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

Prof. Rongyu Lin

Lecture 6
Quinnipiac University
School of Computing and Engineering

Reading: Raschka Ch. 5 (pp. 150-189)
"Model Training Pipeline and Pre-training"

Learning Objectives

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

- Calculate and interpret cross-entropy loss for text generation evaluation
- Implement complete training loops with proper monitoring and evaluation
- Apply advanced text generation techniques (temperature, top-k sampling)
- Load and integrate pretrained weights for transfer learning
- Understand the practical economics and challenges of LLM training
- Build end-to-end training pipelines from data preparation to deployment

From theory to practice: hands-on LLM training

Lecture Overview

Four Main Components:

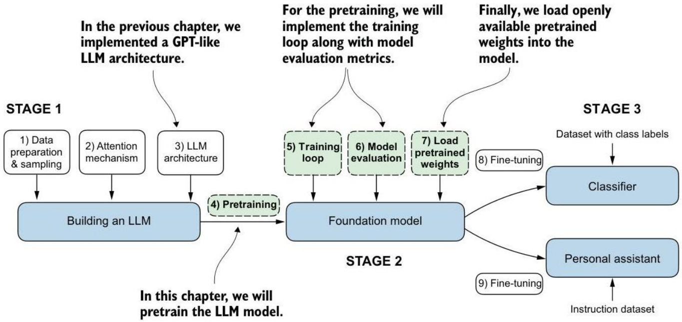
- **Model Evaluation & Loss:** Quantitative assessment of generation quality
- **Training Data & Loops:** Complete training pipeline implementation
- **Advanced Generation:** Temperature and top-k sampling strategies
- **Pretrained Weights:** Leveraging transfer learning for practical deployment

Hands-on Activities:

- Interactive loss calculation exercises
- Mini training loop implementation
- Parameter experimentation with text generation
- Loading and testing pretrained models

Theory → Implementation → Practice

Complete Training Pipeline Overview



Chapter 5 Focus: Stage 2 - Pre-training

- **Step 4:** Training code with loss calculation
- **Step 5:** Training loops with monitoring
- **Step 6:** Performance evaluation and quality assessment
- **Step 7:** Model weight management for deployment

From untrained model to functional LLM

Recap: Text Generation Process

From Chapter 4 - Five-Step Generation:

- **Step 1:** Encode input text into token IDs
- **Step 2:** Model processes tokens → generates logits
- **Step 3:** Convert logits to probabilities (softmax)
- **Step 4:** Sample next token from probability distribution
- **Step 5:** Decode token back to text, repeat process

Key Insight:

Before training, model generates random nonsense
After training, model generates coherent text

Today's Goal: *Transform random model into coherent generator*

Untrained Model Text Generation I

Before Training - Random Output:

```
# Random initialization produces incoherent output
untrained_model = GPTModel(GPT_CONFIG_124M)
start_context = "Hello, I am"

# Generate with untrained model
random_output = generate_text_simple(
    model=untrained_model,
    context=start_context,
    max_new_tokens=15,
)

print("Untrained output:", random_output)
# Output: "Hello, I am? Begins Vor nicht?
# ceremony? FLASH (): igua? booko?"

coherent_target = "Hello, I am a language model"
print("Target output:", coherent_target)
```

Why Random?

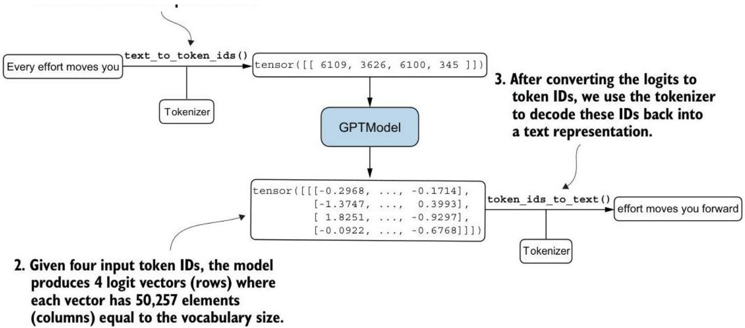
- Weights randomly initialized
- No learned patterns
- Pure statistical noise

Goal:

Random $\rightarrow$ *Coherent*

Training improves text quality dramatically

Text Generation Process - Technical Detail



Five-Step Process:

- **Tokenization:** Text → Token IDs → Tensor
- **Model Processing:** Tokens → Logits (probability scores)
- **Next Token Selection:** Highest probability or sampling
- **Detokenization:** Token ID → Text representation
- **Iteration:** Repeat until desired length or stop token

Foundation for loss calculation and training optimization

Utility Functions for Text Processing

```python
# Standardize input/output handling
def text_to_token_ids(text, tokenizer):
    """Convert raw text to model-ready tensor"""
    encoded = tokenizer.encode(text, allowed_special={'<|endoftext|>'})
    raw_text = torch.tensor(encoded).unsqueeze(0)
    return raw_text

def token_ids_to_text(token_ids, tokenizer):
    """Convert model output back to readable text"""
    flat = token_ids.squeeze(0)
    processed_output = tokenizer.decode(flat.tolist())
    return processed_output

# Enable consistent evaluation across inputs
def generate_and_print_sample(model, tokenizer, device,
                              start_context):
    """Generate text sample and print results"""
    model.eval()
    context_size = model.pos_emb.weight.shape[0]

    encoded = text_to_token_ids(start_context,
                                tokenizer).to(device)
    with torch.no_grad():
        token_ids = generate_text_simple(
            model=model, idx=encoded,
            max_new_tokens=50, context_size=context_size
        )
    decoded_text = token_ids_to_text(token_ids, tokenizer)
    print(f"Output: {decoded_text}")
```

Key Functions:

- Input standardization
- Batch dimension handling
- Evaluation mode setting
- Gradient computation control

Purpose:

Enable consistent evaluation and comparison

Usage:

- Pre-training assessment
- Post-training comparison
- Quality monitoring

CHUNK 1

Model Evaluation & Loss Calculation

Quantitative Assessment of Generation Quality

Why Do We Need Loss Calculation?

The Training Problem:

- Model generates text, but how do we know if it's "good"?
- Humans can judge quality, but we need automatic measurement
- Training requires numerical feedback to adjust weights
- Need objective metric to compare different model versions

Cross-Entropy Loss Solution:

- **Quantitative:** Single number representing prediction quality
- **Differentiable:** Can compute gradients for optimization
- **Interpretable:** Lower loss = better predictions
- **Scalable:** Works across entire datasets efficiently

Analogy: Like scoring a multiple-choice test automatically

Good predictions → Low loss → Better model

Cross-Entropy Loss: Intuitive Understanding

Simple Example:

Prediction Scenario

Context: "The weather is"
Target: "sunny"
Vocabulary: [sunny, rainy, cold, hot, ...]

