GPT-2 Tutorial: Understanding GPT from Scratch

A complete guide to understanding every component of a GPT (Generative Pre-trained Transformer) language model, explained as if you're learning it for the first time.

Author: Kevin Thomas (ket189@pitt.edu)

Note: This tutorial uses simplified, educational variable names (like `embedding_dim`) for clarity. The production code in `micro_gpt.py` follows OpenAI GPT-2 naming conventions (`n_embd`, `n_head`, `n_layer`).

Table of Contents

---

1. Introduction: What is a Language Model?

The Big Picture

Imagine you're playing a game where someone gives you the beginning of a sentence, and you have to guess the next word. For example:

- "The cat sat on the _____" → You'd probably guess "mat" or "chair"
- "I love eating _____" → You might guess "pizza" or "ice cream"

That's exactly what a language model does! It predicts what word (or token) comes next in a sequence.

Why This Matters

Language models power: - ChatGPT and similar chatbots - Auto-complete in your phone - Translation services - Code completion in IDEs

How GPT is Different

GPT (Generative Pre-trained Transformer) uses a special architecture called a **transformer** that can: 1. Look at all previous words at once (not just the last few) 2. Understand which words are important to pay attention to 3. Generate coherent, human-like text

Let's break down every single piece!

---

2. Tokenization: Breaking Text into Pieces

What is Tokenization?

Before a computer can understand text, we need to convert words into numbers. But we don't just convert whole words - we break text into smaller pieces called **tokens**.

Example

The sentence "Hello, world!" might be tokenized as:

```
"Hello, world!" → ["Hello", ",", " world", "!"] → [15496, 11, 995, 0]
```

Each token gets a unique number (an ID) from a vocabulary.

Why Not Just Use Words?

Tokens are better than words because: - **Handles unknown words**: "unhappiness" = "un" + "happiness" - **Smaller vocabulary**: We only need ~50,000 tokens instead of millions of words - **Works for any language**: Even handles code and symbols!

In Our Code

```
# We use tiktoken (OpenAI's tokenizer)
encoding = tiktoken.get_encoding("gpt2")
tokens = encoding.encode("Hello, world!")  # [15496, 11, 995, 0]
```

Key Point: After tokenization, our text is now a sequence of numbers (token IDs).

---

3. Token Embeddings: Numbers to Vectors

The Problem

We have token IDs like `[15496, 11, 995]`, but neural networks can't learn much from single numbers. We need **vectors** (lists of numbers) that can capture meaning.

What is an Embedding?

An embedding converts each token ID into a vector (a list of many numbers). Think of it like this:

Before (Token ID):

```
"cat" → 145
```

After (Embedding Vector):

```
"cat" → [0.2, -0.5, 0.8, 0.1, -0.3, ...]  (maybe 64 or 384 numbers!)
```

Why Vectors?

Vectors can capture relationships: - "king" - "man" + "woman"   "queen" - Similar words have similar vectors - The model learns these relationships during training!

The Analogy

Think of embeddings like describing a person: - Token ID: Just a name "John" (one number) - Embedding: Height, weight, age, personality traits, interests… (many numbers)

The more numbers, the more detailed the description!

In Our Code

```python
class GPT2(nn.Module):
    def __init__(self, vocab_size=50257, embedding_dim=384, ...):
        # Create embedding table: vocab_size × embedding_dim
        self.token_embedding = nn.Embedding(vocab_size, embedding_dim)
```

If `vocab_size=50257` and `embedding_dim=384`: - We have 50,257 tokens - Each token becomes a 384-dimensional vector - Total: 50,257 × 384 = 19 million learnable numbers!

Example

```python
# Token IDs (integers)
tokens = torch.tensor([145, 892, 33])  # Shape: (3,)

# After embedding (vectors)
embedded = token_embedding(tokens)  # Shape: (3, 384)
# Each token is now a 384-dimensional vector!
```

Key Point: Embeddings transform token IDs into rich, meaningful vectors that capture semantic information.

---

4. Positional Embeddings: Teaching Position

The Problem

Look at these two sentences: 1. "The cat chased the mouse" 2. "The mouse chased the cat"

Same words, **completely different meanings!** The model needs to know **where** each word appears in the sequence.

The Solution: Positional Embeddings

Just like we embed tokens, we also embed positions:

```
Position 0 → [0.5, 0.2, -0.1, ...]
Position 1 → [0.3, -0.4, 0.6, ...]
Position 2 → [0.1, 0.8, -0.2, ...]
...
```

How It Works

For the sequence "The cat sat":

Step 1: Token Embeddings

```
"The" → token_embedding[145]   → [0.2, -0.5, 0.8, ...]
"cat" → token_embedding[892]   → [0.1, 0.3, -0.2, ...]
"sat" → token_embedding[33]    → [0.4, 0.1, 0.5, ...]
```

Step 2: Position Embeddings

```
Position 0 → position_embedding[0] → [0.5, 0.2, -0.1, ...]
Position 1 → position_embedding[1] → [0.3, -0.4, 0.6, ...]
Position 2 → position_embedding[2] → [0.1, 0.8, -0.2, ...]
```

Step 3: Add Them Together!

```
"The" (pos 0) = token_emb[145] + pos_emb[0]
"cat" (pos 1) = token_emb[892] + pos_emb[1]
"sat" (pos 2) = token_emb[33]  + pos_emb[2]
```

Why Addition?

We add (not concatenate) because: 1. Keeps the same dimension (384 stays 384) 2. The model learns to separate content from position during training 3. It's simpler and works well!

In Our Code

```python
def _embed_tokens(self, idx: torch.Tensor) -> torch.Tensor:
    T = idx.size(1)  # Sequence length
    # Get token embeddings
    tok_emb = self.token_embedding(idx)  # (B, T, embedding_dim)
    # Get position embeddings
    pos_emb = self.position_embedding(torch.arange(T))  # (T, embedding_dim)
    # Add them together!
    return tok_emb + pos_emb  # (B, T, embedding_dim)
```

Example with Numbers

```python
# Input: 2 sequences of 3 tokens each
idx = torch.tensor([[145, 892, 33],
                    [567, 234, 891]])  # Shape: (2, 3)

# After token embedding
tok_emb.shape  # (2, 3, 384)
```

```
# After position embedding
pos_emb.shape  # (3, 384) - same for all sequences!

# After adding
result.shape   # (2, 3, 384)
```

Key Point: Positional embeddings give the model a sense of order, so it knows "the cat chased the mouse" is different from "the mouse chased the cat".

---

5. The Residual Stream: The Information Highway

The Problem

Imagine you're passing a note through a chain of 6 people. By the time it reaches person 6, the original message might be garbled or lost!

Neural networks have the same problem - information can get lost or corrupted as it passes through many layers.

The Solution: Residual Connections (Skip Connections)

Instead of just passing information forward, we **add a shortcut**:

```
Before (no residual):
Input → Layer 1 → Layer 2 → Layer 3 → Output

After (with residual):
Input → Layer 1  → + → Layer 2  → + → Layer 3 → Output
                   ↑                  ↑
```

How It Works

Instead of:

```
x = layer(x)  # x is completely replaced
```

We do:

```
x = x + layer(x)  # Add the layer output to the original x!
```

The Analogy

Think of it like editing a document: - **Without residual**: Rewrite the whole document from scratch each time - **With residual**: Start with the original and only make changes/additions

The original information always flows through!

