Lecture 6

Model Training Pipeline: Pre-training Large Language Models from Scratch

CSC 375/575 - Generative AI

Prof. Rongyu Lin
Quinnipiac University
School of Computing and Engineering

Required Reading

Required Readings Before Class:

- **Raschka:** "Build a Large Language Model (From Scratch)" - Chapter 5, Pages 150-189
- **Focus Topics:** Model training pipeline, loss calculation, training loops, text generation strategies, pretrained weights
- **Supplementary:** Review Chapter 4 on GPT architecture and text generation fundamentals

Before We Begin: From Random to Coherent

When you initialize a neural network with random weights, it produces complete gibberish. Yet after training, the same architecture generates coherent, meaningful text that can answer questions, write code, and engage in sophisticated reasoning. This transformation from randomness to intelligence is one of the most remarkable achievements in modern AI.

The key to this transformation lies in the training pipeline—a carefully orchestrated process of measuring prediction quality, adjusting weights through gradient descent, and iteratively improving the model's understanding of language patterns. In this lecture, we'll implement this entire pipeline from scratch, demystifying how raw computational power and clever algorithms combine to create intelligent language systems.

Opening Reflection: Consider this fundamental question: How does repeatedly predicting "the next word" in text lead to a model that can reason, plan, and create? The answer reveals the surprising power of self-supervised learning at scale.

1 Learning Objectives

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

1. Calculate and interpret cross-entropy loss for text generation evaluation
2. Implement complete training loops with proper monitoring and evaluation
3. Apply advanced text generation techniques (temperature, top-k sampling)

4. Load and integrate pretrained weights for transfer learning
5. Understand the practical economics and challenges of LLM training
6. Build end-to-end training pipelines from data preparation to deployment

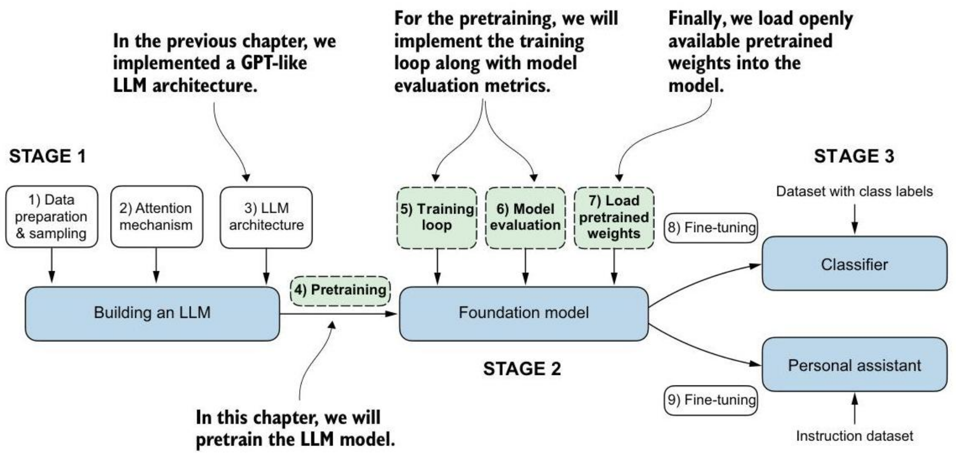
2 The Complete Training Pipeline

2.1 Overview and Context

Training a large language model involves transforming an initially random neural network into a sophisticated text generation system. This process, while conceptually straightforward, requires careful orchestration of multiple components working in harmony. The training pipeline we'll explore today represents the culmination of techniques developed in previous lectures, bringing together tokenization, model architecture, and optimization strategies.

The Training Pipeline Stages:

The complete LLM development pipeline consists of three main stages, with this lecture focusing primarily on the pre-training stage:



In the previous chapter, we implemented a GPT-like LLM architecture.
For the pretraining, we will implement the training loop along with model evaluation metrics.
Finally, we load openly available pretrained weights into the model.
STAGE 1
1) Data preparation & sampling
2) Attention mechanism
3) LLM architecture
Building an LLM
4) Pretraining
5) Training loop
6) Model evaluation
7) Load pretrained weights
Foundation model
STAGE 2
STAGE 3
Dataset with class labels
8) Fine-tuning
Classifier
Personal assistant
9) Fine-tuning
Instruction dataset
In this chapter, we will pretrain the LLM model.

Figure 1: The complete training pipeline showing three stages: data preparation, pre-training, and fine-tuning. Chapter 5 focuses on Stage 2—the pre-training process that transforms random weights into a functional language model. (Source: Raschka, Chapter 5, Figure 5.1)

Stage 1: Data Preparation (Covered in Lectures 3-4)

- Text collection and cleaning
- Tokenization and vocabulary creation
- Data loading and batching infrastructure
- Attention mask and position encoding preparation

Stage 2: Pre-training (This Lecture)

- Loss calculation and evaluation metrics
- Training loop implementation
- Optimization strategies and monitoring
- Model checkpointing and weight management

