Large Language Models: Comprehensive Guide from Zero to Hero

A deep-dive into LLM architecture, training, fine-tuning, and deployment with in-depth explanations and working code examples.

Table of Contents

---

1. Foundations of NLP and Machine Learning

Introduction to NLP: Text Preprocessing Pipeline

Natural Language Processing starts with converting raw text into structured data that neural networks can process. This pipeline is fundamental to all downstream tasks.

Tokenization: Breaking Text into Meaningful Units

Tokenization is the process of splitting text into tokens (words, subwords, or characters). This is more complex than simple whitespace splitting.

Example: Different tokenization approaches

text = "ChatGPT's capabilities are impressive!"

1. Word tokenization (naive)

word_tokens = text.lower().split()

Output: ["chatgpt's", "capabilities", "are", "impressive!"]

2. Subword tokenization (BPE - Byte Pair Encoding)

```
from tokenizers import Tokenizer
tokenizer = Tokenizer.from_file("gpt2.json")
tokens = tokenizer.encode(text)
```

Output: [17894, 10905, 1039, 2620, 28]

3. Character-level tokenization

```
char_tokens = list(text)
```

Output: ['C', 'h', 'a', 't', 'G', 'P', 'T', ''', 's', ...]

Key Concepts:

- **Word-level tokenization**: Vocabulary size ~50,000-100,000. Simple but can't handle OOV (out-of-vocabulary) words
- **Subword tokenization** (BPE/WordPiece): Vocabulary size ~30,000-50,000. Modern standard for large models
- **Character-level**: Vocabulary size ~100-256. Can represent any word but sequence becomes very long

Stemming vs Lemmatization

Both reduce words to base forms, but with different approaches:

```
from nltk.stem import PorterStemmer, WordNetLemmatizer

stemmer = PorterStemmer()
lemmatizer = WordNetLemmatizer()

words = ["running", "runs", "ran", "easily", "fairly"]

print("Stemming:")
for word in words:
print(f"{word} → {stemmer.stem(word)}")
```

Output:

running → run

runs → run

ran → ran (doesn't recognize past tense)

easily → easili

fairly → fairli

print("\nLemmatization:")
for word in words:
print(f"{word} → {lemmatizer.lemmatize(word, 'v')}")

Output:

running → run

runs → run

ran → run (understands past tense)

easily → easily (preserves adverbs)

fairly → fairly

Differences:

- **Stemming**: Rule-based, faster, sometimes produces non-words ("easili")
- **Lemmatization**: Dictionary-based, accurate, requires POS tags

Modern approach: Skip this for LLMs. Transformers learn representations that handle these naturally.

Word Embeddings: Representing Meaning as Vectors

Embeddings convert discrete tokens into continuous vectors that capture semantic meaning.

Word2Vec: Skip-gram and CBOW

Word2Vec uses shallow neural networks to learn word representations by predicting context words.

Skip-gram model concept

Given: "The quick brown fox"

Predict surrounding words from each word

"quick" → predict ["The", "brown"] (context window = 2)

"brown" → predict ["quick", "fox"]

Implementation

```
from gensim.models import Word2Vec
```

```
sentences = [
["the", "quick", "brown", "fox"],
["the", "lazy", "dog"],
]
```

```
model = Word2Vec(
sentences=sentences,
vector_size=100, # embedding dimension
window=2, # context window size
min_count=1, # minimum word frequency
sg=1 # 1=Skip-gram, 0=CBOW
)
```

Access embeddings

```
vector = model.wv["quick"] # shape: (100,)
similarity = model.wv.similarity("quick", "fast") # cosine similarity
```

Key metrics:

- **Vector dimension**: Typically 100-300 for Word2Vec. Higher = more capacity but more data needed
- **Context window**: 2-5 words on each side. Larger window captures broader context but loses syntactic info
- **Training method**: Skip-gram learns better representations but is slower; CBOW is faster

GloVe: Global Vectors for Word Representation

GloVe combines global matrix factorization with local context windows:

GloVe doesn't require separate tokenization; it builds a co-occurrence matrix

and factorizes it to learn embeddings

Using pre-trained GloVe embeddings

import numpy as np

Load pre-trained GloVe (e.g., 100-dim trained on 6B tokens)

embeddings_index = {}
with open('glove.6B.100d.txt') as f:
for line in f:
values = line.split()
word = values[0]
coef = np.asarray(values[1:], dtype='float32')
embeddings_index[word] = coef

Access embedding

queen_embedding = embeddings_index['queen'] # shape: (100,)

Interesting property: word analogies

king - man + woman ≈ queen

king = embeddings_index['king']
man = embeddings_index['man']
woman = embeddings_index['woman']
queen_predicted = king - man + woman

Cosine similarity with actual queen is very high!

Comparison with Word2Vec:

- **GloVe**: Captures both local context and global corpus statistics. Better for similarity tasks
- **Word2Vec**: Simpler, trains faster. Better for specific downstream tasks

Why LLMs don't use static embeddings: Modern Transformer models learn context-dependent embeddings. "bank" in "river bank" vs "savings bank" gets different representations based on context.

Deep Learning Basics: Foundation for Language Models

Neural Networks and Backpropagation

A neural network is a composition of functions that learns from data through gradient descent.

Simplified neural network forward pass

import numpy as np

class SimpleNeuralNetwork:
def **init**(self, input_dim, hidden_dim, output_dim):
# Initialize weights with small random values
self.W1 = np.random.randn(input_dim, hidden_dim) * 0.01
self.b1 = np.zeros((1, hidden_dim))
self.W2 = np.random.randn(hidden_dim, output_dim) * 0.01
self.b2 = np.zeros((1, output_dim))

```
def relu(self, x):
    """Activation function: ReLU(x) = max(0, x)"""
    return np.maximum(0, x)

def softmax(self, x):
    """Converts logits to probabilities"""
    exp_x = np.exp(x - np.max(x, axis=1, keepdims=True))  # numerical stability
```

```
        return exp_x / np.sum(exp_x, axis=1, keepdims=True)

    def forward(self, X):
        """Forward pass: input → hidden layer → output"""
        self.z1 = np.dot(X, self.W1) + self.b1          # Linear transformation
        self.a1 = self.relu(self.z1)                    # Non-linearity
        self.z2 = np.dot(self.a1, self.W2) + self.b2    # Output layer
        self.a2 = self.softmax(self.z2)                 # Probabilities
        return self.a2

    def backward(self, X, y, learning_rate=0.01):
        """Backpropagation: compute gradients and update weights"""
        batch_size = X.shape[0]

        # Output layer gradient
        dz2 = self.a2 - y  # derivative of cross-entropy loss + softmax
        dW2 = np.dot(self.a1.T, dz2) / batch_size
        db2 = np.sum(dz2, axis=0, keepdims=True) / batch_size

        # Hidden layer gradient
        da1 = np.dot(dz2, self.W2.T)
        dz1 = da1 * (self.a1 > 0)  # derivative of ReLU
        dW1 = np.dot(X.T, dz1) / batch_size
        db1 = np.sum(dz1, axis=0, keepdims=True) / batch_size

        # Update weights (gradient descent)
        self.W1 -= learning_rate * dW1
        self.b1 -= learning_rate * db1
        self.W2 -= learning_rate * dW2
        self.b2 -= learning_rate * db2
```

Training loop

nn = SimpleNeuralNetwork(input_dim=10, hidden_dim=64, output_dim=5)
for epoch in range(100):
logits = nn.forward(X_train)
nn.backward(X_train, y_train, learning_rate=0.01)
if epoch % 20 == 0:

```
loss = -np.mean(np.log(logits[np.arange(len(y_train)), y_train.argmax(axis=1)]))
print(f"Epoch {epoch}, Loss: {loss:.4f}")
```

Key concepts:

- **Forward pass**: Data flows through layers, computing predictions
- **Backpropagation**: Compute gradients by applying chain rule from output to input
- **Gradient descent**: Update weights in direction opposite to gradient
- **Learning rate**: Controls step size. Too high = divergence; too low = slow training

Optimization Algorithms: SGD, Adam, and Beyond

Different optimizers have different convergence properties:

```
import torch
import torch.optim as optim

model = YourModel()
```

1. Stochastic Gradient Descent (SGD)

```
optimizer_sgd = optim.SGD(model.parameters(), lr=0.01, momentum=0.9)
```

Momentum: accumulate past gradients to smooth updates

Good for: simple, stable training

Bad for: can get stuck in local minima, slow convergence

2. Adam (Adaptive Moment Estimation) - Modern standard

```
optimizer_adam = optim.Adam(
model.parameters(),
lr=1e-3, # typical learning rate
betas=(0.9, 0.999), # exponential moving average coefficients
eps=1e-8, # numerical stability
weight_decay=1e-5 # L2 regularization
)
```

Maintains per-parameter learning rates

Good for: most modern tasks, handles sparse gradients

Bad for: can be memory-heavy for very large models

3. AdamW (Adam with decoupled weight decay)

optimizer_adamw = optim.AdamW(
model.parameters(),
lr=1e-3,
weight_decay=0.01 # decoupled from gradient-based updates
)

Improvement over Adam: better generalization

Used in: GPT models, BERT fine-tuning

Training loop with optimizer

for epoch in range(num_epochs):
for batch in dataloader:
# Forward pass
outputs = model(batch)
loss = criterion(outputs, targets)

```
    # Backward pass
    optimizer.zero_grad()  # Clear old gradients
    loss.backward()        # Compute new gradients

    # Gradient clipping (prevents exploding gradients)
    torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)
```

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

Comparison:

Algorithm	Convergence	Memory	Generalization	Use Case
SGD	Slow	Low	Good	Simple tasks
Momentum	Faster	Low	Good	Traditional ML
Adam	Fast	Medium	OK	Most modern tasks
AdamW	Fast	Medium	Better	LLM training (preferred)

Loss Functions for Language Modeling

Cross-entropy is the standard loss for predicting probability distributions:

import torch.nn.functional as F

Language modeling task: predict next token

logits = model(input_ids) # shape: (batch_size, seq_len, vocab_size)
targets = next_token_ids # shape: (batch_size, seq_len)

Cross-entropy loss: measures divergence between predicted and true distribution

loss = F.cross_entropy(
logits.view(-1, vocab_size), # flatten: (batch_size * seq_len, vocab_size)
targets.view(-1) # flatten: (batch_size * seq_len)
)

What's happening mathematically:

For each position, softmax(logits) gives probability distribution

Cross-entropy = -log(P(correct_token))

This encourages high probability for the correct token and low for others

Perplexity = exp(loss)

Measures how "surprised" the model is on test data

Lower is better. Perplexity of 10 = model thinks test data is ~10x more likely

perplexity = torch.exp(loss)

Sequential Models: RNNs, LSTMs, and GRUs

Before Transformers, sequential models processed text one token at a time with hidden state.

