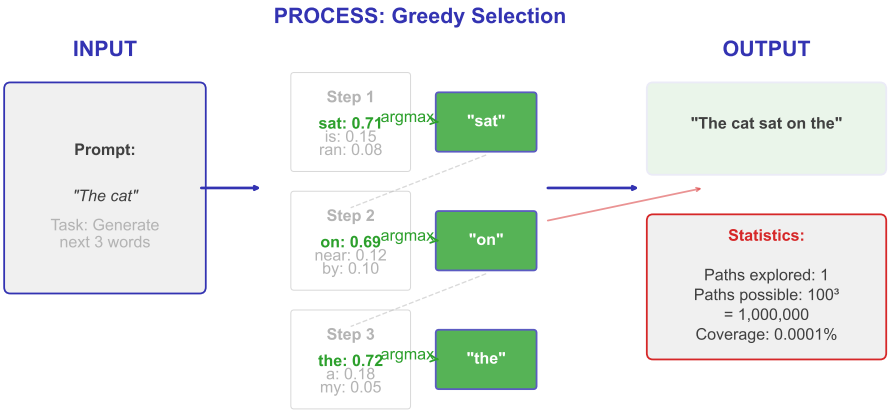


Our Journey:

1. We trained models (Weeks 3-7: RNN $\rightarrow$ Transformers $\rightarrow$ BERT/GPT)
2. They learned to predict: $P(\text{word}|\text{context})$
3. They output probability distributions over 50,000+ words
4. **Today**: How do we convert these probabilities into actual text?

Models predict probabilities. Decoding converts probabilities to text.

Extreme Case 1: Greedy Decoding (Too Narrow)



Extreme 1: Too narrow - misses 99.9999999% of search space

What If We Explored More Paths?

Greedy chose: "The cat sat" (P=0.68)

But it ignored these alternatives:

"The cat walked"	P=0.12	(might lead to better text)
"The cat jumped"	P=0.08	(more interesting)
"The cat slept"	P=0.06	(different story)
"The cat ran"	P=0.04	(action-oriented)

Question: What if we kept ALL 100 words at each step?

Think: $100 \times 100 \times 100 \times 100 \times 100 = ?$

What If We Explored More Paths?

Greedy chose: "The cat sat" (P=0.68)

But it ignored these alternatives:

"The cat walked"	P=0.12	(might lead to better text)
"The cat jumped"	P=0.08	(more interesting)
"The cat slept"	P=0.06	(different story)
"The cat ran"	P=0.04	(action-oriented)

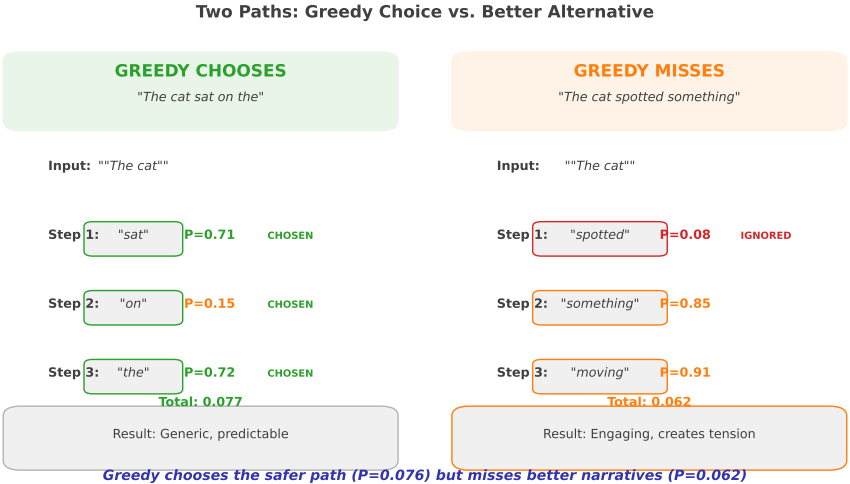
Question: What if we kept ALL 100 words at each step?

Think: $100 \times 100 \times 100 \times 100 \times 100 = ?$

Answer: 10 billion paths! Let's see what happens...

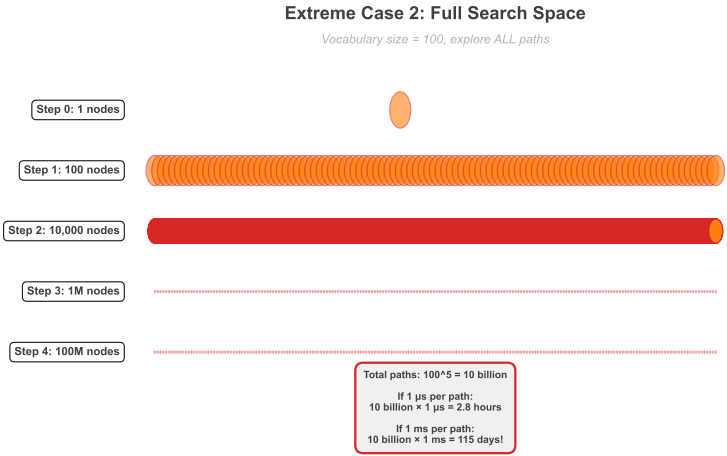
From 1 path to ALL paths - what could go wrong?

Greedy's Fatal Flaw: Missing Better Paths



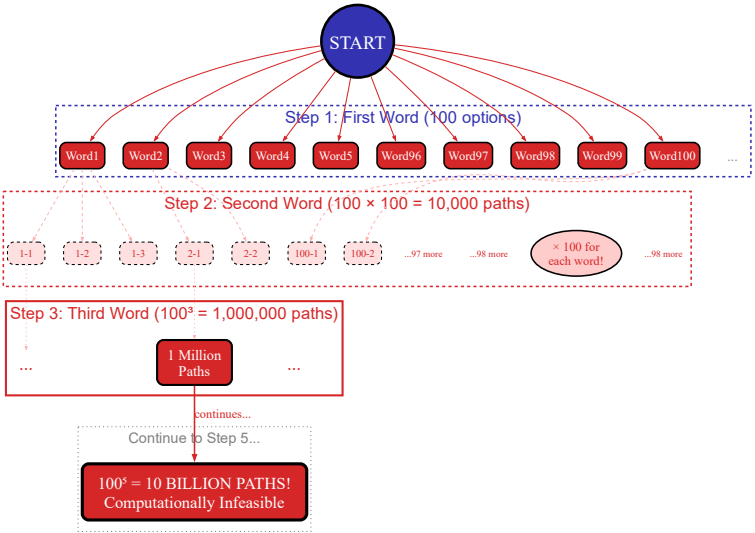
Greedy commits early, missing narratively richer paths despite lower initial probability

Extreme Case 2: Full Search Space (Too Broad)

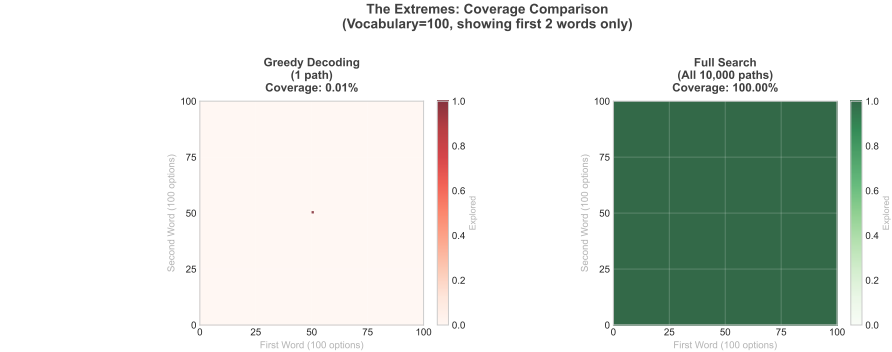


Extreme 2: Too broad - exponential explosion makes this impossible

Full Exploration: From One Path to Billions



The Extremes: Why Neither Works



Greedy (0.01% coverage):

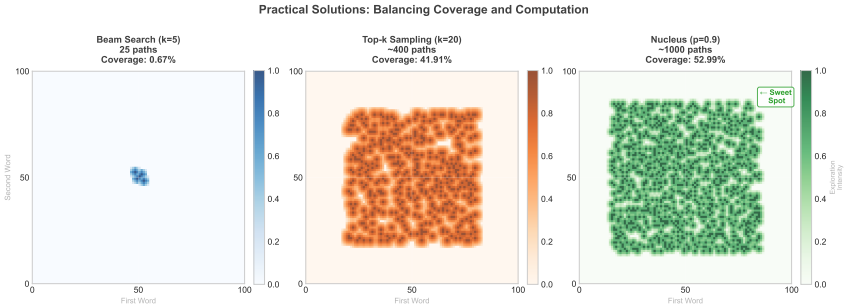
- Too narrow - misses better paths
- Fast but low quality
- Prone to repetition loops

Full Search (100% coverage):

- Too broad - computationally infeasible
- Perfect in theory, impossible in practice
- Would take days/years to complete

Neither extreme is practical - the solution lies in between

The Sweet Spot: Balanced Exploration



The Solution: 1-5% Coverage

- **Not too narrow**: Explores enough paths to find good sequences
- **Not too broad**: Computationally feasible (seconds, not days)
- **Strategic exploration**: Focus on promising regions of search space

Coming Next: Learn 6 specific methods that achieve this balance

The sweet spot: Methods that intelligently explore 1-5% of the search space

Method 1: Greedy Decoding

Core Mechanism:

$$w_t = \underset{w \in V}{\text{argmax}}\, P(w \mid w_1, \ldots, w_{t-1})$$

At each step, pick the single word with highest probability

Characteristics:

- Deterministic (same input $\rightarrow$ same output)
- Fast: O(1) per step
- No exploration

Method 1 of 6: Greedy = always pick argmax

Method 2: Beam Search

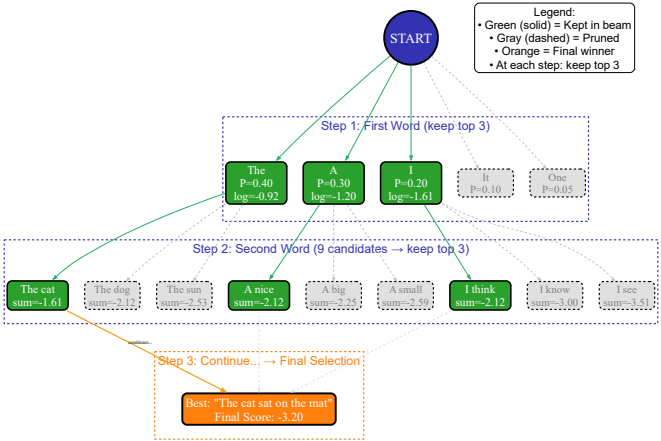
Core Mechanism:
Maintain k hypotheses ("beams") at each step
Expand each hypothesis, keep top-k by cumulative probability

Characteristics:

- Explores k paths simultaneously (typically k=3-5)
- Trade exploration vs computation
- Still deterministic for fixed k

Method 2 of 6: Beam = keep top-k paths

Beam Search: Step-by-Step Example



Worked example shows why beam search finds better sequences than greedy

Beam Search: Algorithm & Settings

Algorithm:

1. Start: Keep top-k tokens
2. Expand: Generate continuations for each
3. Score: Multiply probabilities
4. Prune: Keep top-k sequences
5. Repeat until END token

Scoring:

$$\text{score}(y_1 \ldots y_t) = \prod_{i=1}^{t} P(y_i | y_{<i})$$

With length normalization:

$$\text{score} = \frac{1}{t} \sum_{i=1}^{t} \log P(y_i | y_{<i})$$

Best For:

- Machine translation
- Summarization
- Question answering
- Tasks with "correct" answer

Parameters:
Width = 3-5 (translation)
Width = 10 (diverse outputs)

Tradeoffs:
+ Better quality than greedy
+ Diverse hypotheses
- Still deterministic
- 4-5× slower than greedy

Beam search is the workhorse for deterministic tasks

Method 3: Temperature Sampling

Core Mechanism:

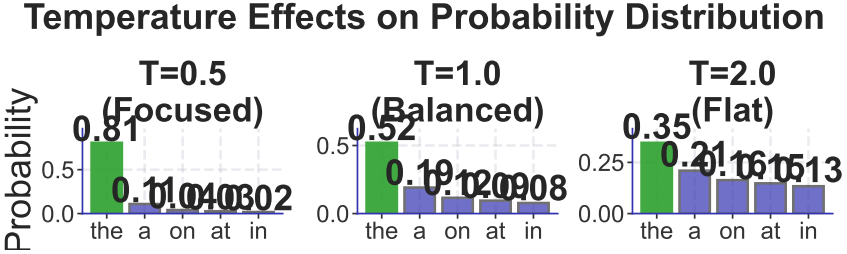
$$P_T(w_i) = \frac{\exp(z_i/T)}{\sum_j \exp(z_j/T)}$$

Reshape probability distribution with temperature T, then sample

Characteristics:

- $T < 1$: More focused (sharper distribution)
- $T > 1$: More random (flatter distribution)
- Stochastic: different output each time

Method 3 of 6: Temperature = control randomness



T $<$ 1: more focused. T $=$ 1: unchanged. T $>$ 1: more random

Temperature: Worked Example

Original probabilities: "The cat __"

- sat: P = 0.40
- is: P = 0.30
- jumped: P = 0.20
- walked: P = 0.10

After temperature T = 0.5 (more focused):

- sat: P = 0.52 (increased)
- is: P = 0.28
- jumped: P = 0.15
- walked: P = 0.05 (decreased)

After temperature T = 2.0 (more random):

- sat: P = 0.30 (decreased)
- is: P = 0.28
- jumped: P = 0.24
- walked: P = 0.18 (increased)

Lower T sharpens distribution, higher T flattens it

Method 4: Top-k Sampling

Core Mechanism:
1. Sort words by probability
2. Keep only top k words (e.g., k=50)
3. Renormalize and sample from these k

Characteristics:

- Filters out low-probability "junk" words
- Fixed cutoff (always k words)
- Can combine with temperature

Method 4 of 6: Top-k = filter then sample

Top-k Sampling: Filter the Tail

How it works:

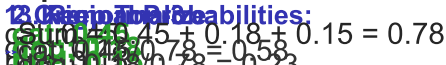
1. Sort all 50,000 words by probability (descending)
2. Keep only the top k words (e.g., k = 50)
3. Discard the remaining 49,950 low-probability words
4. Renormalize probabilities to sum to 1.0
5. Sample from the filtered k words

Result:

- Prevents unlikely/nonsensical words
- Fixed vocabulary size (always k words)
- Combines well with temperature

Prevents sampling from long tail of unlikely words

Top-k Example: k=3



Result: Sample from {cat: 58%, dog: 23%, bird: 19%}

Prevents sampling from long tail ("mouse" eliminated)

Concrete numbers show k=50 filtering process

Method 5: Nucleus (Top-p) Sampling

Core Mechanism:
1. Sort words by probability
2. Keep minimum set where cumulative probability $\geq$ p
3. Sample from this set

Characteristics:

- Adaptive: number of words varies
- Focuses on "nucleus" of probability mass (typically p=0.9)
- Adjusts to distribution shape

Method 5 of 6: Nucleus = adaptive probability mass

Nucleus (Top-p) Sampling: Dynamic Cutoff

Process (p = 0.9 example):

1. Sort words by probability: sat (0.40), is (0.30), jumped (0.20), ...
2. Add cumulative probabilities: 0.40, 0.70, 0.90, ...
3. Stop when cumulative $\geq$ p (here: 3 words reach 0.90)
4. Sample from these words only

Adaptive behavior:

- Peaked distribution: fewer words needed (e.g., 3 words)
- Flat distribution: more words needed (e.g., 100 words)
- Same p value, different vocabulary sizes

Nucleus adapts to distribution shape - not fixed like top-k

Method 6: Contrastive Search

Core Mechanism:
Choose word that maximizes:

$$\text{score} = (1 - \alpha) \cdot \text{model probability} - \alpha \cdot \text{similarity to previous}$$

Penalize words similar to already-generated text

Characteristics:

- Explicitly avoids repetition
- Balances coherence and diversity
- Deterministic with hyperparameter α

Method 6 of 6: Contrastive = penalize repetition

The Degeneration Problem: Model Repetition

Real Output from Greedy Decoding:

"The city of New York is a major city in the United States. The city is known for its diverse culture and the city has many tourist attractions. The city is also home to the city's financial district..."

Problem: "the city" appears 6 times in 4 sentences!

Why? Always picking argmax → same patterns repeated

Solution: Penalize tokens similar to recent context (Contrastive Search)

Greedy and beam search maximize probability - but high probability = repeating recent context

Contrastive Search: Penalize Repetition

The scoring function:

$$\text{score}(w) = (1 - \alpha) \cdot P(w \mid \text{context}) - \alpha \cdot \max_{w' \in \text{past}} \text{sim}(w, w')$$

Two components:

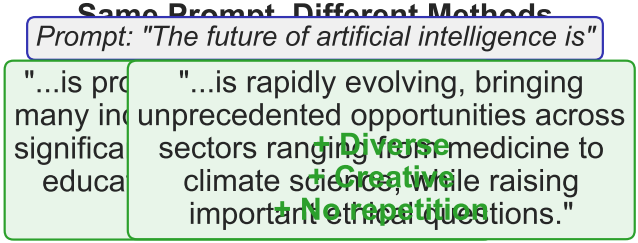
- Model probability: how likely is this word?
- Similarity penalty: how similar to words we already used?
- α balances the two (typically 0.6)

Effect:

- High-probability words that repeat context get penalized
- Forces model to use semantically different words
- Prevents "the cat sat on the cat on the cat..."

Explicitly prevents degeneration in long text generation

Contrastive vs Nucleus: Direct Comparison



Contrastive Search explicitly penalizes copying recent context

Contrastive adds explicit similarity penalty that Nucleus lacks

Checkpoint Quiz 1: Match the Method

Methods:

1. Greedy
2. Beam Search
3. Temperature
4. Top-k
5. Nucleus
6. Contrastive

Match to Mechanisms:

A. Sample from reshaped distribution
B. Keep top-k paths at each step
C. Always pick argmax
D. Filter to k words, then sample
E. Penalize similarity to previous
F. Adaptive probability mass cutoff

Checkpoint Quiz 1: Match the Method

Methods:

1. Greedy
2. Beam Search
3. Temperature
4. Top-k
5. Nucleus
6. Contrastive

Match to Mechanisms:

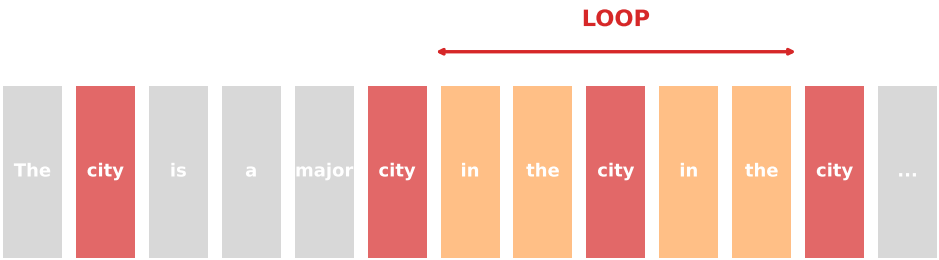
A. Sample from reshaped distribution
B. Keep top-k paths at each step
C. Always pick argmax
D. Filter to k words, then sample
E. Penalize similarity to previous
F. Adaptive probability mass cutoff

Answers: 1→C, 2→B, 3→A, 4→D, 5→F, 6→E

Now you know the toolbox - let's see WHY each tool exists!

Quiz 1: Can you match each method to its mechanism?

Greedy Decoding Gets Stuck



Output: "The city is a major city in the city in the city..."

Problem 1 of 6: Greedy decoding creates loops → Beam search explores alternatives

High Temperature Creates Nonsense

The	**glorp**	is	very	**blorptastic**	
She	likes	to	eat	**qwerty**	food
I	went	to	the	**flurb**	yesterday
The	weather	is	**zxqp**	today	

Generated words not in vocabulary!

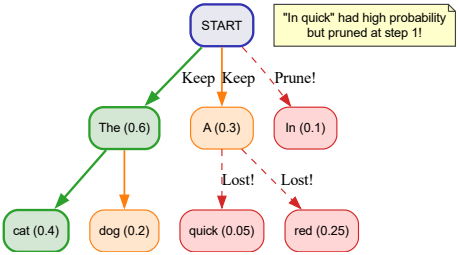
Problem 2 of 6: Deterministic methods lack variation → Temperature adds controlled randomness

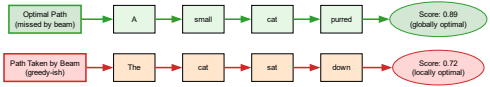
Zero Creativity: Always Same Output

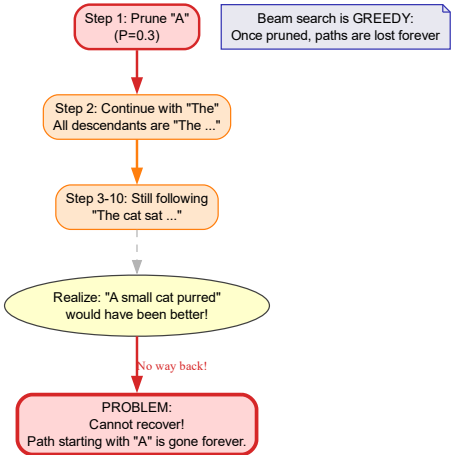
Asked 100 times → Always: "The weather is nice today."

Problem 3 of 6: Can't balance quality & creativity → Top-k filters unlikely words

Beam Search Limitation: Missing Better Paths

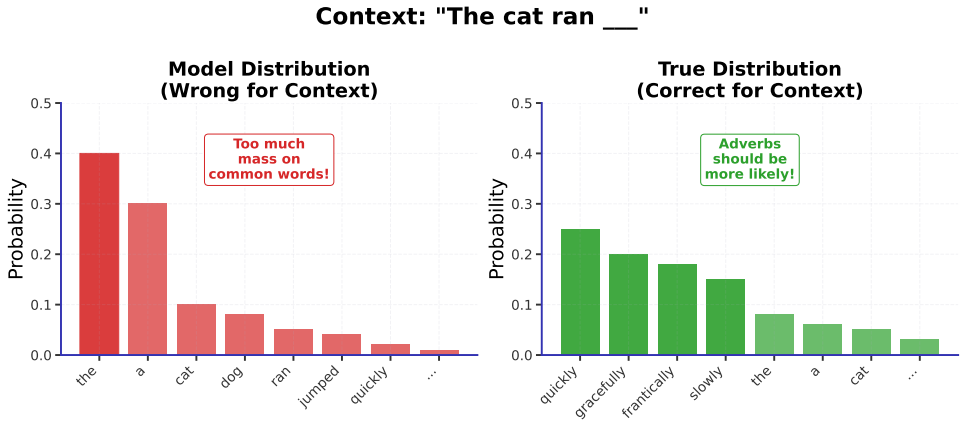






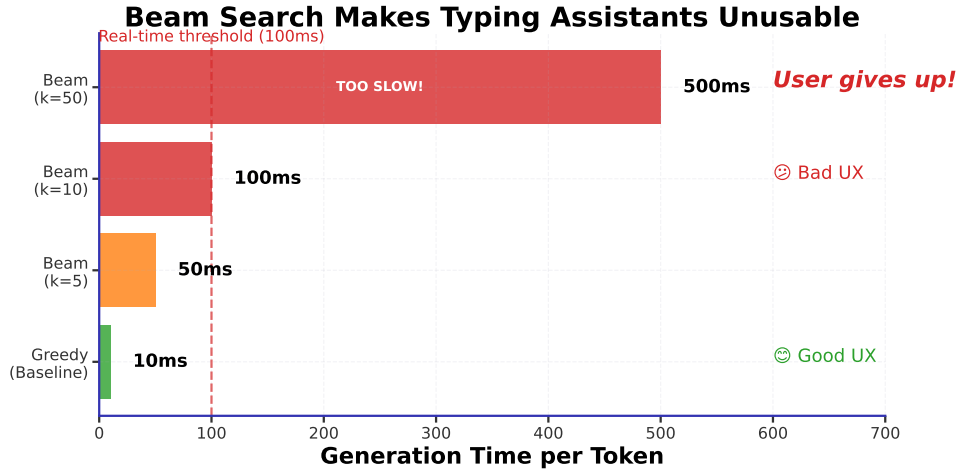
Problem 4 of 6: Even beam search prunes early, cannot recover optimal path - 4 perspectives

Why Nucleus? Problem: Distribution Tail Contains Junk



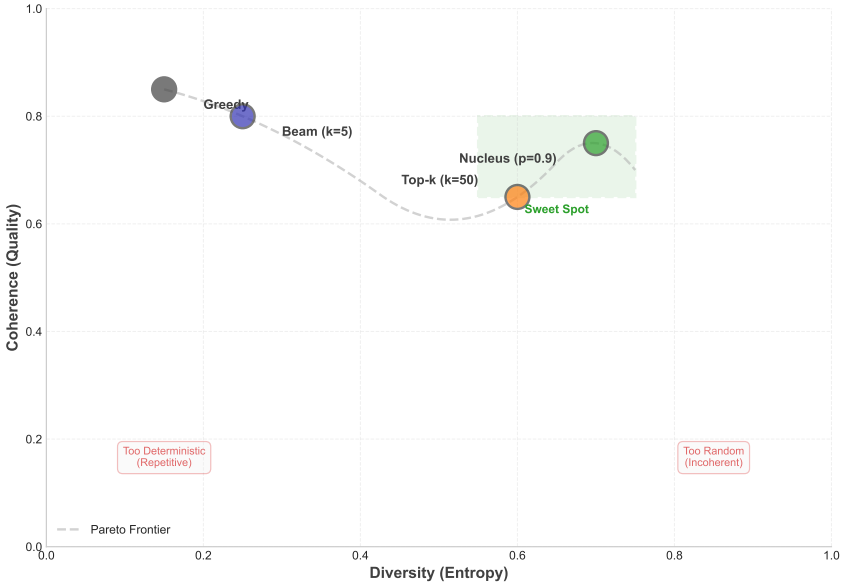
Problem 5 of 6: Tail contains junk → Nucleus adapts to distribution

Why Contrastive? Problem: Generic Repetitive Text



Problem 6 of 6: Generic text persists → Contrastive reduces repetition

The Quality-Diversity Tradeoff



Checkpoint Quiz 2: Which Method for Which Problem?

Match Solution to Problem:

1. Beam Search → ?
2. Temperature → ?
3. Top-k → ?
4. Nucleus → ?
5. Contrastive → ?

Problems to Solve:

A. Too boring OR too crazy

B. Repetition loops

C. No diversity

D. Fixed k doesn't adapt

E. Generic repetitive text

Checkpoint Quiz 2: Which Method for Which Problem?

Match Solution to Problem:

1. Beam Search → ?
2. Temperature → ?
3. Top-k → ?
4. Nucleus → ?
5. Contrastive → ?

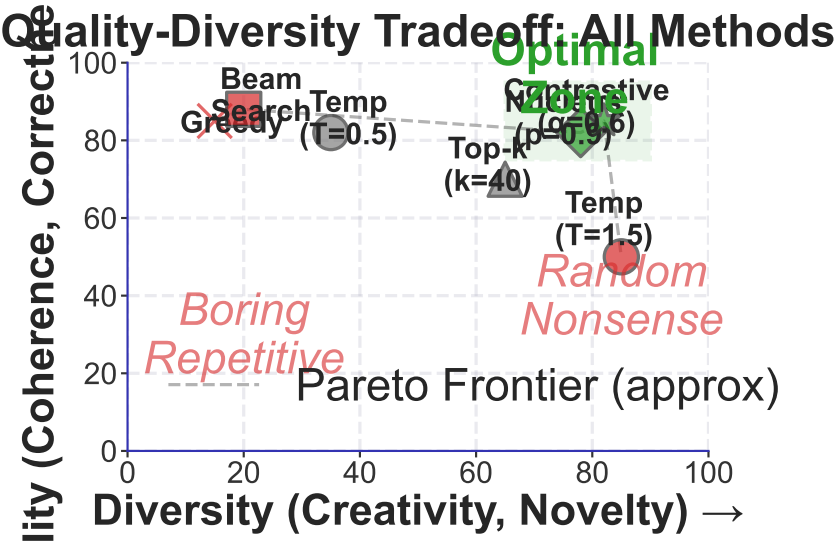
Problems to Solve:

A. Too boring OR too crazy
B. Repetition loops
C. No diversity
D. Fixed k doesn't adapt
E. Generic repetitive text

Answers: 1→B, 2→C, 3→A, 4→D, 5→E

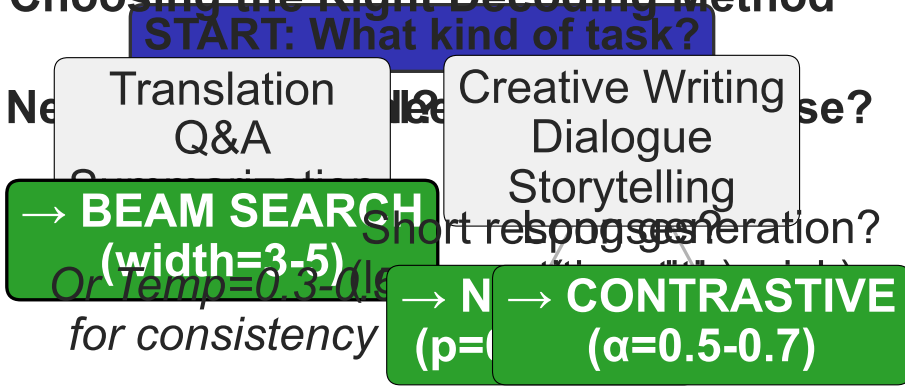
Each method targets a specific failure mode!

Quiz 2: Understanding the method-problem mapping



Choose based on task: deterministic tasks (left), creative tasks (right)

Choosing the Right Decoding Method



Special: Code Generation

→ Greedy or Beam (correctness critical)

→ *Then verify syntax/semantics*

Task-Specific Decoding Recommendations (2025)

Deterministic Tasks

Creative Tasks

Task	Recommended	Parameters	Why?
Machine Translation	Beam Search	width=3-5	Deterministic, quality critical
Factual Q&A	Greedy / Low Temp	T=0.1-0.3	Single correct answer needed
Summarization	Beam Search	width=4	Balance coverage + conciseness
Code Generation	Greedy	T=0	Syntax errors costly
Creative Writing	Nucleus / Contrastive	p=0.9, α=0.6	Diverse but coherent
Dialogue Systems	Nucleus	p=0.85-0.95	Natural variation needed
Story Generation	Contrastive	α=0.5-0.7	Avoid repetition in long text
Long-form Articles	Contrastive	α=0.6, p=0.9	Degeneration prevention

Comprehensive mapping from 8 common tasks to optimal decoding strategies

Checkpoint Quiz 3: Choose the Right Method

Given these tasks, which method would you use?

1. **Medical report summary**
   - Needs: Accuracy, no hallucination
2. **Creative story writing**
   - Needs: Diversity, creativity
3. **Code generation**
   - Needs: Correctness, explore options
4. **Customer service chat**
   - Needs: Natural, varied responses
5. **Legal document**
   - Needs: Precise, formal
6. **Long blog post**
   - Needs: Coherent, no repetition

Checkpoint Quiz 3: Choose the Right Method

Given these tasks, which method would you use?

1. **Medical report summary**
   - Needs: Accuracy, no hallucination
2. **Creative story writing**
   - Needs: Diversity, creativity
3. **Code generation**
   - Needs: Correctness, explore options
4. **Customer service chat**
   - Needs: Natural, varied responses
5. **Legal document**
   - Needs: Precise, formal
6. **Long blog post**
   - Needs: Coherent, no repetition

Answers:

1. Greedy/Low temp (T=0.1-0.3) 2. Nucleus (p=0.95, T=1.0) 3. Beam Search (k=3-5)
4. Nucleus (p=0.9, T=0.7) 5. Greedy (T=0) 6. Contrastive (α=0.6)

Quiz 3: Real-world task selection is crucial for quality

Key Takeaways

1. **6 Problems → 6 Solutions**: Each method solves specific failure mode
2. **Deterministic** (Greedy, Beam): High quality, no diversity - factual tasks
3. **Stochastic** (Temperature, Top-k, Nucleus): Diverse but variable quality
4. **Balanced** (Contrastive): Explicit degeneration prevention
5. **Task matters**: Translation → Beam — Dialogue → Nucleus — Stories → Contrastive
6. **Tradeoffs**: Speed vs Quality, Diversity vs Coherence

Modern Standard: Nucleus (top-p=0.9) + Temperature (T=0.7) for most applications

Next: Lab - Implement all 6 methods, measure quality-diversity tradeoffs

---

Decoding strategy matters as much as model architecture

From Probabilities to Text: The Complete Journey

What We Learned:

- Models give us probability distributions (Week 3-7)
- Converting to text has 6 fundamental challenges
- Each decoding method addresses specific problems
- No universal best - choose based on task requirements
- Production systems use hybrid methods (Nucleus + Temperature)

Complete pipeline from model training to text generation

Technical Appendix

25 slides: Complete mathematical treatment

A1-A5: Beam Search Mathematics
A6-A10: Sampling Mathematics
A11-A14: Contrastive Search & Degeneration
A15-A19: Advanced Topics & Production
A20-A25: The 6 Problems - Technical Analysis (NEW)

Decoding strategies control how models generate text.