Why This is Powerful

1. **Gradients flow better**: Makes training deep networks easier
2. **Information preservation**: Original input never gets lost

3. **Allows depth**: We can stack 100+ layers successfully!

The Residual Stream Concept

In transformers, we think of the data flow as a **residual stream** - a river of information that flows through the network, with each layer adding to it:

```
Input embeddings
    ↓
+ Attention (adds context understanding)
    ↓
+ Feed-forward (adds feature transformations)
    ↓
+ Attention (adds more context)
    ↓
+ Feed-forward (adds more features)
    ↓
Output
```

In Our Code

```python
class Block(nn.Module):
    def forward(self, x: torch.Tensor) -> torch.Tensor:
        # Residual connection around attention
        x = x + self.sa(self.ln1(x))  # x = x + attention_output

        # Residual connection around feed-forward
        x = x + self.ffwd(self.ln2(x))  # x = x + feedforward_output

        return x
```

Example Flow

```python
# Start with embeddings
x = embeddings  # Shape: (2, 10, 384)

# Block 1
x = x + attention(x)     # Add attention's contribution
x = x + feedforward(x)   # Add feedforward's contribution

# Block 2
x = x + attention(x)     # Add more attention
x = x + feedforward(x)   # Add more processing

# Original embeddings are still "in there" plus all the additions!
```

Key Point: Residual connections create an "information superhighway" that preserves important information while allowing the network to learn complex transformations.

---

6. Self-Attention: Learning What's Important

The Big Idea

When you read a sentence, you naturally pay attention to certain words more than others. Attention mechanisms let the model do the same!

The Motivation

Consider: "The animal didn't cross the street because **it** was too tired"

What does "it" refer to? The animal or the street? You know it's the animal because you pay attention to context clues like "tired".

Self-attention lets the model learn these relationships!

The Three Players: Query, Key, Value

Think of attention like a search engine:

1. **Query (Q)**: "What am I looking for?"
   - Like typing a search query: "tired animal"
2. **Key (K)**: "What do I have?"
   - Like document titles: "street description", "animal description"
3. **Value (V)**: "What information can I provide?"
   - The actual content of the documents

The Attention Mechanism (Step by Step)

Step 1: Create Q, K, V For each word, we create three vectors:

```
# For each token embedding (384-dimensional)
q = query_layer(x)    # "What should I look for?"
k = key_layer(x)      # "What do I represent?"
v = value_layer(x)    # "What information do I carry?"
```

Step 2: Calculate Attention Scores We compare each Query with all Keys to see how relevant they are:

```
# Dot product: how similar is each query to each key?
scores = q @ k.transpose(-2, -1)  # (B, T, T)
```

Example: For "it" in our sentence:

```
"it" queries:
  - "animal" key: HIGH similarity (0.9)
  - "street" key: LOW similarity (0.2)
  - "tired" key: MEDIUM similarity (0.6)
```

Step 3: Scale the Scores Divide by √(head_size) to keep gradients stable:

```
scores = scores / (head_size ** 0.5)
```

Why? Prevents scores from getting too large (which would make softmax produce near-0 or near-1 values).

Step 4: Causal Masking (for GPT) **Critical for language models!** We can't look at future words:

```
# Mask out future positions (set them to -infinity)
scores = scores.masked_fill(mask == 0, float('-inf'))
```

Example:

```
Original scores:      After masking:
[0.5  0.3  0.8]      [0.5  -inf -inf]  ← position 0 can't see 1,2
[0.4  0.6  0.2]  →   [0.4  0.6  -inf]  ← position 1 can't see 2
[0.7  0.5  0.9]      [0.7  0.5   0.9]  ← position 2 can see all
```

This ensures: **We only predict based on past context, never future!**

Step 5: Softmax (Make it a Probability) Convert scores to probabilities that sum to 1:

```
attention_weights = softmax(scores)  # (B, T, T)
```

Example for "it":

```
Before softmax:    After softmax:
animal: 0.9   →    animal: 0.55  (55% attention)
street: 0.2   →    street: 0.12  (12% attention)
tired:  0.6   →    tired:  0.33  (33% attention)
                   Total: 1.00   (100%)
```

Step 6: Weighted Sum Use attention weights to mix the Values:

```
output = attention_weights @ v  # (B, T, head_size)
```

What this means: - "it" becomes 55% animal + 12% street + 33% tired - The output captures context from all related words!

Complete Example

```
# Sentence: "The cat sat"
# Positions: 0:"The", 1:"cat", 2:"sat"

# For position 2 ("sat"), the attention might look like:
Attention weights for "sat":
  ↓
  0.1  ← "The"   (10% attention - not very relevant)
  0.6  ← "cat"   (60% attention - the subject!)
  0.3  ← "sat"   (30% attention - the word itself)

# Output for "sat" becomes:
output = 0.1 * value["The"] + 0.6 * value["cat"] + 0.3 * value["sat"]
```

In Our Code

```python
class SelfAttentionHead(nn.Module):
    def forward(self, x: torch.Tensor) -> torch.Tensor:
        # Step 1: Create Q, K, V
        k = self.key(x)    # (B, T, head_size)
        q = self.query(x)  # (B, T, head_size)
        v = self.value(x)  # (B, T, head_size)

        # Steps 2-5: Compute attention weights
        wei = q @ k.transpose(-2, -1)  # (B, T, T) - Step 2
        wei = wei / (k.size(-1) ** 0.5)  # Step 3: scale
        wei = wei.masked_fill(self.tril[:T, :T] == 0, float('-inf'))  # Step 4: mask
        wei = F.softmax(wei, dim=-1)  # Step 5: normalize

        # Step 6: Weighted sum
        output = wei @ v  # (B, T, head_size)
        return output
```

Visualization

```
Input: [The, cat, sat]

After attention, "sat" has looked at its context:
"sat" = 10% "The" + 60% "cat" + 30% "sat" (itself)

So "sat" now knows it's related to a "cat" doing the action!
```

Key Point: Self-attention lets each word look at all other words (in the past) and decide which ones are important for understanding its meaning in context.

---

7. Multi-Head Attention: Multiple Perspectives

The Problem with Single Attention

One attention head can only learn one type of relationship. But language has many types of relationships:

- **Syntactic**: "What's the verb?" "What's the subject?"
- **Semantic**: "What does this refer to?" "What's related?"
- **Positional**: "What came recently?" "What's nearby?"

The Solution: Multiple Heads

Instead of one attention mechanism, use many **in parallel**!

Think of it like asking multiple experts: - Expert 1 (Head 1): Focuses on grammar relationships - Expert 2 (Head 2): Focuses on semantic meaning - Expert 3 (Head 3): Focuses on nearby words - Expert 4 (Head 4): Focuses on sentence structure

Each head learns to pay attention to different things!

