Generative AI - Advanced Handout

Machine Learning for Smarter Innovation

1 Generative AI - Advanced Handout

Target Audience: Data scientists and ML engineers **Duration**: 90 minutes reading **Level**: Advanced (mathematical foundations and production systems)

1.1 Mathematical Foundations

1.1.1 Autoregressive Language Models

Language models estimate probability distributions over token sequences. Given sequence x = (x_1, . . . , x_n), the joint probability factorizes autoregressively:

P(x) = prod_{t=1}^{n} P(x_t | x_1, . . . , x_{t-1})

Each conditional P(x_t | x_{<t}) is modeled by a neural network with parameters theta:

P_theta(x_t | x_{<t}) = softmax(W h_t + b)

where h_t is the hidden state encoding context x_{<t}.

1.1.2 Cross-Entropy Training Objective

Training minimizes negative log-likelihood over corpus D:

L(theta) = -E_{x ~ D}[log P_theta(x)] = -E_{x ~ D}[sum_{t=1}^{n} log P_theta(x_t | x_{<t})]

For one-hot target distribution q (q_v = 1 for correct token v), this equals cross-entropy:

H(q, p) = -sum_v q_v log p_v = -log p_v

Perplexity measures model quality:

PPL = exp(L(theta)) = exp(-1/N sum_{i=1}^{N} log P(x_i))

Lower perplexity indicates better prediction. State-of-art LLMs achieve PPL $<$ 10 on standard benchmarks.

1.1.3 Transformer Architecture

The transformer processes sequences through self-attention and feed-forward layers.

Self-Attention:

Attention(Q, K, V) = softmax(QK^T / sqrt(d_k)) V

For input X in R^{n x d}, we compute: - Q = X W_Q (queries) - K = X W_K (keys) - V = X W_V (values)

The attention weights alpha_{ij} = softmax(q_i . k_j / sqrt(d_k)) define how position i attends to position j.

Causal Masking: For autoregressive generation, mask future positions:

alpha_{ij} = 0 if j > i

This ensures P(x_t) depends only on x_{<t}.

Multi-Head Attention:

MultiHead(Q, K, V) = Concat(head_1, ..., head_h) W_O head_i = Attention(X W_Qˆi, X W_Kˆi, X W_Vˆi)

Multiple heads capture different relationship types. GPT-3 uses h=96 heads with d_k=128 per head.

Feed-Forward Network:

FFN(x) = GELU(x W_1 + b_1) W_2 + b_2

with inner dimension 4 * d_model. GELU activation:

GELU(x) = x * Phi(x) approx 0.5 * x * (1 + tanh(sqrt(2/pi) * (x + 0.044715 * xˆ3)))

1.1.4 Positional Encoding

Transformers lack inherent position information. Options:

Sinusoidal (original): PE(pos, 2i) = sin(pos / 10000ˆ{2i/d}) PE(pos, 2i+1) = cos(pos / 10000ˆ{2i/d})

Rotary Position Embedding (RoPE): Rotate query/key vectors based on position: R_theta,m = [[cos(m*theta), -sin(m*theta)], [sin(m*theta), cos(m*theta)]]

RoPE enables better extrapolation to longer sequences than seen during training.

1.2 Scaling Laws

1.2.1 Compute-Optimal Training

The Chinchilla scaling law relates model size N, data tokens D, and compute C:

L(N, D) = E + A/Nˆalpha + B/Dˆbeta

where E is irreducible loss, alpha approx 0.34, beta approx 0.28.

For fixed compute budget C proportional to N * D: - Optimal model size: N* proportional to Cˆ{0.5} - Optimal data: D* proportional to Cˆ{0.5}

This implies ~20 tokens per parameter for compute-optimal training. GPT-3 (175B params, 300B tokens) was undertrained; Chinchilla (70B params, 1.4T tokens) achieves similar quality with less compute.

1.2.2 Emergent Capabilities

Capabilities emerge suddenly at scale thresholds: - ~10B params: few-shot learning - ~100B params: chain-of-thought reasoning - ~500B params: complex multi-step reasoning

The emergence is not gradual: performance jumps from near-random to near-perfect over small parameter increases.