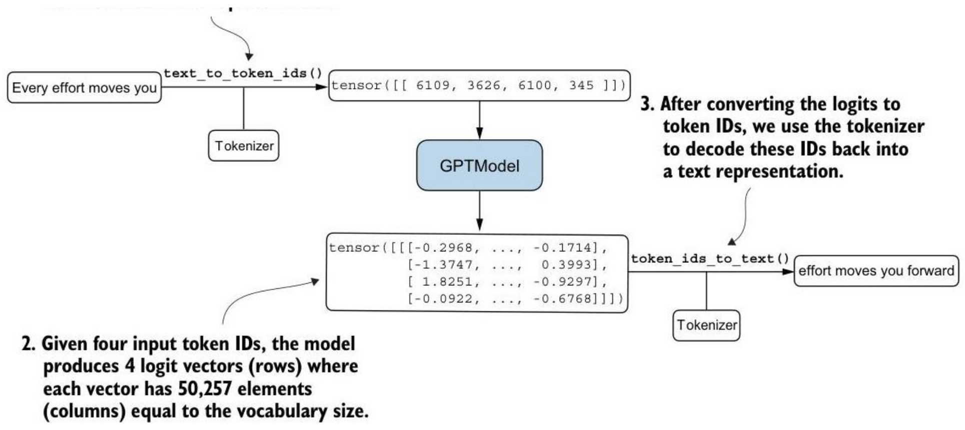
Stage 3: Fine-tuning (Future Lectures)

- Task-specific adaptation
- Instruction tuning
- Alignment with human preferences
- Deployment optimization

2.2 Text Generation Process Recap

Before diving into training, let's revisit the text generation process from Chapter 4. Understanding how models generate text is crucial for comprehending how we evaluate and improve their performance during training.

The Five-Step Generation Process:



Every effort moves you
text_to_token_ids()
tensor([[6109, 3626, 6100, 345]])
Tokenizer
GPTModel
3. After converting the logits to token IDs, we use the tokenizer to decode these IDs back into a text representation.
tensor([[[-0.2968, ..., -0.1714],
[-1.3747, ..., 0.3993],
[1.8251, ..., -0.9297],
[-0.0922, ..., -0.6768]]])
token_ids_to_text()
effort moves you forward
Tokenizer
2. Given four input token IDs, the model produces 4 logit vectors (rows) where each vector has 50,257 elements (columns) equal to the vocabulary size.

Figure 2: The text generation process showing the flow from input text through tokenization, model processing, and token selection to produce output text. This process forms the foundation for both inference and training. (Source: Raschka, Chapter 5, Figure 5.3)

1. **Tokenization:** Convert input text into token IDs that the model can process
2. **Model Processing:** Pass tokens through the transformer to generate logits (raw scores)
3. **Probability Conversion:** Apply softmax to convert logits into probability distributions
4. **Token Selection:** Choose the next token based on probabilities (greedy, sampling, etc.)
5. **Detokenization:** Convert selected token back to text and repeat the process

Before Training vs After Training:
When a model is randomly initialized, this process produces nonsensical output:

- **Input:** "Hello, I am"
- **Untrained Output:** "Hello, I am? Begins Vor nicht? ceremony? FLASH (): igua? booko?"
- **Target Output:** "Hello, I am a language model trained to assist with various tasks."

The training process adjusts the model's weights so that the probability distributions it produces align with patterns in natural language, transforming random noise into coherent generation.

3 Model Evaluation and Loss Calculation

3.1 Why Loss Calculation Matters

Training a neural network requires a quantitative measure of performance—we need to know not just that the model is "wrong" but exactly how wrong it is and in what direction to adjust. For language models, this measure is cross-entropy loss, which quantifies the difference between the model's predicted probability distribution and the actual next token in the training data.

The Fundamental Training Problem:

- **Human Evaluation:** We can easily judge if text is coherent, but this is subjective and non-differentiable
- **Automatic Metrics:** We need numerical feedback that can guide gradient-based optimization
- **Scalability:** The metric must be efficiently computable across billions of training examples
- **Interpretability:** Lower values should consistently indicate better performance

3.2 Understanding Cross-Entropy Loss

Cross-entropy loss, borrowed from information theory, measures the "surprise" or "uncertainty" in the model's predictions. When the model assigns high probability to the correct next token, the loss is low. When it assigns low probability to the correct token, the loss is high.

Intuitive Example:

Consider predicting the next word after "The weather is":

- **Good Model:** Assigns P("sunny") = 0.8, P("rainy") = 0.1, P("cold") = 0.05
- Loss for correct answer "sunny": $-\log(0.8) = 0.22$ (low loss)
- **Bad Model:** Assigns P("sunny") = 0.1, P("rainy") = 0.3, P("xyzzy") = 0.2
- Loss for correct answer "sunny": $-\log(0.1) = 2.30$ (high loss)

Mathematical Foundation:

The cross-entropy loss for a single prediction is:

$$\mathcal{L} = -\log P(y|x)$$

Where:

- y is the correct next token
- x is the input context
- $P(y|x)$ is the model's assigned probability to the correct token

For a batch of predictions, we average the individual losses:

$$\mathcal{L}_{batch} = -\frac{1}{N}\sum_{i=1}^{N}\log P(y_i|x_i)$$

This averaging ensures that the loss magnitude doesn't depend on batch size, allowing consistent optimization regardless of computational constraints.

