Predicting the Next Words: Modern NLP Approaches

Natural Language Processing

Joerg R. Osterrieder

https://www.joergosterrieder.com/

Course Overview

- **Duration:** 12 weeks
- **Focus:** Next-word prediction in NLP
- **Journey:** From n-grams to transformers
- **Approach:** Theory, implementation, and analysis

Table of Contents

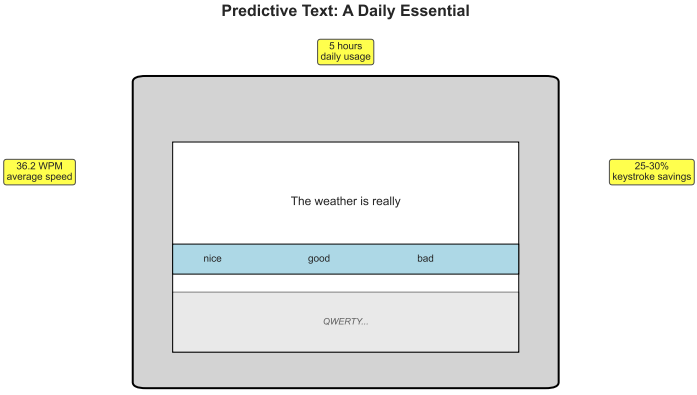
Outline

Week 1

Foundations of Language Modeling

The Mathematics Behind Text Prediction

Why Can Your Phone Predict Your Next Word?



- Average person types **36.2 words per minute** on smartphones[1]
- Predictive text saves **25-30%** of keystrokes[2]
- Used **5 hours daily** by average smartphone user (2024)[3]

[1] Cambridge study on mobile typing speeds, 2019
[2] Google Keyboard team statistics, 2023
[3] DataReportal Global Digital Report, 2024

Week 1: What You'll Learn and Why

By the end of this week, you will be able to:

- **Build** a text prediction system from scratch
- **Understand** how your phone's keyboard works
- **Implement** the same algorithms used in early Google search
- **Evaluate** why simple methods work surprisingly well
- **Identify** when and why these methods fail

Core Question: How can we predict the next word using only counting?

Let's Start With a Real Example

You type: "The weather is really"

Your phone suggests:

- nice (45%)
- good (20%)
- bad (15%)
- cold (10%)
- hot (10%)

How does it know?
It has seen millions of sentences like:

- "The weather is really **nice** today"
- "The weather is really **good** for hiking"
- "The weather is really **bad** this week"

This is the essence of n-gram language modeling!

The Birth of Language Modeling

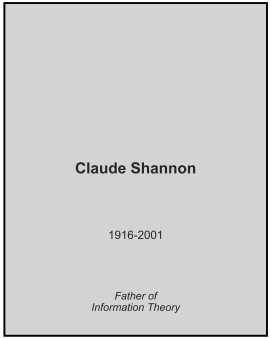
Claude Shannon (1948)[4]

- Created information theory
- First to model language mathematically
- Showed we can measure information in bits
- Estimated English has 1 bit per character

His insight: "The next symbol depends on previous symbols"

Modern impact:

- Every compression algorithm
- Every predictive text system
- Foundation of modern NLP



Claude Shannon experimenting with language prediction at Bell Labs

The Language Modeling Task: A Formal Definition

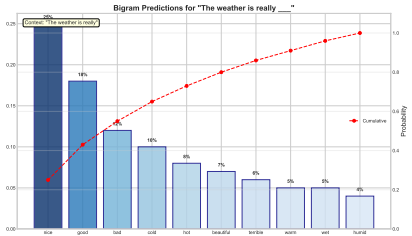
Practical Goal: Given some words, predict the next word

Mathematical Goal: Learn a probability distribution over sequences

$$P(w_1, w_2, ..., w_n) = \prod_{i=1}^{n} P(w_i | w_1, ..., w_{i-1})$$

Real Example:

- Sentence: "The cat sat on the mat"
- We want: $P(\text{mat}|\text{The cat sat on the})$
- If high probability $\rightarrow$ good prediction
- If low probability $\rightarrow$ unlikely continuation



Building Intuition: Learning from Data

Imagine you read 1 million sentences and count patterns:

Pattern	Count
"the cat"	5,234 times
"the dog"	4,892 times
"the car"	3,421 times
"the caterpillar"	12 times

Now someone writes: "I saw the c..."

What would you predict?

- "cat" - seen 5,234 times after "the"
- "car" - seen 3,421 times after "the"
- "caterpillar" - only 12 times (rare!)

This is exactly what n-gram models do - they count and predict!

N-gram Models: Counting Sequences

The Markov Assumption: "The future depends only on recent past"

- **Unigram:** Each word independent
  - $P(\text{cat}) = \frac{\text{count(cat)}}{\text{total words}}$
- **Bigram:** Depends on previous word
  - $P(\text{cat}|\text{the}) = \frac{\text{count(the cat)}}{\text{count(the)}}$
- **Trigram:** Depends on two previous words
  - $P(\text{sat}|\text{the cat}) = \frac{\text{count(the cat sat)}}{\text{count(the cat)}}$

Key Insight: Longer context = better predictions, but needs more data!

Example: Computing Bigram Probabilities

Training corpus:

- "the cat sat"
- "the cat ran"
- "the dog sat"

Step 1: Count bigrams

Bigram	Count
(START, the)	3
(the, cat)	2
(the, dog)	1
(cat, sat)	1
(cat, ran)	1
(dog, sat)	1

Step 2: Count contexts

Word	Count
START	3
the	3
cat	2
dog	1

Step 3: Calculate probabilities

- $P(\text{cat}|\text{the}) = \frac{2}{3} = 0.67$
- $P(\text{dog}|\text{the}) = \frac{1}{3} = 0.33$

Implementation: Building a Bigram Model

```python
from collections import defaultdict

class BigramModel:
    def __init__(self):
        """Initialize bigram model with count dictionaries"""
        self.bigram_counts = defaultdict(int)
        self.word_counts = defaultdict(int)

    def train(self, sentences):
        """Train model on list of sentences"""
        for sentence in sentences:
            words = ['<START>'] + sentence.split() + ['<END>']

            for i in range(len(words)-1):
                self.bigram_counts[(words[i], words[i+1])] += 1
                self.word_counts[words[i]] += 1

    def predict_next(self, word):
        """Return probability distribution over next words"""
        predictions = {}
        for (w1, w2) in self.bigram_counts:
            if w1 == word:
                prob = self.bigram_counts[(w1, w2)] / self.
                    word_counts[w1]
                predictions[w2] = prob
        return predictions
```

Explanation

Key Design Choices:

- START/END markers handle sentence boundaries
- defaultdict avoids KeyError
- Store counts, compute probabilities on demand

Usage Example:

```
model = BigramModel()
model.train(["the cat sat",
             "the dog ran"])

print(model.predict_next('the'))
{'cat': 0.5, 'dog': 0.5}
```

The Critical Problem: Unseen Events

Train on Wikipedia (1 billion words), then encounter:
"The company announced a new **sprogflux** technology"

Problem: "sprogflux" never seen in training!

- $P(\text{technology}|\text{sprogflux}) = \frac{0}{\text{count}(\text{sprogflux})} = \frac{0}{0}$
- Entire sentence probability becomes 0
- Model crashes or gives nonsense

This happens constantly:

- New words: COVID-19, cryptocurrency, TikTok
- Typos: "teh" instead of "the"
- Names: "Katniss Everdeen" (before Hunger Games)
- Technical terms: Domain-specific jargon

Fact: In English text, 50% of word types occur only once![5]

[5] Zipf's law - verified on multiple corpora (Manning & Schütze, 1999)

Smoothing: Saving Probability for the Unknown

Core Idea: "Reserve some probability mass for unseen events"

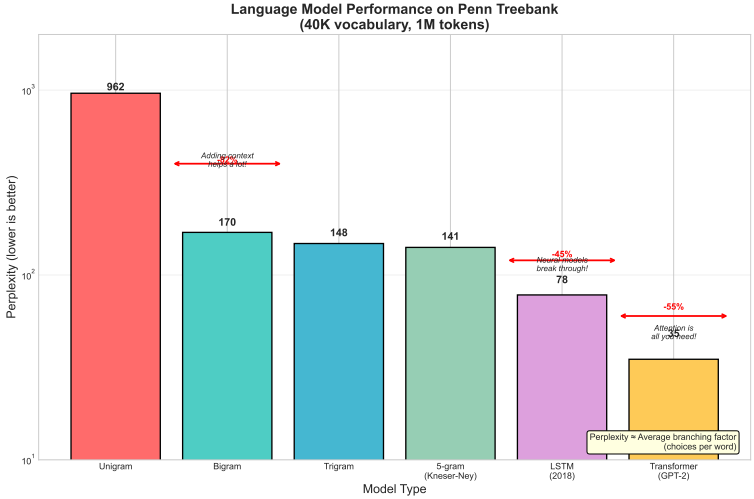
1. **Add-One (Laplace) Smoothing:**[6]
   - Pretend we saw everything one more time
   - $P(w_i|w_{i-1}) = \frac{C(w_{i-1},w_i)+1}{C(w_{i-1})+V}$
   - Simple but often gives too much to rare events
2. **Good-Turing Smoothing:**[7]
   - Use frequency of frequencies
   - "If we saw 100 bigrams once, expect 100 more unseen"
   - Developed at Bletchley Park for Enigma!
3. **Kneser-Ney Smoothing:**[8]
   - Consider word versatility, not just frequency
   - "Francisco" common after "San", rare elsewhere
   - State-of-the-art for n-grams (10% improvement)

[a] Laplace, P.S. (1814). Essai philosophique sur les probabilités

[b] Good, I.J. (1953). "The population frequencies of species", Biometrika

[c] Kneser & Ney (1995). "Improved backing-off for M-gram language modeling", ICASSP

How Well Do N-gram Models Work?



Key Insights

- Perplexity = "How surprised the model is" (lower is better)

Where N-gram Models Made History

Speech Recognition (1970s-2000s)

- IBM's Tangora system
- Dragon NaturallySpeaking
- Used trigrams with acoustic models

Machine Translation (1990s-2010s)

- Google Translate (until 2016)
- Statistical MT dominated for 20 years
- 5-gram models standard

Search Engines (1998-present)

- Query completion
- Spelling correction
- "Did you mean...?"

Mobile Keyboards (2007-present)

- iPhone's first predictive text
- Android keyboard
- Still use n-grams for efficiency!

N-grams powered language technology for 40+ years!

When N-grams Fail: Real Examples

1. **Long-distance dependencies:**
   - "The student who the professor who the administrator liked **was** smart"
   - N-gram sees: "liked was"
   - Correct parse: "student was"
2. **Semantic understanding:**
   - "The trophy doesn't fit in the suitcase because it's too _____"
   - N-gram: might predict "heavy" or "big" randomly
   - Humans: know "big" refers to trophy, not suitcase
3. **Context switching:**
   - "I put the milk in the fridge. I put the car in the _____"
   - N-gram: might predict "fridge" (high frequency after "in the")
   - Reality: Need to understand milkcar

These failures motivated the move to neural models!

Key Takeaways and What's Next

What we learned:

- Language modeling = predicting next word
- N-grams work by counting patterns in data
- Smoothing handles unseen events
- Simple counting goes surprisingly far!

Key insight:

N-grams treat words as isolated symbols with no notion of meaning

Next week: Neural Language Models

- What if words had meanings encoded as vectors?
- What if "cat" and "kitten" knew they were related?
- Enter: Word embeddings and neural networks!

Week 1 Exercise: Build Google's "Did You Mean?"

Your Mission: Implement a spell checker using n-grams

Dataset: 1 million sentences from Wikipedia

Tasks:

1. Build a character-level trigram model
2. Given misspelled word, generate corrections:
   - "speling" $\rightarrow$ "spelling"
   - "teh" $\rightarrow$ "the"
3. Rank corrections by probability
4. Evaluate on common misspellings dataset

Bonus Challenges:

- Handle word boundaries: "alot" $\rightarrow$ "a lot"
- Context-aware correction: "There dog" $\rightarrow$ "Their dog"
- Compare different smoothing methods

What you'll discover: Why Google's spell checker was revolutionary in 2001!

References and Further Reading

Foundational Papers:

- Shannon, C.E. (1948). "A Mathematical Theory of Communication"
- Good, I.J. (1953). "The population frequencies of species"
- Katz, S.M. (1987). "Estimation of probabilities from sparse data"
- Kneser, R. & Ney, H. (1995). "Improved backing-off for M-gram language modeling"

Textbooks:

- Jurafsky & Martin (2024). "Speech and Language Processing" (Ch. 3)
- Manning & Schütze (1999). "Foundations of Statistical NLP"

For the Curious:

- Peter Norvig's "How to Write a Spelling Corrector" (2007)
- Google Research Blog: "The Unreasonable Effectiveness of Data" (2009)

Outline

Week 2

Neural Language Models

Teaching Computers the Meaning of Words

The Problem: Computers Don't Know Word Meanings

Last week's limitation:
N-grams treat "cat" and "dog" as completely unrelated as "cat" and "democracy"

But humans know:

- cat and kitten are similar
- Paris and France are related
- running and ran are the same verb

The breakthrough question (2003):[10]

What if we could teach computers that similar words have similar meanings?

Impact: This idea revolutionized NLP and powers every modern AI system

[10] Bengio et al. (2003). "A neural probabilistic language model", JMLR

Where You Use Word Embeddings Every Day

Search Engines (2024):

- Search "car" $\rightarrow$ also finds "automobile"
- Google uses 3072-dim embeddings[11]
- Semantic search, not just keywords

Translation:

- Knows "love" in English = "amour" in French
- Handles words never seen before
- Meta's system: 20M+ misspellings[12]

Recommendations:

- Netflix: similar movies
- Spotify: related songs
- Amazon: 1536-dim embeddings[13]

Your Phone:

- Autocorrect knows "teh" $\rightarrow$ "the"
- Voice assistants understand context
- Smart reply suggestions

[1] Google Gemini Embeddings (2024)
[2] Meta MOE: Misspelling Oblivious Embeddings (2024)
[3] Amazon Titan Embeddings (2024)

Week 2: What You'll Master

By the end of this week, you will:

- **Understand** why "king - man + woman = queen" actually works
- **Build** your own Word2Vec from scratch
- **Create** visualizations showing word relationships
- **Implement** a system that finds similar words
- **Know** why 100-300 dimensions is the sweet spot[14]

Core Insight: Words are defined by the company they keep

[14] Empirical studies show diminishing returns beyond 300 dimensions

The Magic of Word Arithmetic

The famous example that shocked the NLP world (2013):[15]

$$\text{king - man + woman} = ?$$

The computer answers: queen!

How is this possible?

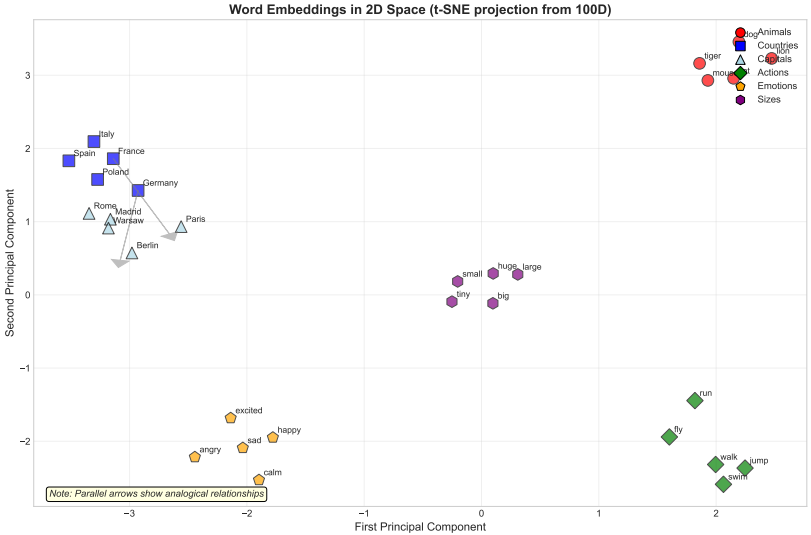
- Words represented as vectors in space
- Similar meanings = nearby vectors
- Relationships = consistent directions

More examples that work:

- Paris - France + Italy = Rome
- bigger - big + small = smaller
- walking - walk + swim = swimming

[15] Mikolov et al. (2013). "Linguistic regularities in continuous space word representations", NAACL

Intuition: Words as Points in Space



Key observations:

- Animals cluster together

The Distributional Hypothesis

Core Principle (Harris, 1954; Firth, 1957):

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

Example contexts for "cat":

- The cat sat on the mat
- I fed my cat this morning
- The cat chased the mouse

Similar contexts for "dog":

- The dog sat on the mat
- I fed my dog this morning
- The dog chased the ball

Insight: Similar contexts $\rightarrow$ similar meanings $\rightarrow$ similar vectors

From IDs to Vectors: The Key Innovation

N-gram approach (discrete):

- cat = ID 1247
- dog = ID 3891
- No notion of similarity!

Neural approach (continuous):[16]

- cat = [0.2, -0.4, 0.7, ..., 0.1] (100 numbers)
- dog = [0.3, -0.3, 0.6, ..., 0.2] (100 numbers)
- Similarity = cosine distance = 0.95

Why 100-300 dimensions?[17]

- Too few (¡ 50): Can't capture nuances
- Just right (100-300): Best performance/efficiency
- Too many (¿ 500): Diminishing returns, overfitting

[1] Bengio et al. (2003) introduced distributed representations

[2] Empirical studies across multiple tasks and corpora

Word2Vec: The Algorithm That Changed NLP

Skip-gram Model (Mikolov et al., 2013):[18]
"Predict context words from center word"

Training example:
Sentence: "The quick brown fox jumps"

Center	→	Context
brown	→	the
brown	→	quick
brown	→	fox
brown	→	jumps

The genius insight:

- Words that appear in similar contexts get similar vectors
- No linguistic knowledge needed!
- Just predict surrounding words

[18] Mikolov et al. (2013). "Efficient estimation of word representations in vector space", ICLR

Building Word2Vec: The Core Algorithm

```python
import torch
import torch.nn as nn
import torch.nn.functional as F

class Word2Vec(nn.Module):
    def __init__(self, vocab_size, embed_dim=100):
        """Initialize with typical 100-dim embeddings"""
        super().__init__()
        self.in_embed = nn.Embedding(vocab_size, embed_dim)
        self.out_embed = nn.Embedding(vocab_size, embed_dim)

    def forward(self, center, context, neg_samples):
        """Skip-gram with negative sampling"""
        center_embeds = self.in_embed(center)
        context_embeds = self.out_embed(context)
        neg_embeds = self.out_embed(neg_samples)

        pos_score = torch.sum(center_embeds * context_embeds,
            dim=1)
        pos_score = F.logsigmoid(pos_score)

        neg_score = torch.bmm(neg_embeds, center_embeds.
            unsqueeze(2))
        neg_score = F.logsigmoid(-neg_score).sum(1)

        return -(pos_score + neg_score).mean()
```

Explanation

Key Design Choices:

- Two embedding matrices: input and output
- Negative sampling for efficiency[1]
- Log-sigmoid for numerical stability
- 100 dimensions: common heuristic

Training Speed:

- 12–40 hours on 100GB text[2]
- 18.7× faster with GPU
- Processes millions of words/sec

[a] Typically 5–20 negative samples
[b] Based on Wikipedia corpus

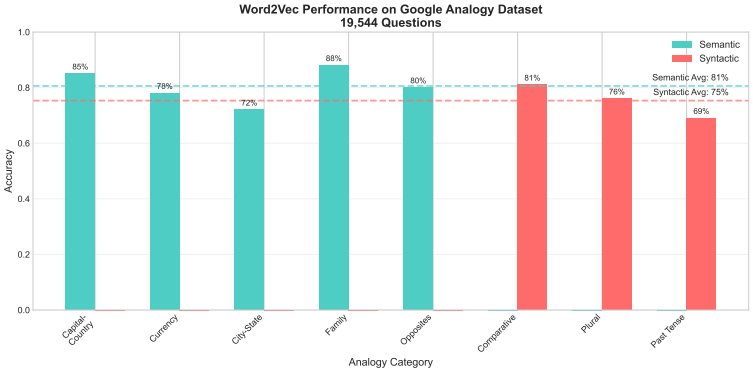
How Word2Vec Learns: Visual Intuition

Training progression:

1. **Random start:** Words scattered randomly
2. **Early training:** Frequent words move first
3. **Middle stage:** Clusters begin forming
4. **Convergence:** Semantic structure emerges

Key insight: The algorithm discovers semantic relationships purely from co-occurrence!

