Chapter 5

Pretraining Large Language Models: Evaluation, Training, and Text Generation

CSC 375 - 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 150-190
- **Focus Topics:** Loss calculation, training loops, text generation strategies, pretrained weights
- **Supplementary:** PyTorch documentation for cross-entropy loss and optimization algorithms

Before We Begin: A Question to Ponder

Imagine you have just built a sophisticated GPT model with millions of parameters, but when you ask it to complete "The weather today is...", it responds with random gibberish like "rentingetic wouldn? refres RexMeCHicular". Your model has the right architecture, the right size, and all the computational power needed—yet it produces meaningless text.

This scenario illustrates the fundamental challenge we face in Chapter 5: transforming a randomly initialized model into an intelligent language generator. How do we measure progress objectively? How do we systematically improve the model's predictions? And once trained, how do we control the creativity and coherence of the generated text?

Opening Reflection: Before reading further, consider this: How would you define "good" text generation mathematically? What would success look like in numerical terms? Think about the relationship between prediction accuracy and language understanding.

1 Learning Objectives

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

1. **Evaluate** generative models using cross-entropy loss and understand the mathematical foundation
2. **Implement** complete training loops for LLMs with proper data handling and optimization
3. **Calculate** and interpret training/validation losses to monitor model performance

4. **Apply** advanced text generation techniques including temperature and top-k sampling
5. **Load** and utilize pretrained model weights for transfer learning applications
6. **Understand** the complete pretraining process from initialization to deployment

2 The Training Pipeline: From Random to Remarkable

2.1 Understanding Weight Parameters in Large Language Models

Large Language Models derive their power from millions or billions of trainable parameters—the weights that determine how information flows through the neural network. Understanding these parameters is crucial for effective training and optimization.

What Are Weight Parameters? Weight parameters are numerical values that control the strength of connections between neurons in the neural network. During training, these weights are systematically adjusted to minimize prediction errors, gradually teaching the model to understand and generate coherent text.

Location of Parameters in LLMs:

- **Embedding Layers:** Convert token IDs to dense vector representations
- **Attention Mechanism Weights:** Query, key, value, and output projection matrices
- **Feed-Forward Network Parameters:** Two linear transformations with activation functions
- **Layer Normalization:** Scale and shift parameters for stabilizing training
- **Position Encoding:** Learned or fixed parameters for sequence order information

Accessing Parameters in PyTorch:

```
# Count total parameters in your model
total_params = sum(p.numel() for p in model.parameters())
print(f"Total parameters: {total_params:,}")

# Access specific layer parameters
for name, param in model.named_parameters():
    print(f"{name}: {param.shape}")

# Example output for GPT-2 Small:
# transformer.wte.weight: torch.Size([50257, 768])  # Token embeddings
# transformer.wpe.weight: torch.Size([1024, 768])   # Position embeddings
# transformer.h.0.attn.c_attn.weight: torch.Size([768, 2304])  # Attention
    weights
```

Scale Perspectives (Reference: Raschka p. 151):

Model	Parameters	Memory (FP32)
GPT-2 Small	124 million	0.5 GB
GPT-2 Medium	350 million	1.4 GB
GPT-2 Large	774 million	3.1 GB
GPT-3	175 billion	700 GB

Scale Reality Check: The jump from GPT-2 to GPT-3 represents a 226x increase in parameters. This dramatic scaling, combined with more training data and compute, leads to the emergence of new capabilities that smaller models cannot achieve.

2.2 The Challenge: Untrained Model Behavior

Before training, even the most sophisticated language model produces incoherent text because its weights are randomly initialized. Understanding this baseline helps us appreciate the magnitude of the training challenge.

Example of Untrained Model Output: When prompted with "Every effort moves you", an untrained GPT model might generate:

```
"Every effort moves you rentingetic wouldn? refres RexMeCHicular stren"
```

Why Untrained Models Fail:

1. **Random Weight Initialization:** Parameters start with random values, typically drawn from normal distributions
2. **No Learned Patterns:** The model has never seen language data and knows nothing about word relationships
3. **Uniform Probability Distributions:** Without training, all tokens in the vocabulary have roughly equal probability
4. **No Contextual Understanding:** The attention mechanisms haven't learned to focus on relevant context

Testing Untrained Model Behavior:

```python
# Generate text with untrained model
start_context = "Every effort moves you"
model.eval()  # Set to evaluation mode
context = torch.tensor(encode_text(start_context)).unsqueeze(0)

# Use the simple generation function
generated = generate_text_simple(
    model,
    context,
    max_new_tokens=10,
    context_size=GPT_CONFIG_124M["context_length"]
)
print(f"Untrained output: {generated}")
```

The Training Imperative: The incoherent output from untrained models demonstrates why we need systematic training with proper loss functions. The goal is to transform random parameter values into meaningful language representations through exposure to vast amounts of text data.