How It Works

Step 1: Create Multiple Heads Instead of one attention head with `head_size=384`, we create multiple smaller heads:

```
# Single head: 384 dimensions
# 6 heads: 6 × 64 = 384 dimensions total

heads = [
    SelfAttentionHead(embedding_dim=384, head_size=64),  # Head 1
    SelfAttentionHead(embedding_dim=384, head_size=64),  # Head 2
    SelfAttentionHead(embedding_dim=384, head_size=64),  # Head 3
    # ... 3 more heads
]
```

Step 2: Run All Heads in Parallel Each head processes the input independently:

```
head_outputs = [head(x) for head in heads]
# Each output: (B, T, 64)
```

Step 3: Concatenate Join all head outputs side-by-side:

```
output = torch.cat(head_outputs, dim=-1)  # (B, T, 384)
# 64 + 64 + 64 + 64 + 64 + 64 = 384
```

Step 4: Project Mix the head outputs together:

```
output = projection_layer(output)  # (B, T, 384)
```

Example

```
Sentence: "The cat sat on the mat"

Head 1 might learn:
  "sat" pays attention to "cat" (subject-verb relationship)

Head 2 might learn:
  "sat" pays attention to "on" (verb-preposition relationship)

Head 3 might learn:
  "sat" pays attention to nearby words (local context)

Head 4 might learn:
  "sat" pays attention to the start of sentence (sentence structure)

Combined output:
  "sat" has information about its subject, object, position, and context!
```

Why Smaller Heads?

Instead of one 384-dim head, we use 6 × 64-dim heads:

Advantages: 1. **Specialization**: Each head learns different patterns 2. **Same computation**: 6×64 = 384 (same total dimensions) 3. **More flexible**: Can learn multiple relationships simultaneously

The Math:

```
Option 1: 1 head × 384 dimensions = 384 total
Option 2: 6 heads × 64 dimensions = 384 total (same!)

But Option 2 can learn 6 different attention patterns!
```

In Our Code

```python
class MultiHeadAttention(nn.Module):
    def __init__(self, embedding_dim, num_heads=6):
        head_size = embedding_dim // num_heads  # 384 // 6 = 64

        # Create multiple heads
        self.heads = nn.ModuleList([
            SelfAttentionHead(embedding_dim, head_size)
            for _ in range(num_heads)
        ])

        # Projection layer to mix head outputs
        self.proj = nn.Linear(embedding_dim, embedding_dim)

    def forward(self, x):
        # Run all heads in parallel
        out = torch.cat([h(x) for h in self.heads], dim=-1)
        # Mix them together
        return self.proj(out)
```

Visualization

```
Input: (B, T, 384)
         ↓

    ↓         ↓    ↓    ↓    ↓    ↓
  Head1    Head2 Head3 Head4 Head5 Head6
  (64)     (64)  (64)  (64)  (64)  (64)



         ↓
    Concatenate
         ↓
      (B, T, 384)
```

```
         ↓
    Projection
         ↓
    Output: (B, T, 384)
```

Real-World Analogy

Imagine writing a book review. You might consider: - Plot (Head 1) - Characters (Head 2) - Writing style (Head 3) - Pacing (Head 4) - Themes (Head 5) - Overall impression (Head 6)

Each "head" focuses on one aspect, then you combine all perspectives for your final review!

Key Point: Multi-head attention lets the model look at sequences from multiple perspectives simultaneously, learning different types of relationships in parallel.

---

8. Feed-Forward Networks: Processing Information

What Comes After Attention?

Attention helps words understand their context by looking at other words. But we also need to **process** and **transform** that information!

That's where Feed-Forward Networks (FFN) come in.

What is a Feed-Forward Network?

It's a simple two-layer neural network that processes each position independently:

`FFN(x) = ReLU(x @ W1 + b1) @ W2 + b2`

Think of it as: **expand → transform → compress**

The Architecture

```
Input: 384 dimensions
   ↓
Expand: 384 → 1536 dimensions (4× larger!)
   ↓
ReLU activation (non-linearity)
   ↓
Compress: 1536 → 384 dimensions (back to original)
   ↓
Output: 384 dimensions
```

Why Expand Then Compress?

The Analogy: Imagine explaining a concept: 1. **Expand**: Break it down into detailed examples (more information) 2. **Process**: Think about each detail, make connections 3. **Compress**: Synthesize back to a clear, refined explanation

In the network: 1. **Expand (384→1536)**: Create more "features" to work with 2. **ReLU**: Add non-linearity (enables learning complex patterns) 3. **Compress (1536→384)**: Distill the useful information

The 4× Rule

Transformers typically use a 4× expansion ratio: - Input: 384 dimensions - Hidden: 384 × 4 = 1536 dimensions - Output: 384 dimensions

Why 4×? - More capacity for learning complex transformations - But not so large that training becomes too slow - It's a sweet spot found through experimentation!

What Does the FFN Learn?

While attention learns **relationships between positions**, FFN learns **transformations at each position**:

```
Attention: "sat" learns it's related to "cat"
FFN: Transforms "sat related to cat" → "past-tense verb action by feline"
```

FFN can learn patterns like: - "Plural nouns need plural verbs" - "Past tense patterns" - "Semantic categories" (animals, food, actions) - "Syntactic structures"

Position-wise Processing

Important: FFN processes each position independently:

```python
# These are equivalent:
output = ffn(x)  # Process whole sequence

# Same as:
output = torch.stack([ffn(x[:, i, :]) for i in range(T)], dim=1)
# Process each position separately!
```

Why independent? - After attention mixed information across positions - Now we refine each position's representation - Attention = communication, FFN = computation

In Our Code

```python
class FeedForward(nn.Module):
    def __init__(self, n_embd, dropout=0.1):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(n_embd, 4 * n_embd),  # Expand: 384 → 1536
            nn.ReLU(),                       # Non-linearity
            nn.Dropout(dropout),             # Regularization
            nn.Linear(4 * n_embd, n_embd),  # Compress: 1536 → 384
            nn.Dropout(dropout),             # More regularization
        )

    def forward(self, x):
        return self.net(x)
```

Step-by-Step Example

```
# Input from attention
x = torch.randn(2, 10, 384)  # (batch=2, seq_len=10, dim=384)

# Step 1: Expand
hidden = linear1(x)  # (2, 10, 1536) - 4× bigger!

# Step 2: ReLU activation
hidden = relu(hidden)  # (2, 10, 1536) - now non-linear

# Step 3: Compress
output = linear2(hidden)  # (2, 10, 384) - back to original size

# Each of the 10 positions processed independently!
```

Why ReLU?

ReLU (Rectified Linear Unit) adds non-linearity:

```
ReLU(x) = max(0, x)

# Examples:
ReLU(-0.5) = 0    # Negative → 0
ReLU(0.3)  = 0.3  # Positive → unchanged
ReLU(-2.1) = 0    # Negative → 0
ReLU(1.7)  = 1.7  # Positive → unchanged
```

Without non-linearity, the network would just be a linear transformation (boring!). ReLU lets it learn complex patterns.

The Role in the Transformer Block

```
Input
  ↓
Attention (look at other positions)
  ↓
+ Add & Norm (residual connection)
  ↓
Feed-Forward (process each position)
  ↓
+ Add & Norm (residual connection)
  ↓
Output
```

Attention = “communicate” (positions talk to each other) FFN = “compute” (each position thinks independently)

Real-World Analogy

Think of a team project: 1. **Attention (Meeting)**: Everyone shares their ideas and listens to others 2. **FFN (Individual Work)**: Each person goes away and processes what they learned

Attention = collaboration, FFN = individual processing

Key Point: Feed-forward networks provide the computational power to transform and refine the information gathered by attention, processing each position independently.