Recurrent Neural Networks (RNNs)

RNNs maintain a hidden state that gets updated at each timestep:

import torch
import torch.nn as nn

class SimpleRNN(nn.Module):
def **init**(self, input_size, hidden_size, output_size):
super().**init**()
# Weight matrices
self.W_h = nn.Linear(hidden_size, hidden_size) # hidden to hidden
self.W_x = nn.Linear(input_size, hidden_size) # input to hidden
self.W_o = nn.Linear(hidden_size, output_size) # hidden to output

```
def forward(self, x):
    # x shape: (batch_size, seq_len, input_size)
    batch_size, seq_len, _ = x.shape
    hidden = torch.zeros(batch_size, self.W_h.out_features)
    outputs = []

    for t in range(seq_len):
        # h_t = tanh(W_x * x_t + W_h * h_{t-1})
        hidden = torch.tanh(
            self.W_x(x[:, t, :]) + self.W_h(hidden)
        )
        output = self.W_o(hidden)
        outputs.append(output.unsqueeze(1))

    # outputs shape: (batch_size, seq_len, output_size)
    return torch.cat(outputs, dim=1)
```

PyTorch's built-in RNN (more efficient)

rnn = nn.RNN(
input_size=100,
hidden_size=256,
num_layers=2, # stack 2 RNN layers
batch_first=True # input: (batch, seq, features)
)

outputs, hidden = rnn(x) # outputs: (batch, seq, hidden_size)

Problem with RNNs: Vanishing/Exploding Gradients

When backpropagating through many timesteps, gradients get multiplied repeatedly:

- Gradient $\approx \partial L/\partial h_0 = (\partial h_1/\partial h_0) \times (\partial h_2/\partial h_1) \times \ldots \times (\partial h_t/\partial h_{t-1})$
- If each factor < 1, product → 0 (vanishing)
- If each factor > 1, product → ∞ (exploding)
- Result: Can't learn long-range dependencies

Long Short-Term Memory (LSTM)

LSTMs solve the vanishing gradient problem with gating mechanisms:

class LSTMCell(nn.Module):
"""Single LSTM cell"""
def **init**(self, input_size, hidden_size):
super().**init**()

self.input_size = input_size
self.hidden_size = hidden_size

```
        # All gates combined into one computation for efficiency
        self.gates = nn.Linear(input_size + hidden_size, 4 * hidden_size)

    def forward(self, x, (h, c)):
        # x: (batch, input_size)
        # h: hidden state (batch, hidden_size)
        # c: cell state (batch, hidden_size)

        # Concatenate input and hidden state
        combined = torch.cat([x, h], dim=1)

        # Compute all gates
        gates = self.gates(combined)  # (batch, 4*hidden_size)

        # Split into 4 gates
        input_gate, forget_gate, candidate_gate, output_gate = \
            gates.chunk(4, dim=1)

        # Apply sigmoid to gates (values in [0, 1])
        input_gate = torch.sigmoid(input_gate)
        forget_gate = torch.sigmoid(forget_gate)
        output_gate = torch.sigmoid(output_gate)
        candidate_gate = torch.tanh(candidate_gate)

        # Update cell state
        # c_t = f_t ⊙ c_{t-1} + i_t ⊙ g_t
        c_new = forget_gate * c + input_gate * candidate_gate

        # Compute hidden state
        # h_t = o_t ⊙ tanh(c_t)
        h_new = output_gate * torch.tanh(c_new)

        return h_new, c_new
```

How LSTMs avoid vanishing gradients:

- **Cell state** acts as a "highway" with additive updates ($\partial c_t/\partial c_{t-1}$ involves addition, not multiplication)
- **Forget gate** controls what information to keep ($f_t \in [0,1]$ usually > 0.5)
- **Gradients** can flow unchanged through cell state: $\partial c_T/\partial c_t$ involves products of values close to 1

PyTorch's built-in LSTM

```
lstm = nn.LSTM(
input_size=100,
hidden_size=256,
num_layers=2,
batch_first=True,
dropout=0.2 # apply dropout between layers
)
```

```
outputs, (h_n, c_n) = lstm(x)
```

outputs: (batch, seq, hidden_size) - all hidden states

h_n: final hidden state

c_n: final cell state

Gated Recurrent Unit (GRU)

GRUs are a simpler variant of LSTMs with fewer gates:

GRU equations (simplified LSTM)

Reset gate: $r_t = \sigma(W_r [h_{t-1}, x_t])$

Update gate: $z_t = \sigma(W_z [h_{t-1}, x_t])$

Candidate: $\tilde{h}_t = \tanh(W_h [r_t \odot h_{t-1}, x_t])$

Hidden state: $h_t = (1 - z_t) \odot \tilde{h}_t + z_t \odot h_{t-1}$

```
gru = nn.GRU(
input_size=100,
hidden_size=256,
num_layers=2,
batch_first=True
)

outputs, h_n = gru(x)
```

Comparison:

Metric	RNN	GRU	LSTM
Parameters	Fewest	Medium	Most
Training Speed	Fastest	Faster	Slower
Long Dependencies	Poor	Good	Best
Computational Cost	Low	Medium	High

Why LSTMs/GRUs are less used now:

- Sequential processing is slow (can't parallelize)
- Transformers with attention handle long-range dependencies much better
- Maximum practical sequence length: ~500-1000 tokens (vs 100K+ for Transformers)

---

2. The Transformer Architecture: Core Concepts

Transformers replaced RNNs by enabling parallel processing of sequences through attention mechanisms.

The Attention Mechanism: Foundation of Everything

Attention allows the model to focus on relevant parts of the input regardless of distance.

Scaled Dot-Product Attention

```
import torch
import torch.nn as nn
import math

def scaled_dot_product_attention(Q, K, V, mask=None):
"""
Q: Query matrix (batch, seq_len, d_k)
K: Key matrix (batch, seq_len, d_k)
V: Value matrix (batch, seq_len, d_v)
```

mask: Optional mask for autoregressive (causal) attention
"""
d_k = Q.shape[-1]

```
# Compute attention scores
# Attention(Q, K, V) = softmax(Q * K^T / sqrt(d_k)) * V
scores = torch.matmul(Q, K.transpose(-2, -1)) / math.sqrt(d_k)
# scores shape: (batch, seq_len, seq_len)

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

# Normalize attention weights to sum to 1
attention_weights = torch.softmax(scores, dim=-1)

# Apply attention weights to values
output = torch.matmul(attention_weights, V)
# output shape: (batch, seq_len, d_v)

return output, attention_weights
```

Example: Sequence of 4 tokens, embedding dim 64

seq_len, d_model = 4, 64
Q = torch.randn(1, seq_len, d_model) # queries
K = torch.randn(1, seq_len, d_model) # keys
V = torch.randn(1, seq_len, d_model) # values

output, weights = scaled_dot_product_attention(Q, K, V)

weights shape: (1, 4, 4) - attention from each token to all tokens

Why scale by √d_k?

- Dot products can be very large, causing softmax to be dominated by one value
- Scaling prevents saturation and keeps gradients stable
- Variance of Q·K^T / √d_k ≈ 1 (normalized)

Multi-Head Attention

Instead of computing attention once, compute it multiple times in parallel with different subspaces:

class MultiHeadAttention(nn.Module):
def **init**(self, d_model, num_heads):
super().**init**()
assert d_model % num_heads == 0

```
        self.d_model = d_model
        self.num_heads = num_heads
        self.d_k = d_model // num_heads  # dimension of each head

        # Linear projections
        self.W_q = nn.Linear(d_model, d_model)
        self.W_k = nn.Linear(d_model, d_model)
        self.W_v = nn.Linear(d_model, d_model)
        self.W_o = nn.Linear(d_model, d_model)

    def forward(self, Q, K, V, mask=None):
        batch_size = Q.shape[0]
        seq_len = Q.shape[1]

        # Linear projections
        Q = self.W_q(Q)  # (batch, seq_len, d_model)
        K = self.W_k(K)
        V = self.W_v(V)

        # Split into multiple heads
        # Reshape: (batch, seq_len, d_model) → (batch, seq_len, num_heads, d_k)
        # Transpose: (batch, num_heads, seq_len, d_k)
        Q = Q.reshape(batch_size, seq_len, self.num_heads, self.d_k).transpose(1, 2)
        K = K.reshape(batch_size, seq_len, self.num_heads, self.d_k).transpose(1, 2)
        V = V.reshape(batch_size, seq_len, self.num_heads, self.d_k).transpose(1, 2)

        # Apply scaled dot-product attention
        attn_output, attn_weights = scaled_dot_product_attention(Q, K, V, mask)
        # attn_output: (batch, num_heads, seq_len, d_k)
```

```
    # Concatenate heads
    attn_output = attn_output.transpose(1, 2).contiguous()
    # (batch, seq_len, num_heads, d_k)
    attn_output = attn_output.reshape(batch_size, seq_len, self.d_model)
    # (batch, seq_len, d_model)

    # Final linear projection
    output = self.W_o(attn_output)

    return output, attn_weights
```

Usage

mha = MultiHeadAttention(d_model=512, num_heads=8)
x = torch.randn(batch_size=32, seq_len=100, d_model=512)
output, weights = mha(x, x, x) # Self-attention: Q=K=V=x

Each head operates on 512/8 = 64-dimensional subspace

Benefits of multi-head attention:

- Different heads attend to different aspects (syntax, semantics, long-range, local patterns)
- Parallelizable: all heads compute simultaneously
- More expressive than single-head attention

The Transformer Block: Self-Attention + FFN

class TransformerBlock(nn.Module):
def **init**(self, d_model, num_heads, d_ff, dropout=0.1):
super().**init**()

```
    # Layer 1: Multi-head self-attention
    self.self_attn = MultiHeadAttention(d_model, num_heads)

    # Layer 2: Feed-forward network (MLP)
    self.ffn = nn.Sequential(
        nn.Linear(d_model, d_ff),
        nn.GELU(),  # Smooth ReLU variant
        nn.Linear(d_ff, d_model)
```

```
        )

        # Normalization and dropout
        self.norm1 = nn.LayerNorm(d_model)
        self.norm2 = nn.LayerNorm(d_model)
        self.dropout = nn.Dropout(dropout)

    def forward(self, x, mask=None):
        # x shape: (batch, seq_len, d_model)

        # Sub-layer 1: Self-attention with residual connection
        attn_output, _ = self.self_attn(x, x, x, mask)
        attn_output = self.dropout(attn_output)
        x = self.norm1(x + attn_output)  # Residual + LayerNorm

        # Sub-layer 2: Feed-forward with residual connection
        ffn_output = self.ffn(x)
        ffn_output = self.dropout(ffn_output)
        x = self.norm2(x + ffn_output)  # Residual + LayerNorm

        return x
```

Stack multiple transformer blocks for deeper model

class Transformer(nn.Module):
def **init**(self, d_model, num_heads, num_layers, d_ff, dropout=0.1):
super().**init**()
self.layers = nn.ModuleList([
TransformerBlock(d_model, num_heads, d_ff, dropout)
for _ in range(num_layers)
])

```
    def forward(self, x, mask=None):
        for layer in self.layers:
            x = layer(x, mask)
        return x
```

Positional Encoding: Injecting Position Information

Attention doesn't have inherent position awareness. We must add position information explicitly.

Sinusoidal Positional Encoding (Transformers)

class PositionalEncoding(nn.Module):
def **init**(self, d_model, max_seq_length=5000):
super().**init**()

```
    # Create positional encoding matrix
    pe = torch.zeros(max_seq_length, d_model)
    position = torch.arange(0, max_seq_length).unsqueeze(1).float()

    # Compute dimension indices
    div_term = torch.exp(
        torch.arange(0, d_model, 2).float() *
        -(math.log(10000.0) / d_model)
    )

    # Apply sin to even indices, cos to odd indices
    pe[:, 0::2] = torch.sin(position * div_term)      # even positions
    pe[:, 1::2] = torch.cos(position * div_term)      # odd positions

    # Register as buffer (not trained, but saved with model)
    self.register_buffer('pe', pe.unsqueeze(0))

def forward(self, x):
    # x shape: (batch, seq_len, d_model)
    x = x + self.pe[:, :x.shape[1], :]
    return x
```

Mathematical formula:

PE(pos, 2i) = sin(pos / 10000^(2i/d_model))

PE(pos, 2i+1) = cos(pos / 10000^(2i/d_model))

Properties:

- Each dimension has different frequency

- Creates unique encoding for each position

- Can extrapolate to longer sequences than training

Why sinusoidal encoding?