3 Model Evaluation: Measuring Language Understanding

3.1 The Evaluation Challenge

Evaluating text generation quality presents unique challenges compared to traditional machine learning tasks. Unlike classification problems with clear right/wrong answers, language generation involves subjective qualities like coherence, creativity, and appropriateness.

Human Evaluation Limitations:

- **Inconsistency:** Different evaluators may rate the same text differently
- **Scalability:** Manual evaluation is time-consuming and expensive
- **Subjectivity:** Quality assessment varies based on context and personal preferences

- **Non-differentiable:** Human judgments cannot drive gradient-based optimization

The Solution: Cross-Entropy Loss Cross-entropy loss provides an objective, mathematical measure of model performance based on next-token prediction accuracy. A model that consistently assigns high probability to the actual next tokens in training text demonstrates better language understanding.

3.2 Cross-Entropy Loss: Mathematical Foundation

Cross-entropy loss measures the difference between predicted probability distributions and the true distribution of language. In language modeling, the "true" distribution assigns probability 1 to the actual next token and 0 to all others.

Mathematical Definition:

$$\text{CrossEntropy} = -\sum_{i=1}^{V} y_i \log(\hat{y}_i)$$

Where:

- V = vocabulary size (typically 50,000+ tokens)
- y_i = true probability (1 for correct token, 0 for all others)
- $\hat{y}_i$ = predicted probability from the model

Simplified Form for Language Modeling: Since only one token has true probability 1, this simplifies to:

$$\text{CrossEntropy} = -\log(\hat{y}_{\text{target}})$$

Where $\hat{y}_{\text{target}}$ is the model's predicted probability for the actual next token.

Intuitive Understanding:

- **Low Loss (Good):** High probability assigned to correct token
  - Example: P("cat" — "The") = 0.9 → Loss = -log(0.9) = 0.046
- **High Loss (Bad):** Low probability assigned to correct token
  - Example: P("cat" — "The") = 0.1 → Loss = -log(0.1) = 1.0

3.3 Step-by-Step Loss Calculation

Let's walk through a complete example of calculating cross-entropy loss for language modeling using real tensors and operations.

Example Setup:

```
# Two training sequences
inputs = torch.tensor([[16833, 3626, 6100], [40, 1107, 588]])
targets = torch.tensor([[3626, 6100, 345], [1107, 588, 11311]])

# Note: targets are inputs shifted by one position
# This creates the next-token prediction task
```

Step 1: Model Forward Pass

```
# Get logits from model
logits = model(inputs)  # Shape: [batch_size, seq_len, vocab_size]
print(f"Logits shape: {logits.shape}")  # [2, 3, 50257]
```

Step 2: Convert Logits to Probabilities Logits are raw model outputs that can be any real numbers. We use softmax to convert them to proper probabilities:

$$\text{Softmax}(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{V} e^{z_j}}$$

```
# Apply softmax transformation
probas = torch.softmax(logits, dim=-1)
print(f"Probabilities shape: {probas.shape}")
print(f"Sum check: {probas[0, 0].sum()}")  # Should be 1.0

# Properties of softmax output:
# - All values between 0 and 1
# - Sum of all probabilities = 1
# - Larger logits $\rightarrow$ higher probabilities
```

Step 3: Extract Target Probabilities We need to extract the probabilities that the model assigned to the actual next tokens:

```
# Get probabilities for target tokens using advanced indexing
# This extracts the probability at each target token position
target_probas = probas[
    range(len(targets)),           # Batch dimension
    range(targets.shape[1]),       # Sequence dimension
    targets.flatten()              # Use target IDs as indices
]

# Reshape to match original structure
target_probas_reshaped = target_probas.reshape(targets.shape)
print(f"Target probabilities: {target_probas_reshaped}")
```

Step 4: Calculate Cross-Entropy Loss

```
# Apply negative logarithm and average
log_probas = torch.log(target_probas_reshaped)
neg_log_probas = -log_probas
loss = neg_log_probas.mean()

print(f"Log probabilities: {log_probas}")
print(f"Negative log probabilities: {neg_log_probas}")
print(f"Cross-entropy loss: {loss.item()}")
```

Why Logarithms?

- **Mathematical Stability:** Prevents numerical underflow with very small probabilities
- **Computational Efficiency:** Converts products to sums for easier calculation
- **Optimization Properties:** Better gradient behavior during backpropagation
- **Information Theory:** Logarithms measure "surprise" or information content

3.4 PyTorch Cross-Entropy Function

While understanding the manual calculation is important, PyTorch provides an optimized implementation that handles edge cases and numerical stability:

```
import torch.nn.functional as F

# Manual calculation (what we just learned)
manual_loss = neg_log_probas.mean()

```

```
# PyTorch built-in function
# Note: Takes logits directly, not probabilities
pytorch_loss = F.cross_entropy(
    logits.flatten(0, 1),      # Reshape to [batch*seq, vocab]
    targets.flatten()          # Reshape to [batch*seq]
)

print(f"Manual calculation: {manual_loss:.6f}")
print(f"PyTorch function: {pytorch_loss:.6f}")
print(f"Difference: {abs(manual_loss - pytorch_loss):.10f}")
```

Advantages of PyTorch's Function:

- **Numerical Stability:** Uses log-sum-exp trick to prevent overflow/underflow
- **Efficiency:** Optimized C++ implementation with CUDA support
- **Convenience:** Takes logits directly, no need for manual softmax
- **Edge Case Handling:** Properly handles zero probabilities and extreme values
- **Memory Efficiency:** Avoids creating intermediate probability tensors

Best Practice: Always use PyTorch's built-in functions for training, but understand the underlying mathematics for debugging and theoretical comprehension.