Model Predictions:

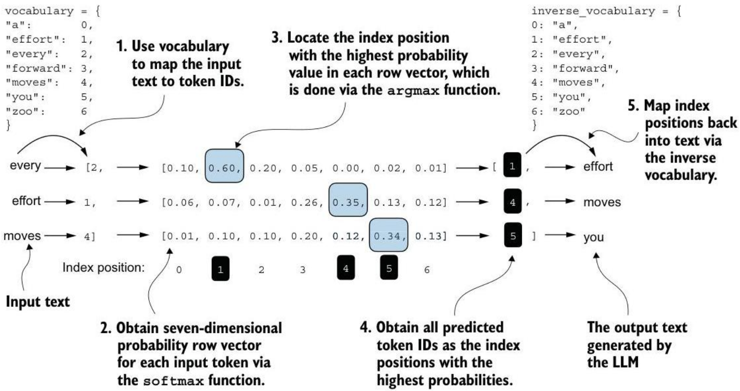
- **Good Model:** P("sunny") = 0.8, P("rainy") = 0.1, P("cold") = 0.05, ...
- **Bad Model:** P("sunny") = 0.1, P("rainy") = 0.3, P("xyzzy") = 0.2, ...

Cross-Entropy Calculation:

- **Good Model:** Loss = -log(0.8) = 0.22 (low loss)
- **Bad Model:** Loss = -log(0.1) = 2.30 (high loss)

Key Insight: Higher probability for correct answer → Lower loss

Loss Calculation: Six-Step Process



Six-Step Process:

- **Steps 1-3:** Calculate token probabilities (already completed in generation)
- **Step 4:** Apply logarithm to probabilities: log(P(correct_token))
- **Step 5:** Apply negative sign: -log(P(correct_token))
- **Step 6:** Average across all predictions in batch

Logits to Probabilities: Softmax Transformation

Logits
[2.1, 1.3, 0.8, -0.5, ...]

Softmax →

$$P_i = \frac{e^{logit_i}}{\sum_j e^{logit_j}}$$

Probabilities
[0.52, 0.31, 0.17, 0.01, ...]

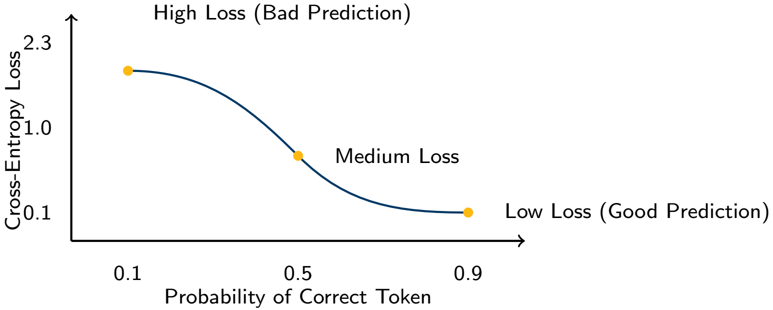
Properties:

- All values between 0 and 1
- Sum equals 1.0 (valid probability distribution)
- Higher logits $\rightarrow$ Higher probabilities
- Differentiable (essential for gradient computation)

Softmax converts raw scores to interpretable probabilities

Cross-Entropy: Mathematical Foundation

Information Theory Background:



Complete Cross-Entropy Formula:

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

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

Intuition: *Negative log probability "punishes" confident wrong predictions more than uncertain predictions*

Target Tensor Preparation

Before calculating loss, we need targets:

Input: "Every effort moves you" → [Every, effort, moves, you]

Shift by 1 position

Target: [effort, moves, you, ¡—endoftext—¿] (shifted by 1)

Why shift targets?

- Model predicts *next* token at each position
- Target[i] = Input[i+1] for training alignment
- Each input token predicts the following token
- Enables self-supervised learning from raw text

Every token becomes both input and target

Manual Loss Calculation - Step by Step

MANUAL APPROACH: Understanding the Math

```python
def calc_loss_manual(input_batch, target_batch, model, device):
    # Move to device
    input_batch = input_batch.to(device)
    target_batch = target_batch.to(device)

    # Forward pass
    logits = model(input_batch)

    # STEP 1: Flatten dimensions
    logits_flat = logits.flatten(0, 1)      # (batch * seq_len, vocab_size)
    targets_flat = target_batch.flatten()   # (batch * seq_len)

    # STEP 2: Convert to probabilities
    probabilities = torch.softmax(logits_flat, dim=1)

    # STEP 3: Select target probabilities
    target_probs = probabilities[range(len(targets_flat)), targets_flat]

    # STEP 4: Compute negative log probability
    log_probs = torch.log(target_probs + 1e-9)  # Add epsilon for stability

    # STEP 5: Average across all predictions
    loss = -log_probs.mean()

    return loss
```

- **Explicit:** Detailed 5-step process reveals mathematical foundations
- **Educational:** Each step clearly demonstrates cross-entropy calculation
- **Transparent:** Every operation visible for understanding and debugging
- **Trade-off:** Slower execution, but maximum learning value

PYTORCH APPROACH: Optimized for Production

```python
def calc_loss_pytorch(input_batch, target_batch, model, device):
    # Move to device
    input_batch = input_batch.to(device)
    target_batch = target_batch.to(device)

    # Forward pass
    logits = model(input_batch)

    # ONE STEP: CrossEntropyLoss handles all operations automatically
    # - Softmax transformation
    # - Log of probabilities
    # - Negative sign
    # - Averaging
    loss = torch.nn.functional.cross_entropy(
        logits.flatten(0, 1),
        target_batch.flatten())

    return loss

# Both methods produce identical results
print(f"Manual: {calc_loss_manual(batch, targets, model, device):.4f}")
print(f"PyTorch: {calc_loss_pytorch(batch, targets, model, device):.4f}")
```

- **Concise:** Single function call replaces multiple operations
- **Optimized:** Highly efficient implementation for production use
- **Stable:** Built-in numerical stability and edge case handling
- **Fast:** Significant performance improvement for large-scale training

Key Insight: *Both approaches yield identical results but serve different purposes*

Hands-On Activity: Loss Calculation

Calculate loss for your own examples!

