Hands-On Guide to Building Your Own GPT Model

Jerome Niyirora, Ph.D.

SUNY Polytechnic Institute

Utica, NY, USA

May 23, 2025

Outline

The NLP Problem

NLP Remains a Hard Problem

"Despite recent advances in large language models (LLMs), natural language processing (NLP) remains a challenging problem."
— From my own observation

- LLMs excel at surface fluency but struggle with reasoning, truthfulness, and context grounding.
- Ambiguity, pragmatics, and commonsense remain challenging.
- Language is deeply tied to culture, context, cognition, and world knowledge—things machines don't truly possess.

NLP Remains a Hard Problem

"Despite recent advances in large language models (LLMs), natural language processing (NLP) remains a challenging problem."
— From my own observation

- LLMs excel at surface fluency but struggle with reasoning, truthfulness, and context grounding.
- Ambiguity, pragmatics, and commonsense remain challenging.
- Language is deeply tied to culture, context, cognition, and world knowledge—things machines don't truly possess.

A Pragmatic Perspective

Why Still Study LLMs?

"While LLMs may not truly understand language, their generative power from existing data is remarkable and worth unpacking."

— A pragmatic perspective

- LLMs can generate coherent, creative, and contextually rich text across domains.
- Understanding how they work helps us demystify their strengths—and their limitations.
- Advancing this field requires not just using LLMs, but also comprehending their underlying architecture.
- **Goal of this talk:** To walk through how **transformer**-based LLMs (e.g., GPT) are constructed and why they work.

Why Still Study LLMs?

"While LLMs may not truly understand language, their generative power from existing data is remarkable and worth unpacking."
— A pragmatic perspective

- LLMs can generate coherent, creative, and contextually rich text across domains.
- Understanding how they work helps us demystify their strengths—and their limitations.
- Advancing this field requires not just using LLMs, but also comprehending their underlying architecture.
- **Goal of this talk:** To walk through how **transformer**-based LLMs (e.g., GPT) are constructed and why they work.

Why Still Study LLMs?

"While LLMs may not truly understand language, their generative power from existing data is remarkable and worth unpacking."

— A pragmatic perspective

- LLMs can generate coherent, creative, and contextually rich text across domains.
- Understanding how they work helps us demystify their strengths—and their limitations.
- Advancing this field requires not just using LLMs, but also comprehending their underlying architecture.
- **Goal of this talk:** To walk through how **transformer**-based LLMs (e.g., GPT) are constructed and why they work.

The Transformer Model

Transformer Architecture in Large Language Models I

Attention is All You Need

The **Transformer architecture**, introduced in the paper "Attention is All You Need" by Vaswani et al. (2017), revolutionized NLP by **enabling models to process words in parallel and capture contextual** information from sequences efficiently.

Transformer Architecture in Large Language Models II

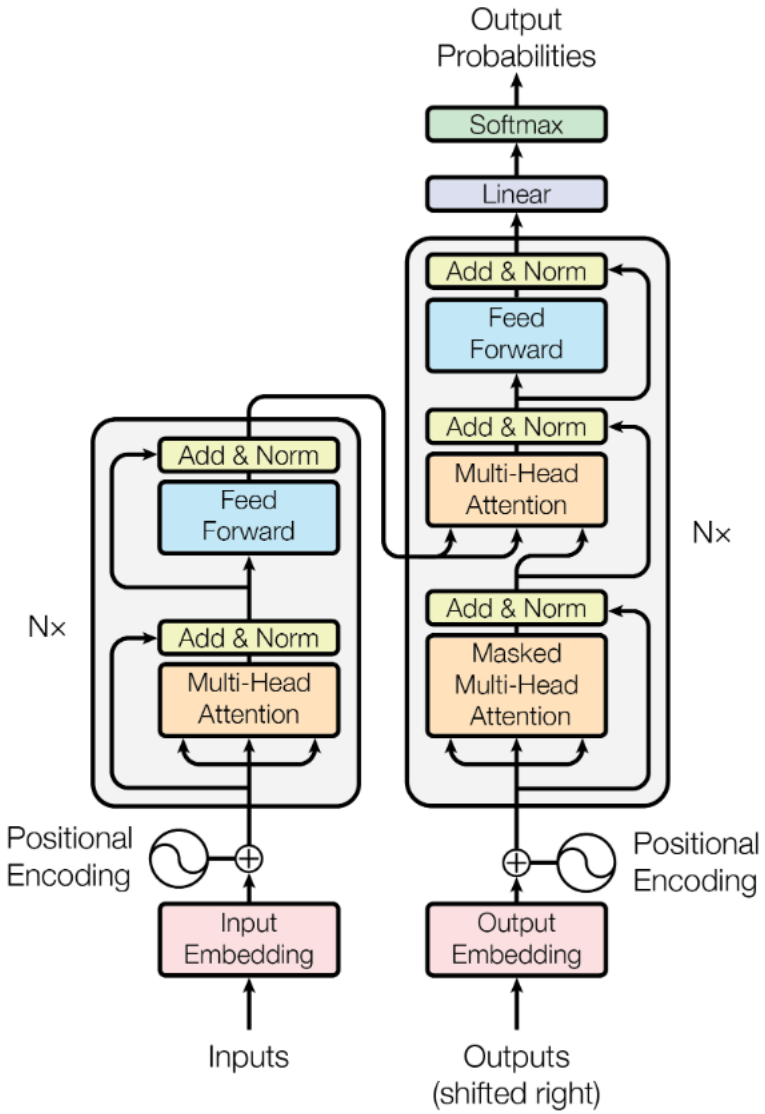


Figure 1: **Source and Implementation Example**: `https://nlp.seas.harvard.edu/2018/04/03/attention.html`

Transformer Architecture in Large Language Models III

Key Components

- **Self-Attention Mechanism**: Allows each word to dynamically focus on other parts of the sentence, capturing intricate dependencies.
- **Positional Encoding**: Injects information about the position of each word in the sequence, compensating for the absence of recurrence.
- **Stacked Encoders and Decoders**: Multiple layers (stacks) of encoders for input processing and decoders for output generation, enabling deep understanding.
- **Feed-Forward Neural Networks**: Each layer contains feed-forward networks for transforming attention-combined inputs into outputs.

Working with Text Data

Text Processing: Preliminaries

- **Tokenization:** Split raw text into words, subwords, or characters.
- **Vocabulary:** Build a unique ID mapping for tokens.
- **Numerical encoding:** Convert tokens into input tensors.
- **Context windowing:** Fixed-length segments (e.g., 512 or 1024 tokens).
- **Padding & batching:** Standardize input sizes across batches.

