Lecture 23: Implementing GPT from Scratch

Building a Language Model in PyTorch 💻

Models of Language and Conversation

Week 7

Today's Journey 🗺️

What We'll Build Today 🏗️

```
<div class="callout warning">
```

Goal

Implement a simplified GPT model from scratch in PyTorch!

```
**Components we'll cover:**
1. 🔤 Tokenization (Byte-Pair Encoding)
```

2. 📦 Embeddings (token + position)
3. 👁️ Masked Multi-Head Attention
4. 🧠 Transformer Decoder Block
5. 🎯 Language Model Head
6. 🎓 Training Loop

Required Libraries 📚

```
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.utils.data import Dataset, DataLoader
import numpy as np
import tiktoken  # OpenAI's BPE tokenizer

# Check for GPU
device = 'cuda' if torch.cuda.is_available() else 'cpu'
print(f"Using device: {device}")

# Hyperparameters
config = {
    'vocab_size': 50257,      # GPT-2 vocabulary size
    'd_model': 384,            # Embedding dimension
    'n_layers': 6,             # Number of transformer blocks
    'n_heads': 6,              # Number of attention heads
    'max_seq_len': 256,        # Maximum sequence length
```

Required Libraries 📚

```
21    'learning_rate': 3e-4,
22    'num_epochs': 10
23}
```

...continued

Install: pip install torch tiktoken

What is Tokenization? 🔤

```
<div class="callout info">
```

Tokenization

Converting text into a sequence of integer IDs that the model can process.

```
**Common strategies:**
1. **Word-level**: Split on spaces
    - ❌ Huge vocabulary, can't handle unknown words
```

2. **Character-level**: Individual characters
   - ❌ Very long sequences, loses word structure
3. **Subword-level (BPE)**: Best of both worlds! ✅
   - Frequent words: one token

- Rare words: multiple subword tokens

Byte-Pair Encoding (BPE) 🧩

```
**How BPE works:**


1. Start with characters as base vocabulary
```

2. Find most frequent pair of adjacent tokens
3. Merge this pair into a new token
4. Repeat until desired vocabulary size

Example

```
    **Text:** "low low low lower lower newest newest"


```

BPE Tokenization in Code 💻

```
1import tiktoken
2
3# Load GPT-2 tokenizer (uses BPE)
4tokenizer = tiktoken.get_encoding("gpt2")
5
6# Example text
7text = "Hello, how are you doing today?"
8
9# Encode: text -> token IDs
10tokens = tokenizer.encode(text)
11print("Tokens:", tokens)
12# Output: [15496, 11, 703, 389, 345, 1804, 1909, 30]
13
14# Decode: token IDs -> text
15decoded = tokenizer.decode(tokens)
16print("Decoded:", decoded)
17# Output: "Hello, how are you doing today?"
18
```

BPE Tokenization in Code 💻

```
    token_str = tokenizer.decode([token_id])
    print(f"{token_id}: '{token_str}'")

# Vocabulary size
print(f"Vocab size: {tokenizer.n_vocab}")  # 50257
```

...continued

Creating a Text Dataset 📚

```
class TextDataset(Dataset):
    def __init__(self, text_file, tokenizer, max_seq_len):
        # Load text
        with open(text_file, 'r', encoding='utf-8') as f:
            text = f.read()

        # Tokenize entire text
        self.tokens = tokenizer.encode(text)
        self.max_seq_len = max_seq_len

    def __len__(self):
        # Number of sequences we can extract
        return len(self.tokens) - self.max_seq_len

    def __getitem__(self, idx):
        # Get sequence of length max_seq_len + 1
        # (we need +1 for the target)
        chunk = self.tokens[idx:idx + self.max_seq_len + 1]
```

Creating a Text Dataset 📚

```
21        x = torch.tensor(chunk[:-1], dtype=torch.long)
22        # Target: all but first token
23        y = torch.tensor(chunk[1:], dtype=torch.long)
24
25        return x, y
26
27# Usage
28dataset = TextDataset('shakespeare.txt', tokenizer, config['max_seq_len'])
29dataloader = DataLoader(dataset, batch_size=config['batch_size'], shuffle=True
```