- Provides absolute position information ($PE_{10} \neq PE_{20}$)
- Encodes relative position information (PE_{pos+k} can be derived from PE_{pos})
- Works for any sequence length without retraining
- Hardware-friendly (no learnable parameters)

Rotary Position Embedding (RoPE) - Modern Standard

RoPE is more efficient and better for long sequences:

class RotaryPositionEmbedding(nn.Module):
def **init**(self, d_model):
super().**init**()
self.d_model = d_model
# Precompute inverse frequencies
inv_freq = 1.0 / (10000 ** (torch.arange(0, d_model, 2).float() / d_model))
self.register_buffer("inv_freq", inv_freq)

```
def forward(self, x, seq_len):
    # x shape: (batch, seq_len, d_model)
    # Compute position indices
    t = torch.arange(seq_len, device=x.device, dtype=self.inv_freq.dtype)

    # Compute frequencies for each position
    freqs = torch.einsum("i,j->ij", t, self.inv_freq)

    # Compute cos and sin
    emb = torch.cat([freqs, freqs], dim=-1)
```

```
    cos_emb = emb.cos()[None, :, :]
    sin_emb = emb.sin()[None, :, :]

    return cos_emb, sin_emb
```

def apply_rotary_pos_emb(q, k, cos_emb, sin_emb):
"""Apply rotary embeddings to queries and keys"""
# Reshape for element-wise operations
q_rot = torch.stack([-q[..., 1::2], q[..., 0::2]], dim=-1).flatten(-2)
k_rot = torch.stack([-k[..., 1::2], k[..., 0::2]], dim=-1).flatten(-2)

```
q = q * cos_emb + q_rot * sin_emb
k = k * cos_emb + k_rot * sin_emb

return q, k
```

Usage in attention

rope = RotaryPositionEmbedding(d_model=512)
cos_emb, sin_emb = rope(x, seq_len)
Q_rotated, K_rotated = apply_rotary_pos_emb(Q, K, cos_emb, sin_emb)

Advantages of RoPE:

- Extrapolates better to longer sequences
- Used in modern models: LLaMA, Mistral
- More efficient than sinusoidal encoding

Layer Normalization and Residual Connections

Layer Normalization vs Batch Normalization

LayerNorm: normalize across feature dimension

BatchNorm: normalize across batch dimension

LayerNorm (used in Transformers)

x = torch.randn(32, 100, 512) # (batch, seq_len, d_model)
norm = nn.LayerNorm(512) # normalize last dimension
y = norm(x)

Each token's 512-dimensional vector is normalized independently

Mean of last dimension = 0, Var = 1

Mathematical:

y = gamma * (x - mean(x)) / sqrt(var(x) + eps) + beta

gamma and beta are learnable parameters

Why LayerNorm in Transformers?

1. Works well with attention (doesn't depend on batch size)

2. Effective for variable-length sequences

3. Stabilizes training of deep models

Residual Connections (Skip Connections)

Without residual: y = f(x)

With residual: y = x + f(x)

class ResidualBlock(nn.Module):
def **init**(self, d_model):
super().**init**()
self.linear1 = nn.Linear(d_model, d_model * 4)
self.linear2 = nn.Linear(d_model * 4, d_model)
self.norm = nn.LayerNorm(d_model)

```
def forward(self, x):
    # Pre-normalization (modern, more stable)
    normed = self.norm(x)
    output = self.linear2(F.gelu(self.linear1(normed)))
    return x + output  # Residual connection
```

Why residual connections?

1. Gradient flow: ∂L/∂x = ∂L/∂y * ∂y/∂x = ∂L/∂y * (1 + ∂f/∂x)

Even if ∂f/∂x ≈ 0, gradient can still flow through

2. Easier optimization: model learns small corrections to identity function

3. Enables training very deep networks (50+layers)

Full Transformer Architecture: Encoder-Decoder Structure

class FullTransformer(nn.Module):
def **init**(self, vocab_size, d_model, num_heads, num_layers, d_ff, max_seq_len, dropout=0.1):
super().**init**()

```
    # Embedding layers
    self.embedding = nn.Embedding(vocab_size, d_model)
```

```
        self.positional_encoding = PositionalEncoding(d_model, max_seq_len)

        # Encoder (processes input)
        self.encoder = Transformer(
            d_model=d_model,
            num_heads=num_heads,
            num_layers=num_layers,
            d_ff=d_ff,
            dropout=dropout
        )

        # Decoder (generates output, attends to encoder)
        self.decoder = Transformer(
            d_model=d_model,
            num_heads=num_heads,
            num_layers=num_layers,
            d_ff=d_ff,
            dropout=dropout
        )

        # Output projection to vocabulary
        self.output_projection = nn.Linear(d_model, vocab_size)

    def forward(self, src, tgt, src_mask=None, tgt_mask=None, memory_mask=None
        # src: source sequence (batch, src_len)
        # tgt: target sequence (batch, tgt_len)

        # Embed and add positional encoding
        src_embed = self.positional_encoding(self.embedding(src))
        tgt_embed = self.positional_encoding(self.embedding(tgt))

        # Encode source
        memory = self.encoder(src_embed, src_mask)

        # Decode with cross-attention to encoder
        # (implementation simplified - actual cross-attention needed)
        tgt_output = self.decoder(tgt_embed, tgt_mask)
```

```
    # Project to vocabulary
    logits = self.output_projection(tgt_output)

    return logits
```

Key Architectural Variants

Model	Type	Attention Pattern	Use Case
BERT	Encoder-only	Bidirectional	Classification, NER, QA
GPT	Decoder-only	Causal (masked)	Text generation
T5	Encoder-decoder	Both types	Seq2seq tasks
RoBERTa	Encoder-only	Bidirectional	General NLP
ELECTRA	Encoder-only	Bidirectional	Efficient pretraining
LLaMA	Decoder-only	Causal + RoPE	Large-scale generation
Mistral	Decoder-only	Causal + RoPE	Fast inference

---

3. LLM Pre-training and Training Objectives

Pre-training on massive text corpora is what gives LLMs their knowledge and capabilities.

Language Modeling Objectives

Autoregressive (Causal) Language Modeling - GPT

Predict the next token given all previous tokens. This is how GPT models are trained.

```
import torch
import torch.nn as nn

class GPTLanguageModel(nn.Module):
"""Autoregressive language model (like GPT)"""
```

```
def __init__(self, vocab_size, d_model, num_heads, num_layers, d_ff):
    super().__init__()

    self.embedding = nn.Embedding(vocab_size, d_model)
    self.positional_encoding = PositionalEncoding(d_model)
    self.transformer = Transformer(d_model, num_heads, num_layers, d_ff)
    self.lm_head = nn.Linear(d_model, vocab_size)

def forward(self, input_ids, labels=None):
    # input_ids: (batch, seq_len)
    # labels: (batch, seq_len) - target tokens for loss computation

    # Get embeddings with positional encoding
    x = self.embedding(input_ids)
    x = self.positional_encoding(x)

    # Create causal mask (can't attend to future positions)
    seq_len = input_ids.shape[1]
    causal_mask = torch.triu(
        torch.ones(seq_len, seq_len),
        diagonal=1
    ).bool().to(input_ids.device)
    # Matrix of shape (seq_len, seq_len) where:
    # causal_mask[i, j] = True if i < j (can't attend to future)

    # Apply transformer
    x = self.transformer(x, mask=causal_mask)

    # Project to vocabulary logits
    logits = self.lm_head(x)  # (batch, seq_len, vocab_size)

    # Compute loss if labels provided
    loss = None
    if labels is not None:
        loss = nn.functional.cross_entropy(
            logits.view(-1, vocab_size),
            labels.view(-1)
```

```
    )

    return logits, loss
```

Training loop

model = GPTLanguageModel(vocab_size=50000, d_model=768, num_heads=12, num_layers=12, d_ff=3072)
optimizer = torch.optim.AdamW(model.parameters(), lr=1e-4)

for epoch in range(num_epochs):
for batch in dataloader:
input_ids = batch['input_ids'] # (batch, seq_len)
labels = batch['labels'] # same as input_ids shifted right

```
    # Forward pass
    logits, loss = model(input_ids, labels=labels)

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

How labels are created:

If input_ids = [101, 2054, 2003, 1998]

labels = [2054, 2003, 1998, 102] (shifted by 1)

Model learns: given [101], predict 2054

given [101, 2054], predict 2003

etc.

Key properties of causal language modeling:

- Only attend to past and current tokens (no future information)
- Predicts next token given context
- Natural for text generation
- Every position contributes to loss

Masked Language Modeling - BERT

Randomly mask tokens and predict them from context. This is how BERT is trained.

class BERTModel(nn.Module):
"""Masked Language Model (like BERT)"""

```
def __init__(self, vocab_size, d_model, num_heads, num_layers, d_ff):
    super().__init__()

    self.embedding = nn.Embedding(vocab_size, d_model)
    self.positional_encoding = PositionalEncoding(d_model)
    self.transformer = Transformer(d_model, num_heads, num_layers, d_ff)
    self.mlm_head = nn.Linear(d_model, vocab_size)
    self.mask_token_id = 103  # [MASK] token id

def create_mlm_labels(self, input_ids, mask_prob=0.15):
    """Create MLM task: randomly mask tokens"""
    labels = input_ids.clone()

    # Identify positions to mask
    random_mask = torch.rand(input_ids.shape) < mask_prob
    special_tokens = (input_ids == 101) | (input_ids == 102)  # [CLS], [SEP]
    mask_positions = random_mask & ~special_tokens

    # For masked positions:
    # 80% → replace with [MASK]
    # 10% → replace with random token
    # 10% → keep original (noise)

    masked_input = input_ids.clone()
```

```
        for i in range(input_ids.shape[0]):
            positions = torch.where(mask_positions[i])[0]
            for pos in positions:
                rand = torch.rand(1).item()
                if rand < 0.8:
                    masked_input[i, pos] = self.mask_token_id
                elif rand < 0.9:
                    masked_input[i, pos] = torch.randint(0, 50000, (1,)).item()
                # else: keep original

        return masked_input, labels

    def forward(self, input_ids, masked_input=None):
        if masked_input is None:
            masked_input = input_ids

        # Get embeddings with positional encoding
        x = self.embedding(masked_input)
        x = self.positional_encoding(x)

        # No causal mask - BERT attends bidirectionally
        x = self.transformer(x, mask=None)

        # Project to vocabulary for each position
        logits = self.mlm_head(x)  # (batch, seq_len, vocab_size)

        return logits
```

Training

model = BERTModel(vocab_size=50000, d_model=768, num_heads=12, num_layers=12, d_ff=3072)
optimizer = torch.optim.AdamW(model.parameters(), lr=1e-4)

for epoch in range(num_epochs):
for batch in dataloader:
input_ids = batch['input_ids']

```
# Create MLM task
masked_input, labels = model.create_mlm_labels(input_ids)

# Forward pass
logits = model(input_ids, masked_input=masked_input)

# Compute loss only on masked positions
loss = 0
for i in range(len(labels)):
    mask_positions = torch.where((masked_input[i] == 103))[0]
    if len(mask_positions) > 0:
        pred = logits[i, mask_positions]
        true = labels[i, mask_positions]
        loss += nn.functional.cross_entropy(pred, true)

# Backward pass
optimizer.zero_grad()
loss.backward()
optimizer.step()
```

Comparison:

Aspect	Autoregressive (GPT)	Masked (BERT)
Training objective	Predict next token	Predict masked tokens
Context	Left-to-right (causal)	Bidirectional
Generation	Natural (token-by-token)	Requires special handling
Pretraining speed	Slower (sequential)	Faster
Fine-tuning	Direct for generation	Needs adaptation for generation
Example models	GPT, GPT-2, GPT-3	BERT, RoBERTa

Tokenization Strategies: Byte-Pair Encoding (BPE)

BPE reduces vocabulary size while maintaining expressiveness through subword units.

from tokenizers import Tokenizer, models, normalizers, pre_tokenizers, decoders, trainers

1. Create a BPE tokenizer

tokenizer = Tokenizer(models.BPE())

2. Set preprocessing

tokenizer.normalizer = normalizers.Sequence([
normalizers.NFC(),
])

tokenizer.pre_tokenizer = pre_tokenizers.ByteLevel(add_prefix_space=True)

3. Train on corpus

trainer = trainers.BpeTrainer(
vocab_size=50000,
min_frequency=2,
special_tokens=["<|endoftext|>", "<|padding|>"]
)

files: list of training files

tokenizer.train(files, trainer)

4. Use the tokenizer

text = "Hello, how are you?"
encoded = tokenizer.encode(text)
print(encoded.tokens) # ['Hello', ',', ' how', ' are', ' you', '?']
print(encoded.ids) # [256, 11, 512, 890, 1024, 34] (example ids)

5. Decode back to text

decoded = tokenizer.decode(encoded.ids)
print(decoded) # "Hello, how are you?"

BPE algorithm (simplified):

Start with character-level vocabulary

Repeat until vocab_size reached:

1. Find most frequent pair of adjacent tokens

2. Merge them into single token

3. Add to vocabulary

Example:

Iteration 0: [h, e, l, l, o] (vocab: 256 characters)

Iteration 1: [h, e, l, l, o] → [h, e, "ll", o] (merge "l" + "l")

Iteration 2: → [h, e, "ll", o] → ["he", "ll", o] (merge "h" + "e")

...continues until vocab_size=50000

Why BPE is better than word-level tokenization:

- **Vocabulary size**: 50K (BPE) vs 100K+ (word-level)
- **OOV handling**: Rare words → subword pieces (not "UNK")
- **Efficiency**: Shorter sequences, faster processing
- **Compression**: Common patterns are merged early

Modern alternatives:

WordPiece (used in BERT)

- Similar to BPE but uses likelihood-based merging

- Slightly better for classification tasks

SentencePiece (used in T5, mBART)

- Language-agnostic

- Treats special characters as separate tokens

- Better for multilingual models

```
from sentencepiece import SentencePieceProcessor
sp = SentencePieceProcessor()
sp.Load("model.model")
tokens = sp.EncodeAsIds("Hello world")
```

Large-Scale Distributed Training

Training LLMs requires distributed training across multiple GPUs/TPUs.