---

9. Transformer Blocks: Putting It Together

The Complete Building Block

A **Transformer Block** combines everything we've learned so far into one powerful unit:

```
Input
  ↓
Layer Norm
  ↓
Multi-Head Attention
  ↓
+ (Residual) ←
  ↓
Layer Norm
  ↓
Feed-Forward
  ↓
+ (Residual) ←
  ↓
Output
```

The Six Key Components

1. Layer Normalization (Pre-Norm) Before each sub-layer, we normalize:

```
# Layer norm: mean=0, std=1 for each position
x_norm = (x - mean) / std

# Example:
x = [1, 2, 3, 4, 5]
x_norm = [-1.4, -0.7, 0, 0.7, 1.4]  # mean=0, std=1
```

Why? - Stabilizes training (prevents exploding/vanishing gradients) - Each layer gets consistent input ranges - Makes training faster and more reliable

Pre-Norm vs Post-Norm: We do it **before** the sub-layer (pre-norm), which works better for deep networks.

2. Multi-Head Attention We covered this! Each position looks at all previous positions.

3. First Residual Connection

`x = x + attention(layer_norm(x))`

The input "skips around" attention and gets added back!

4. Layer Normalization (Again)

Normalize before the feed-forward layer.

5. Feed-Forward Network

We covered this! Transform each position independently.

6. Second Residual Connection

`x = x + feed_forward(layer_norm(x))`

Again, the input skips around and gets added back!

Why This Order?

Pre-Norm Architecture (what we use):

```
x = x + attention(layer_norm(x))
x = x + feed_forward(layer_norm(x))
```

Advantages: - Easier to train deep networks - More stable gradients - Better performance with many layers (6, 12, 24+ blocks)

In Our Code

```python
class Block(nn.Module):
    def __init__(self, embedding_dim, block_size, n_heads, dropout=0.1):
        super().__init__()
        # Components
        self.sa = MultiHeadAttention(embedding_dim, block_size, n_heads, dropout)
        self.ffwd = FeedForward(embedding_dim, dropout)
        self.ln1 = nn.LayerNorm(embedding_dim)  # Before attention
        self.ln2 = nn.LayerNorm(embedding_dim)  # Before feed-forward

    def forward(self, x):
        # Attention block with residual
        x = x + self.sa(self.ln1(x))
        # Feed-forward block with residual
        x = x + self.ffwd(self.ln2(x))
        return x
```

Data Flow Example

```python
# Start with embeddings
x = embeddings  # (2, 10, 384)

# ===== Block 1 =====
# Step 1: Normalize
```

```
x_norm = layer_norm1(x)  # (2, 10, 384)

# Step 2: Attention
attn_out = multi_head_attention(x_norm)  # (2, 10, 384)

# Step 3: Add (residual)
x = x + attn_out  # Original x + attention output

# Step 4: Normalize
x_norm = layer_norm2(x)  # (2, 10, 384)

# Step 5: Feed-forward
ff_out = feed_forward(x_norm)  # (2, 10, 384)

# Step 6: Add (residual)
x = x + ff_out  # Previous x + feed-forward output

# After block 1: x now contains:
# - Original embeddings (via residuals)
# - Attention patterns learned
# - Feed-forward transformations
```

Stacking Blocks

The real power comes from stacking multiple blocks:

```
Embeddings
    ↓
Block 1 (learns basic patterns)
    ↓
Block 2 (learns more complex patterns)
    ↓
Block 3 (learns even more abstract patterns)
    ↓
Block 4 (learns high-level concepts)
    ↓
Block 5 (...)
    ↓
Block 6 (...)
    ↓
Output
```

Each block can focus on different levels of abstraction: - Early blocks: syntax, simple patterns - Middle blocks: semantic relationships - Later blocks: high-level reasoning

Why Multiple Blocks Work

The Residual Stream Concept (revisited):

Think of the data flowing through as a stream, with each block adding layers of understanding:

```
Start: "cat sat mat"
  ↓
Block 1: "cat sat mat" + [syntax: subject-verb-object]
  ↓
Block 2: "cat sat mat" + [syntax] + [semantics: animal, action, object]
  ↓
Block 3: "cat sat mat" + [syntax] + [semantics] + [context: past tense scene]
  ↓
Block 4: "cat sat mat" + [syntax] + [semantics] + [context] + [inference: casual scene]
  ↓
...
```

Each block adds a "layer" of understanding while preserving all previous information!

Real-World Analogy

Think of reading comprehension: - **Block 1**: Identify words and grammar - **Block 2**: Understand sentence structure - **Block 3**: Grasp paragraph meaning - **Block 4**: Recognize themes and implications - **Block 5**: Make inferences and predictions - **Block 6**: Full contextual understanding

Each level builds on the previous ones!

Key Point: Transformer blocks are the core building blocks that combine attention (communication) and feed-forward (computation) with residual connections (preservation) to build increasingly sophisticated representations.

---

10. The Complete GPT Architecture

The Full Picture

Now let's put **everything** together into the complete GPT-2 architecture:

```
Text Input: "The cat sat"
         ↓
    Tokenization
         ↓
Token IDs: [145, 892, 33]
         ↓
    Token Embeddings (384-dim vectors)
         ↓
    + Position Embeddings (add position info)
         ↓
    Dropout (regularization)
         ↓

   Transformer Block 1
   - Layer Norm
   - Multi-Head Attention
   - Residual Connection
   - Layer Norm
   - Feed-Forward
   - Residual Connection

         ↓

   Transformer Block 2
   - (same structure)

         ↓
        ...
         ↓

   Transformer Block N
   - (same structure)

         ↓
     Final Layer Norm
         ↓
     Language Model Head (project to vocab)
         ↓
     Logits: [50257 scores per position]
         ↓
     Softmax (convert to probabilities)
         ↓
     Sample (pick next token)
         ↓
Output: Next token prediction
```

Architecture Hyperparameters

Let's use a small GPT as an example:

```
config = {
    'vocab_size': 50257,       # Number of possible tokens
    'embedding_dim': 384,      # Size of embeddings
    'block_size': 256,         # Max sequence length (context window)
    'n_heads': 6,              # Number of attention heads
    'n_layers': 6,             # Number of transformer blocks
    'dropout': 0.2             # Dropout probability
}
```

Parameter Count

Let's calculate how many parameters this model has:

```
# Token embeddings: vocab_size × embedding_dim
token_emb = 50257 × 384 = 19,298,688
```

```
# Position embeddings: block_size × embedding_dim
pos_emb = 256 × 384 = 98,304

# Each transformer block has:
#   Multi-Head Attention:
#     - Q, K, V projections: 3 × (384 × 64) × 6 heads = 442,368
#     - Output projection: 384 × 384 = 147,456
#   Feed-Forward:
#     - Expand: 384 × 1536 = 589,824
#     - Compress: 1536 × 384 = 589,824
#   Layer Norms: ~1,536
#   Total per block: ~1,771,008

# 6 blocks: 6 × 1,771,008 = 10,626,048

# Final layer norm: 768
# LM head: 384 × 50257 = 19,298,688

# TOTAL: ~49 million parameters!
```