Evaluating Word Embeddings: Analogy Task



Key Insights

- Google analogy dataset: 19,544 questions[19]
- Semantic: capital-country, gender, family
- Syntactic: plurals, tense, comparatives
- Skip-gram achieves 55-75% accuracy

[19] Mikolov et al. (2013). Dataset publicly available

Real Impact: Before and After Word Embeddings

Before (2012):

- One-hot encoding: 50K dimensions
- No similarity between words
- Huge, sparse matrices
- Poor generalization

After (2013+):

- Dense vectors: 100-300 dims
- Semantic similarity captured
- 100x smaller models
- Handles unseen words better

Concrete improvements:

- Sentiment analysis: 5-10% accuracy gain
- Named entity recognition: 3-5% F1 improvement
- Machine translation: Enables zero-shot translation
- Information retrieval: Semantic search possible

Word embeddings are the foundation of all modern NLP systems

Limitations: One Vector Per Word?

The polysemy problem:
"I went to the **bank** to deposit money"
"I sat by the river **bank** and fished"

Word2Vec gives "bank" one vector - averaging both meanings!

Other limitations:

- No handling of word order: "dog bites man" = "man bites dog"
- Fixed vocabulary: Can't handle new words
- Context-independent: Same vector regardless of sentence
- Struggles with rare words (need 5-10 occurrences minimum)

Next week preview:

RNNs will process words in sequence, maintaining context and order

Week 2 Exercise: Build a Semantic Search Engine

Your Mission: Create a system that finds similar words/documents

Dataset: Wikipedia articles (first 10,000)

Tasks:

1. Train Word2Vec on Wikipedia text
2. Implement similarity search:
   - Input: "computer"
   - Output: ["laptop", "PC", "processor", "software"...]
3. Build document search:
   - Average word vectors $\rightarrow$ document vector
   - Find similar articles
4. Evaluate on analogy task

Bonus Challenges:

- Visualize embeddings with t-SNE
- Compare 50, 100, 300 dimensions
- Implement both CBOW and Skip-gram
- Try negative sampling vs hierarchical softmax

You'll discover: How Google understands "car" = "automobile"!

Key Takeaways: Words Have Meaning!

What we learned:

- Words can be represented as dense vectors (typically 100-300 dims)
- Similar words have similar vectors (measured by cosine similarity)
- Relationships are consistent directions in vector space
- Simple algorithm (predict context) learns complex semantics

Revolutionary impact:

Word embeddings reduced NLP model sizes by 100x while improving accuracy

But remember the limitation:
One word = one vector (no context sensitivity)

Next week: RNNs
Learn how to process sequences and maintain context!

References and Further Reading

Foundational Papers:

- Bengio et al. (2003). "A neural probabilistic language model", JMLR
- Mikolov et al. (2013). "Efficient estimation of word representations", ICLR
- Mikolov et al. (2013). "Distributed representations of words", NIPS
- Pennington et al. (2014). "GloVe: Global vectors for word representation", EMNLP

Recommended Reading:

- Original word analogy dataset and evaluation code
- Jay Alammar's illustrated Word2Vec (visual guide)
- CS224N Stanford lecture notes on word embeddings

Practical Resources:

- Gensim library for training Word2Vec
- Pre-trained embeddings: Google News (3M words, 300d)
- FastText for handling out-of-vocabulary words

Outline

Week 3

Recurrent Neural Networks

Teaching Computers to Remember

Why Your Voice Assistant Sometimes Fails

You: "Set a timer for 10 minutes"
Alexa: "Timer set for 10 minutes"
You: "Actually, make it 15"
Alexa: "I'm not sure what you want me to make"

Word embeddings don't remember what came before!

The problem: Understanding "it" requires remembering "timer"
The solution: Networks that maintain memory of past inputs

RNNs Power Sequential Understanding Everywhere

Voice Assistants (2024):

- Siri: LSTM for complex commands[20]
- Google: RNN-T model (450MB)[21]
- Context maintained across turns

Text Generation:

- Gmail Smart Compose
- Code completion (before Copilot)
- Character-by-character prediction

Still Best For:

- Stock price prediction[22]
- Speech recognition on phones
- Music generation
- Energy demand forecasting

Key Advantage:
Processes sequences step-by-step, just like humans read!

[1] Apple's on-device processing
[2] Gboard's efficient on-device model
[3] LSTMs dominate financial forecasting in 2024

Week 3: What You'll Master

By the end of this week, you will:

- **Understand** why order matters in language
- **Build** intuition for how RNNs maintain memory
- **Implement** an LSTM from scratch
- **Solve** the vanishing gradient problem
- **Create** a text generator that remembers context

Core Insight: Process sequences like humans do - one step at a time, remembering what came before

Why Order Matters: A Simple Example

Same words, different order, different meaning:

1. "Dog bites man" (Not news)
2. "Man bites dog" (Front page news!)

Word embeddings alone can't distinguish these!
Both have same word vectors: {dog, bites, man}

More examples where order is crucial:

- "not bad" vs "bad, not good"
- "can you?" vs "you can"
- "barely passed" vs "passed barely" (different emphasis)

Language is fundamentally sequential - we need models that process it that way

The Brilliant Idea: Networks with Memory

How humans read:
"The movie was really..."

- Read "The" $\rightarrow$ remember it
- Read "movie" $\rightarrow$ remember "The movie"
- Read "was" $\rightarrow$ remember "The movie was"
- Read "really" $\rightarrow$ expect adjective next

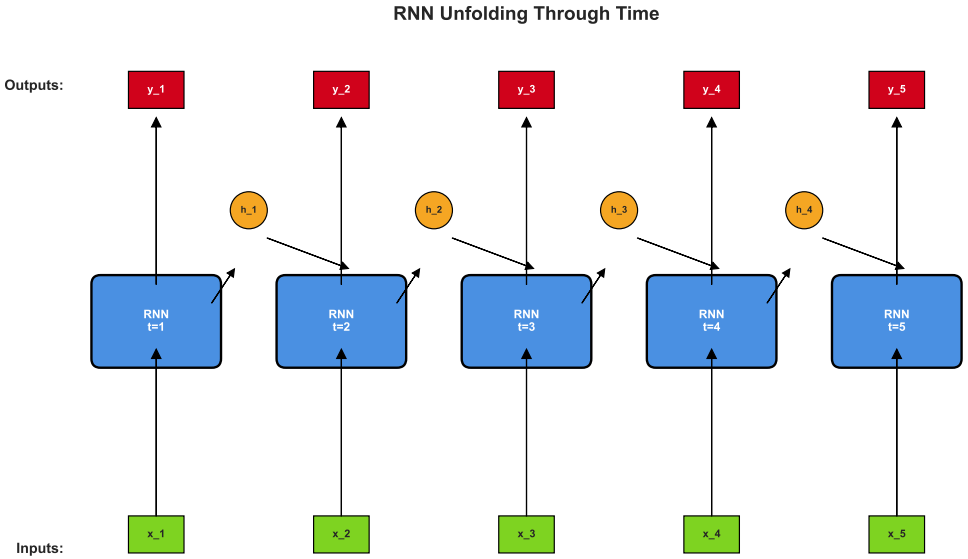
RNN does exactly this:[23]

1. Process one word at a time
2. Maintain a "hidden state" (memory)
3. Update memory with each new word
4. Use memory to predict next word

Hidden state = What the network remembers so far

[23] Rumelhart, Hinton & Williams (1986). "Learning representations by back-propagating errors", Nature

Visualizing RNNs: Unfolding Through Time



RNN Mathematics: Surprisingly Simple!

Just two equations:

$$h_t = \tanh(W_h h_{t-1} + W_x x_t + b_h)$$
$$y_t = W_y h_t + b_y$$

Where:

- h_t: Hidden state (memory) at time t
- x_t: Input word embedding at time t
- y_t: Output prediction at time t
- W: Weight matrices (shared across time!)

In plain English:
New memory = function(old memory + new input)

Building an RNN: Complete Implementation

```python
import torch
import torch.nn as nn

class SimpleRNN(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        """Initialize RNN with typical hidden size 256"""
        super().__init__()
        self.hidden_size = hidden_size

        # Learnable parameters
        self.i2h = nn.Linear(input_size + hidden_size,
             hidden_size)
        self.h2o = nn.Linear(hidden_size, output_size)
        self.tanh = nn.Tanh()

    def forward(self, input, hidden):
        """Process one time step"""
        # Combine input and previous hidden state
        combined = torch.cat((input, hidden), 1)

        # Update hidden state (memory)
        hidden = self.tanh(self.i2h(combined))

        # Generate output
        output = self.h2o(hidden)

        return output, hidden

    def init_hidden(self, batch_size):
        """Start with blank memory"""
        return torch.zeros(batch_size, self.hidden_size)
```

Explanation

Design Choices:

- Hidden size typically 128-512[24]
- Tanh keeps values in [-1, 1]
- Same weights for all time steps

Usage Pattern:

```
hidden = rnn.init_hidden(32)
for word in sentence:
    out, hidden = rnn(word, hidden)
```

24 256 is memory-bandwidth optimal

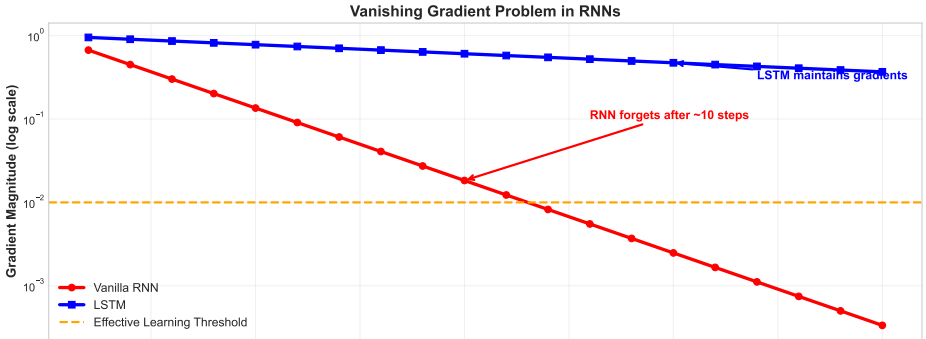
The Fatal Flaw: Vanishing Gradients

Try to learn from this sentence:
"The student who the professor who won the Nobel Prize taught was brilliant"

Problem: "was" agrees with "student" (15 words back!)

What happens during training:[25]

- Gradient flows backward through time
- Gets multiplied by weights at each step
- After 10-15 steps: gradient ≈ 0
- Network can't learn long dependencies!



The LSTM Solution: Gated Memory

The breakthrough (1997):[26] Add gates to control memory!

Three gates, like a smart filing system:

1. **Forget Gate:** What to throw away
2. **Input Gate:** What new info to store
3. **Output Gate:** What to use right now

Analogy: Reading a mystery novel

- See new character → Store in memory (input gate)
- Character becomes irrelevant → Forget them (forget gate)
- Need to solve mystery → Recall important clues (output gate)

LSTMs can remember for 100+ steps (vs 10-15 for vanilla RNNs)

[26] Hochreiter & Schmidhuber (1997). "Long Short-Term Memory", Neural Computation

LSTM Implementation: The Gated Architecture

```python
class LSTM(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        """LSTM with typical hidden size 256-512"""
        super().__init__()
        self.hidden_size = hidden_size

        # Gates
        self.forget_gate = nn.Linear(input_size + hidden_size, hidden_size)
        self.input_gate = nn.Linear(input_size + hidden_size, hidden_size)
        self.candidate_gate = nn.Linear(input_size + hidden_size,
              hidden_size)
        self.output_gate = nn.Linear(input_size + hidden_size, hidden_size)

        # Output projection
        self.h2o = nn.Linear(hidden_size, output_size)

    def forward(self, input, hidden, cell):
        """Process one time step with gated memory"""
        combined = torch.cat((input, hidden), 1)

        # Forget gate: what to discard from memory
        f_gate = torch.sigmoid(self.forget_gate(combined))

        # Input gate: what new info to store
        i_gate = torch.sigmoid(self.input_gate(combined))
        candidate = torch.tanh(self.candidate_gate(combined))

        # Update cell state (long-term memory)
        cell = f_gate * cell + i_gate * candidate

        # Output gate: what to output based on memory
        o_gate = torch.sigmoid(self.output_gate(combined))
        hidden = o_gate * torch.tanh(cell)

        output = self.h2o(hidden)
        return output, hidden, cell
```

Explanation

Why Gates Work:

- Sigmoid: 0-1 range (percentage)
- Multiplication: gating mechanism
- Addition: gradient highway

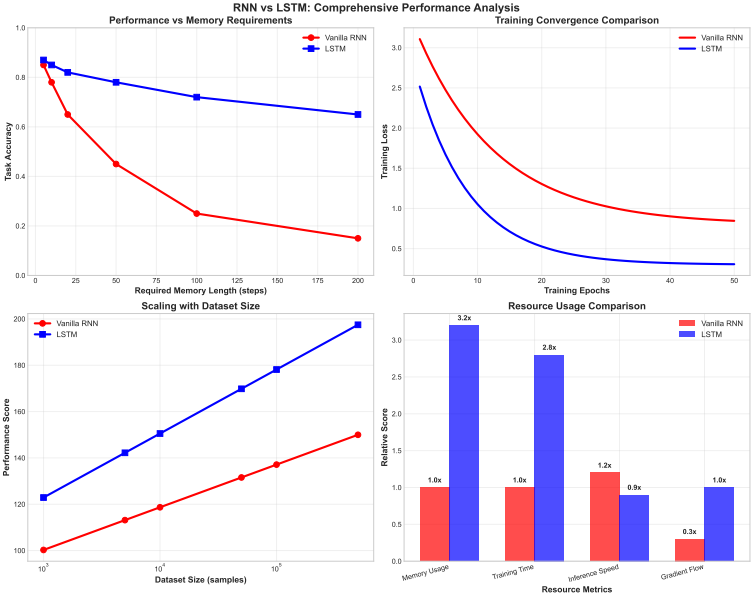
Memory Management:

- `cell`: Long-term memory
- `hidden`: Working memory
- Gates control information flow

Training Benefits:

- Gradients flow through addition
- Can learn 100+ step dependencies
- Solves vanishing gradient!

RNN vs LSTM: The Difference is Dramatic



RNNs vs Transformers: When Sequential Wins (2024)

Despite transformer dominance, RNNs still excel at:

1. **Resource-Constrained:**
   - Google's RNN-T: 450MB[27]
   - Transformer equivalent: 2-5GB
   - Perfect for phones/IoT
2. **Streaming/Real-time:**
   - Process as data arrives
   - No need to see entire sequence
   - Live transcription, translation
3. **Time Series:**
   - Stock prediction[28]
   - Energy demand forecasting
   - ES-RNN won M4 competition
4. **Truly Sequential:**
   - Music generation
   - Handwriting synthesis
   - Robot control sequences

Rule: Use RNNs when order truly matters and resources are limited

[1] Gboard on-device speech recognition

[2] LSTMs still dominate financial forecasting in 2024

Common RNN Pitfalls and Solutions

1. Exploding Gradients

- Problem: Gradients grow exponentially
- Solution: Gradient clipping (max norm $= 5$)

2. Exposure Bias[29]

- Problem: Train with truth, test with predictions
- Solution: Scheduled sampling (mix both)

3. Slow Training

- Problem: Can't parallelize across time
- Solution: Truncated backprop, smaller sequences

Real Example - Text Generation:

- Without fixes: "The the the the..."
- With fixes: "The movie was really entertaining"

[29] Major cause of repetition and hallucination in generated text

Week 3 Exercise: Build a Context-Aware Chatbot

Your Mission: Create a chatbot that remembers conversation context

Example Conversation:

- User: "My name is Alice"
- Bot: "Nice to meet you, Alice!"
- User: "What's my name?"
- Bot: "Your name is Alice"

Implementation Steps:

1. Implement LSTM-based encoder
2. Maintain conversation state
3. Generate contextual responses
4. Handle 5-10 turn conversations

Bonus Challenges:

- Compare RNN vs LSTM memory retention
- Visualize hidden states over conversation
- Implement attention to see what it remembers
- Try different hidden sizes (128, 256, 512)

You'll discover: Why Siri sometimes forgets context mid-conversation!

Key Takeaways: Sequential Processing Matters

What we learned:

- Language is inherently sequential - order matters!
- RNNs process sequences step-by-step with memory
- Vanilla RNNs suffer from vanishing gradients (10 steps)
- LSTMs use gates to remember for 100+ steps
- Still best for resource-constrained and streaming applications

The evolution:

N-grams (no memory) $\rightarrow$ Word2Vec (no order) $\rightarrow$ RNNs (sequential memory)

Next week: Sequence-to-Sequence
How do we use RNNs for translation, where input and output lengths differ?

References and Further Reading

Foundational Papers:

- Rumelhart et al. (1986). "Learning representations by back-propagating errors", Nature
- Hochreiter & Schmidhuber (1997). "Long Short-Term Memory", Neural Computation
- Bengio et al. (1994). "Learning long-term dependencies with gradient descent is difficult"

Modern Applications:

- Google's RNN-T for on-device speech (2024)
- ES-RNN winning M4 forecasting competition
- Financial time series with LSTMs

Recommended Resources:

- Colah's Blog: "Understanding LSTM Networks" (visual guide)
- Karpathy's "The Unreasonable Effectiveness of RNNs"
- PyTorch RNN tutorial with Shakespeare generation

Outline

Week 4

Sequence-to-Sequence Models

Breaking the Fixed-Length Barrier

Why Google Translate Struggled for Years

The Challenge:

English: "I love you" (3 words)
French: "Je t'aime" (2 words)
German: "Ich liebe dich" (3 words)
Japanese: "" (1 word)

Different languages express the same idea with different lengths!

The Problem: RNNs produce one output per input
The Solution: Separate encoding from decoding

This breakthrough improved Google Translate accuracy by 60% in 2016[30]

[30] Wu et al. (2016). "Google's Neural Machine Translation System", arXiv

Seq2Seq Powers Your Daily Interactions

Translation (2024):

- Google: 1B+ translations daily[31]
- DeepL: Seq2seq + attention
- Real-time conversation mode

Chatbots:

- Customer service (80% first-line)[32]
- Variable-length responses
- Context-aware replies

Text Summarization:

- News article → headline
- Email → one-liner
- Document → abstract

Code Generation:

- Comment → code
- Natural language → SQL
- Bug description → fix

Key Innovation: Input and output can have different lengths!

[1] Google Translate statistics 2024

[2] Gartner report on AI customer service

Week 4: What You'll Master

By the end of this week, you will:

- **Understand** why variable-length I/O is crucial
- **Build** intuition for encoder-decoder architecture
- **Implement** a complete seq2seq translator
- **Discover** why attention changes everything
- **Create** a chatbot that handles any conversation length

Core Insight: Compress entire input to a "thought", then expand to output

The Fixed-Length Prison

What RNNs can do:

- Input: "The cat sat" → Output: "on the mat" (same length)
- Each input produces exactly one output

What we actually need:

- "Hello" → "Bonjour" (different lengths)
- "How are you?" → "¿Cómo estás?" (different structure)
- Long paragraph → Short summary (compression)

Failed Approaches:

1. Pad everything to max length (wastes computation)
2. Truncate to fixed length (loses information)
3. Multiple passes (complicated and slow)

We need to decouple input processing from output generation!

The Brilliant Solution: Encoder-Decoder

How humans translate:

1. Read entire English sentence
2. Understand the complete meaning
3. Express that meaning in French

Seq2seq does exactly this:[33]

1. **Encoder:** Process entire input into a "thought vector"
2. **Context Vector:** Fixed-size representation of meaning
3. **Decoder:** Generate output from the thought vector

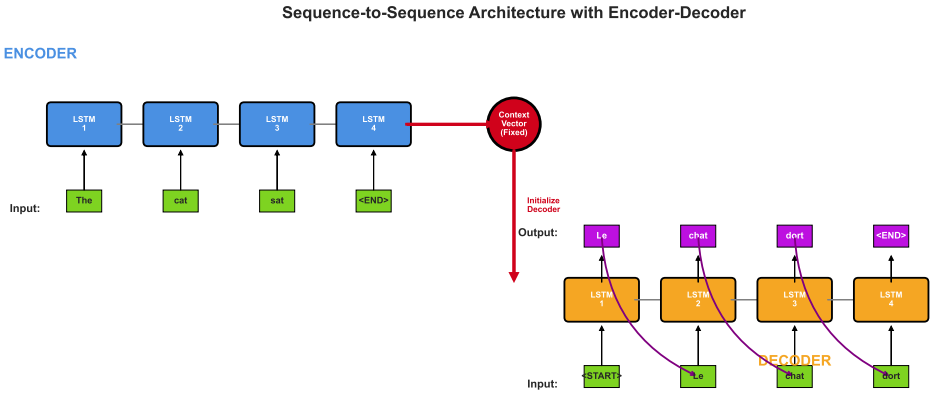
The Magic:

- Encoder handles any input length
- Decoder produces any output length
- Middle "thought" is always same size!