```
import torch.distributed as dist
from torch.nn.parallel import DistributedDataParallel as DDP
```

Initialize distributed training

```
dist.init_process_group("nccl") # NVIDIA Collective Communications Library
rank = dist.get_rank()
world_size = dist.get_world_size()
device = torch.device(f"cuda:{rank}")
```

Model on GPU

model = GPTLanguageModel(...).to(device)
model = DDP(model, device_ids=[rank])

Data loader with DistributedSampler

sampler = DistributedSampler(
dataset,
num_replicas=world_size,
rank=rank,
shuffle=True
)

dataloader = DataLoader(dataset, sampler=sampler, batch_size=32)

Training loop (same as single-GPU)

optimizer = torch.optim.AdamW(model.parameters(), lr=1e-4)

for epoch in range(num_epochs):
for batch in dataloader:
# Forward pass
logits, loss = model(batch['input_ids'], labels=batch['labels'])

```
# Backward pass - gradients are accumulated across all GPUs
optimizer.zero_grad()
loss.backward()

# All-reduce: average gradients across GPUs
# (DDP does this automatically)

optimizer.step()
```

Cleanup

dist.destroy_process_group()

Launching with multiple GPUs:

torchrun --nproc_per_node=4 train.py

Launches 4 processes, one per GPU, automatically handles synchronization

Key distributed training concepts:

- **Data parallelism**: Each GPU gets different batch, same model
- **Model parallelism**: Model sharded across GPUs (for very large models)
- **All-reduce**: Synchronize gradients across GPUs (collective operation)
- **Pipeline parallelism**: Different layers on different GPUs

---

4. Adaptation and Fine-Tuning

Fine-tuning adapts pre-trained models to specific tasks. Full parameter fine-tuning is resource-intensive; parameter-efficient methods enable fine-tuning on consumer hardware.

Full Fine-Tuning: Update All Parameters

Load pre-trained model

model = GPTLanguageModel.from_pretrained("gpt2-medium")

Unfreeze all parameters

for param in model.parameters():
param.requires_grad = True

Fine-tune on task-specific data

optimizer = torch.optim.AdamW(model.parameters(), lr=5e-5)

for epoch in range(num_epochs):
for batch in task_dataloader:
input_ids = batch['input_ids']
labels = batch['labels']

```
logits, loss = model(input_ids, labels=labels)

optimizer.zero_grad()
```

```
    loss.backward()
    torch.nn.utils.clip_grad_norm_(model.parameters(), 1.0)
    optimizer.step()
```

Save fine-tuned model

model.save_pretrained("./finetuned-model")

Requirements:

- Memory: 3× model size (weights + gradients + optimizer states)
- For GPT-2 (1.5B parameters): ~18 GB VRAM needed
- Training time: proportional to learning rate cycles needed

Advantages:

- Highest accuracy (all parameters adapted)
- Simple to implement

Disadvantages:

- Catastrophic forgetting: model loses general knowledge
- Requires large amounts of task data and compute
- Hyperparameter tuning critical (learning rate, warmup, etc.)

LoRA: Low-Rank Adaptation [2023]

Instead of updating all parameters, add small trainable matrices alongside frozen weights.

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

```
def __init__(self, original_layer, rank=8, lora_alpha=16):
    super().__init__()
    self.original_layer = original_layer
    self.rank = rank
    self.lora_alpha = lora_alpha

    # Freeze original weights
    original_layer.weight.requires_grad = False
    original_layer.bias.requires_grad = False

    # LoRA matrices: decompose update as low-rank
    # ΔW = (A @ B) where A ∈ ℝ^(in×r), B ∈ ℝ^(r×out)
    in_features = original_layer.in_features
```

```
        out_features = original_layer.out_features

        self.lora_a = nn.Parameter(
            torch.randn(in_features, rank) * (2 / (in_features + rank))
        )
        self.lora_b = nn.Parameter(torch.zeros(rank, out_features))

        # Scaling factor
        self.scaling = lora_alpha / rank

    def forward(self, x):
        # W_eff = W_0 + α/r * (A @ B)
        # Where W_0 is frozen original weight
        lora_out = (x @ self.lora_a) @ self.lora_b * self.scaling
        return self.original_layer(x) + lora_out
```

Inject LoRA into model

def add_lora_to_model(model, rank=8, lora_alpha=16):
for name, module in model.named_modules():
if isinstance(module, nn.Linear):
setattr(
model.get_submodule(name.rsplit('.', 1)[0]),
name.rsplit('.', 1)[1],
LoRALayer(module, rank=rank, lora_alpha=lora_alpha)
)

model = GPTLanguageModel.from_pretrained("gpt2-medium")
add_lora_to_model(model, rank=8)

Count trainable parameters

total_params = sum(p.numel() for p in model.parameters())
trainable_params = sum(p.numel() for p in model.parameters() if p.requires_grad)
print(f"Total: {total_params/1e6:.1f}M, Trainable: {trainable_params/1e6:.1f}M")

GPT-2 Medium: 345M total, 0.3M trainable (0.09%)

Training is identical

```
optimizer = torch.optim.AdamW(
filter(lambda p: p.requires_grad, model.parameters()),
lr=1e-4
)
```

```
for epoch in range(num_epochs):
for batch in dataloader:
logits, loss = model(batch['input_ids'], labels=batch['labels'])
optimizer.zero_grad()
loss.backward()
optimizer.step()
```

Save only LoRA weights (~1-2 MB)

```
torch.save(
{name: param for name, param in model.named_parameters() if param.requires_grad},
"lora_weights.pt"
)
```

Load LoRA weights

```
lora_weights = torch.load("lora_weights.pt")
for name, param in lora_weights.items():
model.get_parameter(name).data.copy_(param)
```

Mathematical foundation:

Full update matrix (345M parameters):
ΔW = [massive matrix]

LoRA decomposition (0.3M parameters):
$\Delta W = A @ B$ where $A \in \mathbb{R}^{(4096\times8)}$, $B \in \mathbb{R}^{(8\times4096)}$

Memory savings: $(4096\times8 + 8\times4096) / (4096\times4096) \approx 0.1\%$

Computation: $x @ \Delta W$ becomes $(x @ A) @ B$ (still efficient)

LoRA Properties:

- **Rank (r)**: Controls expressiveness. r=8 for small tasks, r=64 for complex tasks
- **Alpha**: Scaling factor. Typical ratio alpha/rank = 16/8 = 2
- **Dropout**: Regularization applied to LoRA matrices (typically 0.05)
- **Target modules**: Usually QKV projections in attention, sometimes all linear layers

QLoRA: Quantized LoRA [2023]

Combine 4-bit quantization with LoRA for even more memory efficiency.

from bitsandbytes.nn import Linear4bit

def quantize_and_lora(model, rank=8):
"""Convert model to 4-bit with LoRA"""

```
for name, module in model.named_modules():
    if isinstance(module, nn.Linear):
        # Replace with 4-bit quantized version
        new_module = Linear4bit(
            module.in_features,
            module.out_features,
            bias=module.bias is not None,
            compute_dtype=torch.bfloat16,  # high precision for computation
            weight_dtype=torch.uint8,      # 4-bit weights
        )

        # Add LoRA adapters on top
        new_module = LoRALayer(new_module, rank=rank)
        setattr(model, name, new_module)
```

Using HuggingFace's built-in QLoRA

from peft import LoraConfig, get_peft_model
from transformers import AutoModelForCausalLM

Load model in 4-bit

model = AutoModelForCausalLM.from_pretrained(
"meta-llama/Llama-2-7b-hf",
load_in_4bit=True,
bnb_4bit_compute_dtype=torch.float16,
bnb_4bit_use_double_quant=True,
device_map="auto",
)

Apply LoRA

lora_config = LoraConfig(
r=16,
lora_alpha=32,
target_modules=["q_proj", "v_proj"], # only QV layers
lora_dropout=0.05,
bias="none",
task_type="CAUSAL_LM"
)

model = get_peft_model(model, lora_config)

Now fine-tune

trainer = transformers.Trainer(
model=model,
args=transformers.TrainingArguments(
output_dir="./output",
learning_rate=2e-4,
num_train_epochs=3,
per_device_train_batch_size=4,
gradient_accumulation_steps=4,
),
train_dataset=dataset,
)

trainer.train()

Memory comparison:

Model: LLaMA-65B

Full Fine-tuning:

- Model: 130 GB (float16)
- Gradients: 130 GB
- Optimizer states: 130 GB
- Total: ~390 GB (costs $50,000+ in H100 GPUs)

LoRA (rank=8):

- Model: 130 GB (frozen)
- LoRA weights: ~0.3 MB (trainable)
- Gradients: ~0.3 MB
- Optimizer states: ~0.6 MB
- Total: ~130 GB (needs one A100 GPU: ~$10,000)

QLoRA:

- Model: 16 GB (4-bit quantized)

- LoRA weights: ~0.3 MB
- Total: ~16 GB (works on RTX 4090: ~$2000)

QLoRA innovations:

- **4-bit NormalFloat (NF4)**: Custom quantization format preserving accuracy
- **Double quantization**: Quantize the quantization constants
- **Paged optimizers**: Swap optimizer states to CPU RAM
- **Result**: Fine-tune 65B parameter models on single RTX 4090

Other Parameter-Efficient Methods

Prefix Tuning

class PrefixTuning(nn.Module):
"""Prepend learnable prefix to attention keys/values"""

```
def __init__(self, d_model, prefix_len=20, num_heads=12):
    super().__init__()
    self.prefix_len = prefix_len
    self.num_heads = num_heads

    # Learnable prefix (hidden dimension × prefix length)
    self.prefix_embeddings = nn.Parameter(
        torch.randn(prefix_len, d_model)
    )

    # Optional projection network
    self.projection = nn.Sequential(
        nn.Linear(d_model, d_model),
        nn.Tanh(),
        nn.Linear(d_model, num_heads * prefix_len)
    )

def forward(self, key, value):
    # Prepend prefix to key and value sequences
    prefix = self.prefix_embeddings.unsqueeze(0)  # (1, prefix_len, d_model)
    key = torch.cat([prefix, key], dim=1)          # prepend
    value = torch.cat([prefix, value], dim=1)
    return key, value
```

Training: only train prefix_embeddings (~2-5 MB for LLMs)

Comparison of PEFT methods:

Method	Memory	Speed	Accuracy	Flexibility
Full FT	3x	Slow	Best	Full
LoRA	1x	Fast	95-99%	High
QLoRA	0.3x	Fast	95-98%	High
Prefix Tuning	1.1x	Fast	90-95%	Medium
Adapter Tuning	1.05x	Fast	92-96%	Medium
BitFit	1x	Fast	85-90%	Low

Instruction Tuning: Supervised Fine-Tuning (SFT)

Convert pre-trained models to instruction-following models through supervised fine-tuning.

Instruction tuning dataset format

dataset = [
{
"instruction": "Summarize this text in one sentence:",
"input": "Transformers have revolutionized NLP...",
"output": "Transformers are the foundation of modern NLP."
},
{
"instruction": "Translate English to French",
"input": "Hello, how are you?",
"output": "Bonjour, comment allez-vous?"
},
... thousands more examples
]

class InstructionTuningDataset:
def **init**(self, examples, tokenizer, max_length=512):
self.examples = examples
self.tokenizer = tokenizer
self.max_length = max_length

```
def __getitem__(self, idx):
    example = self.examples[idx]
```

```
# Format: instruction + input + output
prompt = f"{example['instruction']}\n{example['input']}\n"
full_text = prompt + example['output']