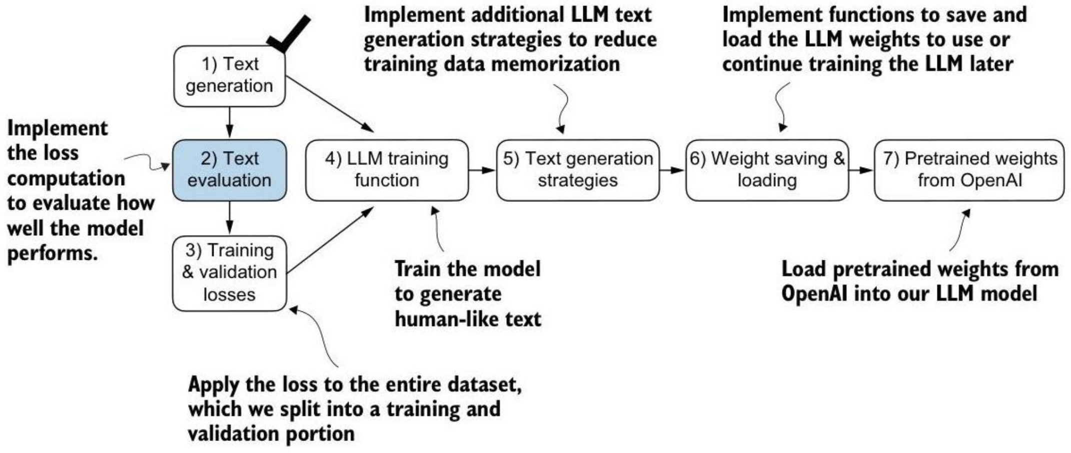

Implement additional LLM text generation strategies to reduce training data memorization
Implement functions to save and load the LLM weights to use or continue training the LLM later
1) Text generation
Implement the loss computation to evaluate how well the model performs.
2) Text evaluation
4) LLM training function
5) Text generation strategies
6) Weight saving & loading
7) Pretrained weights from OpenAI
3) Training & validation losses
Train the model to generate human-like text
Load pretrained weights from OpenAI into our LLM model
Apply the loss to the entire dataset, which we split into a training and validation portion

Figure 3: The six-step process for calculating cross-entropy loss, showing the transformation from logits through probabilities to the final loss value. Each step serves a specific purpose in quantifying prediction quality. (Source: Raschka, Chapter 5, Figure 5.7)

3.3 The Six-Step Loss Calculation Process

Computing cross-entropy loss involves a systematic transformation from raw model outputs to a single scalar value that guides optimization:

Detailed Step Breakdown:

Step 1-3: Generate Token Probabilities These steps mirror the generation process:

- Process input through the model to get logits
- Apply softmax to convert logits to probabilities
- Result: probability distribution over vocabulary for each position

Step 4: Apply Logarithm

- Take the natural logarithm of the probability assigned to the correct token
- Logarithm maps probabilities [0,1] to log-probabilities [-, 0]
- This transformation has desirable mathematical properties for optimization

Step 5: Apply Negative Sign

- Negate the log-probability to get positive loss values
- Higher probability $\rightarrow$ less negative log $\rightarrow$ lower positive loss
- This creates the desired ”lower is better” loss landscape

Step 6: Average Across Batch

- Compute mean loss across all predictions in the batch
- Averaging provides consistent gradients regardless of batch size
- Enables fair comparison across different training configurations

3.4 Implementation: Manual vs PyTorch Approaches

Understanding the loss calculation deeply requires implementing it manually before using optimized library functions. Let's explore both approaches:

Manual Implementation for Learning:

```
def calc_loss_manual(input_batch, target_batch, model, device):
    """
    Manually calculate cross-entropy loss step by step.
    This approach reveals the mathematical operations involved.
    """
    # Move data to appropriate device (CPU/GPU)
    input_batch = input_batch.to(device)
    target_batch = target_batch.to(device)

    # Forward pass through model
    logits = model(input_batch)  # Shape: (batch_size, seq_len, vocab_size)

    # Flatten for easier manipulation
    logits_flat = logits.flatten(0, 1)      # (batch * seq_len, vocab_size)
    targets_flat = target_batch.flatten()   # (batch * seq_len)

    # Convert logits to probabilities using softmax
    probabilities = torch.softmax(logits_flat, dim=1)

    # Extract probabilities for correct tokens
    # Use advanced indexing to select target probabilities
    batch_size = len(targets_flat)
    target_probs = probabilities[range(batch_size), targets_flat]

    # Compute negative log probability
    # Add small epsilon for numerical stability
    log_probs = torch.log(target_probs + 1e-9)

    # Average to get final loss
    loss = -log_probs.mean()

    return loss
```