4 Training Data Preparation and Management

4.1 Training vs. Validation Split Strategy

Proper data splitting is crucial for monitoring model generalization and preventing overfitting. The validation set serves as an independent measure of model performance throughout training.

Split Rationale (Reference: Raschka p. 162):

- **Training Set (90%):** Used for gradient calculation and weight updates
- **Validation Set (10%):** Used only for performance monitoring, never for training
- **Temporal Ordering:** Validation data typically comes from later time periods to simulate real-world deployment

Implementation:

```
# Split the text data
total_size = len(text_data)
train_ratio = 0.90
split_idx = int(train_ratio * total_size)

train_data = text_data[:split_idx]
val_data = text_data[split_idx:]

print(f"Training characters: {len(train_data):,}")
print(f"Validation characters: {len(val_data):,}")
print(f"Split ratio: {len(train_data)/len(text_data):.2%}")

# Verify the split preserves data integrity
assert len(train_data) + len(val_data) == len(text_data)
assert train_data[-10:] + val_data[:10] == text_data[split_idx-10:split_idx+10]
```

Monitoring Generalization:

- **Good Generalization:** Validation loss decreases alongside training loss
- **Overfitting:** Training loss continues decreasing while validation loss increases
- **Underfitting:** Both losses remain high and plateau early

4.2 Efficient Data Loader Implementation

Data loaders handle the complex task of converting raw text into batched tensors suitable for training. Proper implementation is crucial for training efficiency and memory management.

Key Parameters:

- **batch_size:** Number of sequences processed simultaneously (affects memory usage)
- **max_length:** Maximum sequence length (context window)
- **stride:** Overlap between consecutive sequences (prevents context loss)
- **shuffle:** Randomize order for training (True), preserve order for validation (False)

Data Loader Creation:

```
# Configuration for training
batch_size = 2      # Small for learning purposes, typically 16-128 in practice
max_length = 256    # Context window size
stride = 128        # 50% overlap between sequences

# Create training data loader
train_loader = create_dataloader_v1(
    train_data,
    batch_size=batch_size,
    max_length=max_length,
    stride=stride,
    shuffle=True  # Randomize for better generalization
)

# Create validation data loader
val_loader = create_dataloader_v1(
    val_data,
    batch_size=batch_size,
    max_length=max_length,
    stride=stride,
    shuffle=False  # Preserve order for consistent evaluation
)

# Inspect data loader output
for batch in train_loader:
    print(f"Batch shape: {batch.shape}")  # [batch_size, max_length]
    break
```

Memory and Performance Considerations:

- **Batch Size Trade-offs:** Larger batches use more memory but provide more stable gradients
- **Sequence Length:** Longer sequences capture more context but require quadratically more attention computation
- **Stride Strategy:** Smaller strides increase data coverage but create more computational overhead

5 Training Loop Implementation

5.1 Complete Training Loop Architecture

The training loop is the heart of the learning process, orchestrating forward passes, loss calculation, backpropagation, and weight updates in a systematic cycle.

Training Loop Components:

1. **Data Loading:** Fetch batches from training data loader
2. **Forward Pass:** Compute model predictions and loss
3. **Backward Pass:** Calculate gradients via backpropagation
4. **Optimization:** Update model weights using computed gradients
5. **Monitoring:** Track performance on validation data
6. **Logging:** Record metrics for analysis and visualization

Training Loop Structure:

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

    Args:
        model: The language model to train
        train_loader: DataLoader for training data
        val_loader: DataLoader for validation data
        optimizer: Optimization algorithm (e.g., AdamW)
        device: Computing device (CPU/GPU)
        num_epochs: Number of training epochs
        eval_freq: How often to evaluate on validation set
        eval_iter: Number of batches for validation evaluation
        start_context: Text prompt for generation during training
    """

    # Tracking variables
    train_losses, val_losses, track_tokens_seen = [], [], []
    tokens_seen, global_step = 0, -1

    # 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()  # Clear previous gradients

            # Move data to device
            input_batch = input_batch.to(device)
            target_batch = target_batch.to(device)

            # Forward pass
            logits = model(input_batch)
            loss = torch.nn.functional.cross_entropy(
                logits.flatten(0, 1),
                target_batch.flatten()
            )