...continued

Token and Position Embeddings 📦

```
class Embeddings(nn.Module):
    def __init__(self, vocab_size, d_model, max_seq_len, dropout):
        super().__init__()
        # Token embeddings: map token IDs to vectors
        self.token_embed = nn.Embedding(vocab_size, d_model)

        # Position embeddings: encode position information
        self.pos_embed = nn.Embedding(max_seq_len, d_model)

         self.dropout = nn.Dropout(dropout)
         self.d_model = d_model

    def forward(self, x):
        # x shape: (batch_size, seq_len)
        seq_len = x.size(1)

        # Token embeddings
        tok_emb = self.token_embed(x)  # (batch, seq_len, d_model)
```

Token and Position Embeddings 📦

```
        positions = torch.arange(0, seq_len, device=x.device)
        pos_emb = self.pos_embed(positions)  # (seq_len, d_model)

        # Combine (broadcasting handles batch dimension)
        embeddings = tok_emb + pos_emb

        return self.dropout(embeddings)
```

...continued

Masked Multi-Head Attention 👁️

```
class MultiHeadAttention(nn.Module):
    def __init__(self, d_model, n_heads, dropout):
        super().__init__()
        assert d_model % n_heads == 0

        self.d_model = d_model
        self.n_heads = n_heads
        self.head_dim = d_model // n_heads

         # Linear layers for Q, K, V
         self.q_linear = nn.Linear(d_model, d_model)
         self.k_linear = nn.Linear(d_model, d_model)
         self.v_linear = nn.Linear(d_model, d_model)

         # Output projection
         self.out_linear = nn.Linear(d_model, d_model)
         self.dropout = nn.Dropout(dropout)

```

Masked Multi-Head Attention 👁️

```
21
22          # Linear projections and split into heads
23          Q = self.q_linear(x).view(batch_size, seq_len, self.n_heads, self.head
24          K = self.k_linear(x).view(batch_size, seq_len, self.n_heads, self.head
25          V = self.v_linear(x).view(batch_size, seq_len, self.n_heads, self.head
26
27          # Transpose for attention: (batch, n_heads, seq_len, head_dim)
28          Q = Q.transpose(1, 2)
29          K = K.transpose(1, 2)
30          V = V.transpose(1, 2)
```

...continued

Attention Computation (cont.) 🧮

```
# Scaled dot-product attention
        # Scores: (batch, n_heads, seq_len, seq_len)
        scores = torch.matmul(Q, K.transpose(-2, -1)) / np.sqrt(self.head_dim)

        # Apply causal mask (prevent attending to future tokens)
        if mask is not None:
            scores = scores.masked_fill(mask == 0, float('-inf'))

        # Softmax to get attention weights
         attn_weights = F.softmax(scores, dim=-1)
         attn_weights = self.dropout(attn_weights)

         # Apply attention to values
         # Output: (batch, n_heads, seq_len, head_dim)
         attn_output = torch.matmul(attn_weights, V)

         # Concatenate heads
         attn_output = attn_output.transpose(1, 2).contiguous()
```

Attention Computation (cont.) 🧮

```
        # Final linear projection
        output = self.out_linear(attn_output)

        return output
```

...continued

Creating the Causal Mask 🎭

```
def create_causal_mask(seq_len, device):
    """
    Create a causal (lower-triangular) mask for autoregressive generation.

    Returns:
        mask: (seq_len, seq_len) with 1s on and below diagonal, 0s above
    """
    mask = torch.tril(torch.ones(seq_len, seq_len, device=device))
    return mask  # Shape: (seq_len, seq_len)

# Example: 5x5 causal mask
mask = create_causal_mask(5, 'cpu')
print(mask)
# tensor([[1., 0., 0., 0., 0.],
#         [1., 1., 0., 0., 0.],
#         [1., 1., 1., 0., 0.],
#         [1., 1., 1., 1., 0.],
#         [1., 1., 1., 1., 1.]])
```

Feed-Forward Network 🔄

```
1class FeedForward(nn.Module):
2    def __init__(self, d_model, dropout):
3        super().__init__()
4        # GPT uses 4 * d_model as the hidden dimension
5        self.net = nn.Sequential(
6            nn.Linear(d_model, 4 * d_model),
7            nn.GELU(),  # GPT uses GELU activation
8            nn.Linear(4 * d_model, d_model),
9            nn.Dropout(dropout)
10        )
11
12    def forward(self, x):
13        return self.net(x)
```

Why GELU (Gaussian Error Linear Unit)?

