DL Demystified 9 - The Next Frontier - Transformers

January 28, 2025

[1]:
```
#####################################################################
#                                                                   #
#  CS435 Generative AI: Security, Ethics and Governance             #
#                                                                   #
#  Instructor: Dr. Adnan Masood                                     #
#  Contact:    adnanmasood@gmail.com                                #
#                                                                   #
#  Notebook is MIT Licensed                                         #
#####################################################################
```

1 Transformers with Dr. Adnan Masood

Welcome to this comprehensive Jupyter Notebook on Transformers, Large Language Models (LLMs), Neural Networks, and Generative AI!

In this notebook, we will: 1. **Explain Transformers in Building an Intuitive Understanding** (from a middle-school-friendly explanation all the way up to a PhD-level overview). 2. **Explain the intuition** behind Transformer technology. 3. **Briefly cover the history**, invention, and underlying tech. 4. **Explain the math** behind Transformers, building on each layer step by step. 5. **Provide an illustrative example** with code. 6. **Do mock calculations** and define relevant terms (weights, bias, etc.). 7. **Create a step-by-step example** of how to build a Transformer-like model from scratch. 8. **Show an illustrative problem** it solves. 9. **Show a real-world problem** it solves. 10. **Demonstrate** how we can tackle a real-world problem using this tech. 11. **Provide questions** to illustrate the use of this tech, along with answers and code examples. 12. **Present an easy code sample** with TODO items and hints that students can complete. 13. **Give a brief glossary** of related terms at the end.

We'll be using **PyTorch** to illustrate the concepts with code.

In true Dr. Adnan Masood style, let's get started!

2 Chapter 9: The Transformer (Full Content)

Below is the **entire chapter** (Chapter 9 from *Speech and Language Processing. Daniel Jurafsky & James H. Martin. Copyright © 2024. All rights reserved. Draft of January 12, 2025.*) which forms the basis for our discussion and examples:

2.1 CHAPTER 9

The Transformer

> *"The true art of memory is the art of attention"*
> — Samuel Johnson, Idler #74, September 1759

In this chapter we introduce the transformer, the standard architecture for building large language models. Transformer-based large language models have completely changed the field of speech and language processing. Indeed, every subsequent chapter in this textbook will make use of them. We'll focus for now on left-to-right (sometimes called *causal* or *autoregressive*) language modeling, in which we are given a sequence of input tokens and predict output tokens one by one by conditioning on the prior context.

The transformer is a neural network with a specific structure that includes a mechanism called **self-attention** or **multi-head attention**.^1 Attention can be thought of as a way to build contextual representations of a token's meaning by attending to and integrating information from surrounding tokens, helping the model learn how tokens relate to each other over large spans.

1. Although multi-head attention developed historically from the RNN attention mechanism (Chapter 8), we'll d

2.1.1 Stacked Transformer Blocks

Transformer-based language models are complex, and so the details will unfold over the next 5 chapters. In the next sections we'll introduce multi-head attention, the rest of the transformer block, and the input encoding and language modeling head components. Chapter 10 discusses how language models are pretrained, and how tokens are generated via sampling. Chapter 11 introduces masked language modeling and the BERT family of bidirectional transformer encoder models. Chapter 12 shows how to prompt LLMs to perform NLP tasks by giving instructions and demonstrations, and how to align the model with human preferences. Chapter 13 will introduce machine translation with the encoder-decoder architecture.

2.2 9.1 Attention

Recall from Chapter 6 that for word2vec and other static embeddings, the representation of a word's meaning is always the same vector irrespective of the context: the word *chicken*, for example, is always represented by the same fixed vector. So a static vector for the word *it* might somehow encode that this is a pronoun used for animals and inanimate entities. But in context it has a much richer meaning.

Consider *it* in one of these two sentences:

1. The chicken didn't cross the road because it was too tired.

2. The chicken didn't cross the road because it was too wide.

In (1) *it* is the chicken (i.e., the reader knows that the chicken was tired), while in (2) *it* is the road (and the reader knows that the road was wide).^2

That is, if we are to compute the meaning of this sentence, we'll need the meaning of *it* to be associated with the *chicken* in the first sentence and associated with the *road* in the second one,

sensitive to the context. Furthermore, consider reading left to right like a causal language model, processing the sentence up to the word *it*:

The chicken didn't cross the road because it

At this point we don't yet know which thing *it* is going to end up referring to! So a representation of *it* at this point might have aspects of both *chicken* and *road* as the reader is trying to guess what happens next.