# Tokenize
inputs = self.tokenizer(
    full_text,
    max_length=self.max_length,
    truncation=True,
    padding="max_length"
)

# Mask loss for instruction tokens (only learn to predict output)
prompt_tokens = self.tokenizer(
    prompt,
    max_length=self.max_length,
    truncation=True
)
prompt_len = len(prompt_tokens['input_ids'])

# Labels: -100 (ignored by loss) for prompt, token ids for output
labels = [-100] * prompt_len + inputs['input_ids'][prompt_len:]

return {
    'input_ids': inputs['input_ids'],
    'attention_mask': inputs['attention_mask'],
    'labels': labels
}
```

Training

dataloader = DataLoader(
InstructionTuningDataset(dataset, tokenizer),
batch_size=32,
shuffle=True
)

model = GPTLanguageModel.from_pretrained("gpt2-medium")
optimizer = torch.optim.AdamW(model.parameters(), lr=5e-5)

for epoch in range(3):
for batch in dataloader:
input_ids = batch['input_ids'].to(device)
labels = batch['labels'].to(device)

```
logits, loss = model(input_ids, labels=labels)

optimizer.zero_grad()
loss.backward()
optimizer.step()
```

After instruction tuning, model follows instructions:

model.generate("Translate to French: Hello")

→"Bonjour"

RLHF: Reinforcement Learning from Human Feedback [2022]

RLHF is a 3-step process:

1. SFT: Train model on high-quality examples (done above)

2. Reward modeling: Train model to predict human preferences

3. PPO: Optimize policy using reward signal

Step 2: Train reward model

class RewardModel(nn.Module):
def **init**(self, base_model):
super().**init**()
self.model = base_model
self.reward_head = nn.Linear(768, 1) # predict scalar reward

```
def forward(self, input_ids):
    hidden = self.model(input_ids)[0]  # get hidden states
    reward = self.reward_head(hidden[:, -1, :])  # use [EOS] token representation
    return reward
```

Dataset: pairs of responses with human preference labels

preference_dataset = [
{
"prompt": "Summarize this article...",
"chosen": "This article discusses...", # better response
"rejected": "The article is about..." # worse response
},
...
]

Training: maximize log(σ(reward_chosen - reward_rejected))

σ is sigmoid, so we want: reward_chosen > reward_rejected

class PreferenceLoss(nn.Module):
def forward(self, reward_chosen, reward_rejected):
return -torch.log(torch.sigmoid(reward_chosen - reward_rejected)).mean()

Step 3: PPO training (complex, uses policy gradients)

Simplification:

from trl import PPOTrainer

ppo_config = PPOConfig(learning_rate=1e-5)
ppo_trainer = PPOTrainer(
model=sft_model,
config=ppo_config,
dataset=preference_dataset,
reward_model=reward_model,
)

ppo_trainer.train()

Modern alternative: DPO (Direct Preference Optimization) [2023]

DPO simplifies RLHF by directly optimizing policy without separate reward model:

class DPOLoss(nn.Module):
def **init**(self, beta=0.5):
self.beta = beta

```
def forward(self, policy_chosen_logps, policy_rejected_logps,
            reference_chosen_logps, reference_rejected_logps):
    """
    policy_chosen_logps: log probabilities of chosen response under policy
    policy_rejected_logps: log probabilities of rejected response under policy
    reference_*: same but under reference (pre-SFT) model
    """
    # DPO objective
    chosen_log_probs = policy_chosen_logps - reference_chosen_logps
    rejected_log_probs = policy_rejected_logps - reference_rejected_logps

    loss = -torch.log(
        torch.sigmoid(
            self.beta * (chosen_log_probs - rejected_log_probs)
        )
    ).mean()
```

```
    return loss
```

Training is simpler:

For each preference pair:

1. Compute logits from both policy and reference model

2. Apply DPO loss

3. Backprop and update

Result: Models trained with DPO rival RLHF-trained models

Advantage: No separate reward model, simpler pipeline

---

5. Prompt Engineering and Prompting Techniques

The way you frame a question dramatically affects LLM output quality.

Zero-Shot Prompting

Ask the model to solve a task without examples:

Zero-shot classification

prompt = """Classify the sentiment of this text as positive, negative, or neutral.

Text: "I absolutely love this product! It exceeded all my expectations."
Sentiment:"""

response = model.generate(prompt, max_tokens=10)

Output: "Positive"

Zero-shot arithmetic

prompt = """Solve: 15 × 23 = ?"""
response = model.generate(prompt, max_tokens=10)

Output: "345"

Zero-shot question answering

prompt = """Question: What is the capital of France?
Answer:"""
response = model.generate(prompt, max_tokens=10)

Output: "Paris"

Characteristics:

- No task examples provided
- Model relies on general knowledge from pre-training
- Works well for straightforward tasks
- May fail for complex, specialized tasks

Few-Shot Prompting

Provide a few examples to guide the model:

Few-shot sentiment classification

prompt = """Classify sentiment as positive or negative.

Text: "I love this product!"
Sentiment: positive

Text: "This is terrible."
Sentiment: negative

Text: "It's okay, nothing special."
Sentiment:"""

response = model.generate(prompt, max_tokens=10)

Output: "neutral" or "negative"

Few-shot arithmetic

prompt = """Solve the following:

9 × 12 = 108
7 × 8 = 56
6 × 13 = ?"""

response = model.generate(prompt, max_tokens=10)

Output: "78"

Few-shot translation

prompt = """Translate English to French:

English: "Hello"
French: "Bonjour"

English: "Good morning"
French: "Bon matin"

English: "How are you?"
French:"""

response = model.generate(prompt, max_tokens=10)

Output: "Comment allez-vous?"

Properties:

- 2-5 examples typically enough
- In-context learning: model learns from examples without parameter updates
- Improves performance significantly vs zero-shot
- Examples should be diverse and representative

Chain-of-Thought (CoT) Prompting

Explicitly ask the model to show reasoning steps:

Without CoT

prompt = """A store sells apples at $2 each. Sarah buys 5 apples
and gets 10% discount. How much does she pay?"""

response = model.generate(prompt, max_tokens=10)

Output might be wrong due to missing intermediate steps

With CoT

prompt = """Let's solve this step by step.

A store sells apples at $2 each. Sarah buys 5 apples
and gets 10% discount. How much does she pay?

Step 1: Calculate original price
Original price = 5 apples × $2/apple = $10

Step 2: Calculate discount amount
Discount = 10% of $10 = $1

Step 3: Calculate final price
Final price = $10 - $1 = $9

Answer: Sarah pays $9"""

Model much more likely to get this right!

Few-shot CoT

prompt = """Solve problems step-by-step.

Q: If there are 3 cars in the parking lot and 2 more arrive, how many are there?
A: Let me think step-by-step:

- Started with 3 cars
- 2 more arrived
- 3 + 2 = 5 cars
  Total: 5 cars

Q: A bakery makes 20 cookies per hour. How many in 8 hours?
A: Let me think step-by-step:

- Makes 20 cookies per hour
- Working for 8 hours

- 20 × 8 = 160 cookies
  Total: 160 cookies

Q: A book costs $15. Buy 3 with 25% discount. Total cost?
A:"""

Model benefits from CoT pattern in examples

Why CoT works:

- Forces model to break down complex problems
- Makes errors more visible (wrong intermediate step)
- Activates better reasoning capabilities
- Works especially well for math and logic

Tree-of-Thought (ToT) Prompting

Explore multiple reasoning paths and select the best:

class TreeOfThought:
def **init**(self, model, branch_factor=3, depth=3):
self.model = model
self.branch_factor = branch_factor
self.depth = depth

```
def generate_thoughts(self, context, num_thoughts=3):
    """Generate multiple next thoughts"""
    prompt = f"""Given the context below, generate {num_thoughts} diverse
```

and plausible next thoughts to solve the problem.

Context: {context}

Thought 1: """

```
    thoughts = []
    for i in range(num_thoughts):
        response = self.model.generate(prompt, max_tokens=100)
        thoughts.append(response)
        prompt += f"{response}\n\nThought {i+2}: "

    return thoughts

def evaluate_thoughts(self, context, thoughts):
```

```
        """Score each thought for viability"""
        prompt = f"""For the following problem-solving context and proposed thoug
```

rate each thought's quality on a scale of 1-10.

Context: {context}

"""
scores = []
for i, thought in enumerate(thoughts):
prompt += f"Thought {i+1}: {thought}\nRating: "
response = self.model.generate(prompt, max_tokens=10)
try:
score = float(response.split()[0])
except:
score = 5.0 # default if parsing fails
scores.append(score)

```
        return scores

    def solve(self, problem):
        """Solve problem using tree search"""
        # BFS search through thought tree
        queue = [(problem, 0)]  # (context, depth)
        solutions = []

        while queue:
            context, depth = queue.pop(0)

            if depth >= self.depth:
                # Reached max depth, extract answer
                solutions.append(context)
                continue

            # Generate next thoughts
            thoughts = self.generate_thoughts(context, self.branch_factor)

            # Evaluate thoughts
            scores = self.evaluate_thoughts(context, thoughts)

            # Keep top thoughts
```

```
        ranked = sorted(zip(thoughts, scores), key=lambda x: x[1], reverse=True)
        for thought, score in ranked[:self.branch_factor]:
            new_context = context + "\n" + thought
            queue.append((new_context, depth + 1))

    # Return best solution
    return max(solutions, key=lambda x: self._score_solution(x))

def _score_solution(self, solution):
    """Score final solution"""
    prompt = f"How good is this solution? Rate 1-10.\n{solution}\nRating: "
    response = self.model.generate(prompt, max_tokens=10)
    try:
        return float(response.split()[0])
    except:
        return 5.0
```

Usage

tot = TreeOfThought(model, branch_factor=3, depth=2)
answer = tot.solve("Complex reasoning problem...")

When to use:

- Complex problems requiring exploration
- Multiple solution paths exist
- Computational budget allows (expensive due to multiple generations)

---

6. Retrieval-Augmented Generation and Agents

RAG: Retrieving Knowledge During Generation

from sentence_transformers import SentenceTransformer
from sklearn.metrics.pairwise import cosine_similarity
import numpy as np

class SimpleRAG:
def **init**(self, model_name="all-MiniLM-L6-v2"):
self.encoder = SentenceTransformer(model_name)
self.documents = []
self.embeddings = np.array([])

```
def add_documents(self, docs):
    """Index documents"""
    self.documents = docs
    self.embeddings = self.encoder.encode(docs)
    # embeddings shape: (num_docs, embedding_dim)

def retrieve(self, query, top_k=3):
    """Find most relevant documents"""
    query_embedding = self.encoder.encode([query])
    similarities = cosine_similarity(query_embedding, self.embeddings)[0]
    top_indices = np.argsort(similarities)[-top_k:][::-1]
    return [self.documents[i] for i in top_indices]

def generate_with_rag(self, query):
    """RAG pipeline"""
    # 1. Retrieve relevant documents
    context_docs = self.retrieve(query, top_k=3)
    context = "\n".join(context_docs)

    # 2. Augment prompt with retrieved context
    prompt = f"""Answer the question based on the provided context.
```

Context:
{context}

Question: {query}
Answer:"""

```
    # 3. Generate response
    response = llm.generate(prompt, max_tokens=200)
    return response
```

Usage

rag = SimpleRAG()
rag.add_documents([
"Paris is the capital of France.",
"The Eiffel Tower is located in Paris.",
"France is known for wine and cheese.",

"Machine learning is a subfield of AI.",
... more documents
])

answer = rag.generate_with_rag("What is Paris famous for?")

Retrieves relevant documents about Paris

Generates answer with grounded context

Advanced RAG with vector databases:

from langchain.vectorstores import Chroma
from langchain.embeddings.openai import OpenAIEmbeddings
from langchain.document_loaders import PDFLoader
from langchain.text_splitter import RecursiveCharacterTextSplitter

1. Load documents

loader = PDFLoader("research_paper.pdf")
documents = loader.load()

2. Split into chunks

splitter = RecursiveCharacterTextSplitter(
chunk_size=1000,
chunk_overlap=200
)
chunks = splitter.split_documents(documents)

3. Create embeddings and store in vector DB

embeddings = OpenAIEmbeddings()
vector_db = Chroma.from_documents(chunks, embeddings, persist_directory="./db")

4. Retrieve and generate

from langchain.chains import RetrievalQA
from langchain.llms import OpenAI

retriever = vector_db.as_retriever(search_kwargs={"k": 3})

qa = RetrievalQA.from_chain_type(
llm=OpenAI(temperature=0),
chain_type="stuff",

retriever=retriever
)

answer = qa.run("What are the main contributions of this paper?")

LLM Agents: Using Tools

Agents use LLMs to decide which tools to call and how to interpret results:

import json
from enum import Enum

class ToolType(Enum):
CALCULATOR = "calculator"
WEB_SEARCH = "web_search"
WIKIPEDIA = "wikipedia"

class Tool:
def **init**(self, name, description, function):
self.name = name
self.description = description
self.function = function