- Smooth, non-monotonic activation

Transformer Decoder Block 🧱

```
1class TransformerBlock(nn.Module):
2    def __init__(self, d_model, n_heads, dropout):
3        super().__init__()
4        self.attention = MultiHeadAttention(d_model, n_heads, dropout)
5        self.feed_forward = FeedForward(d_model, dropout)
6
7        # Layer normalization (applied before sub-layers in GPT)
8        self.ln1 = nn.LayerNorm(d_model)
9        self.ln2 = nn.LayerNorm(d_model)
10
11    def forward(self, x, mask):
12        # Pre-norm architecture (used in GPT)
13        # Attention with residual connection
14        x = x + self.attention(self.ln1(x), mask)
15
16        # Feed-forward with residual connection
17        x = x + self.feed_forward(self.ln2(x))
18
```

Complete GPT Model 🤖

```
class GPT(nn.Module):
    def __init__(self, vocab_size, d_model, n_layers, n_heads, max_seq_len, dro
        super().__init__()
        self.max_seq_len = max_seq_len

        # Embeddings
        self.embeddings = Embeddings(vocab_size, d_model, max_seq_len, dropout)

        # Transformer blocks
         self.blocks = nn.ModuleList([
             TransformerBlock(d_model, n_heads, dropout)
             for _ in range(n_layers)
         ])

         # Final layer norm
         self.ln_f = nn.LayerNorm(d_model)

         # Language model head (projects to vocabulary)
```

Complete GPT Model 🤖

```
        # Initialize weights
        self.apply(self._init_weights)

    def _init_weights(self, module):
        if isinstance(module, nn.Linear):
            torch.nn.init.normal_(module.weight, mean=0.0, std=0.02)
            if module.bias is not None:
                torch.nn.init.zeros_(module.bias)
        elif isinstance(module, nn.Embedding):
            torch.nn.init.normal_(module.weight, mean=0.0, std=0.02)
```

...continued

GPT Forward Pass ➡ SOON

```
def forward(self, x, targets=None):
        # x shape: (batch_size, seq_len)
        seq_len = x.size(1)

        # Create causal mask
        mask = create_causal_mask(seq_len, x.device)

        # Embeddings
        x = self.embeddings(x)  # (batch, seq_len, d_model)

         # Apply transformer blocks
         for block in self.blocks:
             x = block(x, mask)

         # Final layer norm
         x = self.ln_f(x)

         # Project to vocabulary
```

GPT Forward Pass ➡ SOON

```
        # Compute loss if targets provided
        loss = None
        if targets is not None:
            # Flatten for cross-entropy
            loss = F.cross_entropy(
                logits.view(-1, logits.size(-1)),
                targets.view(-1)
            )

        return logits, loss
```

...continued

Training Loop 🎓

```
# Initialize model
model = GPT(
    vocab_size=config['vocab_size'],
    d_model=config['d_model'],
    n_layers=config['n_layers'],
    n_heads=config['n_heads'],
    max_seq_len=config['max_seq_len'],
    dropout=config['dropout']
).to(device)

# Optimizer (AdamW is used for GPT)
optimizer = torch.optim.AdamW(
    model.parameters(),
    lr=config['learning_rate'],
    betas=(0.9, 0.95),
    weight_decay=0.1
)