This fact that words have rich linguistic relationships with other words that may be far away pervades language. Consider two more examples:

1. The keys to the cabinet are on the table.

2. I walked along the pond, and noticed one of the trees along the bank.

In (1), the phrase *The keys* is the subject of the sentence, and in English and many languages, must agree in grammatical number with the verb *are*; in this case both are plural. In English we can't use a singular verb like *is* with a plural subject like *keys* (we'll discuss agreement more in Chapter 18). In (2), we know that *bank* refers to the side of a pond or river and not a financial institution because of the context, including words like *pond*. (We'll discuss word senses more in Chapter 11.)

The point of all these examples is that these contextual words that help us compute the meaning of words in context can be quite far away in the sentence or paragraph. Transformers can build contextual representations of word meaning, *contextual embeddings*, by integrating the meaning of these helpful contextual words. In a transformer, layer by layer, we build up richer and richer contextualized representations of the meanings of input tokens.

2.2.1 9.1.1 Attention more formally

Attention is the mechanism in the transformer that weighs and combines the representations from appropriate other tokens in the context from layer $k-1$ to layer k. A simplified version of the attention is:

$$a_i = \sum_{j \le i} \alpha_{ij} x_j$$

where α_{ij} is how much x_j should contribute to a_i. In attention, we weight each prior embedding proportionally to how similar it is to the current token x_i. We compute similarity scores via dot product, normalize with softmax, etc.

A single attention head using query, key, and value matrices Transformers use multiple weight matrices W^Q, W^K, W^V to transform each input vector x_i into **query**, **key**, and **value** representations. Then we compute dot products of query vs. key, scale them, apply softmax, and use them to weight the values:

$$q_i = x_i W^Q; \quad k_j = x_j W^K; \quad v_j = x_j W^V;$$

$$score(x_i, x_j) = \frac{q_i \cdot k_j}{\sqrt{d_k}};$$

$$\alpha_{ij} = softmax(score(x_i, x_j)) \quad \forall j \leq i;$$
$$head_i = \sum_{j \leq i} \alpha_{ij} v_j; \quad a_i = head_i W^O.$$

In **multi-head attention**, we repeat this process with multiple heads, each with its own W^Q, W^K, W^V. Then we concatenate the outputs of each head and project them back to dimension d with another weight matrix.

2.3 9.2 Transformer Blocks

The self-attention calculation lies at the core of what's called a transformer block, which, in addition to the self-attention layer, includes three other kinds of layers: (1) a feedforward layer, (2) residual connections, and (3) normalizing layers (layer norm).

2.3.1 Layer Norm

Layer normalization ($LayerNorm$) helps keep the values of hidden layers in a stable range for gradient-based training. For a given vector $x \in \mathbb{R}^d$:

$$\mu = \frac{1}{d} \sum_{i=1}^{d} x_i, \quad \sigma = \sqrt{\frac{1}{d} \sum_{i=1}^{d} (x_i - \mu)^2}.$$

Then

$$\hat{x} = \frac{x - \mu}{\sigma}, \quad LayerNorm(x) = \gamma \hat{x} + \beta.$$

2.3.2 Feedforward Layer

$$FFN(x) = ReLU(xW_1 + b_1)W_2 + b_2.$$

2.3.3 Putting it all together

The function computed by a transformer block can be expressed as:

$$T_1 = LayerNorm(X), \quad T_2 = MultiHeadAttention(T_1), \quad T_3 = T_2 + X,$$
$$T_4 = LayerNorm(T_3), \quad T_5 = FFN(T_4), \quad H = T_5 + T_3.$$

We stack many such transformer blocks for large language models (12, 24, 96, etc.).

2.4 9.3 Parallelizing computation using a single matrix X

Instead of processing one token at a time, we pack input embeddings for the N tokens of the input into a single matrix $X \in \mathbb{R}^{N \times d}$. This lets us efficiently compute attention using matrix multiplication.

2.5 9.4 The input: embeddings for token and position

To represent inputs, we combine **word embeddings** (from an embedding matrix E) and **positional embeddings** to form $X \in \mathbb{R}^{N \times d}$. For left-to-right language models, we also add a **causal mask** so that each token can only attend to tokens at earlier positions.