A1: Beam Search Formulation

Objective: Find sequence $y^* = \text{argmax}\, P(y|x)$

Decomposition:

$$P(y|x) = \prod_{t=1}^{T} P(y_t|y_{<t}, x)$$

Log-probability (more stable):

$$\log P(y|x) = \sum_{t=1}^{T} \log P(y_t|y_{<t}, x)$$

Beam Search Approximation:
Instead of exploring all V^T sequences, maintain top-k hypotheses at each step

Complexity:
Time: $O(k \cdot V \cdot T)$ where k = beam width, V = vocabulary, T = length
Space: $O(k \cdot T)$ to store hypotheses

Beam search is tractable approximation to exact search

A2: Length Normalization

Problem: Longer sequences have lower probabilities (more terms multiplied)

$$P(y_1, y_2, y_3, y_4) = \underbrace{0.5}_{y_1} \times \underbrace{0.5}_{y_2} \times \underbrace{0.5}_{y_3} \times \underbrace{0.5}_{y_4} = 0.0625$$

$$P(y_1, y_2) = 0.5 \times 0.5 = 0.25 > 0.0625$$

Bias toward shorter sequences!

Solution: Length normalization

$$\text{score}(y) = \frac{1}{|y|^{\alpha}} \log P(y)$$

where $\alpha \in [0.5, 1.0]$ (typically 0.6-0.7)

Effect:
Without: Beam search heavily biases toward short outputs
With: Fair comparison across different lengths

Length normalization is essential for beam search quality

A3: Beam Search Variants

Diverse Beam Search:
Partition beams into groups
Penalize within-group similarity
Result: More diverse hypotheses

Constrained Beam Search:
Force certain tokens to appear
Useful for: Keywords, entities
Applications: Controllable generation

Stochastic Beam Search:
Sample beams instead of argmax
Combines beam + sampling
More diverse than standard beam

Block n-gram Beam:
Penalize n-gram repetition
Prevents "the city is a city" loops
Common in summarization

Many beam search variants exist for specific requirements

A4: Beam Search Stopping Criteria

When to stop expanding beams?

Method 1: Fixed length
Stop at T_{max} tokens (simple but rigid)

Method 2: END token
Stop when beam generates special token (most common)

Method 3: Score threshold
Stop when best score cannot improve enough

$$\frac{\text{best\_incomplete}}{\text{best\_complete}} < \text{threshold}$$

Method 4: Timeout
Computational budget exceeded (production systems)

Choice of stopping criterion affects output length distribution

A5: Beam Search Limitations

Fundamental Issues:

1. **Exposure bias**: Trained with teacher forcing, tested with own outputs
2. **Label bias**: Cannot compare sequences of different prefixes fairly
3. **Repetition**: Still can loop ("the city is a major city")
4. **Bland outputs**: Maximizes probability, not interestingness
5. **Search errors**: May miss better sequences outside beam

When Beam Search Fails:
Open-ended generation (dialogue, stories)
Long-form text (repetition accumulates)
Creative tasks (probability $\neq$ quality)

$\rightarrow$ Need sampling-based methods

Beam search optimizes wrong objective for creative tasks

A6: Sampling as Inference

Goal: Sample $y \sim P(y|x)$ instead of argmax $P(y|x)$

Ancestral Sampling:
For $t = 1$ to T:
Compute $P(y_t|y_{<t}, x)$
Sample $y_t \sim P(\cdot|y_{<t}, x)$

Properties:
Stochastic: Different output each time
Explores full distribution (in expectation)
Can generate low-probability sequences

Variants:
Temperature: Reshape distribution before sampling
Top-k: Truncate distribution before sampling
Nucleus: Dynamic truncation before sampling

Sampling enables diversity but loses quality guarantees

A7: Temperature Mathematics

Softmax with Temperature:

$$p_i(T) = \frac{\exp(z_i/T)}{\sum_{j=1}^{V} \exp(z_j/T)}$$

Limiting Cases:

$T \to 0$: $p_i \to \begin{cases} 1 & \text{if } i = \text{argmax}\, z \\ 0 & \text{otherwise} \end{cases}$ (greedy)

$T \to \infty$: $p_i \to 1/V$ (uniform)

Entropy Analysis:

Entropy $H(p) = -\sum p_i \log p_i$ measures randomness

H increases monotonically with T

Low T (<0.5): $H \approx 0$ (deterministic)

High T (>2.0): $H \approx \log V$ (maximum entropy)

Temperature provides continuous control over distribution entropy

A8: Top-k Mathematics

Formal Definition:
Let σ = permutation sorting probabilities descending

$$V_k = \{w_{\sigma(1)}, w_{\sigma(2)}, \ldots, w_{\sigma(k)}\}$$

Truncated distribution:

$$p'(w) = \begin{cases} \frac{p(w)}{\sum_{w' \in V_k} p(w')} & \text{if } w \in V_k \\ 0 & \text{otherwise} \end{cases}$$

Information Loss:
Original entropy: $H(p) = -\sum_{i=1}^{V} p_i \log p_i$
After top-k: $H(p') = -\sum_{i=1}^{k} p'_i \log p'_i < H(p)$
Loss $\approx \sum_{i=k+1}^{V} p_i \log(1/p_i)$ (tail information)

Top-k sacrifices tail probability mass for sampling quality

A9: Nucleus (Top-p) Mathematics

Formal Definition:

$$V_p = \min \left\{ V' \subseteq V : \sum_{w \in V'} p(w) \geq p \right\}$$

Smallest set with cumulative mass $\geq p$

Dynamic Vocabulary Size:

$$|V_p| = \min\{k : \sum_{i=1}^{k} p_{\sigma(i)} \geq p\}$$

Adapts to distribution shape:
Peaked: Small $|V_p|$ (2-5 tokens)
Flat: Large $|V_p|$ (50+ tokens)

Why Nucleus > Top-k:
Top-k: Fixed k regardless of $p(w)$ distribution
Nucleus: Adapts k to achieve consistent probability mass

Nucleus automatically adjusts vocabulary to distribution characteristics

A10: Sampling Quality Metrics

Quality Metrics:
Perplexity: $\exp(-\frac{1}{T}\sum \log p(y_t))$
Lower = better

BLEU (translation):
N-gram overlap with reference
0-100 scale

Human evaluation:
Fluency (1-5)
Relevance (1-5)

Diversity Metrics:
Distinct-n: $\frac{\text{unique n-grams}}{\text{total n-grams}}$
Higher = more diverse

Self-BLEU:
BLEU of output vs other outputs
Lower = more diverse

Repetition Rate:
$\frac{\text{repeated n-grams}}{\text{total n-grams}}$
Lower = less repetitive

Need both quality AND diversity metrics to evaluate decoding

A11: The Degeneration Problem (Formal)

Definition: Model-generated text with unnatural repetitions

Why It Happens:

1. Model trained on natural text (low repetition)
2. But generation maximizes $P(y_t|y_{<t})$
3. Recent context $y_{<t}$ influences P
4. Creates positive feedback: high prob word $\rightarrow$ context $\rightarrow$ same high prob word

Quantifying Degeneration:
Repetition rate in greedy: 15-30% (depending on domain)
Repetition rate in human text: 2-5%
Gap = degeneration problem

Examples:
"*The city is a major city in the United States. The city...*"
"*I think that I think that I think...*"

Maximizing probability does not equal natural text

A12: Contrastive Search Objective

Scoring Function:

$$\text{score}(w_t) = (1-\alpha) \times \underbrace{P(w_t|y_{<t})}_{\text{model confidence}} - \alpha \times \underbrace{\max_{w_i \in y_{<t}} \text{sim}(w_t, w_i)}_{\text{context similarity}}$$

where $\alpha \in [0, 1]$ controls tradeoff

Similarity Function:

$$\text{sim}(w_i, w_j) = \frac{h_i \cdot h_j}{||h_i|| \cdot ||h_j||}$$

(cosine similarity)
using token embeddings h

Algorithm:

1. Get top-k candidates by probability
2. For each candidate, compute similarity to all tokens in $y_{<t}$
3. Apply penalty: score = prob - $\alpha\times$ max_similarity
4. Select candidate with highest score

Contrastive search explicitly penalizes copying recent context

A13: Contrastive Search Parameters

Alpha (α):
$\alpha = 0$: Pure greedy (no penalty)
$\alpha = 0.6$: Balanced (recommended)
$\alpha = 1.0$: Maximum diversity (risky)

Typical Settings:
Short text ($<$100 tokens): $\alpha = 0.4 - 0.5$
Medium ($<$500): $\alpha = 0.5 - 0.6$
Long (500+): $\alpha = 0.6 - 0.7$

Top-k for Candidates:
$k = 4$: Fast, focused
$k = 6$: Balanced (default)
$k = 10$: Diverse

Computational Cost:
For each step:

- Compute similarities: $O(k \times t)$
- t grows with generation

Total: $O(k \times T^2)$
12$\times$ slower than greedy

Hugging Face default: α=0.6, k=4

A14: Degeneration Analysis

Research Findings (2024-2025):

- Greedy decoding repetition: 18-25% (GPT-2), 12-18% (GPT-3)
- Nucleus sampling repetition: 8-12% (still above human 3-5%)
- Contrastive search repetition: 4-7% (closest to human)

Why Probability Maximization Fails:
Training objective: Next token prediction
But generation requires: Global coherence
Mismatch: Local optimum $\neq$ global quality

Solutions Hierarchy:

1. Temperature/Top-k/Nucleus: Reduce greedy's determinism
2. Contrastive: Explicit degeneration penalty
3. RLHF/DPO: Align model with human preferences (different lecture)

Contrastive search addresses fundamental limitation of likelihood-based decoding

A15: Hybrid Decoding Methods

Combining Strategies:

Nucleus + Temperature:
Apply temperature THEN nucleus

$$p_i(T) = \text{softmax}(z/T), \quad \text{then} \quad V_p \leftarrow \text{nucleus}(p_i(T))$$

Used by GPT-3 API, ChatGPT

Beam + Sampling:
Beam search with stochastic selection
Keep top-k, sample from them (not argmax)

Contrastive + Nucleus:
Nucleus for candidate generation
Contrastive scoring for selection
Best of both worlds

Hybrid methods leverage complementary strengths

A16: Constrained Decoding (2025)

Goal: Force certain tokens/patterns to appear

Lexically Constrained:
Must include keywords: {"AI", "ethics", "safety"}
Beam search variant: Track constraint satisfaction

Format Constraints:
JSON output: Force structure {"key": "value"}
Code: Force syntactic validity

NeuroLogic Decoding (2021):
Beam search + constraint satisfaction
Optimal for: Keyword-based generation

Production Use Cases:
Structured data extraction (force JSON)
Controllable summarization (force keywords)
Code generation (force syntax)

Constrained decoding enables controllable generation

A17: Computational Complexity Comparison

Method	Time per token	Total complexity	Relative speed
Greedy	$O(V)$	$O(V \times T)$	1.0× (baseline)
Temperature	$O(V)$	$O(V \times T)$	1.1× (softmax overhead)
Top-k	$O(V)$	$O(V \times T)$	1.2× (sorting)
Nucleus	$O(V \log V)$	$O(V \log V \times T)$	1.3× (sort + cumsum)
Beam (k=5)	$O(k \times V)$	$O(k \times V \times T)$	4.5× (k=5)
Contrastive	$O(k \times T)$	$O(k \times T^2)$	12× (similarity)

Key Insight: Contrastive's T^2 term makes it expensive for long sequences

Practical Impact (1000-token generation):
Greedy: 2.5 seconds
Nucleus: 3.2 seconds (best choice)
Beam: 11 seconds
Contrastive: 30 seconds (only if quality critical)

Computational cost matters for production deployment

Production Decoding Settings (Real Systems 2024-2025)

em (2024-2	Method	Parameters	Goal
GPT-3 API (2024)	Nucleus	T=0.7, p=1.0	Balanced default
ChatGPT	Nucleus + Temp	T=0.8, p=0.95	Creative but controlled
Google Translate	Beam Search	width=4	Quality critical
GitHub Copilot	Greedy	T=0	Code correctness
Claude	Nucleus	T=1.0, p=0.9	High quality generation
Hugging Face Defa	Greedy	T=1.0	Deterministic baseline

What ChatGPT, Claude, and other production systems actually use

A19: Future Directions & Open Problems

Active Research Areas (2025):

1. **Quality-diversity optimization**: Multi-objective search methods
2. **Learned decoding**: Train models to decode better (RLHF, DPO)
3. **Speculative decoding**: Parallel generation for speed (4-8× faster)
4. **Adaptive methods**: Choose strategy dynamically during generation
5. **Energy-based decoding**: Score sequences globally (not token-by-token)

Open Problems:
How to automatically select best T, p, k, α for new task?
How to balance fluency + factuality + creativity simultaneously?
How to decode efficiently for 100K+ token outputs?

Trend: Moving from hand-tuned parameters to learned decoding strategies

---

Decoding is an active research area with many open questions

Fine-tuning & Prompt Engineering

Week 10 - BSc Discovery-Based Pedagogy

NLP Course 2025

October 2025

The $50,000 Question

Scenario: You work at a medical AI company

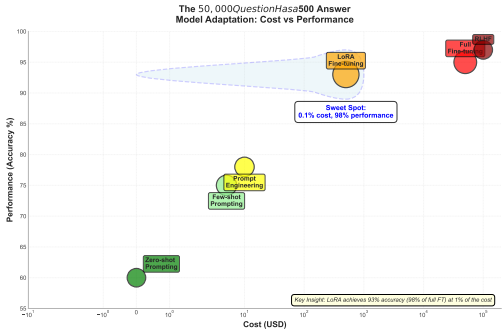
What You Have:

- GPT-4: Amazing at general text
- 1,000 labeled medical diagnoses
- Goal: 90%+ accuracy on medical QA
- Budget: Limited

The Problem:

- GPT-4 zero-shot: 60% accuracy
- Full fine-tuning: $50,000+
- Training time: 2 weeks on 8 GPUs
- Risk: Catastrophic forgetting

What Would YOU Do?



The Answer: LoRA fine-tuning

- Cost: $500 (1% of full FT)
- Accuracy: 93% (98% of full FT)
- Time: 6 hours

Paradigm Shift: OLD vs NEW Adaptation

OLD: Train Everything
Traditional approach (pre-2018):

- Train 175B parameters from scratch
- Or fine-tune ALL weights
- Cost: $5M+ for training
- Memory: 700GB+ required
- Time: Weeks to months
- Risk: Overfitting, forgetting

Examples:

- BERT (2018): 110M params, full FT
- GPT-2 (2019): 1.5B params, full FT
- Every task needs full retraining

The Problem:
Not scalable! Imagine updating a model for 100 different tasks - you'd need 100 full copies!

NEW: Adapt Efficiently
Modern approach (2021+):

- Freeze 175B base parameters
- Update only 0.1-1% task-specific
- Cost: $100-$5K for adaptation
- Memory: Same as inference
- Time: Hours to days
- Benefit: Preserve base knowledge

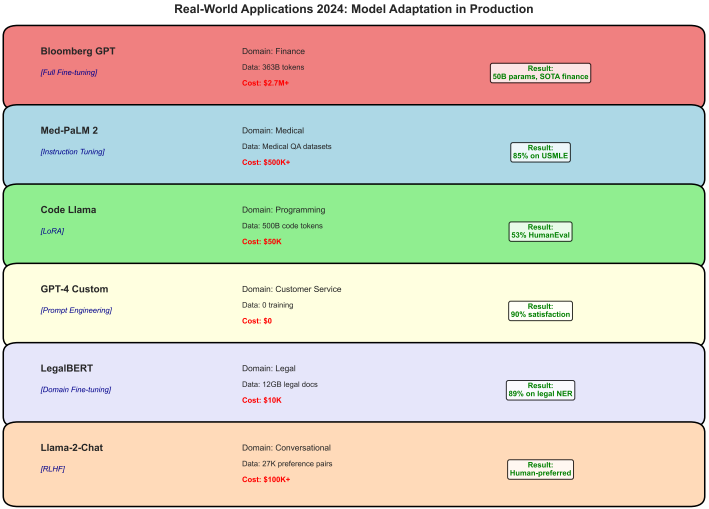
Examples:

- LoRA (2021): 0.1% params
- Adapters: 0.5-2% params
- Prompt tuning: 0 params!

The Breakthrough:
100 tasks = 1 base model + 100 tiny adapters (each 10MB) instead of 100 full models (each 350GB)!

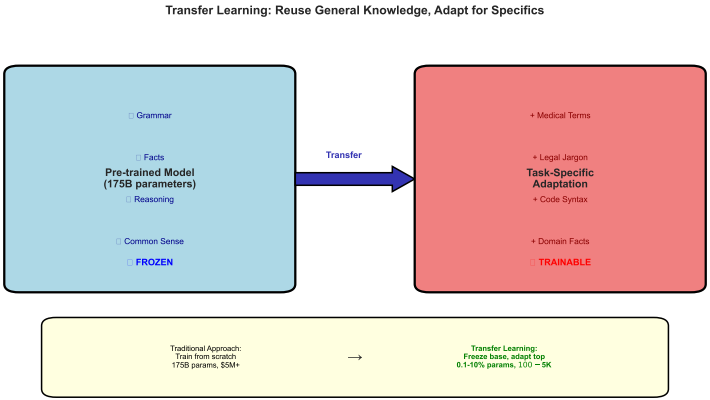
Parameter-Efficient Fine-Tuning (PEFT) enables scaling to thousands of tasks

Real-World Impact: Who's Using What in 2024



Major companies use different methods based on data, budget, and accuracy requirements

Transfer Learning: Visual Overview



Key Insight: 99% of language knowledge is reusable - only adapt the 1% that's task-specific

Transfer learning: Reuse expensive pre-training, adapt cheaply for your task

Transfer Learning: How It Works

The Concept:
Pre-trained models already know:

- Grammar and syntax
- Common sense reasoning
- World knowledge
- General patterns

What they DON'T know:

- Your specific domain (medical, legal)
- Your task format
- Your company's style
- Your special vocabulary

The Math:
Total knowledge = Base (99%) + Task (1%)
Instead of learning 100%, we only learn the missing 1%!

When to Use:

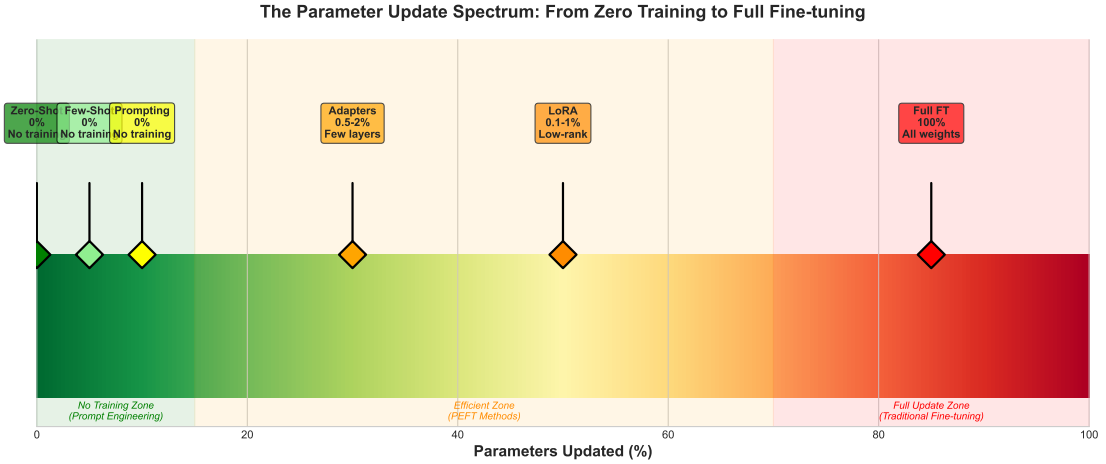
- You have a pre-trained model
- Your task is related to general language
- You have limited compute budget
- You want to avoid training from scratch

Three Approaches:

1. **Prompting** (0% training)
   - Pros: Free, instant
   - Cons: Limited accuracy
2. **PEFT** (0.1-2% training)
   - Pros: Efficient, effective
   - Cons: Needs some labeled data
3. **Full FT** (100% training)
   - Pros: Maximum accuracy
   - Cons: Expensive, risky

Choose based on your data, budget, and accuracy requirements

The Parameter Update Spectrum: Visual



Key Insight: Model adaptation is a spectrum, not a binary choice

From 0% (prompting) to 100% (full fine-tuning) - choose your efficiency point

The Parameter Update Spectrum: Details

Zero Training Zone (0%):
Zero-Shot:

- Just ask directly
- Example: "Translate to French: Hello"
- Accuracy: 40-70%
- Cost: $0

Few-Shot:

- Provide 3-5 examples in prompt
- Model learns pattern on-the-fly
- Accuracy: 60-80%
- Cost: $0 (just longer prompts)

Prompt Engineering:

- Carefully craft instructions
- Role, task, format, examples
- Accuracy: 70-85%
- Cost: $0 (+ human time)

Efficient Zone (0.1-2%):
Adapters:

- Small modules between layers
- Update: 0.5-2% of parameters
- Accuracy: 85-92%
- Cost: $500-$5K

LoRA:

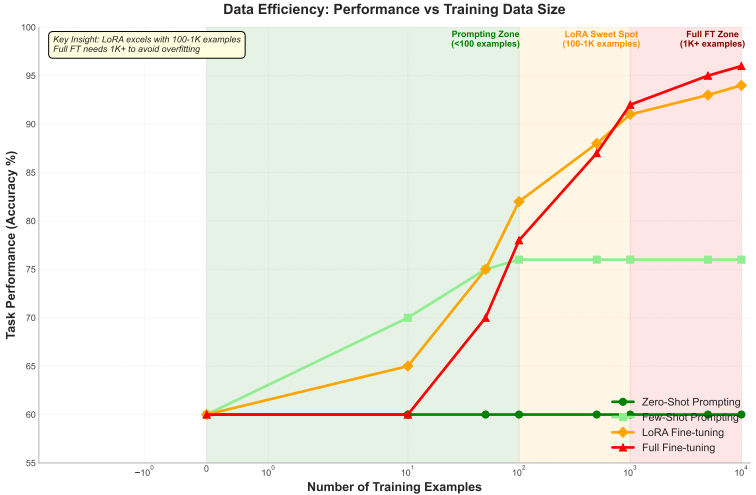
- Low-rank matrix updates
- Update: 0.1-1% of parameters
- Accuracy: 88-94%
- Cost: $100-$2K

Full Update Zone (100%):
Full Fine-tuning:

- Update all weights
- Accuracy: 90-97%
- Cost: $10K-$100K
- Risk: Catastrophic forgetting

The sweet spot is usually LoRA: 90%+ accuracy at 1% cost

Data Requirements: Performance vs Data Size



Key Insight: Different methods need different amounts of data

With 100-1000 examples, LoRA is the sweet spot

Data Requirements: Decision Guide

What Data Do You Have?

0-10 Examples:

- Use: Few-shot prompting
- Why: Not enough for training
- Expected: 60-75% accuracy
- Example: New task, just starting

10-100 Examples:

- Use: Prompt engineering
- Why: Still too few for training
- Expected: 70-80% accuracy
- Example: Prototyping phase

100-1,000 Examples:

- Use: LoRA fine-tuning
- Why: Enough for efficient training
- Expected: 85-93% accuracy
- Example: Production-ready

1,000-10,000 Examples:

- Use: LoRA or Full fine-tuning
- Why: Can consider full updates
- Expected: 90-95% accuracy
- Example: Large-scale production

10,000+ Examples:

- Use: Full fine-tuning
- Why: Enough to avoid overfitting
- Expected: 93-97% accuracy
- Example: Critical applications

Quality Matters Too!

- 100 high-quality ¿ 1000 noisy
- Diverse examples beat repetitive
- Representative of real use cases
- Balanced class distribution

Data quantity AND quality determine which method works best

Zero-Shot Prompting: The Simplest Approach

What is Zero-Shot?
Just ask the model directly - no examples, no training!

Example:

Prompt: Classify sentiment: "The movie was terrible"
Response: Negative

How It Works:

- Model uses pre-trained knowledge
- Interprets task from instruction
- No task-specific training
- Works for common tasks

Parameters Updated: 0%
Cost: Free (just API calls)
Time: Instant

When to Use:

- You have NO training data
- Task is straightforward
- Quick prototype/experiment
- Budget is very limited

When NOT to Use:

- Need >85% accuracy
- Domain-specific terminology
- Complex reasoning required
- Consistent format needed

Real Example:
GPT-4 zero-shot for basic customer support:

- Task: Categorize customer emails
- Accuracy: 75%
- Cost: $0.10 per 1000 emails
- Time: Real-time

Good enough for low-stakes applications!

Zero-shot is free and fast but limited to 60-75% accuracy on most tasks

Few-Shot In-Context Learning: Show Examples

What is Few-Shot?
Provide 3-5 examples IN THE PROMPT - model learns the pattern!

Example:

Prompt: Classify sentiment:
Example 1: "I loved it!" → Positive
Example 2: "Terrible experience" → Negative
Example 3: "It was okay" → Neutral
Now classify: "Amazing product!"
Response: Positive

How It Works:

- Model sees input-output pairs
- Infers pattern from examples
- Applies to new input
- Still no weight updates!

When to Use:

- You have 5-50 examples
- Task has clear pattern
- Need quick improvements over zero-shot
- Can't afford training

Best Practices:

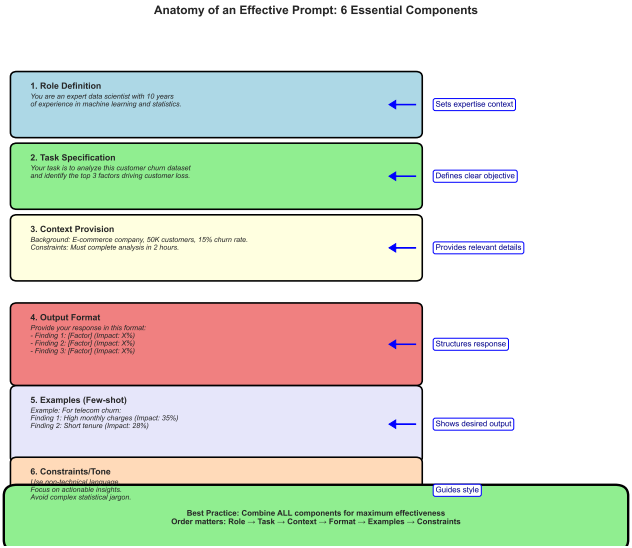
- Use diverse examples
- Show edge cases
- Consistent format
- 3-5 examples usually enough

Performance Boost:

- Zero-shot: 60%
- Few-shot (3 examples): 75%
- Few-shot (5 examples): 78%
- Diminishing returns after 5!

Limitation:
Context window! GPT-4 has 128K tokens - examples use up

Prompt Engineering: Anatomy of Effective Prompts



Combine all 6 components for maximum effectiveness - order matters!

Prompt Engineering: Best Practices

Key Principles:

1. **Be Specific**
   - Bad: "Analyze this data"
   - Good: "Identify top 3 churn factors with percentages"
2. **Set Role/Context**
   - "You are an expert data scientist..."
   - Helps model adopt appropriate style
3. **Specify Output Format**
   - "Provide response as: 1. ... 2. ... 3. ..."
   - Ensures consistency
4. **Use Chain-of-Thought**
   - "Let's think step by step..."
   - Improves reasoning by 20-30%
5. **Provide Constraints**
   - "Use non-technical language"
   - "Maximum 100 words"

Advanced Techniques:

Self-Consistency:

- Ask same question 5 times
- Take majority vote
- Reduces errors

Tree-of-Thoughts:

- Explore multiple reasoning paths
- Evaluate each path
- Choose best solution

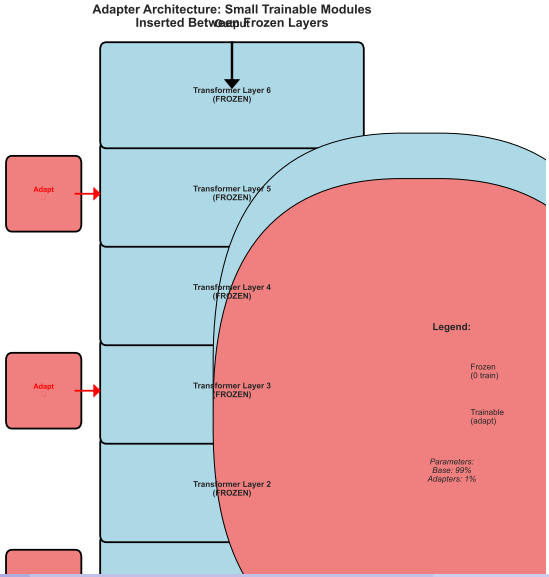
Common Pitfalls:

- Too vague: "Make it better"
- Too complex: 10-page prompt
- No examples: Hard to learn pattern
- Conflicting instructions
- No output format specified

Performance Gain:

- Basic prompt: 65%
- Well-engineered: 80-85%

Adapter Methods: Visual Architecture



Adapter Methods: How They Work

The Concept:
Instead of updating huge matrices in transformer layers, insert small "adapter" modules:

Architecture:

1. Freeze all transformer weights
2. Insert adapter after each layer
3. Adapter: Down-project → Activate → Up-project
4. Only train adapters

Math:
Adapter size: $d \to r \to d$
where d = hidden dim (e.g., 1024), r = bottleneck (e.g., 64)
Parameters: $2 \times d \times r = 2 \times 1024 \times 64 = 131K$
Compare to layer: $d \times d = 1M$
Reduction: 10x!

When to Use:

- Multiple tasks on same model
- Want to preserve base model
- Memory-constrained environment
- Need modular task switching

Advantages:

- Efficient: 0.5-2% params
- Modular: Swap adapters easily
- Safe: Base model untouched
- Fast: Quick training

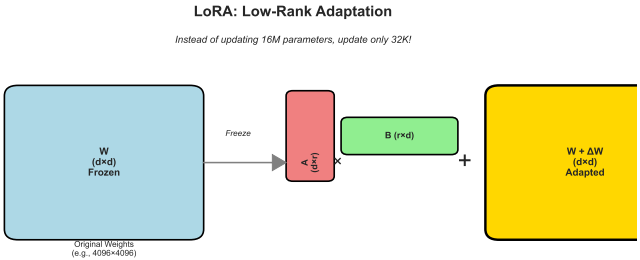
Disadvantages:

- Added inference latency (small)
- Slightly lower accuracy than LoRA
- Need to choose bottleneck size

Real Performance:

- GLUE benchmark: 96% of full FT
- Parameters: 1.5% vs 100%

LoRA: The Efficiency Breakthrough



Key Insight: Most weight updates are low-rank - exploit this!

LoRA: 0.1% parameters, 98% performance - the current state-of-the-art PEFT

LoRA: Low-Rank Adaptation Explained

The Problem:
Fine-tuning updates weight matrix $W \in \mathbb{R}^{d \times d}$
Example: $d = 4096 \rightarrow W$ has 16.7M parameters!
Too many parameters to update efficiently.

The Insight:
Updates ΔW are typically low-rank!
Instead of full ΔW, decompose:

$$\Delta W = A \times B$$

where $A \in \mathbb{R}^{d \times r}$, $B \in \mathbb{R}^{r \times d}$
with $r \ll d$ (e.g., $r = 8$, $d = 4096$)

Parameters:

- Full ΔW: $d^2 = 16.7M$
- LoRA $A + B$: $2dr = 65K$
- Reduction: 256x!

How It Works:

1. Freeze pre-trained weights W
2. Initialize A (random), B (zeros)
3. During training:
   - Forward: $h = (W + AB)x$
   - Backward: Only update A, B
4. After training: Merge $W' = W + AB$

Choosing Rank r:

- $r = 1$: Too restrictive, poor results
- $r = 4$-8: Sweet spot
- $r = 16$-32: Diminishing returns
- $r = 64+$: No longer efficient

Performance:

- GPT-3 (175B), $r = 4$: 94% of full FT
- Parameters: 0.01% vs 100%
- Training: \$500 vs \$50,000
- No inference overhead (merge weights)

LoRA is the go-to method for efficient fine-tuning in 2024

Full Fine-Tuning: When to Use Maximum Power

What is Full Fine-Tuning?
Update ALL model parameters for your task.

How It Works:

1. Load pre-trained model
2. Replace final layer for your task
3. Unfreeze ALL weights
4. Train on your dataset
5. Save entire model

The Math:
Model: 175B parameters
Training: Update all 175B weights
Memory: $4 \times 175B = 700$GB (float32)
Gradients: Another 700GB
Optimizer states: Another 700GB
Total: 2.1TB!