1.3 Reinforcement Learning from Human Feedback

1.3.1 Reward Modeling

Train reward model r_phi(x, y) predicting human preferences. Given comparison (y_w, y_l) where y_w is preferred:

L(phi) = -E_{(x,y_w,y_l)}[log sigmoid(r_phi(x, y_w) - r_phi(x, y_l))]
This Bradley-Terry model assumes:
P(y_w > y_l | x) = sigmoid(r(x, y_w) - r(x, y_l))

1.3.2 PPO for Language Models

Proximal Policy Optimization fine-tunes the policy pi_theta to maximize expected reward:
max_theta E_{x~D, y~pi_theta}[r_phi(x, y) - beta * KL(pi_theta || pi_ref)]
The KL penalty prevents divergence from reference policy pi_ref (the supervised fine-tuned model).
PPO objective:
L_PPO(theta) = E[min(rho_t * A_t, clip(rho_t, 1-epsilon, 1+epsilon) * A_t)]
where rho_t = pi_theta(a_t|s_t) / pi_{old}(a_t|s_t) is probability ratio, A_t is advantage estimate.

1.3.3 Direct Preference Optimization (DPO)

DPO eliminates explicit reward modeling. The optimal policy under RLHF objective:
pi*(y|x) proportional to pi_ref(y|x)* exp(r(x,y) / beta)
This implies:
r(x, y) = beta * log(pi*(y|x) / pi_ref(y|x)) + beta* log Z(x)
Substituting into Bradley-Terry:
L_DPO(theta) = -E[log sigmoid(beta * log(pi_theta(y_w|x)/pi_ref(y_w|x)) - beta * log(pi_theta(y_l|x)/pi_ref(y_l|x))
DPO trains directly on preference pairs without reward model, simplifying the pipeline.

1.4 Decoding Strategies

1.4.1 Temperature Sampling

Temperature T controls distribution sharpness:
P(x_t | x_{<t}) = softmax(logits / T)

- T -> 0: greedy decoding (argmax)
- T = 1: standard sampling
- T > 1: more random, creative outputs

1.4.2 Nucleus Sampling (Top-p)

Sample from smallest set V_p where cumulative probability exceeds p:
V_p = argmin_{V: sum_{v in V} P(v) >= p} |V|
Renormalize within V_p:
P'(v) = P(v) / sum_{v' in V_p} P(v') if v in V_p else 0
Top-p = 0.9 typically balances quality and diversity.

1.4.3 Top-k Sampling

Restrict to k highest-probability tokens:

P'(v) = P(v) / sum_{i=1}^{k} P(v_i) if v in top-k else 0

Combines with temperature: sample from top-k with adjusted temperature.

1.4.4 Beam Search

Maintain b candidate sequences, expanding each by top-k tokens:

score(y_{1:t}) = sum_{i=1}^{t} log P(y_i | y_{<i})

Length normalization prevents preference for short sequences:

score_norm = score(y) / |y|^alpha

with alpha in [0.6, 0.8]. Beam search produces high-likelihood but potentially repetitive outputs.

1.5 Low-Rank Adaptation (LoRA)

1.5.1 Mathematical Foundation

Full fine-tuning updates weight matrix W in R^{d x k}. LoRA decomposes the update:

W' = W + Delta W = W + B A

where B in R^{d x r} and A in R^{r x k} with r « min(d, k).

Trainable parameters: $dr + rk$ « d*k (e.g., r=8 with d=4096 gives 99.8% reduction).

1.5.2 Rank Selection

The optimal rank depends on task complexity: - r = 4-8: style transfer, simple tasks - r = 16-32: domain adaptation - r = 64-128: significant capability changes

Intrinsic dimensionality of fine-tuning updates is often low, justifying LoRA's effectiveness.

1.5.3 Training Dynamics

Forward pass with LoRA:

h = W x + Delta W x = W x + B A x

Initialize A from N(0, sigma^2), B = 0, so Delta W = 0 initially. Scaling factor alpha/r controls learning rate effective magnitude.