2.6 9.5 The Language Modeling Head

Finally, for language modeling, we have a **language modeling head** on top of the final transformer block output. Often we *tie* the unembedding matrix to the embedding matrix E by using its transpose. If h_N^L is the final hidden state for the last token, then the logits u and word probabilities y are:

$$u = h_N^L E^T, \quad y = softmax(u).$$

We then sample (or otherwise decode) from these probabilities.

2.7 9.6 Summary

- Transformers use **multi-head self-attention** to combine information from across the input.
- A **transformer block** has self-attention, a feedforward layer, residual connections, and layer norms.
- For language modeling, we use a **decoder-only** transformer that is masked so each token can only attend to previous tokens.

2.8 Bibliographical and Historical Notes

The transformer (Vaswani et al., 2017) was developed from earlier concepts of self-attention and memory networks. The concept of attention originated in RNN-based sequence-to-sequence models (Bahdanau et al., 2015), then extended to self-attention (Cheng et al., 2016). Other aspects derived from memory networks (Weston et al., 2015; Graves et al., 2014).

---

2. We say that *it* corefers with *chicken* in the first example and with *road* in the second. We return to coreferenc

---

3 Building an Intuitive Understanding

We will now explain **Transformers** in different levels of detail. Pick the explanation that resonates with your current understanding, and feel free to read them all for a layered perspective.

Imagine you're reading a story. Each word in the story needs to know about the words before it to make sense of what is happening. A **Transformer** is like a big helper that looks at all the words you've already read and decides which ones are most important to understanding the next word. It does this using a trick called **attention**, which helps the model "focus" on the right words when figuring out the meaning of the current word. By doing this many times, Transformers learn to predict and create text that sounds just like people wrote it.

A Transformer takes a sentence (or a large chunk of text) and turns each word into a bunch of numbers (an *embedding*). Then, it uses something called **self-attention** to see how each word connects to every other word. This way, the model isn't just looking at neighbors (like a small window around a word), it can look at the *whole sentence or paragraph* at once. This self-attention step is repeated in layers, each time refining how the words relate to each other. In the end, the model learns a powerful representation of the text that lets it do tasks like predicting the next word or completing entire sentences.

Transformers represent each token (word or subword) in a vector space. They compute **query**, **key**, and **value** vectors for each token in parallel. The **queries** are used to look up relevant **keys** across all tokens, and the corresponding **values** are combined based on similarity scores. This process, known as **multi-head self-attention**, is repeated several times with **feedforward layers**, **residual connections**, and **layer normalization**. By stacking multiple Transformer blocks, the model captures global context and learns intricate linguistic patterns. The final layer, known as the **language modeling head**, transforms these representations into a distribution over possible next tokens (words).

A Transformer encodes each input token x_i into a vector. Within each Transformer block, a **multi-head self-attention** layer is applied: each head has distinct learnable weight matrices W^Q, W^K, W^V to create query (q_i), key (k_j), and value (v_j) vectors. We compute dot products $q_i \cdot k_j$, scale them by $\sqrt{d_k}$, apply a softmax, and multiply by v_j to get the final attention output. Multiple heads are concatenated and projected to maintain dimension consistency. This output feeds into a **feedforward network** with potential dimension expansion. **Residual connections** add the input of each sub-layer to its output, and **layer normalization** maintains stable activations. Stacking multiple such layers yields a deep, contextualized representation of all tokens. Positional embeddings or encodings are added to represent sequence order. For language modeling, a final projection (often tied to the embedding matrix) provides logits over the vocabulary.

The Transformer can be viewed as a composition of multiple learned functions on top of an initial embedded input $X \in \mathbb{R}^{N \times d}$. Each **Transformer block** implements:

$$T_1 = LayerNorm(X), \quad T_2 = MultiHeadAttention(T_1), \quad T_3 = T_2 + X,$$
$$T_4 = LayerNorm(T_3), \quad T_5 = FFN(T_4), \quad H = T_5 + T_3.$$

We can interpret multi-head attention as multiple parallel subspaces focusing on distinct relational patterns. The attention matrix is $\text{softmax}((XW^Q)(XW^K)^T/\sqrt{d_k})$. The feedforward network introduces a non-linear transformation (often ReLU) with dimension expansion to d_{ff}, then back to d. These blocks are repeated L times to form deeper models. Causal (left-to-right) masking ensures that the model cannot attend to future tokens, preserving autoregressive capabilities. Output embeddings are mapped back to vocabulary logits via weight tying with E^T. This architecture's parallelizable design yields remarkable efficiency and expressivity, forming the backbone of modern large language models such as GPT variants.