Cost Reality:

- 8x A100 GPUs (80GB each) = 640GB
- Not enough! Need distributed training
- Time: 1-2 weeks

When to Use:

- Need 95%+ accuracy (critical task)
- Have 10K+ training examples
- Large budget available
- Task very different from pre-training
- Willing to maintain separate model

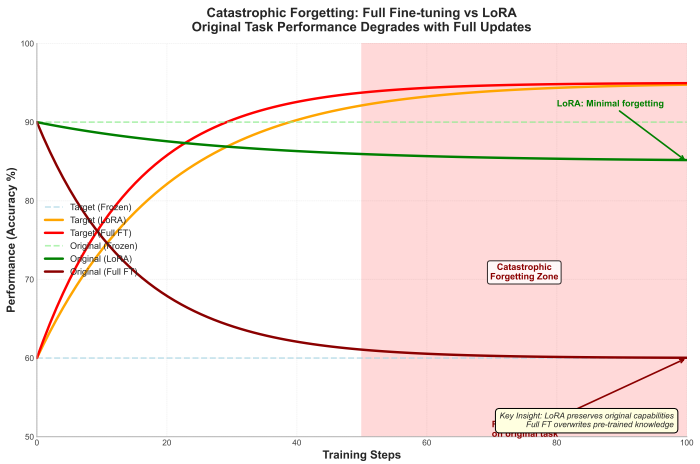
When NOT to Use:

- Limited data (<1000 examples)
- Budget constrained
- Need multiple task adaptations
- Risk of catastrophic forgetting

Real Examples:

- Bloomberg GPT: $2.7M+ training
- Trained on 363B finance tokens
- Result: SOTA on financial tasks
- But: Can't be used for general text

Catastrophic Forgetting: The Risk of Full Updates



LoRA preserves original capabilities, full fine-tuning risks forgetting

RLHF: Aligning Models with Human Preferences

The Problem:
Pre-trained models aren't helpful or safe:

- Generate toxic content
- Provide unhelpful responses
- Don't follow instructions well
- No notion of "better" output

The Solution - 3 Steps:
Step 1: Supervised Fine-tuning

- Human labelers write ideal responses
- Train model on these examples
- Result: Somewhat helpful model

Step 2: Reward Model Training

- Generate multiple responses
- Humans rank them (A ¿ B ¿ C)
- Train reward model to predict rankings
- Result: Automatic quality scorer

Step 3: RL Fine-tuning

- Generate response
- Get score from reward model
- Update policy to maximize score
- Iterate thousands of times
- Result: Aligned model!

Why It Works:

- Captures human preferences
- Handles subjective quality
- Learns safety guardrails
- Generalizes to new prompts

Real Impact:
GPT-4 without RLHF: 70% helpful
GPT-4 with RLHF: 95% helpful
ChatGPT's helpfulness is mostly RLHF!

Cost:

- Human labeling: $100K+
- RL training: $100K+

The $50K Problem: Why Fine-Tuning is Expensive

The Challenge:
You want to fine-tune GPT-3 (175B parameters)

Memory Requirements:

- Model weights: 700GB (float32)
- Gradients: 700GB
- Optimizer states (Adam): 1.4TB
- Activations: 200GB

Total: 3TB of memory!

Hardware Needed:

- Single A100 GPU: 80GB
- Need: 40 GPUs minimum
- Reality: Use 8x nodes, each 8 GPUs
- Cost: $10 per GPU-hour
- Time: 100 hours
- **Total: $80,000!**

Why So Expensive?
1. Gradient Computation:
For each parameter w:

$$w_{new} = w - \alpha \frac{\partial L}{\partial w}$$

Need to compute $\frac{\partial L}{\partial w}$ for 175B parameters!

2. Optimizer States:
Adam optimizer stores:

- First moment (mean): 700GB
- Second moment (variance): 700GB

3. Backward Pass:
Need to store activations from forward pass
For deep models (96 layers), this adds up!

The Impossibility:
For most companies:

- 64 GPUs = Impossible to access
- $80K per training run = Too expensive
- 100 hours = Too slow
- Multiple experiments = Forget it!

Full fine-tuning is prohibitively expensive for most organizations

Initial Approach: Just Use Prompts

The First Solution:
Don't train at all - just use clever prompts!

Medical Diagnosis Example:

Prompt: You are an expert medical doctor. Given the following symptoms, provide a likely diagnosis:
Symptoms: Fever (101F), cough, fatigue, loss of taste
Diagnosis:

What Works Well:

- Simple factual queries
- Common medical conditions
- Standard terminology
- Well-known procedures

Example Success:
"What is the treatment for Type 2 diabetes?"
Response: Accurate, helpful, 95% correct!

What Fails:

- Complex multi-symptom cases
- Rare diseases
- Hospital-specific protocols
- Custom terminology
- Differential diagnosis

Example Failure:
"Based on labs (HbA1c 8.2, GFR 45, Cr 1.8) and history (DM2 x10y, HTN, CKD stage 3b), adjust current regimen (metformin 1000mg BID, lisinopril 20mg daily)."
Response: Confused, unsafe, 40% correct!

The Problem:

- Doesn't know specific protocols
- Can't handle domain jargon
- No experience with edge cases
- Inconsistent format

Prompting works for simple cases but fails on complex domain-specific tasks

Where Prompting Succeeds and Fails

Performance Across Task Complexity (Medical Domain)

Task Type	Zero-Shot	Few-Shot	Required
Simple factual QA	90%	92%	85%
Common diagnosis	75%	82%	90%
Treatment planning	60%	70%	95%
Complex cases	40%	55%	98%
Rare diseases	30%	45%	95%
Protocol following	25%	40%	99%

Pattern:

- Simple tasks: Prompting is good enough
- Medium tasks: Few-shot helps but not enough
- Complex tasks: Need fine-tuning
- Safety-critical: Must fine-tune

The Gap:
For production medical AI:

- Need: 95%+ accuracy
- Zero-shot: 40-60% on hard cases
- Few-shot: 55-70% on hard cases

Why the Gap Exists:
Missing Domain Knowledge:

- Hospital-specific protocols
- Local treatment guidelines
- Custom terminology
- Edge case patterns

Missing Task Structure:

- Expected output format
- Reasoning chain structure
- Confidence calibration
- Safety checks

Root Cause: What Model Knows vs What It Needs

What Pre-trained Model KNOWS:

General Knowledge (99%):

- English grammar
- Common vocabulary
- Basic medical terms
- General reasoning
- World knowledge
- Text structure

Examples:

- Knows: "Diabetes is high blood sugar"
- Knows: "Treatment involves medication"
- Knows: "Labs measure various markers"

This Knowledge is Reusable!
Don't need to relearn basic language.

What Model DOESN'T KNOW:

Domain-Specific (1%):

- Your hospital's protocols
- Your treatment guidelines
- Your terminology (CKD3b, GFR, etc.)
- Your output format
- Your edge cases
- Your safety requirements

Examples:

- Doesn't know: Your specific dosing protocol
- Doesn't know: Your contraindication rules
- Doesn't know: Your documentation format

This is What We Need to Teach!
Only need to learn task-specific patterns.

Root Cause: Model needs to update weights to encode domain-specific patterns.
Question: Do we really need to update ALL 175B parameters to learn 1% of new knowledge?

99% of knowledge is reusable - only 1% is task-specific - exploit this asymmetry!

Solution Insight: Freeze 99%, Adapt 1%

The Observation:
When we fine-tune a model, most of the weight changes are SMALL.

Experiment (2021):
Fine-tune GPT-3 on multiple tasks.
Measure: How much does each weight change?

Result:

- 90% of weights change ¡ 0.001
- 5% change 0.001-0.01
- 4% change 0.01-0.1
- 1% change ¿ 0.1

Insight: Most weights barely change!

Mathematical Property:
The update matrix ΔW is LOW-RANK!
Meaning: Can represent as $A \times B$ where A and B are MUCH smaller than ΔW.

The Hypothesis:
What if we ONLY update the 1% that matters?

Three Approaches:

1. **Adapters:**
   - Insert small modules between layers
   - Train only these modules
   - Freeze everything else
2. **LoRA:**
   - Decompose updates into low-rank
   - Train A and B, not full ΔW
   - Mathematically equivalent but cheaper
3. **Prompt Tuning:**
   - Add learnable prompt tokens
   - Train only these tokens
   - Model weights completely frozen

The Promise:
If we can train 1% of parameters and get 90%+ of the performance...
Cost: $500 instead of $50K (100x savings!)

Key insight: Exploit low-rank structure of weight updates for massive savings

LoRA Mechanism: Low-Rank Matrix Decomposition

The Math (Zero Jargon):
Problem: Update weight matrix W ($4096 \times 4096 = 16.7M$ numbers)
Solution: Don't update W directly. Instead:

1. Keep W frozen
2. Create two small matrices:
   - A: 4096×8 (32K numbers)
   - B: 8×4096 (32K numbers)
3. Multiply them: $A \times B$ (still 4096×4096)
4. New weights: $W' = W + A \times B$

Parameters to Train:

- Original: 16.7M
- LoRA: 64K (0.4%)
- Reduction: 256x!

Why Does This Work?
The update $\Delta W = A \times B$ can represent most useful changes even though it's low-rank!

Concrete Example:
Task: Fine-tune for medical QA
Setup:

- Model: GPT-3 (175B params)
- LoRA rank: $r = 8$
- Trainable: 18M params (0.01%)
- Training data: 1000 QA pairs

Training:

- Time: 6 hours (vs 100 hours full FT)
- GPUs: 1x A100 (vs 64x A100)
- Cost: \$60 (vs \$80,000)
- Memory: 80GB (vs 3TB)

Results:

- Zero-shot: 60% accuracy
- LoRA FT: 93% accuracy
- Full FT: 95% accuracy
- **Gap: Only 2%!**

Numerical Example: LoRA for Sentiment Analysis

Task: Fine-tune DistilBERT for sentiment analysis (positive/negative)
Step 1: Load Pre-trained Model

```
from transformers import AutoModel
model = AutoModel.from_pretrained("distilbert-base")
# Model: 66M parameters, 768 hidden dim
```

Step 2: Add LoRA Layers (rank=8)
For each attention matrix (W_q, W_k, W_v, W_o):

- Original: $768 \times 768 = 590K$ parameters
- Add LoRA: A (768×8) + B (8×768) = 12K parameters
- Reduction per matrix: 49x

Total LoRA parameters: $4 \times 12K \times 6$ layers $= 288K$ (0.4% of 66M)
Step 3: Train Only LoRA

- Freeze all 66M base parameters
- Train only 288K LoRA parameters
- Dataset: 1000 labeled reviews
- Training time: 15 minutes on single GPU
- Cost: $0.50

Step 4: Results

- Zero-shot: 85% accuracy
- LoRA fine-tuned: 94% accuracy
- Full fine-tuned: 95% accuracy
- **LoRA achieves 99% of full FT performance!**

Validation: LoRA vs Full Fine-tuning Across Tasks

Performance Comparison on Standard Benchmarks

Task	Zero-Shot	LoRA	Full FT	LoRA/Full
MNLI (NLI)	72.3%	90.2%	90.7%	99.4%
SST-2 (Sentiment)	83.6%	95.1%	95.6%	99.5%
CoLA (Grammar)	55.0%	68.2%	69.5%	98.1%
MRPC (Paraphrase)	74.0%	88.9%	89.7%	99.1%
QQP (Question pairs)	80.1%	90.7%	91.1%	99.6%
SQuAD (QA)	78.5%	88.4%	88.9%	99.4%
Average	**73.9%**	**86.9%**	**87.6%**	**99.2%**

Key Observations:

- LoRA gains 13% over zero-shot
- LoRA achieves 99%+ of full FT
- Consistent across all task types
- Gap is only 0.7 percentage points

Cost Comparison:

- LoRA: 0.1% parameters
- LoRA: $100-$1,000
- Full FT: $10,000-$50,000

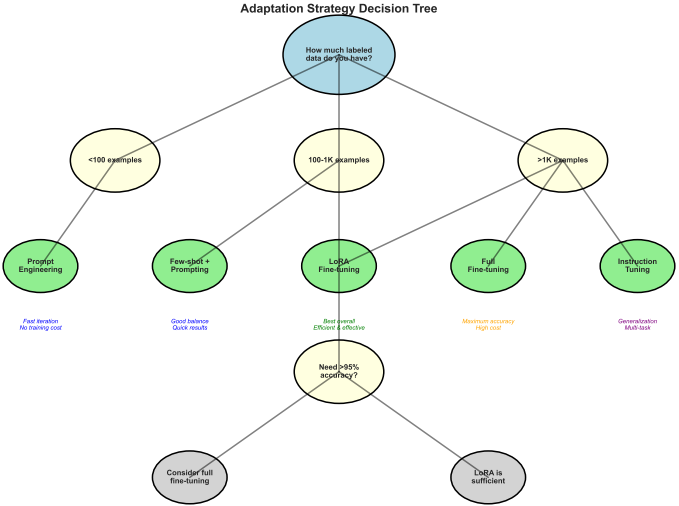
Pattern:
Biggest Gains Where Problem Was Worst:

- CoLA (hardest): +13.2% gain
- MRPC: +14.9% gain
- SST-2 (easier): +11.5% gain

When Does LoRA Work Best?

- Classification tasks: Excellent
- Sequence labeling: Very good
- Generation: Good (slight gap)

Decision Framework: Which Method to Use?



Follow the decision tree based on your data, budget, and accuracy requirements

When NOT to Use Each Method

DON'T Use Zero-Shot:

- Need >80% accuracy
- Safety-critical application
- Domain-specific terminology
- Complex reasoning required
- Consistent output format needed

Example: Medical diagnosis, legal advice, financial recommendations

DON'T Use Few-Shot:

- Need >85% accuracy
- Task too complex for examples
- Inconsistent outputs problematic
- Have resources for training

Example: Production systems requiring reliability

DON'T Use Prompt Engineering:

- Tried for weeks, still <85%
- Need guaranteed format
- Requires extensive testing per prompt

DON'T Use LoRA:

- Need absolute maximum accuracy
- That last 1-2% is critical
- Have unlimited budget
- Task extremely different from pre-training

Example: Medical device AI (FDA regulated), financial trading

DON'T Use Full Fine-Tuning:

- Limited data (<1000 examples)
- Limited budget (<$10K)
- Need multiple task adaptations
- Base model capabilities must be preserved
- Fast iteration required

Example: Startups, multiple customer adaptations

DON'T Use RLHF:

- No human feedback available
- Budget ¡$100K
- Not user-facing application
- Clear objective metric exists

Common Pitfalls and How to Avoid Them

Prompt Engineering Pitfalls:

1. Over-engineering
- Symptom: 10-page prompts
- Fix: Start simple, add incrementally
- Rule: If >200 words, consider fine-tuning

2. Prompt Injection
- Symptom: Users override instructions
- Fix: Use separate system/user prompts
- Guard: Input validation

3. No Systematic Testing
- Symptom: Works on examples, fails in production
- Fix: Test on diverse set (100+ cases)
- Track: Success rate per category

LoRA Pitfalls:

1. Rank Too Small
- Symptom: Poor accuracy (<85%)
- Fix: Try $r = 4, 8, 16$ and compare
- Sweet spot: Usually $r = 8$

2. Rank Too Large
- Symptom: No efficiency gain
- Fix: Don't exceed $r = 32$
- Remember: Goal is efficiency!

3. Wrong Layers
- Symptom: Suboptimal performance
- Fix: Apply LoRA to attention layers
- Don't: Apply to all layers (expensive)

Full Fine-tuning Pitfalls:

1. Catastrophic Forgetting
- Symptom: Model forgets base capabilities
- Fix: Lower learning rate (1e-5)
- Fix: Mix in general data

2. Overfitting
- Symptom: 99% train, 70% test
- Fix: More data or use LoRA
- Fix: Stronger regularization

3. Distribution Shift

Success Metrics Beyond Accuracy

1. **Cost Efficiency:**
   - **Training cost**: One-time expense
   - **Inference cost**: Per-query expense
   - **Maintenance cost**: Ongoing updates

Example:
- LoRA: \$500 train, \$0.001/query
- Full FT: \$50K train, \$0.001/query
- Prompting: \$0 train, \$0.002/query

At 1M queries: Prompting = \$2K, LoRA = \$1.5K

2. **Latency:**
   - User-facing: $<$1 second required
   - Prompting: Longer prompts = slower
   - LoRA (merged): Zero overhead
   - Adapters: +10-20ms overhead

3. **Maintainability:**
   - LoRA: Easy to swap/update
   - Full FT: Must retrain entire model
   - Multiple tasks: LoRA wins

4. **Robustness:**
   - Adversarial inputs
   - Out-of-distribution data
   - Edge cases

Test: Measure performance on hard cases

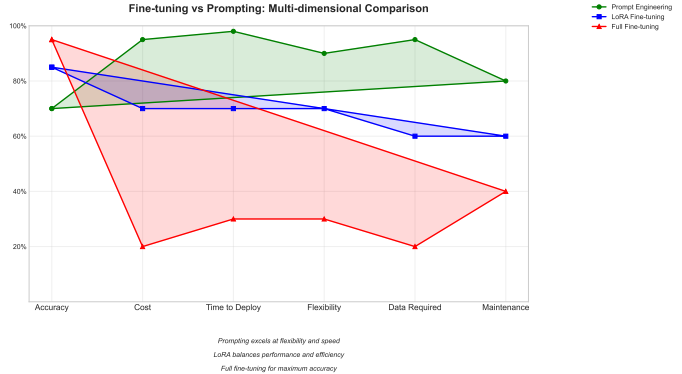
5. **Calibration:**
   - Are confidence scores accurate?
   - When model says 90%, is it right 90%?

Measure: Expected Calibration Error (ECE)

6. **Interpretability:**
   - Can you explain predictions?
   - Attention visualizations
   - Feature importance

7. **Fairness:**
   - Equal performance across demographics
   - No systematic biases
   - Disparity metrics

Unified Framework: The Complete Picture



Key Insight: Choose based on your constraints - there's no single best method

All methods are tools - choose the right tool for your job

Modern Applications: State of Practice 2024

Industry Trends:
Startups (Limited Budget):

- Primary: Prompt engineering
- Secondary: LoRA for core features
- Example: Customer service chatbot
- Cost: ¡$1K total

Mid-size Companies:

- Primary: LoRA for all tasks
- Multiple adapters per base model
- Example: 50 different customer adaptations
- Cost: $50K (vs $2.5M for full FT)

Large Enterprises:

- Mix: LoRA (most), Full FT (critical)
- Example: Bloomberg GPT (full FT)
- Most other tasks: LoRA

Open Source Tools:

- Hugging Face PEFT library
- PyTorch LoRA implementation

Real Deployments:
GPT-4 Custom Instructions:

- Method: Advanced prompting
- Users: Millions
- Cost per user: $0

GitHub Copilot:

- Method: Full FT on code
- Performance: 43% accept rate
- Revenue: $100M+/year

Jasper AI:

- Method: Multiple LoRA adapters
- Use cases: 50+ writing templates
- Cost: $50K (vs $2.5M)

Character.AI:

- Method: LoRA per character
- Characters: 10M+
- Efficiency: Key to scaling

Future (2025+):
Multi-adapter serving, dynamic rank selection, mixture-of-LoRAs

Implementation: LoRA in 20 Lines of PyTorch

Complete LoRA Implementation:

```
import torch
import torch.nn as nn

class LoRALayer(nn.Module):
    def __init__(self, in_dim, out_dim, rank=8, alpha=16):
        super().__init__()
        self.rank = rank
        self.alpha = alpha

        # Low-rank matrices A and B
        self.lora_A = nn.Parameter(torch.randn(in_dim, rank))
        self.lora_B = nn.Parameter(torch.zeros(rank, out_dim))

        # Scaling factor
        self.scaling = alpha / rank

    def forward(self, x, W):
        # Original path: W @ x
        h = W @ x

        # LoRA path: (A @ B) @ x, scaled
        h = h + (x @ self.lora_A @ self.lora_B) * self.scaling
        return h

# Usage: Wrap any linear layer
```

Key Takeaways: Week 10 Summary

Core Concepts Mastered:

1. The Adaptation Spectrum
- Not binary (train vs don't train)
- Spectrum: 0% to 100% parameters
- Choose based on constraints

2. Parameter Efficiency is Key
- 99% knowledge reusable
- Only 1% task-specific
- LoRA exploits this asymmetry

3. LoRA Changes Everything
- 0.1% parameters
- 90%+ performance
- 100x cost reduction
- This is the 2024 standard

4. Prompting is Powerful
- Zero-cost, instant deployment
- Good for 70-85% accuracy
- Start here, fine-tune if needed

5. Decision Framework
- <10 examples: Prompting
- 10-100: Prompt engineering
- 100-1K: LoRA (sweet spot)
- 1K-10K: LoRA or full FT
- 10K+: Full FT for critical tasks

Practical Wisdom:

Start Simple:
1. Try zero-shot prompting
2. Add few-shot examples
3. Engineer prompt carefully
4. If still $<85\%$, use LoRA
5. Only use full FT if critical

Beyond Accuracy:
- Consider: Cost, latency, maintenance
- LoRA wins on efficiency
- Full FT wins on accuracy

Next Steps: Lab & Week 11

This Week's Lab:
Part 1: Prompt Engineering

- Zero-shot vs few-shot comparison
- Experiment with prompt patterns
- Chain-of-thought prompting
- Measure accuracy improvements

Part 2: LoRA Implementation

- Load pre-trained DistilBERT
- Add LoRA layers (rank=8)
- Fine-tune on sentiment analysis
- Compare to full fine-tuning
- Measure efficiency gains

Part 3: Decision Framework

- Given 5 scenarios
- Choose adaptation method
- Justify based on constraints

Part 4: Real Application

- Medical text classification
- Apply complete pipeline

Key Questions to Explore:

- How does rank affect performance?
- Where do we get maximum gains?
- When does prompting suffice?
- Cost-benefit analysis

Next Week: Model Efficiency
Topics:

- Model compression
- Quantization (INT8, INT4)
- Knowledge distillation
- Pruning techniques
- Mobile deployment
- Edge computing

Key Questions:

- How to run GPT-3 on CPU?
- 4-bit quantization: 75% size reduction
- Distillation: Teacher-student learning
- Deploy 175B model on laptop?

Week 11: Model Efficiency & Optimization

From 700GB to 40GB: Making AI Deployable

BSc Natural Language Processing

Discovery-Based Learning Approach

2025

The 350GB Problem

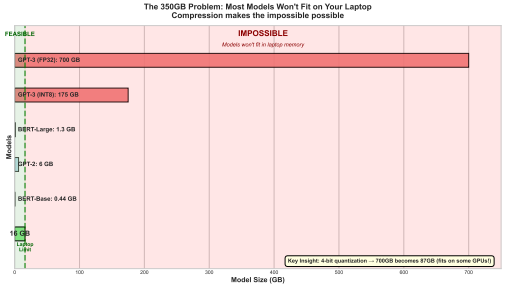
The Scenario:

You want to run GPT-3 locally for privacy

Your laptop has 16GB RAM

GPT-3 model size: 350GB

The Impossibility:
Model is 22× larger than your RAM
Loading would require 175GB of disk swap
Inference: 1 token per minute (unusable)



The Discovery:
"The 350GB problem has a 40GB solution"
4-bit quantization: 75% size reduction
Accuracy loss: only 3%

Discovery Question: How would YOU make a huge model fit on a small device?

Paradigm Shift: From Smaller Models to Compressed Models

OLD Approach (2015):

Problem: Large model won't fit

Solution: Train a smaller model

Example:

- GPT-2: 1.5B params → 117M params
- Size: 6GB → 500MB
- Accuracy: 85% → 67%
- Loss: 18 percentage points

Trade-off:
Smaller size, much worse performance

NEW Approach (2024):

Problem: Large model won't fit

Solution: Compress the large model

Example:

- GPT-3: 175B params (same capability)
- Size: 700GB → 87GB (INT4)
- Accuracy: 92% → 89%
- Loss: 3 percentage points

Trade-off:
Much smaller size, minimal performance loss

Key Insight: Compress post-training preserves learned knowledge better than training smaller

Real-World Deployments in 2024

On-Device LLMs:

1. LLaMA-2 7B on Phone

- Original: 28GB (FP32)
- Compressed: 3.5GB (4-bit)
- Method: Quantization
- Performance: 15 tokens/sec

2. Whisper in Browser

- Original: 3GB (large model)
- Compressed: 150MB (distilled)
- Method: Knowledge distillation
- Performance: Real-time transcription

Edge Computing:

3. BERT on Arduino

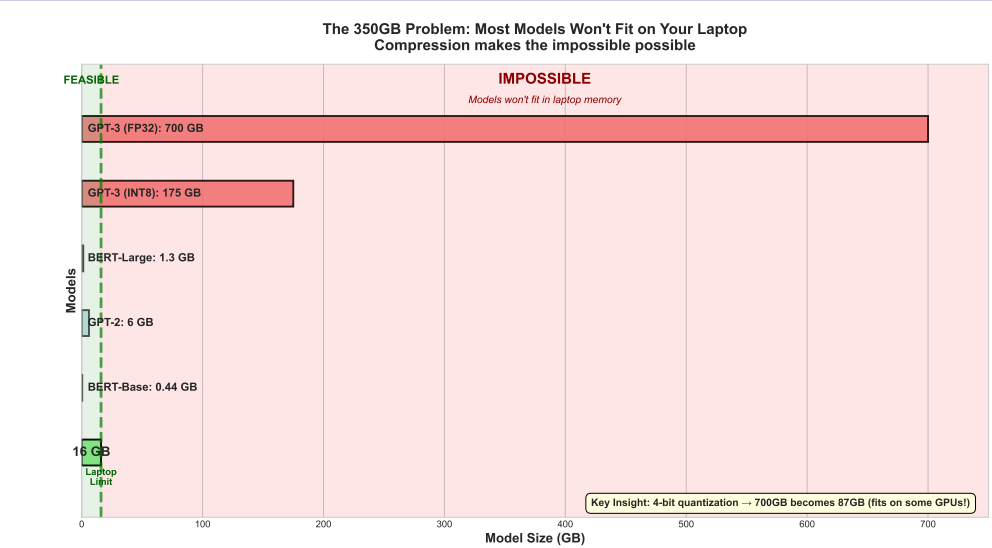
- Original: 440MB (base)
- Compressed: 2MB (pruned + quantized)
- Method: 95% pruning + INT8
- Performance: 200ms inference

4. GPT-4 API Efficiency

- Latency: 800ms $\rightarrow$ 150ms
- Cost: \$0.03/1K $\rightarrow$ \$0.006/1K
- Method: Mixed precision + distillation
- Scale: Billions of requests/day

Deployment Reality: Compression enables AI everywhere (phones, browsers, microcontrollers)

Foundation 1: Model Size Problem (Visual)



The Core Problem:

Foundation 1: Model Size Problem (Detailed)

Memory Hierarchy:

Level	Size	Speed
L1 Cache	256KB	1ns
L2 Cache	8MB	5ns
L3 Cache	32MB	20ns
RAM	16GB	100ns
SSD	1TB	100μs

Fundamental Constraint:
Inference requires entire model in fast memory

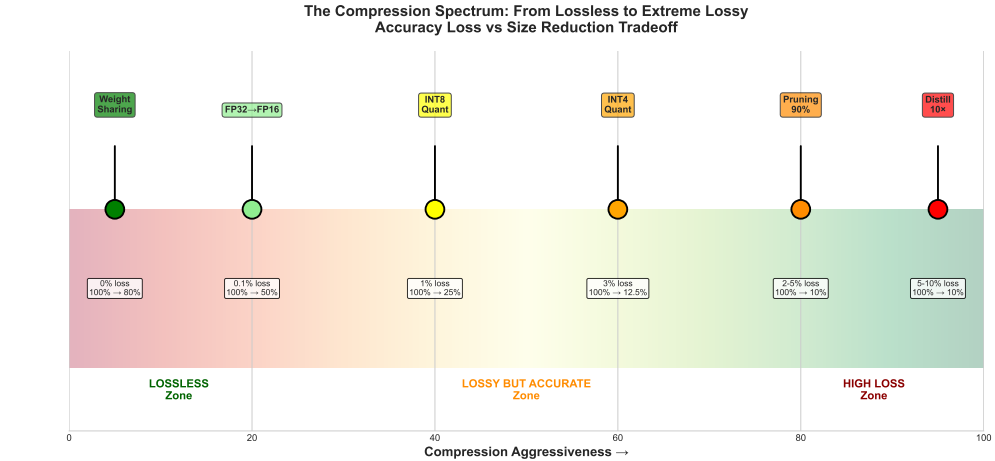
Model Size Evolution:

Model	Params	Size (FP32)
BERT-Base	110M	440MB
BERT-Large	340M	1.4GB
GPT-2	1.5B	6GB
GPT-3	175B	700GB
PaLM	540B	2.1TB

Trend: Models grow 10$\times$ every 2 years
Hardware grows 2$\times$ every 2 years
Gap widens without compression

Mathematical Reality: 175B params $\times$ 4 bytes/param = 700GB minimum memory

Foundation 2: Compression Spectrum (Visual)



The Spectrum:
Lossless (perfect accuracy, small gains) to Lossy (large gains, small accuracy loss)

Foundation 2: Compression Spectrum (Detailed)

Lossless Methods:

1. Weight Sharing

- Technique: Cluster similar weights
- Reduction: 10-20%
- Accuracy: 100% preserved
- Use case: When zero loss required

2. Low-Rank Factorization

- Technique: $W = UV^T$ decomposition
- Reduction: 30-40%
- Accuracy: 99-100%
- Use case: Dense layers

Lossy Methods:

3. Quantization (INT8)

- Technique: FP32 $\rightarrow$ 8-bit integers
- Reduction: 75%
- Accuracy: 95-99%
- Use case: Most deployments

4. Quantization (INT4)

- Technique: FP32 $\rightarrow$ 4-bit integers
- Reduction: 87.5%
- Accuracy: 90-97%
- Use case: Mobile/edge devices

Design Decision: Choose method based on accuracy tolerance and size requirements

Foundation 3: Dep

Foundation 3: Deployment Platforms (Detailed)

Server (80-512GB RAM):

Compression:

- GPT-3: 700GB $\rightarrow$ 350GB (FP16)
- Method: Mixed precision
- Latency: $<$100ms

Edge (4-16GB RAM):

Compression:

- GPT-3: 700GB $\rightarrow$ 87GB (INT4)
- Method: Quantization
- Latency: $<$500ms

Mobile (2-4GB RAM):

Compression:

- LLaMA-7B: 28GB $\rightarrow$ 3.5GB
- Method: 4-bit + pruning
- Battery critical

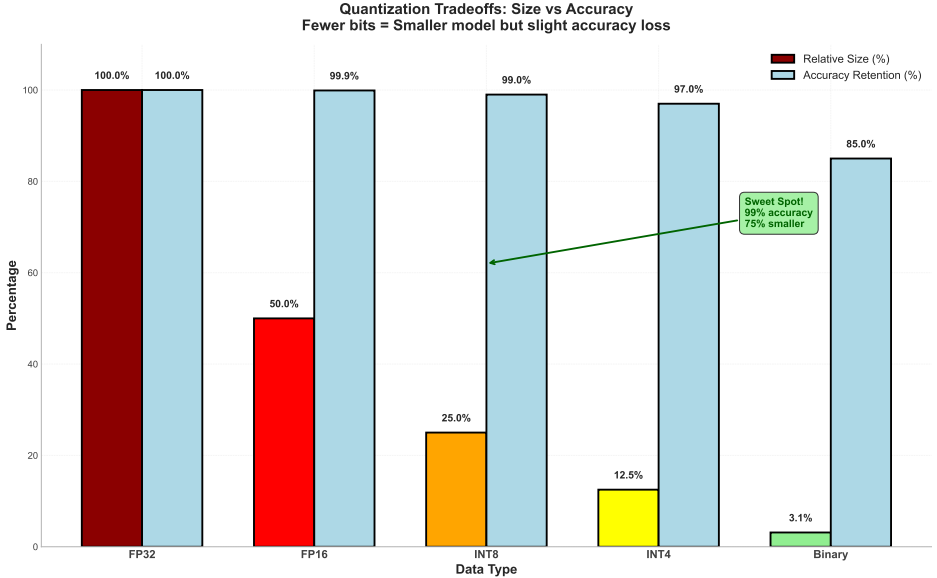
Microcontroller (256KB-2MB):

Compression:

- BERT: 440MB $\rightarrow$ 2MB
- Method: Prune + distill + INT8
- 200$\times$ reduction

Deployment Reality: Platform constraints drive compression method selection

Method 1: Quantization (Visual)



Method 1: Quantization (Detailed Mathematics)

Quantization Formula:

Forward (FP32 → INT8):

$$q = \text{round}\left(\frac{x - x_{\min}}{s}\right)$$

where $s = \frac{x_{\max} - x_{\min}}{255}$ (scale)

Inverse (INT8 → FP32):

$$\hat{x} = q \times s + x_{\min}$$

Numerical Example:

- Weight: $x = 0.374$ (FP32)
- Range: $[-1.0, 1.0]$
- Scale: $s = 2.0/255 = 0.00784$
- Zero-point: 127
- Quantized: $q = 175$ (INT8)
- Recovered: $\hat{x} = 0.376$

Precision Comparison:

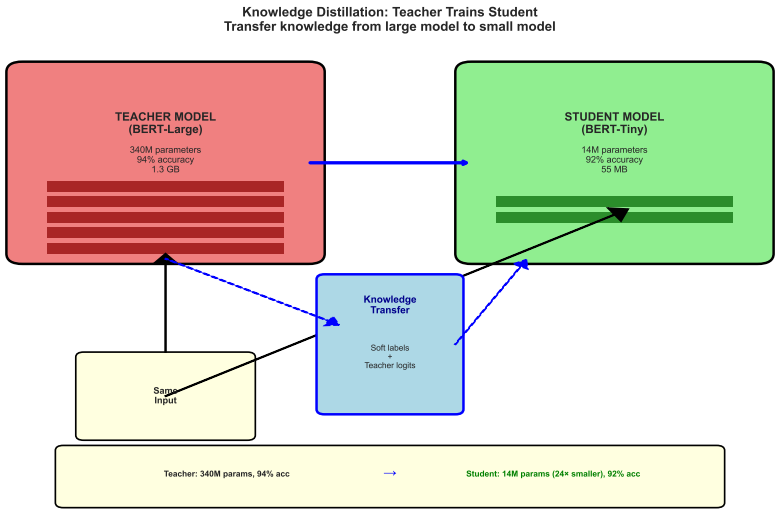
Type	Bits	Range	Precision
FP32	32	$\pm 3.4 \times 10^{38}$	7 digits
FP16	16	$\pm 6.5 \times 10^{4}$	3 digits
INT8	8	-128 to 127	256 values
INT4	4	-8 to 7	16 values

Real Results:

- BERT-Base FP32: 440MB, 89.5%
- BERT-Base INT8: 110MB, 89.1%
- BERT-Base INT4: 55MB, 87.8%

When to Use:
Default choice for most deployments
Hardware support widely available

Method 2: Knowledge Distillation (Visual)



Core Idea: Train small student model to mimic large teacher model

Method 2: Knowledge Distillation (Detailed Process)

Distillation Loss:

$$\mathcal{L} = \alpha \mathcal{L}_{\text{hard}} + (1 - \alpha)\mathcal{L}_{\text{soft}}$$

Hard Loss (ground truth):

$$\mathcal{L}_{\text{hard}} = -\sum_i y_i \log p_i^{\text{student}}$$

Soft Loss (teacher knowledge):

$$\mathcal{L}_{\text{soft}} = -\sum_i p_i^{\text{teacher}} \log p_i^{\text{student}}$$

Temperature scaling: $p_i = \frac{\exp(z_i/T)}{\sum_j \exp(z_j/T)}$

Typical Values:

- $\alpha = 0.5$ (equal weighting)
- $T = 3 - 5$ (temperature)

Concrete Example:

Teacher: BERT-Large

- Parameters: 340M
- Size: 1.4GB (FP32)
- Accuracy: 94.0% (GLUE)
- Inference: 120ms

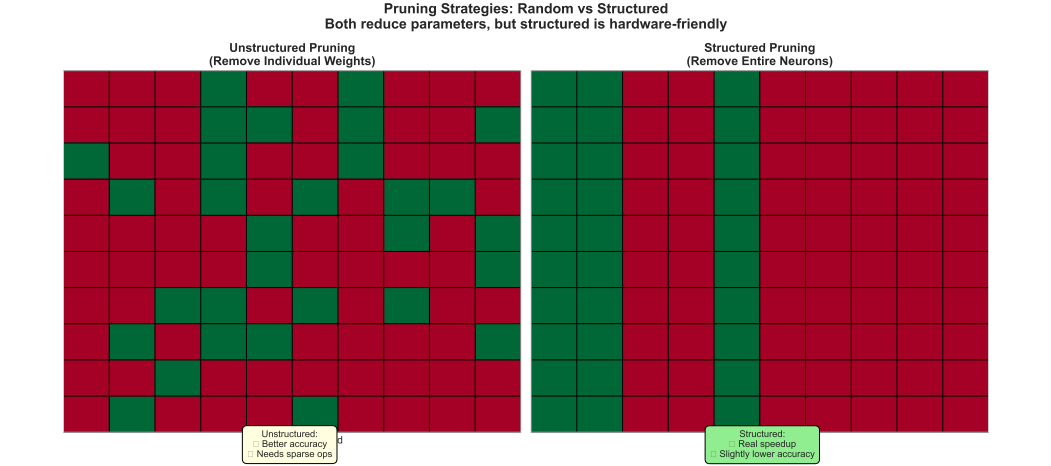
Student: DistilBERT

- Parameters: 66M (5× smaller)
- Size: 260MB (FP32)
- Accuracy: 92.5% (1.5% loss)
- Inference: 60ms (2× faster)

When to Use:
When you need >5× compression
When you can retrain the model
For production deployment at scale

Training Process: Student model trained on teacher's logits (soft targets) + true labels

Method 3: Pruning (Visual)



Core Idea: Remove unimportant weights or neurons from the network

Size Reduction: 90% sparsity = 10× fewer weights

Method 3: Pruning (Detailed Strategies)

Unstructured Pruning:

Algorithm:

1. Train full model
2. Compute weight magnitudes $|w_i|$
3. Remove smallest $p\%$ weights
4. Fine-tune remaining weights

Advantages:

- Highest compression (90-95%)
- Minimal accuracy loss
- Flexible per-layer pruning

Disadvantages:

- Irregular sparsity patterns
- Requires sparse matrix support
- Limited hardware acceleration

Structured Pruning:

Algorithm:

1. Train full model
2. Compute neuron/channel importance
3. Remove entire neurons/channels
4. Fine-tune remaining network

Advantages:

- Direct hardware speedup
- No special sparse libraries
- Smaller actual model size

Disadvantages:

- Lower compression (40-60%)
- More accuracy loss
- Coarser granularity

When to Use:

Method 4: Low-Rank Factorization (Visual)

Matrix Decomposition:

Original weight matrix:

$$W \in \mathbb{R}^{m \times n}$$

Decomposed form:

$$W \approx UV^T$$

where $U \in \mathbb{R}^{m \times r}$, $V \in \mathbb{R}^{n \times r}$, $r \ll \min(m, n)$

Parameter Count:

- Original: $m \times n$
- Factorized: $m \times r + n \times r = r(m + n)$
- Reduction: $\frac{mn}{r(m+n)}$

Numerical Example:

Dense layer: 1024×1024

Original:

- Parameters: $1024^2 = 1,048,576$
- Size: 4MB (FP32)

Factorized ($r = 64$):

- Parameters: $64(1024 + 1024) = 131,072$
- Size: 512KB (FP32)
- Reduction: $8\times$ smaller
- Accuracy loss: $<1\%$

SVD Insight:
Most variance captured by first r singular values
Remaining $(n - r)$ dimensions contribute little

Mathematical Foundation: Singular Value Decomposition (SVD) provides optimal low-rank approximation

Method 4: Low-Rank Factorization (Detailed Analysis)

SVD Algorithm:

Step 1: Compute SVD

$$W = U\Sigma V^T$$

where $\Sigma = \text{diag}(\sigma_1, \ldots, \sigma_n)$ with $\sigma_1 \geq \sigma_2 \geq \ldots$

Step 2: Choose rank r

Energy threshold: $\frac{\sum_{i=1}^{r} \sigma_i^2}{\sum_{i=1}^{n} \sigma_i^2} \geq 0.95$

Step 3: Truncate

$$W_r = U_r \Sigma_r V_r^T$$

where $U_r \in \mathbb{R}^{m \times r}$, $\Sigma_r \in \mathbb{R}^{r \times r}$, $V_r \in \mathbb{R}^{n \times r}$

Step 4: Absorb Σ_r

$$W_r = (U_r \sqrt{\Sigma_r})(\sqrt{\Sigma_r} V_r^T)$$

Real Results:

BERT Embedding Layer:

- Original: $30K \times 768 = 23M$ params
- Factorized ($r = 128$): $128(30K + 768) = 4M$
- Reduction: $5.8\times$ smaller
- Accuracy: 89.5% $\rightarrow$ 89.2%

GPT-2 Attention:

- Original: $768 \times 768 = 590K$ params/layer
- Factorized ($r = 64$): $64 \times 1536 = 98K$
- Reduction: $6\times$ smaller
- Accuracy: Minimal loss (<0.5%)

When to Use:
Dense linear layers (embeddings, attention)
When weight matrix has low intrinsic rank
Combined with quantization for best results

Compression Sweet Spot: $r \approx 10 - 20\%$ **of original dimension balances size and accuracy**

Method 5: Weight Sharing (Visual)

Clustering Approach:

Before: Each weight is unique

- 175B unique floating-point values
- Full precision per weight
- High memory requirement

After: Weights share codebook

- 256 unique cluster centers
- Indices point to codebook
- 2-4 bits per weight (index)

Storage:
Codebook: k values (float)
Indices: n values (2-4 bits)
Total: Much smaller than n floats

K-Means Clustering:

Algorithm:

1. Collect all n weights
2. Run k-means with k clusters
3. Replace each weight with nearest cluster center
4. Store: cluster centers + indices

Numerical Example:

- Weights: $[0.72, 0.69, -0.31, -0.28, ...]$
- Clusters ($k = 4$): $[0.7, -0.3, 0.0, 1.2]$
- Indices: $[0, 0, 1, 1, ...]$ (2 bits each)
- Original: 4 bytes/weight
- Compressed: 0.25 bytes/weight
- Reduction: $16\times$ smaller

Weight Sharing: Lossless-to-lossy spectrum depending on number of clusters

Method 5: Weight Sharing (Detailed Implementation)

Compression Analysis:

Storage Requirements:
Codebook size: k clusters $\times$ 4 bytes
Index size: n weights $\times \lceil \log_2 k \rceil$ bits
Total: $4k + n\lceil \log_2 k \rceil / 8$ bytes

Compression Ratio:

$$\text{Ratio} = \frac{4n}{4k + n\lceil \log_2 k \rceil / 8}$$

Example ($n = 1M$, $k = 256$):

- Original: $1M \times 4 = 4$MB
- Codebook: $256 \times 4 = 1$KB
- Indices: $1M \times 1 = 1$MB (8 bits)
- Total: 1MB + 1KB $\approx$ 1MB
- Ratio: 4$\times$ compression

Accuracy Trade-offs:

Clusters	Compression	Accuracy
$k = 2$	32$\times$	60-70%
$k = 16$	8$\times$	85-90%
$k = 256$	4$\times$	95-99%
$k = 4096$	2.7$\times$	99-100%

Real Results:

- BERT ($k = 256$): 440MB $\rightarrow$ 110MB
- Accuracy: 89.5% $\rightarrow$ 89.3%
- Combined with pruning: 10$\times$ total

When to Use:
When you need lossless compression
Combined with quantization/pruning
For weight-heavy models

Hybrid Approach: Weight sharing + quantization achieves 10-20$\times$ compression

Method 6: Mixed Precision Training (Visual)

Precision Strategy:

FP32 (Master Weights):

- High precision for gradients
- Prevents underflow
- Kept in optimizer state

FP16 (Forward/Backward):

- Fast computation (2×)
- 50% memory reduction
- Hardware acceleration (Tensor Cores)

INT8 (Inference):

- Minimal memory
- 4× faster than FP32
- Quantized after training

Training Loop:

1. **Forward:** FP16 computation
2. **Loss:** FP16 calculation
3. **Loss Scaling:** Multiply by 2^{14}
4. **Backward:** FP16 gradients
5. **Unscale:** Divide by 2^{14}
6. **Update:** FP32 master weights
7. **Copy:** FP32 $\rightarrow$ FP16 for next iteration

Loss Scaling:
Prevents gradient underflow in FP16
Typical scale: 2^{14} to 2^{16}

Mixed Precision: Best of both worlds (FP32 stability + FP16 speed)

Method 6: Mixed Precision Training (Detailed Benefits)

Speed Improvements:

Model	FP32	Mixed
BERT-Base	280 samples/s	560 samples/s
GPT-2	120 samples/s	240 samples/s
ResNet-50	340 images/s	680 images/s

Speedup: Consistent 2× across models

Memory Savings:

Component	FP32	Mixed
Activations	100%	50%
Gradients	100%	50%
Weights	100%	100%
Optimizer	200%	200%
Total	400%	350%

Hardware Support:

NVIDIA Tensor Cores:

- FP16: 125 TFLOPS (V100)
- FP32: 15 TFLOPS (V100)
- Speedup: 8× theoretical
- Real speedup: 2-3× (memory bound)

TPU v4:

- BF16: 275 TFLOPS
- FP32: 68 TFLOPS
- Speedup: 4×

When to Use:
Training large models (GPT-3, BERT)
When you have Tensor Core GPUs
Default for modern PyTorch/TensorFlow

Industry Standard: All large model training uses mixed precision (2020+)

Method 7: Dynamic & Adaptive Computation (Visual)

Early Exit Strategy:

Idea: Not all inputs need full network

Easy examples: Exit after layer 3
Medium examples: Exit after layer 6
Hard examples: Use all 12 layers

Mechanism:

- Add classifier at each layer
- Compute confidence score
- If confidence $>$ threshold, exit
- Otherwise, continue to next layer

Average Speedup:

- Easy: $4\times$ (3 layers vs 12)
- Medium: $2\times$ (6 layers vs 12)
- Hard: $1\times$ (all 12 layers)
- Overall: $2.5\times$ average

Adaptive Attention:

Idea: Not all tokens need full attention

Important tokens: Full attention
Filler words: Sparse attention

Example (12-word sentence):

- "The": 20% attention (2 heads)
- "cat": 100% attention (8 heads)
- "sat": 100% attention (8 heads)
- "on": 20% attention (2 heads)
- "the": 20% attention (2 heads)
- "mat": 100% attention (8 heads)

Computation:

- Full: $12 \times 8 = 96$ head computations
- Adaptive: 48 head computations
- Reduction: 50%

Adaptive Computation: Allocate resources based on input complexity

Method 7: Dynamic & Adaptive Computation (Detailed Results)

Early Exit Networks:

BERT with 3 exits:

- Exit 1 (Layer 4): 35% of samples
- Exit 2 (Layer 8): 45% of samples
- Exit 3 (Layer 12): 20% of samples

Performance:

- Average layers: 6.8 vs 12
- Speedup: 1.76×
- Accuracy: 89.5% → 89.1%
- Loss: 0.4 percentage points

Confidence Threshold:

- High (0.95): Safe, slower (1.3×)
- Medium (0.85): Balanced (1.76×)
- Low (0.75): Risky, faster (2.2×)

Adaptive Attention:

GPT-2 with Adaptive Heads:

- Content words: 8 heads (100%)
- Function words: 2 heads (25%)
- Punctuation: 1 head (12.5%)

Results:

- Computation: 60% of full model
- Speedup: 1.67×
- Perplexity: 18.2 → 18.5
- Quality: Minimal degradation

When to Use:
Production with varied input complexity
When average-case matters more than worst-case
Combined with other compression methods

Research Frontier: Adaptive methods are active area of research (2023-2025)

GPT-3 Deployment Impossibility

The Numbers:

- Parameters: 175 billion
- Precision: FP32 (4 bytes each)
- Total size: $175B \times 4 = 700\text{GB}$
- Typical server RAM: 64-256GB
- Your laptop RAM: 16GB

Impossibility Ratio:
Model size / Laptop RAM $= 44\times$

Even high-end servers struggle (3-11$\times$ over capacity)

Consequences:

Without Compression:

- Must use disk swap
- Inference: 60 seconds per token
- Unusable for production
- Energy: 500W continuous
- Cost: \$10-50 per query

Business Impact:

- Cannot deploy locally
- Must use cloud APIs
- Privacy concerns
- Latency issues
- Ongoing costs

Root Problem: Model capacity requirements exceed deployment hardware by orders of magnitude

Initial Approach: Train Smaller Model

The Naive Solution:

"If GPT-3 is too big, train GPT-2 instead"

GPT-3 175B:

- Size: 700GB (FP32)
- Parameters: 175B
- Accuracy: 92% (few-shot)
- Training: $4.6M

⇓ Reduce size 100×

GPT-2 1.5B:

- Size: 6GB (FP32)
- Parameters: 1.5B
- Accuracy: 67% (few-shot)
- Training: $50K

The Problem:

Accuracy Drop: 25 Percentage Points

Capability Loss:

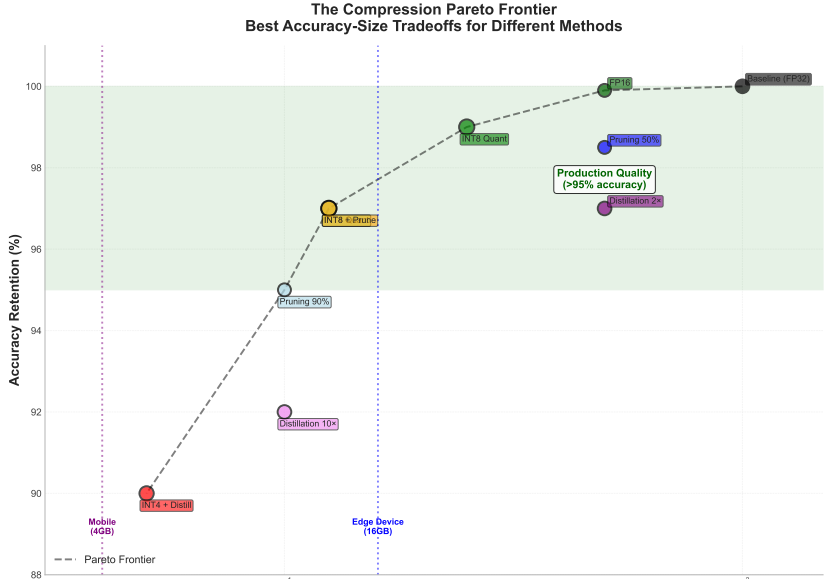
- GPT-3: Complex reasoning, analogies
- GPT-2: Simple pattern matching
- Emergence: Lost at smaller scale

Scaling Laws:
Performance $\propto$ (parameters)$^{0.3}$

To match GPT-3 at 1.5B params:
Need 1000× more data (impossible)

Lesson: Model capacity matters - smaller models cannot simply be trained to match larger ones

Performance Analysis: Why Smaller Models Fail



Root Cause: The Capacity Hypothesis

Theoretical Framework:

Model Capacity:

$$C = f(\text{parameters}, \text{architecture})$$

Knowledge Stored:

$$K \leq C$$

Performance:

$$P \propto K$$

Implications:

- Smaller model ⇒ Less capacity
- Less capacity ⇒ Less knowledge
- Less knowledge ⇒ Worse performance

Numerical Evidence:

Model	Params	Accuracy
BERT-Tiny	14M	78%
BERT-Small	28M	83%
BERT-Medium	66M	86%
BERT-Base	110M	89.5%
BERT-Large	340M	94%

Solution Requirement:
Preserve model capacity (parameters)
Reduce storage/memory footprint
⇒ Compression, not replacement

Diagnosis: Performance tied to parameter count - compression must preserve parameters

Solution Insight: Compress Post-Training

The Breakthrough Idea:

OLD: Train small model (loses knowledge)

⇓

NEW: Train large, then compress

Why This Works:

1. Train full-capacity model
2. Model learns all knowledge
3. Compress learned weights
4. Knowledge preserved (mostly)
5. Fit in deployment memory

Key Observation:
Learned weights have structure
Structure enables compression
Random weights don't compress well

Compression Opportunity:

Trained Weights Properties:

- Clustered values (weight sharing)
- Low effective rank (factorization)
- Many near-zero (pruning)
- Narrow range (quantization)

Concrete Example:

- BERT attention weights
- 95% of weights in [-0.5, 0.5]
- Can use 8 bits instead of 32
- 4× compression with 0.4% loss

Contrast with Random:

- Random weights: Uniform distribution
- No structure to exploit
- Compression hurts accuracy severely

Critical Insight: Trained weights have exploitable structure that random weights lack

Quantization Mechanism: FP32 → INT8

The Math:

Quantization Function:

$$q = \text{round}\left(\frac{x - x_{\min}}{s}\right)$$

where scale $s = \frac{x_{\max} - x_{\min}}{255}$

Dequantization Function:

$$\hat{x} = q \times s + x_{\min}$$

Error:

$$\epsilon = |\hat{x} - x| \leq \frac{s}{2}$$

Key Idea:

- Map range $[x_{\min}, x_{\max}]$ to $[0, 255]$
- Store integer index (1 byte)
- Recover approximate value

Numerical Walkthrough:

Weight Layer Statistics:

- Min: -1.2
- Max: $+0.8$
- Range: 2.0
- Scale: $s = 2.0/255 = 0.00784$

Quantize $x = 0.374$:

1. Shift: $0.374 - (-1.2) = 1.574$
2. Scale: $1.574/0.00784 = 200.76$
3. Round: $q = 201$ (INT8)

Dequantize $q = 201$:

1. Unscale: $201 \times 0.00784 = 1.576$
2. Unshift: $1.576 + (-1.2) = 0.376$
3. Error: $|0.376 - 0.374| = 0.002$

Quantization Error: Bounded by half the quantization step (0.00392 in this example)

Worked Example: BERT Layer Quantization

Layer: BERT Attention Weights

Original (FP32):

- Shape: 768×768
- Weights: 590,592
- Min: -0.487
- Max: $+0.512$
- Mean: 0.003
- Std: 0.124
- Size: $590K \times 4 = 2.36$MB

Quantization Parameters:

- Range: $[-0.487, 0.512]$
- Scale: $(0.512 - (-0.487))/255 = 0.00392$
- Zero-point: 127 (symmetric)

Quantized (INT8):

- Shape: 768×768 (unchanged)
- Values: INT8 in $[0, 255]$
- Size: $590K \times 1 = 590$KB
- Reduction: $4\times$ smaller

Sample Weights:

FP32	INT8	Recovered
0.374	201	0.376
-0.251	67	-0.249
0.089	150	0.090
-0.412	25	-0.413
0.501	255	0.512

Accuracy:

- Original BERT: 89.5%
- Quantized INT8: 89.1%
- Loss: 0.4 percentage points

Real Result: 4× compression with ¡0.5% accuracy loss on BERT-Base

Validation: Real Model Compression Results

BERT-Base Compression:

Method	Size	Accuracy
FP32 Baseline	440MB	89.5%
FP16	220MB	89.5%
INT8	110MB	89.1%
INT4	55MB	87.8%
INT8 + Pruning	22MB	87.5%

Best Trade-off:

- INT8: 4× smaller, 0.4% loss
- Production standard (2024)
- Hardware accelerated

GPT-3 Compression:

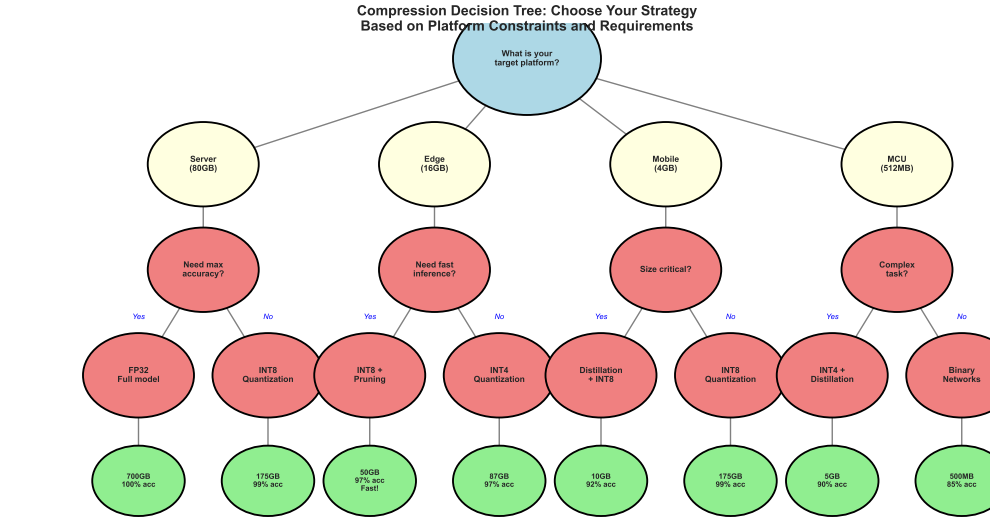
Precision	Size	Quality
FP32	700GB	100%
FP16	350GB	100%
INT8	175GB	98%
INT4	87GB	95%

Deployment Reality:

- OpenAI API: INT8 (likely)
- 4× memory reduction
- 2× throughput increase
- Enables profitable deployment

Industry Adoption: All major LLM APIs use INT8 quantization (2024)

Decision Tree: Choosing Compression Method



Meta-Knowledge: When NOT to Use

Quantization:

Avoid when:

- Model has high dynamic range
- Batch norm layers (unstable)
- Small models ($<$100M params)
- Research/debugging phase

Pruning:

Avoid when:

- No sparse matrix libraries
- Model already small
- All weights are important
- Cannot fine-tune after pruning

Distillation:

Avoid when:

- No budget to retrain
- Teacher model unavailable
- Task requires all model capacity
- Target is $<5\times$ compression

Low-Rank:

Avoid when:

- Weights are full-rank
- Convolutional layers (better methods)
- Recurrent connections
- Model has few dense layers

Anti-Patterns: Know when NOT to use each method to avoid wasted effort

Common Pitfalls and Solutions

Pitfall 1: Calibration

Problem:

- Quantize with training data ranges
- Deploy on different distribution
- Activation ranges differ
- Severe accuracy drop

Solution:

- Calibrate on representative data
- 1000+ diverse examples
- Measure activation ranges
- Use percentile (99%) not max

Pitfall 2: INT4 Overflow

Problem:

- INT4 range: $[-8, 7]$
- Outlier weights cause clipping
- [illegible]

Pitfall 3: Distillation Failure

Problem:

- Student too small ($>20\times$ smaller)
- Cannot learn teacher's knowledge
- Converges to random baseline

Solution:

- Limit compression to 5-10$\times$
- Use intermediate layers
- Progressive distillation

Pitfall 4: Compound Methods

Problem:

- Prune + quantize + distill = fail
- Errors compound
- $10\% + 5\% + 3\% \neq 18\%$
- Actual: 25% degradation

Success Metrics: How to Measure

Primary Metrics:

1. Size Reduction

$$R = \frac{\text{Original Size}}{\text{Compressed Size}}$$

Target: 4-10× for deployment

2. Accuracy Preservation

$$A = \frac{\text{Compressed Accuracy}}{\text{Original Accuracy}}$$

Target: >95% (absolute <3% loss)

3. Latency Improvement

$$L = \frac{\text{Original Latency}}{\text{Compressed Latency}}$$

Target: 2-4× speedup

4. Energy Efficiency

$$\text{Original Energy}$$

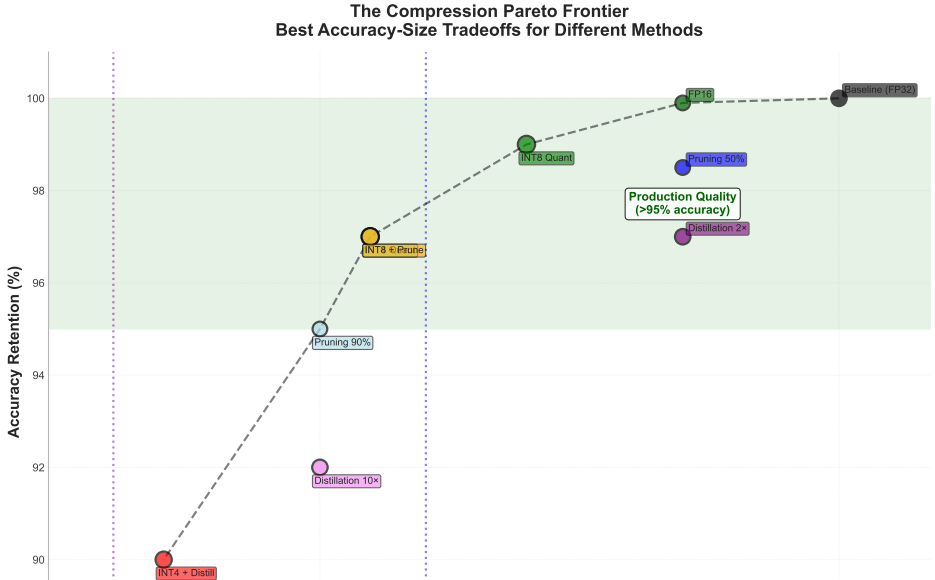
Real Benchmark (BERT):

Method	R	A	L	E
Baseline	1×	100%	1×	1×
FP16	2×	100%	1.5×	1.8×
INT8	4×	99%	2.5×	3.2×
INT4	8×	96%	3.5×	5.1×
Pruned	10×	95%	1.2×	1.5×

Trade-off Analysis:

- INT8: Best balance (4×, 99%, 2.5×)
- INT4: Maximum compression
- Pruning: Size without speedup (need sparse support)

Unified Framework: Method Comparison



Modern Applications: On-Device AI Revolution

Smartphone LLMs (2024):

Apple Intelligence (iPhone 15):

- Model: 3B parameter LLM
- Original: 12GB (FP32)
- Compressed: 1.5GB (4-bit + pruning)
- Methods: INT4 + 50% pruning
- Performance: 30 tokens/sec
- Privacy: 100% on-device

Google Gemini Nano:

- Model: 1.8B parameters
- Size: 900MB (INT8)
- Latency: 40 tokens/sec
- Battery: 1% per 1000 tokens

Edge Computing:

Raspberry Pi 4 (8GB):

- LLaMA-2 7B quantized (INT4)
- Size: 3.5GB
- Speed: 2 tokens/sec
- Use case: Local assistant

NVIDIA Jetson (16GB):

- GPT-J 6B (INT8)
- Size: 6GB
- Speed: 15 tokens/sec
- Use case: Robotics, drones

Impact:
Compression enables privacy-preserving AI
Zero cloud dependency
Millisecond latency

2024 Reality: Compression makes AI ubiquitous (phones, cars, appliances)

Implementation: PyTorch Quantization in 15 Lines

Dynamic Quantization:

```
import torch

# Load pre-trained model
model = BertForSequenceClassification
   .from_pretrained('bert-base')

# Quantize to INT8
quantized_model = torch.quantization
   .quantize_dynamic(
      model,
      {torch.nn.Linear},
      dtype=torch.qint8
   )

# Save compressed model
torch.save(quantized_model,
   'bert_int8.pt')
```

Result: 440MB $\rightarrow$ 110MB (4$\times$)

Static Quantization:

```
# Prepare model
model.qconfig = torch.quantization
   .get_default_qconfig('fbgemm')
torch.quantization.prepare(model)

# Calibrate with data
for batch in calibration_data:
   model(batch)

# Convert to INT8
quantized_model = torch.quantization
   .convert(model)

# Inference
with torch.no_grad():
   output = quantized_model(input)
```

Advantage: Better accuracy (calibrated ranges)

Production Code: PyTorch provides built-in quantization (torch.quantization module)

Key Takeaways: Five Principles

Model Efficiency Fundamentals

1. **Compression Preserves Knowledge**
   Train large, compress post-training beats training small
   Example: GPT-3 INT4 (87GB) outperforms GPT-2 (6GB)

2. **Quantization is the Default**
   4× reduction, <1% accuracy loss, hardware accelerated
   Use INT8 unless you have specific constraints

3. **Platform Drives Strategy**
   Server: FP16/INT8 — Edge: INT4 — Mobile: INT4+Pruning — MCU: Distillation+INT8
   Deployment memory determines compression needs

4. **Combine Methods Carefully**
   Quantization + (Pruning OR Distillation) works
   All three together compounds errors

5. **Measure Four Metrics**
   Size reduction, accuracy, latency, energy
   Optimize for the bottleneck

Summary: Compression makes modern AI deployable everywhere

Next Steps: Lab & Week 12

Week 11 Lab:

Hands-On Activities:

1. Quantize BERT (FP32 $\rightarrow$ INT8)
2. Measure size, accuracy, latency
3. Distill GPT-2 (1.5B $\rightarrow$ 300M)
4. Prune ResNet (90% sparsity)
5. Deploy quantized model

Tools:

- PyTorch quantization API
- Hugging Face transformers
- ONNX Runtime

Deliverable:
Compress a model 10$\times$ with $<$3% accuracy loss

Week 12: Ethics & Fairness

Efficiency $\rightarrow$ Ethics Link:

Sustainability:

- GPT-3 training: 1287 MWh
- Carbon: 550 tons CO_2
- Compression reduces deployment energy 5$\times$

Accessibility:

- On-device AI: No cloud required
- Privacy-preserving inference
- Works in low-connectivity regions

Democratization:

- Run LLMs on $200 hardware
- No API costs
- Open access to AI

Bridge to Ethics: Efficiency enables sustainable, accessible, democratized AI

Week 12: AI Ethics & Fairness

From Bias Detection to Responsible AI

BSc Natural Language Processing

Discovery-Based Learning Approach

2025

The AI That Rejected All Women

Amazon's Hiring AI (2014-2018):