The Three Phases

Phase 1: Encoding (Embeddings)

```
def _embed_tokens(self, idx):
    T = idx.size(1)
    # Convert tokens to vectors
    tok_emb = self.token_embedding(idx)
    # Add position information
    pos_emb = self.position_embedding(torch.arange(T))
    # Combine and regularize
    return self.dropout(tok_emb + pos_emb)
```

What happens: - Token IDs → 384-dim vectors - Each vector knows what it is (content) and where it is (position)

Phase 2: Transformation (Blocks)

```
def _apply_blocks(self, x):
    for block in self.blocks:
        x = block(x)  # Apply attention + feed-forward
    return x
```

What happens: - 6 transformer blocks refine the representations - Each block adds understanding through attention and computation - Information flows through the residual stream

Phase 3: Prediction (Output)

```
# Normalize
x = self.ln_f(x)
```

```python
# Project to vocabulary
logits = self.head(x)  # (B, T, vocab_size)

# For each position, we now have 50,257 scores
# Higher score = more likely to be the next token
```

The Complete Forward Pass

```python
class GPT2(nn.Module):
    def forward(self, idx, targets=None):
        # Phase 1: Embed tokens with positions
        x = self._embed_tokens(idx)  # (B, T, 384)

        # Phase 2: Apply transformer blocks
        x = self._apply_blocks(x)  # (B, T, 384)

        # Phase 3: Predict next tokens
        x = self.ln_f(x)  # Final normalization
        logits = self.head(x)  # (B, T, 50257)

        # If training, compute loss
        loss = None
        if targets is not None:
            loss = self._compute_loss(logits, targets)

        return logits, loss
```

Example Flow with Real Numbers

```python
# Input: "The cat sat" → [145, 892, 33]
idx = torch.tensor([[145, 892, 33]])  # (1, 3)

# Step 1: Token Embeddings
tok_emb = token_embedding(idx)  # (1, 3, 384)
# [[145] → [0.23, -0.41, 0.56, ...],  (384 numbers)
#  [892] → [0.12, 0.87, -0.34, ...],  (384 numbers)
#  [33]  → [-0.56, 0.23, 0.91, ...]]  (384 numbers)

# Step 2: Add Positions
pos_emb = position_embedding([0, 1, 2])  # (3, 384)
x = tok_emb + pos_emb  # (1, 3, 384)

# Step 3: Transform through 6 blocks
x = block1(x)  # (1, 3, 384)
x = block2(x)  # (1, 3, 384)
x = block3(x)  # (1, 3, 384)
x = block4(x)  # (1, 3, 384)
```

```
x = block5(x)  # (1, 3, 384)
x = block6(x)  # (1, 3, 384)

# Step 4: Final prediction
x = layer_norm(x)  # (1, 3, 384)
logits = lm_head(x)  # (1, 3, 50257)

# For each of the 3 positions, we have 50,257 scores!
# Position 0 ("The"): predicts next token after "The"
# Position 1 ("cat"): predicts next token after "The cat"
# Position 2 ("sat"): predicts next token after "The cat sat"

# Highest scoring token at position 2 might be "on" or "down"
```

Why This Architecture Works

1. **Embeddings**: Transform discrete tokens into continuous vectors that can be processed
2. **Position Info**: Let the model understand sequence order
3. **Attention**: Let each position gather context from previous positions
4. **Feed-Forward**: Process and transform the contextualized information
5. **Residual Connections**: Preserve information flow through many layers
6. **Layer Norms**: Stabilize training
7. **Stacking**: Build increasingly abstract representations
8. **Final Projection**: Convert representations back to vocabulary space

Key Point: The complete GPT architecture is a carefully designed pipeline that transforms text into embeddings, refines those embeddings through multiple transformer blocks, and produces probability distributions over the next token.

---

11. Training: Teaching the Model

What is Training?

Training is the process of adjusting the model's ~49 million parameters so it gets better at predicting the next token.

The Training Loop

```
for step in range(max_steps):
    # 1. Get a batch of data
    x, y = get_batch(train_data, block_size, batch_size)

    # 2. Forward pass: make predictions
    logits, loss = model(x, y)

    # 3. Backward pass: compute gradients
    loss.backward()
    # 4. Update parameters
    optimizer.step()
    optimizer.zero_grad()
```

Let's break down each step!

Step 1: Get Training Data

```
def get_batch(data, block_size, batch_size):
    # Pick random starting positions
    ix = torch.randint(len(data) - block_size, (batch_size,))

    # Extract sequences
    x = torch.stack([data[i:i+block_size] for i in ix])
    y = torch.stack([data[i+1:i+block_size+1] for i in ix])

    return x, y
```

Example:

```
Data: "The cat sat on the mat and slept"
Block size: 4

Sample 1:
x: "The cat sat on"  → Predict each next token
y: "cat sat on the" → Target for each position

Sample 2:
x: "on the mat and"
y: "the mat and slept"

The model learns: given x, predict y!
```

Step 2: Forward Pass & Loss Computation

```
logits, loss = model(x, targets=y)
```

What happens:

```
# x: "The cat sat on" → [145, 892, 33, 670]
# y: "cat sat on the" → [892, 33, 670, 145]

# Model predicts:
logits = model(x)  # (batch, 4, 50257)

# For position 0: model predicts what comes after "The"
#   Target: "cat" (token 892)
#   Model's scores: [0.01, 0.02, ..., 0.95, ..., 0.01]
#                                      ↑
#                                    token 892
#   Loss: How wrong was this prediction?
# Cross-entropy loss:
loss = -log(probability_of_correct_token)

# If model was very confident (prob=0.95): loss = -log(0.95) = 0.05 (good!)
# If model was unsure (prob=0.1): loss = -log(0.1) = 2.3 (bad!)
```

Cross-Entropy Loss measures: "How surprised are you that the correct answer is X?" - Low surprise (high probability) = low loss = good! - High surprise (low probability) = high loss = bad!