1.6 Implementation

1.6.1 Setup

```
import torch
import torch.nn as nn
import torch.nn.functional as F
from transformers import (
    AutoModelForCausalLM, AutoTokenizer,
    TrainingArguments, Trainer
)
from peft import LoraConfig, get_peft_model, TaskType
import numpy as np
from typing import List, Dict, Optional
```

```
import warnings
warnings.filterwarnings('ignore')

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
```

1.6.2 Custom Transformer Block

```
class TransformerBlock(nn.Module):
    """Single transformer block with causal attention."""

    def __init__(self, d_model, n_heads, d_ff, dropout=0.1):
        super().__init__()
        self.d_model = d_model
        self.n_heads = n_heads
        self.d_k = d_model // n_heads

        # Multi-head attention projections
        self.w_q = nn.Linear(d_model, d_model)
        self.w_k = nn.Linear(d_model, d_model)
        self.w_v = nn.Linear(d_model, d_model)
        self.w_o = nn.Linear(d_model, d_model)

        # Feed-forward network
        self.ffn = nn.Sequential(
            nn.Linear(d_model, d_ff),
            nn.GELU(),
            nn.Dropout(dropout),
            nn.Linear(d_ff, d_model),
            nn.Dropout(dropout)
        )

        # Layer normalization
        self.ln1 = nn.LayerNorm(d_model)
        self.ln2 = nn.LayerNorm(d_model)
        self.dropout = nn.Dropout(dropout)

    def forward(self, x, mask=None):
        # Multi-head self-attention with residual
        attn_out = self._multihead_attention(self.ln1(x), mask)
        x = x + self.dropout(attn_out)

        # Feed-forward with residual
        ffn_out = self.ffn(self.ln2(x))
        x = x + ffn_out

        return x

    def _multihead_attention(self, x, mask):
        batch_size, seq_len, _ = x.shape

        # Project to Q, K, V
        Q = self.w_q(x).view(batch_size, seq_len, self.n_heads, self.d_k).
    transpose(1, 2)
        K = self.w_k(x).view(batch_size, seq_len, self.n_heads, self.d_k).
    transpose(1, 2)
        V = self.w_v(x).view(batch_size, seq_len, self.n_heads, self.d_k).
    transpose(1, 2)

        # Scaled dot-product attention
        scores = torch.matmul(Q, K.transpose(-2, -1)) / np.sqrt(self.d_k)
```

```
        if mask is not None:
            scores = scores.masked_fill(mask == 0, -1e9)

        attn_weights = F.softmax(scores, dim=-1)
        attn_output = torch.matmul(attn_weights, V)

        # Concatenate heads
        attn_output = attn_output.transpose(1, 2).contiguous().view(
            batch_size, seq_len, self.d_model
        )

        return self.w_o(attn_output)
```

1.6.3 LoRA Implementation

```
class LoRALinear(nn.Module):
    """Linear layer with LoRA adaptation."""

    def __init__(self, in_features, out_features, rank=8, alpha=16):
        super().__init__()
        self.in_features = in_features
        self.out_features = out_features
        self.rank = rank
        self.alpha = alpha
        self.scaling = alpha / rank

        # Original frozen weights
        self.weight = nn.Parameter(torch.randn(out_features, in_features))
        self.weight.requires_grad = False

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

        # Initialize A with small values
        nn.init.kaiming_uniform_(self.lora_A, a=np.sqrt(5))

    def forward(self, x):
        # Original forward
        base_output = F.linear(x, self.weight)

        # LoRA adaptation
        lora_output = F.linear(F.linear(x, self.lora_A), self.lora_B)

        return base_output + self.scaling * lora_output

    def merge_weights(self):
        """Merge LoRA weights into base weights."""
        self.weight.data += self.scaling * torch.matmul(self.lora_B, self.
    lora_A)
        self.lora_A.data.zero_()
        self.lora_B.data.zero_()
```