```
def execute(self, *args, **kwargs):
    return self.function(*args, **kwargs)
```

def calculator(expression):
"""Safely evaluate math expression"""
try:
return eval(expression)
except:
return "Invalid expression"

def web_search(query):
"""Search the web for information"""
Placeholder - would use real API
return f"Search results for: {query}"

def wikipedia_lookup(topic):
"""Look up topic on Wikipedia"""
Placeholder - would use real API
return f"Wikipedia article about {topic}"

Define tools

tools = [
Tool("calculator", "Perform mathematical calculations", calculator),
Tool("web_search", "Search the internet for information", web_search),
Tool("wikipedia", "Look up information on Wikipedia", wikipedia_lookup),
]

class Agent:
def **init**(self, llm, tools):
self.llm = llm
self.tools = {tool.name: tool for tool in tools}
self.tool_descriptions = "\n".join([
f"- {tool.name}: {tool.description}"
for tool in tools
])

```
def plan(self, query):
    """Use LLM to decide which tools to use"""
    prompt = f"""You are a helpful assistant with access to these tools:
```

{self.tool_descriptions}

For the user's question, use the tools available to solve it.
Respond in JSON format with your reasoning and tool calls.

Format: {{"thought": "...", "tool": "tool_name", "input": "..."}}

User question: {query}
Your response:"""

```
    response = self.llm.generate(prompt, max_tokens=500)
    try:
        return json.loads(response)
    except:
        return {"thought": "No tool needed", "tool": None, "input": None}

def execute(self, query):
    """Solve query using tools"""
    max_iterations = 5
    thought_process = []

    for iteration in range(max_iterations):
        # 1. Plan: which tool to use
        plan = self.plan(query)
        thought_process.append(plan)

        # 2. Check if done
        if plan.get("tool") is None:
            return plan.get("thought", "Unable to solve")
```

```
        # 3. Execute tool
        tool_name = plan["tool"]
        if tool_name not in self.tools:
            query = f"Tool {tool_name} not found. Try another."
            continue

        tool = self.tools[tool_name]
        result = tool.execute(plan["input"])

        # 4. Update query with result
        query = f"Tool {tool_name} returned: {result}\n\nContinue solving the orig

    return "Max iterations reached"
```

Usage

agent = Agent(llm, tools)
answer = agent.execute("What's 15 * 23? And what's the current weather in Paris?")

Modern agent frameworks:

LangChain agents

from langchain.agents import AgentExecutor, create_react_agent
from langchain.tools import tool
from langchain.prompts import PromptTemplate

@tool
def multiply(a: float, b: float) -> float:
"""Multiply two numbers"""
return a * b

@tool
def web_search(query: str) -> str:
"""Search the web"""
return f"Results for {query}"

tools = [multiply, web_search]

Create agent

from langchain.llms import OpenAI
llm = OpenAI(temperature=0)

agent = create_react_agent(llm, tools)
executor = AgentExecutor.from_agent_and_tools(agent=agent, tools=tools)

result = executor.invoke({"input": "What's 15 * 23?"})

---

7. Evaluation, Optimization, and Safety

Evaluation Metrics

Perplexity: How Surprised is the Model?

import torch
import torch.nn.functional as F

def calculate_perplexity(model, dataloader):
"""Perplexity = exp(average_cross_entropy_loss)"""
total_loss = 0
total_tokens = 0

```
model.eval()
with torch.no_grad():
    for batch in dataloader:
        input_ids = batch['input_ids']
        labels = batch['labels']

        logits = model(input_ids)
        loss = F.cross_entropy(
            logits.view(-1, vocab_size),
            labels.view(-1),
            reduction='sum'
        )

        total_loss += loss.item()
        total_tokens += labels.numel()

average_loss = total_loss / total_tokens
perplexity = torch.exp(torch.tensor(average_loss))
```

```
return perplexity
```

Interpretation:

Perplexity = 10: Model thinks test data is ~10x more surprising than training data

Lower is better

GPT-2: ~20-30 on WikiText

GPT-3: ~10-15 on various benchmarks

BLEU Score: Comparing Translations

from nltk.translate.bleu_score import sentence_bleu, corpus_bleu

BLEU-4 (most common)

reference = [["the", "cat", "is", "on", "the", "mat"]]
candidate = ["the", "cat", "is", "on", "mat"]

bleu = sentence_bleu(reference, candidate, weights=(0.25, 0.25, 0.25, 0.25))
print(f"BLEU score: {bleu:.4f}")

Corpus BLEU

references = [
[["the", "cat"], ["a", "cat"]], # Multiple valid translations
[["dog", "runs"]],
]
candidates = [
["the", "cat"],
["dog", "runs"],
]

corpus_bleu_score = corpus_bleu(references, candidates)

ROUGE: Summary Evaluation

from rouge_score import rouge_scorer

scorer = rouge_scorer.RougeScorer(['rouge1', 'rougeL'], use_stemmer=True)

ROUGE-1: overlap of unigrams

ROUGE-L: longest common subsequence

reference = "The quick brown fox jumps over the lazy dog"
prediction = "A fast brown fox jumps over the lazy dog"

scores = scorer.score(reference, prediction)
print(f"ROUGE-1 F1: {scores['rouge1'].fmeasure:.4f}")
print(f"ROUGE-L F1: {scores['rougeL'].fmeasure:.4f}")

LLM-as-Judge: Using LLMs to Evaluate LLMs

def llm_as_judge(question, model_response, reference_response=None):
"""Use GPT-4 to evaluate model output"""

```
if reference_response:
    prompt = f"""You are an expert evaluator. Rate the following response on a sc
```

Question: {question}

Reference (ideal) response: {reference_response}

Model response: {model_response}

Evaluation criteria:

- Correctness: Does it answer the question accurately?
- Completeness: Does it cover all important points?
- Clarity: Is it well-written and easy to understand?

Rating (1-10):"""
else:
prompt = f"""You are an expert evaluator. Rate the following response on a scale of 1-10.

Question: {question}

Response: {model_response}

Evaluation criteria:

- Correctness and accuracy
- Completeness and relevance
- Clarity and presentation

Rating (1-10):"""

```
rating_response = llm.generate(prompt, max_tokens=10)
try:
    rating = int(rating_response.split()[0])
    return min(10, max(1, rating))  # Clamp to 1-10
except:
    return 5  # Default if parsing fails
```

Advantages of LLM-as-Judge:

+ Can evaluate on complex criteria humans care about

+ No need for human annotations

- Biased toward generator LLM style

- Can be gamed

Model Optimization

Quantization: Reducing Precision

import torch
from torch.quantization import quantize_dynamic

Dynamic quantization: quantize weights, keep activations in float32

model = GPTLanguageModel.from_pretrained("gpt2-medium")

quantized_model = quantize_dynamic(
model,
{torch.nn.Linear}, # which modules to quantize
dtype=torch.qint8 # quantize to int8
)

Results:

Model size: 345 MB → 86 MB (4× reduction)

Speed: Slightly slower (dequantization overhead)

Accuracy: <1% degradation on most tasks

Pruning: Removing Unimportant Parameters

```
import torch.nn.utils.prune as prune
```

```
model = GPTLanguageModel.from_pretrained("gpt2-medium")
```

Magnitude pruning: remove smallest weights

```
for name, module in model.named_modules():
if isinstance(module, torch.nn.Linear):
prune.l1_unstructured(module, name='weight', amount=0.3)
# Remove 30% of weights with smallest magnitude
prune.remove(module, 'weight') # Make permanent
```

Results:

Model size: ~30% reduction

Speed: ~2× faster (fewer operations)

Accuracy: 2-5% degradation

Fine-tuning after pruning can recover accuracy

Distillation: Training Small Models with Large Models

class StudentModel(nn.Module):
"""Small model to distill knowledge into"""
def **init**(self, vocab_size):
super().**init**()
# Smaller config: 12M parameters vs 345M teacher
self.embedding = nn.Embedding(vocab_size, 256)
self.transformer = Transformer(
d_model=256,
num_heads=8,
num_layers=4, # vs 12 in teacher
d_ff=1024
)
self.lm_head = nn.Linear(256, vocab_size)

```
def forward(self, input_ids):
    x = self.embedding(input_ids)
    x = self.positional_encoding(x)
    x = self.transformer(x)
    logits = self.lm_head(x)
    return logits
```

def distillation_loss(student_logits, teacher_logits, temperature=4.0):
"""
Soft targets from teacher guide student training
temperature: higher = softer probability distribution
"""
# Soft targets from teacher
teacher_probs = F.softmax(teacher_logits / temperature, dim=-1)

```
# Student log probabilities
student_log_probs = F.log_softmax(student_logits / temperature, dim=-1)

# KL divergence
kl_loss = F.kl_div(student_log_probs, teacher_probs, reduction='batchmean')

return kl_loss * (temperature ** 2)
```

Training loop

teacher = GPTLanguageModel.from_pretrained("gpt2-medium")
teacher.eval() # Don't update teacher

student = StudentModel(vocab_size=50000)
optimizer = torch.optim.AdamW(student.parameters(), lr=1e-4)

for epoch in range(num_epochs):
for batch in dataloader:
input_ids = batch['input_ids']
labels = batch['labels']

```
# Forward passes
with torch.no_grad():
    teacher_logits = teacher(input_ids)

student_logits = student(input_ids)

# Combined loss: distillation + task-specific
alpha = 0.7
distill_loss = distillation_loss(student_logits, teacher_logits)
task_loss = F.cross_entropy(student_logits.view(-1, vocab_size),
                            labels.view(-1))

loss = alpha * distill_loss + (1 - alpha) * task_loss

optimizer.zero_grad()
loss.backward()
optimizer.step()
```

Results:

Student: 12M vs 345M parameters (28× smaller)

Speed: 10× faster

Accuracy: 85-95% of teacher performance

Safety: Detecting and Mitigating Harmful Outputs

Hallucination Detection

class HallucinationDetector:
"""Detect if model is making up facts"""

```
def __init__(self, knowledge_base):
    self.kb = knowledge_base  # Facts we know are true

def check_factuality(self, text):
    """Extract facts and verify against KB"""
    # 1. Extract factual claims from text
    facts = self.extract_facts(text)

    # 2. Verify each fact
    hallucinations = []
    for fact in facts:
        if not self.verify_fact(fact):
            hallucinations.append(fact)

    return hallucinations

def extract_facts(self, text):
    """Simple extraction: proper noun + predicate"""
    # Use NER + dependency parsing
    # Returns: [(subject, verb, object), ...]
    pass

def verify_fact(self, fact):
    """Check if fact exists in KB"""
    subject, verb, obj = fact
    return (subject, verb, obj) in self.kb
```

Usage

detector = HallucinationDetector(knowledge_base)
response = llm.generate("Who won the 2024 Nobel Prize?")
hallucinations = detector.check_factuality(response)

if hallucinations:
print(f"Warning: Potential hallucinations detected: {hallucinations}")

Jailbreak Prevention

class SafetyFilter:
"""Filter harmful outputs"""

```
def __init__(self):
    self.harmful_keywords = [
        "bomb", "illegal", "harmful", "dangerous", ...
    ]
    self.unsafe_intents = [
        "request_illegal_activity",
        "request_violence",
        "request_hate_speech",
    ]

def is_safe(self, text):
    """Check if text contains harmful content"""
    # 1. Keyword filtering
    text_lower = text.lower()
    for keyword in self.harmful_keywords:
        if keyword in text_lower:
            return False