4 Intuition Behind the Technology

The **key idea** behind Transformers is **attention**: - Instead of looking at text in a strict left-to-right manner (like older recurrent networks), or in fixed windows (like convolutional networks), the Transformer can look at every token in a sequence (or a chunk of the sequence) in relation to every other token. - This means each word can gather meaning from all relevant context words at once. - Multi-head attention means multiple ways of looking at these relationships.

This simultaneously addresses the need for long-range context (the model can easily connect distant parts of a paragraph) and the bottleneck of sequential processing (the Transformer architecture is parallelizable).

5 Brief History, Invention, and Underlying Tech

Historically: - **Attention** first appeared in RNN-based sequence models for machine translation (Bahdanau et al., 2015), allowing the decoder to look back at relevant portions of the input. - The idea of **self-attention** was explored to let each token attend to the rest of the sequence. - The **Transformer** was introduced by Vaswani et al. (2017) in the seminal paper *"Attention Is All You Need"*. It got rid of recurrent and convolutional structures entirely and relied solely on attention mechanisms.

The underlying technology includes: - **Dot-product attention** for similarity computation. - **Feedforward layers** for non-linear transformations. - **Positional embeddings** to handle the ordering of tokens. - **Residual connections** and **layer normalization** to facilitate training of deep networks. - Parallel processing enabled by representing the entire sequence as a matrix.

6 The Math Behind Transformers

1. **Input Representation**: Suppose you have a sequence of tokens (words) $[w_1, w_2, ..., w_N]$. Each token is turned into a numerical vector of size d (the *embedding*). We place these row-wise into a matrix $X \in \mathbb{R}^{N \times d}$.
2. **Query, Key, Value**: For each token, we create three versions of its embedding:
   - **Query**: $Q = XW^Q$
   - **Key**: $K = XW^K$
   - **Value**: $V = XW^V$ where W^Q, W^K, W^V are learnable matrices, and the resulting Q, K, V each have shape $(N \times d_k), (N \times d_k), (N \times d_v)$ respectively.
3. **Attention Weights**: Compute the similarity of every query with every key by multiplying QK^T. This gives an $(N \times N)$ matrix of similarity scores. Scale by $\sqrt{d_k}$ and apply a softmax, yielding an attention matrix α of shape $(N \times N)$.
4. **Weighted Sum of Values**: Multiply α by V to combine the value vectors in proportion to their relevance.
5. **Multi-head**: Repeat the above process with multiple sets of (W^Q, W^K, W^V) to capture different aspects of the input. Concatenate all the heads' outputs and project back to dimension d.
6. **Feedforward + Residual + LayerNorm**: For each token's output, apply a small 2-layer MLP, add a residual connection, and do a layer normalization.
7. **Stack** the above block multiple times. Finally, use a linear + softmax (the **language modeling head**) to produce probabilities over possible next tokens.

7 Intuitive Example + Code

Let's illustrate the self-attention idea with code in a simple setting. We'll create a tiny vocabulary of words and do a toy demonstration.

[2]:
```
# Let's do a small demonstration with a few tokens.
import torch
import torch.nn.functional as F

# Suppose we have an input of 3 tokens, each mapped to an embedding dim=4 (just␣
↪for demonstration)
```

```python
N = 3  # number of tokens
d = 4  # embedding dimension

# Example random embeddings for each token
x = torch.randn(N, d)
print('Input embedding matrix x (shape: 3x4):')
print(x)

# We'll create separate W^Q, W^K, W^V for a single head
d_k = d_v = 4  # for simplicity, keep them the same as d in this tiny example
W_Q = torch.randn(d, d_k)
W_K = torch.randn(d, d_k)
W_V = torch.randn(d, d_v)

# Step 1: compute Q, K, V
Q = x @ W_Q  # shape [3, 4]
K = x @ W_K  # shape [3, 4]
V = x @ W_V  # shape [3, 4]

# Step 2: compute similarity scores QK^T
scores = Q @ K.t()  # shape [3,3]

# Step 3: scale by sqrt(d_k)
scores = scores / (d_k ** 0.5)

# Step 4: softmax to get attention weights
alpha = F.softmax(scores, dim=-1)

# Step 5: Weighted sum of V
attn_out = alpha @ V  # shape [3, 4]

print('\nAttention weights (alpha):')
print(alpha)
print('\nOutput of single-head self-attention (attn_out):')
print(attn_out)
```

```
Input embedding matrix x (shape: 3x4):
tensor([[-0.8706,  1.2082, -0.3282, -0.1205],
        [-0.5702,  1.2321,  0.0736, -2.2898],
        [ 1.0732, -0.3480, -0.0392, -0.2137]])

Attention weights (alpha):
tensor([[3.0493e-02, 9.2348e-01, 4.6022e-02],
        [1.2505e-05, 9.9994e-01, 5.2140e-05],
        [9.1948e-01, 1.5564e-02, 6.4958e-02]])

Output of single-head self-attention (attn_out):
tensor([[ 1.5170, -0.5098,  3.7124,  2.6618],
```

```
[ 1.6211, -0.4794,  3.9923,  2.8912],
[ 0.6890,  1.5290,  1.8116,  1.1870]])
```