Optimized PyTorch Implementation:

```
def calc_loss_pytorch(input_batch, target_batch, model, device):
    """
    Use PyTorch's optimized cross-entropy implementation.
    This is preferred for actual training due to efficiency and stability.
    """
    input_batch = input_batch.to(device)
    target_batch = target_batch.to(device)

    # Forward pass
    logits = model(input_batch)

    # PyTorch's cross_entropy handles everything internally:
    # - Softmax computation
    # - Log probability extraction
    # - Negative sign application
    # - Averaging across batch
    # - Numerical stability optimizations
    loss = torch.nn.functional.cross_entropy(
        logits.flatten(0, 1),
        target_batch.flatten()
    )

    return loss
```

```

# Verification that both produce identical results
manual_loss = calc_loss_manual(batch, targets, model, device)
pytorch_loss = calc_loss_pytorch(batch, targets, model, device)
print(f"Manual: {manual_loss:.6f}, PyTorch: {pytorch_loss:.6f}")
# Output: Manual: 10.750950, PyTorch: 10.750950
```

Key Differences and Trade-offs:

- **Manual Approach:** Educational value, transparency, easier debugging
- **PyTorch Approach:** Production-ready, numerically stable, highly optimized
- **Performance:** PyTorch version is significantly faster for large-scale training
- **Numerical Stability:** PyTorch handles edge cases and prevents overflow/underflow

3.5 Target Preparation and Data Alignment

A crucial but often overlooked aspect of training language models is properly aligning inputs with targets. Since the model learns to predict the next token, we need to shift our data appropriately:

The Shift-by-One Pattern:

```
# Original text sequence
text = "Every effort moves you forward"
tokens = [Every, effort, moves, you, forward]

# Training pairs (input      target)
Input[0]: "Every"       Target[0]: "effort"
Input[1]: "effort"      Target[1]: "moves"
Input[2]: "moves"       Target[2]: "you"
Input[3]: "you"         Target[3]: "forward"
Input[4]: "forward"     Target[4]: <|endoftext|>

# In practice, we process entire sequences at once
input_batch = tokens[:-1]  # [Every, effort, moves, you]
target_batch = tokens[1:]  # [effort, moves, you, forward]
```

This shift-by-one pattern enables self-supervised learning—we don't need manually labeled data because every token in our corpus serves as both an input (predicting the next token) and a target (being predicted by the previous token).

4 Training Data Management and DataLoaders

4.1 Training vs Validation Split

Proper data management is crucial for training models that generalize well. The standard practice involves splitting data into training and validation sets, each serving distinct purposes in the training pipeline.

Data Split Strategy:

- **Training Set (90%):** Used for computing gradients and updating weights
- **Validation Set (10%):** Used for monitoring generalization and detecting overfitting
- **Test Set (separate):** Held out for final evaluation (not used during training)

Why Validation Matters:
The validation set acts as an early warning system for overfitting:

- **Training Loss:** Continuously decreases as the model memorizes training data
- **Validation Loss:** Initially decreases, then may increase if overfitting occurs
- **Divergence Point:** When validation loss stops improving, consider stopping training

4.2 Implementing Efficient DataLoaders

DataLoaders handle the complexity of batching, shuffling, and efficiently feeding data to the model during training. For language models, we need specialized loaders that handle sequential data properly:

```python
def create_dataloader_v1(txt, batch_size=4, max_length=256,
                         stride=128, shuffle=True, drop_last=True,
                         num_workers=0):
    """
    Create a DataLoader for LLM training with overlapping windows.

    Args:
        txt: Raw text data
        batch_size: Number of sequences per batch
        max_length: Length of each sequence
        stride: Overlap between consecutive sequences
        shuffle: Whether to randomize sequence order
        drop_last: Drop incomplete final batch
    """
    # Tokenize entire text
    tokenizer = tiktoken.get_encoding("gpt2")
    encoded = tokenizer.encode(txt)
    token_ids = torch.tensor(encoded).long()

    # Create dataset with sliding window
    dataset = GPTDatasetV1(token_ids, max_length, stride)

    # Create DataLoader
    dataloader = DataLoader(
        dataset,
        batch_size=batch_size,
        shuffle=shuffle,
        drop_last=drop_last,
        num_workers=num_workers
    )

    return dataloader

class GPTDatasetV1(Dataset):
    """Custom dataset for GPT training with sliding windows."""

    def __init__(self, txt, max_length, stride):
        self.input_ids = []
        self.target_ids = []

        # Create overlapping sequences
        for i in range(0, len(txt) - max_length, stride):
            input_chunk = txt[i:i + max_length]
            target_chunk = txt[i + 1:i + max_length + 1]
            self.input_ids.append(input_chunk)
            self.target_ids.append(target_chunk)

    def __len__(self):
        return len(self.input_ids)

    def __getitem__(self, idx):
        return self.input_ids[idx], self.target_ids[idx]
```

Key Design Decisions:
Sliding Window with Stride:

- Creates overlapping sequences to maximize data utilization
- Stride ¡ max_length means sequences share some tokens
- Helps model learn continuity across sequence boundaries

Batch Processing:

- Groups multiple sequences for parallel processing
- Improves GPU utilization and training speed
- Requires careful memory management for large models

5 The Training Loop

5.1 Core Training Loop Structure

The training loop is the heart of the model training process, orchestrating the forward pass, loss calculation, backpropagation, and weight updates. A well-designed training loop includes monitoring, checkpointing, and early stopping mechanisms.