Step 3: Backward Pass (Backpropagation)

```
loss.backward()
```

This is where the magic happens! PyTorch automatically computes **gradients**:

What's a gradient? - Tells each parameter: "If you increase, does loss go up or down?" - Tells us: "How should we adjust this parameter to reduce loss?"

```
# For each parameter (49 million of them!):
# gradient =  loss/ parameter

# Example:
weight = 0.5  (current value)
gradient = 0.3  (tells us: increasing weight increases loss)

# So we should DECREASE this weight!
```

Step 4: Parameter Update (Optimization)

```
optimizer.step()  # Update parameters using gradients
optimizer.zero_grad()  # Reset gradients for next iteration
```

Adam Optimizer (most common):

```
# For each parameter:
new_value = old_value - learning_rate * gradient

# Example:
old_weight = 0.5
gradient = 0.3  (loss increases with weight)
learning_rate = 0.001

new_weight = 0.5 - 0.001 * 0.3 = 0.4997

# Weight decreased slightly, which should reduce loss!
```

The Learning Rate

Critical hyperparameter!

```
# Too large (lr = 0.1):
weight = 0.5 - 0.1 * 0.3 = 0.47  # Big jumps, might overshoot!

# Too small (lr = 0.0001):
weight = 0.5 - 0.0001 * 0.3 = 0.49997  # Tiny steps, very slow!

# Just right (lr = 0.001):
weight = 0.5 - 0.001 * 0.3 = 0.4997  # Steady progress!
```

Gradient Clipping

```
torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)
```

Why? Sometimes gradients get HUGE (exploding gradients):

```
# Without clipping:
gradient = 10000  (huge!)
update = -0.001 * 10000 = -10  (massive change, breaks training!)

# With clipping (max=1.0):
if gradient > 1.0:
    gradient = 1.0
update = -0.001 * 1.0 = -0.001  (reasonable change)
```

The Complete Training Step

```
def _train_step(model, optimizer, train_data, block_size, batch_size, device):
    # 1. Sample batch
    xb, yb = get_batch(train_data, block_size, batch_size, device)

    # 2. Forward pass
    logits, loss = model(xb, yb)

    # 3. Zero gradients from previous step
    optimizer.zero_grad()

    # 4. Backward pass
    loss.backward()

    # 5. Clip gradients (prevent explosion)
    torch.nn.utils.clip_grad_norm_(model.parameters(), 1.0)

    # 6. Update parameters
    optimizer.step()

    # 7. Return loss for monitoring
    return loss.item()
```

Evaluation

We also evaluate on a **validation set** (data the model hasn't seen during training):

```python
def _eval_model(model, val_data, block_size, batch_size, device, eval_iters):
    model.eval()  # Turn off dropout
    losses = []

    with torch.no_grad():  # Don't compute gradients
        for _ in range(eval_iters):
            xv, yv = get_batch(val_data, block_size, batch_size, device)
            _, loss = model(xv, yv)
            losses.append(loss.item())

    model.train()  # Turn dropout back on
    return sum(losses) / len(losses)
```

Why evaluate? - Check if model is actually learning (validation loss should decrease) - Detect overfitting (validation loss increases while training loss decreases)

Training Progress

```
Step 0    | Train Loss: 10.83 | Val Loss: 10.85  (random initialization)
Step 100  | Train Loss:  6.45 | Val Loss:  6.52  (learning basic patterns)
Step 500  | Train Loss:  4.23 | Val Loss:  4.31  (learning syntax)
Step 1000 | Train Loss:  3.15 | Val Loss:  3.28  (learning semantics)
Step 5000 | Train Loss:  2.31 | Val Loss:  2.45  (generating coherent text)
Step 10000| Train Loss:  1.89 | Val Loss:  2.12  (good performance!)
```

Checkpointing

Save the best model:

```python
def save_checkpoint(model, optimizer, step, val_loss, filepath):
    torch.save({
        'step': step,
        'model_state_dict': model.state_dict(),
        'optimizer_state_dict': optimizer.state_dict(),
        'val_loss': val_loss,
    }, filepath)
```

Why? - Training might crash (save progress!) - Want to use the best model (not the last) - Can resume training later

The Big Picture

```
Start: Random parameters (model outputs gibberish)
   ↓
Train for 100 steps (model learns common words)
   ↓
Train for 1000 steps (model learns grammar)
```

```
   ↓
Train for 10000 steps (model writes coherent sentences)
   ↓
End: Trained model (can generate human-like text!)
```

Key Point: Training adjusts millions of parameters through gradient descent, gradually teaching the model to predict the next token by showing it countless examples and adjusting parameters to minimize prediction errors.

---

12. Generation: Creating New Text

From Training to Generation

Training: Learn to predict next tokens from examples **Generation**: Use those learned patterns to create new text!

The Generation Process

```python
def generate(model, start_tokens, max_new_tokens, temperature=1.0):
    """
    Start with some tokens, generate new ones!

    Args:
        start_tokens: e.g., "The cat" → [145, 892]
        max_new_tokens: How many new tokens to generate
        temperature: Controls randomness
    """
    current_tokens = start_tokens

    for _ in range(max_new_tokens):
        # 1. Get model predictions
        logits, _ = model(current_tokens)

        # 2. Focus on last position (next token)
        next_token_logits = logits[:, -1, :]

        # 3. Apply temperature
        next_token_logits = next_token_logits / temperature

        # 4. Convert to probabilities
        probs = F.softmax(next_token_logits, dim=-1)

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

        # 6. Append to sequence
        current_tokens = torch.cat([current_tokens, next_token], dim=1)
    return current_tokens
```

Let's break this down!

Step-by-Step Generation Example

Initial Context

```
Input: "The cat sat on the"
Tokens: [145, 892, 33, 670, 145]
```

Iteration 1: Generate Token 6

```
# 1. Forward pass
logits, _ = model([145, 892, 33, 670, 145])
# logits shape: (1, 5, 50257)
#                ↑  ↑  ↑
#             batch, seq_len, vocab_size

# 2. Get predictions for position after "the"
next_logits = logits[:, -1, :]  # (1, 50257)
# These are raw scores for each possible next token

# Example logits (just first 10 tokens):
# token 0 ("!"):   -2.3
# token 1 ("."):   -1.5
# token 2 ("the"): -3.1
# token 3 ("mat"): 2.8  ← High score!
# token 4 ("dog"): 1.2
# ...

# 3. Apply temperature (we'll explain this next!)
next_logits = next_logits / 1.0  # temperature = 1.0

# 4. Convert to probabilities with softmax
probs = softmax(next_logits)
# token 3 ("mat"): 0.45  ← 45% probability
# token 4 ("dog"): 0.12  ← 12% probability
# token 8 ("floor"): 0.08 ← 8% probability
# ... all sum to 1.0

# 5. Sample from probability distribution
next_token = sample(probs)  # Let's say we get token 3 ("mat")

# 6. Append to sequence
tokens = [145, 892, 33, 670, 145, 3]
text = "The cat sat on the mat"
```

Iteration 2: Generate Token 7

```
# Now we have: "The cat sat on the mat"
# Model predicts: What comes after "mat"?

logits, _ = model([145, 892, 33, 670, 145, 3])
next_logits = logits[:, -1, :]  # Position after "mat"

# Possible predictions:
# "." : 0.35  (35% - end the sentence)
# "and": 0.25 (25% - continue the sentence)
# ",": 0.18   (18% - add a clause)

# Sample: Let's say we get "."
tokens = [145, 892, 33, 670, 145, 3, 13]
text = "The cat sat on the mat."
```

Temperature: Controlling Randomness

Temperature is crucial! It controls how "creative" vs "deterministic" the model is.