            # Backward pass
            loss.backward()  # Compute gradients
```

```
            optimizer.step()  # Update weights

            # Update tracking
            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}, Val loss {val_loss:.3f}")

    return train_losses, val_losses, track_tokens_seen
```

5.2 Optimizer Selection: AdamW

The choice of optimizer significantly impacts training dynamics and final model performance. AdamW (Adam with Weight Decay) is the standard choice for transformer model training.

AdamW Advantages:

- **Adaptive Learning Rates:** Different learning rates for each parameter based on gradient history
- **Momentum:** Accumulates gradient direction over time for smoother optimization
- **Weight Decay:** Proper L2 regularization to prevent overfitting
- **Stability:** Robust performance across different model sizes and datasets

AdamW Setup:

```
import torch.optim as optim

# Configure AdamW optimizer
optimizer = optim.AdamW(
    model.parameters(),
    lr=0.0004,           # Learning rate (typical range: 1e-5 to 1e-3)
    weight_decay=0.1     # L2 regularization strength
)

# Learning rate scheduling (optional but recommended)
scheduler = optim.lr_scheduler.CosineAnnealingLR(
    optimizer,
    T_max=num_epochs,    # Period of cosine decay
    eta_min=1e-6         # Minimum learning rate
)
```

Key Hyperparameters (Reference: Raschka p. 166):

Parameter	Typical Range	Our Choice
Learning Rate	1e-5 to 1e-3	4e-4
Weight Decay	0.01 to 0.3	0.1
Beta1	0.9 to 0.95	0.9
Beta2	0.95 to 0.999	0.999

5.3 Training Progress Monitoring

Effective monitoring during training provides insights into model behavior and helps identify issues early in the training process.

Evaluation Function:

```python
def evaluate_model(model, train_loader, val_loader, device, eval_iter):
    """
    Evaluate model performance on training and validation sets

    Args:
        model: Model to evaluate
        train_loader: Training data loader
        val_loader: Validation data loader
        device: Computing device
        eval_iter: Number of batches to evaluate

    Returns:
        train_loss: Average loss on training set
        val_loss: Average loss on validation set
    """
    model.eval()  # Set to evaluation mode

    with torch.no_grad():  # Disable gradient computation
        train_loss = calc_loss_loader(train_loader, model, device,
                                      num_batches=eval_iter)
        val_loss = calc_loss_loader(val_loader, model, device,
                                    num_batches=eval_iter)

    model.train()  # Return to training mode
    return train_loss, val_loss

def calc_loss_loader(data_loader, model, device, num_batches=None):
    """Calculate average loss over specified number of batches"""
    total_loss = 0.0
    if num_batches is None:
        num_batches = len(data_loader)
    else:
        num_batches = min(num_batches, len(data_loader))

    for i, (input_batch, target_batch) in enumerate(data_loader):
        if i >= num_batches:
            break

        input_batch = input_batch.to(device)
        target_batch = target_batch.to(device)

        logits = model(input_batch)
        loss = torch.nn.functional.cross_entropy(
            logits.flatten(0, 1),
            target_batch.flatten()
        )
        total_loss += loss.item()

    return total_loss / num_batches
```

Key Monitoring Metrics:

- **Training Loss:** Primary optimization target (should consistently decrease)
- **Validation Loss:** Generalization indicator (should track training loss)
- **Generation Quality:** Periodic text samples to assess coherence

- **Training Speed:** Tokens processed per second for efficiency tracking

6 Advanced Text Generation Strategies

6.1 Beyond Deterministic Generation

Simple text generation uses greedy decoding—always selecting the token with highest probability. While deterministic and fast, this approach often produces repetitive and boring text. Advanced sampling strategies introduce controlled randomness for more diverse and engaging outputs.

Limitations of Greedy Decoding:

- **Repetition:** Tends to produce repetitive phrases and ideas
- **Lack of Creativity:** Always chooses the "safest" option
- **Exposure Bias:** Training uses teacher forcing, but generation is autoregressive
- **Local Optima:** May get stuck in suboptimal continuation paths

6.2 Temperature Sampling: Controlling Creativity

Temperature sampling modifies the probability distribution before sampling, allowing fine-grained control over generation randomness and creativity.

Mathematical Foundation: Temperature scaling adjusts logits before applying softmax:

$$\text{Softmax}_T(z_i) = \frac{e^{z_i/T}}{\sum_{j=1}^{V} e^{z_j/T}}$$

Where T is the temperature parameter controlling the "sharpness" of the distribution.

Temperature Effects:

- **T ¡ 1 (Conservative):** Sharpens distribution, more deterministic
  - T = 0.1: Very focused, minimal creativity
  - T = 0.5: Somewhat focused, reduced randomness
- **T = 1 (Default):** Original model distribution, balanced
- **T ¿ 1 (Creative):** Flattens distribution, more random
  - T = 1.5: Increased creativity, some randomness
  - T = 2.0: High creativity, significant randomness

Temperature Sampling Implementation:

```
def generate_with_temperature(model, tokenizer, prompt, max_tokens=50,
                              temperature=1.0, context_size=1024):
    """
    Generate text using temperature sampling