Essential Components:

```
def train_model_simple(model, train_loader, val_loader, optimizer,
                       device, num_epochs, eval_freq, eval_iter,
                       start_context, tokenizer):
    """
    Complete training loop with validation monitoring.

    Components:
    1. Epoch iteration
    2. Batch processing
    3. Loss computation
    4. Backpropagation
    5. Weight updates
    6. Validation evaluation
    7. Progress monitoring
    """
    # Tracking lists
    train_losses, val_losses, track_tokens_seen = [], [], []
    tokens_seen, global_step = 0, -1

    # Main training loop
    for epoch in range(num_epochs):
        model.train()  # Set to training mode

        for input_batch, target_batch in train_loader:
            optimizer.zero_grad()  # Reset gradients

            # Forward pass and loss calculation
            loss = calc_loss_batch(input_batch, target_batch,
                                   model, device)

            # Backpropagation
            loss.backward()

```

```python
            # Update weights
            optimizer.step()

            # Track progress
            tokens_seen += input_batch.numel()
            global_step += 1

            # Periodic evaluation
            if global_step % eval_freq == 0:
                train_loss, val_loss = evaluate_model(
                    model, train_loader, val_loader,
                    device, eval_iter
                )
                train_losses.append(train_loss)
                val_losses.append(val_loss)
                track_tokens_seen.append(tokens_seen)

                print(f"Ep {epoch+1} (Step {global_step:06d}): "
                      f"Train loss {train_loss:.3f}, "
                      f"Val loss {val_loss:.3f}")

        # Generate sample text to visualize progress
        generate_and_print_sample(
            model, tokenizer, device, start_context
        )

    return train_losses, val_losses, track_tokens_seen
```

5.2 Understanding Gradient Descent and Optimization

The optimizer is responsible for updating model weights based on computed gradients. For LLMs, AdamW (Adam with weight decay) is the standard choice due to its adaptive learning rates and momentum.

AdamW Optimizer Configuration:

```python
optimizer = torch.optim.AdamW(
    model.parameters(),
    lr=0.0004,         # Learning rate
    betas=(0.9, 0.98), # Momentum parameters
    eps=1e-9,          # Numerical stability
    weight_decay=0.1   # L2 regularization
)

# Learning rate affects convergence speed and stability
# Too high: Training unstable, loss explodes
# Too low: Training slow, may get stuck in local minima
# Typical range for LLMs: 1e-4 to 6e-4
```

Key Hyperparameters:

- **Learning Rate:** Controls step size in weight updates
- **Betas:** Momentum parameters for adaptive learning
- **Weight Decay:** Regularization to prevent overfitting
- **Epsilon:** Small constant for numerical stability

5.3 Monitoring Training Progress

Effective monitoring is essential for successful training. Key metrics to track include:

Training Metrics:

- **Training Loss:** Should steadily decrease
- **Validation Loss:** Should decrease initially, watch for increases (overfitting)
- **Loss Gap:** Difference between training and validation loss
- **Tokens Processed:** Total computational work performed
- **Generated Samples:** Qualitative assessment of model output

Typical Training Progression:

```
# Early training (random weights)
Epoch 1 (Step 000000): Train loss 10.751, Val loss 10.742
Output: "Every effort moves you? Begins Vor nicht ceremony..."

# Mid training (learning patterns)
Epoch 10 (Step 002500): Train loss 6.234, Val loss 6.451
Output: "Every effort moves you forward and the time when..."

# Late training (coherent generation)
Epoch 50 (Step 012500): Train loss 0.234, Val loss 1.567
Output: "Every effort moves you forward in understanding how neural networks
    learn from data..."
```

5.4 Common Training Challenges and Solutions

Training LLMs presents several common challenges that require careful attention:

Challenge 1: Overfitting

- **Symptoms:** Training loss decreases while validation loss increases
- **Solutions:** Early stopping, dropout, weight decay, data augmentation
- **Prevention:** Regular validation monitoring, larger datasets

Challenge 2: Gradient Instability

- **Symptoms:** Loss spikes, NaN values, training collapse
- **Solutions:** Gradient clipping, learning rate warmup, careful initialization
- **Prevention:** Stable optimizer settings, proper normalization

Challenge 3: Slow Convergence

- **Symptoms:** Loss decreases very slowly or plateaus early
- **Solutions:** Adjust learning rate, change batch size, modify architecture
- **Prevention:** Proper hyperparameter tuning, learning rate scheduling

6 Advanced Text Generation Strategies

6.1 The Problem with Greedy Decoding

The simplest text generation strategy—always selecting the highest probability token—often produces repetitive and uninteresting text. This ”greedy” approach lacks the diversity and creativity we expect from language models.

Greedy Decoding Limitations:

```python
# Greedy generation (always pick highest probability)
def generate_greedy(model, start_context, max_tokens):
    for _ in range(max_tokens):
        logits = model(context)
        probs = softmax(logits)
        next_token = argmax(probs)  # Always highest probability
        context.append(next_token)
    return context

# Typical greedy output (repetitive and boring):
"The cat sat on the mat. The mat was the mat. The mat was the mat..."
```

Why Greedy Fails:

- Natural language has inherent variability
- Always choosing ”safe” options leads to generic text
- Real communication involves controlled randomness
- Creativity requires exploring lower-probability options

6.2 Temperature Sampling

Temperature sampling introduces controlled randomness by scaling the logits before applying softmax. This simple technique dramatically improves generation quality.