Key Insight

These steps transform messy text into structured input for a transformer model.

Text Processing: Preliminaries

- **Tokenization:** Split raw text into words, subwords, or characters.
- **Vocabulary:** Build a unique ID mapping for tokens.
- **Numerical encoding:** Convert tokens into input tensors.
- **Context windowing:** Fixed-length segments (e.g., 512 or 1024 tokens).
- **Padding & batching:** Standardize input sizes across batches.

Key Insight

These steps transform messy text into structured input for a transformer model.

Tokenization

Tokenization: Building the Vocabulary

- **What is Tokenization?**
  - The process of splitting raw text into smaller units called tokens.
  - These tokens are used to construct a vocabulary — a mapping of each unique token to an integer ID.
  - The vocabulary is essential for converting input text into a numerical format that neural networks can process.

Example

`Input text:` *"Apple's quarterly revenue rose 12% year-over-year."*

`Tokens:` ["Apple", "'s", "quarterly", "revenue", "rose", "12", "%", "year", "-", "over", "-", "year", "."]

- The set of all unique tokens across the training data becomes the model's vocabulary.

Tokenization: Building the Vocabulary

- **What is Tokenization?**
  - The process of splitting raw text into smaller units called tokens.
  - These tokens are used to construct a vocabulary — a mapping of each unique token to an integer ID.
  - The vocabulary is essential for converting input text into a numerical format that neural networks can process.

Example

`Input text:` *"Apple's quarterly revenue rose 12% year-over-year."*

`Tokens:` ["Apple", "'s", "quarterly", "revenue", "rose", "12", "%", "year", "-", "over", "-", "year", "."]

- The set of all unique tokens across the training data becomes the model's vocabulary.

Tokenization: Turning Text into Tokens

Python Code (Tokenization and Vocabulary)

```python
import re

text = "Apple's quarterly revenue rose 12% year-over-year."

# Tokenization using regex that preserves contractions like 's
tokens = re.findall(r"\w+|[^\w\s]", text)
print("Tokens:", tokens)

# Create sorted vocabulary
vocab = sorted(set(tokens))
print("Vocabulary:", vocab)

# Create token-to-ID dictionary
token2id = {token: idx for idx, token in enumerate(vocab)}
print("Text IDs:", token2id)
```

Converting Tokens to Token IDs

- After tokenization, each unique token is assigned an integer ID based on a vocabulary.
- The input sequence is then represented numerically using these token IDs.

Example: Apple‘s quarterly revenue

Tokens: ['Apple', "'s", 'quarterly', 'revenue', 'rose', '12', '%', 'year', '-', 'over', '-', 'year', '.']

Vocabulary: ['%', "'s", '-', '.', '12', 'Apple', 'over', 'quarterly', 'revenue', 'rose', 'year']

Token IDs: '%': 0, "'s": 1, '-': 2, '.': 3, '12': 4, 'Apple': 5, 'over': 6, 'quarterly': 7, 'revenue': 8, 'rose': 9, 'year': 10

Text IDs: [5, 1, 7, 8, 9, 4, 0, 10, 2, 6, 2, 10, 3]

- Vocabulary is alphabetically sorted before assigning token IDs.

A Comprehensive Vocabulary

Special Tokens and Unknown Words

- Some words in the input may not exist in the model's vocabulary.
- Byte Pair Encoding (BPE) helps break rare or unknown words into subword units.
- These are replaced with a special `<unk>` (unknown) token.
- Other special tokens include:
  - `<pad>` (padding), `<endoftext>` (end of input)
- These tokens are added to the vocabulary with reserved IDs.

Example

Sentence: `"Apple's revenue skyrocketed"`

Known Tokens: `Apple (5), revenue (7)`

Unknown Token: `skyrocketed → <unk> (783)`

Token IDs: `[5, 1, 8, 783]`

A Comprehensive Vocabulary

- Large language models rely on a comprehensive vocabulary to handle a wide range of inputs (GPT2's vocabulary size = 50,257)

Python Code: GPT-2 Tokenizer (BPE)

```
import tiktoken
text = "Apple's quarterly revenue rose 12% year-over-year."
tokenizer = tiktoken.get_encoding("gpt2")
integers = tokenizer.encode(text)

print(integers)
# Output: [16108, 338, 27868, 6426, 8278, 1105, 4, 614, 12,
    2502, 12, 1941, 13]
decoded_ids = tokenizer.decode(integers)
print(decoded_ids) #Apple's quarterly revenue rose 12% year
   -over-year.
```

- Each number corresponds to a BPE subword token in GPT-2's vocabulary.
- BPE helps cover nearly all English words using a manageable vocabulary size.

Embedding Tokens

Embedding Token IDs into Vectors

- Token IDs are mapped to high-dimensional vectors called embeddings.
- These vectors capture semantic and syntactic information about each token.

Example: Apple's quarterly revenue

Tokens: `[Apple, 's, quarterly, revenue]`

Token IDs: `[5, 1, 7, 8]`

Embeddings (first 2 dimensions shown):

```
Apple (5):      [-1.57, 0.97, ...]
's (1):         [0.66, -0.22, ...]
quarterly (7):  [1.05, -0.39, ...]
revenue (8):    [-0.2, -2.27, ...]
```

- These embeddings are input into the GPT model to begin the prediction process.

Embedding Token IDs into Vectors

Python Code: GPT-2 Tokenizer (BPE)

```
import torch

# Simulated vocab size and input
vocab_size = 10  # len(vocab)
input_ids = torch.tensor([5, 1, 7, 8])
output_dim = 5

# Initialize embedding layer
torch.manual_seed(123)
embedding_layer = torch.nn.Embedding(vocab_size, output_dim)

# Get embeddings
embeddings = embedding_layer(input_ids)

# print(embeddings): Print token ID and rounded embedding vector
Token ID    Embedding Vector
----------------------------------------
5        [-1.57, 0.97, -1.15, -1.16, 0.33]
1        [0.66, -0.22, -0.38, 0.77, -1.19]
7        [1.05, -0.39, -0.93, -0.5, 0.14]
8        [-0.2, -2.27, -0.91, -0.42, 1.31]
```