    Args:
        model: Trained language model
        tokenizer: Text tokenizer
        prompt: Starting text
        max_tokens: Maximum tokens to generate
        temperature: Sampling temperature
        context_size: Maximum context length
```

```
    Returns:
        Generated text string
    """
    model.eval()

    # Convert prompt to tokens
    context = torch.tensor(tokenizer.encode(prompt)).unsqueeze(0)

    with torch.no_grad():
        for _ in range(max_tokens):
            # Truncate context if too long
            context_trunc = context[:, -context_size:]

            # Get logits from model
            logits = model(context_trunc)

            # Apply temperature scaling
            logits = logits[:, -1, :] / temperature

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

            # Sample next token
            next_token = torch.multinomial(probabilities, num_samples=1)

            # Append to context
            context = torch.cat([context, next_token], dim=1)

    # Convert back to text
    return tokenizer.decode(context.squeeze(0).tolist())
```

Temperature Selection Guidelines:

Use Case	Temperature	Characteristics
Technical Writing	0.1 - 0.3	Focused, accurate, minimal creativity
News Articles	0.3 - 0.7	Balanced accuracy and readability
Creative Writing	0.7 - 1.2	Good balance of coherence and creativity
Poetry/Art	1.0 - 2.0	High creativity, experimental language

6.3 Top-k Sampling: Focused Creativity

Top-k sampling combines the benefits of probabilistic sampling with quality control by restricting sampling to the k most likely tokens.

Algorithm:

1. Compute token probabilities from model logits
2. Sort tokens by probability (highest to lowest)
3. Keep only the top-k most likely tokens
4. Set probabilities of other tokens to zero
5. Renormalize remaining probabilities to sum to 1
6. Sample from this restricted distribution

Top-k Implementation:

```python
def sample_top_k(logits, k=10, temperature=1.0):
    """
    Sample from top-k most likely tokens

    Args:
        logits: Model output logits [vocab_size]
        k: Number of top tokens to consider
        temperature: Sampling temperature

    Returns:
        Sampled token ID
    """
    # Apply temperature scaling
    logits = logits / temperature

    # Get top-k logits and indices
    top_logits, top_indices = torch.topk(logits, k)

    # Convert to probabilities
    top_probabilities = torch.softmax(top_logits, dim=-1)

    # Sample from top-k
    sample_idx = torch.multinomial(top_probabilities, num_samples=1)

    # Return original token index
    return top_indices[sample_idx]
```

Optimal k Values:

- **k = 1:** Equivalent to greedy decoding (deterministic)
- **k = 10-20:** Good balance for most applications
- **k = 50-100:** More creative, suitable for creative writing
- **k = vocab_size:** Equivalent to pure temperature sampling

6.4 Combined Sampling Strategy

The most effective approach often combines temperature and top-k sampling to achieve both quality control and creative flexibility.

Advanced Generation Function:

```python
def generate_advanced(model, tokenizer, prompt, max_tokens=100,
                      temperature=1.0, top_k=50, context_size=1024):
    """
    Advanced text generation with temperature and top-k sampling

    Args:
        model: Trained language model
        tokenizer: Text tokenizer
        prompt: Starting prompt text
        max_tokens: Maximum number of tokens to generate
        temperature: Sampling temperature (controls randomness)
        top_k: Number of top tokens to consider
        context_size: Maximum context window size

    Returns:
        Generated text as string
    """
    model.eval()
```

```
    # Tokenize initial prompt
    context = torch.tensor(tokenizer.encode(prompt)).unsqueeze(0)

    generated_tokens = []

    with torch.no_grad():
        for _ in range(max_tokens):
            # Manage context window
            context_input = context[:, -context_size:]

            # Forward pass
            logits = model(context_input)
            logits = logits[:, -1, :]  # Get last position logits

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

            # Apply top-k filtering
            if top_k > 0:
                # Get top-k values and indices
                top_logits, top_indices = torch.topk(logits,
                                                     min(top_k, logits.size(-1)))

                # Create filtered logits tensor
                filtered_logits = torch.full_like(logits, float('-inf'))
                filtered_logits.scatter_(1, top_indices, top_logits)
                logits = filtered_logits

            # Convert to probabilities and sample
            probabilities = torch.softmax(logits, dim=-1)
            next_token = torch.multinomial(probabilities, num_samples=1)

            # Add to context and generated tokens
            context = torch.cat([context, next_token], dim=1)
            generated_tokens.append(next_token.item())

            # Stop if end-of-text token generated
            if next_token.item() == tokenizer.encode('<|endoftext|>')[0]:
                break