Exercise: Loss Calculation

Context: "The cat sat on the" **Target:** "mat"

Model Predictions: P("mat") = 0.6, P("floor") = 0.2, P("chair") = 0.1, P("table") = 0.1

Your turn:

1. Calculate: $Loss = -\log(P(target)) = -\log(0.6) =$ ________
2. Good or bad prediction? ________

Discussion Questions:

- What if P("mat") = 0.1?
- How does loss relate to model confidence?

Chunk 1 Summary: Model Evaluation & Loss

Key Concepts Mastered:

- **Cross-entropy loss:** Quantitative measure of prediction quality
- **Loss calculation:** Six-step process from logits to final loss value
- **Target preparation:** Shift input tokens by one position
- **Implementation:** Both manual and PyTorch approaches work identically

Practical Skills:

- Calculate loss manually for understanding
- Use PyTorch functions for efficiency
- Interpret loss values for model quality assessment
- Prepare data correctly for training

Next: Use loss calculation to build complete training loops

CHUNK 2

Training Data & Training Loops

Complete LLM Training Infrastructure

Training vs Validation Data Split

Complete Dataset
"The Verdict" by Edith Wharton (20,479 tokens)

90/10 Split

Training Set (90%)
18,431 tokens

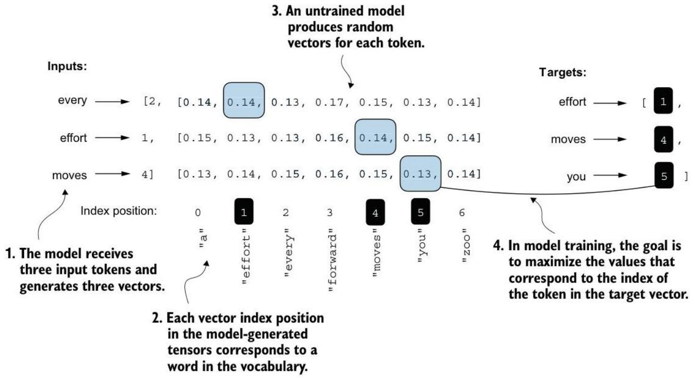
Validation Set (10%)
2,048 tokens

Used for:
Weight updates
Gradient computation
Model optimization

Used for:
Performance monitoring
Overfitting detection
Model selection

Why Split? *Validation data provides unbiased estimate of model performance*

Data Loader Architecture



Data Loader Pipeline:

- **Step 1:** Split text into training and validation portions
- **Step 2:** Tokenize text into numerical representations
- **Step 3:** Divide into chunks of specified length (context size)
- **Step 4:** Shuffle rows to prevent overfitting to sequence order
- **Step 5:** Organize into batches for efficient processing

Creating Training DataLoaders - Part 1

```python
# Process entire dataset efficiently
from torch.utils.data import DataLoader

# Read the training data
with open("the-verdict.txt", "r", encoding="utf-8") as f:
    raw_text = f.read()

# Split into training and validation sets
split_idx = int(0.90 * len(raw_text))  # 90% for training
train_data = raw_text[:split_idx]
val_data = raw_text[split_idx:]

print(f"Training characters: {len(train_data):,}")
print(f"Validation characters: {len(val_data):,}")

# Create dataset objects for batch processing
train_dataset = GPTDatasetV1(
    train_data,
    tokenizer,
    GPT_CONFIG_124M["context_length"],
    GPT_CONFIG_124M["context_length"]
)
```

Data Preparation:

- Read raw text file
- Split 90
- Create dataset objects

Dataset Parameters:

- **Text data:** Raw characters
- **Tokenizer:** Convert to tokens
- **Context length:** Sequence size

Creating Training DataLoaders - Part 2

```python
val_dataset = GPTDatasetV1(
    val_data,
    tokenizer,
    GPT_CONFIG_124M["context_length"],
    GPT_CONFIG_124M["context_length"]
)

# Create data loaders for efficient batch processing
train_loader = DataLoader(
    train_dataset,
    batch_size=2,
    shuffle=True,
    drop_last=True)
val_loader = DataLoader(
    val_dataset,
    batch_size=2,
    shuffle=False,
    drop_last=False)

print(f"Training batches: {len(train_loader)}")
print(f"Validation batches: {len(val_loader)}")

# Test data loader
for batch_idx, (input_batch, target_batch) in enumerate(train_loader):
    print(f"Batch {batch_idx}: Input shape {input_batch.shape}")
    if batch_idx >= 2:  # Show first few batches only
        break
```

Key Parameters:

- **batch_size:** 2 (small for demo)
- **shuffle:** True for training
- **drop_last:** Handle incomplete batches

Data Flow:

- Raw text $\rightarrow$ Tokens
- Tokens $\rightarrow$ Chunks
- Chunks $\rightarrow$ Batches
- Batches $\rightarrow$ Model

Memory Efficiency:

Load only needed batches, not entire dataset

Loss Calculation Function - Part 1

```python
# Process entire dataset efficiently
def calc_loss_batch(input_batch, target_batch, model, device):
    """Calculate loss for a single batch"""
    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()
    )
    return loss

def calc_loss_loader(data_loader, model, device,
                     num_batches=None):
    """Accumulate loss across batches for entire dataset"""
    total_loss = 0.0
    batch_count = 0

    if num_batches is None:
        num_batches = len(data_loader)
    else:
        num_batches = min(num_batches, len(data_loader))
```

Single Batch Function:

- Move tensors to correct device
- Run forward pass through model
- Calculate cross-entropy loss
- Return loss tensor for this batch

Full Dataset Function:

- Initialize accumulators
- Handle optional batch limits
- Prepare for batch iteration

Loss Calculation Function - Part 2

```python
    # Continue from previous frame
    for i, (input_batch, target_batch) in enumerate(data_loader):
        if i < num_batches:
            loss = calc_loss_batch(input_batch, target_batch,
                                   model, device)
            total_loss += loss.item()  # Convert tensor to float
            batch_count += 1
        else:
            break

    average_loss = total_loss / batch_count
    return average_loss

# Usage example
train_loss = calc_loss_loader(train_loader, model, device)
val_loss = calc_loss_loader(val_loader, model, device)

print(f"Training loss: {train_loss:.4f}")
print(f"Validation loss: {val_loss:.4f}")
```

Implementation Details:

- Process each batch
- Accumulate losses
- Convert tensor to scalar
- Calculate average loss

Key Features:

- Device handling (GPU/CPU)
- Memory-efficient processing
- Partial dataset evaluation
- Average across batches

Usage:

- Training progress monitoring
- Validation evaluation
- Overfitting detection

Model Evaluation Function - Part 1

```python
# Disable training mode for consistent evaluation
def evaluate_model(model, train_loader, val_loader, device, eval_iter):
    """Evaluate model on both training and validation sets"""
    model.eval()  # Set to evaluation mode - disables dropout, etc.
    with torch.no_grad():  # No gradient computation saves memory
        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)
    # Return to training mode for continued optimization
    model.train()
    return train_loss, val_loss
```

Evaluation Protocol:

- Switch to eval() mode
- Disable gradients
- Process limited batches
- Switch back to train()

Why eval() mode?