Above: 1. We started with a random embedding matrix x of shape $(3, 4)$, representing 3 tokens, each mapped to 4 dimensions. 2. We created random projection matrices for Query, Key, Value (W^Q, W^K, W^V). 3. We computed $\mathbf{Q = xW\hat{}Q}$, $\mathbf{K = xW\hat{}K}$, $\mathbf{V = xW\hat{}V}$. 4. We calculated the dot-product between every Query and every Key (QK^T), scaled it, and then did a softmax to get our attention distribution. 5. Finally, we did a weighted sum of the Value vectors, producing a new representation.

This is the core of how a Transformer decides "which tokens to pay attention to."

8 Example Calculations

Let's define some key terms often encountered in Transformer math:

- **Weight (W)**: A learnable parameter matrix (like W^Q in our code). Shapes typically match input dimensions and output dimensions.
- **Bias (b)**: An additional learnable parameter (vector) sometimes used in feedforward layers (not shown in the above snippet, but used in standard linear layers).
- **Dot Product**: The multiplication of two vectors, resulting in a scalar that indicates how similar or aligned they are.
- **Softmax**: A function that turns a vector of real numbers into probabilities that sum to 1.
- **Mask**: In causal language modeling, we mask out future tokens so the model doesn't "cheat" by seeing them.

8.0.1 Example Calculation Step-by-step (Conceptual):

1. **Token embedding**: The word "Cat" might be mapped to [0.2, 0.9, -0.1, 0.05].
2. **Query projection**: Multiply [0.2, 0.9, -0.1, 0.05] by W^Q (4x4 in our toy code) = new 4D vector.
3. **Key projection**: Similarly, each token's embedding is mapped by W^K.
4. **Dot-product**: For the current token's query, we do dot-products with each key vector, including itself.
5. **Scale & Softmax**: Divide by $\sqrt{d_k}$, apply softmax to get weights.
6. **Weighted sum**: Multiply these weights by each token's value vector, then sum.
7. **Output**: This is the new representation of the current token, capturing context from other tokens.

9 Step by Step: Building the Technology from Scratch

Here's a **conceptual** 7-step approach to building a minimal Transformer-like language model **from scratch**:

1. **Tokenization**: Decide how to split text into tokens (words, subwords, etc.) and map them to integer IDs.
2. **Create Embeddings**: Initialize an embedding matrix E with shape $(|V|, d)$. For each token, you get its vector by index lookup.

3. **Positional Encoding**: Add position encodings (sine/cosine or learned) to each token's embedding to incorporate order.
4. **Multi-head Self-Attention**: Implement code that for each token, calculates q, k, v vectors, does dot-product attention, and sums the results.
5. **Feedforward Layer**: A small 2-layer MLP with a ReLU (or another activation) and possibly dimension expansion.
6. **Residual Connections & Layer Norm**: Add skip connections and layer normalization around the attention and feedforward sub-layers.
7. **Language Modeling Head**: Tie or create a new matrix to map final hidden states back to the vocabulary, produce a distribution via softmax.

Repeat multiple times (stack more Transformer blocks) to get more capacity.

10 Illustrative Problem It Solves

A classic illustrative problem is **next-word prediction** or **text auto-completion**. By looking at the prior sequence of words, the Transformer predicts the *most likely* next token. This is the basis of technologies like GPT, ChatGPT, etc. If your input text is:

> "Neural networks are a class of models loosely inspired by biological…"

A Transformer-based language model can guess the next word might be "brains," "neurons," or "neural processes," etc., ranking them by probability.