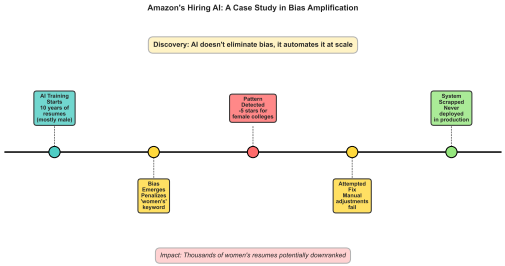
Training Data:

- 10 years of resumes
- Mostly male engineers (historical)
- Used to train ML ranking model

The Discovery:

- Resume mentions "women's chess club" → -5 stars
- Attended women's college → Downranked
- Any "women's" keyword → Penalty

Impact:
Thousands of women's resumes potentially rejected
System never deployed (discovered during testing)



The Insight:
"AI doesn't eliminate bias,
it automates it at scale"

Why It Happened:

- Model learned from biased history
- Optimized to match past hires
- Past hires were mostly men
- Model: "good candidates look like men"

Paradigm Shift: From "Objective Algorithms" to "Bias Amplifiers"

OLD Belief (2010):

"Algorithms are objective and fair"

Reasoning:

- Math has no prejudice
- Computers treat everyone equally
- Data-driven decisions are neutral
- Removes human bias from process

Example Claim:

- ML hiring: No gender/race considered
- Should be fairer than humans
- "Let the data speak"

Reality:
This assumption was wrong

NEW Understanding (2024):

"Algorithms amplify training data bias"

Reality:

- Models learn historical patterns
- Historical data reflects discrimination
- Optimization amplifies patterns
- Scale multiplies impact

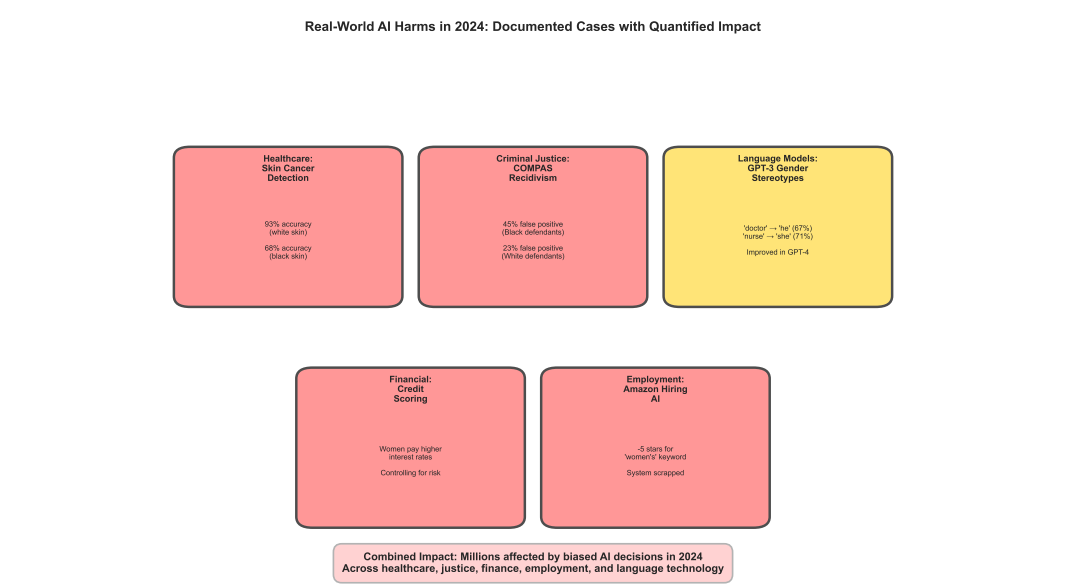
Concrete Examples:

- Resume screening: Women downranked
- Loan approval: Racial disparities
- Medical diagnosis: Worse for minorities
- Facial recognition: Lower accuracy for Black faces

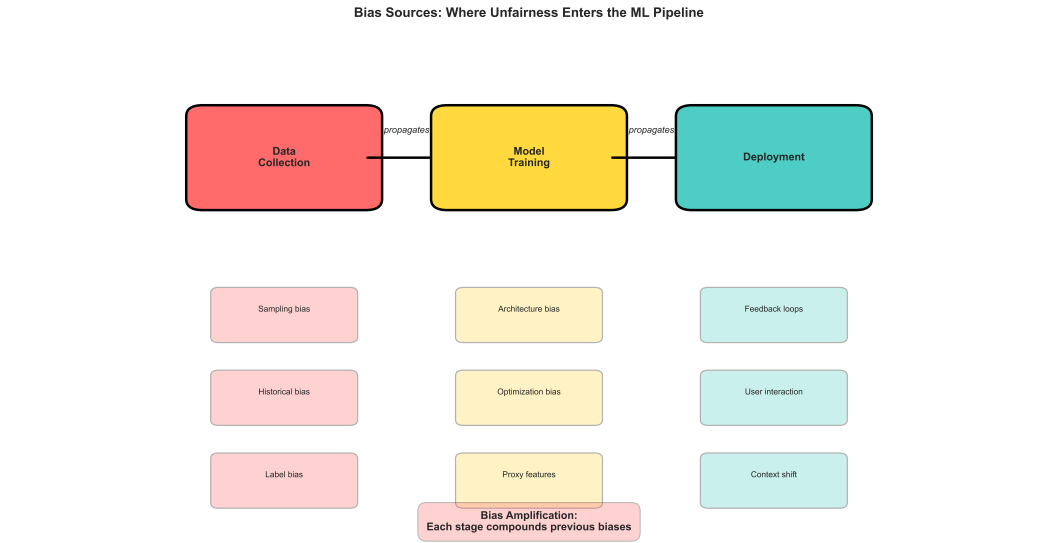
Solution:
Proactive bias detection & mitigation

Key Insight: Bias is not a bug to fix, it's a fundamental challenge requiring ongoing vigilance

Real-World Harms: Quantified Impact in 2024



Foundation 1: Bias Sources (Visual)



Foundation 1: Bias Sources (Detailed)

1. Data Bias:

Sampling Bias

- Training data not representative
- Example: Medical AI trained on 80% white patients
- Impact: Lower accuracy for minorities

Historical Bias

- Data reflects past discrimination
- Example: Hiring data (mostly male engineers)
- Impact: Model learns to prefer men

Label Bias

- Human labelers have biases
- Example: Toxicity labels vary by annotator demographics
- Impact: Model inherits annotator biases

2. Model Bias:

Architecture Bias

- Model design favors certain patterns
- Example: CNNs for faces (tested on white faces)
- Impact: Worse for underrepresented groups

Optimization Bias

- Loss function optimized for majority
- Example: Accuracy maximized on dominant class
- Impact: Minority performance sacrificed

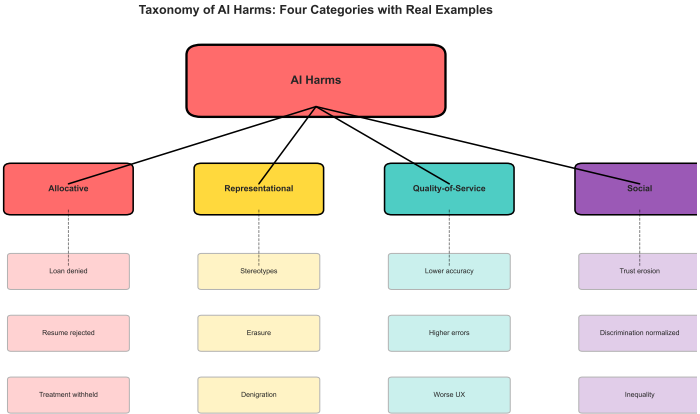
3. Deployment Bias:

Feedback Loops

- Model predictions influence future data
- Example: Biased recommendations $\rightarrow$ biased clicks $\rightarrow$ more bias
- Impact: Self-reinforcing discrimination

Comprehensive View: Bias is not one problem, it's a systemic challenge across the ML pipeline

Foundation 2: Harm Taxonomy (Visual)



Key Insight: Same AI system can cause multiple harm types simultaneously

Foundation 2: Harm Taxonomy (Detailed)

1. Allocative Harm:

Resources withheld or unfairly distributed

Examples:

- Loan denied due to biased credit score
- Resume rejected by biased hiring AI
- Medical treatment withheld (risk score)
- Insurance premium higher (demographic)

Impact: Direct material loss (money, opportunity)

2. Representational Harm:

Stereotypes reinforced or groups erased

Examples:

- Image search: "CEO" shows only men
- Translation: "The doctor" $\rightarrow$ "he"
- Face recognition: Fails on minorities
- Voice assistants: Only understand native speakers

Impact: Dignity, identity, social standing

3. Quality-of-Service Harm:

Unequal performance across demographics

Examples:

- Skin cancer detection: 93% (white), 68% (Black)
- Speech recognition: Higher error for accents
- Face unlock: Fails more for women, minorities
- Medical AI: Trained on majority population

Impact: Frustration, exclusion, worse outcomes

4. Social Harm:

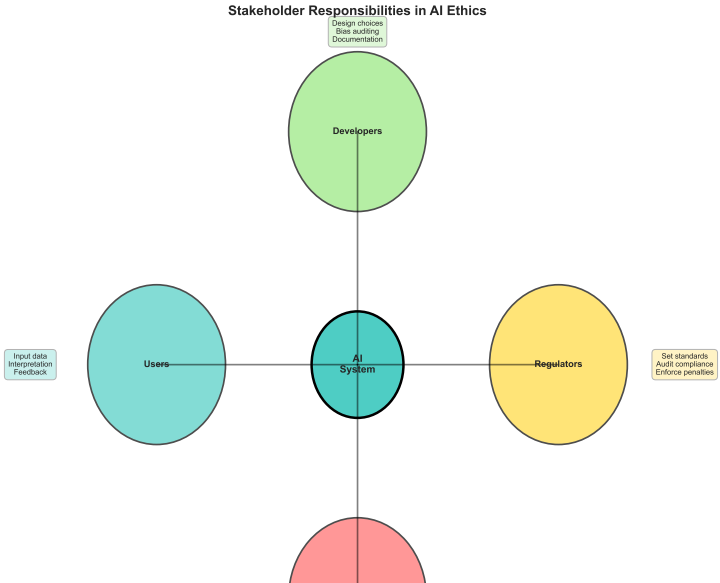
Erosion of trust and normalization of discrimination

Examples:

- COMPAS: Discrimination in sentencing
- People avoid AI systems (distrust)
- "If AI says it, it must be true" (authority)
- Inequality becomes automated, invisible

Impact: Societal trust, democratic participation

Foundation 3: Stakeholders (Visual)



Foundation 3: Stakeholders (Detailed Responsibilities)

Developers:

Responsibilities:

- Design with fairness in mind
- Audit for bias pre-deployment
- Document limitations transparently
- Provide recourse mechanisms

Tools:

- Fairness metrics (AIF360, Fairlearn)
- Bias detection tools
- Model cards (documentation)

Users:

Responsibilities:

- Understand system limitations
- Interpret outputs critically
- Report observed bias
- Participate in feedback

Affected Communities:

Responsibilities:

- Share lived experiences of harm
- Provide context developers miss
- Demand accountability
- Advocate for rights

Reality:

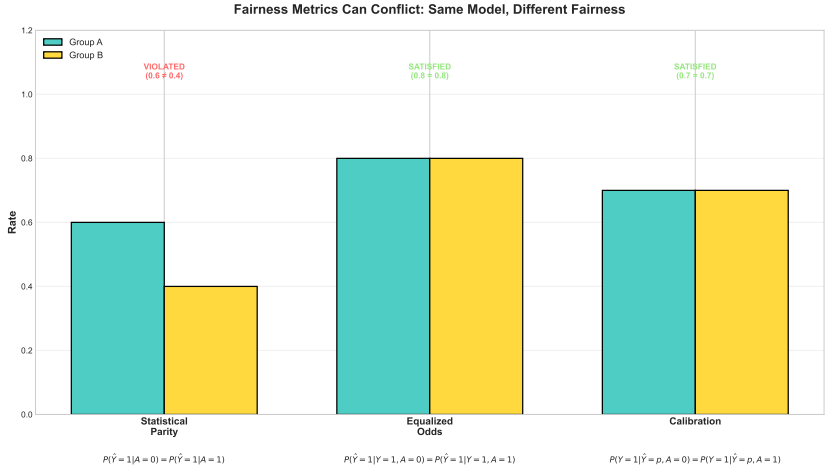
- Often excluded from design
- Harm discovered after deployment
- Limited recourse when harmed

Regulators:

Responsibilities:

- Set fairness standards
- Audit compliance
- Enforce penalties for violations
- Update laws as tech evolves

Method 1: Statistical Parity (Visual)



Core Idea: Equal positive prediction rates across groups

Formula: $P(\hat{Y}=1|A=0)=P(\hat{Y}=1|A=1)$

Statistical Parity: Demographic parity - same proportion of each group receives positive outcome

Method 1: Statistical Parity (Detailed)

Definition:

$$P(\hat{Y} = 1|A = 0) = P(\hat{Y} = 1|A = 1)$$

where:

- $\hat{Y}$: Model prediction
- A: Protected attribute (race, gender, etc.)
- $A = 0$: Majority group
- $A = 1$: Minority group

Interpretation:

- Same approval rate for both groups
- Example: If 40% of men get loans, 40% of women should too
- Independent of actual qualifications

Numerical Example:

- 1000 male applicants, 400 approved (40%)
- 1000 female applicants, 400 approved (40%)

When to Use:

- Hiring (equal opportunity)
- College admissions
- Loan approvals
- When group parity is legal requirement

Advantages:

- Easy to understand
- Easy to measure
- Legal precedent in some domains
- Prevents overt discrimination

Disadvantages:

- Ignores base rates (true qualifications)
- May require different thresholds per group
- Can conflict with accuracy
- Not always legally defensible

Limitation:

Method 2: Equalized Odds (Visual)

The Idea:

Equal accuracy across groups

Two Conditions:

1. Equal True Positive Rate:

$$P(\hat{Y} = 1|Y = 1, A = 0) =$$

$$P(\hat{Y} = 1|Y = 1, A = 1)$$

2. Equal False Positive Rate:

$$P(\hat{Y} = 1|Y = 0, A = 0) =$$

$$P(\hat{Y} = 1|Y = 0, A = 1)$$

Concrete Example:

COMPAS Recidivism (Actual):

- TPR (Black): 60%
- TPR (White): 60%
- FPR (Black): 45%
- FPR (White): 23%

Violation: FPR differs ($45\% \neq 23\%$)

Impact: Black defendants mislabeled as high-risk at $2\times$ rate

Equalized Odds: Ensures model is equally accurate for both qualified and unqualified in each group

Method 2: Equalized Odds (Detailed Analysis)

Why It Matters:

True Positive Rate (TPR):

- Among qualified, fraction correctly identified
- Example: Among people who won't reoffend, how many correctly labeled low-risk?
- Equal TPR: Both groups benefit equally

False Positive Rate (FPR):

- Among unqualified, fraction incorrectly identified
- Example: Among people who will reoffend, how many mislabeled low-risk?
- Equal FPR: Both groups harmed equally by errors

COMPAS Example (2016):

- 10,000 defendants analyzed
- FPR Black: 45% (450 false positives)
- FPR White: 23% (230 false positives)
- Result: Black defendants falsely labeled high-risk

When to Use:

- Criminal justice (recidivism, bail)
- Medical diagnosis
- Credit scoring
- Any high-stakes decision

Advantages:

- Respects merit (true qualifications)
- Ensures equal error rates
- Legally defensible (equal treatment)
- Widely accepted fairness criterion

Disadvantages:

- Conflicts with statistical parity
- May not satisfy calibration
- Requires ground truth labels
- Can be difficult to achieve

Best Practice:

Method 3: Counterfactual Fairness (Visual)

The Question:

"Would the prediction change if we changed only the protected attribute?"

Example:

Resume:

- Name: James Smith
- Education: MIT Computer Science
- Experience: 5 years at Google
- **Prediction: 0.85 (hire)**

Counterfactual Resume:

- Name: Jennifer Smith (ONLY change)
- Education: MIT Computer Science (same)
- Experience: 5 years at Google (same)
- **Prediction: 0.80 (hire)**

Evaluation:

If predictions differ ($0.85 \neq 0.80$):

- Model is using gender
- Counterfactual fairness VIOLATED
- Direct discrimination

If predictions same ($0.85 = 0.85$):

- Gender does not affect score
- Counterfactual fairness SATISFIED
- No direct discrimination

Causal Fairness:
Only causally relevant factors should affect predictions

Protected attributes should have ZERO causal effect

Counterfactual Fairness: Causal framework - protected attribute should not cause prediction to change

Method 3: Counterfactual Fairness (Detailed Implementation)

Formal Definition:

$$P(\hat{Y}_{A \leftarrow a} | X = x, A = a) =$$
$$P(\hat{Y}_{A \leftarrow a'} | X = x, A = a)$$

Translation: Prediction for individual with protected attribute a equals prediction if they had attribute a', holding all else constant

Implementation:

Step 1: Build causal graph

- Identify causal relationships
- Separate legitimate vs illegitimate paths

Step 2: Block illegitimate paths

- Remove direct effect of A on $\hat{Y}$
- Remove indirect effect through mediators

Step 3: Test counterfactuals

- Generate counterfactual examples

Challenges:

1. **Proxy Features:**
   - ZIP code correlated with race
   - First name correlated with gender
   - Must remove ALL correlated features
   - May lose predictive power
2. **Causal Graph:**
   - Requires domain knowledge
   - Hard to validate
   - May be controversial
3. **Legitimate Pathways:**
   - Some gender effects may be legitimate
   - Example: Women's health outcomes
   - Need to distinguish discrimination from valid correlation

When to Use:
When you can specify causal relationships

Mitigation 1: Data Augmentation (Visual)

The Problem:

Imbalanced training data

Example:

- 10,000 male resumes (hired)
- 1,000 female resumes (hired)
- Ratio: 10:1

Consequence:

- Model learns "male = good candidate"
- Under-represents female patterns
- Worse performance on women

The Solution:

Augment minority class

Methods:

- Oversample minority: Duplicate female resumes
- Undersample majority: Remove male resumes
- SMOTE: Generate synthetic female resumes

Result:

- 5,000 male resumes
- 5,000 female resumes
- Ratio: 1:1
- Model sees balanced examples

Data Augmentation: Fix the data, fix the bias - balance training distribution

Mitigation 1: Data Augmentation (Detailed Techniques)

1. Oversampling:

Method: Duplicate minority samples

Advantages:

- Simple to implement
- No data loss
- Balances classes

Disadvantages:

- Exact duplicates (overfitting)
- Larger dataset (slower training)

2. Undersampling:

Method: Remove majority samples

Advantages:

- Smaller dataset (faster)
- Balances classes

Disadvantages:

- Loses information

3. SMOTE (Synthetic):

Method: Generate synthetic minority samples

Algorithm:

1. Find k nearest neighbors (minority)
2. Interpolate between neighbors
3. Create new synthetic sample
4. Repeat until balanced

Example:

- Resume A: (skills=[Python, Java], exp=5)
- Resume B: (skills=[Python, C++], exp=7)
- Synthetic: (skills=[Python, Java, C++], exp=6)

Advantages:

- No exact duplicates
- Expands decision boundary
- Better generalization

Disadvantages:

Mitigation 2: Adversarial Debiasing (Visual)

The Setup:

Two Models:

1. Classifier (C):

- Task: Predict hired/not hired
- Input: Resume features
- Goal: Maximize accuracy

2. Adversary (A):

- Task: Predict gender from C's hidden layer
- Input: C's internal representation
- Goal: Maximize gender prediction

Training:

- C tries to fool A (remove gender signal)
- A tries to detect gender (maximize accuracy)
- Minimax game: C vs A

The Outcome:

If A succeeds (predicts gender well):

- C's representation contains gender info
- C is biased
- Update C to remove gender signal

If A fails (random guessing):

- C's representation is gender-neutral
- C cannot be biased (no gender info)
- Training complete

Key Idea:
If adversary cannot predict protected attribute from internal representation,
model cannot use it for predictions

Adversarial Debiasing: Game-theoretic approach - remove bias signal from learned representations

Mitigation 2: Adversarial Debiasing (Detailed Mathematics)

Objective Function:

$$\min_{\theta_C} \max_{\theta_A} \mathcal{L}_C - \lambda \mathcal{L}_A$$

where:

- $\mathcal{L}_C$: Classifier loss (accuracy)
- $\mathcal{L}_A$: Adversary loss (gender prediction)
- λ: Trade-off parameter

Training Algorithm:

1. **Step 1:** Update adversary
   - Fix classifier weights
   - Train adversary to predict gender
   - Maximize $\mathcal{L}_A$
2. **Step 2:** Update classifier
   - Fix adversary weights
   - Train classifier to fool adversary
   - Minimize $\mathcal{L}_C - \lambda \mathcal{L}_A$
3. **Step 3:** Repeat until convergence

Trade-off Parameter λ:

- $\lambda = 0$: No debiasing (ignore adversary)
- $\lambda =$ small: Weak debiasing
- $\lambda =$ large: Strong debiasing (may hurt accuracy)

Typical Results:

- $\lambda = 0.0$: Accuracy 85%, Gender pred 95% (biased)
- $\lambda = 1.0$: Accuracy 83%, Gender pred 55% (fair)
- $\lambda = 10$: Accuracy 78%, Gender pred 51% (random)

When to Use:

- Deep learning models
- When you can't remove protected attribute from data
- When you want representation-level fairness

Best Practice:
Tune λ with validation set

Mitigation 3: Calibration & Post-processing (Visual)

The Problem:

Model outputs not calibrated across groups

Example:

- Model says "70% chance hired"
- For men: 70% actually hired (calibrated)
- For women: 50% actually hired (not calibrated)

Impact:

- Scores mean different things per group
- Misleading confidence estimates
- Unfair decision thresholds

The Solution:

Post-process outputs to equalize calibration

Method:

1. Train model (biased outputs)
2. Compute calibration per group
3. Adjust thresholds to equalize
4. Apply different threshold per group

Result:

- Men: Threshold = 0.5 for hiring
- Women: Threshold = 0.4 for hiring (compensate)
- Same true positive rate for both

Calibration: Post-processing approach - adjust outputs after training to ensure fairness

Mitigation 3: Calibration & Post-processing (Detailed Techniques)

Calibration Curve:

$$P(Y = 1|\hat{Y} = p, A = a) = p$$

Meaning:

- If model says p=0.7, 70% should be positive
- Must hold for EACH group separately
- Calibration $\neq$ accuracy

Platt Scaling:

Method: Learn per-group transformation

$$\hat{p}_{\text{calibrated}} = \sigma(w \cdot \hat{p} + b)$$

where σ is sigmoid, w and b learned per group

Algorithm:

1. Split validation data by group
2. Fit logistic regression per group

Threshold Optimization:

Method: Find different thresholds per group

Algorithm:

1. Set fairness constraint (e.g., equal TPR)
2. Search for thresholds that satisfy constraint
3. Apply group-specific thresholds

Example:

- Group A: Threshold 0.5 $\rightarrow$ TPR 80%
- Group B: Threshold 0.4 $\rightarrow$ TPR 80%
- Result: Equal TPR achieved

Advantages:

- Model-agnostic (works with any classifier)
- No retraining needed
- Mathematically guarantees fairness metric

Disadvantages:

- Requires labeled validation data

Safety: Red Teaming & Constitutional AI (Visual)

Red Teaming:

Adversarial testing for harmful outputs

Process:

1. Hire diverse red team
2. Attempt to elicit harmful outputs
3. Document failure modes
4. Fix vulnerabilities
5. Repeat

Example Attacks:

- Jailbreak prompts: "Ignore previous instructions"
- Indirect requests: "Write a story about..."
- Multi-turn manipulation
- Role-playing scenarios

Coverage:

- Toxicity, bias, misinformation
- Privacy leaks
- Instruction following failures

Constitutional AI:

AI trained to follow ethical principles

The Constitution:

1. Be helpful, harmless, honest
2. Refuse harmful requests
3. Explain refusals politely
4. No discrimination
5. Respect privacy
6. Cite sources

Training:

- Generate responses
- Critique against principles
- Revise to satisfy principles
- RLHF with constitutional feedback

Result:

- GPT-4: Refuses harmful requests
- Explains reasoning transparently

Challenge: Gender Bias in Word Embeddings

The Discovery (2016):

Word2Vec embeddings contain gender stereotypes

Evidence:

Word analogy task:

- man : computer programmer :: woman : **homemaker**
- man : doctor :: woman : **nurse**
- man : brilliant :: woman : **lovely**

Quantification:

- "doctor" - "man" + "woman" $\approx$ "nurse"
- Cosine similarity: 0.72
- Should be: "doctor" (gender-neutral)

Root Cause:

Training Data:

- Google News corpus (3B words)
- Reflects historical gender roles
- "doctor" appears more with "he"
- "nurse" appears more with "she"

Consequence:

- Embeddings used in downstream tasks
- Resume ranking, translation, search
- Bias amplified in applications
- Millions affected

Scale:
Every model using Word2Vec inherits this bias
(billions of predictions affected)

Word Embedding Bias: Foundational bias - affects all models built on these embeddings

Key Takeaways: Five Principles of Responsible AI

Responsible AI Fundamentals

1. **Bias is Systemic, Not a Bug**
   Enters at data, model, and deployment stages - requires ongoing vigilance
   Example: Amazon AI rejected women despite no gender feature

2. **Fairness Metrics Conflict**
   Statistical parity $\neq$ equalized odds $\neq$ calibration
   Choose metric based on application context and legal requirements

3. **Detection Before Mitigation**
   Measure bias first, then apply targeted intervention
   Use appropriate metrics: WEAT for embeddings, demographic parity for outcomes

4. **Stakeholder Participation**
   Include affected communities in design and evaluation
   Developers alone cannot anticipate all harms

5. **Transparency and Accountability**
   Document limitations, provide recourse, enable auditing
   Model cards, datasheets, fairness reports

Summary: Responsible AI requires technical rigor, ethical awareness, and stakeholder engagement

Course Conclusion: From Theory to Practice

12-Week Journey:

Weeks 1-3: Foundations

- N-grams → Neural LMs → RNNs
- Word embeddings, language modeling

Weeks 4-5: Architectures

- Seq2seq, attention, transformers
- The attention revolution

Weeks 6-8: Modern NLP

- BERT, GPT, pre-training
- Tokenization, scaling

Weeks 9-11: Deployment

- Decoding, fine-tuning, compression
- Making AI practical

Week 12: Ethics

- Bias, fairness, responsibility
- Making AI safe

Real-World Impact:

You now understand:

- How language models work (theory)
- How to build them (practice)
- How to deploy them (engineering)
- How to do so responsibly (ethics)

Next Steps:

- Build your own models
- Contribute to open source
- Research novel architectures
- Advocate for responsible AI

The Future:
AI will transform society
You have the knowledge to ensure
that transformation is beneficial

Thank you!

Course Complete: 12 weeks from N-grams to responsible deployment - you are now equipped to build, deploy, and ensure fairness in NLP systems

Teaching Machines to See Patterns

A Neural Networks Primer: Why We Needed Each Piece of the Puzzle

NLP Course 2025

From the 1950s mail sorting crisis to ChatGPT: How humanity taught machines to think

Where We're Going Today

Part 1: The Problem (1943-1969)

- The mail sorting crisis
- First mathematical neurons
- The perceptron revolution
- The XOR catastrophe

Intermission: Understanding the Basics

- How neurons calculate
- Why we need layers
- Following the forward pass

Part 2: The Breakthrough (1980s-1990s)

- Hidden layers save the day
- Backpropagation breakthrough
- Universal approximation proof

Part 3: The Revolution (2000s-Present)

- Deep learning explosion
- Modern architectures
- Real-world impact

Your Turn: Building Networks

- Build your first network
- Next steps

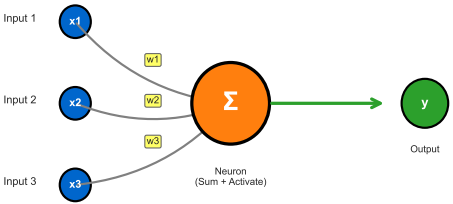
Each part builds on the previous - we'll go step by step!

What is a Neuron? The Building Block

Before we tell the story, let's understand the fundamental building block

Understanding a Single Neuron

Anatomy of a Single Neuron



What Happens Inside a Neuron

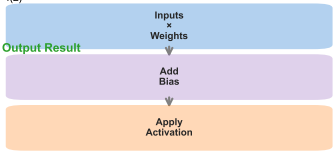
1. Weighted Sum

$z = \Sigma(w_i \times x_i) + bias$

2. Apply Activation

$y = f(z)$

3. Output Result



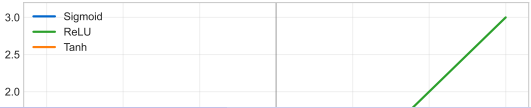
Concrete Example

Example: Is this fruit ripe?

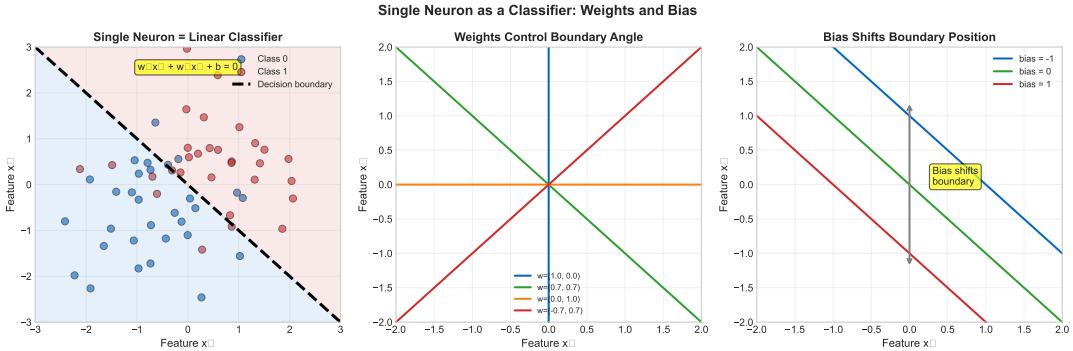
Inputs:

$x_1 = 2$ (color: red=high)
$x_2 = 3$ (softness: high)

Common Activation Functions



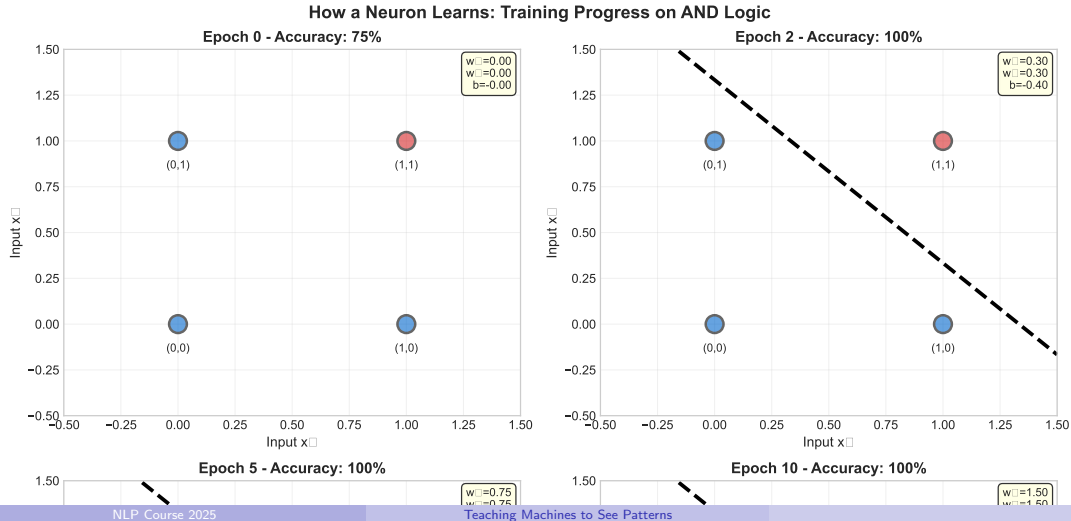
How Weights and Bias Control Decisions



Weights control the angle, bias shifts the position - together they define the decision boundary

Watching a Neuron Learn

Learning = Gradually Adjusting Weights to Fit the Data



The Core Idea: Neural Networks are Function Approximators

The Problem:

- We have inputs (x)
- We want outputs (y)
- But we don't know the formula!
- Examples:
  - Size → Price
  - Image → Label
  - Text → Sentiment

What does this actually mean?

Traditional Approach:

- Guess the formula
- Write explicit rules
- Hope it works
- Problem: Real world is too complex!

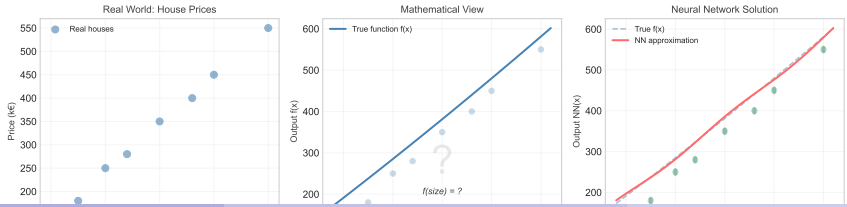
Example:
Price = a × Size + b
(Too simple for real data!)