Temperature Scaling Mathematics:

The temperature parameter T modifies the softmax calculation:

$$P(x_i) = \frac{e^{logit_i/T}}{\sum_j e^{logit_j/T}}$$

Temperature Effects:

- **T = 0:** Approaches greedy decoding (deterministic)
- **T = 0.7:** Slightly conservative, coherent generation
- **T = 1.0:** Original distribution (no modification)
- **T = 1.5:** More creative, occasional surprises
- **T ¿ 2.0:** Very random, often incoherent

```python
def apply_temperature(logits, temperature):
    """
    Scale logits by temperature before sampling.
    Lower temperature = more focused
    Higher temperature = more random
    """
    if temperature == 0.0:
        # Special case: return argmax for deterministic output
        return torch.argmax(logits, dim=-1)

    # Scale logits
    scaled_logits = logits / temperature

    # Convert to probabilities
    probs = torch.softmax(scaled_logits, dim=-1)

    # Sample from distribution
    return torch.multinomial(probs, num_samples=1)

# Examples with different temperatures
temp_0_7 = "The cat sat on the mat and looked around."  # Coherent
temp_1_5 = "The cat leaped onto the refrigerator, meowing loudly!"  # Creative
temp_2_5 = "The cat purple democracy verbose sandwich."  # Incoherent
```

6.3 Top-k Sampling

Top-k sampling restricts token selection to the k most likely candidates, preventing the model from choosing extremely unlikely tokens that could derail generation.

Top-k Algorithm:

```
def top_k_sampling(logits, k, temperature=1.0):
    """
    Sample from only the top k most likely tokens.
    Combines quality control with diversity.
    """
    # Find top k logits and indices
    top_logits, top_indices = torch.topk(logits, k)

    # Apply temperature scaling
    top_logits = top_logits / temperature

    # Convert to probabilities (only for top k)
    top_probs = torch.softmax(top_logits, dim=-1)

    # Sample from top k distribution
    sample_idx = torch.multinomial(top_probs, num_samples=1)

    # Map back to vocabulary index
    return top_indices.gather(-1, sample_idx)

# Example: k=50 prevents selecting from 50,000+ unlikely tokens
# Maintains creativity while avoiding nonsense
```

Choosing k:

- **k = 1:** Equivalent to greedy decoding
- **k = 10:** Very focused, limited diversity
- **k = 50:** Good balance for most applications
- **k = 100:** More diverse, occasional surprises
- **k = vocab_size:** No filtering (standard sampling)

6.4 Combining Temperature and Top-k

The most effective generation strategy combines both temperature and top-k sampling, leveraging the strengths of each approach:

```
def generate_advanced(model, idx, max_new_tokens, context_size,
                      temperature=0.7, top_k=50, eos_id=None):
    """
    Advanced generation with temperature and top-k sampling.
    Production-ready implementation used in real systems.
    """
    for _ in range(max_new_tokens):
```

```python
        # Crop context to maximum length
        idx_cond = idx[:, -context_size:]

        # Get predictions
        with torch.no_grad():
            logits = model(idx_cond)

        # Focus on last position
        logits = logits[:, -1, :]

        # Apply top-k filtering
        if top_k is not None:
            # Keep only top k logits, set others to -inf
            v, _ = torch.topk(logits, top_k)
            logits[logits < v[:, [-1]]] = float('-inf')

        # Apply temperature and sample
        probs = torch.softmax(logits / temperature, dim=-1)
        idx_next = torch.multinomial(probs, num_samples=1)

        # Check for end-of-sequence token
        if idx_next == eos_id:
            break

        # Append to sequence
        idx = torch.cat((idx, idx_next), dim=1)

    return idx

# Practical settings for different use cases
creative_writing = {"temperature": 1.2, "top_k": 100}
code_generation = {"temperature": 0.3, "top_k": 30}
factual_qa = {"temperature": 0.1, "top_k": 10}
```

7 Pretrained Weights and Transfer Learning

7.1 The Economics of Pretraining

Training large language models from scratch is prohibitively expensive for most organizations. Understanding the economics helps appreciate why pretrained weights are so valuable:

Training Costs Breakdown:

- **GPT-2 (1.5B parameters):**
  - Training time: 1 week on 32 V100 GPUs
  - Estimated cost: $50,000-$80,000
  - Dataset: 40GB of text (WebText)
- **GPT-3 (175B parameters):**
  - Training time: 34 days on 10,000 V100 GPUs
  - Estimated cost: $4.6 million
  - Dataset: 570GB of filtered text
- **GPT-4 (estimated 1.7T parameters):**
  - Training time: 3-6 months on 25,000+ A100 GPUs
  - Estimated cost: $100+ million

    – Dataset: Multiple terabytes of curated data

Beyond Compute Costs:

- Data collection and cleaning: Months of engineering effort
- Hyperparameter tuning: Hundreds of experimental runs
- Infrastructure development: Custom distributed training systems
- Expertise: Teams of specialized ML engineers and researchers