Embedding Dimension vs. Context Size

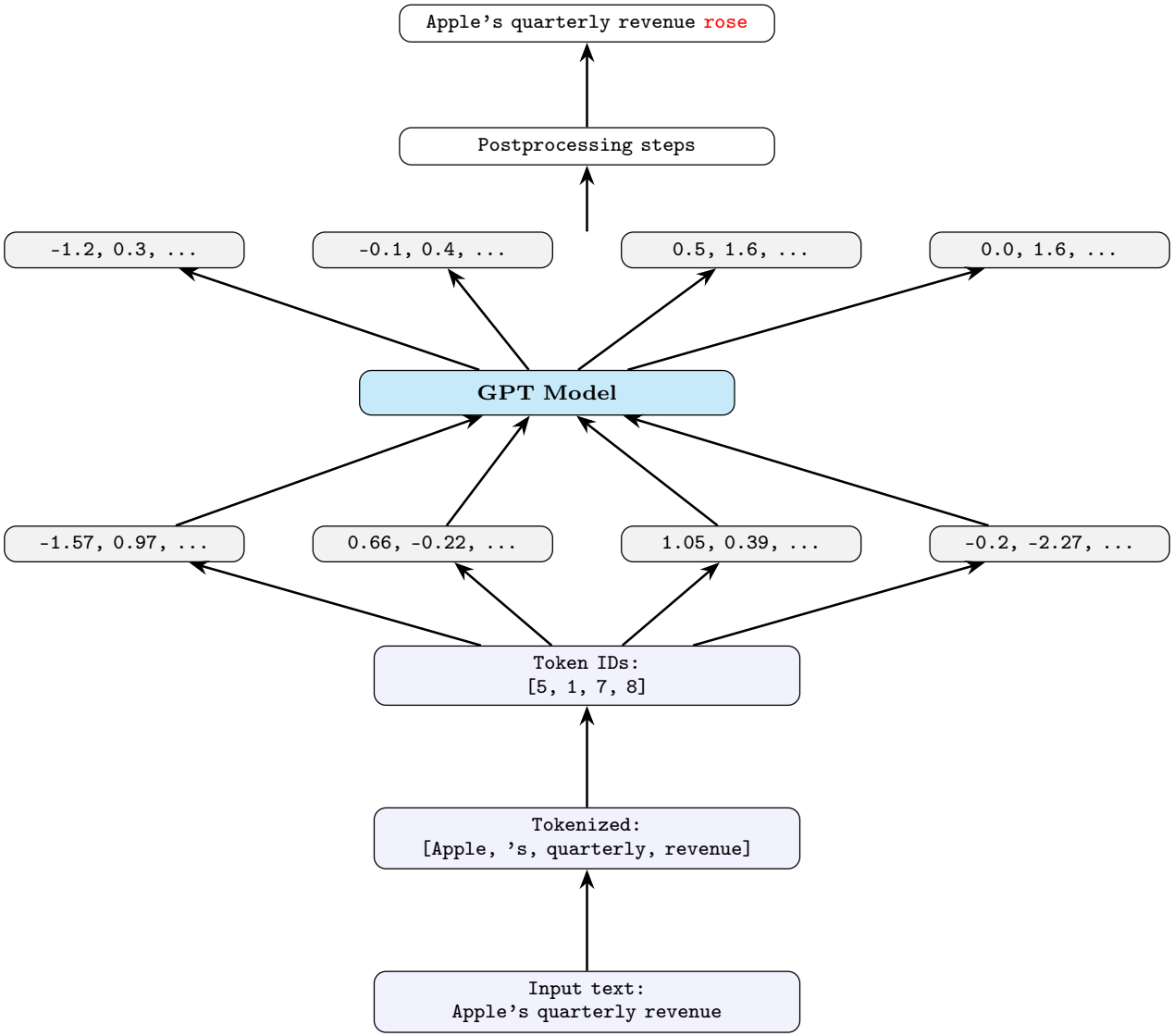
- **Embedding Dimension (output_dim):**
  - The size of the vector representing each token.
  - Defines the dimensionality of the token embeddings and hidden layers.
  - Typical values: 768 (GPT-2 small), 1024 (GPT-2 medium), etc.

- **Context Size (Sequence Length):**
  - The maximum number of tokens the model can process at once.
  - Also known as sequence length, block size, or context window.
  - Example: GPT-2 uses a context size of 1024 tokens.

- **Key Difference:**
  - *Embedding dimension* defines the size of each token vector.
  - *Context size* defines how many tokens the model can handle in a single input.

Model 1

From Tokenization to Prediction



Adapted from Sebastian Raschka's 2024 guide: *Build a Language Model from Scratch*.

Positional Encoding

Positional Encoding

- Transformer models process all tokens in parallel and do not inherently capture word order.
- To inject information about token position, a positional encoding is added to each token embedding.
- This helps the model understand the sequence structure during training and inference.