- Consistent behavior
- Disables dropout
- Batch norm uses running stats
- Reproducible results

Model Evaluation Function - Part 2

```python
# Usage during training loop
eval_freq = 5      # Evaluate every 5 epochs
eval_iter = 1      # Use 1 batch for quick evaluation during training

for epoch in range(num_epochs):
    # ... training code here ...
    if epoch % eval_freq == 0:
        train_loss, validation_loss = evaluate_model(
            model, train_loader, val_loader, device, eval_iter)
        print(f"Ep {epoch:03d}: "
              f"Train loss {train_loss:.4f}, "
              f"Val loss {validation_loss:.4f}")

        # Generate text sample for qualitative assessment
        generate_and_print_sample(
            model, tokenizer, device, "Every effort moves you")
```

Monitoring Strategy:

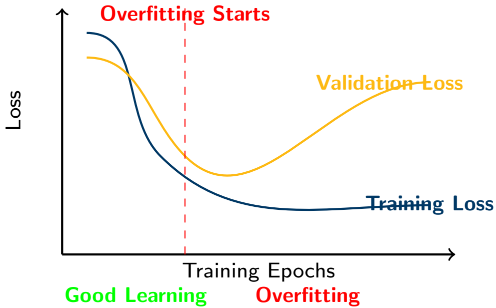
- Quantitative: Loss values
- Qualitative: Text samples
- Regular intervals
- Early stopping signals

Implementation:

- Evaluate every 5 epochs
- Limited batch processing
- Print metrics for tracking
- Generate sample text for quality check

Understanding Overfitting in LLMs

Training vs Validation Loss Patterns:



Signs of Overfitting:

- Training loss continues decreasing
- Validation loss starts increasing or plateaus
- Model memorizes training data
- Poor generalization to new text

Solutions: *Early stopping, regularization, more diverse training data*

Optimizer Setup - Part 1

```python
import torch.optim as optim

# AdamW optimizer - state-of-the-art for transformer training
optimizer = optim.AdamW(model.parameters(),
                        lr=0.0004,          # Learning rate - key hyperparameter
                        weight_decay=0.1)    # L2 regularization to prevent overfitting
# Learning rate scheduler (optional but recommended)
from torch.optim.lr_scheduler import CosineAnnealingLR
scheduler = CosineAnnealingLR(optimizer,
                              T_max=num_epochs,    # Maximum epochs
                              eta_min=1e-6)        # Minimum learning rate

print(f"Optimizer: {optimizer.__class__.__name__}")
print(f"Learning rate: {optimizer.param_groups[0][’lr’]}")
print(f"Weight decay: {optimizer.param_groups[0][’weight_decay’]}")
```

AdamW Benefits:

- Adaptive learning rates
- Momentum for smooth updates
- Effective weight decay
- Proven for transformers

Hyperparameters:

- **lr=0.0004:** Conservative but stable
- **weight_decay=0.1:** Regularization strength

Scheduler:

- Gradually decreases learning rate
- Improves convergence
- Prevents plateaus

Optimizer Setup - Part 2

```python
# Number of trainable parameters
print(f"Number of parameters: {sum(p.numel() for p in model.parameters()):,}"
    )

# Gradient clipping setup (prevent exploding gradients)
max_grad_norm = 1.0  # Clip gradients above this norm

# Display current learning rate
def get_lr():
    for param_group in optimizer.param_groups:
        return param_group['lr']

print(f"Initial learning rate: {get_lr():.6f}")
```

Additional Features:

- **grad_clip:** Stability insurance
- Parameter counting for monitoring
- Learning rate tracking

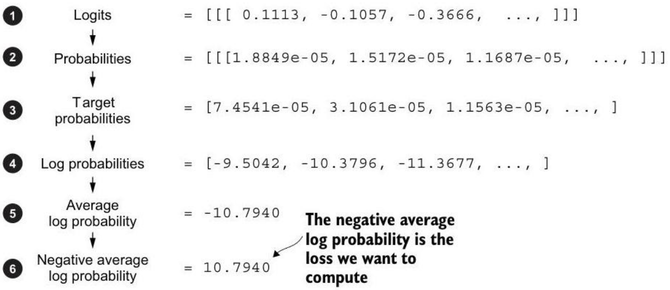
Learning Rate:

Too high $\rightarrow$ *unstable*
Too low $\rightarrow$ *slow learning*

Gradient Clipping:

- Prevents exploding gradients
- Stabilizes training
- Crucial for transformer models

Training Loop Structure



Eight-Step Training Process:

- **Step 1:** Iterate over epochs (full dataset passes)
- **Step 2:** Process each batch in training set
- **Step 3:** Reset gradients to zero (crucial!)
- **Step 4:** Forward pass - calculate loss
- **Step 5:** Backward pass - compute gradients
- **Step 6:** Update model weights using optimizer
- **Step 7:** Clip gradients if necessary (stability)
- **Step 8:** Update learning rate schedule (optional)

Complete Training Loop - Part 1

```python
def train_model_simple(model, train_loader, val_loader,
                       optimizer, device, num_epochs):
    train_losses, val_losses = [], []
    tokens_seen = 0

    for epoch in range(num_epochs):
        model.train()

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

            # Move to device and forward pass
            input_batch = input_batch.to(device)
            target_batch = target_batch.to(device)
            logits = model(input_batch)

            # Calculate loss and backpropagate
            loss = torch.nn.functional.cross_entropy(
                logits.flatten(0, 1), target_batch.flatten()
            )
            loss.backward()

            # Clip gradients and update weights
            torch.nn.utils.clip_grad_norm_(model.parameters(), 1.0)
            optimizer.step()
            tokens_seen += input_batch.numel()
```

Core Training Steps:

- Zero gradients
- Forward pass
- Compute loss
- Backward pass
- Clip gradients
- Update weights

Critical Details:

- **zero_grad():** Must clear old gradients
- **device:** Move tensors to GPU/CPU
- **grad_clip:** Prevent exploding gradients
- **tracking:** Monitor tokens seen

Debugging:

Print loss every few batches to monitor progress

Complete Training Loop - Part 2

```python
        # Evaluation and monitoring
        if epoch % eval_freq == 0:
            train_loss, val_loss = evaluate_model(
                model, train_loader, val_loader, device
            )
            train_losses.append(train_loss)
            val_losses.append(val_loss)

            print(f"Epoch {epoch}: Train {train_loss:.4f}, "
                  f"Val {val_loss:.4f}")

            # Generate sample for qualitative assessment
            model.eval()
            with torch.no_grad():
                sample = generate_text_simple(model, start_context, 30)
                print("Sample:", sample[:50])
            model.train()

    return train_losses, val_losses, tokens_seen

# Training execution
history = train_model_simple(model, train_loader, val_loader,
                             optimizer, device, num_epochs=10)
```

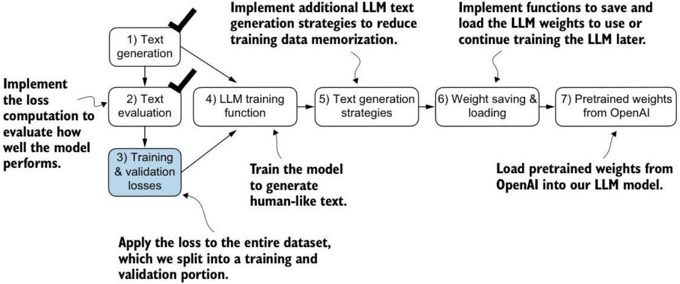
Key Features:

- Loss tracking & sample generation
- Progress reporting & metrics storage
- Regular evaluation intervals
- Training curves & performance data

Training Result:

Complete training history ready for analysis

Training Progress Monitoring



Interpreting Training Curves:

- **Initial Phase:** Both losses decrease rapidly (good learning)
- **Middle Phase:** Training loss continues down, validation plateaus
- **Late Phase:** Training loss keeps improving, validation may increase
- **Optimal Point:** Where validation loss is minimized (epoch 2)

Key Insights:

- Model is learning effectively early on
- Overfitting begins around epoch 2
- Would benefit from early stopping

Common Training Challenges

Memory Issues
- GPU memory overflow
- Large batch sizes
- Long sequences
- Model size limits

Training Time
- Hours to days
- Expensive compute
- Progress monitoring
- Checkpointing needs

Convergence Issues
- Learning rate tuning
- Gradient problems
- Loss plateaus
- Unstable training

Solutions:
Gradient accumulation • Mixed precision • Learning rate scheduling • Gradient clipping
Regular checkpointing • Distributed training • Early stopping • Hyperparameter tuning

Practical Advice: *Start small, monitor closely, scale gradually*

Hands-On Activity: Mini Training Loop

Implement a 3-epoch training loop!

Exercise: Mini Training

Task: Write code to train model for 3 epochs

Requirements:

1. Use provided data loaders and model
2. Print loss every epoch
3. Generate text sample after each epoch
4. Track improvement in text quality

Observations to Make:

- Does loss decrease each epoch?
- How does generated text improve?
- What happens if you skip `zero_grad()`?

Starter Code Template:

- `for epoch in range(3):`
- `optimizer.zero_grad()`
- `loss.backward()`
- `optimizer.step()`

Chunk 2 Summary: Training Data & Loops

Infrastructure Mastered:

- **Data Management:** Train/validation split with efficient DataLoaders
- **Loss Calculation:** Batch processing across entire datasets
- **Training Loops:** Complete 8-step optimization process
- **Progress Monitoring:** Both quantitative metrics and qualitative samples

Practical Skills:

- Implement complete training pipelines
- Monitor training progress effectively
- Recognize and handle overfitting
- Debug training issues systematically

Training Results: *Model generates coherent text after just a few epochs!*

Next: Enhance text generation with advanced sampling techniques

CHUNK 3

Advanced Text Generation

Temperature & Top-k Sampling for Quality Control

The Problem with Greedy Generation

Current Text Generation Issue:

Greedy Decoding Problems

Method: Always select token with highest probability

Issues:

- **Repetitive:** Gets stuck in loops
- **Deterministic:** Same input $\rightarrow$ same output always
- **Memorization:** Reproduces training data verbatim
- **Boring:** Lacks creativity and diversity

Example:

Input: "The weather is"

Greedy: "The weather is nice. The weather is nice. The weather is nice..."

Solution Needed:

Controlled randomness to improve diversity while maintaining quality

Temperature Sampling: Controlling Creativity

Core Concept:

- **Temperature (T):** Controls randomness in token selection
- **Low T (0.1):** More focused, deterministic generation
- **High T (2.0):** More creative, random generation
- **T = 1.0:** Standard probability distribution

Temperature Effects:

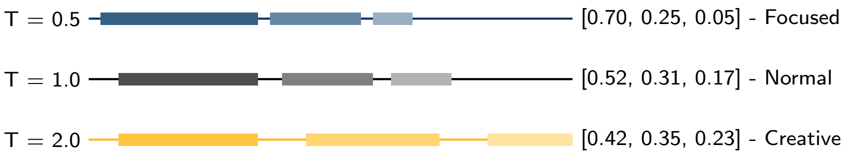
Temperature	Behavior	Use Case
0.1 - 0.5	Focused, conservative	Technical docs, factual text
0.7 - 1.0	Balanced creativity	General conversation
1.2 - 2.0	Creative, diverse	Creative writing, brainstorming
> 2.0	Highly random	Experimental, abstract text

Like adjusting the "creativity dial" on a writing assistant

Temperature: Mathematical Foundation

Formula: $P_i = \frac{e^{logit_i / T}}{\sum_j e^{logit_j / T}}$

Original Logits: [2.1, 1.3, 0.8]



Key Effects:

- **T** $<$ **1:** Sharpens distribution, more deterministic
- **T = 1:** Unchanged distribution
- **T** $>$ **1:** Flattens distribution, more random
- **T** $\rightarrow$ **0:** Approaches greedy (argmax)
- **T** $\rightarrow \infty$**:** Approaches uniform random

Top-k Sampling: Quality Control

The Problem:

- Temperature alone can still select very unlikely tokens
- High temperature may choose nonsensical words
- Need to limit vocabulary to reasonable options
- Balance diversity with quality