    # Decode generated tokens
    generated_text = tokenizer.decode(generated_tokens)
    return prompt + generated_text
```

Sampling Strategy Guidelines:

Application	Temperature	Top-k	Expected Quality
Code Generation	0.1	10	High precision, low creativity
Technical Writing	0.3	20	Accurate, professional
General Chat	0.7	40	Balanced, engaging
Creative Writing	1.0	60	Creative, diverse
Experimental Art	1.5	100	Highly creative, unpredictable

7 Loading and Using Pretrained Weights

7.1 The Pretraining Advantage

Training large language models from scratch requires enormous computational resources—millions of dollars and months of training time. Pretrained models offer a practical alternative by pro-

viding weights learned from massive datasets that can be fine-tuned for specific applications.

Benefits of Pretrained Models:

- **Cost Efficiency:** Avoid expensive pretraining computation
- **Time Savings:** Start with learned language representations
- **Performance:** Benefit from training on massive, diverse datasets
- **Accessibility:** Makes advanced AI capabilities available to smaller organizations

Transfer Learning Process:

1. **Pretraining:** Large-scale training on diverse text (e.g., web crawl data)
2. **Fine-tuning:** Adapt pretrained model to specific tasks or domains
3. **Evaluation:** Test performance on target applications
4. **Deployment:** Use fine-tuned model in production systems

7.2 OpenAI GPT-2 Weights

OpenAI provides pretrained GPT-2 models in multiple sizes, offering a balance between performance and computational requirements.

Available Model Sizes:

Model	Parameters	Layers	Hidden Size
GPT-2 117M	117 million	12	768
GPT-2 345M	345 million	24	1024
GPT-2 762M	762 million	36	1280
GPT-2 1542M	1.5 billion	48	1600

Model Configuration Alignment: Before loading pretrained weights, ensure your model configuration exactly matches the pretrained model architecture:

```
# Configuration for GPT-2 117M (small)
GPT_CONFIG_124M = {
    "vocab_size": 50257,      # Vocabulary size
    "context_length": 1024,   # Maximum sequence length
    "emb_dim": 768,           # Embedding dimension
    "n_heads": 12,            # Number of attention heads
    "n_layers": 12,           # Number of transformer layers
    "drop_rate": 0.1,         # Dropout rate
    "qkv_bias": False         # Query-key-value bias
}

# Verify configuration matches pretrained model
def verify_config(config, model_name="gpt2"):
    """Verify configuration matches pretrained model requirements"""
    if model_name == "gpt2":  # 117M model
        assert config["vocab_size"] == 50257
        assert config["context_length"] == 1024
        assert config["emb_dim"] == 768
        assert config["n_heads"] == 12
        assert config["n_layers"] == 12
        print("\checkmark\ Configuration verified for GPT-2 117M")
    else:
        raise ValueError(f"Unknown model: {model_name}")

verify_config(GPT_CONFIG_124M)
```

7.3 Weight Loading Process

Loading pretrained weights requires careful attention to tensor shapes, parameter names, and model architecture compatibility.

Weight Loading Implementation:

```
import requests
import json
import numpy as np
from tqdm import tqdm

def download_and_load_gpt2(model_size="124M", models_dir="gpt2"):
    """
    Download and load GPT-2 pretrained weights

    Args:
        model_size: Model size ("124M", "355M", "774M", "1558M")
        models_dir: Directory to store model files

    Returns:
        Dictionary containing model weights and configuration
    """

    # Create model directory
    os.makedirs(models_dir, exist_ok=True)

    # Download model files
    base_url = "https://openaipublic.blob.core.windows.net/gpt-2/models"

    for filename in ["checkpoint", "encoder.json", "hparams.json",
                     "model.ckpt.data-00000-of-00001", "model.ckpt.index",
                     "model.ckpt.meta", "vocab.bpe"]:

        file_url = f"{base_url}/{model_size}/{filename}"
        file_path = os.path.join(models_dir, filename)

        if not os.path.exists(file_path):
            print(f"Downloading {filename}...")
            response = requests.get(file_url)
            with open(file_path, "wb") as f:
                f.write(response.content)

    # Load weights from TensorFlow checkpoint
    import tensorflow as tf

    tf_ckpt_path = os.path.join(models_dir, "model.ckpt")
    tf_weights = {}

    for name, _ in tf.train.list_variables(tf_ckpt_path):
        array = tf.train.load_variable(tf_ckpt_path, name)
        tf_weights[name] = array

    return tf_weights

def load_weights_into_gpt(gpt_model, weights_dict):
    """
    Load pretrained weights into GPT model

    Args:
        gpt_model: PyTorch GPT model instance
        weights_dict: Dictionary of pretrained weights
    """

```

```
    # Weight mapping from TensorFlow to PyTorch naming
    weight_mapping = {
        "wte": "transformer.wte.weight",              # Token embeddings
        "wpe": "transformer.wpe.weight",              # Position embeddings
        "ln_f/g": "transformer.ln_f.weight",          # Final layer norm gamma
        "ln_f/b": "transformer.ln_f.bias",            # Final layer norm beta
    }

    # Load layer-specific weights
    for layer_idx in range(gpt_model.config["n_layers"]):
        # Attention weights
        weight_mapping.update({
            f"h{layer_idx}/attn/c_attn/w": f"transformer.h.{layer_idx}.attn.
                c_attn.weight",
            f"h{layer_idx}/attn/c_attn/b": f"transformer.h.{layer_idx}.attn.
                c_attn.bias",
            f"h{layer_idx}/attn/c_proj/w": f"transformer.h.{layer_idx}.attn.
                c_proj.weight",
            f"h{layer_idx}/attn/c_proj/b": f"transformer.h.{layer_idx}.attn.
                c_proj.bias",