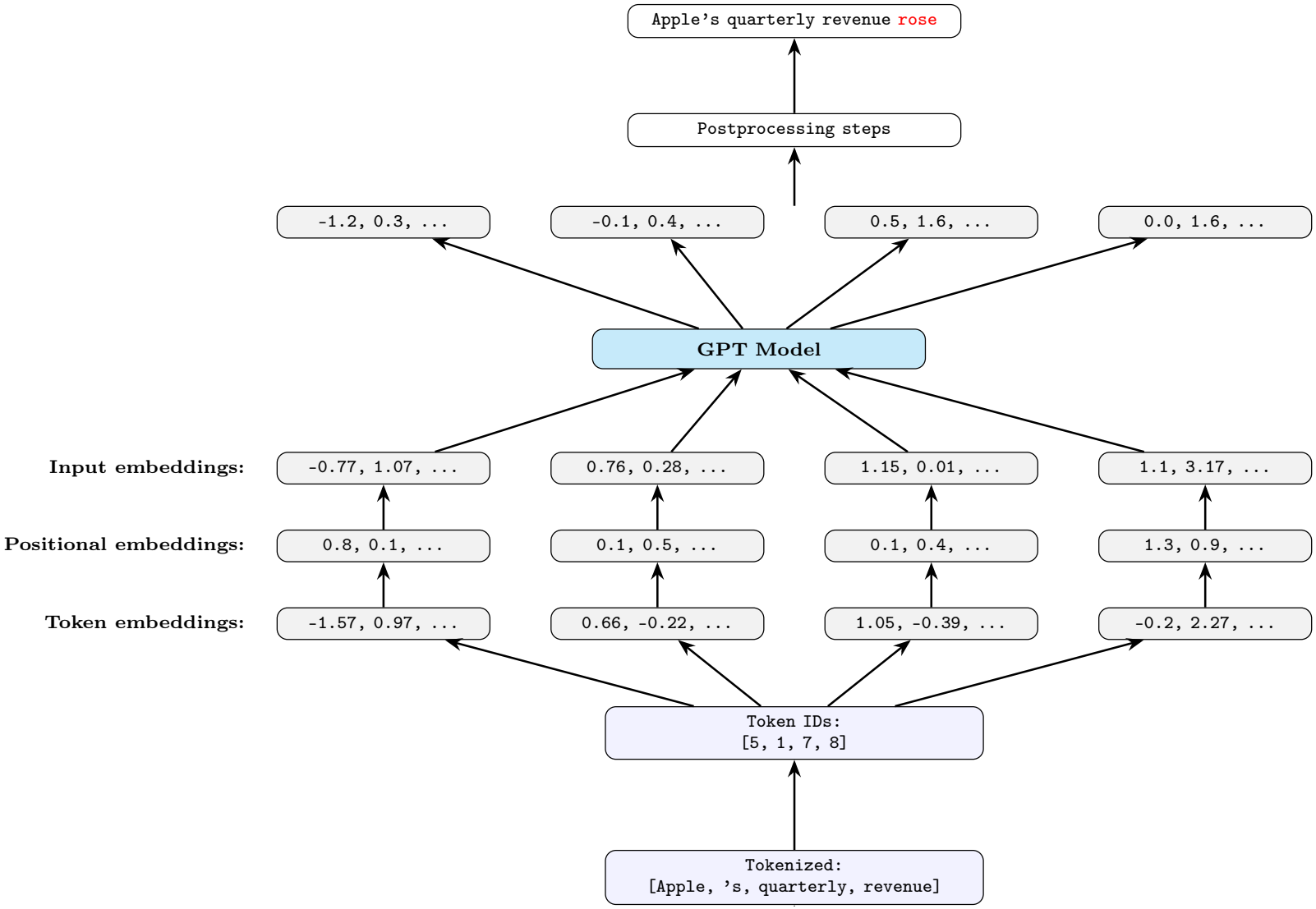
Embedding Formula

```
Input embedding = Token embedding + Positional embedding
```

- Positional encodings can be:
  - *Learned*: optimized during training
  - *Fixed*: based on sine and cosine functions (*Attention is All You Need*, Vaswani et al., 2017))
  - The position index will always have the same positional embedding

Model 2

From Tokenization to Prediction



Adapted from Sebastian Raschka's 2024 guide: *Build a Language Model from Scratch*.

Single-Head Attention Mechanism

Why Do We Need Attention Mechanisms?

- Traditional sequence models (e.g., RNNs) process inputs one token at a time and rely on a fixed-size hidden state to carry context.
- These models struggle with:
  - Capturing long-range dependencies.
  - Remembering and emphasizing relevant information from earlier tokens.
- Attention mechanisms address this by:
  - Allowing the model to weigh and access all input tokens at each step.
  - Dynamically focusing on the most relevant parts of the sequence.
- This capability is essential for tasks like translation, summarization, and question answering.

From RNNs to Transformers: The Rise of Attention

- Attention mechanisms were first introduced in 2014 to improve encoder-decoder RNNs in machine translation.
  - Bahdanau et al. (2014) proposed attention to allow the decoder to focus on relevant parts of the input sequence.
- Later, Vaswani et al. (2017) demonstrated that **attention alone** — without recurrence — was sufficient for sequence modeling.
  - This led to the Transformer architecture, enabling faster training and better long-range dependency modeling.
- This marked a shift from recurrent models to fully attention-based models in NLP.

Computing Attention from Input Embeddings

- In transformer models, attention scores are computed using the dot product between query and key embeddings.
- Embeddings for input tokens are obtained from a learned embedding layer.

Python Code: Attention Weights

```
import torch
attn_scores = input_embeddings @ input_embeddings.T
print(attn_scores)

#Normalize each row to sum up to 1
attn_weights = torch.softmax(attn_scores, dim=-1)
print(attn_weights)
```

- Scores are then normalized (typically with softmax) to produce attention weights with a causal mask.

Computing Attention from Input Embeddings

Python Code: Attention Weights

```
...
# Display matrices
print_matrix(attn_weights, "Attention Weights (Unmasked)")
print_matrix(masked_weights, "Attention Weights (Causal Masked)")


# === Attention Weights (Unmasked) ===
#              |      Apple |         's | quarterly |    revenue |
# ---------------------------------------------------------------
#        Apple |      1.000 |      0.000 |      0.000 |      0.000 |
#           's |      0.005 |      0.592 |      0.135 |      0.268 |
#    quarterly |      0.000 |      0.011 |      0.929 |      0.060 |
#      revenue |      0.000 |      0.011 |      0.032 |      0.957 |

# === Attention Weights (Causal Masked) ===
#              |      Apple |         's | quarterly |    revenue |
# ---------------------------------------------------------------
#        Apple |      1.000 |      0.000 |      0.000 |      0.000 |
#           's |      0.009 |      0.991 |      0.000 |      0.000 |
#    quarterly |      0.000 |      0.011 |      0.989 |      0.000 |
#      revenue |      0.000 |      0.011 |      0.032 |      0.957 |
```

A Small Detour: Transformer's Building Blocks

Defining a Function in Python

Python functions group reusable code into named blocks.

- Defined using the `def` keyword
- May accept inputs ("parameters") and return a result

Example: Add two numbers

```
1 def add(a, b):
2     result = a + b
3     return result
4
5 print(add(3, 5))  # Output: 8
```

Creating a Simple Class in PyTorch

In PyTorch, we use classes to define custom models or layers.

- Inherit from `nn.Module`
- Define the structure in `__init__()`
- Define forward behavior in `forward()`

Example: Tiny Linear Model

```
1  import torch.nn as nn
2
3  class TinyModel(nn.Module):
4      def __init__(self):
5          super().__init__()
6          self.linear = nn.Linear(2, 1)
7          # 2 inputs to 1 output
8
9      def forward(self, x):
10         return self.linear(x)
```

Attention from Input Embeddings

Limitations of Simple Dot Product Attention

- A basic approach to self-attention computes similarity between input embeddings using a simple dot product:

$$\text{score}(x^{(i)}, x^{(j)}) = x^{(i)} \cdot x^{(j)}$$

- **Limitation:** This formulation assumes a single embedding space serves multiple roles:
  - Asking for information (*query*)
  - Representing content (*key*)
  - Delivering content (*value*)
- This restricts the model's ability to distinguish between these roles and to learn task-specific interactions between tokens.