    # 2. Semantic safety check with classifier
    intent = self.classify_intent(text)
    if intent in self.unsafe_intents:
        return False

    return True

def classify_intent(self, text):
    """Use safety classifier"""
```

```
    # Could be a small fine-tuned model
    # Returns: safe, potentially_harmful, clearly_harmful
    pass
```

Usage

safety_filter = SafetyFilter()

prompt = "How do I make a bomb?"
if not safety_filter.is_safe(prompt):
print("Refusing to process unsafe prompt")
else:
response = llm.generate(prompt)

Bias Detection

import numpy as np

def measure_gender_bias(model, words_male, words_female, template):
"""
Measure association bias between gender words and target words
Example: measure association of gender with STEM fields
"""

```
results = {"male": {}, "female": {}}

for target_word in target_words:
    # Prompt with male-associated words
    prompt_male = template.format(gender_word=random.choice(words_male),
                                target=target_word)
    prob_male = model.get_probability_of_next_word(prompt_male, target_word

    # Prompt with female-associated words
    prompt_female = template.format(gender_word=random.choice(words_fema
                                  target=target_word)
    prob_female = model.get_probability_of_next_word(prompt_female, target_w

    results["male"][target_word] = prob_male
    results["female"][target_word] = prob_female

# Compute bias metric
bias = np.mean([
```

```
    results["male"][w] - results["female"][w]
    for w in target_words
])

return bias  # 0 = no bias, >0 = male bias, <0 = female bias
```

Example usage

words_male = ["man", "boy", "father", "son"]
words_female = ["woman", "girl", "mother", "daughter"]
target_words = ["engineer", "nurse", "CEO", "teacher"]
template = "The {gender_word} is a {target}"

bias = measure_gender_bias(model, words_male, words_female, template)
print(f"Gender bias: {bias:.4f}")

---

8. Deployment and Serving

Inference Optimization: vLLM

vLLM is a fast inference library that optimizes memory usage and throughput:

pip install vllm

from vllm import LLM, SamplingParams

Load model

llm = LLM(
model="meta-llama/Llama-2-7b-hf",
tensor_parallel_size=2, # distributed across 2 GPUs
dtype="float16",
gpu_memory_utilization=0.9,
)

Configure sampling

sampling_params = SamplingParams(
temperature=0.7,
top_p=0.95,
max_tokens=256,
)

Batch inference

prompts = [
"What is machine learning?",
"Explain deep learning",
"What are transformers?",
]

outputs = llm.generate(prompts, sampling_params)

for output in outputs:
print(f"Prompt: {output.prompt}")
print(f"Generated: {output.outputs[0].text}\n")

Performance improvements with vLLM:

- Paged Attention: GPU memory as virtual memory (4× throughput)

- Continuous batching: dynamic request scheduling

- KV cache optimizations

- Quantization support

API Deployment

from fastapi import FastAPI
from pydantic import BaseModel

app = FastAPI()

Load model once at startup

llm = LLM(model="meta-llama/Llama-2-7b-hf")

class GenerationRequest(BaseModel):
prompt: str
max_tokens: int = 256
temperature: float = 0.7

class GenerationResponse(BaseModel):
prompt: str

generated_text: str

@app.post("/generate", response_model=GenerationResponse)
async def generate(request: GenerationRequest):
"""Generate text from prompt"""

```
sampling_params = SamplingParams(
    temperature=request.temperature,
    max_tokens=request.max_tokens,
)

outputs = llm.generate(
    [request.prompt],
    sampling_params,
)

return GenerationResponse(
    prompt=request.prompt,
    generated_text=outputs[0].outputs[0].text,
)
```

Run: uvicorn app:app --host 0.0.0.0 --port 8000

Usage:

curl -X POST "http://localhost:8000/generate" \

-H "Content-Type: application/json" \

-d '{"prompt": "What is AI?", "max_tokens": 256}'

Container Deployment (Docker)

Dockerfile

FROM nvidia/cuda:12.1.0-runtime-ubuntu22.04

WORKDIR /app

Install dependencies

RUN apt-get update && apt-get install -y python3-pip
COPY requirements.txt .
RUN pip install -r requirements.txt

Copy application

COPY app.py .
COPY model/ ./model/

Download model weights (if not included)

RUN python3 -c "from transformers import AutoTokenizer, AutoModelForCausalLM;
AutoTokenizer.from_pretrained('meta-llama/Llama-2-7b-hf');
AutoModelForCausalLM.from_pretrained('meta-llama/Llama-2-7b-hf')"

Expose port

EXPOSE 8000

Start server

CMD ["uvicorn", "app:app", "--host", "0.0.0.0", "--port", "8000"]

Build and run

docker build -t my-llm-app .
docker run --gpus all -p 8000:8000 my-llm-app

---

9. Training Your Model on Custom Data

Complete step-by-step guide to train a language model on your own data.

Step 1: Data Preparation

import os
import json
from datasets import Dataset, DatasetDict
from pathlib import Path

class DataPrepipeline:
def **init**(self, data_path, tokenizer, max_seq_len=512):
self.data_path = Path(data_path)
self.tokenizer = tokenizer
self.max_seq_len = max_seq_len

```
def load_raw_data(self):
    """Load raw text files"""
    raw_texts = []

    for file_path in self.data_path.glob("*.txt"):
        with open(file_path, 'r', encoding='utf-8') as f:
            raw_texts.append(f.read())

    return raw_texts

def clean_text(self, text):
    """Clean and normalize text"""
    # Remove extra whitespace
    text = ' '.join(text.split())

    # Remove control characters
    text = ''.join(ch for ch in text if ord(ch) >= 32 or ch in '\n\t')

    # Remove very long sequences (likely corrupted)
    if len(text) > 1e7:
        text = text[:int(1e7)]

    return text

def tokenize_function(self, examples):
```

```
        """Tokenize text with special handling for document boundaries"""
        # Add document separator token
        tokenized = self.tokenizer(
            examples['text'],
            truncation=False,  # Don't truncate yet
            return_attention_mask=False,
        )

        # Concatenate all tokens with </s> separator
        all_ids = []
        for token_ids in tokenized['input_ids']:
            all_ids.extend(token_ids + [self.tokenizer.eos_token_id])

        # Create chunks of max_seq_len
        total_len = len(all_ids)
        chunk_len = (total_len // self.max_seq_len) * self.max_seq_len
        all_ids = all_ids[:chunk_len]

        chunks = [all_ids[i:i+self.max_seq_len] for i in range(0, len(all_ids), self.max_

        return {
            'input_ids': chunks,
            'labels': chunks.copy(),  # For language modeling, labels = input_ids shifted
        }

    def prepare_dataset(self, train_split=0.9, val_split=0.05, test_split=0.05):
        """Prepare train/val/test splits"""

        # 1. Load raw data
        raw_texts = self.load_raw_data()
        print(f"Loaded {len(raw_texts)} files")

        # 2. Clean data
        cleaned_texts = [self.clean_text(text) for text in raw_texts]
        print(f"Cleaned {len(cleaned_texts)} texts")

        # 3. Create dataset
        dataset = Dataset.from_dict({'text': cleaned_texts})
```

```
# 4. Tokenize
tokenized = dataset.map(
    self.tokenize_function,
    batched=True,
    batch_size=1000,
    remove_columns=['text'],
)

# 5. Split
splits = tokenized.train_test_split(
    train_size=train_split,
    test_size=1 - train_split,
    seed=42
)

# Further split test set into val and test
val_test = splits['test'].train_test_split(
    train_size=val_split / (1 - train_split),
    seed=42
)

final_dataset = DatasetDict({
    'train': splits['train'],
    'validation': val_test['train'],
    'test': val_test['test'],
})

print(f"Dataset split:")
print(f"  Train: {len(final_dataset['train'])} samples")
print(f"  Val: {len(final_dataset['validation'])} samples")
print(f"  Test: {len(final_dataset['test'])} samples")

return final_dataset
```

Usage

from transformers import AutoTokenizer

tokenizer = AutoTokenizer.from_pretrained("gpt2")
prep = DataPrepipeline("./data/raw", tokenizer, max_seq_len=512)
dataset = prep.prepare_dataset()

Step 2: Initialize Model

from transformers import AutoConfig, AutoModelForCausalLM
import torch

class ModelInitializer:
@staticmethod
def create_custom_config(model_size="small"):
"""Create custom model configuration"""

```
configs = {
    "small": {
        "hidden_size": 256,
        "num_hidden_layers": 4,
        "num_attention_heads": 8,
        "intermediate_size": 1024,
    },
    "medium": {
        "hidden_size": 512,
        "num_hidden_layers": 8,
        "num_attention_heads": 8,
        "intermediate_size": 2048,
    },
    "large": {
        "hidden_size": 768,
        "num_hidden_layers": 12,
        "num_attention_heads": 12,
        "intermediate_size": 3072,
    },
}

config_dict = configs[model_size]
config = AutoConfig.from_pretrained("gpt2")
```

```
        for key, value in config_dict.items():
            setattr(config, key, value)

        return config

    @staticmethod
    def initialize_weights(model, std=0.02):
        """Initialize weights (important for training stability)"""
        for name, param in model.named_parameters():
            if "weight" in name and param.dim() > 1:
                torch.nn.init.normal_(param, mean=0.0, std=std)
            elif "bias" in name:
                torch.nn.init.constant_(param, 0.0)

    @staticmethod
    def create_model(model_size="medium", use_pretrained_weights=False):
        """Create model from scratch or with pretrained weights"""

        if use_pretrained_weights:
            model = AutoModelForCausalLM.from_pretrained("gpt2-medium")
            print("Loaded pretrained GPT-2 weights")
        else:
            config = ModelInitializer.create_custom_config(model_size)
            model = AutoModelForCausalLM.from_config(config)
            ModelInitializer.initialize_weights(model)
            print(f"Created {model_size} model from scratch")

        # Count parameters
        total_params = sum(p.numel() for p in model.parameters())
        trainable_params = sum(p.numel() for p in model.parameters() if p.requires_

        print(f"Model size: {total_params/1e6:.1f}M parameters ({trainable_params/1e

        return model
```

Usage

model = ModelInitializer.create_model(model_size="medium", use_pretrained_weights=False)

Step 3: Training Configuration

from transformers import TrainingArguments, Trainer
import wandb

Initialize weights & biases for monitoring

wandb.init(project="llm-training")

training_args = TrainingArguments(
output_dir="./outputs",

```
# Training
num_train_epochs=3,
per_device_train_batch_size=32,
per_device_eval_batch_size=32,
gradient_accumulation_steps=1,

# Optimization
learning_rate=5e-5,
warmup_steps=1000,
weight_decay=0.01,
lr_scheduler_type="cosine",
max_grad_norm=1.0,

# Evaluation
evaluation_strategy="steps",
eval_steps=500,
save_strategy="steps",
save_steps=500,
save_total_limit=3,

# Logging
logging_dir="./logs",
logging_steps=100,
```

```
    # Hardware
    fp16=True,  # Mixed precision
    gradient_checkpointing=True,  # Save memory
    dataloader_num_workers=4,

    # Misc
    seed=42,
    report_to="wandb",
```

)

Initialize trainer

trainer = Trainer(
model=model,
args=training_args,
train_dataset=dataset['train'],
eval_dataset=dataset['validation'],
data_collator=transformers.default_data_collator,
)

Step 4: Training Loop

Train

train_result = trainer.train()

Evaluate

eval_results = trainer.evaluate()
print(f"Validation loss: {eval_results['eval_loss']:.4f}")
print(f"Validation perplexity: {torch.exp(torch.tensor(eval_results['eval_loss'])):.4f}")

Save model

model.save_pretrained("./final-model")
tokenizer.save_pretrained("./final-model")

Test

test_results = trainer.evaluate(eval_dataset=dataset['test'])
print(f"Test loss: {test_results['eval_loss']:.4f}")

Step 5: Inference on Custom Model

from transformers import pipeline

Load trained model

model = AutoModelForCausalLM.from_pretrained("./final-model")
tokenizer = AutoTokenizer.from_pretrained("./final-model")

Create text generation pipeline

generator = pipeline("text-generation", model=model, tokenizer=tokenizer, device=0)

Generate

prompt = "The future of artificial intelligence"
output = generator(
prompt,
max_length=256,
num_return_sequences=3,
temperature=0.7,
top_p=0.9,
do_sample=True,
)

for i, seq in enumerate(output):
print(f"\nGeneration {i+1}:")
print(seq['generated_text'])

---

10. Replicating GPT-2 from Scratch

Building GPT-2 from first principles to understand architecture deeply.

Architecture Overview

import torch
import torch.nn as nn
import torch.nn.functional as F
import math

class GPT2Config:
"""Configuration for GPT-2"""
def **init**(

self,
vocab_size=50257,
block_size=1024,
n_layer=12,
n_head=12,
n_embd=768,
dropout=0.1,
):
self.vocab_size = vocab_size
self.block_size = block_size # Maximum sequence length
self.n_layer = n_layer # Number of transformer blocks
self.n_head = n_head # Number of attention heads
self.n_embd = n_embd # Embedding dimension
self.dropout = dropout

class CausalSelfAttention(nn.Module):
"""Multi-head self-attention with causal mask"""