```

Training Loop 🎓

```
for epoch in range(config['num_epochs']):
    total_loss = 0
    for batch_idx, (x, y) in enumerate(dataloader):
        x, y = x.to(device), y.to(device)

        # Forward pass
        logits, loss = model(x, targets=y)

        # Backward pass
        optimizer.zero_grad()
        loss.backward()
        torch.nn.utils.clip_grad_norm_(model.parameters(), 1.0)
        optimizer.step()

        total_loss += loss.item()

    avg_loss = total_loss / len(dataloader)
    print(f"Epoch {epoch+1}/{config['num_epochs']}, Loss: {avg_loss:.4f}")
```

Training Tips 💡

```
**Best practices for training GPT:**



1. **Gradient clipping**
    - Prevents exploding gradients
```

- Clip to max norm of 1.0

2. **Learning rate schedule**
   - Warmup for first few thousand steps
- Cosine decay afterwards

3. **AdamW optimizer**
   - Adam with decoupled weight decay

Greedy Decoding 🎯

```
@torch.no_grad()
def generate_greedy(model, tokenizer, prompt, max_new_tokens=50):
    model.eval()

    # Encode prompt
    tokens = tokenizer.encode(prompt)
    x = torch.tensor([tokens], dtype=torch.long, device=device)

    for _ in range(max_new_tokens):
        # Get predictions (crop to max_seq_len if needed)
        x_crop = x[:, -model.max_seq_len:]
        logits, _ = model(x_crop)

        # Focus on last token's predictions
        logits = logits[:, -1, :]  # (batch, vocab_size)

        # Get token with highest probability
        next_token = torch.argmax(logits, dim=-1, keepdim=True)
```

Greedy Decoding 🎯

```
        x = torch.cat([x, next_token], dim=1)

        # Stop if we generate end-of-sequence token
        if next_token.item() == tokenizer.eot_token:
            break

    # Decode and return
    generated_text = tokenizer.decode(x[0].tolist())
    return generated_text

# Example usage
prompt = "Once upon a time"
generated = generate_greedy(model, tokenizer, prompt, max_new_tokens=100)
print(generated)
```

...continued

Sampling Strategies 🎲

```
**Different ways to sample next token:**



1. **Greedy Decoding**
    - Always pick most likely token
```

- ✅ Deterministic, fast
- ❌ Repetitive, boring

2. **Temperature Sampling**
   - Scale logits by temperature T

- $T < 1$: More conservative (peaked distribution)
- $T > 1$: More random (flat distribution)

Top-k and Nucleus Sampling 🎰

```
def sample_next_token(logits, temperature=1.0, top_k=None, top_p=None):
    """
    Sample next token from logits with various strategies.

    Args:
        logits: (vocab_size,) unnormalized log probabilities
        temperature: Temperature for sampling
        top_k: If set, only sample from top k tokens
        top_p: If set, only sample from nucleus (top-p)
    """
    # Apply temperature
    logits = logits / temperature

    # Top-k filtering
    if top_k is not None:
        top_k = min(top_k, logits.size(-1))
        indices_to_remove = logits < torch.topk(logits, top_k)[0][..., -1, None]
        logits[indices_to_remove] = float('-inf')
```

Top-k and Nucleus Sampling 🎰

```
    if top_p is not None:
        sorted_logits, sorted_indices = torch.sort(logits, descending=True)
        cumulative_probs = torch.cumsum(F.softmax(sorted_logits, dim=-1), dim=

        # Remove tokens with cumulative probability above threshold
        sorted_indices_to_remove = cumulative_probs > top_p
        sorted_indices_to_remove[..., 1:] = sorted_indices_to_remove[..., :-1]
        sorted_indices_to_remove[..., 0] = 0

        indices_to_remove = sorted_indices[sorted_indices_to_remove]
        logits[indices_to_remove] = float('-inf')

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

    return next_token
```

Complete Generation Function 🎨

```
@torch.no_grad()
def generate(model, tokenizer, prompt, max_new_tokens=100,
             temperature=1.0, top_k=40, top_p=0.9):
    model.eval()

    tokens = tokenizer.encode(prompt)
    x = torch.tensor([tokens], dtype=torch.long, device=device)

    for _ in range(max_new_tokens):
        x_crop = x[:, -model.max_seq_len:]
        logits, _ = model(x_crop)
        logits = logits[:, -1, :]  # Last token

        # Sample next token
        next_token = sample_next_token(
            logits[0],
            temperature=temperature,
            top_k=top_k,
```