            # Feed-forward weights
            f"h{layer_idx}/mlp/c_fc/w": f"transformer.h.{layer_idx}.mlp.c_fc.
                weight",
            f"h{layer_idx}/mlp/c_fc/b": f"transformer.h.{layer_idx}.mlp.c_fc.
                bias",
            f"h{layer_idx}/mlp/c_proj/w": f"transformer.h.{layer_idx}.mlp.
                c_proj.weight",
            f"h{layer_idx}/mlp/c_proj/b": f"transformer.h.{layer_idx}.mlp.
                c_proj.bias",

            # Layer normalization
            f"h{layer_idx}/ln_1/g": f"transformer.h.{layer_idx}.ln_1.weight",
            f"h{layer_idx}/ln_1/b": f"transformer.h.{layer_idx}.ln_1.bias",
            f"h{layer_idx}/ln_2/g": f"transformer.h.{layer_idx}.ln_2.weight",
            f"h{layer_idx}/ln_2/b": f"transformer.h.{layer_idx}.ln_2.bias",
        })

    # Load weights into model
    with torch.no_grad():
        for tf_name, pytorch_name in weight_mapping.items():
            if tf_name in weights_dict:
                tf_tensor = torch.from_numpy(weights_dict[tf_name])

                # Handle weight transpose for linear layers
                if "c_attn/w" in tf_name or "c_proj/w" in tf_name or "c_fc/w"
                    in tf_name:
                    tf_tensor = tf_tensor.t()  # Transpose for PyTorch format

                # Get corresponding PyTorch parameter
                pytorch_param = gpt_model.state_dict()[pytorch_name]

                # Verify shapes match
                assert tf_tensor.shape == pytorch_param.shape, \
                    f"Shape mismatch for {pytorch_name}: {tf_tensor.shape} vs {
                        pytorch_param.shape}"

                # Copy weights
                pytorch_param.copy_(tf_tensor)

        print("\checkmark\ Pretrained weights loaded successfully")
```

7.4 Model Size and Performance Comparison

Understanding the trade-offs between different model sizes helps in selecting the appropriate pretrained model for specific applications.

Performance vs. Resource Trade-offs:

Model	Memory (GB)	Inference Speed	Quality	Use Case
GPT-2 117M	0.5	Very Fast	Good	Development, testing
GPT-2 345M	1.4	Fast	Better	Production apps
GPT-2 762M	3.1	Medium	Very Good	High-quality applications
GPT-2 1542M	6.2	Slower	Excellent	Research, premium apps

Model Selection Guidelines:

- **Prototyping:** Start with 117M for fast iteration
- **Production:** Use 345M for most commercial applications
- **High Quality:** Choose 762M for demanding tasks
- **Research:** Use 1542M for maximum performance

7.5 Pretrained Model Capabilities

Pretrained GPT-2 models demonstrate remarkable capabilities across various language tasks without task-specific fine-tuning.

Zero-Shot Capabilities:

```
# Test pretrained model capabilities
test_prompts = [
    "Translate to French: Hello, how are you?",
    "Q: What is the capital of France? A:",
    "Write a Python function to calculate factorial:",
    "Complete the sentence: The weather today is",
    "Summarize: Large language models are AI systems..."
]

def test_model_capabilities(model, tokenizer, prompts, max_tokens=50):
    """Test various capabilities of pretrained model"""

    model.eval()
    results = {}

    for prompt in prompts:
        generated = generate_advanced(
            model=model,
            tokenizer=tokenizer,
            prompt=prompt,
            max_tokens=max_tokens,
            temperature=0.7,
            top_k=40
        )

        results[prompt] = generated
        print(f"Prompt: {prompt}")
        print(f"Generated: {generated}")
        print("-" * 80)