Introducing Learnable Q, K, V Projections

- To address the limitations of raw dot products, self-attention uses three learnable matrices:

$$W_q, \quad W_k, \quad W_v$$

- Each input embedding $x^{(i)}$ is projected into:
  - **Query:** $q^{(i)} = x^{(i)} W_q$
  - **Key:** $k^{(i)} = x^{(i)} W_k$
  - **Value:** $v^{(i)} = x^{(i)} W_v$
- These projections allow the model to:
  - Compare tokens using queries and keys
  - Retrieve information using values
- The attention weights between tokens are computed using:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right) V$$

where d_k is the dimension of the key vectors.

Computing Single-Head Attention

Python Code: A simplified Single-Head Attention Class With Neural Network Layers

```python
#d_in: Dimension of input embeddings
#d_out: Dimension of Q/K/V vectors per head
class SelfAttention_v2(nn.Module):

    def __init__(self, d_in, d_out, qkv_bias=False):
        super().__init__()
        self.W_query = nn.Linear(d_in, d_out, bias=qkv_bias)
        self.W_key   = nn.Linear(d_in, d_out, bias=qkv_bias)
        self.W_value = nn.Linear(d_in, d_out, bias=qkv_bias)

    def forward(self, x):
        keys = self.W_key(x)
        queries = self.W_query(x)
        values = self.W_value(x)

        attn_scores = queries @ keys.T
        attn_weights = torch.softmax(attn_scores / keys.shape[-1]**0.5, dim
            =-1)

        context_vec = attn_weights @ values
        return context_vec

torch.manual_seed(789)
sa_v2 = SelfAttention_v2(d_in, d_out)
print(sa_v2(inputs))
```

Extending to Multi-Head Attention

- **Motivation:** A single attention head may not capture all relationships between tokens.
- **Idea:** Run attention multiple times in parallel, each with different learned projections:

$$\text{head}_i = \text{Attention}(XW_q^{(i)}, XW_k^{(i)}, XW_v^{(i)})$$

- The heads are then concatenated and projected back into the model's embedding space:

$$\text{MultiHead}(X) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h)W_o$$

- W_o: a learnable output projection matrix.

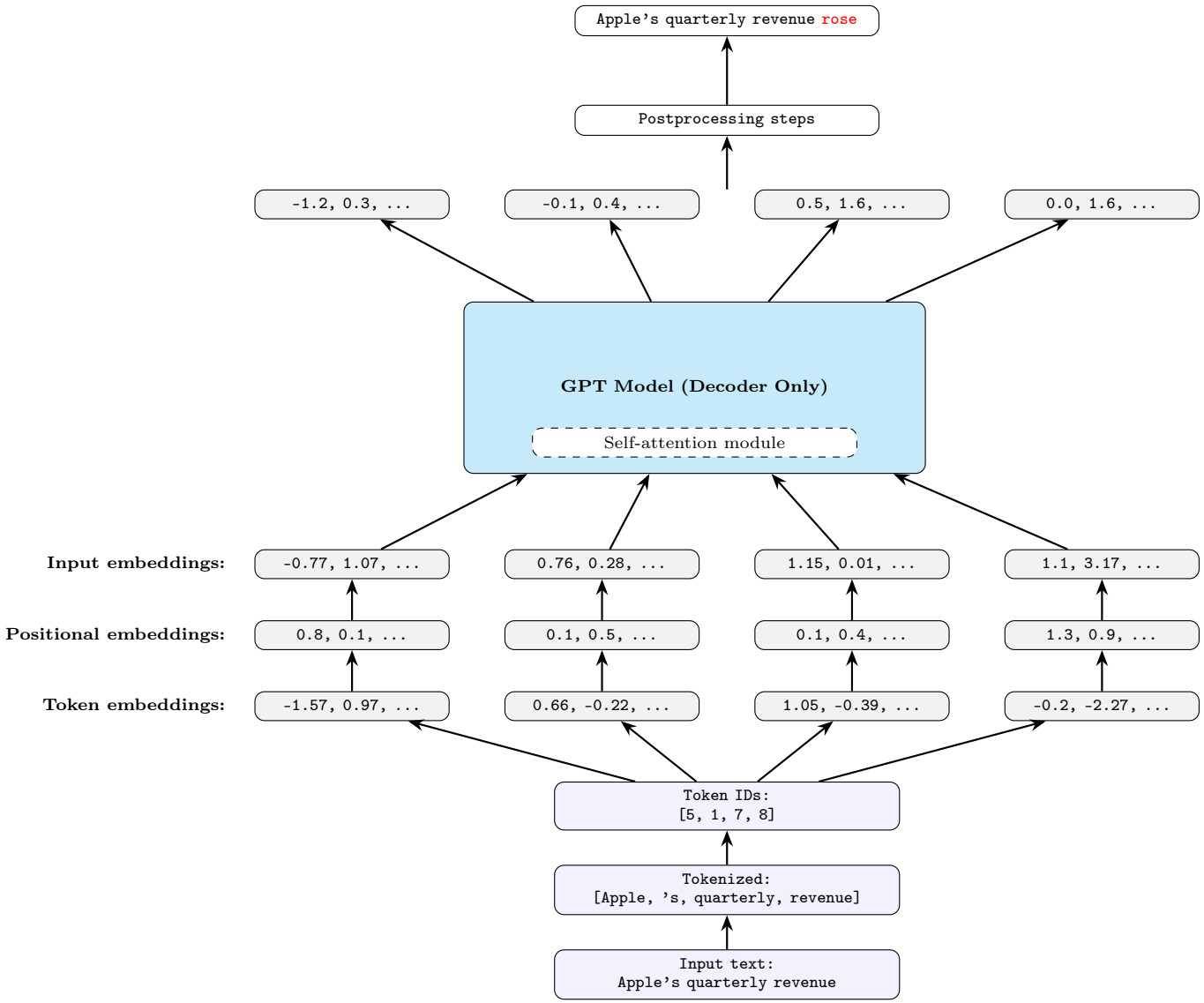
Computing Attention from Input Embeddings

Python Code: A simplified Multi-Head Wrapper

```
# context_length: sequence length
class MultiHeadAttentionWrapper(nn.Module):

    def __init__(self, d_in, d_out, context_length, dropout, num_heads,
        qkv_bias=False):
         super().__init__()
         self.heads = nn.ModuleList(
         [CausalAttention(d_in, d_out, context_length, dropout, qkv_bias)
         for _ in range(num_heads)]
         )

    def forward(self, x):
         return torch.cat([head(x) for head in self.heads], dim=-1)

torch.manual_seed(123)

context_length = batch.shape[1] # This is the number of tokens
d_in, d_out = 3, 2
mha = MultiHeadAttentionWrapper(
d_in, d_out, context_length, 0.0, num_heads=2
)

context_vecs = mha(batch)

print(context_vecs)
print("context_vecs.shape:", context_vecs.shape)
```

Model 3

From Tokenization to Prediction With Attention



Adapted from Sebastian Raschka's 2024 guide: *Build a Language Model from Scratch*.