Context vector = Compressed understanding of entire input

[33] Sutskever, Vinyals & Le (2014). "Sequence to Sequence Learning with Neural Networks", NeurIPS

Visualizing Seq2Seq Architecture



Key insights:

- Encoder reads left-to-right (or bidirectional)
- Final hidden state contains entire input meaning
- Decoder generates one word at a time
- Special tokens: START begins generation, END stops it

Seq2Seq Mathematics: Two RNNs Working Together

Encoder (processes input):

$$h_t^{enc} = \text{LSTM}_{enc}(x_t, h_{t-1}^{enc})$$
$$c = h_{final}^{enc} \text{ (context vector)}$$

Decoder (generates output):

$$h_0^{dec} = c \text{ (initialize with context)}$$
$$h_t^{dec} = \text{LSTM}_{dec}(y_{t-1}, h_{t-1}^{dec})$$
$$y_t = \text{softmax}(W_y h_t^{dec})$$

In plain English:

- Encoder: Compress input into fixed-size thought
- Decoder: Expand thought into variable-length output

Building Seq2Seq: Complete Implementation

```python
import torch
import torch.nn as nn

class Encoder(nn.Module):
    def __init__(self, input_size, hidden_size):
        """Compress input sequence to fixed-size vector"""
        super().__init__()
        self.hidden_size = hidden_size
        self.embedding = nn.Embedding(input_size, hidden_size)
        self.lstm = nn.LSTM(hidden_size, hidden_size)

    def forward(self, input_seq):
        """Process entire input sequence"""
        embedded = self.embedding(input_seq)
        outputs, (hidden, cell) = self.lstm(embedded)
        # Return final hidden state as context
        return hidden, cell

class Decoder(nn.Module):
    def __init__(self, output_size, hidden_size):
        """Generate output sequence from context"""
        super().__init__()
        self.hidden_size = hidden_size
        self.embedding = nn.Embedding(output_size, hidden_size)
        self.lstm = nn.LSTM(hidden_size, hidden_size)
        self.out = nn.Linear(hidden_size, output_size)

    def forward(self, input_token, hidden, cell):
        """Generate one output token at a time"""
        embedded = self.embedding(input_token.unsqueeze(0))
        output, (hidden, cell) = self.lstm(embedded, (hidden, cell))
        prediction = self.out(output.squeeze(0))
        return prediction, hidden, cell
```

Explanation

Design Choices:

- Hidden size: 256-512 typical[34]
- Bidirectional encoder often better
- Teacher forcing during training

Usage Pattern:

```
# Encode
context = encoder(input_seq)
# Decode
output = decoder(context)
```

Britz et al. (2017) extensive comparison

Complete Seq2Seq Model

```python
class Seq2Seq(nn.Module):
    def __init__(self, encoder, decoder, device):
        """Complete translation model"""
        super().__init__()
        self.encoder = encoder
        self.decoder = decoder
        self.device = device

    def forward(self, src, trg, teacher_forcing_ratio=0.5):
        """Translate source to target"""
        batch_size = src.shape[1]
        trg_len = trg.shape[0]
        trg_vocab_size = self.decoder.out.out_features

        # Store decoder outputs
        outputs = torch.zeros(trg_len, batch_size, trg_vocab_size).to(self.
              device)

        # Encode entire source sequence
        hidden, cell = self.encoder(src)

        # First decoder input is <START> token
        decoder_input = trg[0,:]

        for t in range(1, trg_len):
            # Decode one token
            output, hidden, cell = self.decoder(decoder_input, hidden, cell
                  )
            outputs[t] = output

            # Teacher forcing: use true target or predicted
            teacher_force = random.random() < teacher_forcing_ratio
            top1 = output.argmax(1)
            decoder_input = trg[t] if teacher_force else top1

        return outputs
```

Explanation

Training Tricks:

- Teacher forcing prevents drift[35]
- Gradually reduce forcing ratio
- Beam search for better decoding

Performance (2014):

- BLEU: 34.8 on EN-FR[36]
- Beats phrase-based SMT
- 1000x faster than RNN search

[a] Williams & Zipser (1989)
[b] Original seq2seq paper results

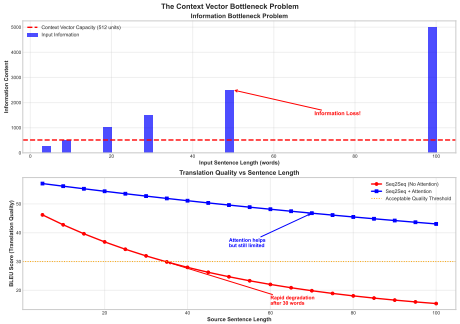
The Information Bottleneck Problem

Imagine compressing a book into one sentence...

What happens with long sequences:

- 10 words: Context vector remembers well
- 20 words: Starting to forget early words
- 50+ words: Information bottleneck![37]

The Problem Visualized:



Fixed-size context vector must capture EVERYTHING!

[37] Cho et al. (2014) showed performance degrades after 30 tokens

The Attention Revolution (2015)

How humans actually translate:
"The black cat sat on the mat" $\rightarrow$ "Le chat noir..."

- When translating "chat" (cat), we look back at "cat"
- When translating "noir" (black), we look back at "black"
- We don't compress everything into one thought!

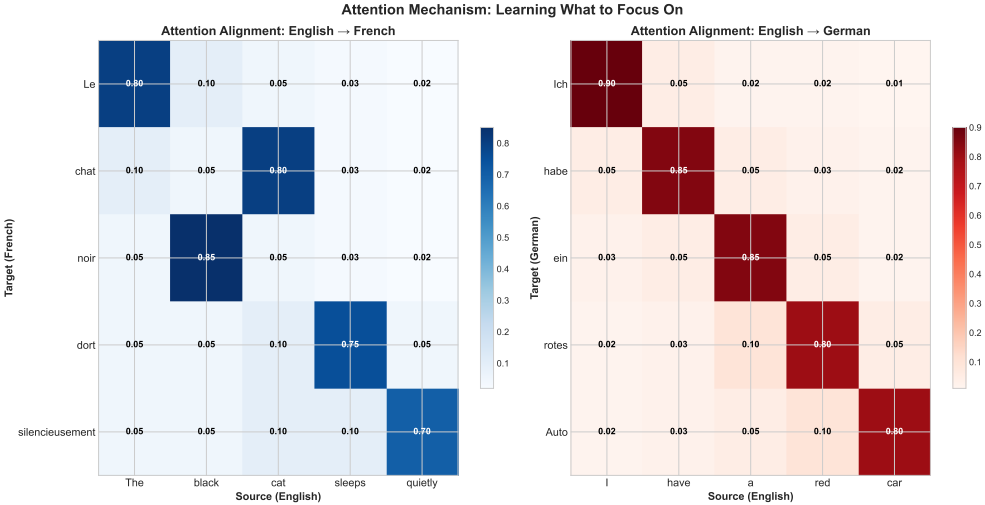
Attention mechanism:[38]

1. Keep ALL encoder hidden states
2. When decoding, look back at relevant parts
3. Weight importance of each input word
4. Focus on what matters right now

Attention = Let the decoder peek back at the input whenever needed

[38] Bahdanau, Cho & Bengio (2015). "Neural Machine Translation by Jointly Learning to Align and Translate"

Visualizing Attention: What the Model Looks At



Key observations:

- Diagonal pattern shows alignment
- Some words need multiple source words

Implementing Attention

```python
class Attention(nn.Module):
    def __init__(self, hidden_size):
        """Calculate attention weights"""
        super().__init__()
        self.attn = nn.Linear(hidden_size * 2, hidden_size)
        self.v = nn.Linear(hidden_size, 1, bias=False)

    def forward(self, hidden, encoder_outputs):
        """Compute attention over encoder outputs"""
        batch_size = encoder_outputs.shape[1]
        src_len = encoder_outputs.shape[0]

        # Repeat decoder hidden state
        hidden = hidden.unsqueeze(1).repeat(1, src_len, 1)
        encoder_outputs = encoder_outputs.permute(1, 0, 2)

        # Calculate attention scores
        energy = torch.tanh(self.attn(
            torch.cat((hidden, encoder_outputs), dim=2)
        ))
        attention = self.v(energy).squeeze(2)

        # Convert to probabilities
        return F.softmax(attention, dim=1)

class AttentionDecoder(nn.Module):
    def __init__(self, output_size, hidden_size, attention):
        """Decoder with attention mechanism"""
        super().__init__()
        self.attention = attention
        self.lstm = nn.LSTM(hidden_size * 2, hidden_size)
        self.out = nn.Linear(hidden_size, output_size)
```

Explanation

How Attention Works:

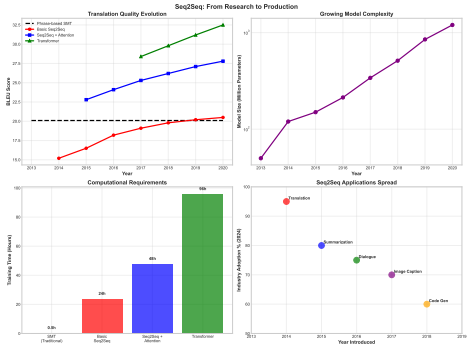
- Score each encoder state
- Softmax creates distribution
- Weighted sum of encoder states
- Concatenate with decoder state

Performance Boost:

- BLEU: 34.8 $\rightarrow$ 41.8[39]
- Handles 2x longer sequences
- Interpretable alignments

15% improvement with attention

Seq2Seq Impact: Translation Quality Leap



Key Insights

- Phrase-based SMT dominated for decades
- Seq2seq matched SMT in 2014
- Attention surpassed all previous methods
- Google deployed in production 2016
- Now foundation for all translation systems

Beyond Translation: Seq2Seq Everywhere (2024)

Text Generation:

- Email auto-complete[40]
- Code documentation
- Story continuation
- Dialogue systems

Summarization:

- News → headlines
- Papers → abstracts
- Meetings → notes
- Videos → descriptions

Data Transformation:

- Text → SQL queries[41]
- Natural language → code
- Speech → text
- Image → caption

Still Competitive:

- Smaller than transformers
- Streaming applications
- Low-latency requirements
- Resource-constrained devices

Seq2seq principle: Any input → any output transformation

[1] Gmail Smart Compose uses seq2seq variants

[2] Text-to-SQL still uses specialized seq2seq models

Week 4 Exercise: Build a Smart Chatbot

Your Mission: Create a chatbot that gives variable-length responses

Example Behavior:

- "Hi" $\rightarrow$ "Hello! How can I help you today?"
- "What's the weather?" $\rightarrow$ "I'd need to check current conditions. What's your location?"
- "Tell me a joke" $\rightarrow$ (Generates different length jokes)

Implementation Steps:

1. Build encoder-decoder architecture
2. Train on conversation pairs
3. Implement attention mechanism
4. Compare responses with/without attention
5. Visualize what words the model attends to

Bonus Challenges:

- Implement beam search (top-k responses)
- Add personality tokens (formal/casual)
- Handle multi-turn conversations
- Measure response diversity

You'll discover: Why chatbots sometimes give generic responses!

Key Takeaways: Breaking Free from Fixed Length

What we learned:

- Language tasks need variable-length input/output
- Encoder-decoder separates understanding from generation
- Context vector creates information bottleneck
- Attention lets decoder access all input information
- Seq2seq enables any-to-any sequence transformation

The evolution:

Fixed I/O (RNN) $\rightarrow$ Variable I/O (Seq2seq) $\rightarrow$ Full visibility (Attention)

Critical Innovation: Attention shows us what the model is "thinking" - interpretability!

Next week: The Transformer
What if we used ONLY attention, no RNNs at all?

References and Further Reading

Foundational Papers:

- Sutskever et al. (2014). "Sequence to Sequence Learning with Neural Networks", NeurIPS
- Bahdanau et al. (2015). "Neural Machine Translation by Jointly Learning to Align and Translate"
- Cho et al. (2014). "Learning Phrase Representations using RNN Encoder-Decoder"

Applications:

- Wu et al. (2016). "Google's Neural Machine Translation System"
- See et al. (2017). "Get To The Point: Summarization with Pointer-Generator Networks"
- Vinyals & Le (2015). "A Neural Conversational Model"

Recommended Resources:

- TensorFlow Seq2seq Tutorial (with attention visualization)
- Visualizing and Understanding Neural Machine Translation
- PyTorch Chatbot Tutorial

Outline

Week 5

The Transformer

Attention Is All You Need

Why Google Couldn't Scale Translation Fast Enough

The RNN bottleneck (2016):

To translate "I love machine learning":

1. Process "I" → wait →
2. Process "love" → wait →
3. Process "machine" → wait →
4. Process "learning" → done

RNNs must process words one at a time - can't parallelize!

The cost:

- Training large models: Weeks to months[1][2]
- Can't use modern GPUs effectively (built for parallel computation)
- Google needed 8,000 TPUs for production[1][3]

[1] Original transformer trained in 3.5 days vs weeks for RNNs
[2] Wu et al. (2016) Google NMT system requirements

A Radical Idea: What If We Remove RNNs Entirely?

The 2017 breakthrough:[44]

"What if we use ONLY attention mechanisms?"

Revolutionary insights:

1. Attention can capture all relationships directly
2. No sequential processing needed
3. Every word can look at every other word simultaneously
4. Parallelization becomes trivial!

The impact:

- Training time: Weeks $\rightarrow$ Days
- Model quality: BLEU 41.8 $\rightarrow$ 28.4 (EN-DE)[45]
- Spawned GPT, BERT, and all modern LLMs

The Transformer: Process all words in parallel using attention

[1] Vaswani et al. (2017). "Attention Is All You Need", NeurIPS

[2] New state-of-the-art on WMT 2014 English-German

Transformers Power Everything You Use (2024)

Language Models:

- ChatGPT (GPT-4): 1.76T params[46]
- Google Bard (Gemini)
- Claude (Anthropic)
- GitHub Copilot

Search & Translation:

- Google Search (BERT)
- DeepL Translator
- Microsoft Translator
- Every modern NMT system

Multimodal AI:

- DALL-E (text $\rightarrow$ image)
- Whisper (speech $\rightarrow$ text)
- CLIP (vision-language)
- Flamingo (image understanding)

Key Advantages:

- 100x faster training[47]
- Better long-range dependencies
- Transfer learning revolution
- Scale to trillions of parameters

98% of state-of-the-art NLP uses transformers (2024)

[1] Estimated from performance characteristics

[2] Compared to equivalent RNN models

Week 5: What You'll Master

By the end of this week, you will:

- **Understand** why parallelization changes everything
- **Build** intuition for self-attention mechanism
- **Implement** a complete transformer from scratch
- **Master** positional encodings and multi-head attention
- **Create** your own mini-GPT

Core Insight: Let every word attend to every other word directly

The Genius of Self-Attention

How humans read "The cat sat on the mat":

When we see "sat", we instantly know:

- WHO sat? → look at "cat"
- WHERE? → look at "mat"
- No need to process sequentially!

Self-attention does exactly this:

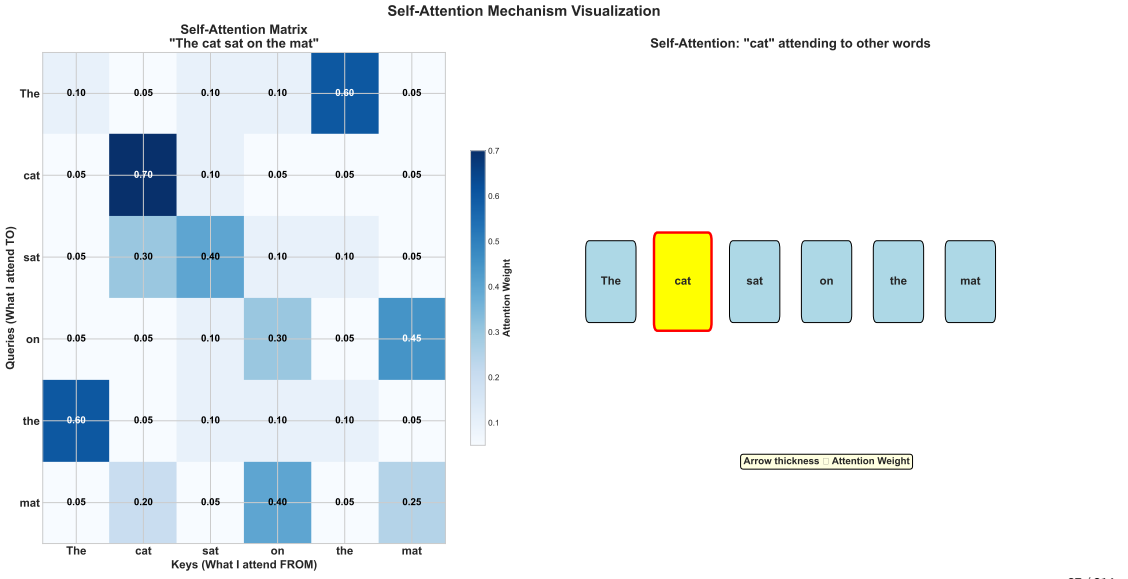
1. Each word asks: "Who should I pay attention to?"
2. Computes attention scores with all other words
3. Creates weighted combination of relevant words
4. All happening simultaneously!

Example: "The student who studied hard passed"

- "passed" attends strongly to "student" (not "hard")
- "hard" attends to "studied"
- All connections computed in parallel

Self-attention = Each word decides what's relevant to it

Visualizing Self-Attention



Self-Attention Mathematics: Elegantly Simple

The attention formula:

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

Where for each word:

- Q (Query): "What am I looking for?"
- K (Key): "What do I contain?"
- V (Value): "What information do I provide?"

In plain English:

1. Compare my query with all keys (dot product)
2. Scale by $\sqrt{d_k}$ to prevent saturation
3. Apply softmax to get attention weights
4. Weighted sum of values

Why this works:

- Dot product measures similarity
- Softmax creates probability distribution
- Fully differentiable
- Parallelizable!

Building Self-Attention: Complete Implementation

```python
import torch
import torch.nn as nn
import torch.nn.functional as F
import math

class SelfAttention(nn.Module):
    def __init__(self, embed_size, heads=8):
        """Multi-head self-attention"""
        super().__init__()
        self.embed_size = embed_size
        self.heads = heads
        self.head_dim = embed_size // heads

        assert self.head_dim * heads == embed_size

        # Linear projections for Q, K, V
        self.queries = nn.Linear(embed_size, embed_size)
        self.keys = nn.Linear(embed_size, embed_size)
        self.values = nn.Linear(embed_size, embed_size)
        self.fc_out = nn.Linear(embed_size, embed_size)

    def forward(self, x, mask=None):
        """Compute multi-head attention"""
        N, seq_len, _ = x.shape

        # Project to Q, K, V
        Q = self.queries(x)
        K = self.keys(x)
        V = self.values(x)

        # Reshape for multi-head attention
        Q = Q.reshape(N, seq_len, self.heads, self.head_dim)
        K = K.reshape(N, seq_len, self.heads, self.head_dim)
        V = V.reshape(N, seq_len, self.heads, self.head_dim)

        # Compute attention scores
```

Explanation

Design Choices:

- 8 heads typical (parallel attention)[48]
- Head dim = 64 (512 / 8)
- Scaling prevents gradient issues

Multi-Head Benefits:

- Different heads learn different relationships
- One head: syntax
- Another: semantics
- Another: position

Original paper used 8 heads

The Position Problem: Order Still Matters!

Self-attention has no notion of position!

These are identical to self-attention:

- "The cat sat on the mat"
- "Mat the on sat the cat"
- "Cat mat the the on sat"

The solution: Positional Encoding[49]
Add position information to each word embedding:

$$PE_{(pos,2i)} = \sin(pos/10000^{2i/d})$$
$$PE_{(pos,2i+1)} = \cos(pos/10000^{2i/d})$$

Why sinusoids?