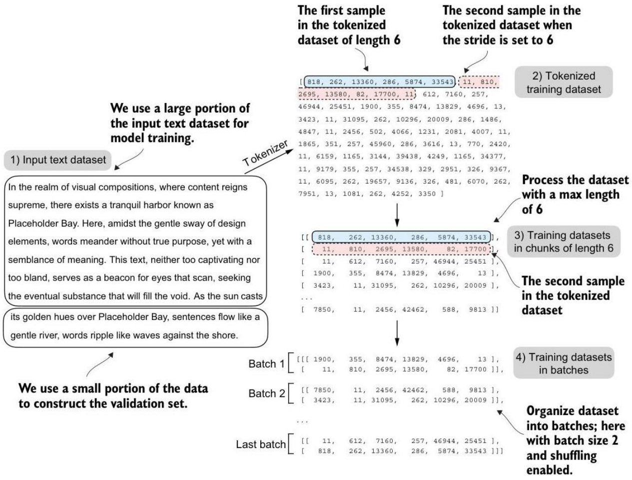
Top-k Solution:

- **Step 1:** Rank all tokens by probability
- **Step 2:** Keep only top k tokens (e.g., k=25)
- **Step 3:** Set other probabilities to 0
- **Step 4:** Renormalize remaining probabilities
- **Step 5:** Sample from filtered distribution

Analogy: Like multiple-choice question where you eliminate obviously wrong answers first

Common k values: k=10 (focused), k=25 (balanced), k=50 (diverse)

Top-k Sampling: Step-by-Step Algorithm



Five-Step Process:

- **Step 1:** Start with logits from model

Combining Temperature and Top-k Sampling

Best of Both Worlds:

Logits → Apply Temperature ($T = 1.4$) → Top-k Filter ($k = 25$) → Sample

Combined Benefits:

- Controlled creativity from temperature
- Quality assurance from top-k filtering
- Prevents both boring and nonsensical output
- Tunable for different applications

Recommended Combinations:

- **Conservative:** T=0.8, k=10 (technical writing)
- **Balanced:** T=1.2, k=25 (general conversation)
- **Creative:** T=1.8, k=50 (creative writing)

Advanced Generate Function Implementation (Part 1)

```python
def generate_text_advanced(model, idx, max_new_tokens, context_size,
                          temperature=1.0, top_k=None, eos_id=None):
    """
    Enhanced text generation with temperature and top-k sampling

    Args:
        temperature: Controls creativity (0.1=focused, 2.0=creative)
        top_k: Keep only top k tokens (None=no filtering)
        eos_id: End-of-sequence token ID for early stopping
    """
    model.eval()

    for _ in range(max_new_tokens):
        # Crop context to fit model's context window
        idx_cond = idx[:, -context_size:]

        with torch.no_grad():
            # Get logits from model
            logits = model(idx_cond)
            logits = logits[:, -1, :]  # Focus on last time step
```

Key Features:

- Temperature scaling for creativity control
- Top-k filtering for quality assurance
- Early stopping with EOS tokens

Function Overview:

- Sets up generation loop
- Handles context window
- Gets logits from model

Advanced Generate Function Implementation (Part 2)

```python
# Apply temperature scaling - controls creativity
if temperature > 0.0:
    logits = logits / temperature_param

    # Apply top-k filtering if specified - ensures quality
    if top_k is not None:
        # Keep only top-k most probable tokens
        top_k_value = min(top_k, logits.size(-1))
        top_k_logits, top_k_indices = torch.topk(logits, top_k_value)

        # Set non-top-k logits to negative infinity
        filtered_probabilities = torch.full_like(logits, float('-inf'))
        filtered_probabilities.scatter_(1, top_k_indices, top_k_logits)
        logits = filtered_probabilities

    # Sample proportionally to probabilities
    probs = torch.softmax(logits, dim=-1)
    sampled_token = torch.multinomial(probs, num_samples=1)
else:
    # Temperature = 0: greedy decoding
    sampled_token = torch.argmax(logits, dim=-1, keepdim=True)

# Append sampled token to sequence
idx = torch.cat((idx, sampled_token), dim=1)

# Check for end-of-sequence token
if eos_id is not None and sampled_token.item() == eos_id:
```

Advanced Features:

- Multinomial sampling
- Fallback to greedy if T=0
- Token sequence building
- Early stopping detection

Parameter Effects:

- **temperature:** Creativity dial
- **top_k:** Quality filter
- **Both combined:** Optimal results

Usage Flexibility:

Adjust parameters for different text types and applications

Multinomial Sampling: Technical Details (Part 1)

```python
# Sample proportionally to probabilities
import torch

def demonstrate_multinomial_sampling():
    """Show how multinomial sampling works with examples"""

    # Example probability distribution
    probabilities = torch.tensor([[0.5, 0.3, 0.2]])  # 3 possible tokens
    token_names = ["sunny", "rainy", "cloudy"]

    print("Probability distribution:")
    for i, (name, prob) in enumerate(zip(token_names, probabilities[0])):
        print(f"  {name}: {prob:.1f}")

    # Sample multiple times to see distribution
    samples = []
    for _ in range(1000):
        sampled_indices = torch.multinomial(probabilities, num_samples=1)
        samples.append(sampled_indices.item())
```

Multinomial Benefits:

- Probabilistic sampling
- Respects token probabilities
- Introduces controlled randomness
- Prevents deterministic repetition

Sampling Process:

- Create probability distribution
- Draw weighted random samples
- Higher probabilities = more frequent
- Maintains distribution statistics

Multinomial Sampling: Technical Details (Part 2)

```python
    # Count occurrences
    from collections import Counter
    counts = Counter(samples)
    print("\nSampling results (1000 samples):")
    for i, name in enumerate(token_names):
        count = counts[i]
        observed_prob = count / 1000
        expected_prob = probabilities[0][i].item()
        print(f"  {name}: {count} times ({observed_prob:.3f} vs expected {expected_prob:.3f})")

# Reproducible sampling with random seeds
def reproducible_generation():
    """Generate text with consistent random behavior"""
    torch.manual_seed(42)  # Set random seed for reproducibility

    # Same parameters will always produce same output
    random_seed = 123
    torch.manual_seed(random_seed)
    probability_distribution = torch.softmax(torch.tensor([[2.1, 1.3, 0.8]]), dim=-1)
    sampled_token = torch.multinomial(probability_distribution, num_samples=1)
    return sampled_token

# Demonstrate
demonstrate_multinomial_sampling()
token = reproducible_generation()
print(f"Reproducible sample: {token}")
```

Statistical Analysis:

- Verify sampling distribution
- Count occurrences of each token
- Compare observed vs expected

Key Properties:

- More probable tokens chosen more often
- Less probable tokens still possible
- Reproducible with random seeds
- Computationally efficient

Sampling Strategies: Side-by-Side Comparison

Same prompt, different strategies:

Input: "The future of artificial intelligence"

Greedy (deterministic):

"The future of artificial intelligence is bright. The future of artificial intelligence is bright..."