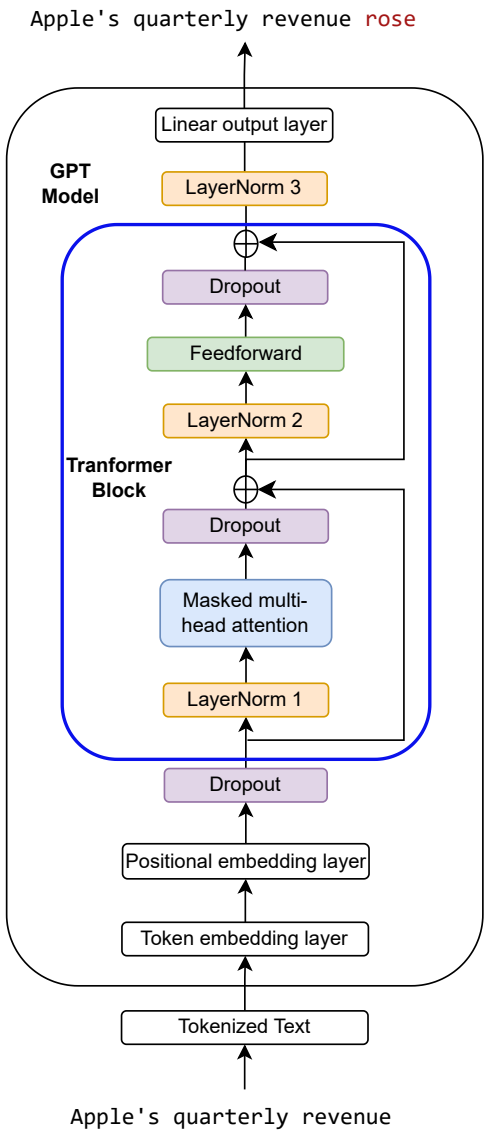
The Transformer Block

Implementing and Training an LLM Model

1. **Token Embedding:** Map tokens to dense vectors.
2. **Multi-Head Attention:** Attend to different positions/subspaces.
3. **LayerNorm:** Stabilize inputs before attention or MLP layers.
4. **GELU Activation:** Smooth, differentiable function for non-linearity.
5. **Feedforward Layers:** Use linear layers with residual connections.
6. **Dropout:** Prevent overfitting via random neuron deactivation.

All steps are integrated within a transformer block and repeated across layers.

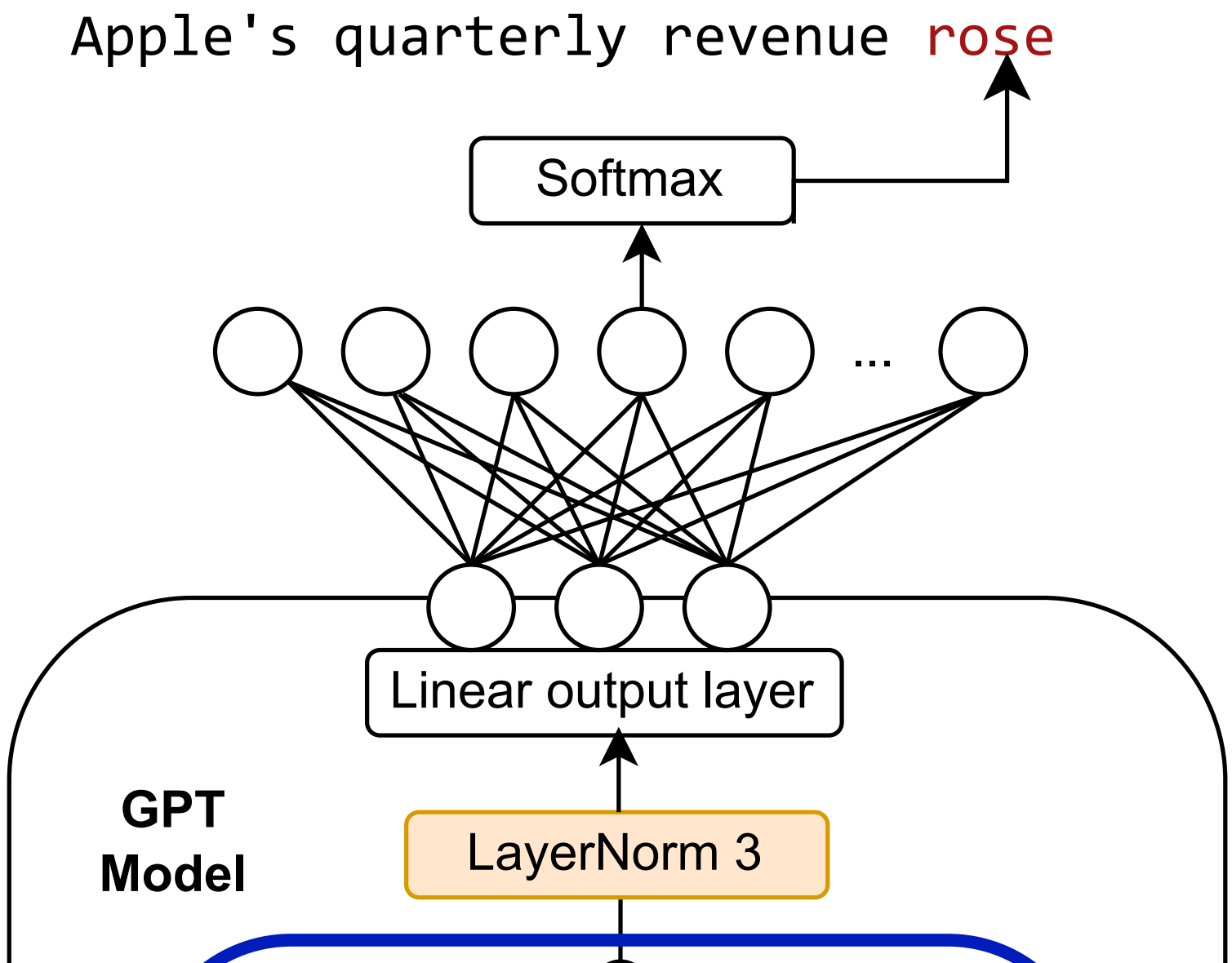
From Tokenization to Prediction With Attention