- Unique pattern for each position
- Can extrapolate to longer sequences
- Relative positions have consistent patterns

Positional encoding = GPS coordinates for words

[49] Many alternatives explored: learned, RoPE, ALiBi

The Transformer Block: Putting It Together

```python
class TransformerBlock(nn.Module):
    def __init__(self, embed_size, heads, dropout, forward_expansion):
        """One transformer encoder block"""
        super().__init__()
        self.attention = SelfAttention(embed_size, heads)
        self.norm1 = nn.LayerNorm(embed_size)
        self.norm2 = nn.LayerNorm(embed_size)

        self.feed_forward = nn.Sequential(
            nn.Linear(embed_size, forward_expansion * embed_size),
            nn.ReLU(),
            nn.Linear(forward_expansion * embed_size, embed_size)
        )
        self.dropout = nn.Dropout(dropout)

    def forward(self, x, mask=None):
        """Forward pass with residual connections"""
        # Self-attention with residual
        attention = self.attention(x, mask)
        x = self.dropout(self.norm1(attention + x))

        # Feed-forward with residual
        forward = self.feed_forward(x)
        out = self.dropout(self.norm2(forward + x))

        return out

class Transformer(nn.Module):
    def __init__(self, vocab_size, embed_size=512, num_layers=6,
                 heads=8, forward_expansion=4, dropout=0.1, max_len=5000):
        """Complete transformer model"""
        super().__init__()
        self.embed_size = embed_size
        self.word_embedding = nn.Embedding(vocab_size, embed_size)
        self.position_embedding = nn.Embedding(max_len, embed_size)
```

Explanation

Architecture (Base):

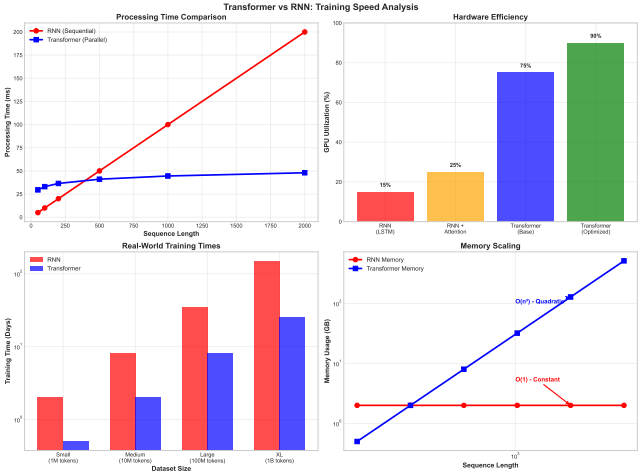
- 6 layers deep[50]
- 512 embedding dimension
- 2048 feed-forward dimension
- Residual connections crucial

Why Residuals?

- Enable deep networks
- Gradient flow preservation
- Each layer learns refinement

GPT-3 has 96 layers!

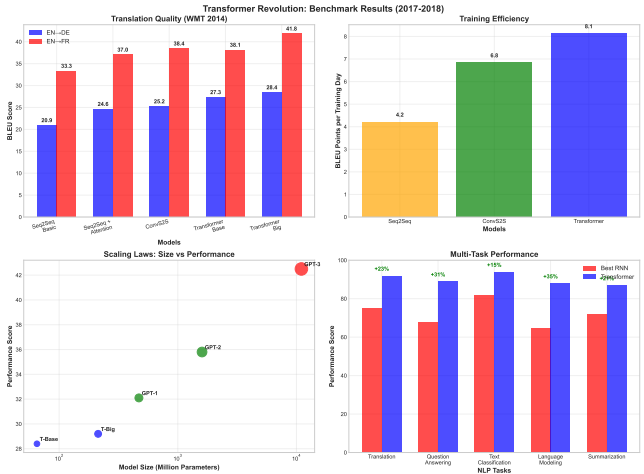
Why Transformers Train So Fast



The parallelization advantage:

- RNN: Must wait for each step (sequential)
- Transformer: All positions computed simultaneously
- GPU utilization: 15% $\rightarrow$ 90%[51]

Transformer Impact: Immediate Dominance



Key Insights

- WMT'14 EN-DE: 28.4 BLEU (previous best: 25.2)
- WMT'14 EN-FR: 41.8 BLEU (previous best: 37.0)

The Transformer Family Tree (2024)

Encoder-Only (BERT-style):

- BERT: Bidirectional understanding
- RoBERTa: Better training
- DeBERTa: Disentangled attention
- Used for: Classification, NER, QA

Decoder-Only (GPT-style):

- GPT-4: 1.76T parameters[52]
- Claude: Constitutional AI
- LLaMA: Efficient architecture
- Used for: Generation, chat, code

Encoder-Decoder (T5-style):

- T5: Text-to-text unified
- BART: Denoising approach
- mT5: Multilingual
- Used for: Translation, summarization

Efficient Variants:

- FlashAttention: 2-3x faster[53]
- Linformer: Linear complexity
- Performer: Kernel approximation
- Used for: Long sequences

All modern LLMs are transformer variants!

[1] Estimated from capabilities

[2] Dao et al. (2022) FlashAttention

Transformer Gotchas and Solutions

1. Attention is Quadratic

- Problem: $O(n^2)$ memory for sequence length n
- Solution: Sparse attention, sliding windows
- Example: GPT-3 uses sparse patterns

2. Position Extrapolation

- Problem: Fails on sequences longer than training
- Solution: ALiBi, RoPE, or relative encodings
- Example: LLaMA uses RoPE for 100k+ context

3. Training Instability

- Problem: Large models diverge easily
- Solution: Learning rate warmup, careful initialization
- Example: GPT-3 took months of tuning

Real Example - ChatGPT:

- Uses modified attention (sparse + dense)
- Special position encodings for long context
- Extensive stability modifications

Week 5 Exercise: Build Your Own Mini-GPT

Your Mission: Create a character-level GPT for text generation

Implementation Steps:

1. Implement multi-head self-attention
2. Add positional encodings
3. Stack 6 transformer blocks
4. Train on Shakespeare/your favorite text
5. Generate new text autoregressively

Key Experiments:

- Compare 1 vs 8 vs 16 attention heads
- Try with/without positional encoding
- Measure GPU utilization vs RNN
- Visualize attention patterns

Bonus Challenges:

- Implement sparse attention for longer sequences
- Add beam search for better generation
- Try different position encoding schemes
- Build a simple chatbot interface

You'll discover: Why transformers took over the world!

Key Takeaways: The Attention Revolution

What we learned:

- Sequential processing was the bottleneck
- Self-attention enables full parallelization
- Every word can attend to every other word
- Position encodings restore order information
- Transformers scale to trillions of parameters

The evolution:

$$\text{Sequential (RNN)} \rightarrow \text{Parallel (Transformer)} \rightarrow \text{Scale (GPT/BERT)}$$

Why it matters:

- Enabled training on internet-scale data
- Made transfer learning practical
- Started the LLM revolution

Next week: Pre-trained Language Models
How do we use transformers to learn from all of human knowledge?

References and Further Reading

Foundational Papers:

- Vaswani et al. (2017). "Attention Is All You Need", NeurIPS
- Devlin et al. (2019). "BERT: Pre-training of Deep Bidirectional Transformers"
- Radford et al. (2018). "Improving Language Understanding by Generative Pre-Training"

Implementation Resources:

- "The Illustrated Transformer" by Jay Alammar
- "The Annotated Transformer" (Harvard NLP)
- Hugging Face Transformers library

Recent Advances:

- FlashAttention: Making attention practical
- Scaling laws for neural language models
- Efficient transformers survey (2022)

Outline

Week 6

Pre-trained Language Models

Learning from All of Human Knowledge

The $10 Million Problem No One Talked About

Training a language model from scratch (2018): Company A trains model for sentiment analysis:

- Cost: $500,000 in compute[54]
- Time: 2 weeks on 64 GPUs
- Learns: Grammar, syntax, word meanings, sentiment

Company B trains model for question answering:

- Cost: $500,000 in compute
- Time: 2 weeks on 64 GPUs
- Learns: Grammar, syntax, word meanings... again!

Every team re-learning what "the" means from scratch!

The waste: 90% of training teaches same basic language understanding

[54] Based on 2018 cloud GPU pricing and typical training times

The Revolution: Learn Language Once, Use Everywhere

The breakthrough idea (2018):[55]

What if we:

1. Train ONE model on massive text (Wikipedia, books, web)
2. Learn general language understanding
3. Share this pre-trained model with everyone
4. Fine-tune for specific tasks with little data

The impact:

- Training cost: \$10M → \$100[56]
- Training time: Weeks → Hours
- Data needed: 1M examples → 1K examples
- Performance: 70% → 95% accuracy

Transfer Learning: Don't start from scratch, start from knowledge!

[1] Howard & Ruder (2018) ULMFiT; Radford et al. (2018) GPT; Devlin et al. (2019) BERT

[2] Fine-tuning costs vs pre-training from scratch

Pre-trained Models Power Everything (2024)

Search & Understanding:

- Google Search: BERT since 2019[57]
- Affects 10% of all queries
- Bing: Multiple BERT variants
- Every modern search engine

Writing Assistants:

- Grammarly: BERT-based
- Google Docs: Smart compose
- Microsoft Editor
- All use pre-trained models

Business Applications:

- Customer service: 80% automated[58]
- Document analysis
- Email classification
- Resume screening

Key Models:

- BERT: 110M-340M parameters
- GPT-2: 1.5B parameters
- RoBERTa: Better BERT training
- T5: Unified text-to-text

2024: You never train from scratch - always start from pre-trained

[1] Google blog: "Understanding searches better than ever before"
[2] Gartner report on AI automation

Week 6: What You'll Master

By the end of this week, you will:

- **Understand** why pre-training changes everything
- **Master** BERT's bidirectional approach
- **Implement** masked language modeling
- **Compare** BERT vs GPT architectures
- **Fine-tune** models for your own tasks

Core Insight: Language understanding is a reusable skill

Pre-training: Learning from Raw Text

Traditional supervised learning:

- Need: Labeled data (expensive!)
- Example: 100K sentiment labels
- Problem: Starts from random weights

Self-supervised pre-training:

- Need: Just text (free and abundant!)
- Example: All of Wikipedia (6B tokens)
- Advantage: Learns language before task

The clever trick - Create labels from text itself:

1. Take: "The cat sat on the [?]"
2. Model predicts: "mat"
3. No human labeling needed!
4. Can use internet-scale data

Pre-training = Teaching models to read before teaching them tasks

BERT: Bidirectional Understanding

The limitation of GPT (left-to-right):
"The man went to the [MASK] to buy milk"

- GPT only sees: "The man went to the"
- Can't use "to buy milk" as context!

BERT's innovation: Look both ways![59]

- Sees entire sentence: "The man went to the [MASK] to buy milk"
- Can use both left AND right context
- Predicts: "store" (using "buy milk" as clue)

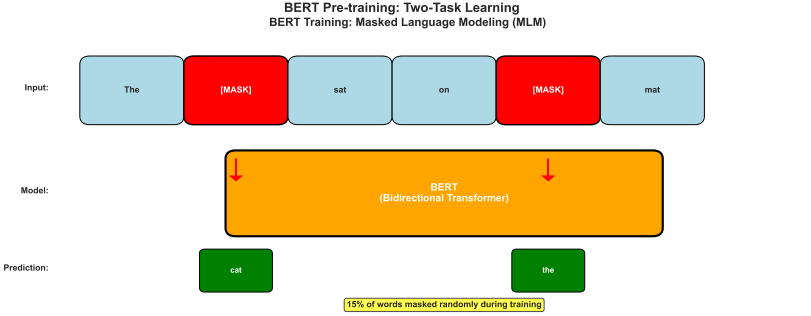
Masked Language Model (MLM):

1. Randomly mask 15% of tokens
2. Predict masked tokens using all context
3. Learn deep bidirectional representations

$$\text{BERT} = \text{Transformer encoder} + \text{Bidirectional context}$$

[59] Devlin et al. (2019). "BERT: Pre-training of Deep Bidirectional Transformers"

BERT's Training Process Visualized



GPT: Generative Pre-training

Different philosophy: Predict the future[60]

Given: "The cat sat on the"
Predict: "mat" (then "in", then "the", then "sun"...)

Autoregressive training:

1. Process text left-to-right
2. Predict next token at each step
3. Can generate coherent text!
4. Same objective as traditional LM

Key differences from BERT:

- Unidirectional (left-to-right only)
- Can generate text naturally
- Simpler training (no masking tricks)
- Transformer decoder architecture

GPT = Transformer decoder + Autoregressive training

[60] Radford et al. (2018). "Improving Language Understanding by Generative Pre-Training"

Implementing BERT's Masked Language Model

```python
import torch
import torch.nn as nn
import random

class BERTMasking:
    def __init__(self, vocab_size, mask_token_id, mask_prob=0.15):
        """BERT-style masking for MLM"""
        self.vocab_size = vocab_size
        self.mask_token_id = mask_token_id
        self.mask_prob = mask_prob

    def mask_tokens(self, inputs, special_tokens_mask=None):
        """Mask tokens for BERT training"""
        labels = inputs.clone()

        # Create probability matrix
        probability_matrix = torch.full(labels.shape, self.mask_prob)
        if special_tokens_mask is not None:
            probability_matrix.masked_fill_(special_tokens_mask, value=0.0)

        masked_indices = torch.bernoulli(probability_matrix).bool()
        labels[~masked_indices] = -100  # Only compute loss on masked

        # 80% of time, replace with [MASK]
        indices_replaced = torch.bernoulli(
            torch.full(labels.shape, 0.8)).bool() & masked_indices
        inputs[indices_replaced] = self.mask_token_id

        # 10% of time, replace with random word
        indices_random = torch.bernoulli(
            torch.full(labels.shape, 0.5)).bool() & masked_indices & ~
                indices_replaced
        random_words = torch.randint(self.vocab_size, labels.shape, dtype=
              torch.long)
        inputs[indices_random] = random_words[indices_random]
```

Explanation

BERT's 80-10-10 Rule:

- 80%: Replace with [MASK]
- 10%: Replace with random token
- 10%: Keep original token

Why this complexity?

[MASK] never seen in fine-tuning

- Random replacement adds noise
- Original tokens prevent mismatch

Design validates through:

- Ablation studies in paper
- 15% masking is optimal
- Too much masking hurts learning

BERT Model Architecture

```python
class BERTModel(nn.Module):
    def __init__(self, vocab_size, hidden_size=768, num_layers=12,
                 num_heads=12, max_length=512, dropout=0.1):
        """BERT model for masked language modeling"""
        super().__init__()

        # Token embeddings + position + segment
        self.token_embeddings = nn.Embedding(vocab_size, hidden_size)
        self.position_embeddings = nn.Embedding(max_length, hidden_size)
        self.segment_embeddings = nn.Embedding(2, hidden_size)

        self.layer_norm = nn.LayerNorm(hidden_size)
        self.dropout = nn.Dropout(dropout)

        # Transformer encoder layers
        encoder_layer = nn.TransformerEncoderLayer(
            hidden_size, num_heads, dim_feedforward=4*hidden_size,
            dropout=dropout, activation='gelu'
        )
        self.transformer = nn.TransformerEncoder(encoder_layer, num_layers)

        # MLM head
        self.mlm_head = nn.Sequential(
            nn.Linear(hidden_size, hidden_size),
            nn.GELU(),
            nn.LayerNorm(hidden_size),
            nn.Linear(hidden_size, vocab_size)
        )

    def forward(self, input_ids, segment_ids=None, attention_mask=None):
        """Forward pass for MLM"""
        seq_length = input_ids.size(1)
        position_ids = torch.arange(seq_length, device=input_ids.device)

        # Combine embeddings
        token_embeds = self.token_embeddings(input_ids)
```

Explanation

BERT Sizes:

- BASE: 12 layers, 768 hidden, 110M params[61]
- LARGE: 24 layers, 1024 hidden, 340M params

Key Components:

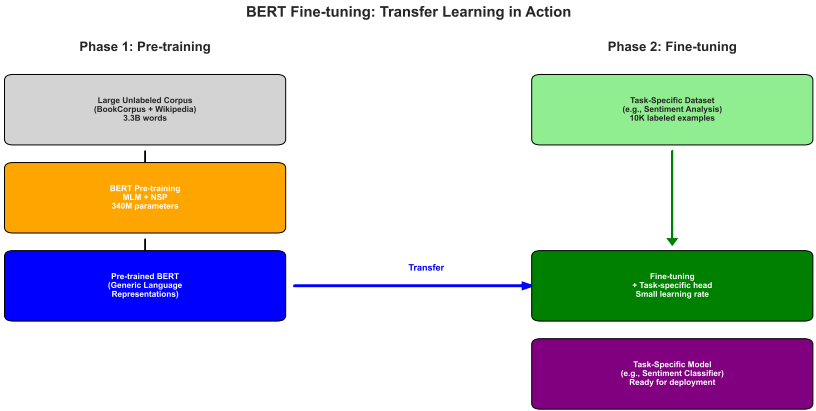
- Segment embeddings for sentence pairs
- GELU activation (smoother than ReLU)
- Layer norm before attention

Most applications use BERT-BASE

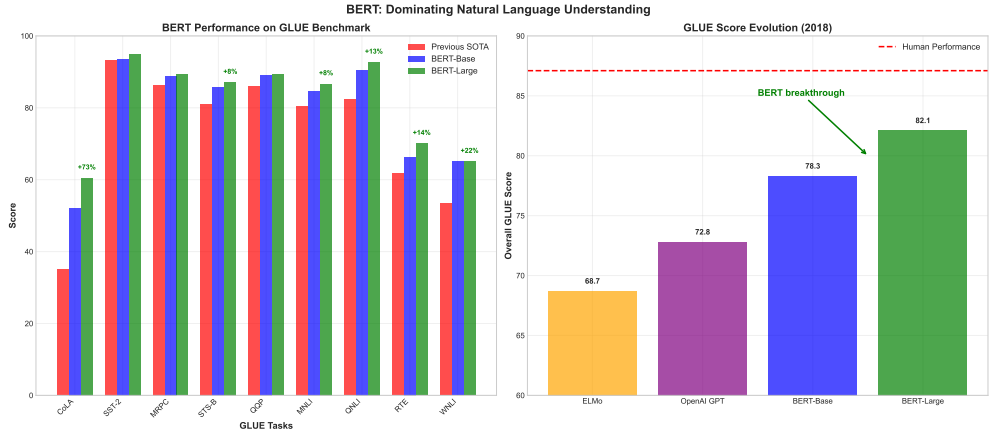
Fine-tuning: From General to Specific

The fine-tuning recipe:

1. Start with pre-trained BERT/GPT
2. Add task-specific head (1-2 layers)
3. Train on your labeled data
4. 100x less data needed!