Complete Generation Function 🎨

```
21
22         x = torch.cat([x, next_token.unsqueeze(0)], dim=1)
23
24         if next_token.item() == tokenizer.eot_token:
25             break
26
27     return tokenizer.decode(x[0].tolist())
28
29# Creative generation
30text = generate(model, tokenizer, "The AI revolution",
31                temperature=0.8, top_p=0.9)
32print(text)
```

...continued

Model Size vs. Performance 📊

```
1 10M ->  100M ->  1B ->  10B ->  100B+
```

```
**Practical model sizes for different use cases:**
- **10M-100M**: Learning/experimentation, simple tasks
```

- **100M-1B**: Specialized domains, resource-constrained
- **1B-10B**: General-purpose, good quality
- **10B-100B+**: State-of-the-art performance

Computational Requirements 💻

```
**Training a 125M parameter GPT:**



    | Training Time | 1–2 days (single GPU) |
```

| --- | --- |

| Training Data | $\sim$10-100 GB text |

| Total Compute | $\sim$100 GPU-hours |

```
**Scaling up to GPT-3 (175B):**
- 1000x more parameters
```

- ~10,000x more compute needed

Common Issues and Debugging 🔧

```
**Problems you might encounter:**



1. **Loss not decreasing**
    - Check learning rate (try 1e-4 to 3e-4)
```

- Verify data pipeline
- Check for NaN/Inf values

2. **Out of memory**
   - Reduce batch size

- Reduce sequence length
- Use gradient accumulation
- Enable mixed precision

Extensions and Improvements 🚀

```
**Ways to enhance your GPT implementation:**



1. **Architectural improvements**
    - Rotary Position Embeddings (RoPE)
```

- Flash Attention (faster attention)
- Grouped-Query Attention

2. **Training techniques**
   - Learning rate warmup and decay

- Gradient accumulation
- Mixed precision training (FP16/BF16)

Resources for Further Learning 📚

```
**Recommended resources:**



- **Andrej Karpathy's nanoGPT**
```

- Clean, minimal GPT implementation
- github.com/karpathy/nanoGPT

```
  \item **Andrej Karpathy's "Let's build GPT" video**
  - Excellent step-by-step tutorial
```

- YouTube

```
  \item **HuggingFace Transformers**
  - Production-ready implementations
```

Hands-On Exercise 💪

```
<div class="callout warning">
```

Your Task

Implement and train a small GPT model on your own text corpus!

```
**Steps:**
1. Choose a dataset (Shakespeare, Wikipedia, your own text)
```

2. Set up the data pipeline
3. Initialize the model (start small: 6 layers, 384 d_model)
4. Train for 10-20 epochs
5. Experiment with generation
6. Try different sampling strategies

Key Takeaways 🔑

```
1. **GPT is conceptually simple**
    - Stack of transformer decoder blocks
```

- Predict next token

2. **Key components**
   - BPE tokenization

- Token + position embeddings
- Masked multi-head attention
- Feed-forward networks

3. **Training requires care**
   - Good data, proper hyperparameters

Readings 📖

```
<div class="callout info">
```

Required Readings

1. **Vaswani et al. (2017)** - "Attention is All You Need" \
   [ArXiv]
2. **Radford et al. (2018)** - "Improving Language Understanding by Generative Pre-Training" \
   [PDF]

```
<div class="callout info">
```

Code Resources

- **nanoGPT** by Andrej Karpathy \
  github.com/karpathy/nanoGPT

Next Week 🔮

```
**Week 8: No Classes – Instructor Away**



**Use this week to:**
- Complete the GPT implementation exercise
```

- Catch up on readings
- Work on assignments
- Experiment with different architectures

  Week 9: RAG & Mixture of Experts

  - Retrieval Augmented Generation
- Mixture of Experts architectures

```
Questions? 💬


Happy Coding! 🚀
```