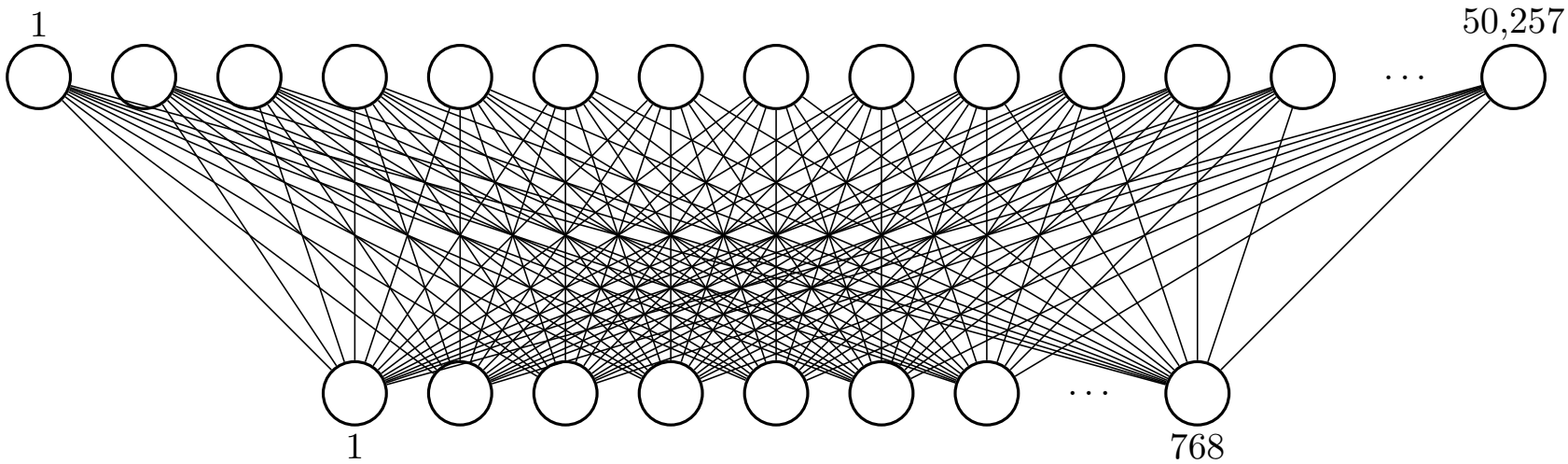
Adapted from Sebastian Raschka's 2024 guide: *Build a Language Model from Scratch*.

The Output Layer

From Tokenization to Prediction With Attention



Predicting the Next Token: Softmax and Cross-Entropy



Adapted from Sebastian Raschka's 2024 guide: *Build a Language Model from Scratch*.

- The output of the final linear layer is a vector of 50,257 logits—one per vocabulary token.
- A softmax function converts these logits into a probability distribution over the next possible tokens.
- During training, cross-entropy loss compares this predicted distribution to the true next token.