    return results

```

```
# Run capability tests
capabilities = test_model_capabilities(model, tokenizer, test_prompts)
```

Observed Capabilities:

- **Language Translation:** Basic translation between common languages
- **Question Answering:** Factual responses to straightforward questions
- **Code Generation:** Simple programming tasks and code completion
- **Text Completion:** Coherent continuation of various text types
- **Summarization:** Condensing longer texts into key points

8 Practical Applications and Deployment Considerations

8.1 Fine-tuning vs. Pretraining Decisions

Understanding when to use pretrained models versus training from scratch is crucial for project success and resource management.

Use Pretrained Models When:

- **Limited Computational Resources:** Budget constraints on training infrastructure
- **General Language Tasks:** Applications requiring broad language understanding
- **Fast Development Cycles:** Rapid prototyping and iteration needs
- **Standard Domains:** Working with common text types (news, conversations, etc.)

Consider Training from Scratch When:

- **Specialized Domains:** Highly technical or niche vocabularies
- **Privacy Requirements:** Sensitive data that cannot use external models
- **Specific Architectures:** Novel model designs not available pretrained
- **Ultimate Performance:** Maximum quality requirements with unlimited resources

Hybrid Approaches:

- **Domain Adaptation:** Continue pretraining on domain-specific data
- **Task-Specific Fine-tuning:** Adapt pretrained models to specific applications
- **Progressive Training:** Start with pretrained weights, then fine-tune with custom data

8.2 Ethical Considerations and Responsible Deployment

Deploying language models responsibly requires understanding their limitations and implementing appropriate safeguards.

Key Ethical Concerns:

- **Bias and Fairness:** Models reflect biases present in training data
- **Misinformation:** Potential for generating false or misleading content
- **Privacy:** Risk of reproducing sensitive information from training data

- **Transparency:** Users should understand they're interacting with AI systems

Mitigation Strategies:

- **Content Filtering:** Implement post-generation filtering for harmful content
- **User Education:** Clearly communicate AI limitations and appropriate usage
- **Monitoring Systems:** Track model outputs for quality and safety issues
- **Regular Evaluation:** Continuously assess model performance across diverse scenarios

9 Chapter 5 Summary and Key Takeaways

Essential Insights from Chapter 5:

1. **Objective Evaluation:** Cross-entropy loss provides a mathematical foundation for measuring and improving language model performance, bridging the gap between subjective text quality and quantitative optimization.
2. **Training Fundamentals:** Successful language model training requires careful orchestration of data preparation, loss calculation, optimization, and monitoring—each component crucial for learning language patterns.
3. **Advanced Generation:** Simple greedy decoding produces repetitive text; temperature and top-k sampling enable controllable creativity while maintaining coherence and quality.
4. **Pretrained Power:** Leveraging pretrained weights democratizes access to sophisticated language capabilities, enabling smaller organizations to build powerful applications without massive computational investments.
5. **Practical Deployment:** Real-world applications require balancing model capabilities with computational constraints, ethical considerations, and user expectations.

Technical Mastery Achieved:

- Understanding cross-entropy loss calculation and its role in optimization
- Implementing complete training loops with proper monitoring and evaluation
- Applying advanced sampling strategies for controlled text generation
- Loading and utilizing pretrained model weights effectively
- Making informed decisions about model selection and deployment strategies

Conceptual Understanding Developed:

- The relationship between mathematical optimization and language understanding
- How training transforms random parameters into meaningful representations
- The trade-offs between deterministic and probabilistic generation approaches
- The value proposition and limitations of transfer learning in NLP
- Ethical responsibilities in deploying language generation systems

10 Exercises and Practice Problems

Exercise 1: Loss Calculation Analysis Implement manual cross-entropy loss calculation for the following scenarios and compare with PyTorch's built-in function:

1. Calculate loss for a model that assigns uniform probability to all tokens
2. Calculate loss for a model with perfect next-token prediction
3. Analyze how loss changes as model predictions improve during training
4. Plot the relationship between prediction confidence and resulting loss values

Exercise 2: Training Loop Implementation Build a complete training loop with the following enhancements:

1. Add learning rate scheduling (cosine annealing or step decay)
2. Implement gradient clipping to prevent exploding gradients
3. Add early stopping based on validation loss plateaus
4. Include periodic model checkpointing for recovery
5. Implement training resume functionality from saved checkpoints

Exercise 3: Generation Strategy Comparison Compare different text generation strategies:

1. Generate 10 completions for the same prompt using different temperature values (0.1, 0.5, 1.0, 1.5, 2.0)
2. Test top-k sampling with k values of 1, 10, 50, and 100
3. Combine temperature and top-k sampling and evaluate the results
4. Design evaluation criteria for assessing generation quality
5. Analyze trade-offs between creativity and coherence

Exercise 4: Pretrained Model Analysis Explore pretrained model capabilities:

1. Load different sizes of GPT-2 (117M, 345M) and compare performance
2. Test zero-shot capabilities across various tasks
3. Measure inference speed and memory usage for different model sizes
4. Analyze the relationship between model size and output quality
5. Design benchmark tests for evaluating pretrained model performance

Exercise 5: Ethical AI Implementation Design responsible deployment strategies:

1. Identify potential biases in model outputs across different demographic groups
2. Implement content filtering systems for inappropriate or harmful text
3. Design user interfaces that clearly communicate AI limitations
4. Create monitoring systems for tracking model performance in production

5. Develop guidelines for appropriate use cases and applications

Next Class Preview: We'll dive deeper into the transformer architecture itself, building the attention mechanism from scratch and understanding how self-attention enables the powerful language modeling capabilities we've been utilizing.

Preparation for Lecture 6:

- **Raschka:** Chapter 3, Sections 3.1-3.5 (Pages 51-89) - Attention mechanisms
- **Review:** Linear algebra concepts (matrix multiplication, dot products)
- **Supplementary:** "Attention Is All You Need" paper introduction and overview