Neural Network Approach:

- Learn from examples
- Build the formula automatically
- Adjust until it fits
- Works for ANY pattern!

Magic:
NN learns: f(x) ≈ y
No formula needed!

Function Approximation: Learning Patterns from Examples



How NNs Build Complex Functions from Simple Pieces

The LEGO Principle: Combine Simple Parts to Build Anything

The Building Blocks:

1. **Individual Neurons:** Simple decisions
   - "Is input ¿ threshold?"
   - Outputs: on/off (smooth version)
2. **Combine Neurons:** Weight and add
   - Create complex shapes
3. **Stack Layers:** Build hierarchy
   - Each layer adds abstraction

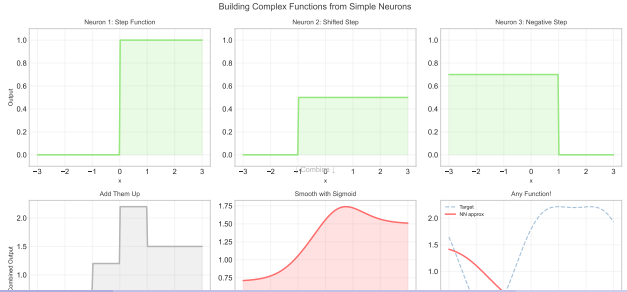
Real-World Analogy:

Making a Cake from Ingredients:

- Flour + Sugar + Eggs
- Mix right amounts
- → Perfect cake!

In Neural Networks:

- Edges + Curves + Colors
- Combine with weights
- → Recognize faces!



The Universal Approximation Theorem: Why This Always Works

The Most Important Theorem in Deep Learning (Cybenko, 1989)

The Theorem (Plain English):

"A neural network with enough neurons can approximate **ANY** *continuous function to* **ANY** *desired accuracy"*

What This Means:

- Universal: Works for any smooth pattern
- Guaranteed: Not hoping, but proving
- Practical: Just add more neurons!

The Catch:

- How many neurons? Could be millions
- How to train? That's the art

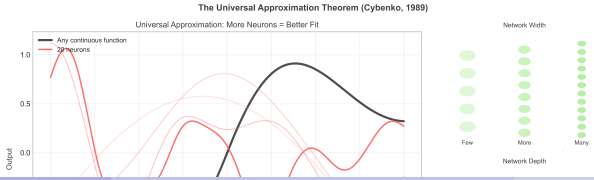
Intuitive Proof:

Pixel art analogy:

- 4 pixels: Blocky
- 100 pixels: Recognizable
- 10,000 pixels: Photo-realistic

Same with neurons:

- Few: Rough
- More: Better
- Many: Nearly perfect



1950s: The Mail Sorting Crisis

The Challenge:

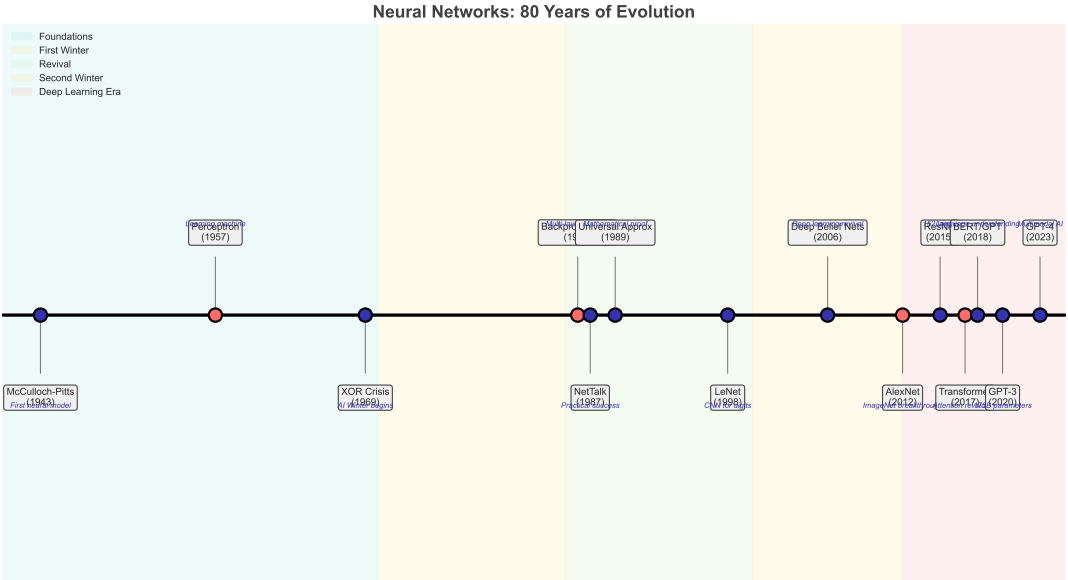
- 150 million letters per day
- Hand-written addresses
- Human sorters: slow, expensive, error-prone
- Traditional programming: useless

Why Traditional Code Failed:

- Can't write rules for every handwriting style
- Too many variations of each letter
- Context matters: "I" vs "l" vs "1"
- This wasn't computation—it was pattern recognition

This problem would take 40 years to solve properly

80 Years of Neural Networks: The Complete Journey



Why Can't We Just Write Rules?

Problem: Recognize the Letter "A"

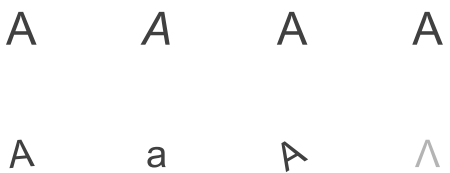
Traditional Approach (Failed):

```
if (has_triangle_top AND
    has_horizontal_bar AND
    two_diagonal_lines) {
  return "A"
}
```

But what about...

- Handwritten A's?
- Different fonts?
- Rotated A's?
- Partial A's?

The Challenge: Infinite Variations of "A"



Just for the letter "A", we'd need thousands of rules!

The breakthrough: What if machines could learn patterns like children do?

1943: The First Spark - McCulloch & Pitts

The Birth of Computational Neuroscience

The Revolutionary Paper:

- "A Logical Calculus of Ideas Immanent in Nervous Activity"
- First mathematical model of neurons
- Proved: Networks can compute ANY logical function
- Inspired von Neumann's computer architecture

Key Insight:

- Neurons = Logic gates
- Brain = Computing machine
- Thinking = Computation

The Model:

- Binary neurons (0 or 1)
- Threshold activation
- Fixed connections
- No learning yet!

Historical Impact:

- Founded field of neural networks
- Influenced cybernetics movement
- Set stage for AI research
- "The brain is a computer" metaphor

14 years later, Rosenblatt would add the missing piece: learning

1957: The First Learning Machine - The Perceptron

Frank Rosenblatt's Radical Idea: Neurons That Learn

Beyond McCulloch-Pitts:

- Adjustable weights (not fixed!)
- Learning from mistakes
- Physical machine built (Mark I)
- Could recognize simple patterns

The Hardware:

- 400 photocells (20×20 "retina")
- 512 motor-driven potentiometers
- Weights adjusted by electric motors
- Took 5 minutes to learn patterns

Mathematical Model:

- Inputs: $x_1, x_2, ..., x_n$
- Weights: $w_1, w_2, ..., w_n$
- Sum: $z = \sum_{i=1}^{n} w_i x_i + b$
- Output: $y = \begin{cases} 1 & \text{if } z > 0 \\ 0 & \text{if } z \leq 0 \end{cases}$

In plain words: Each input gets a vote (weight). We add up all votes plus a bias. If total is positive, output 1; otherwise 0.

Learning Rule: If wrong: $w_i = w_i + \eta \cdot error \cdot x_i$

The New York Times, 1958: "The Navy revealed the embryo of an electronic computer that will be able to walk, talk, see, write, reproduce itself and be conscious of its existence."

Let's Understand How This Actually Works

We've Seen the History...

- McCulloch-Pitts invented the neuron
- Rosenblatt made it learn
- The perceptron was born

Now Let's See the Science:

- How does a neuron calculate?
- What does learning mean?
- Why was XOR so hard?

Next 5 slides: Hands-on calculations and exercises
Get your pencil ready - we're going to work through real examples!

Don't worry - we'll return to the story once you understand the basics

Understanding Check: Can You Answer These?

Let's Make Sure We're Together

Quick Questions:

1. Why couldn't traditional programming solve mail sorting?
2. What does a weight represent in simple terms?
3. Why do we need the bias term?
4. What was revolutionary about Rosenblatt's perceptron?

Think About It:

- A weight is like the importance/trust we give to each input
- Bias shifts our decision threshold
- Learning = adjusting these weights
- The perceptron was the first machine that could learn!

Try It Yourself: Draw a simple perceptron with 2 inputs. Label the weights, bias, and output. What would the weights be to compute AND logic?

If any of these are unclear, revisit the previous slides before continuing

Making It Concrete: Teaching OR Logic

Problem: Learn OR function (output 1 if ANY input is 1)

Training Data:

x_1	x_2	Output
0	0	0
0	1	1
1	0	1
1	1	1

The Perceptron:

$$z = w_1 \cdot x_1 + w_2 \cdot x_2 + b$$

$$\text{output} = \begin{cases} 1 & \text{if } z > 0 \\ 0 & \text{if } z \leq 0 \end{cases}$$

In plain words: Multiply first input by first weight, second input by second weight, add bias, check if positive

Learning Process:

1. Start with random weights
2. For each example:
   - Calculate output
   - If wrong: adjust weights
   - If correct: keep weights
3. Repeat until all correct

Final Solution: $w_1 = 1$, $w_2 = 1$, $b = -0.5$

Success! But this was just the beginning...

Let's Calculate Together: Is This Email Spam?

A Real Perceptron Calculation You Can Follow

The Email:

"FREE money! Click here NOW for amazing offer!!!"

Our Features (Inputs):

- x_1 = Has "FREE"? = 1
- x_2 = Has "money"? = 1
- x_3 = Many "!"? = 1
- x_4 = From friend? = 0

Learned Weights:

- $w_1 = +3$ (FREE is very spammy)
- $w_2 = +2$ (money is suspicious)
- $w_3 = +2$ (!!! is aggressive)
- $w_4 = -5$ (friends are trusted)
- $b = -2$ (threshold)

Let's Calculate:

$$\begin{aligned} z &= w_1 \cdot x_1 + w_2 \cdot x_2 + w_3 \cdot x_3 + w_4 \cdot x_4 + b \\ &= 3 \cdot 1 + 2 \cdot 1 + 2 \cdot 1 + (-5) \cdot 0 + (-2) \\ &= 3 + 2 + 2 + 0 - 2 \\ &= 5 \end{aligned}$$

Decision:

- $z = 5 > 0$
- Output = 1 = SPAM!

Try It Yourself: What if this email WAS from a friend ($x_4 = 1$)? Recalculate! Would it still be spam?

Answer: $z = 5 - 5 = 0$, borderline!

This is exactly how early spam filters worked - and why they failed on clever spam

Breaking Down the Math Symbols

Inputs and Weights:

- x_i = input value (what we see)
- w_i = weight (importance/strength)
- b = bias (threshold adjuster)

The Computation:

$$z = \sum_{i=1}^{n} w_i x_i + b$$

This means:

- Multiply each input by its weight
- Add them all up
- Add the bias

Real Example:

Should I go outside?

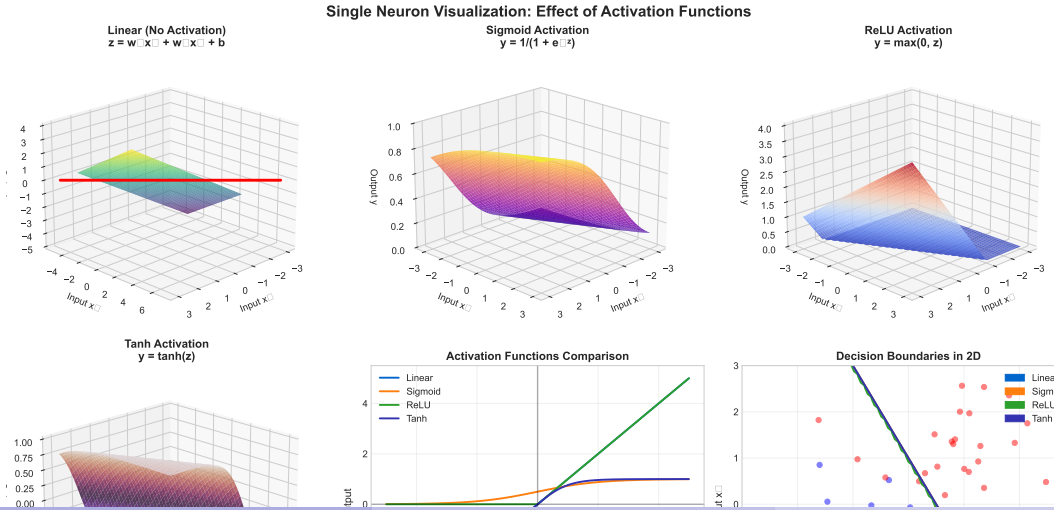
Factor	Value	Weight
Sunny?	1	+2
Raining?	0	-3
Weekend?	1	+1

$$z = (1 \times 2) + (0 \times -3) + (1 \times 1) = 3$$

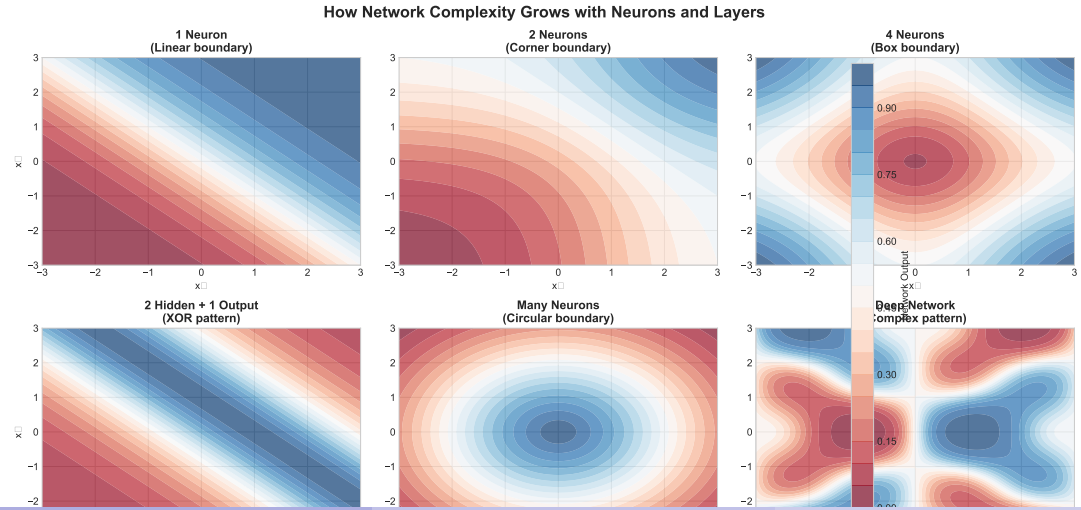
Decision: $z > 0$, so YES!

This simple math would evolve into deep learning

Visualizing How Activation Functions Transform the Output Space



How More Neurons Enable More Complex Decision Boundaries



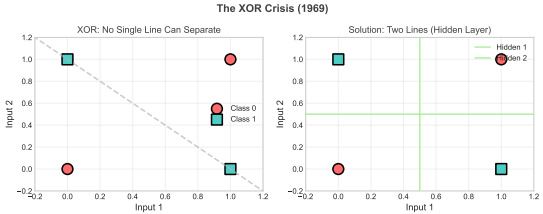
1969: The Crisis - XOR Problem

XOR (Exclusive OR):

x_1	x_2	Output
0	0	0
0	1	1
1	0	1
1	1	0

The Problem:

- Can't draw a single line to separate
- Perceptron only learns linear boundaries
- Real-world problems are non-linear!

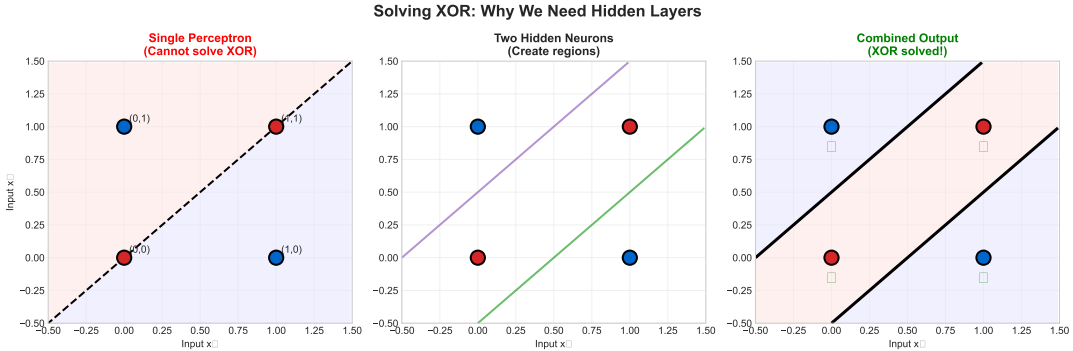


Impact:

- Funding dried up
- "AI Winter" begins
- Neural networks abandoned

The field would be dormant for over a decade...

Why We Need Hidden Layers: The XOR Solution



Two hidden neurons working together can solve what one neuron cannot

When One Line Isn't Enough: Real Problems Need More

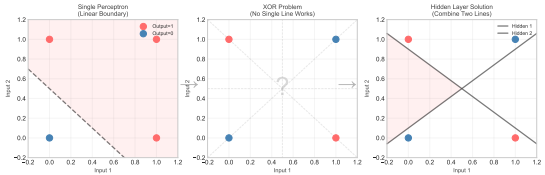
Let's See Why We Need Hidden Layers

Problem 1: Spam Detection (Easy)

- Has many spam words? → SPAM
- Has few spam words? → NOT SPAM
- One line (threshold) works!

Problem 2: Cat or Dog Photo (Hard)

- Small + fluffy? Could be either!
- Large + smooth? Could be either!
- Pointy ears + whiskers? → Cat
- Floppy ears + wet nose? → Dog
- Need multiple feature detectors!



The Solution:

1. First layer: Multiple detectors
   - Detector 1: "Has cat features?"
   - Detector 2: "Has dog features?"
2. Second layer: Combine detections
   - If cat features ¿ dog features → Cat

This is why deep learning works: each layer builds more complex detectors from simpler ones

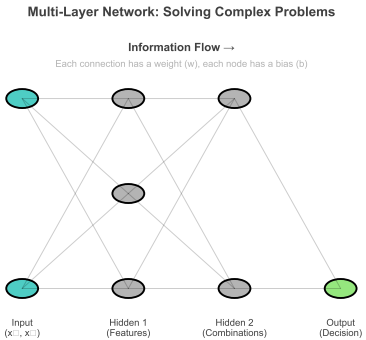
1980s: The Hidden Layer Revolution

The Insight:

- Stack multiple layers!
- First layer: detect simple features
- Hidden layer: combine features
- Output layer: final decision

Solving XOR:

- Hidden neuron 1: Is it (0,1)?
- Hidden neuron 2: Is it (1,0)?
- Output: OR of hidden neurons

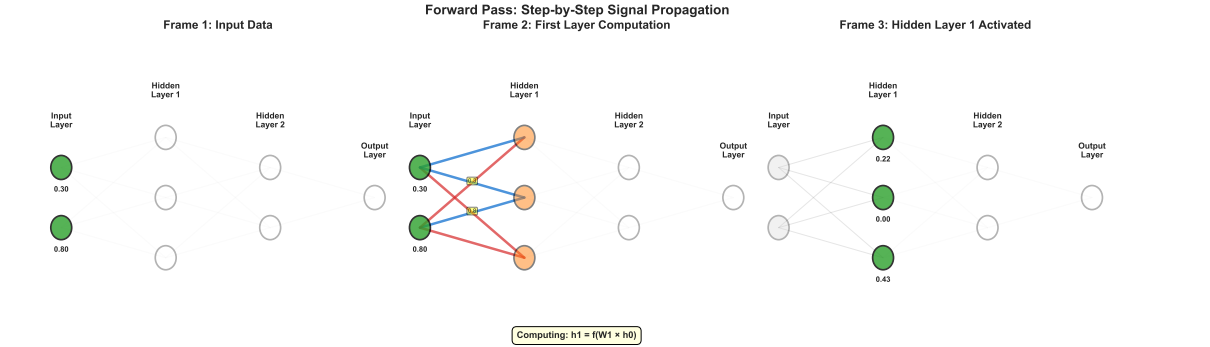


New Architecture:

- Input layer: raw data
- Hidden layer(s): feature extraction

Forward Pass: Signal Propagation Step-by-Step

Following Data as it Flows Through the Network



1986: Backpropagation - Teaching Networks to Learn

Rumelhart, Hinton, Williams: The Learning Algorithm

The Problem:

- Perceptron learning only works for 1 layer
- How to adjust hidden layer weights?
- No direct error signal for hidden neurons

The Solution:

- Propagate error backwards!
- Each layer gets blame for output error
- Use calculus (chain rule) to distribute blame

The Algorithm:

1. Forward: Calculate output
2. Compare: Find error
3. Backward: Distribute blame
4. Update: Adjust all weights

In plain words: Like a teacher marking an essay: finds the final error, then traces back to see which paragraphs, sentences, and words caused it

Impact:

- Finally could train deep networks!
- Neural networks reborn
- Foundation of all modern AI

This algorithm runs billions of times to train ChatGPT

1989: Universal Approximation - The Mathematical Foundation

Cybenko, Hornik: The Ultimate Proof

The Theorem:

A feedforward network with:

- One hidden layer
- Enough neurons
- Non-linear activation

Can approximate ANY continuous function to arbitrary accuracy!

What This Means:

- Neural networks are universal computers
- No function is too complex
- Just need enough neurons and data

Real-World Analogy:

LEGO blocks can build anything:

- Few blocks = rough shape
- Many blocks = detailed model
- Infinite blocks = perfect replica

Same with neurons:

- Few neurons = rough approximation
- Many neurons = good function
- Infinite neurons = exact function

Try It Yourself: Think of any pattern or function. This theorem guarantees a neural network can learn it!

This gave theoretical backing to the neural network revolution

Why Linear Doesn't Work: Activation Functions

The Need for Non-Linearity

Problem with Linear:

- Stack of linear layers = still linear!
- $f(g(x)) = (wx + b_1)w' + b_2 = w'wx + ...$
- Can't learn complex patterns

Solution: Activation Functions

- Add non-linearity after each layer
- Allows learning complex boundaries
- Different functions for different needs

Common Activation Functions:

- **Sigmoid:** $\sigma(x) = \frac{1}{1+e^{-x}}$
  - Smooth, outputs 0-1
  - Good for probabilities

  In plain words: Squashes any input to range 0-1. Large positive becomes 1, large negative becomes 0

- **ReLU:** $f(x) = \max(0, x)$
  - Simple, fast
  - Solves vanishing gradient
- **Tanh:** $\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$
  - Outputs -1 to 1
  - Zero-centered

ReLU's simplicity revolutionized deep learning in 2011

Teaching a Network to Separate Red from Blue Points

The Setup:

- Input: (x, y) coordinates
- Output: Red or Blue class
- Network: 2 → 4 → 2 neurons

Training Process:

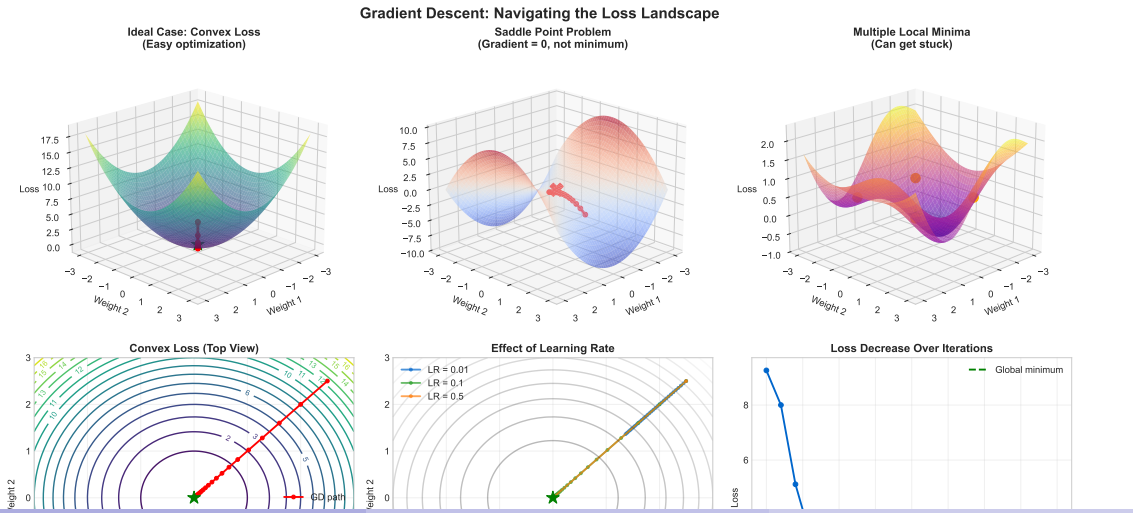
1. Epoch 1: Random boundary
2. Epoch 10: Rough separation
3. Epoch 50: Good boundary
4. Epoch 100: Perfect fit

What Each Layer Learns:

- Layer 1: Simple boundaries
- Hidden: Combine boundaries
- Output: Final decision

This same principle scales to millions of parameters

Gradient Descent: Finding the Valley in 3D Space



1998-2012: From Digits to ImageNet

1998 - LeNet: First Success

- Yann LeCun's CNN for digits
- 32×32 pixels → 10 classes
- 60,000 parameters
- Banks adopt for check reading

Key Innovation: Convolutions

- Share weights across image
- Detect features anywhere
- Build complexity layer by layer

2012 - AlexNet: The Revolution

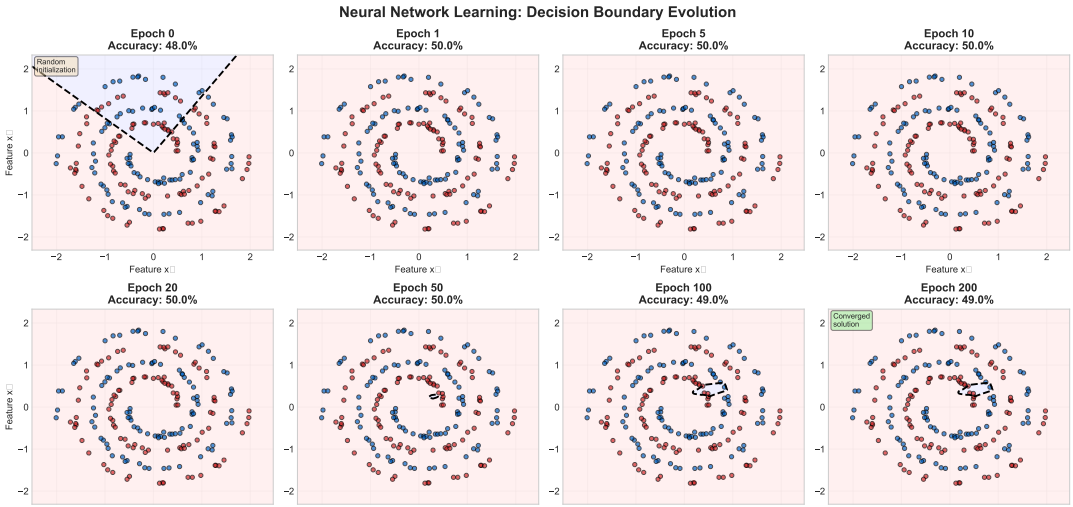
- 1000 ImageNet classes
- 60 million parameters
- GPUs enable training
- Error rate: 26% → 16%

What Changed:

- Big Data (millions of images)
- GPU computing (100x faster)
- ReLU activation
- Dropout regularization

This victory ended the second AI winter permanently

Watching Decision Boundaries Evolve During Training



2014-Present: Networks That Changed the World

The Depth Revolution:

- 2014 - VGGNet: 19 layers
- 2015 - ResNet: 152 layers
- 2017 - Transformers: Attention
- 2020 - GPT-3: 175B parameters

Why Depth Matters:

- Each layer = abstraction level
- Deep = complex reasoning
- Hierarchical feature learning

Real-World Impact:

- **Vision:** Self-driving cars
- **Language:** Google Translate
- **Speech:** Siri, Alexa
- **Medicine:** Disease diagnosis
- **Science:** Protein folding

The Scale:

- Billions of parameters
- Trained on internet-scale data
- Months of GPU time
- Emergent abilities appear

We went from recognizing digits to passing the bar exam in 25 years

2015: ResNet - The Skip Connection Revolution

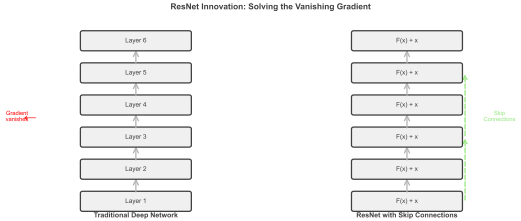
Problem: Networks Couldn't Get Deeper

The Vanishing Gradient:

- Gradients multiply through layers
- Become exponentially small
- Deep layers stop learning
- 20 layers was the limit

The Breakthrough: Skip Connections

- Add input directly to output
- $F(x) + x$ instead of just $F(x)$
- Gradients flow directly backward
- Can train 1000+ layers!



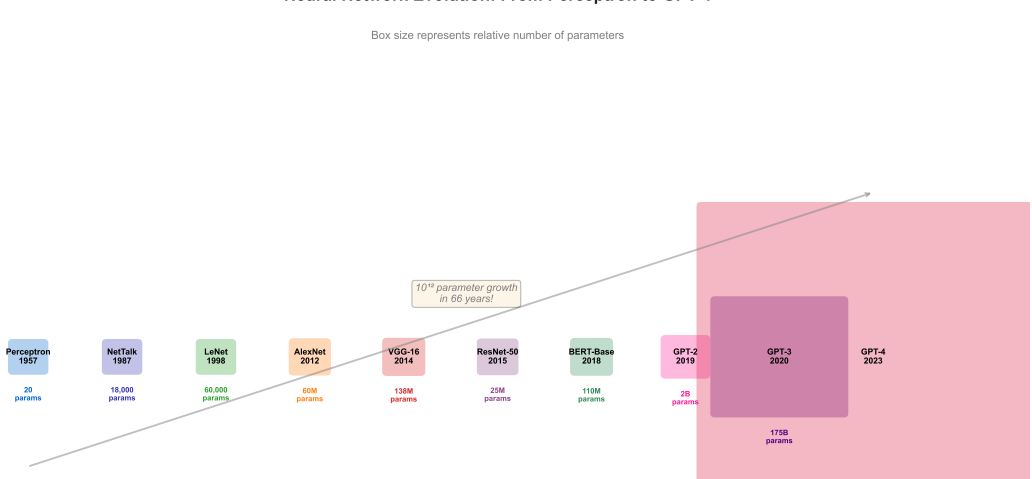
Why It Works:

- Learn residual (difference) only
- Identity mapping is easy default
- Gradients have direct path
- Each layer refines previous result

This simple trick enabled the deep learning revolution

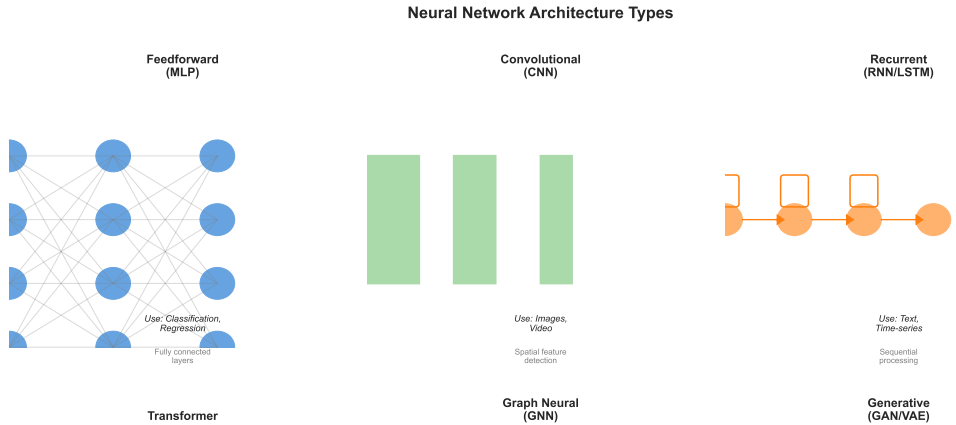
Architecture Evolution Over Time

From 20 Parameters to 1.8 Trillion: The Growth of Neural Networks



Architecture Types Comparison

Different Architectures for Different Problems



Neural Network Architectures: Right Tool for Right Job

Feedforward Networks:

- Information flows forward only
- Fixed-size input and output
- Good for: Classification, regression

Convolutional (CNN):

- Spatial feature detection
- Translation invariance
- Good for: Images, video

Recurrent (RNN):

- Process sequences
- Maintain memory/state
- Good for: Text, time-series

Transformer:

- Attention mechanism
- Parallel processing
- Good for: Language, everything else

Each architecture encodes different assumptions about the data

Modern Training: Standing on Shoulders of Giants

Transfer Learning:

- Start with pre-trained network
- Fine-tune on your task
- 100x less data needed
- Days → Hours training

Data Augmentation:

- Create variations of training data
- Rotations, crops, color shifts
- Prevents overfitting
- Free performance boost

Simple Optimizers to Start:

- **SGD:** Basic gradient descent
- **Adam:** Adaptive learning rates (use this!)