1.6.4 Production Generation Pipeline

```
class ProductionLLM:
    """Production-ready LLM with caching and monitoring."""
```

```
    def __init__(self, model_name, device='cuda'):
        self.tokenizer = AutoTokenizer.from_pretrained(model_name)
        self.model = AutoModelForCausalLM.from_pretrained(
            model_name, torch_dtype=torch.float16, device_map='auto'
        )
        self.model.eval()
        self.device = device

        # KV cache for efficient generation
        self.kv_cache = {}

        # Metrics
        self.total_tokens = 0
        self.total_time = 0

    @torch.no_grad()
    def generate(self, prompt, max_length=256, temperature=0.7,
                 top_p=0.9, top_k=50):
        """Generate with nucleus sampling."""
        input_ids = self.tokenizer.encode(prompt, return_tensors='pt').to(self
    .device)

        import time
        start = time.time()

        generated = []
        for _ in range(max_length):
            outputs = self.model(input_ids, use_cache=True)
            logits = outputs.logits[:, -1, :]

            # Apply temperature
            logits = logits / temperature

            # 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
            sorted_logits, sorted_indices = torch.sort(logits, descending=True
    )
            cumulative_probs = torch.cumsum(F.softmax(sorted_logits, dim=-1),
    dim=-1)
            sorted_indices_to_remove = cumulative_probs > top_p
            sorted_indices_to_remove[..., 1:] = sorted_indices_to_remove[...,
    :-1].clone()
            sorted_indices_to_remove[..., 0] = 0
            indices_to_remove = sorted_indices_to_remove.scatter(
                1, sorted_indices, sorted_indices_to_remove
            )
            logits[indices_to_remove] = -float('inf')

            # Sample
            probs = F.softmax(logits, dim=-1)
            next_token = torch.multinomial(probs, num_samples=1)

            if next_token.item() == self.tokenizer.eos_token_id:
                break

            generated.append(next_token.item())
            input_ids = torch.cat([input_ids, next_token], dim=-1)
```

```
        self.total_time += time.time() - start
        self.total_tokens += len(generated)

        return self.tokenizer.decode(generated)

    def get_metrics(self):
        """Return generation metrics."""
        return {
            'total_tokens': self.total_tokens,
            'total_time': self.total_time,
            'tokens_per_second': self.total_tokens / max(self.total_time, 1e
    -6)
        }
```

1.6.5 Fine-tuning with LoRA

```
class LoRATrainer:
    """Fine-tune LLM with LoRA."""

    def __init__(self, model_name, lora_rank=8, lora_alpha=16):
        self.model_name = model_name
        self.lora_rank = lora_rank
        self.lora_alpha = lora_alpha

    def prepare_model(self):
        """Load model and apply LoRA."""
        model = AutoModelForCausalLM.from_pretrained(
            self.model_name, torch_dtype=torch.float16
        )

        lora_config = LoraConfig(
            r=self.lora_rank,
            lora_alpha=self.lora_alpha,
            target_modules=['q_proj', 'v_proj', 'k_proj', 'o_proj'],
            lora_dropout=0.05,
            bias='none',
            task_type=TaskType.CAUSAL_LM
        )

        model = get_peft_model(model, lora_config)
        model.print_trainable_parameters()

        return model

    def train(self, model, train_dataset, output_dir, epochs=3):
        """Train with LoRA."""
        training_args = TrainingArguments(
            output_dir=output_dir,
            num_train_epochs=epochs,
            per_device_train_batch_size=4,
            gradient_accumulation_steps=4,
            learning_rate=2e-4,
            fp16=True,
            logging_steps=10,
            save_strategy='epoch',
            warmup_ratio=0.1,
            optim='adamw_torch'
        )

        trainer = Trainer(
            model=model,
```

```
            args=training_args,
            train_dataset=train_dataset
        )

        trainer.train()
        return model
```

1.6.6 RAG System

```
class RAGSystem:
    """Retrieval-Augmented Generation system."""

    def __init__(self, llm, retriever, n_docs=5):
        self.llm = llm
        self.retriever = retriever
        self.n_docs = n_docs

    def query(self, question):
        """Answer question using retrieved context."""
        # Retrieve relevant documents
        docs = self.retriever.search(question, top_k=self.n_docs)

        # Format context
        context = '\n\n'.join([doc['text'] for doc in docs])

        # Construct prompt
        prompt = f"""Answer the question based on the context below.

Context:
{context}

Question: {question}

Answer:"""

        # Generate
        answer = self.llm.generate(prompt, max_length=256, temperature=0.3)

        return {
            'answer': answer,
            'sources': [doc['source'] for doc in docs],
            'context_used': len(docs)
        }


class VectorRetriever:
    """Simple vector similarity retriever."""

    def __init__(self, embeddings, documents):
        self.embeddings = embeddings  # [n_docs, dim]
        self.documents = documents

    def search(self, query, top_k=5):
        """Search by vector similarity."""
        query_embedding = self._embed(query)

        # Cosine similarity
        similarities = np.dot(self.embeddings, query_embedding)
        similarities /= np.linalg.norm(self.embeddings, axis=1)
        similarities /= np.linalg.norm(query_embedding)
```

```
        # Top-k indices
        top_indices = np.argsort(similarities)[-top_k:][::-1]

        return [self.documents[i] for i in top_indices]

    def _embed(self, text):
        """Embed text (placeholder for actual embedding model)."""
        # In production, use SentenceTransformer or similar
        return np.random.randn(768)
```