Low Temperature (temperature < 1.0) - Focused, Deterministic

```
# Original logits:
logits = [1.0, 2.0, 3.0, 1.5]

# With temperature = 0.5 (divide by 0.5 = multiply by 2)
scaled = [2.0, 4.0, 6.0, 3.0]  # Differences amplified!

# After softmax:
probs = [0.01, 0.12, 0.86, 0.01]
#                    ↑
#        Model is very confident about this choice!
```

Effect: Model picks the most likely token almost every time - **Pros**: More coherent, grammatically correct - **Cons**: Boring, repetitive ("the the the…")

High Temperature (temperature > 1.0) - Creative, Random

```
# Original logits:
logits = [1.0, 2.0, 3.0, 1.5]

# With temperature = 2.0
scaled = [0.5, 1.0, 1.5, 0.75]  # Differences reduced!

# After softmax:
probs = [0.19, 0.29, 0.38, 0.14]
#        More evenly distributed!
```

Effect: Model considers less likely tokens - **Pros**: More creative, varied, interesting - **Cons**: Can be incoherent or make mistakes

Temperature = 1.0 - Balanced

```
# Original logits (unchanged)
probs = softmax(logits)  # Normal sampling
```

Effect: Sample exactly as the model learned - Good default choice!

Special Case: Temperature = 0 (Greedy Decoding)

```
# Don't sample - just pick the highest probability token
next_token = argmax(probs)
```

Effect: Always pick the most likely token - Most deterministic - Useful for reproducible output

Examples with Different Temperatures

Prompt: "Once upon a time, there was a"

Temperature = 0.5 (Conservative):

```
"Once upon a time, there was a little girl who lived in a
small village with her mother and father."
```

(Predictable, grammatically perfect)

Temperature = 1.0 (Balanced):

```
"Once upon a time, there was a young wizard who discovered
a mysterious book in his grandfather's attic."
```

(Coherent but more interesting)

Temperature = 1.5 (Creative):

```
"Once upon a time, there was a talking fish who dreamed
of becoming a ballet dancer on the moon."
```

(Creative but might get weird!)

Temperature = 2.0 (Very Random):

```
"Once upon a time, there was a purple mathematics equation
that sang opera to the refrigerator's grandmother."
```

(Too random, incoherent)

Context Window Management

GPT has a **limited context window** (e.g., 256 tokens). What if our generated text exceeds this?

```
def _generate_step(self, idx, temperature):
    # Crop to last block_size tokens
    idx_cond = idx[:, -self.block_size:]
    # Forward pass with cropped context
    logits, _ = self(idx_cond)

    # Rest of generation...
```

Example:

```
Context window: 256 tokens
Generated so far: 300 tokens

Use only: tokens[300-256:300] = last 256 tokens
Model doesn't see: the first 44 tokens

This is why GPT can "forget" earlier context in long conversations!
```

The Complete Generation Function

```
def generate(self, idx, max_new_tokens, temperature=1.0):
    """Generate new tokens autoregressively."""
    for _ in range(max_new_tokens):
        # Crop context if needed
        idx_cond = idx[:, -self.block_size:]

        # Get predictions
        logits, _ = self(idx_cond)

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

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

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

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

    return idx
```

Beam Search (Alternative to Sampling)

Instead of sampling one token at a time, we can keep track of multiple possibilities:

```
Start: "The cat"

Beam 1: "The cat sat" (prob=0.8)
Beam 2: "The cat jumped" (prob=0.6)
Beam 3: "The cat ran" (prob=0.5)
```

```
Continue each beam...
Pick the sequence with highest total probability!
```

Not implemented in GPT-2, but used in some applications.

Key Point: Generation is the reverse of training - instead of predicting the next token from known text, we use the model's learned predictions to create new text autoregressively, with temperature controlling the balance between coherence and creativity.

---

13. Putting It All Together

The Complete Pipeline

Let's trace a single example through the **entire system**:

Input: "The cat sat on the mat"

Phase 1: Tokenization

```
text = "The cat sat on the mat"
tokens = tokenizer.encode(text)
# [145, 892, 33, 670, 145, 3]
```

Each word/piece gets a unique ID.

Phase 2: Embedding

```
# Token embeddings (what are these tokens?)
tok_emb = token_embedding([145, 892, 33, 670, 145, 3])
# Shape: (6, 384)
# [[0.23, -0.41, 0.56, ...],    ← "The"
#  [0.12, 0.87, -0.34, ...],    ← "cat"
#  [-0.56, 0.23, 0.91, ...],    ← "sat"
#  [0.34, -0.12, 0.45, ...],    ← "on"
#  [0.23, -0.41, 0.56, ...],    ← "the" (same as position 0!)
#  [0.78, 0.34, -0.23, ...]]    ← "mat"

# Position embeddings (where are these tokens?)
pos_emb = position_embedding([0, 1, 2, 3, 4, 5])
# Shape: (6, 384)
# [[0.11, 0.22, -0.33, ...],    ← position 0
#  [0.44, -0.55, 0.66, ...],    ← position 1
#  [0.77, 0.88, -0.99, ...],    ← position 2
#  [0.12, -0.23, 0.34, ...],    ← position 3
#  [0.45, 0.56, -0.67, ...],    ← position 4
#  [0.78, -0.89, 0.90, ...]]    ← position 5

# Combined embeddings
x = tok_emb + pos_emb  # (6, 384)
# Each position now knows WHAT it is and WHERE it is!
```

Phase 3: Transformer Block 1

```
# Input: (6, 384)

# Sub-layer 1: Multi-Head Attention
x_norm = layer_norm(x)

# Each position looks at previous positions:
# "The" (pos 0): Only sees itself
# "cat" (pos 1): Sees "The" and "cat"
# "sat" (pos 2): Sees "The", "cat", "sat"
# "on" (pos 3): Sees "The", "cat", "sat", "on"
# "the" (pos 4): Sees all previous
# "mat" (pos 5): Sees all previous

attn_out = multi_head_attention(x_norm)  # (6, 384)
x = x + attn_out  # Residual connection

# Sub-layer 2: Feed-Forward
x_norm = layer_norm(x)
ff_out = feed_forward(x_norm)  # (6, 384)
x = x + ff_out  # Residual connection

# After Block 1: (6, 384)
# Each position now has context-aware representations!
```

Phase 4: Transformer Blocks 2-6

```
# Same process 5 more times...
x = block2(x)  # (6, 384)
x = block3(x)  # (6, 384)
x = block4(x)  # (6, 384)
x = block5(x)  # (6, 384)
x = block6(x)  # (6, 384)

# After 6 blocks:
# "mat" has gathered information from:
# - Direct context: "on the"
# - Subject: "cat"
# - Action: "sat"
# - Full sentence structure and meaning
```

Phase 5: Output Projection

```
# Final layer norm
x = layer_norm(x)  # (6, 384)

# Project to vocabulary
logits = lm_head(x)  # (6, 50257)
```

```
# For each position, we have 50,257 scores (one per token!)
```

Phase 6: Predictions

```
# Position 0: "The" predicts next token
#   Highest scores: "cat" (892), "dog" (1234), "man" (5678)
#   Model predicts: "cat"  Correct!

# Position 1: "The cat" predicts next token
#   Highest scores: "sat" (33), "ran" (789), "jumped" (456)
#   Model predicts: "sat"  Correct!

# Position 2: "The cat sat" predicts next token
#   Highest scores: "on" (670), "down" (234), "still" (567)
#   Model predicts: "on"  Correct!

# Position 3: "The cat sat on" predicts next token
#   Highest scores: "the" (145), "a" (678), "my" (901)
#   Model predicts: "the"  Correct!

# Position 4: "The cat sat on the" predicts next token
#   Highest scores: "mat" (3), "floor" (890), "couch" (567)
#   Model predicts: "mat"  Correct!

# Position 5: "The cat sat on the mat" predicts next token
#   Highest scores: "." (13), "and" (234), "," (456)
#   Model predicts: "." (end of sentence!)
```