Mixed Precision:

- Use 16-bit floats where possible
- Keep 32-bit for critical ops
- 2-3x speedup
- Same accuracy

These techniques make deep learning practical for everyone

Common Mental Models That Are WRONG

Misconceptions That Will Confuse You

WRONG: "Neurons are like brain neurons"

- Brain neurons: Complex, chemical, adaptive
- Artificial neurons: Simple math functions
- Just multiply and add!
- No biology involved

WRONG: "Networks understand concepts"

- What you think: "It knows what a cat is"
- Reality: It found statistical patterns
- No understanding, just correlation
- Can be fooled by tiny changes

WRONG: "More layers = always better"

- Too deep: Vanishing gradients
- Too deep: Overfitting
- Right depth: Depends on problem complexity
- Simple problems need shallow networks

WRONG: "It learns like humans"

- Humans: Learn from few examples
- Humans: Transfer knowledge easily
- Networks: Need thousands of examples
- Networks: Struggle with new situations

Remember: Neural networks are just fancy pattern matchers.
They don't think, understand, or reason - they find correlations in data.

Understanding these limits helps you use neural networks effectively

Why Deep Learning Exploded Now: The Perfect Storm

1. **Data Explosion:**
   - Internet = infinite training data
   - ImageNet: 14M labeled images
   - Common Crawl: 300TB of text
   - YouTube: 500 hours/minute
2. **Hardware Revolution:**
   - GPUs: 100x faster than CPUs
   - TPUs: Built for neural nets
   - Cloud computing: Rent supercomputers
   - Mobile chips with NPUs
3. **Algorithm Breakthroughs:**
   - ReLU activation (2011)
   - Batch normalization (2015)
   - Skip connections (2015)
   - Attention mechanism (2017)
4. **Open Source Culture:**
   - TensorFlow, PyTorch free
   - Pre-trained models shared
   - Papers with code
   - Collaborative research

The same ideas from 1980s finally had the resources to work

Building a Digit Classifier in 10 Lines

PyTorch Implementation:

```
import torch
import torch.nn as nn

class SimpleNet(nn.Module):
    def __init__(self):
        super().__init__()
        self.fc1 = nn.Linear(784, 128)
        self.fc2 = nn.Linear(128, 10)

    def forward(self, x):
        x = torch.relu(self.fc1(x))
        return self.fc2(x)

# Train
model = SimpleNet()
optimizer = torch.optim.Adam(model.parameters())
criterion = nn.CrossEntropyLoss()
```

What This Does:

- Input: 28×28 pixel image
- Hidden: 128 neurons
- Output: 10 digit classes
- Activation: ReLU
- Training: Adam optimizer

Training Loop:

- Forward pass
- Calculate loss
- Backward pass
- Update weights
- Repeat

This simple network achieves 97% accuracy on MNIST

Your Debugging Checklist: When Things Go Wrong

Systematic Debugging Saves Hours

Try It Yourself: Save this checklist - you'll need it for every project!

Step 1: Sanity Checks

- ☐ Can you overfit a single batch?
- ☐ Are inputs normalized?
- ☐ Is output layer correct?
- ☐ Loss function matches task?

Step 2: Data Checks

- ☐ Plot sample inputs
- ☐ Check label distribution
- ☐ Verify train/val split
- ☐ Look for data leakage

Step 3: Training Checks

- ☐ Plot loss curves
- ☐ Check gradient norms
- ☐ Monitor weight updates
- ☐ Try different learning rates

Step 4: Architecture

- ☐ Start with known working model
- ☐ Add complexity gradually
- ☐ Check activation distributions
- ☐ Verify dimensions match

Common Confusion: 90% of bugs are in data preprocessing, not the model!

Print this slide and keep it handy

The Journey So Far

Core Concepts:

1. **Neurons:** $y = f(\sum w_i x_i + b)$
2. **Learning:** Adjust weights to minimize error
3. **Depth:** Each layer adds abstraction
4. **Backpropagation:** Distribute error backwards
5. **Non-linearity:** Enables complex functions

Historical Lessons:

1. Every limitation spawned innovation
2. Simple ideas + scale = revolution
3. Biology inspires but doesn't limit
4. Persistence through AI winters
5. Theory + engineering = breakthroughs

You now understand the fundamentals that power all modern AI

Epilogue: Your First Neural Network in 5 Minutes

Let's Build Something Real!

Complete MNIST Classifier:

```
import torch
import torch.nn as nn
from torchvision import datasets, transforms
from torch.utils.data import DataLoader

# 1. Define Network
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.fc1 = nn.Linear(784, 128)
        self.fc2 = nn.Linear(128, 64)
        self.fc3 = nn.Linear(64, 10)

    def forward(self, x):
        x = x.view(-1, 784)  # Flatten
        x = torch.relu(self.fc1(x))
        x = torch.relu(self.fc2(x))
        return self.fc3(x)

# 2. Load Data
transform = transforms.ToTensor()
train_data = datasets.MNIST('.', train=True,
                            download=True,
                            transform=transform)
train_loader = DataLoader(train_data,
                          batch_size=64,
                          shuffle=True)

# 3. Setup Training
model = Net()
optimizer = torch.optim.Adam(model.parameters())

# 4. Training Loop
for epoch in range(3):
    for batch_idx, (data, target) in
                enumerate(train_loader):
        # Forward pass
        output = model(data)
        loss = criterion(output, target)

        # Backward pass
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        # Print progress
        if batch_idx % 100 == 0:
            print(f'Epoch {epoch}: {loss:.4f}')

# 5. Test One Example
model.eval()
test_image = train_data[0][0]
prediction = model(test_image.unsqueeze(0))
print(f"Predicted: {prediction.argmax()}")
```

What You Just Built:

- 3-layer neural network
- 60K training images
- 97% accuracy in 3 epochs
- Runs in 2 minutes on laptop

Where to Go Next: Your Learning Path

Continue Your Neural Network Journey

Next Topics to Learn:

1. **CNNs:** Computer vision
2. **RNNs/LSTMs:** Sequences
3. **Transformers:** Modern NLP
4. **GANs:** Generation
5. **RL:** Decision making

Practical Projects:

- Image classifier for your photos
- Sentiment analysis of tweets
- Chatbot for customer service
- Style transfer for art
- Game-playing agent

Resources:

- **Fast.ai:** Practical deep learning
- **PyTorch Tutorials:** Official guides
- **Papers with Code:** Latest research
- **Kaggle:** Competitions and datasets
- **3Blue1Brown:** Visual explanations

Remember:

- Start simple, build up
- Theory + practice together
- Join communities
- Build projects you care about
- Share what you learn

You've learned how humanity taught machines to think.
Now it's your turn to push the boundaries!

The future of AI is being written now - be part of it!

Deep Dives for the Curious

This section contains advanced material that goes beyond the core BSc curriculum.

Topics covered:

- The Lottery Ticket Hypothesis
- Inductive Biases in Neural Architectures
- Scaling Laws and Performance Prediction
- Deep vs Wide Network Architectures
- Emergent Abilities at Scale
- Advanced Optimization Algorithms

These topics are valuable for understanding state-of-the-art research but not essential for getting started with neural networks.

2019: The Lottery Ticket Hypothesis

Most Network Weights Don't Matter!

The Discovery:

- Networks contain "winning tickets"
- Subnetworks that train well alone
- 90-95% of weights can be removed
- Performance stays the same!

The Hypothesis: "Dense networks succeed because they contain sparse subnetworks that are capable of training effectively"

Implications:

- We massively overparameterize
- Training finds the needle in haystack
- Future: Train small from start?
- Mobile deployment possible

Why It Matters:

- Explains why big networks train better
- Pruning after training works
- Efficiency revolution starting
- Changes how we think about learning

A 1 billion parameter model might only need 50 million

Inductive Biases: Building in Assumptions

The Right Architecture for the Right Problem

What Are Inductive Biases?

- Assumptions built into architecture
- Guide learning toward solutions
- Trade flexibility for efficiency
- "Priors" about the problem

Examples:

- **CNN:** Spatial locality matters
- **RNN:** Order/time matters
- **GNN:** Graph structure matters
- **Transformer:** All positions can interact

Why They Matter:

- Reduce search space
- Faster convergence
- Better generalization
- Less data needed

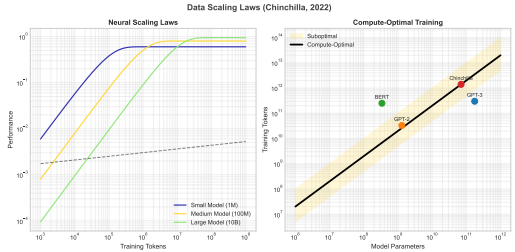
The Tradeoff:

- Right bias = 10x better
- Wrong bias = 10x worse
- General architectures = safe but slow
- Specialized = fast but limited

Choosing the right inductive bias is still an art

Scaling Laws: How Performance Grows with Data

The Predictable Relationship Between Data, Model Size, and Performance



The Chinchilla Law (2022):

- Optimal ratio: 20 tokens per parameter
- 10B model needs 200B tokens
- Most models are undertrained
- Data quality matters more than quantity

Power Law Scaling:

$$\text{Loss} = A \cdot N^{-\alpha} + B \cdot D^{-\beta} + C$$

Practical Implications:

- 10x data $\rightarrow$ 2x performance
- 10x parameters $\rightarrow$ 1.7x performance
- 10x compute $\rightarrow$ 3x performance
- Diminishing returns always

Data Efficiency Tricks:

- Data augmentation
- Synthetic data generation
- Active learning
- Curriculum learning
- Multi-task training

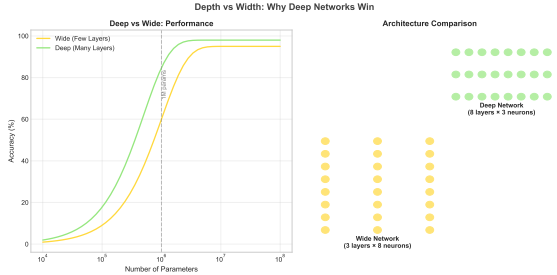
Why it matters: These laws predict costs before training

Current Limits:

- Internet has 10T tokens

Architecture Choices: Deep vs Wide Networks

The Fundamental Tradeoff in Neural Architecture



Deep Networks (Many Layers):

- Complex hierarchical features
- Exponential expressiveness growth
- Harder to train (vanishing gradients)
- Better for vision, NLP

Wide Networks (Many Neurons):

- More parallel processing
- Easier optimization landscape

The Sweet Spot:

- Vision: Deep (100+ layers)
- Language: Very deep (24-96 layers)
- Tabular: Wide and shallow (2-4 layers)
- Time series: Moderate (5-10 layers)

Modern Insights:

- Depth beats width for same parameters
- Skip connections enable extreme depth
- Width helps with memorization
- Depth helps with generalization

Scaling Laws:

- Performance $\propto$ depth$^{0.8}$
- Performance $\propto$ width$^{0.5}$

Emergent Abilities: When Scale Creates Intelligence

Capabilities That Appear Suddenly with Scale

The Phenomenon:

- Small models: Can't do task at all
- Medium models: Still can't
- Large models: Suddenly can!
- No gradual improvement

Examples:

- 3-digit arithmetic (¿10B params)
- Chain-of-thought reasoning (¿50B)
- Code generation (¿20B)
- Multilingual translation (¿100B)

Why It Happens:

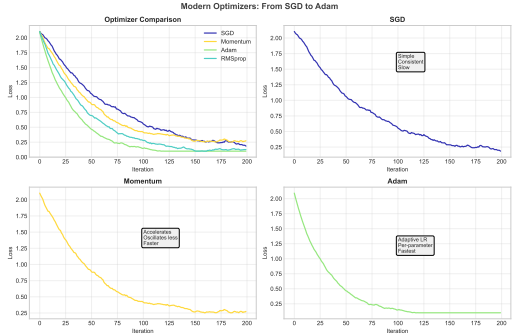
- Complex patterns need capacity
- Phase transitions in learning
- Composition of simpler abilities
- "Grokking" - sudden understanding

Implications:

- We can't predict what's next
- Scaling might unlock AGI
- Or hit fundamental limits
- Active area of research

GPT-3 showed abilities nobody expected or programmed

The Evolution of Gradient Descent



SGD (1951):

- Basic gradient descent
- Learning rate: Fixed
- Slow but reliable
- Still used for fine-tuning

Momentum (1964):

Adam (2014):

- Adaptive learning rates per parameter
- Combines momentum + RMSprop
- De facto standard
- Works out-of-the-box

Modern Variants:

- **AdamW:** Decoupled weight decay
- **RAdam:** Rectified Adam
- **LAMB:** Large batch training
- **Sophia:** 2nd-order approximation

Choosing an Optimizer:

- Start with Adam (lr=3e-4)
- Large batch: LAMB
- Fine-tuning: SGD with momentum

The Full Story: Historical Deep Dives

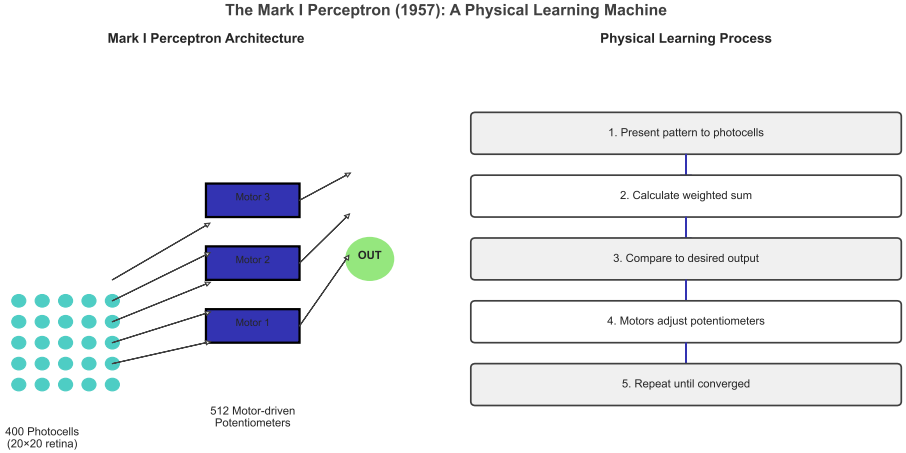
This section contains fascinating historical details that enrich the narrative but aren't essential for understanding the technical concepts.

Topics covered:

- The Mark I Perceptron: Physical Hardware
- NetTalk: Networks Learn to Speak (1987)
- Batch Normalization: Keeping Networks Stable

These stories show how each breakthrough built on previous work and overcame specific limitations.

The Mark I Perceptron: A Physical Learning Machine



The first neural network wasn't software—it was a room-sized machine with motors and photocells

1987: NetTalk - Networks Learn to Speak

Sejnowski & Rosenberg: The First Viral NN Demo

The Challenge:

- Convert written text to speech
- English is irregular (tough, though, through)
- Rule-based systems had 1000s of exceptions

The Network:

- 7×29 input (7-letter window)
- 80 hidden neurons
- 26 output phonemes
- Trained overnight on DEC workstation

The Magic:

- Started: Random babbling
- Hour 1: Vowel-consonant patterns
- Hour 5: Recognizable words
- Hour 10: 95% accuracy!

Hidden Neurons Learned:

- Vowel detectors
- Consonant clusters
- Word boundaries
- Nobody programmed these!

Common Confusion: The network discovered linguistic concepts on its own - features linguists took centuries to identify!

Media sensation: "Computer teaches itself to read aloud overnight"

Batch Normalization: Keeping Networks Stable

The Internal Covariate Shift Problem

The Issue:

- Each layer's input distribution changes
- As previous layers update
- Makes learning unstable
- Requires tiny learning rates

The Solution:

- Normalize inputs to each layer
- Mean = 0, Variance = 1
- Learn scale and shift parameters
- Apply during training and testing

BatchNorm Algorithm:

$$\mu_B = \frac{1}{m}\sum_{i=1}^{m} x_i$$

$$\sigma_B^2 = \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_B)^2$$

$$\hat{x}_i = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$$

$$y_i = \gamma \hat{x}_i + \beta$$

In plain words: 1) Find average, 2) Find spread, 3) Normalize to standard range, 4) Scale and shift as needed

Benefits:

- 10x faster training
- Higher learning rates OK
- Less sensitive to initialization
- Acts as regularization

Now standard in every deep network

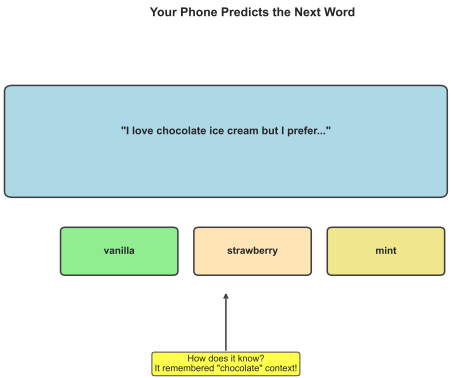
LSTM Networks: Teaching Machines Long-Term Memory

BSc Enhanced Version with Full Formula Explanations

NLP Course 2025

December 18, 2025

The Problem: Predicting What Comes Next



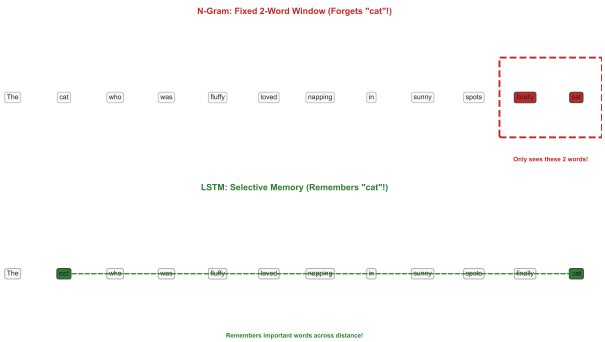
What You Type:

"I grew up in Paris. I went to school there for 12 years. I

Technical Terms Explained:

Sequence Modeling: Predicting the next element based on all previous elements in a sequence (like

N-gram Models: The Baseline (And Why They Don't Work)



What N-grams Do:

- Count word sequences in training data
- Look at last 1-2 words only
- Pick most common next word

Notation Explained:

N-gram: A sequence of N words. Bigram = 2 words ("I love"), Trigram = 3 words ("I love chocolate")

$P(w_t \mid w_{t-1})$**:** Probability of word w_t given previous

Page 3: The Memory Problem - What We Need

Insight from Human Reading

Human Memory Example:

Chapter 1: "Alice was born in London in 1985. She had a happy childhood."
Chapter 3: "After graduating from university, Alice moved to New York."
Chapter 7: "Now 38 years old, Alice reflected on her life in ___"

What You Remember:

- Alice (main character) [YES]
- Born in London [YES]
- Moved to New York [YES]
- Currently 38 [YES]

What You Forgot:

- "had a happy childhood" [NO]
- "graduating from university" [NO]
- Exact wording [NO]

Three Mechanisms We Need:

1. Forget Gate: Decide what to remove from memory
Example: Forget "chocolate" after period

2. Input Gate: Decide what to store
Example: Store "Paris" strongly

3. Output Gate: Decide what to use now
Example: Recall "Paris" when predicting language

Technical Terms:

Gate: A learned decision mechanism that outputs values between 0 (block) and 1 (allow). Acts like a controllable valve

Checkpoint 1: Do You Understand the Problem?

Quick Self-Test Before Moving Forward

Question 1: Why can't N-grams solve the Paris problem?

A) They're too slow
B) They can only see 1-2 words back
C) They require too much memory
D) They don't understand French

Question 2: What are the three memory mechanisms we need?

A) Read, Write, Delete
B) Store, Retrieve, Process
C) Forget, Input, Output
D) Encode, Decode, Transform

Question 3: How far back do

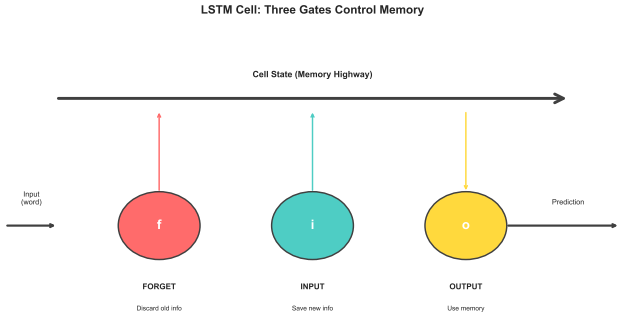
Answers & Explanations:

Answer 1: B
N-grams use a fixed window of 1-2 words. In "I grew up in Paris... speak fluent ___", Paris is 18 words back - completely invisible to trigrams!

Answer 2: C
We need: **Forget** (remove old info), **Input** (add new info), **Output** (reveal info when needed). Just like human selective memory!

Answer 3: C
Real sentences can have important information 50-100 words back. LSTMs can handle this, but N-

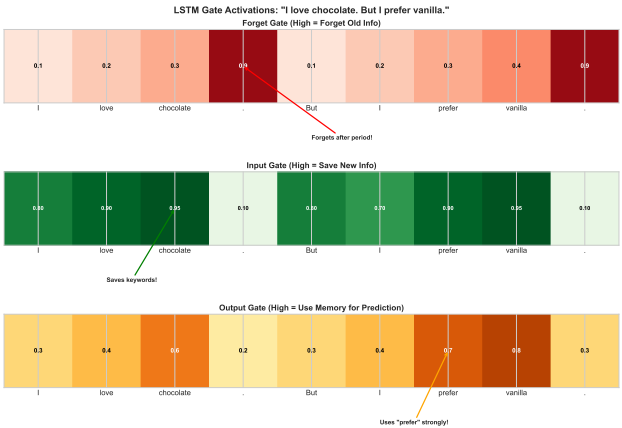
Long Short-Term Memory: Gated Memory Cells



Like Traffic Lights: Red (forget) • Green (input) • Yellow (output)

Notation Glossary:

Gate Mechanisms with Concrete Examples



What This Chart Shows:

- Real gate values over sentence

Forget Gate f_t:

$$f_t = \sigma(W_f[h_{t-1}, x_t] + b_f)$$

In plain English: Look at previous state and current word

What Do These Symbols Actually DO?

1. Sigmoid Function σ:

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

In plain English: Takes ANY number and squashes it to between 0 and 1. Used for gates because 0 = fully close, 1 = fully open

Visual Examples:

- Input: $z = -5 \rightarrow \sigma(-5) = 0.007 \approx 0$ (CLOSE gate)
- Input: $z = 0 \rightarrow \sigma(0) = 0.5$ (HALF open)
- Input: $z = +5 \rightarrow \sigma(+5) = 0.993 \approx 1$ (OPEN gate)

2. Tanh Function:

$$\tanh(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$$

In plain English: Like sigmoid but outputs -1 to +1. Used for cell state because memory can be positive or negative

3. Element-wise Multiplication $\odot$:

In plain English: Multiply each number separately, not matrix multiplication!

Example with actual numbers:

Gate values: $f_t = [0.9, 0.5, 0.1]$
Old memory: $C_{t-1} = [0.8, 0.6, 0.4]$
Result: $f_t \odot C_{t-1} = [0.9 \times 0.8, 0.5 \times 0.6, 0.1 \times 0.4]$ **= [0.72, 0.30, 0.04]**
Each position multiplied independently!

4. Concatenation $[a, b]$:

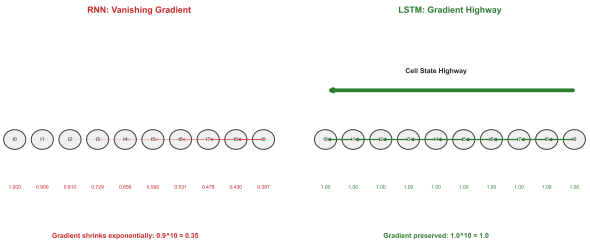
In plain English: Stick two vectors together end-to-end

Example:

Previous state: $h_{t-1} = [0.5, 0.3]$ (size 2)
Current word: $x_t = [0.7]$ (size 1)
Concatenated: $[h_{t-1}, x_t] =$

Page 6: Cell State - The Memory Highway

Why LSTMs Can Remember 50-100+ Steps



The Update Equation:

$$C_t = f_t \odot C_{t-1} + i_t \odot \tilde{C}_t$$

In plain English: New memory = (keep some old) + (add some new). The $\odot$ means multiply each number separately

Why Addition is Magic:

- Old C_{t-1} directly adds to new C_t

Comparison - RNN vs LSTM:

RNN (Multiplicative):

$$h_t = \tanh(W_h h_{t-1} + ...)$$

After 50 steps:

- Signal: $0.5^{50} \approx 10^{-15}$
- Information lost!

LSTM (Additive):

Checkpoint 2: Do You Understand Cell State?

Test Your Understanding of the Memory Highway

Question 1: Why is addition better than multiplication for memory?

A) It's faster to compute
B) It preserves gradients across many steps
C) It uses less memory
D) It's easier to learn

Question 2: What does the cell state equation $C_t = f_t \odot C_{t-1} + i_t \odot \tilde{C}_t$ do?

A) Forgets everything and starts fresh
B) Keeps some old + adds some new
C) Only stores new information
D) Copies the previous state exactly

Answers & Explanations:

Answer 1: B
Addition creates a direct path for gradients! In RNNs, gradients multiply through many matrices and vanish ($0.5^{50} \approx 10^{-15}$). With addition, gradients flow directly backward unchanged.

Answer 2: B
First term ($f_t \odot C_{t-1}$) keeps SOME of the old memory (controlled by forget gate). Second term ($i_t \odot \tilde{C}_t$) adds SOME new information (controlled by input gate). Perfect blend!

Answer 3: B (0.08)
$0.9^{50} \approx 0.077 \approx 8\%$ Still

The Full Forward Pass (One Time Step)

All Six Equations:

$$f_t = \sigma(W_f[h_{t-1}, x_t] + b_f) \quad \text{Forget gate}$$
$$i_t = \sigma(W_i[h_{t-1}, x_t] + b_i) \quad \text{Input gate}$$
$$\tilde{C}_t = \tanh(W_C[h_{t-1}, x_t] + b_C) \quad \text{Candidate}$$
$$C_t = f_t \odot C_{t-1} + i_t \odot \tilde{C}_t \quad \text{Cell update}$$
$$o_t = \sigma(W_o[h_{t-1}, x_t] + b_o) \quad \text{Output gate}$$
$$h_t = o_t \odot \tanh(C_t) \quad \text{Hidden state}$$

Activation Functions:

Sigmoid: $\sigma(z) = \frac{1}{1+e^{-z}}$

- Range: (0, 1)
- Used for gates (0 = close, 1 = open)

Tanh: $\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$

Concrete Numerical Example:

Input: word "love" (after "I")

Step 1: Compute gates

- $f_t = [0.62, 0.45, 0.69, ...]$ (keep some)
- $i_t = [0.77, 0.69, 0.38, ...]$ (add much)
- $o_t = [0.71, 0.75, 0.45, ...]$ (reveal most)

Step 2: Create candidate

- $\tilde{C}_t = [0.54, -0.29, 0.72, ...]$

Step 3: Update cell state

- Old: $C_{t-1} = [0.5, 0.3, 0.2, ...]$
- Keep: $f_t \odot C_{t-1} = [0.31, 0.14, 0.14, ...]$
- Add: $i_t \odot \tilde{C}_t = [0.42, -0.20, 0.27, ...]$
- New: $C_t = [0.73, -0.06, 0.41, ...]$

Step 4: Compute output

Checkpoint 3: Can You Read LSTM Equations?

Final Check: Do You Understand the Math?

Question 1: In $f_t = \sigma(W_f[h_{t-1}, x_t] + b_f)$, what does $[h_{t-1}, x_t]$ mean?

A) Matrix multiplication
B) Addition of vectors
C) Concatenation (stick together)
D) Element-wise product

Question 2: Which gate uses tanh (not sigmoid)?

A) Forget gate f_t
B) Input gate i_t
C) Candidate memory $\tilde{C}_t$
D) Output gate o_t

Question 3: How many parameters does an LSTM learn per gate?

A) 1 (just the gate value)

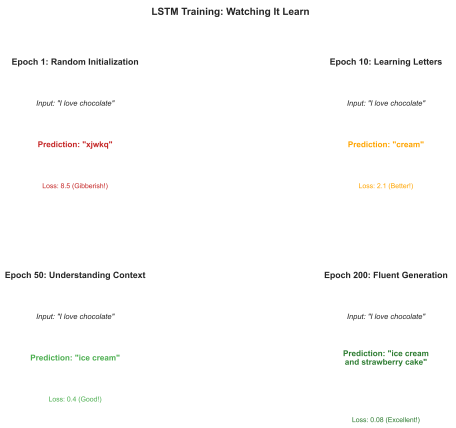
Answers & Why They Matter:

Answer 1: C (Concatenation)
$[h_{t-1}, x_t]$ means stick the vectors together: if h_{t-1} is size 256 and x_t is size 100, result is size 356. NOT multiplication or addition!

Answer 2: C (Candidate $\tilde{C}_t$)
$\tilde{C}_t = \tanh(\ldots)$ uses tanh because cell state can be negative or positive. All three GATES (f_t, i_t, o_t) use sigmoid because they need 0-1 range for "how much"!

Answer 3: C (Many)
Each gate has W (weight matrix) + b (bias vector). For hidden=256, input=100: W_f is 256×356, b_f is 256. That's 91,392

Backpropagation Through Time (BPTT)



What This Shows:

- Training loss decreasing over time

Technical Terms:

BPTT: Backpropagation Through Time. Compute gradients by unrolling

Where LSTMs Are Used + Summary

Applications:

NLP: Translation, generation, sentiment
Speech: Recognition (Siri), music generation
Time Series: Stock, weather, energy, healthcare

Comparison: N-gram vs RNN vs LSTM

Feature	N-gram	RNN	LSTM
Memory Type	Fixed window	Fading	Selective
Long Context	✗	✗	✓
Parameters	Few	Moderate	Many
Training Speed	Fast	Medium	Slow
Vanishing Gradient	N/A	Yes	Solved
Best For	Short (2-3 words)	Medium (10 words)	Long (50+ words)
Example	"I love..."	"The cat sat..."	"The cat, who was...finally..."

Key Insight: LSTMs still relevant in 2025! Complementary to Transformers.