1.7 Evaluation Metrics

1.7.1 Generation Quality

Perplexity: Lower is better, measures prediction quality.

BLEU: N-gram overlap with references (0-100, higher better).

ROUGE: Recall-based overlap, especially for summarization.

BERTScore: Embedding similarity, captures semantic equivalence.

1.7.2 Human Evaluation

Helpfulness: Does the response address the user's need?

Harmlessness: Does the response avoid harmful content?

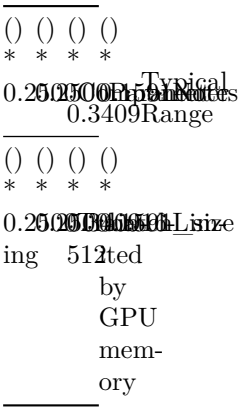
Honesty: Does the model acknowledge uncertainty appropriately?

Use Likert scales (1-5) with multiple annotators. Inter-annotator agreement (Krippendorff's alpha) should exceed 0.7.

1.8 Common Parameters

() *	() *	() *	() *
0.2	5	0	Typical 0.3409 Range
() *	() *	() *	() *
0.2	5	96	heads for larger models
() *	() *	() *	() *
0.2	5	12	8 h model size

*	*	*	*
0.2	5	20	0.3409 Typical Range
*	*	*	*
0.2	5	20	* Standard exploration
*	*	*	*
0.2	5	20	1.0 Lower for factual, higher for creative
*	*	*	*
0.2	5	20	0.95 Nucleus sampling threshold
*	*	*	*
0.2	5	20	100 Top-k sampling
*	*	*	*
0.2	5	20	64 Higher for complex adaptation
*	*	*	*
0.2	5	20	32 Usually 2x rank
*	*	*	*
0.2	5	20	5 to 5e-4 Lower for larger models



1.9 Practice Problems

1. **Attention Analysis:** Implement attention visualization for a transformer. Analyze which tokens attend to which for a given prompt. Identify patterns for different prompt types.
2. **Temperature Calibration:** Generate 100 responses at temperatures 0.1, 0.5, 0.7, 1.0, 1.5. Have humans rate diversity and quality. Find the optimal temperature for your use case.
3. **LoRA Comparison:** Fine-tune the same base model with ranks 4, 8, 16, 32. Measure task performance vs training time. Identify the rank that balances efficiency and quality.
4. **RAG Evaluation:** Build a QA system with and without retrieval. Compare accuracy on a test set. Analyze failure modes of each approach.
5. **Scaling Experiment:** Train language models with 10M, 50M, 100M parameters on fixed data. Plot loss curves and verify scaling law predictions.

1.10 References

1. Vaswani, A., et al. (2017). "Attention Is All You Need." NeurIPS.
2. Brown, T., et al. (2020). "Language Models are Few-Shot Learners." NeurIPS.
3. Hoffmann, J., et al. (2022). "Training Compute-Optimal Large Language Models." arXiv:2203.15556.
4. Ouyang, L., et al. (2022). "Training Language Models to Follow Instructions with Human Feedback." NeurIPS.
5. Hu, E. J., et al. (2022). "LoRA: Low-Rank Adaptation of Large Language Models." ICLR.
6. Rafailov, R., et al. (2023). "Direct Preference Optimization: Your Language Model is Secretly a Reward Model." NeurIPS.
7. Lewis, P., et al. (2020). "Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks." NeurIPS.

Generative AI combines theoretical insights from language modeling, attention mechanisms, and reinforcement learning with practical engineering for deployment. Mathematical understanding of scaling laws, training dynamics, and generation algorithms enables building systems that are both capable and controllable.