Understanding What Each Component Does

Token + Position Embeddings

- **Input**: Discrete token IDs
- **Output**: Dense vectors with content and position info
- **Purpose**: Create a rich, continuous representation

Multi-Head Attention

- **Input**: Contextualized vectors from previous layer
- **Output**: Vectors enriched with information from other positions
- **Purpose**: "Reading comprehension" - understanding relationships

Feed-Forward Networks

- **Input**: Attention-enriched vectors
- **Output**: Transformed vectors with extracted features
- **Purpose**: "Thinking" - processing information at each position

Residual Connections

- **Input/Output**: Same shape
- **Purpose**: "Information highway" - preserve original information

Layer Normalization

- **Input**: Any vector
- **Output**: Normalized vector (mean=0, std=1)
- **Purpose**: Stabilize training, prevent exploding/vanishing values

Final Projection

- **Input**: Final transformer representations (384-dim)
- **Output**: Vocabulary logits (50257-dim)
- **Purpose**: Convert learned representations to token predictions

Training vs Generation

During Training:

```
# We have both input and targets
x = "The cat sat on the"
y = "cat sat on the mat"  # Shifted by 1

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

# Compute how wrong predictions are
loss.backward()  # Compute gradients
optimizer.step()  # Update parameters

# Model gets slightly better at predicting!
```

During Generation:

```
# We only have input (no targets)
x = "The cat sat on the"

# Forward pass
logits, _ = model(x, targets=None)  # No loss

# Sample next token
next_token = sample(logits[:, -1, :])  # "mat"

# Append and repeat
x = "The cat sat on the mat"
next_token = sample(logits[:, -1, :])  # "."

# Result: "The cat sat on the mat."
```

Why GPT Works So Well

1. **Self-Attention**: Can look at any previous position (long-range dependencies!)
2. **Multi-Head**: Learns multiple types of relationships simultaneously
3. **Depth**: 6+ layers build increasingly abstract representations
4. **Residual Stream**: Information flows smoothly through many layers
5. **Scale**: 49M+ parameters can memorize patterns and facts
6. **Autoregressive**: Simple training objective (predict next token)
7. **Massive Data**: Trained on billions of tokens from the internet

What Makes It "Intelligent"?

GPT doesn't truly "understand" - it recognizes patterns:

```
Training data includes:
"The cat sat on the mat"
"The dog sat on the floor"
"The bird sat on the tree"

GPT learns:
- [animal] + "sat on the" + [location]
- Verbs agree with subjects
- Prepositions connect actions to places
- Sentence structure patterns

When generating:
"The cat sat on the ___"
→ Likely completions: mat, floor, chair, couch
→ Unlikely: sky, galaxy, thought (doesn't fit pattern!)
```

Limitations

1. **Context Window**: Can only see last N tokens (256, 2048, etc.)
2. **No Real Understanding**: Pattern matching, not reasoning
3. **Training Data**: Only knows what it was trained on
4. **No Memory**: Each generation is independent (unless you include conversation history)
5. **Computational Cost**: Requires powerful GPUs for training
6. **Factual Errors**: Can confidently generate wrong information

But It's Still Amazing!

From simple principles: - "Predict the next token" - Self-attention - Feed-forward layers - Residual connections

We get a model that can: - Write coherent essays - Translate languages - Answer questions - Write code - Have conversations

All from learning to predict the next token!

Conclusion

What We've Learned

You now understand every component of GPT:

1. **Tokenization**: Text → Numbers
2. **Token Embeddings**: Numbers → Vectors
3. **Position Embeddings**: Adding position information
4. **Residual Connections**: Information superhighway
5. **Self-Attention**: Looking at context
6. **Multi-Head Attention**: Multiple perspectives
7. **Feed-Forward Networks**: Processing information
8. **Transformer Blocks**: Combining it all
9. **Complete Architecture**: The full pipeline
10. **Training**: Teaching the model
11. **Generation**: Creating new text

The Core Insight

GPT is "just" predicting the next token, but by doing this billions of times on massive amounts of text, it learns: - Grammar and syntax - Facts and knowledge - Reasoning patterns - Writing styles - And much more!

Next Steps

1. **Run the code**: Train your own GPT-2!
2. **Experiment**: Try different hyperparameters
3. **Visualize**: Plot attention weights to see what the model focuses on
4. **Scale up**: Train a bigger model (more layers, larger n_embd)
5. **Fine-tune**: Train on specific data (code, poems, dialogue)

Key Takeaway

Understanding GPT demystifies AI: - It's not magic - It's clever mathematics and engineering - It's accessible (you can build one!) - But it's also incredibly powerful when scaled up

You now know how ChatGPT-style models work under the hood!

Appendix: Quick Reference

Dimensions Cheat Sheet

```
vocab_size = 50257          # Number of tokens
embedding_dim = 384         # Size of vectors
block_size = 256            # Max sequence length
n_heads = 6                 # Number of attention heads
head_size = 64              # embedding_dim // n_heads
n_layers = 6                # Number of transformer blocks
batch_size = 4              # Number of sequences processed together
```

```
# Shapes through the network:
Input tokens: (batch_size, seq_len)
             → (4, 256)

After embeddings: (batch_size, seq_len, embedding_dim)
                 → (4, 256, 384)

After attention head: (batch_size, seq_len, head_size)
                     → (4, 256, 64)

After multi-head attention: (batch_size, seq_len, embedding_dim)
                           → (4, 256, 384)

After feed-forward: (batch_size, seq_len, embedding_dim)
                   → (4, 256, 384)

After transformer block: (batch_size, seq_len, embedding_dim)
                        → (4, 256, 384)

Final logits: (batch_size, seq_len, vocab_size)
            → (4, 256, 50257)
```

Hyperparameter Guidelines

Parameter	Tiny	Small	Medium	Large
embedding_dim	64	384	768	1024
n_layers	2	6	12	24
n_heads	2	6	12	16
block_size	64	256	1024	2048
Parameters	~1M	~49M	~125M	~350M

Important Equations

Attention:

`Attention(Q, K, V) = softmax(Q @ K^T / √d_k) @ V`

Feed-Forward:

`FFN(x) = ReLU(x @ W1 + b1) @ W2 + b2`

Residual Connection:

`output = input + layer(input)`

Layer Norm:

`LayerNorm(x) =   × (x - mean) / std +`

Cross-Entropy Loss:

```
Loss = -log(P(correct_token))
```

Key Concepts Summary

- **Autoregressive**: Predict one token at a time, based on all previous
- **Causal Masking**: Can't look at future tokens
- **Temperature**: Controls randomness in generation
- **Gradient Descent**: How we train neural networks
- **Backpropagation**: How we compute gradients efficiently
- **Attention**: Mechanism for focusing on relevant information
- **Residual Stream**: Information highway through the network

---

Happy Learning!

For code implementation, see: `micro_gpt.py`
For comprehensive tests, see: `test_micro_gpt.py`
For running examples, see: `example.py`