Temperature = 0.8:

"The future of artificial intelligence looks promising, with advances in machine learning and deep neural networks opening new possibilities..."

Temperature = 1.5, Top-k = 25:

"The future of artificial intelligence might revolutionize how we approach complex problems, potentially transforming industries through innovative applications..."

Temperature = 2.0, Top-k = 10:

"The future of artificial intelligence could bring unexpected breakthroughs, perhaps leading to remarkable discoveries in computational creativity..."

Observations: *Higher temperature + top-k = more diverse yet coherent text*

Hands-On Activity: Parameter Experimentation

Experiment with different sampling parameters!

Exercise: Parameter Effects

Task: Generate text with different parameter combinations

Fixed Context: "In the year 2030,"

Try These Combinations:

1. Temperature=0.5, Top-k=10 (conservative)
2. Temperature=1.0, Top-k=25 (balanced)
3. Temperature=1.8, Top-k=50 (creative)
4. Temperature=2.5, Top-k=5 (experimental)

Evaluate:

- Which produces most coherent text?
- Which is most creative/diverse?
- What happens with extreme parameters?

Discussion: *How would you tune parameters for technical documentation vs creative writing?*

Chunk 3 Summary: Advanced Text Generation

Techniques Mastered:

- **Temperature Sampling:** Control creativity and randomness in generation
- **Top-k Sampling:** Maintain quality while allowing diversity
- **Combined Strategies:** Optimal results from parameter combination
- **Multinomial Sampling:** Probabilistic token selection implementation

Practical Applications:

- Tune parameters for different content types
- Avoid repetitive and boring outputs
- Balance coherence with creativity
- Generate diverse text from same model

Key Insight: *Simple parameter adjustments dramatically improve text quality*

Next: Leverage pretrained weights for instant quality improvement

CHUNK 4

Pretrained Weights & Transfer Learning

Practical Deployment with OpenAI GPT-2

The Pretraining Computational Challenge

Training Costs
GPT-2: $50K - $200K
GPT-3: $4.6M
GPT-4: $63M+
Time: Weeks to months

Compute Requirements
Hundreds of GPUs
Terabytes of data
Specialized infrastructure
Expert engineering teams

Barriers to Entry
Only tech giants can afford
Months of development time
Huge financial investment
Technical complexity

Solution: Pretrained Weights
Download in minutes
• Start with trained model • Focus on fine-tuning • Leverage others' investments

Economic Reality: *Training from scratch is prohibitively expensive for most organizations*

Transfer Learning Advantage: *Start with quality baseline, adapt for your needs*

Understanding Transfer Learning in LLMs

Transfer Learning Analogy:

Like learning to drive a truck after mastering a car
Basic skills transfer, specific adaptation needed

LLM Transfer Learning Process:

1. Pretrained Model
General language understanding
Trained on diverse internet text

→

2. Task Adaptation
Fine-tune for specific domain
Add task-specific capabilities

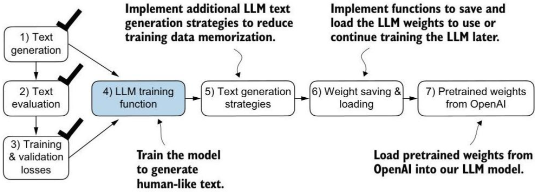
→

3. Deployment
Ready for production use
Specialized performance

Benefits: *Faster development • Lower costs • Better performance • Proven reliability*

Examples: *GPT-2 → ChatGPT, BERT → Search engines, LLaMA → Specialized chatbots*

GPT-2 Model Sizes and Capabilities



GPT-2 Model Variants:

- **GPT-2 124M:** 12 layers, 768 dimensions, 12 attention heads
- **GPT-2 355M:** 24 layers, 1024 dimensions, 16 attention heads
- **GPT-2 774M:** 36 layers, 1280 dimensions, 20 attention heads
- **GPT-2 1558M:** 48 layers, 1600 dimensions, 25 attention heads

Scaling Pattern:

Same architecture, different sizes → Different capabilities
Larger models = better performance but higher computational cost

Today's Focus: *GPT-2 124M - perfect for learning and experimentation*

Weight Loading Process (Part 1)

```python
# Download pretrained weights from OpenAI
import torch
import json
from urllib.request import urlretrieve

def download_and_load_gpt2_weights():
    """Map OpenAI format to our architecture"""
    # Download OpenAI GPT-2 model files
    model_size = "124M"
    base_url = f"https://openaipublic.azureedge.net/gpt-2/models/{model_size}
      "

    # Download model files
    files_to_download = ["encoder.json", "vocab.bpe",
                         "pytorch\_model.bin", "config.json"]

    for file in files_to_download:
        url = f"{base_url}/{file}"
        print(f"Downloading {file}...")
        urlretrieve(url, file)

    # Load the PyTorch state dictionary
    pretrained_weights = torch.load("pytorch\_model.bin", map_location="cpu")

    print("Downloaded weights keys:")
    for key in list(pretrained_weights.keys())[:5]:  # First 5 keys
        print(f"  {key}: {pretrained_weights[key].shape}")
    return pretrained_weights
```

Download Process:

- Access OpenAI public model repository
- Download key model files:
  - encoder.json (tokenizer mapping)
  - vocab.bpe (vocabulary)
  - pytorch_model.bin (weights)
  - config.json (model config)
- Load weights into memory
- Examine tensor structure

Key Features:

- Direct download from Azure CDN
- CPU-based loading (no GPU required)
- Automatic file retrieval

Weight Loading Process (Part 2)

```python
def load_weights_into_model(model, pretrained_weights):
    """Verify tensor shapes match and load weights"""

    # Create mapping from OpenAI naming to our naming
    weight_mapping = create_weight_mapping()

    # Load each weight tensor
    with torch.no_grad():
        for openai_name, our_name in weight_mapping.items():
            if openai_name in pretrained_weights:
                # Verify tensor shapes match
                pretrained_tensor = pretrained_weights[openai_name]
                our_tensor = get_parameter_by_name(model, our_name)

                assert pretrained_tensor.shape == our_tensor.shape, \
                    f"Shape mismatch: {pretrained_tensor.shape} vs {
   our_tensor.shape}"

                # Copy pretrained weights
                our_tensor.copy_(pretrained_tensor)
                print(f"Loaded {our_name}")

    print("All weights loaded successfully!")
    return model

# Usage
loaded_model = download_and_load_gpt2_weights()
gpt2_model = load_weights_into_model(model, loaded_model)
```