7.2 Loading Pretrained Weights

Fortunately, many organizations release pretrained weights that can be loaded and fine-tuned for specific tasks. Here's how to load and use pretrained GPT-2 weights:

```python
def load_gpt2_weights(model, model_size="124M"):
    """
    Load pretrained GPT-2 weights from OpenAI.
    Transforms and maps weights to our model architecture.
    """
    # Download weights from OpenAI (or use cached version)
    import tensorflow as tf
    from transformers import GPT2Model

    # Load TensorFlow checkpoint
    tf_checkpoint_path = f"gpt2-{model_size}/model.ckpt"

    # Map TensorFlow variable names to PyTorch
    def convert_tf_weight_name_to_pytorch(tf_name):
        # Complex mapping logic
        # Example: "model/h0/attn/c_attn/w" -> "transformer.h.0.attn.c_attn.
            weight"
        # ... (detailed mapping code)
        pass

    # Load and convert weights
    tf_to_pytorch = {}
    reader = tf.train.load_checkpoint(tf_checkpoint_path)

    for name in reader.get_variable_to_shape_map():
        pytorch_name = convert_tf_weight_name_to_pytorch(name)
        tf_to_pytorch[pytorch_name] = reader.get_tensor(name)

    # Assign weights to model
    model_dict = model.state_dict()
    for name, param in model_dict.items():
        if name in tf_to_pytorch:
            param.data = torch.from_numpy(tf_to_pytorch[name])

    return model

# Simplified loading using HuggingFace
from transformers import GPT2LMHeadModel

def load_pretrained_simple(model_name="gpt2"):
    """
    Simple loading using HuggingFace transformers library.
    """
    model = GPT2LMHeadModel.from_pretrained(model_name)
    return model

```

```python
# Usage example
model = load_pretrained_simple("gpt2-medium")
```

7.3 Weight Structure and Architecture Alignment

Understanding the structure of pretrained weights is crucial for successful transfer learning:

GPT-2 Weight Organization:

```python
# Examining loaded weights
def examine_model_weights(model):
    """
    Explore the structure and statistics of model weights.
    """
    total_params = 0

    for name, param in model.named_parameters():
        param_count = param.numel()
        total_params += param_count

        print(f"{name:50} | Shape: {str(param.shape):20} | "
              f"Params: {param_count:10,}")

        # Weight statistics
        if param.dim() > 1:
            mean = param.mean().item()
            std = param.std().item()
            print(f"{'':50} | Mean: {mean:.6f}, Std: {std:.6f}")

    print(f"\nTotal parameters: {total_params:,}")

    return total_params

# Example output for GPT-2 small (124M)
"""
transformer.wte.weight                    | Shape: (50257, 768)     | Params:
    38,597,376
transformer.wpe.weight                    | Shape: (1024, 768)      | Params:
    786,432
transformer.h.0.ln_1.weight              | Shape: (768,)            | Params: 768
transformer.h.0.attn.c_attn.weight       | Shape: (768, 2304)       | Params:
    1,769,472
transformer.h.0.attn.c_proj.weight       | Shape: (768, 768)        | Params:
    589,824
...
Total parameters: 124,439,808
"""
```

Key Components:

- **Token Embeddings (wte):** Maps token IDs to vectors
- **Position Embeddings (wpe):** Encodes position information
- **Transformer Blocks (h.N):** N layers of self-attention and FFN
- **Layer Norms (ln_1, ln_2):** Stabilize activations
- **Output Head (lm_head):** Projects to vocabulary for prediction

7.4 Before and After: The Impact of Pretrained Weights

The difference between random initialization and pretrained weights is dramatic:

Comparison Example:

```python
# Generate with random weights
random_model = GPTModel(config)
random_output = generate(random_model, "The future of AI is")
print("Random:", random_output)
# Output: "The future of AI is@#& nicht blob%.? SYSTEM}}]["

# Generate with pretrained weights
pretrained_model = load_pretrained_simple("gpt2")
pretrained_output = generate(pretrained_model, "The future of AI is")
print("Pretrained:", pretrained_output)
# Output: "The future of AI is likely to involve more sophisticated
#          reasoning capabilities and better understanding of context"
```

Why Pretrained Weights Work:

- Learned general language patterns from massive datasets
- Captured world knowledge and common sense reasoning
- Developed robust internal representations
- Can be fine-tuned with much less data and compute

8 Fine-tuning vs Pretraining

8.1 The Two-Stage Training Process

Modern LLM development follows a two-stage process that maximizes both general capability and task-specific performance:

Stage 1: Pretraining (This Lecture)

- **Objective:** Learn general language understanding
- **Data:** Massive, diverse text corpora
- **Duration:** Weeks to months
- **Cost:** Millions of dollars
- **Result:** Foundation model with broad capabilities

Stage 2: Fine-tuning (Future Lectures)

- **Objective:** Adapt to specific tasks or domains
- **Data:** Smaller, task-specific datasets
- **Duration:** Hours to days
- **Cost:** Hundreds to thousands of dollars
- **Result:** Specialized model for particular application

8.2 Fine-tuning Strategies

There are several approaches to fine-tuning pretrained models:

Full Fine-tuning:

- Update all model parameters
- Best performance but computationally expensive
- Risk of catastrophic forgetting

Parameter-Efficient Fine-tuning (PEFT):

- Update only a small subset of parameters
- Methods: LoRA, Adapters, Prompt Tuning
- Maintains most pretrained knowledge

Instruction Tuning:

- Train model to follow instructions
- Uses formatted prompt-response pairs
- Creates more helpful and controllable models

9 Practical Considerations and Best Practices

9.1 Memory Management

Training LLMs requires careful memory management to avoid out-of-memory errors:

```python
# Memory-efficient training techniques
def estimate_memory_usage(model, batch_size, seq_length):
    """
    Estimate GPU memory requirements for training.
    """
    param_memory = sum(p.numel() * 4 for p in model.parameters())  # 4 bytes
        per float32

    # Gradients need same memory as parameters
    gradient_memory = param_memory

    # Optimizer states (Adam needs 2x parameters for momentum)
    optimizer_memory = param_memory * 2

    # Activations (rough estimate)
    activation_memory = batch_size * seq_length * model.d_model * 100 * 4

    total_memory = param_memory + gradient_memory + optimizer_memory +
        activation_memory

    print(f"Estimated memory usage: {total_memory / 1e9:.2f} GB")

    return total_memory

# Techniques to reduce memory usage
1. Gradient accumulation: Simulate larger batches
2. Mixed precision training: Use float16 where possible
3. Gradient checkpointing: Trade compute for memory
4. Model parallelism: Split model across GPUs
```

9.2 Training Stability

Ensuring stable training requires several techniques:

Gradient Clipping:

```python
# Prevent gradient explosion
torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)
```

Learning Rate Scheduling:

```python
# Warmup followed by cosine decay
scheduler = get_cosine_schedule_with_warmup(
    optimizer,
    num_warmup_steps=100,
    num_training_steps=10000
)
```

Checkpoint Saving:

```python
# Save model periodically
if step % save_interval == 0:
    torch.save({
        'model_state_dict': model.state_dict(),
        'optimizer_state_dict': optimizer.state_dict(),
        'step': step,
        'loss': loss
    }, f'checkpoint_step_{step}.pt')
```

10 Summary and Key Takeaways

This lecture covered the complete pipeline for pretraining language models from scratch. Let's consolidate the key concepts:

10.1 Core Concepts Mastered

Loss Calculation and Evaluation:

- Cross-entropy loss quantifies prediction quality
- Lower loss indicates better next-token prediction
- Manual implementation reveals mathematical foundations
- PyTorch functions provide optimized implementations

Training Infrastructure:

- DataLoaders efficiently batch and shuffle training data
- Training loops orchestrate forward passes, backpropagation, and updates
- Validation monitoring prevents overfitting
- Proper optimization requires careful hyperparameter tuning

Generation Strategies:

- Temperature controls randomness vs determinism
- Top-k sampling prevents selection of unlikely tokens

- Combining both achieves optimal generation quality
- Different tasks require different generation settings

Transfer Learning:

- Pretraining is extremely expensive but creates general capabilities
- Pretrained weights can be loaded and fine-tuned efficiently
- Two-stage training (pretrain + fine-tune) is the standard paradigm
- Understanding weight structure enables effective transfer learning

10.2 Practical Skills Developed

Through this lecture, you've gained the ability to:

1. Implement complete training loops with proper monitoring
2. Calculate and interpret loss values for model evaluation
3. Apply advanced generation techniques for better text quality
4. Load and utilize pretrained models for transfer learning
5. Debug common training issues and apply solutions
6. Make informed decisions about training vs fine-tuning

10.3 Looking Ahead

In the next lecture, we'll explore fine-tuning techniques that transform general-purpose pretrained models into specialized systems for specific tasks. We'll cover:

- Instruction tuning for creating helpful assistants
- Parameter-efficient fine-tuning methods
- Alignment with human preferences
- Evaluation metrics for task-specific performance
- Deployment considerations for production systems

The journey from random initialization to intelligent text generation represents one of the most remarkable achievements in modern AI. By understanding the training pipeline in detail, you're equipped to not just use these models but to truly understand how they learn and improve.

Further Reading

Essential Papers:

- Radford, A., et al. "Language Models are Unsupervised Multitask Learners" (GPT-2, 2019)
- Brown, T., et al. "Language Models are Few-Shot Learners" (GPT-3, 2020)

- Kaplan, J., et al. ”Scaling Laws for Neural Language Models” (2020)
- Hoffmann, J., et al. ”Training Compute-Optimal Large Language Models” (Chinchilla, 2022)

Additional Resources:

- The Illustrated GPT-2 (Jay Alammar’s visual guide)
- ”Distributed Training: Train BART/T5 for Summarization using Transformers and Amazon SageMaker”
- OpenAI’s GPT-2 release blog post and code repository
- HuggingFace Transformers documentation on training

Practical Tutorials:

- ”Fine-tuning GPT-2 from scratch” (HuggingFace course)
- ”Training a causal language model from scratch” (PyTorch tutorial)
- ”Distributed Data Parallel Training” (PyTorch documentation)