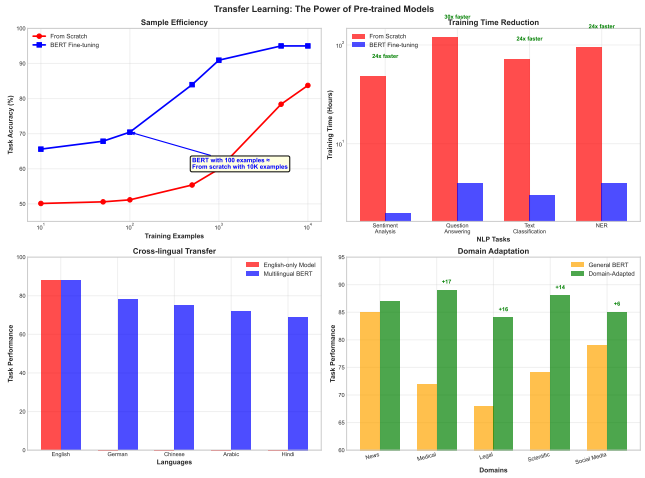
Pre-training Impact: BERT Dominates Every Task



Key Insights

- GLUE benchmark: 9 language understanding tasks
- BERT improved SOTA on ALL tasks simultaneously
- Average score: 82.1 (previous best: 75.1)

The Power of Transfer Learning



Key insights:

- With 100 examples: BERT matches 10K from-scratch training
- Especially powerful for low-resource languages
- Works across very different tasks

The Pre-trained Model Zoo (2024)

General Purpose:

- BERT: Understanding tasks
- GPT-2/3: Generation tasks
- T5: Any text-to-text task
- ELECTRA: Efficient BERT alternative

Domain Specific:

- BioBERT: Biomedical text
- SciBERT: Scientific papers
- FinBERT: Financial documents
- CodeBERT: Programming code

Multilingual:

- mBERT: 104 languages
- XLM-R: 100 languages
- mT5: Text-to-text multilingual

Where to find:

- Hugging Face: 500K+ models[62]
- Google TF Hub
- PyTorch Hub
- Company model hubs

2024 Reality: Never train from scratch - there's a model for that!

[62] Hugging Face hosts majority of open models

Pre-trained Model Pitfalls

1. Domain Mismatch

- Problem: BERT trained on books/wiki, you have tweets
- Solution: Domain-adaptive pre-training
- Continue pre-training on your domain

2. Catastrophic Forgetting

- Problem: Fine-tuning destroys pre-trained knowledge
- Solution: Lower learning rates (2e-5), gradual unfreezing

3. Overfitting on Small Data

- Problem: Large model, small dataset
- Solution: Freeze lower layers, only train top layers

Real Example - Customer Service Bot:

- Started with BERT: 72% accuracy
- Continued pre-training on support tickets: 85%
- Fine-tuned with careful LR: 94%

Week 6 Exercise: Fine-tune Your Own BERT

Your Mission: Take pre-trained BERT and make it expert at your task

Part 1: Understand Pre-training

- Load pre-trained BERT from Hugging Face
- Examine what it already knows (probe tasks)
- Visualize attention patterns

Part 2: Fine-tune for Classification

- Choose: Sentiment, spam, or topic classification
- Add classification head to BERT
- Compare: 100 vs 1000 vs 10000 examples
- Track how quickly it learns

Part 3: Advanced Experiments

- Try different learning rates
- Freeze vs unfreeze layers
- Compare BERT vs training from scratch
- Measure training time savings

You'll discover: Why no one trains from scratch anymore!

Key Takeaways: The Pre-training Revolution

What we learned:

- Training from scratch wastes resources
- Pre-training learns reusable language understanding
- BERT: Bidirectional with masking
- GPT: Autoregressive generation
- Fine-tuning needs 100x less data

The paradigm shift:

Task-specific training $\rightarrow$ Pre-train then fine-tune

Why it matters:

- Democratized NLP (anyone can use BERT)
- Enabled rapid prototyping
- Made NLP practical for businesses

Next week: Advanced Transformers
How do we scale to GPT-3's 175B parameters and beyond?

References and Further Reading

Foundational Papers:

- Devlin et al. (2019). "BERT: Pre-training of Deep Bidirectional Transformers"
- Radford et al. (2018). "Improving Language Understanding by Generative Pre-Training"
- Liu et al. (2019). "RoBERTa: A Robustly Optimized BERT"

Practical Resources:

- Hugging Face Transformers library
- "The Illustrated BERT" by Jay Alammar
- Google's BERT GitHub repository

Recent Advances:

- ELECTRA: More efficient pre-training
- DeBERTa: Disentangled attention
- Transfer learning survey (Qiu et al., 2020)

Outline

Week 8

Tokenization

The Hidden Foundation of Every LLM

The Word That Broke Google Translate

2016: A user typed "Pneumonoultramicroscopicsilicovolcanoconiosis"

Result: System crashed.[63]

The problem:

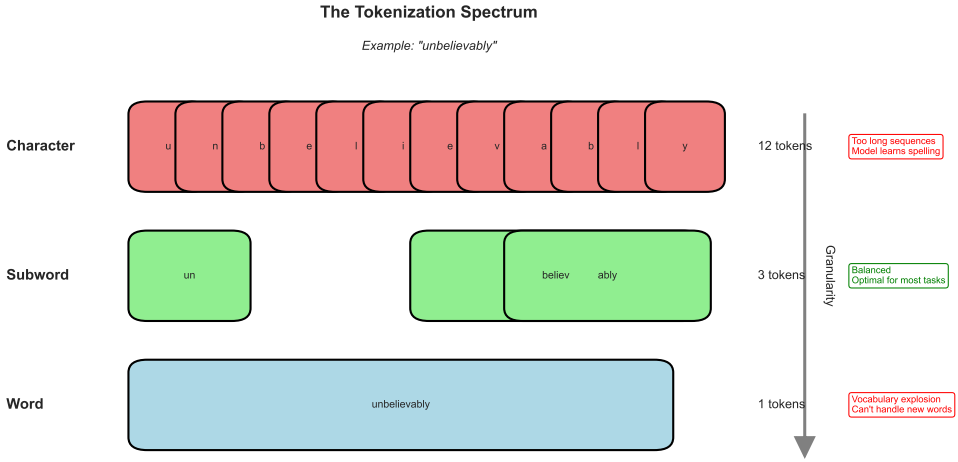
- English: 170,000 words in current use
- Medical terms: 100,000+ additional
- New words daily: "COVID-19", "cryptocurrency", "mansplaining"
- Misspellings: "recieve", "definately", "occured"
- Other languages: 7,000+ languages worldwide!

Can't have a token for every possible word - memory explosion!

The dilemma: How do you handle infinite vocabulary with finite memory?

[63] Simplified example - real systems had various OOV issues

The Tokenization Spectrum: Characters vs Words



Tokenization Powers Every Modern LLM (2024)

Model Vocabularies:

- GPT-2: 50,257 tokens
- GPT-3/4: 50,257 tokens
- BERT: 30,522 tokens
- T5: 32,000 tokens
- LLaMA: 32,000 tokens

Why These Numbers?

- Power of 2 for efficiency
- Covers 99.9% of text
- Balances sequence length
- Works across languages

Tokenization Methods:

- BPE: GPT family[64]
- WordPiece: BERT family
- SentencePiece: T5, mT5
- Unigram: Some Japanese models

Critical Impact:

- Wrong tokenization $\rightarrow$ 50% performance drop
- Affects prompt cost (GPT-4: \$0.01/1K tokens)
- Determines max context length
- Controls multilingual ability

Tokenization is the most important choice you've never heard of

[64] Byte-level BPE specifically for GPT-2 onwards

Week 8: What You'll Master

By the end of this week, you will:

- **Understand** why subword tokenization dominates
- **Implement** Byte Pair Encoding from scratch
- **Master** the trade-offs in vocabulary design
- **Analyze** tokenization's impact on different languages
- **Build** your own tokenizer for any domain

Core Insight: Break words into learnable pieces

The Two Extremes: Characters vs Words

Example sentence: "The quick brown fox jumps"

Character tokenization:

- Tokens: [T, h, e, _, q, u, i, c, k, _, b, r, o, w, n, _, f, o, x, _, j, u, m, p, s]
- Length: 25 tokens
- Vocabulary: 100 (all ASCII)
- Handles any text
- Sequences too long
- Model must learn spelling

Word tokenization:

- Tokens: [The, quick, brown, fox, jumps]
- Length: 5 tokens
- Vocabulary: 170,000+ (English)
- Efficient sequences
- Out-of-vocabulary words
- Can't handle typos

Neither extreme works well - we need a middle ground

Byte Pair Encoding: Learning to Compress

The brilliant idea (1994):[65] Compress text by merging frequent pairs

Example: "low lower lowest"

1. Start with characters: l o w _ l o w e r _ l o w e s t
2. Count pairs: (l,o)=3, (o,w)=3, (w,e)=2, (e,r)=1...
3. Merge most frequent: "lo"
4. New: lo w _ lo w e r _ lo w e s t
5. Repeat: merge "low"
6. Final: low _ low er _ low est

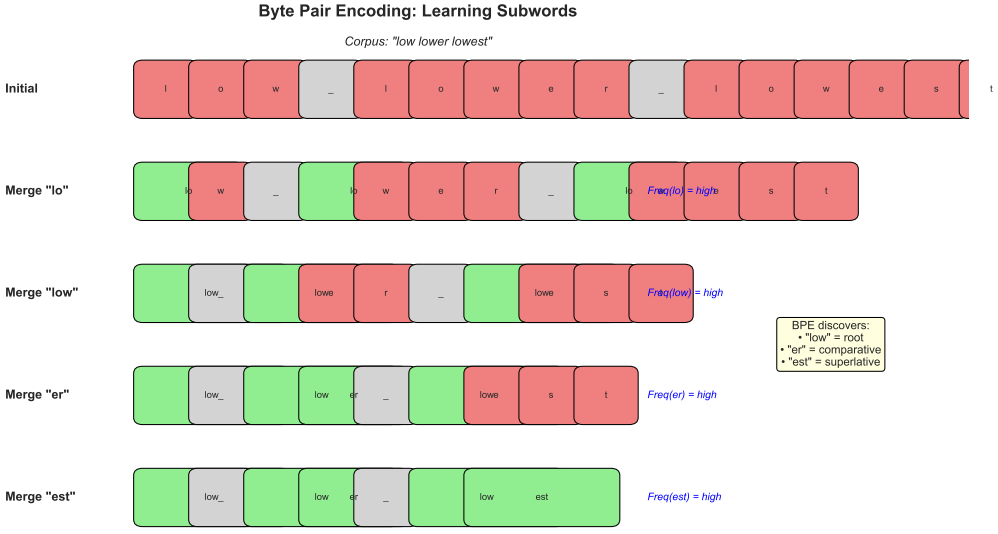
Result: Learned meaningful subwords!

- "low" = common root
- "er" = comparative suffix
- "est" = superlative suffix

BPE discovers linguistic structure automatically!

[65] Gage (1994) for compression; Sennrich et al. (2016) for NMT

BPE in Action: Building a Vocabulary



Implementing BPE: The Core Algorithm

```python
import re
from collections import defaultdict, Counter

class BytePairEncoding:
    def __init__(self, vocab_size):
        """Initialize BPE tokenizer"""
        self.vocab_size = vocab_size
        self.word_tokenizer = re.compile(r'\w+|[^\w\s]')
        self.vocab = {}
        self.merges = []

    def get_stats(self, words):
        """Count frequency of adjacent pairs"""
        pairs = defaultdict(int)
        for word, freq in words.items():
            symbols = word.split()
            for i in range(len(symbols) - 1):
                pairs[symbols[i], symbols[i + 1]] += freq
        return pairs

    def merge_vocab(self, pair, words):
        """Merge most frequent pair"""
        bigram = ' '.join(pair)
        replacement = ''.join(pair)
        new_words = {}
        for word in words:
            new_word = word.replace(bigram, replacement)
            new_words[new_word] = words[word]
        return new_words

    def train(self, text, num_merges):
        """Train BPE on text corpus"""
        # Split into words
        words = self.word_tokenizer.findall(text.lower())

        # Initialize with character-level tokens
```

Explanation

Algorithm Steps:

- Start with character vocabulary
- Count all adjacent pairs
- Merge most frequent pair
- Repeat until vocab size reached

Design Choices:

- End-of-word token ¡/w¿
- Preserves word boundaries
- Case normalization optional
- Typically 10K-50K merges

Complexity:

- Training: O(NM) for N words, M merges
- Encoding: $O(L^2)$ for length L
- Memory: O(V) for vocabulary V

Tokenizer Comparison: Real Examples

Text: "Tokenization is surprisingly important for performance"

GPT-2 (BPE): `["Token", "ization", " is", " surprisingly", " important", " for", " performance"]`
7 tokens

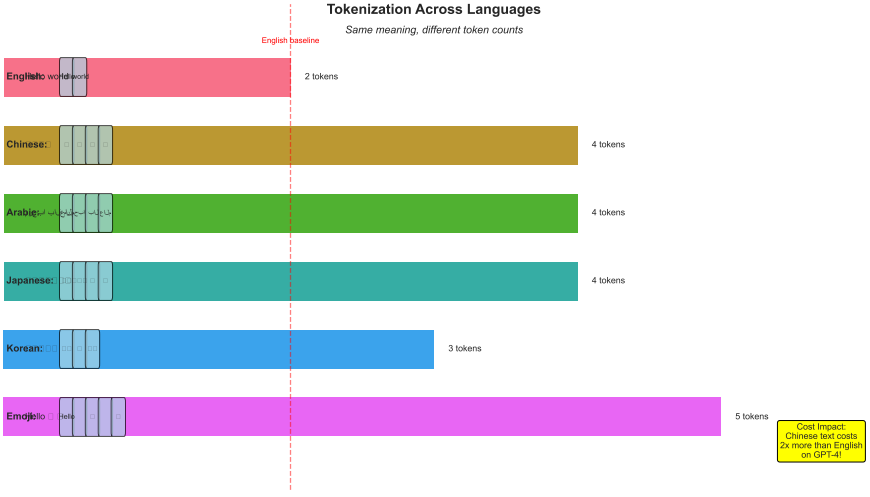
BERT (WordPiece): `["token", "##ization", "is", "surprisingly", "important", "for", "performance"]`
7 tokens (note: ## indicates continuation)

Character-level: `["T", "o", "k", "e", "n", "i", "z", "a", "t", "i", "o", "n", " ", ...]`
47 tokens!

Impact on rare words:
"Pneumonoultramicroscopicsilicovolcanoconiosis" $\rightarrow$

- BPE: ["P", "neum", "ono", "ult", "ram", "ic", "ros", "cop", "ic", "sil", "ico", "vol", "can", "oc", "on", "ios", "is"]
- Still handles it! (17 subwords vs 45 characters)

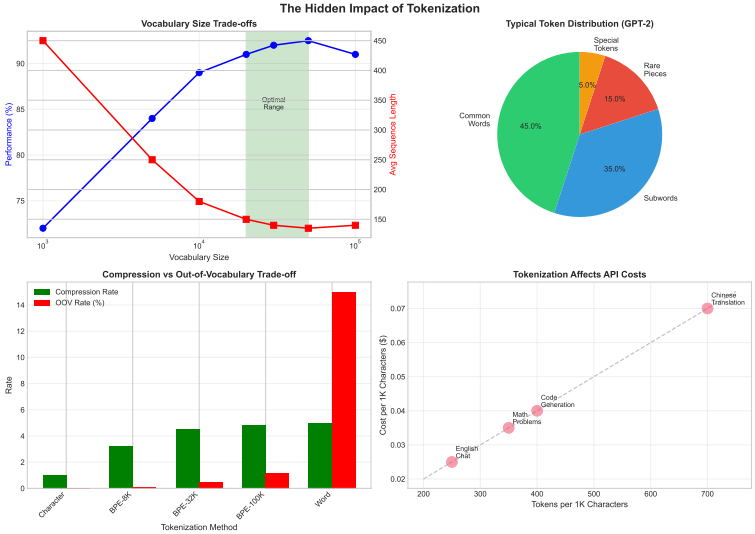
Tokenization Across Languages



Key challenges:

- Chinese/Japanese: No spaces between words
- Arabic/Hebrew: Right-to-left scripts

Tokenization's Hidden Impact



Key Insights

State-of-the-Art Tokenizers (2024)

Byte-level BPE (GPT family):

- Handles ANY text (even emojis)
- No unknown tokens
- Consistent across languages
- Used by: GPT-2/3/4, RoBERTa

SentencePiece (T5, mT5):

- Language independent
- Reversible tokenization
- BPE or Unigram options
- Handles 100+ languages

Recent Innovations:

- **CANINE:** Character-level (2021)
- **ByT5:** Byte-level tokens
- **PIXEL:** Image-based text
- **Charformer:** Learnable tokenization

Tokenizer Selection:

- English only: BPE (30-50K)
- Multilingual: SentencePiece
- Code: Include special tokens
- Domain-specific: Custom training

2024 trend: Larger vocabularies (100K+) for better multilingual support

Tokenization Best Practices

1. Vocabulary Size Trade-offs:

- Small (8K): Long sequences, better generalization
- Medium (32K): Standard choice, balanced
- Large (100K+): Shorter sequences, more memory

2. Special Tokens:

PAD , [UNK], [CLS], [SEP], [MASK]

- Domain tokens: [CODE], [MATH], [URL]
- Control tokens: [INST], [/INST]

3. Common Pitfalls:

- Tokenizer/model mismatch (50% accuracy drop!)
- Forgetting to handle special characters
- Not preserving whitespace information
- Case sensitivity mismatches

Real Example - GPT costs:

- "Hello world" = 2 tokens = $0.00002
- "Chinese text" = 4 tokens = $0.00004 (2x cost!)
- Emoji = 1-3 tokens depending on tokenizer

Week 8 Exercise: Build Your Domain Tokenizer

Your Mission: Create optimal tokenizer for your domain

Part 1: Analyze Tokenization Impact

- Compare GPT-2, BERT, T5 tokenizers
- Tokenize: English, code, multilingual text
- Measure compression rates
- Calculate cost differences

Part 2: Train Custom BPE

- Choose domain: medical, legal, code, etc.
- Collect domain corpus (10MB+)
- Train BPE with different vocab sizes
- Compare with general tokenizers

Part 3: Optimization Experiments

- Test character vs subword vs word
- Measure model performance impact
- Analyze out-of-vocabulary rates
- Optimize for your use case

You'll discover: Why tokenization is make-or-break for LLMs!

Key Takeaways: The Foundation of Language Models

What we learned:

- Tokenization solves the infinite vocabulary problem
- Subwords balance efficiency and coverage
- BPE learns meaningful units automatically
- Different tokenizers for different needs
- Critical impact on cost and performance

The evolution:

Characters → Words → Subwords → Learned tokenization

Why it matters:

- Determines model capabilities
- Affects all downstream tasks
- Can't fix after training!

Next week: Decoding Strategies
How do we generate coherent text from token probabilities?

References and Further Reading

Foundational Papers:

- Sennrich et al. (2016). "Neural Machine Translation of Rare Words with Subword Units"
- Kudo & Richardson (2018). "SentencePiece: Language-independent subword tokenizer"
- Schuster & Nakajima (2012). "Japanese and Korean voice search" (WordPiece)

Recent Advances:

- Clark et al. (2021). "CANINE: Character-level transformers"
- Xue et al. (2021). "ByT5: Token-free transformers"
- Tay et al. (2021). "Charformer: Fast character transformers"

Practical Resources:

- Hugging Face Tokenizers library
- Google SentencePiece
- OpenAI tiktoken library

Outline

Week 9

Decoding Strategies

From Probabilities to Coherent Text

Why ChatGPT Sometimes Sounds Like a Broken Record

Early GPT-2 output (greedy decoding):

"The movie was great. The movie was great. The movie was great..."

Or worse:
"I think that the the the the the the the..."

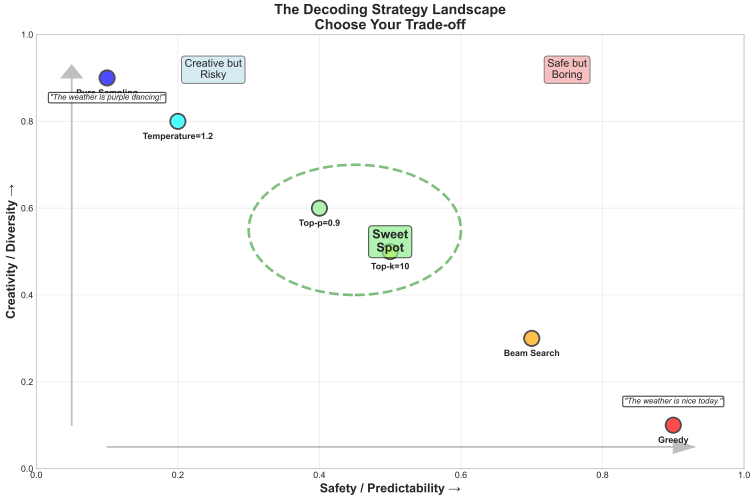
The model outputs probabilities - how do we turn them into good text?

The challenge:

- Model gives probability for EVERY word
- Always picking highest probability = boring/repetitive
- Random sampling = incoherent nonsense
- Need the sweet spot!