Generating Text

Generating text one word at a time

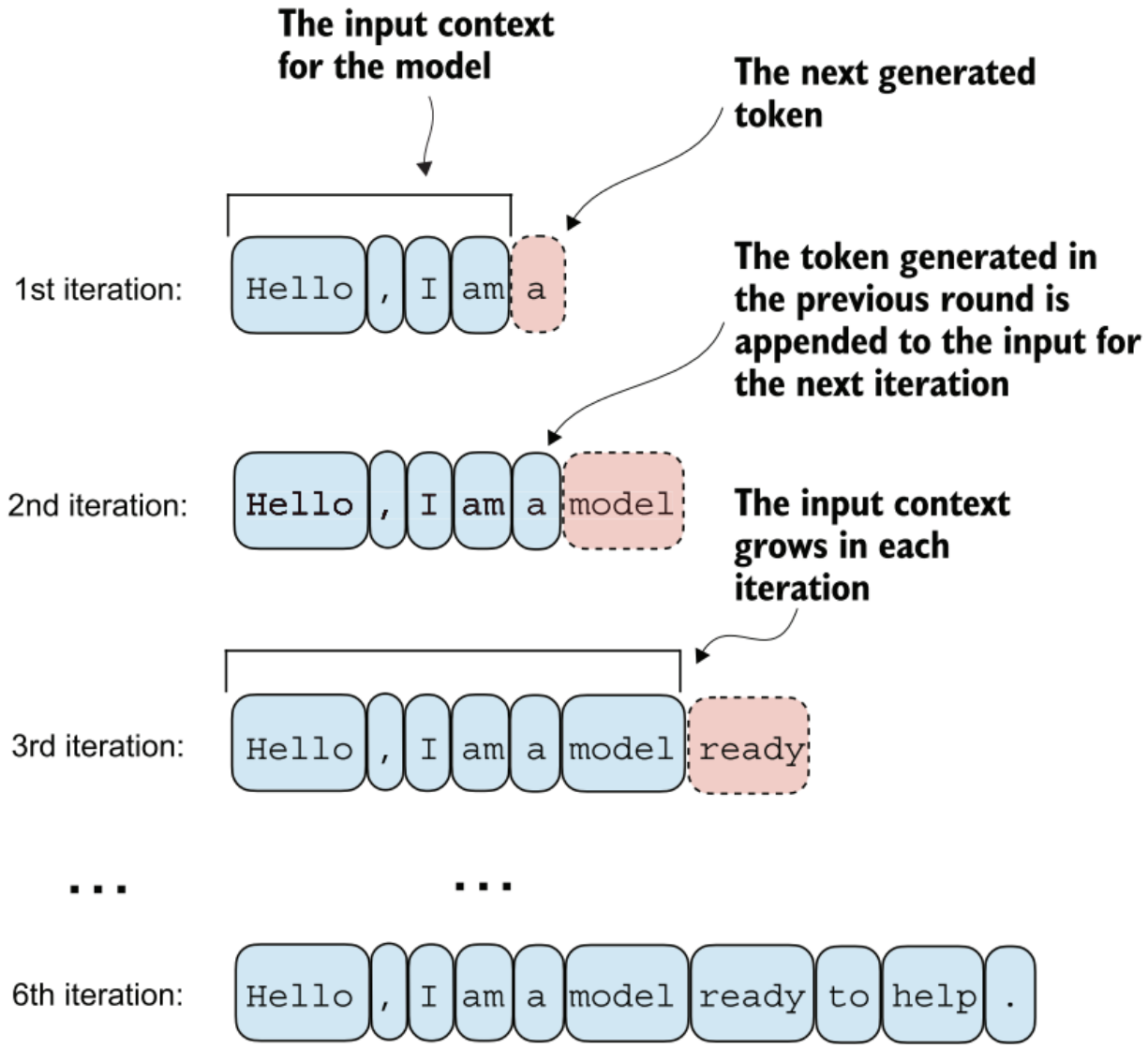


Fig 4.16: Sebastian Raschka's 2024 guide: *Build a Language Model from Scratch*.

Generating text one word at a time

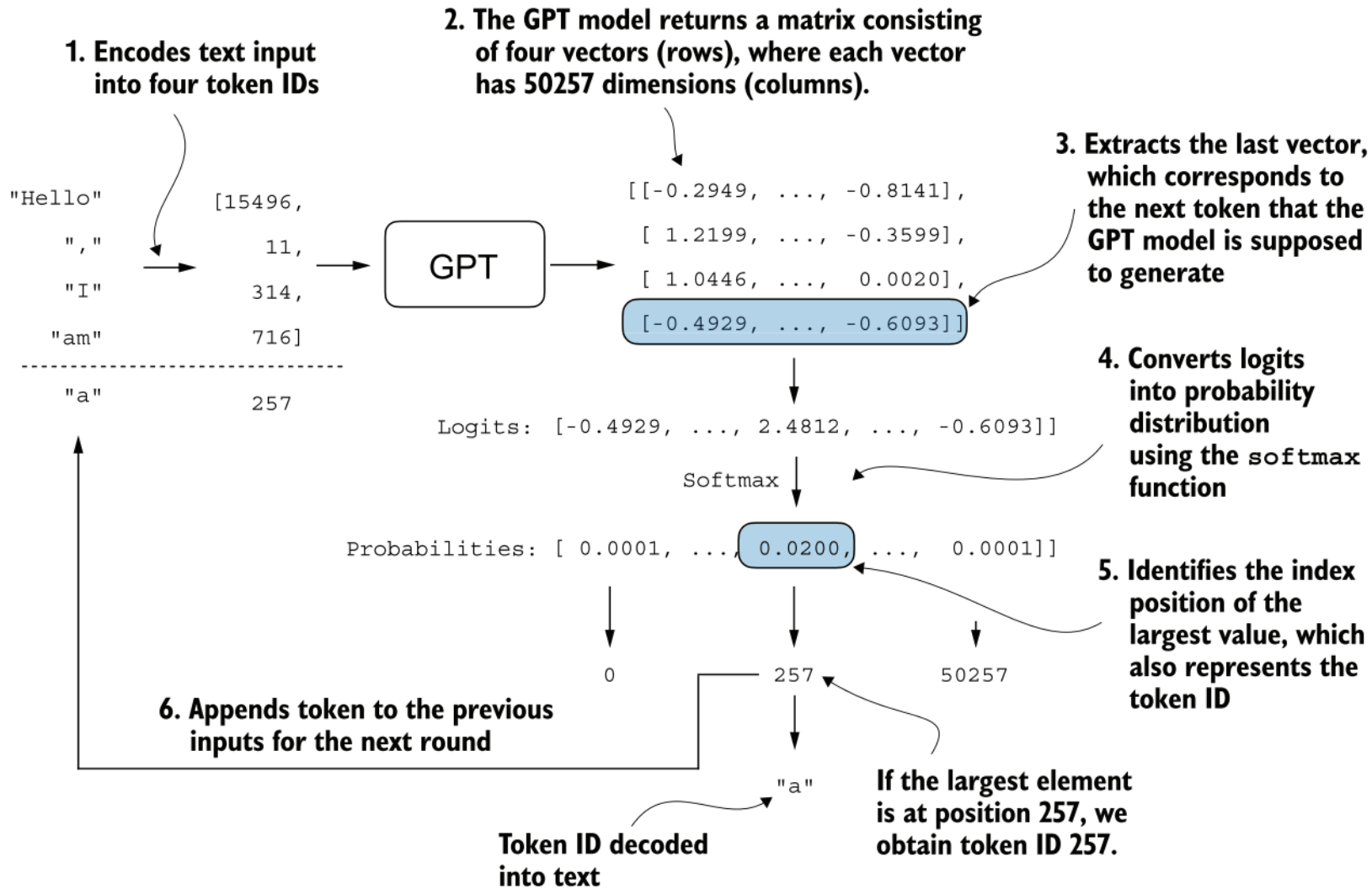


Fig 4.17: Sebastian Raschka's 2024 guide: *Build a Language Model from Scratch*.

Generating text one word at a time

Python Code: A simplified Multi-Head Wrapper

```python
def generate_text_simple(model, idx, max_new_tokens, context_size):
    # idx is (batch, n_tokens) array of indices in the current context
    for _ in range(max_new_tokens):

        # Crop current context if it exceeds the supported context size
        # E.g., if LLM supports only 5 tokens, and the context size is 10
        # then only the last 5 tokens are used as context
        idx_cond = idx[:, -context_size:]

        # Get the predictions
        with torch.no_grad():
        logits = model(idx_cond)

        # Focus only on the last time step
        # (batch, n_tokens, vocab_size) becomes (batch, vocab_size)
        logits = logits[:, -1, :]

        # Apply softmax to get probabilities
        probas = torch.softmax(logits, dim=-1)  # (batch, vocab_size)

        # Get the idx of the vocab entry with the highest probability value
        idx_next = torch.argmax(probas, dim=-1, keepdim=True)  # (batch, 1)

        # Append sampled index to the running sequence
        idx = torch.cat((idx, idx_next), dim=1)  # (batch, n_tokens+1)

    return idx
```

Reference

Reference:
Raschka, S. (2024). *Build a Large Language Model (From Scratch)*. Manning Publications. Retrieved from `https://www.manning.com/books/build-a-large-language-model-from-scratch`

?