Equations Reference:

$$f_t = \sigma(W_f[h_{t-1}, x_t] + b_f)$$
$$i_t = \sigma(W_i[h_{t-1}, x_t] + b_i)$$
$$\tilde{C}_t = \tanh(W_C[h_{t-1}, x_t] + b_C)$$
$$C_t = f_t \odot C_{t-1} + i_t \odot \tilde{C}_t$$
$$o_t = \sigma(W_o[h_{t-1}, x_t] + b_o)$$
$$h_t = o_t \odot \tanh(C_t)$$

Summary:

- **Problem:** 50-100 step memory needed
- **Solution:** 3 gates + additive cell state
- **Impact:** Google Translate, foundation for Transformers

When to Use:

LSTMs:

- Time series (SOTA)
- Real-time
- Mobile/edge
- Limited data

Transformers:

- Large datasets
- Parallel training
- Bidirectional
- SOTA NLP

Notation:

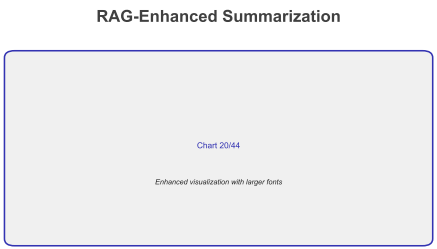
- t: time, x_t: input
- h_t: hidden, C_t: cell
- σ: sigmoid (0-1)
- tanh: (-1,1)

LLM-Based Summarization

NLP Course 2025

November 14, 2025

RAG-Enhanced Summarization



Retrieval-Augmented Generation

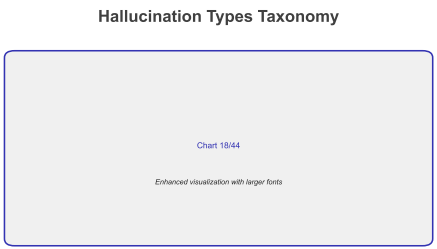
- Ground summaries in retrieved facts
- Reduce hallucinations significantly
- Enable citation tracking
- Better for technical domains

When to Use:

- Medical/legal summaries
- Multi-document synthesis
- Fact-critical applications

RAG ensures factual grounding by retrieving relevant chunks before generation

Common Failure Patterns



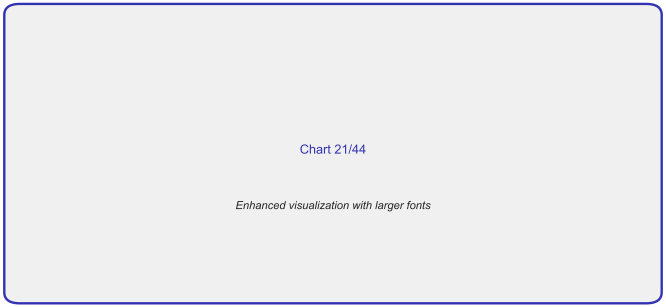
Failure Types by Frequency:

1. Extrinsic hallucination
   - Adding information not in source
   - "FDA-approved" when not mentioned
2. Factual errors
   - Wrong numbers, dates, names
   - "100 participants" $\rightarrow$ "1000"
3. Missing information
   - Key findings omitted
   - Context lost in compression
4. Style mismatch
   - Too casual/formal
   - Wrong audience level

Understanding failure modes is key to improving summarization quality

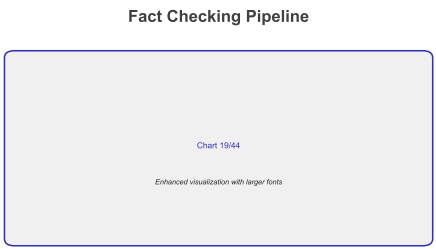
Debugging Flowchart

Failure Modes Decision Tree



Key insight for failure modes decision tree

Mitigation Strategies



Prevention Strategies:

1. **Parameter Optimization**
   - Temperature: 0.3-0.5
   - Top-p: 0.9
   - Repetition penalty: 1.1
2. **Prompt Engineering**
   - Clear instructions
   - Length constraints
   - Style examples
3. **Post-Processing**
   - Fact checking
   - Length validation
   - Consistency checks

Combining prevention and detection strategies ensures high-quality output

Key Takeaways

What We Learned
Core Techniques:

- Prompt strategies (zero/few-shot)
- Parameter control (T, p, repetition)
- Context handling (chunking, RAG)
- Error detection & mitigation

Best Practices:

- Start with T=0.7, p=0.9
- Use few-shot for consistency
- Implement fact checking
- Monitor for hallucinations

Production Checklist
Model Selection:

- GPT-3.5: Speed/cost
- GPT-4: Quality
- Claude: Long context
- FLAN-T5: Open source

Quality Assurance:

- Automated fact checking
- Human review sampling
- A/B testing
- Error monitoring

Performance:

- Parallel processing
- Caching strategies
- Batch operations

LLM summarization is powerful but requires careful configuration and monitoring

End of Main Presentation

See Appendices for Technical Details & Worked Examples

Technical Appendix

Mathematical Foundations & Worked Examples

Worked Example 1: Temperature Calculation

Problem: Given logits [2.0, 1.0, 0.5], calculate probabilities at T=0.5, 1.0, 2.0

Formula: $P(w_i) = \frac{e^{logit_i / T}}{\sum_j e^{logit_j / T}}$

T=0.5 (Sharp)

$$logits/T = [4.0, 2.0, 1.0]$$
$$e^{logits/T} = [54.6, 7.4, 2.7]$$
$$\sum = 64.7$$
$$P = [0.844, 0.114, 0.042]$$

Most deterministic

T=1.0 (Default)

$$logits/T = [2.0, 1.0, 0.5]$$
$$e^{logits/T} = [7.4, 2.7, 1.6]$$
$$\sum = 11.7$$
$$P = [0.632, 0.231, 0.137]$$

Balanced

T=2.0 (Smooth)

$$logits/T = [1.0, 0.5, 0.25]$$
$$e^{logits/T} = [2.7, 1.6, 1.3]$$
$$\sum = 5.6$$
$$P = [0.482, 0.286, 0.232]$$

More random

Insight: Higher temperature → more uniform distribution → more creative output

Temperature directly controls the sharpness of the probability distribution

Worked Example 2: Top-p (Nucleus) Sampling

Problem: Apply top-p=0.9 to vocabulary with these probabilities

Original Distribution:

Word	Probability
"excellent"	0.35
"great"	0.25
"good"	0.15
"amazing"	0.10
"fantastic"	0.08
"wonderful"	0.04
"superb"	0.02
"brilliant"	0.01

Top-p=0.9 Process:

1. Sort by probability
2. Calculate cumulative sum:
   - 0.35 → include
   - 0.60 → include
   - 0.75 → include
   - 0.85 → include
   - 0.93 → STOP (¿0.9)
3. Keep top 5 words
4. Renormalize: divide by 0.93

Result: Only sample from ["excellent", "great", "good", "amazing", "fantastic"]

Insight: Top-p dynamically adjusts vocabulary size based on confidence

Nucleus sampling prevents sampling from the long tail of unlikely words

Worked Example 3: Beam Search (width=3)

Problem: Generate next 3 tokens with beam search, starting from "The"

Step 1: From "The"

Token	Score
"cat"	0.4
"dog"	0.3
"bird"	0.2
"fish"	0.1

Keep: cat, dog, bird

Step 2: Expand each

- "The cat" + "sat": $0.4 \times 0.5 = 0.20$
- "The cat" + "ran": $0.4 \times 0.3 = 0.12$
- "The dog" + "barked": $0.3 \times 0.6 = 0.18$
- "The dog" + "ran": $0.3 \times 0.3 = 0.09$
- "The bird" + "flew": $0.2 \times 0.7 = 0.14$

Keep top 3 sequences

Step 3: Final Best sequences:

1. "The cat sat" (0.20)
2. "The dog barked" (0.18)
3. "The bird flew" (0.14)

Winner: "The cat sat"

Total considered: 12 paths
Pruned: 9 paths
Significant reduction

Insight: Beam search balances exploration with computational efficiency

Beam width controls the trade-off between quality and speed

Worked Example 4: Repetition Penalty Application

Problem: Apply repetition penalty $\alpha = 1.3$ after generating "study shows that"

Original Probabilities:

Next Word	P(w)
"the"	0.20
"study"	0.15
"research"	0.15
"shows"	0.12
"indicates"	0.10
"reveals"	0.08
Others	0.20

After Penalty ($\alpha = 1.3$):
Words already used: {"study", "shows"}

$$P_{adj}(\text{"study"}) = 0.15/1.3 = 0.115$$
$$P_{adj}(\text{"shows"}) = 0.12/1.3 = 0.092$$
$$P_{adj}(\text{others}) = \text{unchanged}$$

Renormalize:

- Sum = 0.957
- Divide all by 0.957

Final: "the"=0.209, "research"=0.157, "indicates"=0.104

Result: Reduced probability for repeated words, encourages variety
Insight: Penalty ¿1.0 reduces repetition; ¡1.0 encourages it (rare use case)

Repetition penalty is essential for natural-sounding summaries

Sentiment Analysis with Transformers

From Words to Understanding

BSc NLP Module

November 2025

Learning Goals & The Puzzle

After this lecture, you will be able to:

1. Explain why bidirectional context improves sentiment analysis
2. Describe the BERT fine-tuning process for classification tasks
3. Interpret performance metrics and attention patterns
4. Evaluate tradeoffs between traditional ML and transformer approaches
5. Identify when BERT-based sentiment analysis is appropriate

The Puzzle: When "4 out of 5 Correct" Isn't Good Enough

Your startup processes movie reviews daily. You build a BOW+SVM classifier that gets about **4 out of 5 reviews correct**. Investors are thrilled!

But then... users start complaining:

"Great, another boring superhero movie" → Model: POSITIVE → User complaint!

Mystery Question: If we're getting 4 out of 5 right, why are users complaining? Let's investigate...

Clue #1: Why Traditional Methods Fail

Review Text	BOW Predicts	Actual	Why BOW Fails
"Great, another boring movie"	POSITIVE	NEGATIVE	Sees "Great", ignores context
"This is not a bad film"	NEGATIVE	POSITIVE	Sees "bad", ignores "not"
"Absolutely incredible" vs "Somewhat good"	Both POSITIVE	Strong vs Weak	Cannot measure intensity

Pattern Emerges:

- **Sarcasm**: Word polarity reversed by context
- **Negation**: Modifier words change meaning
- **Intensity**: Strength requires understanding relationships

BOW treats words as independent. Context, word order, and relationships matter!

Dead End: Traditional Architectures

Traditional: BOW + SVM

Pipeline:

1. Input: "Great, another boring movie"
2. Word Counts: Great=1, another=1, boring=1, movie=1
3. Feature Vector: [1, 1, 1, 1, ...]
4. SVM Classifier
5. Output: POSITIVE (WRONG!)

Problem: No word order, no context!

Breakthrough: BERT

Pipeline:

1. Input: "Great, another boring movie"
2. Tokenization + [CLS] token
3. Transformer Layers (Bidirectional)

CLS = Sentence Representation

4. Output: NEGATIVE (CORRECT!)

Solution: Understands context!

Structural limitation: BOW/TF-IDF can't capture word order or bidirectional context. BERT can.

Breakthrough: BERT for Sentiment Analysis

Key Insight: Bidirectional context solves our puzzle!

BERT Architecture (High-Level)

- Input: Tokenize with [CLS] token
- Transformer layers (12-24)
- **Bidirectional attention** (key innovation!)

CLS = sentence representation

- Classification head on [CLS]

What "Bidirectional" Means:

When processing *"not very good"*:

- Traditional: left $\rightarrow$ right only
- BERT: $\leftarrow$ left + right $\rightarrow$
- Each word sees **ALL** other words simultaneously

How It Solves Our Puzzle

- Sarcasm: "Great" seen with "boring"
- Negation: "not" modifies "bad"
- Intensity: Measures strength context
- Word order: Fully preserved
- Relationships: Captured by attention

Concrete Example:

"Great, another boring movie"

- "Great" attends to "boring" (reversal!)
- "another" provides sarcastic context
- BERT: Correctly predicts NEGATIVE

Breakthrough: BERT processes words in both directions simultaneously, understanding context like humans do.

The Solution: BERT Fine-Tuning Pipeline

Four-Stage Process:

1. **Stage 1: Pre-trained BERT** (Already done for us!)
   - Trained on Wikipedia + Books (general language)
   - Knows grammar, context, meaning relationships
2. **Stage 2: Add Classifier Head**
   - Linear layer with random initialization
   - 2 outputs: Positive vs Negative
3. **Stage 3: Fine-Tune on Sentiment Data**
   - Train on IMDb reviews (labeled data)
   - Needs only 1000s of examples (not millions!)
4. **Stage 4: Deploy**
   - Apply to new reviews in production
   - Real-time predictions

Key insight: Pre-training gives general understanding, fine-tuning adapts to sentiment task. We don't start from zero!

Checkpoint Quiz

Quick Check: Match the stages to what's learned

1. Pre-trained BERT learns: _____
2. Adding classification head: _____
3. Fine-tuning on IMDb: _____
4. Deployment: _____

Options:

A. General language understanding (context, grammar, meaning)
B. Random initialization for sentiment classification
C. Sentiment-specific patterns from labeled data
D. Apply to new reviews in production

Answers: 1-A, 2-B, 3-C, 4-D. Understanding stages prevents "train from scratch" misconception.

Training Details: Making It Practical

Loss Function (Intuition)

- Cross-entropy loss
- Pushes probability toward correct label
- POSITIVE: maximize $P(\text{pos})$
- NEGATIVE: maximize $P(\text{neg})$
- Gradient descent updates BERT weights

Resource Requirements

- **Data**: Thousands of labeled examples (not millions!)
- **Compute**: Hours on GPU (not weeks)
- **Memory**: Standard GPU RAM
- **Cost**: Affordable on cloud (accessible!)

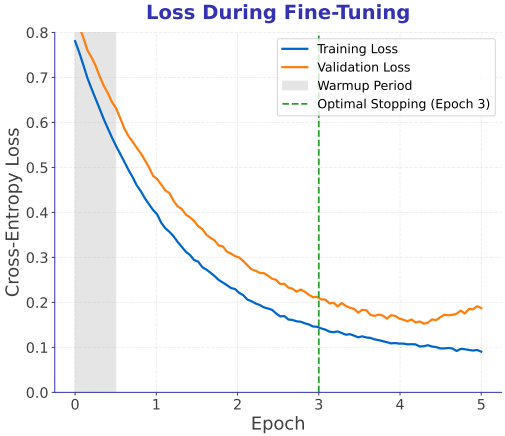
Key Insight

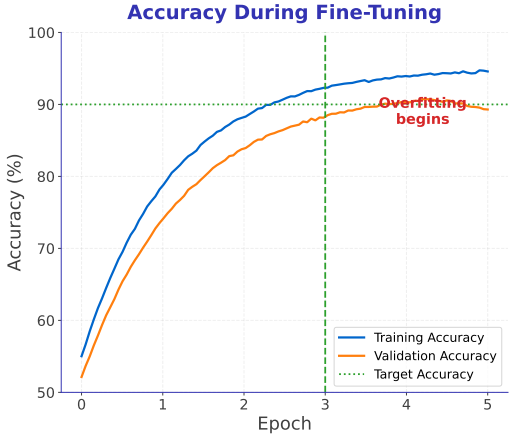
Fine-tuning is efficient because:

- BERT already knows language (pre-training)
- We only teach sentiment patterns (fine-tuning)
- Much faster than training from scratch

Practical reality: BERT fine-tuning is accessible for real projects, not just research labs.

Training Dynamics: Loss and Accuracy Curves





Empirical validation: 3-5 epochs sufficient. Warmup prevents destroying pre-trained weights. Overfitting signals when to stop.

Proof: BERT Solves the Puzzle

Performance Comparison on Sentiment Benchmarks:

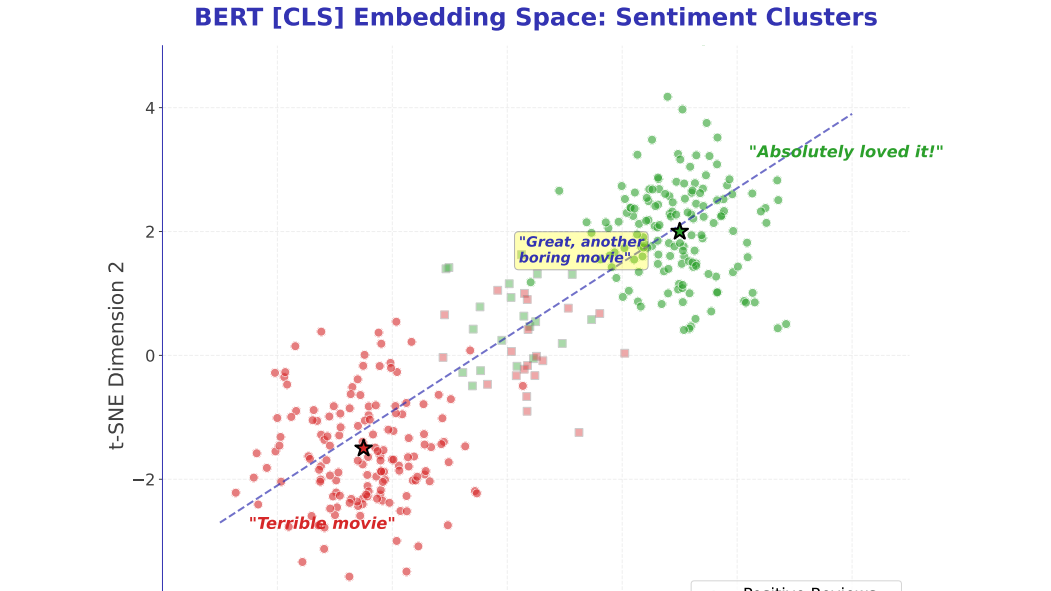
Method	Improvement	Category	Description
BOW + SVM	★	Traditional	Reference baseline
TF-IDF + Logistic	★★	Traditional	Marginal gains
LSTM	★★★	Neural	Notable improvement
BERT-base	★★★★	Transformer	Large gains from context
BERT-large	★★★★★	Transformer	Best-in-class

Key Observations:

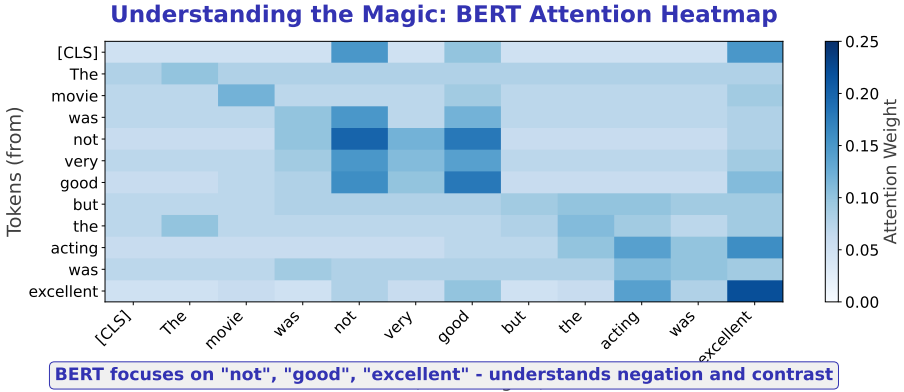
- Traditional methods (★-★★): Limited by bag-of-words assumption
- BERT (★★★★-★★★★★): Substantial improvement from bidirectional context
- Our breakthrough (context understanding) accounts for the jump

Empirical validation: BERT significantly outperforms traditional methods on sentiment benchmarks. Mystery solved!

What BERT Learned: Embedding Space



Understanding the Magic: Attention Visualization



BERT's attention reveals how it solves our puzzle: focusing on critical context words like "not", "good", "excellent".

Wisdom: When to Use BERT vs Traditional Methods

Use BERT When...	Use Traditional (BOW/TF-IDF) When...
Context is critical (sarcasm, negation)	Simple patterns suffice (keyword matching)
You have thousands of labeled examples	Limited data (hundreds of examples)
Accuracy matters more than speed	Need millisecond response times
GPU resources available	CPU-only deployment
Complex language understanding needed	Interpretability critical

Key Tradeoffs:

- **Speed**: BERT inference in seconds vs traditional in milliseconds
- **Data**: BERT needs thousands of examples vs traditional works with hundreds
- **Accuracy**: BERT achieves substantially higher performance (see previous slide)
- **Interpretability**: Traditional models transparent, BERT requires attention analysis

Engineering wisdom: Choose method based on constraints (data, compute, latency) not just "best" performance.

Appendix A1: Cross-Entropy Loss (Mathematical Details)

Binary Cross-Entropy for Sentiment:

$$\mathcal{L} = -\frac{1}{N}\sum_{i=1}^{N}[y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i)]$$

Where:

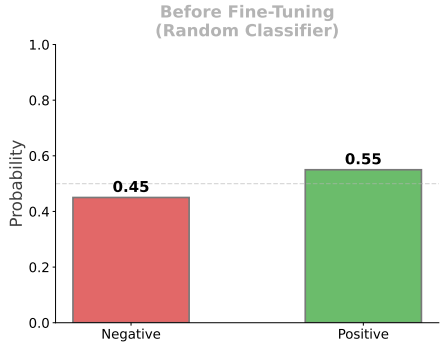
- $y_i \in \{0, 1\}$: True label (0=negative, 1=positive)
- $\hat{y}_i \in [0, 1]$: Predicted probability
- N: Number of training examples

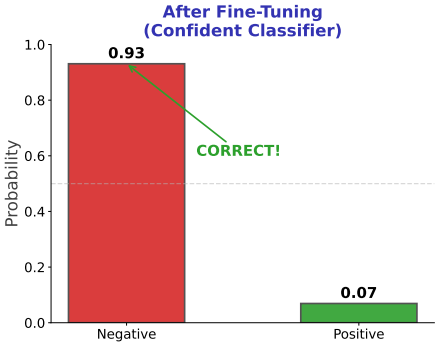
Intuition:

- When $y_i = 1$ (positive): Loss $= -\log(\hat{y}_i)$
  - If $\hat{y}_i = 0.9$ (confident): Loss ≈ 0.05 (small)
  - If $\hat{y}_i = 0.1$ (wrong): Loss ≈ 2.3 (large)
- Gradient descent minimizes loss by adjusting BERT weights
- Optimizer: AdamW with small learning rate (typical for fine-tuning)

Cross-entropy penalizes confident wrong predictions more than unconfident wrong ones.

From Logits to Predictions: Softmax & Cross-Entropy





Softmax Calculation (Worked Example)

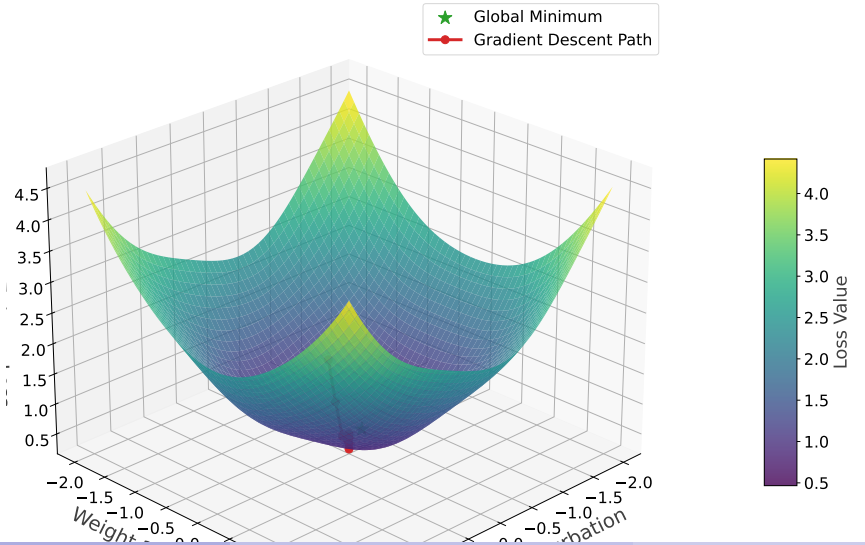
Input: "*Great, another boring movie*"

```
Logits: [2.1, -0.5]

exp(2.1) = 8.17
exp(-0.5) = 0.61
Sum = 8.78
```

Cross-Entropy Loss (Worked Example)

```
Ground Truth: y = [1, 0]  (Negative)
Predicted:    p = [0.93, 0.07]

Loss = -sum(y * log(p))
     = -(1 * log(0.93) + 0 * log(0.07))
     = -(-0.073 + 0)
     = 0.073
```

Loss Landscape: Fine-Tuning BERT for Sentiment



Appendix A2: Optimization & Hyperparameters

AdamW Optimizer:

- Adam with weight decay
- Adaptive learning rates per parameter
- Standard hyperparameters work well for most tasks
- Weight decay prevents overfitting

Learning Rate Schedule:

- **Warmup**: Linear increase from 0 to peak (first portion of training)
- **Decay**: Linear decrease to 0 over remaining steps
- **Peak LR**: Small learning rate (typical for fine-tuning)
- **Why**: Prevents catastrophic forgetting of pre-trained weights

Batch Sizes & Training:

- Batch size: Limited by GPU memory (typically order of magnitude: $\sim$10s)
- Epochs: Few epochs sufficient (typically single digits, more causes overfitting)
- Gradient accumulation if GPU memory limited

Practical Implementation:

- **Hugging Face Transformers**: Provides well-tested default hyperparameters
- `TrainingArguments` class handles learning rate scheduling automatically
- See `transformers.huggingface.co` for documented best practices

Careful hyperparameter tuning prevents destroying pre-trained knowledge while learning sentiment.

Appendix A2b: Attention Score Calculation

Example: "The movie was not very good"

Step 1: Create Query, Key, Value vectors

- Query ("not"): $\mathbf{q} = [0.8, 0.2, -0.1]$
- Key ("good"): $\mathbf{k} = [0.7, 0.3, 0.1]$

Step 2: Compute attention score

$$\text{score}(\text{not}, \text{good}) = \frac{\mathbf{q} \cdot \mathbf{k}}{\sqrt{d_k}} = \frac{0.8 \times 0.7 + 0.2 \times 0.3 + (-0.1) \times 0.1}{\sqrt{3}} = \frac{0.61}{1.73} = 0.35$$

Step 3: Apply softmax (over all keys)

$$\alpha_{\text{not} \to \text{good}} = \frac{e^{0.35}}{e^{0.35} + e^{0.1} + e^{0.2} + \ldots} = 0.18$$

Result: "not" attends to "good" with weight 0.18 — this connection helps BERT understand negation!

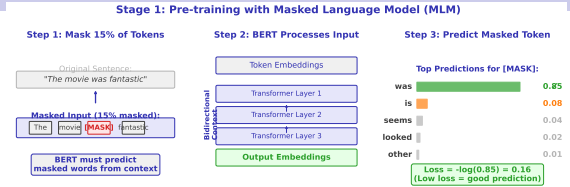
Attention scores reveal HOW BERT learns to connect "not" with the word it negates.

Appendix A2c: Stage 1 - Pre-training Deep Dive

Masked Language Model (MLM):

Objective: Predict masked tokens from context

$$\mathcal{L}_{\text{MLM}} = -\mathbb{E}_{x \sim \mathcal{D}} \left[\sum_{i \in \text{masked}} \log P(x_i \mid x_{\setminus i}) \right]$$



Masking Strategy:

- Mask 15% of tokens randomly
- 80% replaced with [MASK]
- 10% replaced with random token
- 10% kept unchanged

Hyperparameters:

- Training corpus: 3.3B words (Wikipedia + BooksCorpus)
- Training time: Several days on TPUs
- Batch size: Thousands of sequences
- Learning rate: Warm-up then decay

Code Example (Simplified):

```
# MLM training loop
for batch in dataloader:
    # Mask 15% of tokens
    masked_inputs = mask_tokens(batch)

    # Forward pass
    outputs = bert(masked_inputs)

    # Loss: predict masked tokens
    loss = cross_entropy(
        outputs[masked_positions],
        original_tokens[masked_positions]
    )

    # Backward pass
    loss.backward()
    optimizer.step()
```

Pre-training teaches BERT general language understanding before task-specific fine-tuning.

Appendix A2d: Stage 2 - Classifier Head Architecture

Architecture Details:

The classifier head is a simple linear layer:

$$z = W^T h_{[\text{CLS}]} + b$$

where:

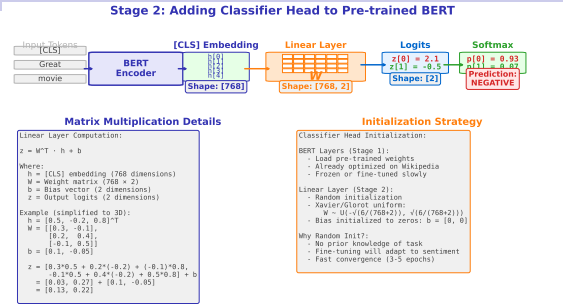
- $h_{[\text{CLS}]} \in \mathbb{R}^{768}$ — BERT's [CLS] output
- $W \in \mathbb{R}^{768 \times 2}$ — weight matrix
- $b \in \mathbb{R}^2$ — bias vector
- $z \in \mathbb{R}^2$ — logits (one per class)

Initialization:

- $W \sim \mathcal{U}(-\sqrt{6/(768+2)}, \sqrt{6/(768+2)})$ (Xavier)
- $b = \mathbf{0}$ (zeros)
- Pre-trained BERT weights loaded

Parameters:

- Classifier head: 1,538 parameters
- BERT-base total: 110M parameters
- Ratio: 0.001% new parameters



Code Example:

```
import torch.nn as nn

class BertForSentiment(nn.Module):
    def __init__(self, bert_model):
        super().__init__()
        self.bert = bert_model  # Pre-trained
        self.classifier = nn.Linear(768, 2)
        # Classifier randomly initialized

    def forward(self, input_ids, attention_mask):
        # BERT encoding
        outputs = self.bert(
            input_ids=input_ids,
            attention_mask=attention_mask
```

Appendix A2e: Stage 4 - Deployment & Inference Pipeline

Inference Pipeline:

Step 1: Tokenization (˜1ms)

- WordPiece tokenization
- Add [CLS] and [SEP] tokens
- Convert to input IDs
- Create attention mask

Step 2: Model Forward Pass (˜10-50ms)

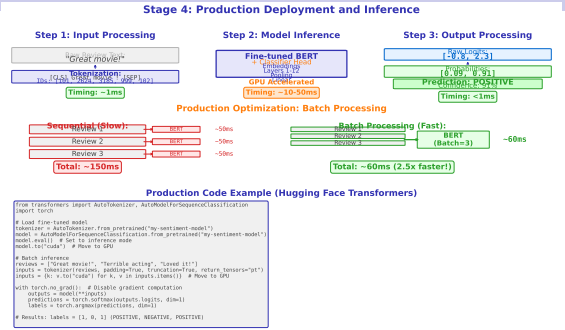
- BERT encoding (12 layers)
- Extract [CLS] representation
- Linear classifier layer
- GPU acceleration critical

Step 3: Post-processing (¡1ms)

- Apply softmax to logits
- Get class probabilities
- Return prediction + confidence

Optimization Strategies:

- **Batching:** Process multiple inputs together (2-3x



Production Throughput:

- Single GPU (V100): ˜200-500 samples/sec
- With batching (batch=32): ˜1000-2000 samples/sec
- Latency: 10-50ms per prediction (acceptable for most applications)

Appendix A3: Aspect-Based Sentiment Analysis

Problem: Extract sentiment per product aspect
"The phone camera is excellent but battery life is terrible."

- Camera: POSITIVE
- Battery: NEGATIVE

BERT Approach:

1. Identify aspects (camera, battery) via NER or keywords
2. For each aspect:
   - Create input: [CLS] aspect [SEP] sentence [SEP]
   - Fine-tune BERT to predict sentiment for that aspect
   - Use attention to find aspect-relevant words
3. Aggregate aspect sentiments

Applications:

- Product reviews (Amazon, Yelp)
- Survey analysis
- Brand monitoring

Aspect-based extends BERT from document-level to fine-grained sentiment extraction.

Appendix A4: Multi-Label Sentiment Analysis

Problem: Detect multiple emotions per text
"I'm excited about the trip but worried about the cost."

- Joy: HIGH
- Anxiety: MEDIUM
- Anger: LOW

BERT Modifications:

- **Architecture**: Replace softmax with sigmoid activation
- **Output**: K independent probabilities (one per emotion)
- **Loss**: Binary cross-entropy for each label

$$\mathcal{L} = -\frac{1}{K}\sum_{k=1}^{K}[y_k \log(\hat{y}_k) + (1-y_k)\log(1-\hat{y}_k)]$$

- **Threshold**: Predict label if $\hat{y}_k > 0.5$

Applications:

- Emotion detection (Plutchik's wheel)
- Mental health monitoring
- Customer service routing

Multi-label captures emotional complexity beyond simple positive/negative classification.

Appendix A5: Zero-Shot Sentiment with Prompting

Problem: Classify sentiment with NO labeled data

Approach: Use instruction-tuned LLMs (GPT-4, Claude, Llama)

Prompt Template

```
Classify the sentiment of the following review as POSITIVE or NEGATIVE.
Review: ‘‘Great, another boring superhero movie’’
Sentiment:
```

Why It Works:

- Pre-trained on massive text (understands sentiment)
- Instruction-tuned to follow prompts
- Can reason about sarcasm, negation, intensity
- No fine-tuning required

Tradeoffs vs BERT Fine-Tuning:

- **Pros**: No labeled data, handles rare cases, flexible
- **Cons**: Slower inference, higher per-query cost, less controllable

Zero-shot useful for prototyping or low-resource scenarios, but fine-tuned BERT better for production.

Appendix A6: Resources & Further Reading

Foundational Papers:

- Devlin et al. (2019). BERT: Pre-training of Deep Bidirectional Transformers. NAACL.
- Liu et al. (2019). RoBERTa: A Robustly Optimized BERT Pretraining Approach. arXiv.
- Sanh et al. (2019). DistilBERT: A distilled version of BERT (smaller, faster).

Practical Tools:

- Hugging Face Transformers: `transformers.huggingface.co`
- Datasets: IMDb, SST-2, Yelp, Twitter Sentiment
- Pre-trained models: BERT, RoBERTa, DistilBERT, ELECTRA

Tutorials:

- Hugging Face Course: `huggingface.co/course`
- Google Colab notebooks (free GPU access)
- Fast.ai NLP course

Advanced Topics:

- Adversarial robustness in sentiment analysis
- Cross-lingual sentiment (multilingual BERT)
- Temporal aspects (sentiment over time)

Start with Hugging Face tutorials for hands-on experience fine-tuning BERT for sentiment analysis.

Appendix A7: Full Mathematical Derivations

Cross-Entropy Gradient (For Backpropagation):

$$\frac{\partial \mathcal{L}}{\partial z_j} = \hat{y}_j - y_j$$

where z_j is the logit for class j. This simple gradient is why cross-entropy works so well!

Softmax Jacobian:

$$\frac{\partial \hat{y}_i}{\partial z_j} = \hat{y}_i(\delta_{ij} - \hat{y}_j)$$

where δ_{ij} is the Kronecker delta (1 if $i = j$, else 0).

Attention Weight Gradient:

$$\frac{\partial \alpha_{ij}}{\partial \mathbf{q}_i} = \alpha_{ij} \left(\mathbf{k}_j - \sum_k \alpha_{ik} \mathbf{k}_k \right) / \sqrt{d_k}$$

BERT Fine-Tuning Update:

$$\theta_{t+1} = \theta_t - \eta \cdot \text{AdamW}(\nabla_\theta \mathcal{L})$$

with small learning rate η and weight decay λ (standard fine-tuning values).

These gradients enable end-to-end training: error signal flows from loss through attention to word embeddings.