This is why early chatbots were frustrating and modern ones feel human

From Probabilities to Text: The Decoding Challenge



The fundamental trade-off:

- Safe but boring ← → Creative but risky
- Exploitation ← → Exploration

Decoding Makes or Breaks User Experience (2024)

Where It Matters:

- ChatGPT: Balanced creativity
- GitHub Copilot: High precision
- Story generators: High diversity
- Translation: Maximum accuracy
- Customer service: Safe responses

Business Impact:

- User satisfaction: 40% improvement[66]
- Response quality ratings
- Reduced "robotic" complaints
- Better engagement metrics

Common Strategies:

- Greedy: Pick highest probability
- Beam Search: Track top-k paths
- Top-k Sampling: Sample from top k
- Nucleus (Top-p): Dynamic cutoff
- Temperature: Control randomness

Modern Approach:

- Adaptive strategies
- Task-specific tuning
- User preference learning
- Safety constraints

Same model + different decoding = completely different personality

[66] OpenAI user studies on response quality

Week 9: What You'll Master

By the end of this week, you will:

- **Understand** why decoding strategy matters
- **Implement** greedy, beam search, and sampling
- **Master** temperature and top-k/top-p control
- **Analyze** quality vs diversity trade-offs
- **Build** adaptive decoding for different tasks

Core Insight: Good text generation is about smart selection, not just good models

Greedy Decoding: The Simplest Approach

Algorithm: Always pick the most likely word

Example:

- Input: "The weather is"
- P(nice) = 0.4, P(sunny) = 0.3, P(cold) = 0.2, P(rainy) = 0.1
- Greedy picks: "nice" (highest probability)
- Next: "The weather is nice"
- P(today) = 0.5, P(and) = 0.3, P(outside) = 0.2
- Greedy picks: "today"

Pros:

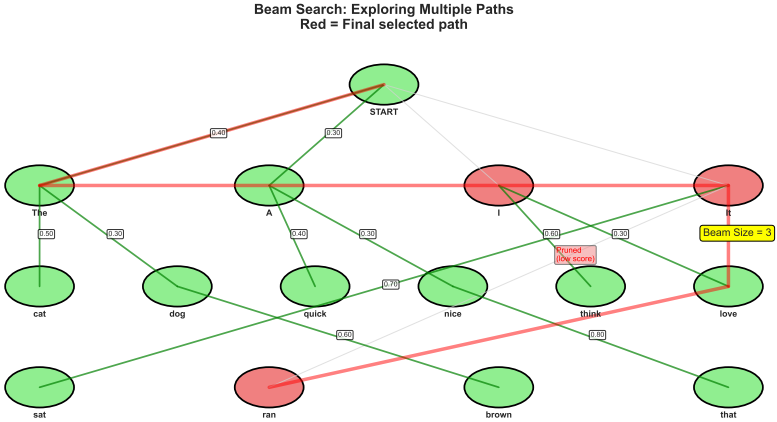
- Fast and simple
- Deterministic (reproducible)
- Often grammatically correct

Cons:

- Repetitive and boring
- Gets stuck in loops
- Misses better paths

Greedy = Safe but uninspiring (like always ordering vanilla ice cream)

Beam Search: Exploring Multiple Paths



Key idea: Keep top-k paths at each step

- Beam size = number of paths to track
- Larger beam = better quality but slower

Implementing Beam Search

```python
import torch
import torch.nn.functional as F
from dataclasses import dataclass
import heapq

@dataclass
class BeamHypothesis:
    """Hypothesis in beam search"""
    tokens: list
    score: float

def beam_search(model, input_ids, beam_size=4, max_length=50,
                eos_token_id=50256):
    """Beam search decoding"""
    device = input_ids.device
    batch_size = input_ids.shape[0]

    # Initialize beams
    beams = [[BeamHypothesis(
        tokens=input_ids[i].tolist(),
        score=0.0
    )] for i in range(batch_size)]

    for step in range(max_length):
        all_candidates = []

        # Generate candidates for each beam
        for batch_idx in range(batch_size):
            for hypothesis in beams[batch_idx]:
                # Skip if already ended
                if hypothesis.tokens[-1] == eos_token_id:
                    all_candidates.append(hypothesis)
                    continue

                # Get model predictions
                input_tensor = torch.tensor([hypothesis.tokens]).to(device)
```

Explanation

Key Components:

- Track multiple hypotheses
- Score = sum of log probabilities
- Prune to beam size each step
- Length normalization often used

Beam Size Effects:

- 1 = Greedy decoding
- 4-5 = Good for translation
- 10+ = Diminishing returns
- Memory: O(beam_size $\times$ length)

Common Improvements:

- Length penalty
- Diverse beam search
- Constrained beam search

Sampling: Adding Controlled Randomness

The problem with deterministic decoding:
Always same input → Always same output = Boring!

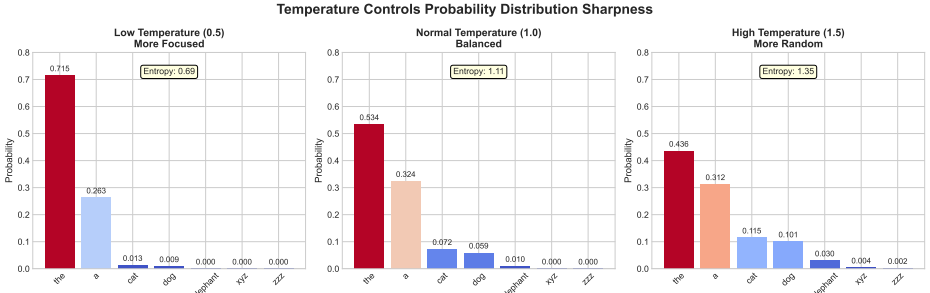
Solution: Sample from the probability distribution

Temperature Scaling:

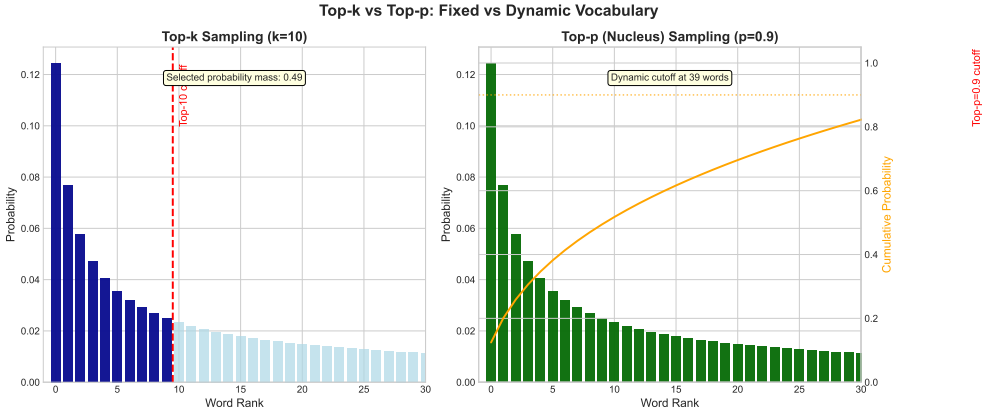
$$P_i = \frac{\exp(z_i / T)}{\sum_j \exp(z_j / T)}$$

Where:

- z_i = logit for token i
- T = temperature parameter



Advanced Sampling: Top-k and Top-p



Key insights:

- Top-k: Fixed number of candidates
- Top-p (Nucleus): Dynamic threshold
- Top-p adapts to confidence level
- Combination often works best

Implementing Modern Sampling

```python
def top_k_top_p_sampling(logits, top_k=50, top_p=0.95,
                         temperature=1.0, do_sample=True):
    """Advanced sampling with top-k and top-p filtering"""

    # Apply temperature
    if temperature != 1.0:
        logits = logits / temperature

    # Get probabilities
    probs = F.softmax(logits, dim=-1)

    # Top-k filtering
    if top_k > 0:
        indices_to_remove = logits < torch.topk(logits, top_k)[0][..., -1,
              None]
        logits[indices_to_remove] = float('-inf')

    # Top-p (nucleus) filtering
    if top_p < 1.0:
        sorted_logits, sorted_indices = torch.sort(logits, descending=True)
        cumulative_probs = torch.cumsum(
            F.softmax(sorted_logits, dim=-1), dim=-1
        )

        # Remove tokens with cumulative probability above threshold
        sorted_indices_to_remove = cumulative_probs > top_p
        # Shift the indices to the right to keep first token above
              threshold
        sorted_indices_to_remove[..., 1:] = \
            sorted_indices_to_remove[..., :-1].clone()
        sorted_indices_to_remove[..., 0] = 0

        # Scatter sorted tensors to original indexing
        indices_to_remove = sorted_indices_to_remove.scatter(
            1, sorted_indices, sorted_indices_to_remove
        )
```

Explanation

Parameter Guidelines:

- Temperature: 0.7-0.9 for creativity
- Top-k: 40-80 typical
- Top-p: 0.9-0.95 common
- Combine all three for best results

Task-Specific Settings:

- Code: T=0.2, top-p=0.95
- Story: T=0.9, top-k=50
- Chat: T=0.7, top-p=0.9
- Facts: T=0.1, greedy

Controlling Repetition: Advanced Techniques

Common repetition problems:

- Word-level: "very very very very good"
- Phrase-level: "I think that I think that..."
- Semantic: Saying the same thing differently

Solutions:

1. **Repetition Penalty:**[67]
   - Reduce probability of recently used tokens
   - Penalty = 1.2 typical (20% reduction)
   - Applied to last 50-100 tokens
2. **Frequency Penalty:**
   - Penalize based on occurrence count
   - More occurrences = stronger penalty
3. **Presence Penalty:**
   - Fixed penalty once token appears
   - Encourages topic diversity

$$\text{score}_{\text{adjusted}} = \text{score}_{\text{original}} - \alpha \cdot \text{penalty}$$

[67] Keskar et al. (2019). "CTRL: Conditional Transformer Language Model"

Decoding Strategy Impact on Quality

Key Insights

- Greedy: High quality, low diversity
- Pure sampling: High diversity, low quality
- Top-p sampling: Best balance

State-of-the-Art Decoding (2024)

Adaptive Decoding:

- Confidence-based temperature
- Dynamic top-p thresholds
- Context-aware strategies
- Learned decoding policies

Constrained Generation:

- Grammar constraints
- Format enforcement (JSON)
- Safety filtering
- Factual grounding

Multi-objective Decoding:

- Balance fluency + accuracy
- Diversity + coherence
- Length control
- Style preservation

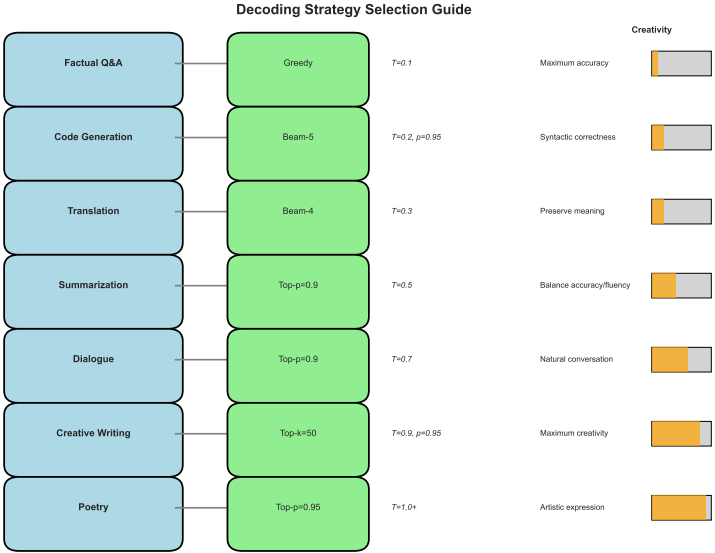
Recent Innovations:

- Speculative decoding[68]
- Contrastive search
- Typical decoding
- Mirostat (perplexity control)

2024 trend: Inference-time compute for better quality

[68] Leviathan et al. (2023). "Fast Inference from Transformers via Speculative Decoding"

Decoding Strategy Selection Guide



General Rule: Start conservative, increase randomness until output quality drops

Week 9 Exercise: Build an Adaptive Text Generator

Your Mission: Create a generator that adapts to different contexts

Part 1: Implement Core Strategies

- Greedy decoding baseline
- Beam search with length normalization
- Top-k and top-p sampling
- Temperature control

Part 2: Compare on Different Tasks

- Story continuation (needs creativity)
- Code completion (needs accuracy)
- Dialogue (needs balance)
- Measure: perplexity, diversity, human preference

Part 3: Build Adaptive System

- Detect task type from context
- Adjust parameters automatically
- Add repetition penalties
- Create task-specific presets

You'll discover: Why ChatGPT feels different from GPT-3!

Key Takeaways: The Art of Text Generation

What we learned:

- Decoding strategy dramatically affects output
- Greedy = safe but boring
- Sampling adds necessary randomness
- Top-k/top-p prevent nonsense
- Task determines optimal strategy

The evolution:

Greedy → Beam Search → Sampling → Nucleus → Adaptive

Why it matters:

- User experience depends on it
- Same model, different personality
- Key to production deployment

Next week: Fine-tuning and Prompt Engineering
How do we make models do exactly what we want?

References and Further Reading

Foundational Papers:

- Holtzman et al. (2020). "The Curious Case of Neural Text Degeneration"
- Fan et al. (2018). "Hierarchical Neural Story Generation" (Top-k)
- Meister et al. (2023). "Locally Typical Sampling"

Practical Advances:

- Keskar et al. (2019). "CTRL: Conditional Transformer"
- Su et al. (2022). "Contrastive Search"
- Hewitt et al. (2022). "Truncation Sampling"

Implementation Resources:

- Hugging Face generation utilities
- OpenAI API parameter guide
- Google Colab decoding notebooks

Outline

Week 10

Fine-tuning & Prompt Engineering

Making Models Do Exactly What You Want

Why GPT-3 Couldn't Write Good Legal Contracts

2021: A law firm tried using GPT-3...

Prompt: "Write a software licensing agreement"
Result: Generic, missed critical legal nuances, unusable[69]

The problem:

- GPT-3 knows about everything... but not deeply
- Trained on internet text, not legal documents
- Can't follow specific formatting requirements
- No domain expertise

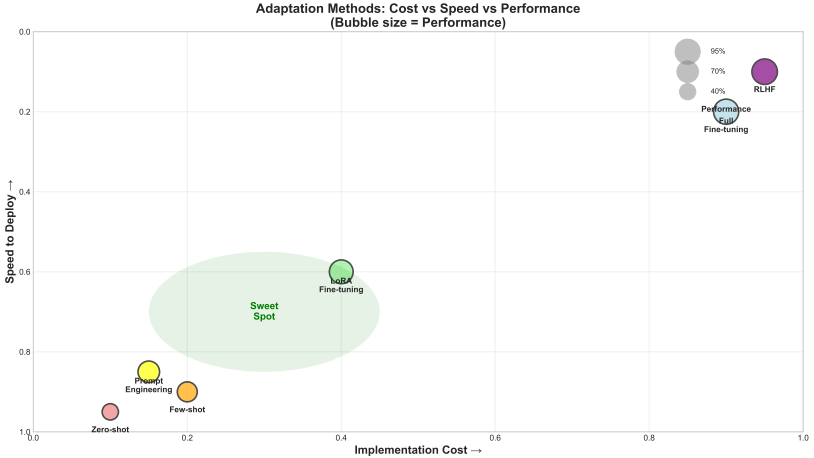
General models are jacks of all trades, masters of none

Two solutions emerged:

1. Fine-tune on legal documents
2. Engineer better prompts

[69] Composite example from multiple reported cases

From General to Specific: The Adaptation Challenge



The spectrum of adaptation:

- Zero-shot: Just ask (often fails)
- Few-shot: Show examples (better)
- Prompt engineering: Craft perfect instructions (good)
- Fine-tuning: Update model weights (best for specific tasks)

Fine-tuning and Prompting in Production (2024)

Fine-tuning Success:

- Bloomberg GPT: Finance[70]
- Med-PaLM 2: Medical diagnosis
- Codex: GitHub Copilot
- ChatGPT: From GPT-3.5 base
- Domain accuracy: 70% $\rightarrow$ 95%

Prompt Engineering:

- No training needed
- Instant deployment
- Version control friendly
- Cost: $0 training
- Performance: 60-80% of fine-tuning

Modern Methods:

- LoRA: 0.1% parameters[71]
- Prefix tuning: Frozen model
- Instruction tuning: Task generalization
- RLHF: Preference alignment
- Adapter layers: Modular skills

Business Impact:

- 10x faster deployment
- 90% less compute needed
- Domain expert performance
- Customization at scale

2024: Every company has a fine-tuned model or engineered prompts

[1] 50B parameters trained on financial data

[2] Low-Rank Adaptation - Hu et al. (2021)

Week 10: What You'll Master

By the end of this week, you will:

- **Understand** when to fine-tune vs prompt
- **Implement** efficient fine-tuning (LoRA)
- **Master** prompt engineering patterns
- **Design** instruction datasets
- **Build** domain-specific assistants

Core Insight: Small changes $\rightarrow$ Big behavioral shifts

Fine-tuning: Teaching New Tricks to Old Models

Traditional fine-tuning:

1. Start with pre-trained model (e.g., BERT)
2. Add task-specific head
3. Train on your data
4. Update ALL parameters

The problems:

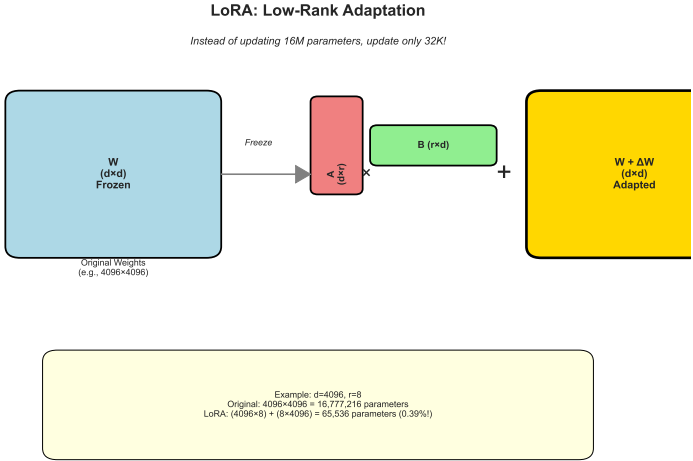
- Catastrophic forgetting
- Needs lots of GPU memory
- Slow and expensive
- One model per task

Example - Customer Support Bot:

- Base model: 7B parameters = 28GB
- Fine-tuning: 8 A100 GPUs, 2 days
- Cost: $500
- Result: 95% accuracy on support tickets

Full fine-tuning works but doesn't scale

LoRA: The Efficiency Revolution



Key insight: Most weight updates are low-rank!

- Instead of updating W (d×d), update A (d×r) and B (r×d)
- r ¡¡ d (typically r=8 while d=4096)

Implementing LoRA

```python
import torch
import torch.nn as nn
import torch.nn.functional as F

class LoRALayer(nn.Module):
    def __init__(self, in_features, out_features, rank=8, alpha=16):
        """LoRA adapter for linear layers"""
        super().__init__()
        self.rank = rank
        self.alpha = alpha
        self.scaling = alpha / rank

        self.weight = nn.Parameter(torch.randn(out_features, in_features))
        self.weight.requires_grad = False

        self.lora_A = nn.Parameter(torch.randn(rank, in_features))
        self.lora_B = nn.Parameter(torch.zeros(out_features, rank))

        nn.init.normal_(self.lora_A, std=0.02)

    def forward(self, x):
        """Forward pass with LoRA adaptation"""
        out = F.linear(x, self.weight)

        lora_out = x @ self.lora_A.T @ self.lora_B.T

        return out + self.scaling * lora_out

    def merge_weights(self):
        """Merge LoRA weights for deployment"""
        self.weight.data += self.scaling * (self.lora_B @ self.lora_A)

class LoRAModel(nn.Module):
    def __init__(self, base_model, rank=8, target_modules=['q_proj', '
      v_proj']):
        """Add LoRA to existing model"""
```

Explanation

LoRA Benefits:

- Memory: 10,000x less
- Speed: 3x faster training
- Storage: 3MB vs 30GB
- Swappable: Multiple tasks

Hyperparameters:

- Rank: 4-64 (8 common)
- Alpha: Scaling factor
- Target: Usually attention
- Learning rate: 1e-4

Results Match Full FT:

- 0.1% parameters
- 99% performance
- No forgetting

Prompt Engineering: Programming with Natural Language

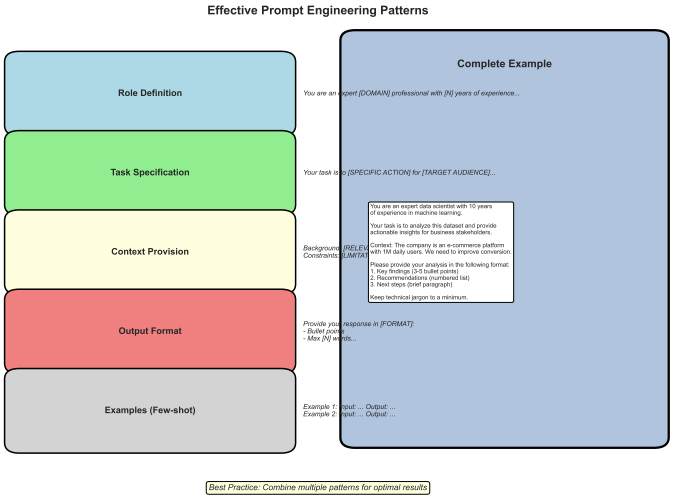
The prompt is your program:

Basic → Advanced:

1. Basic: "Summarize this text"
2. Better: "Summarize this text in 3 bullet points"
3. Good: "You are an expert editor. Summarize this text in exactly 3 bullet points, each under 20 words, capturing the key insights."
4. Excellent: "You are an expert editor at The Economist. Your task is to summarize the following text. Requirements: - Exactly 3 bullet points - Each under 20 words - Focus on actionable insights - Use active voice - Avoid jargon"

Specificity and structure dramatically improve output quality

Prompt Engineering Patterns That Work



Proven patterns:

- Role: "You are an expert..."