```
def __init__(self, config):
    super().__init__()
    assert config.n_embd % config.n_head == 0

    self.n_head = config.n_head
    self.n_embd = config.n_embd
    self.head_dim = config.n_embd // config.n_head

    # Project to Q, K, V all at once
    self.c_attn = nn.Linear(config.n_embd, 3 * config.n_embd)
    self.c_proj = nn.Linear(config.n_embd, config.n_embd)

    # Dropout
    self.attn_dropout = nn.Dropout(config.dropout)
    self.proj_dropout = nn.Dropout(config.dropout)

    # Register causal mask
    self.register_buffer(
        "bias",
        torch.tril(torch.ones(config.block_size, config.block_size)).view(
            1, 1, config.block_size, config.block_size
        )
    )
```

```python
def forward(self, x):
    B, T, C = x.shape  # batch_size, seq_len, n_embd

    # Project to Q, K, V
    qkv = self.c_attn(x)  # (B, T, 3*C)
    q, k, v = qkv.split(self.n_embd, dim=2)  # Each (B, T, C)

    # Reshape for multi-head: (B, T, C) → (B, T, n_head, head_dim)
    q = q.reshape(B, T, self.n_head, self.head_dim).transpose(1, 2)
    k = k.reshape(B, T, self.n_head, self.head_dim).transpose(1, 2)
    v = v.reshape(B, T, self.n_head, self.head_dim).transpose(1, 2)
    # Now: (B, n_head, T, head_dim)

    # Scaled dot-product attention
    scores = torch.matmul(q, k.transpose(-2, -1)) / math.sqrt(self.head_dim)
    # scores: (B, n_head, T, T)

    # Apply causal mask
    scores = scores.masked_fill(self.bias[:, :, :T, :T] == 0, float('-inf'))

    # Softmax
    attn_weights = F.softmax(scores, dim=-1)
    attn_weights = self.attn_dropout(attn_weights)

    # Apply to values
    attn_output = torch.matmul(attn_weights, v)  # (B, n_head, T, head_dim)

    # Concatenate heads
    attn_output = attn_output.transpose(1, 2).contiguous()
    # (B, T, n_head, head_dim)
    attn_output = attn_output.reshape(B, T, C)  # (B, T, C)

    # Project
    out = self.c_proj(attn_output)
    out = self.proj_dropout(out)

    return out
```

class MLP(nn.Module):
"""Feed-forward network in transformer block"""

```
def __init__(self, config):
    super().__init__()
    self.c_fc = nn.Linear(config.n_embd, 4 * config.n_embd)  # Expand
    self.c_proj = nn.Linear(4 * config.n_embd, config.n_embd)  # Contract
    self.dropout = nn.Dropout(config.dropout)

def forward(self, x):
    x = F.gelu(self.c_fc(x))  # GELU activation
    x = self.c_proj(x)
    x = self.dropout(x)
    return x
```

class TransformerBlock(nn.Module):
"""Single transformer block"""

```
def __init__(self, config):
    super().__init__()
    self.ln_1 = nn.LayerNorm(config.n_embd)
    self.attn = CausalSelfAttention(config)
    self.ln_2 = nn.LayerNorm(config.n_embd)
    self.mlp = MLP(config)

def forward(self, x):
    # Pre-normalization
    x = x + self.attn(self.ln_1(x))  # Residual connection
    x = x + self.mlp(self.ln_2(x))
    return x
```

class GPT2(nn.Module):
"""Full GPT-2 model"""

```
def __init__(self, config):
    super().__init__()
    self.config = config
```

```
    # Embeddings
    self.transformer = nn.ModuleDict(dict(
        wte=nn.Embedding(config.vocab_size, config.n_embd),  # Token embedding
        wpe=nn.Embedding(config.block_size, config.n_embd),  # Position embedd
        drop=nn.Dropout(config.dropout),
        h=nn.ModuleList([TransformerBlock(config) for _ in range(config.n_layer)
        ln_f=nn.LayerNorm(config.n_embd),
    ))

    # Language modeling head
    self.lm_head = nn.Linear(config.n_embd, config.vocab_size)

    # Weight tying: share weights between embedding and output projection
    self.transformer.wte.weight = self.lm_head.weight

    # 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)

def forward(self, idx, targets=None):
    B, T = idx.shape

    # Get embeddings
    token_emb = self.transformer.wte(idx)  # (B, T, n_embd)
    pos_emb = self.transformer.wpe(torch.arange(T, device=idx.device))  # (T, n_e
    x = self.transformer.drop(token_emb + pos_emb)

    # Apply transformer blocks
    for block in self.transformer.h:
        x = block(x)
```

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

    # Project to vocabulary
    logits = self.lm_head(x)  # (B, T, vocab_size)

    # Compute loss if targets provided
    loss = None
    if targets is not None:
        loss = F.cross_entropy(
            logits.view(-1, self.config.vocab_size),
            targets.view(-1)
        )

    return logits, loss

@torch.no_grad()
def generate(self, idx, max_new_tokens, temperature=1.0, top_k=None):
    """Generate tokens autoregressively"""
    for _ in range(max_new_tokens):
        # Crop input to block_size
        idx_cond = idx if idx.shape[1] <= self.config.block_size else idx[:, -self.confi

        # Forward pass
        logits, _ = self(idx_cond)

        # Get logits for next token
        logits = logits[:, -1, :] / temperature

        # Optional top-k sampling
        if top_k is not None:
            v, _ = torch.topk(logits, top_k)
            logits[logits < v[:, [-1]]] = float('-inf')

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

```
        # Append
        idx = torch.cat([idx, idx_next], dim=1)

    return idx
```

Training

config = GPT2Config()
model = GPT2(config)
optimizer = torch.optim.AdamW(model.parameters(), lr=6e-4)

for epoch in range(num_epochs):
for batch_idx, (X, Y) in enumerate(dataloader):
logits, loss = model(X, Y)

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

    if batch_idx % 100 == 0:
        print(f"Epoch {epoch}, Batch {batch_idx}, Loss: {loss.item():.4f}")
```

Inference

model.eval()
context = torch.zeros((1, 1), dtype=torch.long) # Start token
generated = model.generate(context, max_new_tokens=100, temperature=0.9)

Optimizing Training Speed

class OptimizedGPT2(GPT2):
"""GPT-2 with optimizations for faster training"""

```
@torch.compile  # PyTorch 2.0+ JIT compilation
def forward_compiled(self, idx, targets=None):
    return self.forward(idx, targets)
```

Enable mixed precision

from torch.cuda.amp import autocast, GradScaler

model = GPT2(config).to('cuda')
scaler = GradScaler()
optimizer = torch.optim.AdamW(model.parameters(), lr=6e-4)

for epoch in range(num_epochs):
for X, Y in dataloader:
X, Y = X.to('cuda'), Y.to('cuda')

```
    with autocast(dtype=torch.float16):
        logits, loss = model(X, Y)

    optimizer.zero_grad()
    scaler.scale(loss).backward()
    scaler.unscale_(optimizer)
    torch.nn.utils.clip_grad_norm_(model.parameters(), 1.0)
    scaler.step(optimizer)
    scaler.update()
```

Use Flash Attention for faster attention computation

Requires: pip install flash-attn

from flash_attn import flash_attn_func

In CausalSelfAttention.forward:

attn_output = flash_attn_func(q, k, v, causal=True)

Evaluation on GPT-2 Benchmarks

def evaluate_on_hellaswag(model, tokenizer, num_examples=100):
"""Evaluate on HellaSwag benchmark (reading comprehension)"""

```
from datasets import load_dataset

dataset = load_dataset("hellaswag")

correct = 0

for i, example in enumerate(dataset['test'][:num_examples]):
    context = example['ctx']
    endings = example['endings']
    label = int(example['label'])

    best_score = float('-inf')
    best_choice = 0

    for j, ending in enumerate(endings):
        full_text = context + " " + ending
        tokens = tokenizer.encode(full_text)
        tokens = torch.tensor(tokens).unsqueeze(0)

        with torch.no_grad():
            logits, _ = model(tokens)

        # Score: average log probability
        probs = F.softmax(logits[0, -len(ending.split()):], dim=-1)
        score = torch.log(probs + 1e-8).sum().item()

        if score > best_score:
            best_score = score
            best_choice = j

    if best_choice == label:
        correct += 1

accuracy = correct / num_examples
```

```
print(f"HellaSwag accuracy: {accuracy:.3f}")
return accuracy
```

Run evaluation

accuracy = evaluate_on_hellaswag(model, tokenizer)

---

11. Production LLM Systems

Scaling Considerations

Model parallelism for very large models

from torch.nn.parallel import DataParallel, DistributedDataParallel

Single node, multiple GPUs: DataParallel

model = DataParallel(model, device_ids=[0, 1, 2, 3])

Multi-node, distributed training: DistributedDataParallel

model = DistributedDataParallel(model, device_ids=[rank])

For models > 40B parameters, use tensor parallelism

(model layers split across GPUs)

from megatron.distributed import initialize_megatron

Or use inference engines that handle it:

- vLLM (recommended)

- Text Generation Inference (TGI)

- DeepSpeed Inference

Monitoring and Observability

import logging
from torch.utils.tensorboard import SummaryWriter

Setup logging

logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(**name**)

TensorBoard for metrics

writer = SummaryWriter()

def log_metrics(step, loss, lr, grad_norm):
"""Log training metrics"""
logger.info(f"Step {step}: loss={loss:.4f}, lr={lr:.6f}, grad_norm={grad_norm:.2f}")

```
writer.add_scalar('Loss/train', loss, step)
writer.add_scalar('Learning_rate', lr, step)
writer.add_scalar('Gradient_norm', grad_norm, step)
```

Usage in training loop

for step, (X, Y) in enumerate(dataloader):
logits, loss = model(X, Y)

```
optimizer.zero_grad()
loss.backward()
grad_norm = torch.nn.utils.clip_grad_norm_(model.parameters(), 1.0)
optimizer.step()

if step % 100 == 0:
    lr = optimizer.param_groups[0]['lr']
    log_metrics(step, loss.item(), lr, grad_norm)
```

Conclusion

This comprehensive guide has covered:

1. **Foundations**: NLP, embeddings, neural networks, optimization
2. **Architecture**: Transformers, attention, position encoding
3. **Pre-training**: Autoregressive and masked language modeling
4. **Fine-tuning**: Full tuning, LoRA, QLoRA, instruction tuning, RLHF
5. **Prompting**: Zero/few-shot, CoT, ToT techniques
6. **RAG & Agents**: Knowledge retrieval, tool use
7. **Evaluation**: Metrics, benchmarking, safety
8. **Deployment**: Inference optimization, serving
9. **Custom Training**: Complete pipeline from data to inference
10. **GPT-2 Replication**: Understanding architecture deeply
11. **Production**: Scaling, monitoring, optimization

Next steps:

- Start with small experiments (GPT-2 small, 124M params)
- Scale up gradually as you understand the concepts
- Use modern frameworks: HuggingFace, LangChain, vLLM
- Contribute to open-source LLM projects
- Stay updated with latest techniques (Vision Transformers, Multimodal models, etc.)

The field of LLMs evolves rapidly. The fundamentals outlined here will remain relevant, but always check latest research papers and community practices.