Integration Steps:

- Map naming conventions
- Verify tensor shapes
- Copy weights to model
- Safety checks throughout

Key Challenges:

- Different parameter naming
- Shape compatibility validation
- Architecture differences
- Version compatibility

- Different naming conventions
- Shape compatibility
- Memory management
- Version compatibility

Verification:

Always verify shapes match before loading

Exploring Loaded Weight Structure (Part 1)

```python
# Understand parameter organization after loading
def inspect_loaded_model(model):
    """Verify successful loading and understand model structure"""

    print("Model parameter summary:")
    total_params = 0

    # Check token embeddings
    token_embeddings = model.tok_emb.weight
    print(f"Token embeddings: {token_embeddings.shape}")
    print(f"  Vocabulary size: {token_embeddings.shape[0]:,}")
    print(f"  Embedding dimension: {token_embeddings.shape[1]}")
    total_params += token_embeddings.numel()

    # Check position embeddings
    position_embeddings = model.pos_emb.weight
    print(f"Position embeddings: {position_embeddings.shape}")
    print(f"  Context length: {position_embeddings.shape[0]}")
    total_params += position_embeddings.numel()
```

Inspection Goals:

- Verify parameter count
- Understand model structure
- Check embedding dimensions
- Validate weight loading

Exploring Loaded Weight Structure (Part 2)

```python
# Check transformer blocks
print(f"Number of transformer blocks: {len(model.trf_blocks)}")
for i, block in enumerate(model.trf_blocks):
    if i == 0:  # Show details for first block only
        attn_params = sum(p.numel() for p in block.att.parameters())
        ff_params = sum(p.numel() for p in block.ff.parameters())
        print(f"  Block {i}: Attention params: {attn_params:,}, FF params: {ff_params:,}")
    block_params = sum(p.numel() for p in block.parameters())
    total_params += block_params

# Final layer norm and output head
final_norm_params = model.final_norm.weight.numel()
out_head_params = model.out_head.weight.numel()
total_params += final_norm_params + out_head_params

print(f"Total parameters: {total_params:,}")
return total_params
```

Generation Test:

- Validate model functionality with prompts
- Apply temperature and top-k sampling
- Verify coherent output from model

Key Insights:

- Token + position embeddings
- Transformer blocks structure
- Total parameter verification
- Generation quality check

Debugging:

Compare parameter count with official specifications

Before and After: Pretrained Weights Impact

Dramatic Quality Transformation:

Random Initialization (Before)

Input: "The future of artificial intelligence"
Output: "*rones purch random? Nonetheless? ceremony? FLASH STAR igua? booko? purch? randomlated? purch?*"
Quality: Completely incoherent, random tokens, no language understanding

GPT-2 Pretrained Weights (After)

Input: "The future of artificial intelligence"
Output: "*is bright, with advances in machine learning enabling new applications across healthcare, education, and scientific research. These developments promise to...*""
Quality: Coherent, contextually appropriate, demonstrates language understanding

Key Improvements:

- **Coherence:** Logical flow and structure
- **Context:** Appropriate responses to prompts
- **Knowledge:** Factual understanding
- **Grammar:** Proper language mechanics

Result: Instant transformation from gibberish to human-like text

Fine-tuning vs Pretraining: Two-Stage Process

Stage 1: Pretraining

- Massive diverse datasets
- General language patterns
- Months of training
- Expensive computation
- Foundation model output

Examples:

- GPT-2 base model
- BERT foundation
- LLaMA weights

Stage 2: Fine-tuning

- Specific task datasets
- Targeted capabilities
- Days to weeks training
- Affordable computation
- Specialized model output

Examples:

- ChatGPT (conversation)
- GitHub Copilot (code)
- Medical LLMs (healthcare)

Transfer Learning: Leveraging pretrained knowledge for specific tasks

Economic Advantage:

- **Pretraining:** One-time massive investment (shared cost)
- **Fine-tuning:** Affordable customization (individual organizations)
- **Result:** Democratization of advanced AI capabilities

Next Lectures: *Deep dive into fine-tuning techniques and applications*

Hands-On Activity: Pretrained Model Testing

Load and test GPT-2 pretrained model!

Exercise: Quality Comparison

Task: Compare text generation before and after loading pretrained weights

Steps:

1. Generate text with randomly initialized model
2. Load GPT-2 124M pretrained weights
3. Generate text with same prompt using pretrained model
4. Compare quality, coherence, and relevance

Test Prompts:

- "The weather today is"
- "Machine learning algorithms"
- "In the year 2050, technology will"

Analysis Questions:

- What specific improvements do you notice?
- How does the model handle different domains?
- What limitations still exist in pretrained model?

Chunk 4 Summary: Pretrained Weights & Transfer Learning

Transfer Learning Mastery:

- **Economic Understanding:** Why pretraining is expensive, transfer learning is practical
- **Weight Loading:** Successfully integrate OpenAI GPT-2 weights
- **Quality Transformation:** Experience dramatic improvement
- **Deployment Strategy:** Two-stage pretraining + fine-tuning

Practical Skills:

- Download and integrate pretrained weights
- Verify model architecture compatibility
- Compare model performance before/after
- Plan cost-effective LLM strategies

Key Insight: *Pretrained weights provide instant access to years of training investment*

Foundation Complete: Ready for specialized fine-tuning applications

Lecture 6 Summary: Complete Training Pipeline

Four Chunks Mastered:

- **Loss Calculation:** Quantitative evaluation using cross-entropy
- **Training Infrastructure:** Data loaders, optimization loops, progress monitoring
- **Advanced Generation:** Temperature and top-k sampling for quality control
- **Transfer Learning:** Leveraging pretrained weights for practical deployment

End-to-End Capabilities:

- Build complete training pipelines from scratch
- Implement state-of-the-art text generation techniques
- Load and utilize industry-standard pretrained models
- Balance computational efficiency with model performance

Next Lecture Preview:

- **Lecture 7:** Fine-tuning for Classification and Instruction Following
- Specialized model adaptation techniques
- Task-specific performance optimization

Questions and Discussion

Thank you!

Questions?

Next: Advanced fine-tuning techniques and specialized applications

Office Hours: Available for training pipeline implementation help