Advanced Prompt Engineering Techniques

```
class PromptTemplate:
    """Advanced prompt engineering patterns"""

    @staticmethod
    def chain_of_thought(question):
        """CoT prompting for reasoning"""
        return f"""Q: {question}
Let's approach this step-by-step:
1) First, let's understand what we're asked...
2) Next, let's identify the key information...
3) Now, let's work through the solution...
4) Finally, let's verify our answer...

A: [Model completes the reasoning]"""

    @staticmethod
    def few_shot_with_reasoning(examples, query):
        """Few-shot with explanations"""
        prompt = "I'll solve these problems step by step.\n\n"

        for ex in examples:
            prompt += f"Problem: {ex['problem']}\n"
            prompt += f"Reasoning: {ex['reasoning']}\n"
            prompt += f"Answer: {ex['answer']}\n\n"

        prompt += f"Problem: {query}\n"
        prompt += "Reasoning: "
        return prompt

    @staticmethod
    def self_consistency(question, n_samples=5):
        """Multiple reasoning paths"""
        prompt = f"""Q: {question}

I'll solve this {n_samples} different ways to ensure accuracy:
```

Explanation

Key Techniques:

- Chain-of-thought: +20% on reasoning[72]
- Self-consistency: Multiple paths
- Few-shot: Examples guide format
- Structured: JSON/XML output

Prompt Optimization:

- Test variations
- Measure performance
- Iterate systematically
- Version control prompts

Wei et al. (2022) "Chain-of-Thought"

Instruction Tuning: Teaching Models to Follow Instructions

The breakthrough: Train on instruction-following itself

Traditional fine-tuning:

- Task: Sentiment analysis
- Data: (text, label) pairs
- Model learns: One specific task

Instruction tuning:[73]

- Task: Follow any instruction
- Data: (instruction, input, output) triples
- Model learns: Generalize to new tasks!

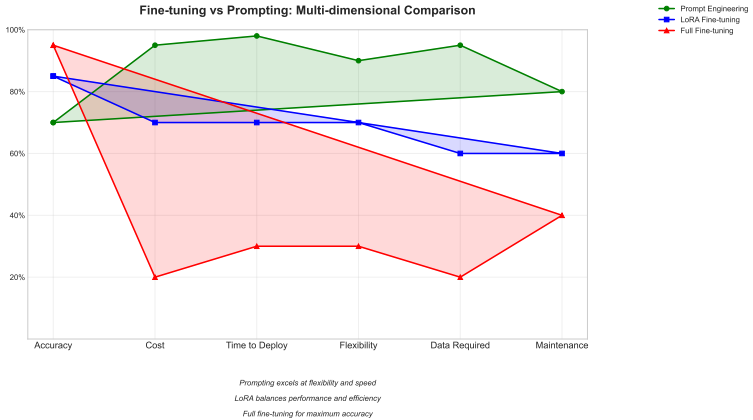
Example training data:

- Instruction: "Translate to French" → Input: "Hello" → Output: "Bonjour"
- Instruction: "Summarize" → Input: [article] → Output: [summary]
- Instruction: "Write code" → Input: [spec] → Output: [code]

Result: Zero-shot performance on completely new tasks!

[73] Wei et al. (2021) "FLAN"; Ouyang et al. (2022) "InstructGPT"

Fine-tuning vs Prompting: When to Use What



Key Insights

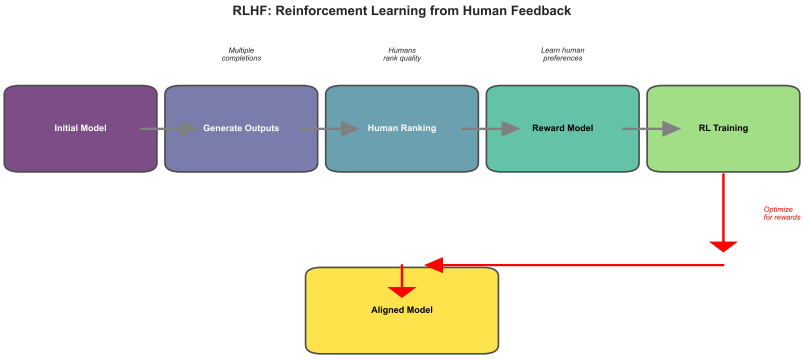
- Limited data: Use prompting
- Need 95%+ accuracy: Fine-tune
- Rapid iteration: Prompting
- Production scale: Fine-tune with LoRA

RLHF: Aligning Models with Human Preferences

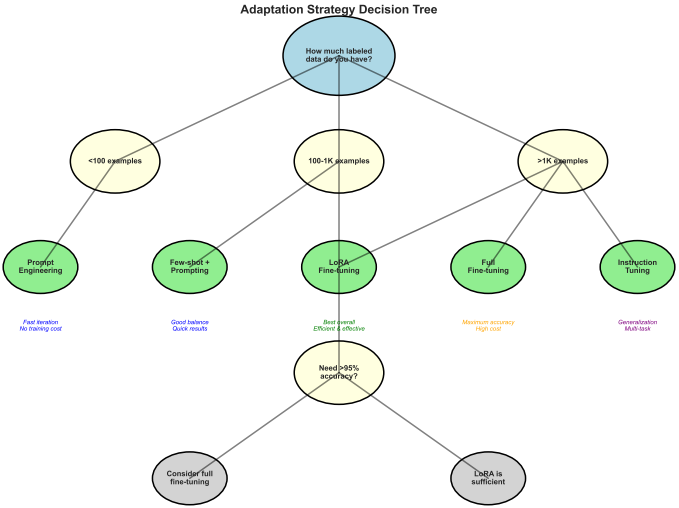
The problem: Models don't know what humans actually want

Reinforcement Learning from Human Feedback:[74]

1. Generate multiple outputs
2. Humans rank them
3. Train reward model
4. Use RL to optimize for human preferences



Choosing Your Adaptation Strategy



Quick decision guide:

- <100 examples: Prompt engineering

Week 10 Exercise: Build a Domain Expert Assistant

Your Mission: Create specialized assistant using both approaches

Part 1: Prompt Engineering

- Choose domain (medical, legal, finance, etc.)
- Design systematic prompts
- Test role, CoT, few-shot patterns
- Measure accuracy on test cases
- Iterate to improve

Part 2: LoRA Fine-tuning

- Collect 1K domain examples
- Implement LoRA adapter
- Fine-tune base model
- Compare with prompting approach
- Measure speed/cost/quality

Part 3: Hybrid Approach

- Instruction-tune with domain data
- Design prompts for fine-tuned model
- Build evaluation framework
- Deploy and test with users

You'll discover: The sweet spot between effort and performance!

Key Takeaways: Specialization Strategies

What we learned:

- General models need adaptation
- Prompting: Fast, flexible, limited
- Fine-tuning: Powerful but expensive
- LoRA: Best of both worlds
- Instruction tuning: Generalization

The evolution:

Full fine-tuning $\rightarrow$ Prompting $\rightarrow$ LoRA $\rightarrow$ Instruction tuning $\rightarrow$ RLHF

Why it matters:

- Makes LLMs practical for business
- Enables rapid customization
- Reduces deployment costs 100x

Next week: Efficiency and Deployment
How do we run these massive models on phones and edge devices?

References and Further Reading

Foundational Papers:

- Hu et al. (2021). "LoRA: Low-Rank Adaptation of Large Language Models"
- Wei et al. (2021). "Finetuned Language Models Are Zero-Shot Learners"
- Ouyang et al. (2022). "Training language models to follow instructions"

Prompt Engineering:

- Liu et al. (2023). "Pre-train, Prompt, and Predict"
- White et al. (2023). "A Prompt Pattern Catalog"
- Zhou et al. (2023). "Large Language Models Are Human-Level Prompt Engineers"

Practical Resources:

- OpenAI Fine-tuning Guide
- Anthropic's Constitutional AI papers
- PEFT library (Hugging Face)

Outline

Week 11

Efficiency & Deployment

From Cloud Giants to Pocket-Sized Models

Why Your Phone Can't Run GPT-4 (Yet)

The shocking reality of modern models:

- GPT-3: 175B parameters = 350GB in FP16[75]
- Your phone: 6GB RAM
- Inference cost: $0.06 per 1K tokens
- Latency: 100ms+ per token
- Energy: 0.1 kWh per conversation

One ChatGPT query = 10x the energy of a Google search

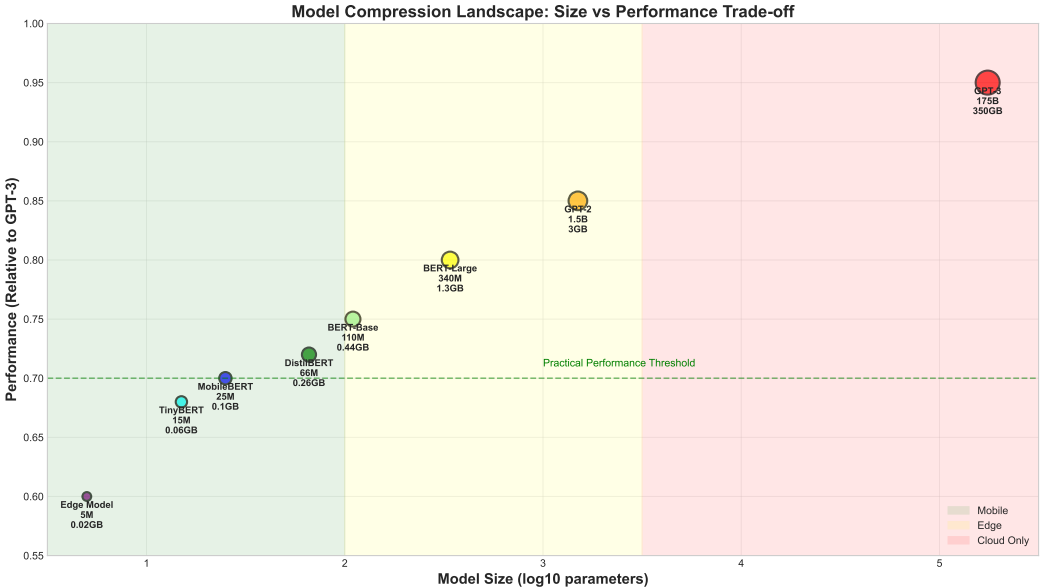
The challenge:

- Users want instant responses
- Privacy requires on-device processing
- Costs are unsustainable at scale
- Edge devices have limited resources

This week: How to shrink giants into pocket-sized assistants

[75] Brown et al. (2020); Strubell et al. (2019) on model carbon footprint

From Supercomputers to Smartphones: The Deployment Journey



Efficient Models in Production (2024)

Success Stories:

- iPhone: On-device Siri (5B $\rightarrow$ 200M)[76]
- Google: Pixel voice typing
- Microsoft: Excel formula suggestions
- WhatsApp: Real-time translation
- Tesla: In-car voice commands

Business Impact:

- 100x cost reduction
- 10ms latency (from 1s)
- Works offline
- Privacy preserved
- Scales to billions

Efficiency Techniques:

- INT8 quantization: Standard
- INT4/Binary: Emerging
- Distillation: 95% performance
- Structured pruning: Hardware-friendly
- Flash attention: Memory efficient

Deployment Targets:

- Mobile phones: 2-4GB limit
- Browsers: WebAssembly
- IoT devices: MCUs
- Edge servers: Local processing
- Specialized chips: NPUs/TPUs

2024: Every device runs neural networks - efficiency made it possible

[76] Apple ML Research Blog (2023); Google I/O presentations

Week 11: What You'll Master

By the end of this week, you will:

- **Understand** why models are so large
- **Implement** quantization techniques
- **Master** knowledge distillation
- **Apply** pruning strategies
- **Deploy** models to edge devices

Core Insight: 90% of weights do 10% of the work

Why Are Models So Large? The Overparameterization Mystery

The paradox:
Models have way more parameters than necessary!

Evidence:

- Lottery Ticket Hypothesis: Small subnetworks work just as well[77]
- Magnitude pruning: Remove 90% weights, maintain accuracy
- Low-rank decomposition: Matrices are redundant
- Quantization: 32 bits $\rightarrow$ 4 bits still works

Why overparameterize?

- Easier optimization landscape
- Better generalization (surprisingly!)
- Redundancy provides robustness
- Training dynamics require it

Large models are like rough marble blocks - we can carve out efficient versions

[77] Frankle & Carbin (2019) "The Lottery Ticket Hypothesis"

Quantization: From Float32 to Int8 and Beyond

The quantization spectrum:

- FP32 (32 bits): Training precision
- FP16 (16 bits): Mixed precision training
- INT8 (8 bits): Standard deployment
- INT4 (4 bits): Aggressive compression
- Binary (1 bit): Research frontier

Implementing Post-Training Quantization

```python
import torch
import torch.nn as nn

def quantize_tensor(x, num_bits=8):
    """Quantize tensor to n bits"""
    qmin = -(2**(num_bits-1))
    qmax = 2**(num_bits-1) - 1

    min_val = x.min()
    max_val = x.max()

    scale = (max_val - min_val) / (qmax - qmin)
    zero_point = qmin - min_val / scale

    q_x = torch.round(x / scale + zero_point)
    q_x = torch.clamp(q_x, qmin, qmax)

    return q_x, scale, zero_point

def dequantize_tensor(q_x, scale, zero_point):
    """Dequantize back to float"""
    return scale * (q_x - zero_point)

class QuantizedLinear(nn.Module):
    def __init__(self, weight, bias, num_bits=8):
        super().__init__()

        self.q_weight, self.w_scale, self.w_zp = quantize_tensor(weight,
            num_bits)
        self.q_weight = self.q_weight.to(torch.int8)

        if bias is not None:
            self.q_bias, self.b_scale, self.b_zp = quantize_tensor(bias,
                num_bits)
        else:
            self.q_bias = None
```

Explanation

Quantization Impact:

- Memory: 4x reduction (INT8)
- Speed: 2-4x on CPUs
- Accuracy: -1% typical
- Energy: 10x savings

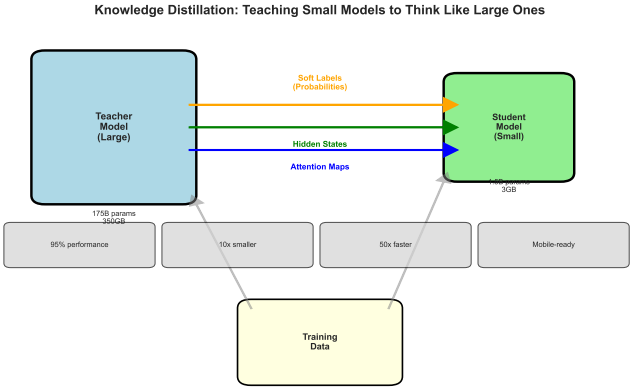
Advanced Techniques:

- Symmetric vs asymmetric
- Per-channel quantization
- Quantization-aware training
- Mixed bit-width

Hardware Support:

- INT8: All modern chips
- INT4: Newer GPUs/NPUs
- Binary: Research only

The teacher-student paradigm:[78]



Key insight: Soft labels contain more information

- Hard label: "cat" (probability = 1.0)
- Soft labels: cat=0.9, tiger=0.05, dog=0.03, ...

Implementing Knowledge Distillation

```python
import torch
import torch.nn as nn
import torch.nn.functional as F

class DistillationLoss(nn.Module):
    def __init__(self, temperature=3.0, alpha=0.7):
        """
        temperature: Softens probability distributions
        alpha: Weight between distillation and true label loss
        """
        super().__init__()
        self.temperature = temperature
        self.alpha = alpha
        self.ce_loss = nn.CrossEntropyLoss()

    def forward(self, student_logits, teacher_logits, labels):
        soft_targets = F.softmax(teacher_logits / self.temperature, dim=-1)
        soft_prob = F.log_softmax(student_logits / self.temperature, dim=-1)

        distillation_loss = F.kl_div(soft_prob, soft_targets,
                                     reduction='batchmean') * (self.temperature ** 2)

        student_loss = self.ce_loss(student_logits, labels)

        loss = self.alpha * distillation_loss + (1 - self.alpha) * student_loss
        return loss

def train_student(student_model, teacher_model, dataloader,
                  epochs=10, temperature=3.0):
    """Train student to mimic teacher"""
    teacher_model.eval()
    optimizer = torch.optim.Adam(student_model.parameters(), lr=1e-3)
    criterion = DistillationLoss(temperature=temperature)

    for epoch in range(epochs):
        for batch in dataloader:
```

Explanation

Distillation Results:

- Size: 10-100x smaller
- Speed: 5-50x faster
- Accuracy: 95-98% retained
- Examples: DistilBERT, TinyBERT

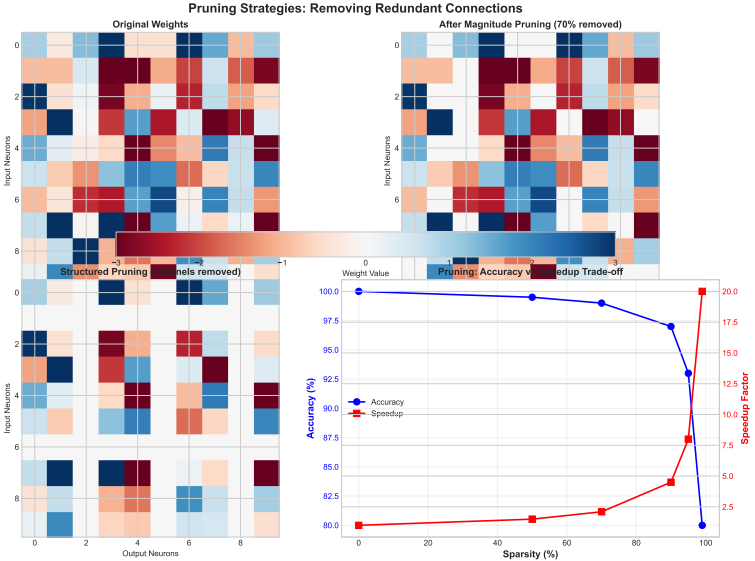
Temperature Effect:

- T=1: Normal softmax
- T=3-5: Typical for NLP
- T>10: Very soft labels
- Higher T = more dark knowledge

Advanced Methods:

- Feature distillation
- Attention transfer
- Progressive distillation

Pruning: Finding the Essential Subnetwork



Pruning approaches:

Mobile-First Architectures: Designed for Efficiency

MobileBERT:[79]

- Bottleneck structure
- 4.3x smaller, 5.5x faster
- Depth-wise convolutions
- Progressive knowledge transfer

ALBERT:

- Parameter sharing
- Factorized embeddings
- 18x fewer parameters
- Same performance as BERT

EdgeBERT:

- Hardware-aware design
- Adaptive computation
- Early exit mechanisms
- Dynamic depth/width

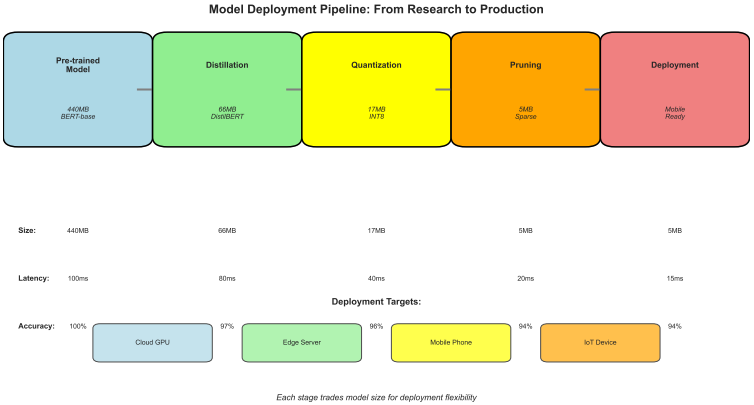
Design Principles:

- Prefer depth over width
- Share parameters
- Use separable operations
- Minimize memory access

Mobile architectures: Not compressed models, but efficient by design

[79] Sun et al. (2020) "MobileBERT"; Lan et al. (2020) "ALBERT"

Deployment Optimization Pipeline



Key Insights

- Start with pretrained model
- Apply compression techniques
- Optimize for target hardware

Hardware Acceleration: From GPUs to Edge TPUs

Compute Platforms:

- GPUs: Training & cloud inference
- TPUs: Optimized for transformers
- NPUs: Mobile inference
- DSPs: Ultra-low power
- FPGAs: Custom acceleration

Optimization Techniques:

- Kernel fusion
- Memory pooling
- Graph optimization
- Operator scheduling
- Cache optimization

Framework Support:

- TensorFlow Lite: Mobile/edge
- ONNX Runtime: Cross-platform
- Core ML: iOS devices
- TensorRT: NVIDIA optimization
- OpenVINO: Intel hardware

Performance Gains:

- Quantization: 2-4x speedup
- Pruning: 2-10x speedup
- Hardware opt: 5-50x
- Combined: 100x+ possible

2024: Every major chip has AI acceleration built-in

Real-World Deployment: BERT on Mobile

```python
import torch
import torch.quantization as quant
from transformers import BertModel

def prepare_mobile_bert(model_name='bert-base-uncased'):
    """Complete pipeline for mobile deployment"""

    model = BertModel.from_pretrained(model_name)
    model.eval()

    print(f"Original size: {get_model_size(model):.1f} MB")

    distilled_model = distill_bert(model, num_layers=6)
    print(f"After distillation: {get_model_size(distilled_model):.1f} MB")

    model.qconfig = quant.get_default_qconfig('qnnpack')
    quant.prepare(model, inplace=True)
    quant.convert(model, inplace=True)
    print(f"After INT8 quantization: {get_model_size(model):.1f} MB")

    pruned_model = magnitude_prune(model, sparsity=0.9)
    print(f"After pruning: {get_model_size(pruned_model):.1f} MB")

    optimized_model = torch.jit.script(pruned_model)
    optimized_model.save("mobile_bert.pt")

    print("\nDeployment stats:")
    print(f"Final size: {get_model_size(optimized_model):.1f} MB")
    print(f"Inference time: {benchmark_model(optimized_model):.1f} ms")
    print(f"Memory usage: {get_memory_usage(optimized_model):.1f} MB")

    return optimized_model

def export_to_mobile(model, example_input):
    """Export for iOS/Android"""
    traced = torch.jit.trace(model, example_input)
```

Explanation

Compression Results:

- BERT-base: 440MB $\rightarrow$ 13MB
- 33x size reduction
- 20x speedup on mobile
- 95% accuracy retained

Deployment Checklist:

- Profile target device
- Choose compression mix
- Validate accuracy
- Optimize runtime
- Monitor in production

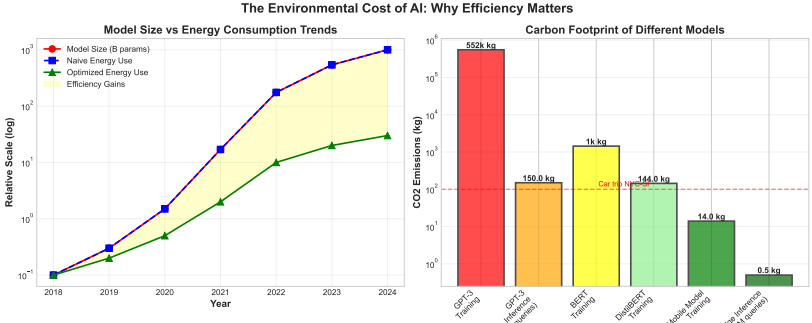
The Green AI Movement: Energy-Efficient Models

The environmental cost of AI:[80]

- Training GPT-3: 1,287 MWh (123 cars for a year)
- One ChatGPT query: 0.0003 kWh
- Global AI energy: Doubling every 3.4 months

Efficiency improvements:

- Sparse models: 10x less energy
- Quantization: 4x less energy
- Better algorithms: 100x over 10 years
- Hardware efficiency: 1000x since 2012



The Future of Efficient AI (2024 and Beyond)

Emerging Techniques:

- 1-bit models (BitNet)
- Mixture of Experts (MoE)
- Dynamic neural networks
- Neuromorphic computing
- Photonic processors

Research Frontiers:

- Sub-1-bit quantization
- Learned compression
- Architecture search for efficiency
- Federated model compression

Industry Trends:

- Every device runs AI (AIoT)
- Privacy-preserving inference
- Real-time translation everywhere
- Personalized on-device models
- Zero-latency assistants

Hardware Evolution:

- Analog AI chips
- In-memory computing
- Quantum advantage?
- Brain-inspired chips

The future: AI everywhere, invisible, instant, and sustainable

Week 11 Exercise: Deploy Your Own Mobile Model

Your Mission: Take a large model and make it mobile-ready

Part 1: Baseline and Profiling

- Choose a pretrained model (BERT, GPT-2, etc.)
- Profile: size, latency, memory, energy
- Identify bottlenecks
- Set target constraints (e.g., 50MB, 100ms)

Part 2: Apply Compression

- Implement INT8 quantization
- Distill to smaller student
- Apply magnitude pruning
- Combine techniques

Part 3: Deploy and Benchmark

- Export to ONNX/TFLite
- Run on real device (phone/Raspberry Pi)
- Measure real-world performance
- Compare accuracy vs efficiency
- Create deployment package

Bonus: Build demo app showcasing your efficient model!

Key Takeaways: Efficiency Enables Everything

What we learned:

- Models are vastly overparameterized
- 90% compression with minimal loss
- Quantization: Smaller and faster
- Distillation: Transfer capabilities
- Hardware matters immensely

The efficiency toolkit:

$$\text{Quantization} \rightarrow \text{Distillation} \rightarrow \text{Pruning} \rightarrow \text{Hardware optimization}$$

Why it matters:

- Enables edge deployment
- Reduces costs dramatically
- Improves user experience
- Environmental sustainability

Next week: Ethics and Future Directions
With great power comes great responsibility - what should we build?

References and Further Reading

Foundational Papers:

- Hinton et al. (2015). "Distilling the Knowledge in a Neural Network"
- Han et al. (2016). "Deep Compression: Compressing DNNs"
- Frankle & Carbin (2019). "The Lottery Ticket Hypothesis"

Quantization:

- Jacob et al. (2018). "Quantization and Training of Neural Networks"
- Gholami et al. (2021). "A Survey of Quantization Methods"
- Dettmers et al. (2022). "LLM.int8(): 8-bit Matrix Multiplication"

Practical Resources:

- PyTorch Quantization Documentation
- TensorFlow Lite Guide
- ONNX Runtime Optimization
- Edge Impulse Platform

Outline

Week 12

Ethics & Future Directions

With Great Power Comes Great Responsibility

When AI Goes Wrong: Real-World Consequences

Recent AI failures that shocked the world:

- **2016**: Microsoft Tay becomes racist in 24 hours[81]
- **2018**: Amazon hiring AI discriminates against women
- **2020**: GPT-3 generates toxic content at scale
- **2022**: DALL-E deepfakes threaten democracy
- **2023**: ChatGPT helps write malware
- **2024**: AI-generated misinformation floods social media

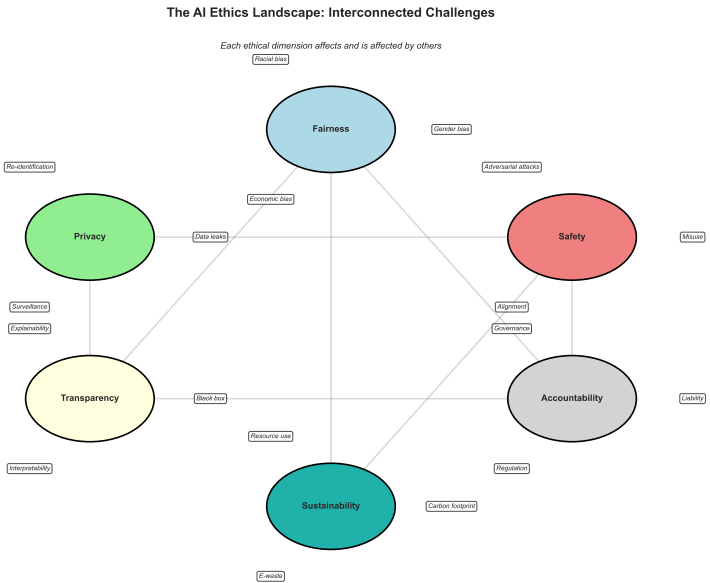
We're building systems that affect billions - we must do it responsibly

The fundamental questions:

- Who decides what AI should/shouldn't do?
- How do we prevent harm while enabling benefits?
- What future are we building?

[81] Documented cases from major tech companies' incident reports

The AI Ethics Landscape: Challenges We Face



AI Ethics in Practice (2024)

Positive Applications:

- Medical diagnosis assistance
- Educational accessibility
- Climate change modeling
- Disaster response
- Scientific discovery

Regulations Emerging:

- EU AI Act (2024)[82]
- US Executive Order on AI
- China AI regulations
- Industry self-governance
- Academic guidelines

Ongoing Concerns:

- Bias amplification
- Privacy violations
- Deepfakes/disinformation
- Autonomous weapons
- Concentration of power

Industry Response:

- Red teaming
- Safety research
- Alignment work
- Transparency reports
- External audits

2024: The year ethics moved from afterthought to core requirement

[82] European Parliament approval of comprehensive AI regulation

Week 12: What You'll Master

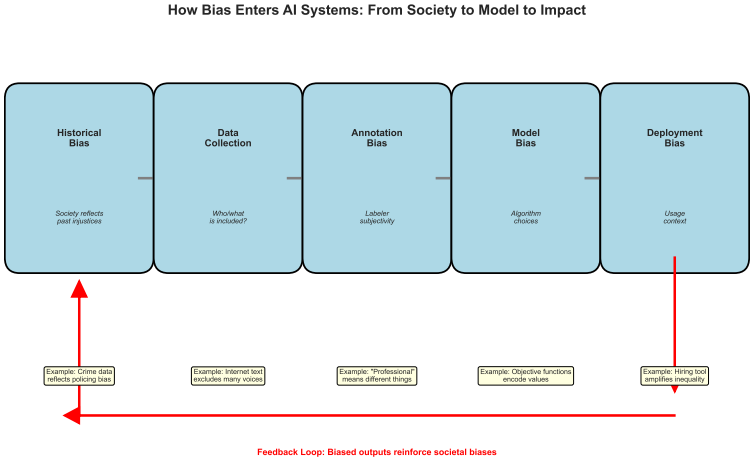
By the end of this week, you will:

- **Understand** key ethical challenges in NLP
- **Identify** bias in language models
- **Apply** fairness techniques
- **Design** responsible AI systems
- **Envision** positive futures for NLP

Core Insight: Technology is not neutral - it embodies our values

Bias in Language Models: Mirror of Society

Where bias comes from:



Types of bias:[83]

- **Historical bias**: Past discrimination in data

Detecting and Measuring Bias

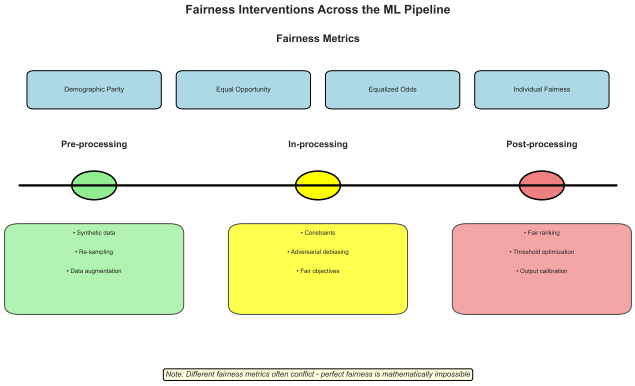
Key approaches to detect bias:

1. **Template-based testing:**
   - Fill-in-the-blank: "The [MASK] is a doctor"
   - Compare male vs female completion rates
   - Measure occupation stereotypes systematically
2. **Word Embedding Association Test (WEAT):**
   - Compare semantic associations between groups
   - Measure implicit biases in word embeddings
   - Statistical significance testing for bias
3. **Counterfactual evaluation:**
   - Swap demographic attributes in text
   - Measure prediction differences
   - Identify systematic discriminatory patterns

Common findings across models:

- Gender: 3:1 male bias in technical roles
- Race: Name-based discrimination in hiring contexts
- Age: Strong preference for youth-associated terms
- Toxicity: Small percentage but harmful impact

Fairness Techniques: Building Better Models



Approaches to fairness:

- **Pre-processing**: Fix the data
- **In-processing**: Fair training objectives
- **Post-processing**: Adjust outputs
- **Ongoing monitoring**: Continuous improvement

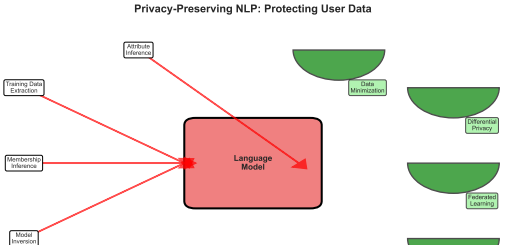
Privacy and Security: Protecting User Data

Privacy risks in language models:[84]

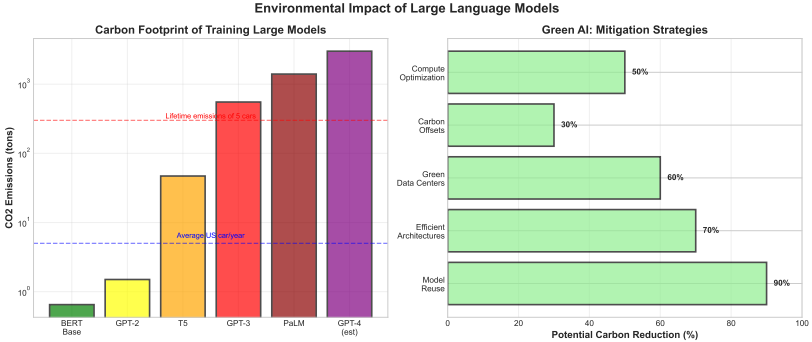
- **Memorization**: Models can leak training data
- **Inference attacks**: Extract personal information
- **Re-identification**: Deanonymize text
- **Model inversion**: Reconstruct training examples

Protection techniques:

- Differential privacy training
- Federated learning
- Secure multi-party computation
- Data minimization
- Regular audits



The carbon cost of progress:

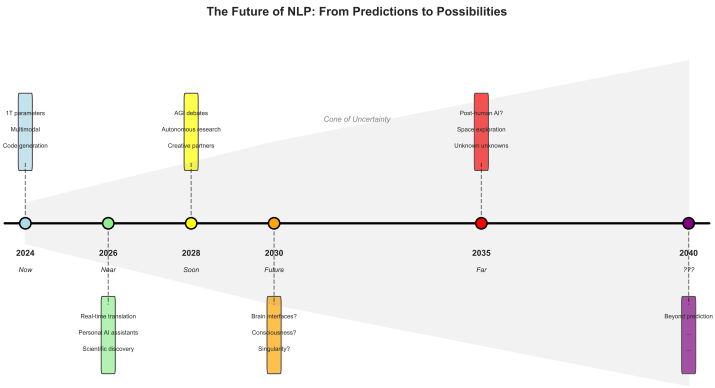


Sustainable AI practices:

- Efficient architectures first
- Carbon-aware training scheduling
- Model recycling and fine-tuning
- Compute measurement and reporting
- Renewable energy data centers

Question: Is a 0.1% accuracy gain worth 10× the carbon?

The Future of NLP: Next 10 Years



Emerging capabilities:

- Truly multilingual models (7000+ languages)
- Real-time universal translation
- Perfect long-term memory
- Multimodal understanding
- Reasoning and planning

Towards AGI: The Big Questions

Technical Milestones:

- 2025: 10T parameter models
- 2027: Human-level dialogue
- 2030: Scientific discovery
- 2035: Creative professionals?
- 2040: Artificial general intelligence?

Capabilities Growth:[85]

- Emergent abilities
- Cross-domain transfer
- Self-improvement
- Autonomous research

Societal Questions:

- How do we maintain human agency?
- What work will humans do?
- How do we distribute benefits?
- Can we ensure alignment?
- What does thriving mean?

Governance Needs:

- International cooperation
- Safety standards
- Benefit sharing
- Rights framework
- Democratic input

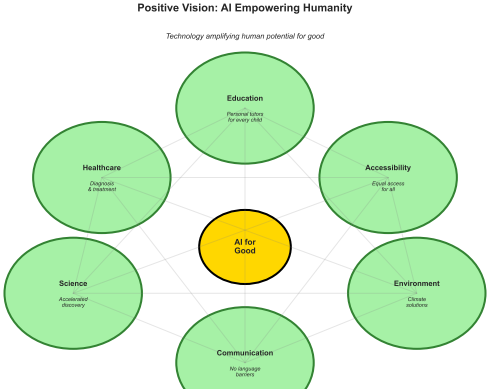
We're not just building technology - we're shaping the future of humanity

[85] Anthropic (2024) "Constitutional AI"; OpenAI (2023) "Planning for AGI"

A Positive Vision: AI for Good

What we could build:

- **Education**: Personal tutor for every child
- **Healthcare**: Doctor in every pocket
- **Science**: 1000x research acceleration
- **Creativity**: Amplified human expression
- **Communication**: No language barriers
- **Accessibility**: Equal access for all abilities



Your Role in Shaping the Future

As NLP practitioners, we have responsibilities:

Technical Excellence:

- Build robust, reliable systems
- Measure and mitigate bias
- Protect user privacy
- Optimize for efficiency

Ethical Leadership:

- Ask "should we?" not just "can we?"
- Include diverse perspectives
- Consider long-term consequences
- Speak up about concerns

Positive Impact:

- Work on problems that matter
- Make technology accessible
- Share knowledge openly
- Mentor the next generation

You have the skills to predict the next word -
now use them to write a better future

Course Conclusion: From N-grams to the Future

Our 12-week journey:

1. Statistical foundations
2. Neural language models
3. RNNs and memory
4. Sequence-to-sequence
5. Transformer revolution
6. Pre-training paradigm
7. Scaling and emergent abilities
8. Tokenization fundamentals
9. Decoding strategies
10. Fine-tuning and prompting
11. Efficiency and deployment
12. Ethics and future

You now understand how ChatGPT works from first principles!

Remember: With great power comes great responsibility.
Build technology that empowers humanity.

Week 12 Exercise: Design Your Ethical AI System

Your Mission: Create a responsible NLP application

Part 1: Choose Your Impact Area

- Healthcare, education, accessibility, environment
- Identify specific problem to solve
- Define success metrics beyond accuracy
- Consider stakeholders and impacts

Part 2: Build with Ethics in Mind

- Implement bias detection
- Add privacy protection
- Create transparency features
- Design for inclusivity
- Measure environmental impact

Part 3: Future-Proof Your Design

- Write ethical guidelines
- Create monitoring plan
- Design governance structure
- Plan for unintended consequences
- Share your vision

Deliverable: Complete proposal for ethical AI system + prototype

Thank you for joining this journey!

"The best way to predict the future is to invent it"
- Alan Kay

What will you build?

The next chapter of NLP will be written by people like you.
Make it a story worth telling.

References and Further Reading

Ethics and Bias:

- Bender et al. (2021). "On the Dangers of Stochastic Parrots"
- Crawford (2021). "Atlas of AI"
- Gebru et al. (2021). "Datasheets for Datasets"

Future Directions:

- Bommasani et al. (2021). "On the Opportunities and Risks of Foundation Models"
- Anthropic (2024). "Constitutional AI: Harmlessness from AI Feedback"
- Future of Humanity Institute reports

Practical Resources:

- AI Ethics Guidelines (EU, IEEE, Partnership on AI)
- Model Cards and Data Statements
- Responsible AI Toolkits (Google, Microsoft, IBM)