11 Real World Problem It Solves

One major real-world application of Transformers is **Machine Translation**. By using an *encoder-decoder* Transformer architecture (we only covered the decoder-only part here), you can translate from, say, French to English by learning to attend to relevant words in the French sentence for each word you generate in English.

Another example is **summarization**: Transformers can condense large documents into shorter summaries while retaining the salient points.

12 How to Solve a Real-World Problem (Text Summarization Example)

1. **Collect Data**: Get a dataset of documents and reference summaries.
2. **Build/Use a Transformer**: Use an encoder-decoder style or a large language model that can handle summarization tasks.
3. **Train/Fine-tune**: If building from scratch, train the model on your data. If using a pre-trained model, fine-tune on your summarization dataset.
4. **Inference**: For a new document, pass the text to the model. The model attends to the important parts of the text and generates a concise summary.

Common frameworks: **HuggingFace Transformers**, **PyTorch Lightning**, etc.

13 Questions to Illustrate the Use of This Tech

Here are some prompts or questions you could ask to highlight the use of this tech:

1. **"How can a Transformer handle long-distance dependencies in text?"**
   Answer: By using self-attention that can attend to any token in the sequence, not just local neighbors.
2. **"Why do we need positional embeddings if the model can see all tokens at once?"**
   Answer: Because without some notion of order, the model wouldn't know how the tokens are arranged in time/space.
3. **"What is multi-head attention?"**
   Answer: Multiple attention heads allow the model to learn different types of relationships in parallel.
4. **"Why use residual connections and layer norm?"**
   Answer: They help stabilize and speed up training, preventing vanishing or exploding gradients and allowing deeper architectures.
5. **"How is a Transformer different from an RNN?"**
   Answer: An RNN processes tokens sequentially, which can be slower for long sequences, while a Transformer processes all tokens in parallel via attention.
6. **"What does the language modeling head do?"**
   Answer: It projects the final hidden state of each token position into a distribution over the vocabulary, allowing next-token prediction.
7. **"How do we prevent a Transformer from looking at future tokens during training for language modeling?"**
   Answer: Use a causal mask that sets attention weights to zero for tokens that occur after the current position.

14 A Sample Exercise

Below is a minimal PyTorch example for a small Transformer-based language model. Some parts are marked as **TODO** for you to complete!

[4]:
```
import torch
import torch.nn as nn
import torch.nn.functional as F

########################################
# HYPERPARAMETERS
########################################
d_model = 16  # Embedding dimension
num_tokens = 50  # Just a small vocab for demonstration
max_seq_len = 10  # We'll limit sequence length for simplicity
num_heads = 2   # Multi-head attention heads
num_layers = 2  # Number of Transformer blocks
batch_size = 4  # We'll process 4 sequences at a time
```

```
########################################
# SAMPLE DATA (RANDOM)
# In real life, you'd have a proper dataset of token IDs
########################################
# We'll create random sequences of integer token IDs in [0, num_tokens)
torch.manual_seed(42)
dummy_inputs = torch.randint(0, num_tokens, (batch_size, max_seq_len))
dummy_targets = torch.randint(0, num_tokens, (batch_size, max_seq_len))

########################################
# POS EMBEDDING (LEARNED)
########################################
class PositionalEncoding(nn.Module):
    def __init__(self, d_model, max_len=5000):
        super().__init__()
        # Create a long enough positional embedding for all positions
        pe = torch.zeros(max_len, d_model)
        position = torch.arange(0, max_len, dtype=torch.float).unsqueeze(1)
        div_term = torch.exp(torch.arange(0, d_model, 2).float() * -(torch.
 ↪log(torch.tensor(10000.0)) / d_model))
        pe[:, 0::2] = torch.sin(position * div_term)
        pe[:, 1::2] = torch.cos(position * div_term)
        self.register_buffer('pe', pe)
    def forward(self, x):
        # x shape: (batch_size, seq_len, d_model)
        seq_len = x.size(1)
        x = x + self.pe[:seq_len, :]
        return x

########################################
# BASIC TRANSFORMER BLOCK
########################################
class TransformerBlock(nn.Module):
    def __init__(self, d_model, num_heads, dim_feedforward=64):
        super().__init__()
        # TODO: create a nn.MultiheadAttention layer
        self.attn = nn.MultiheadAttention(embed_dim=d_model,␣
 ↪num_heads=num_heads, batch_first=True)

        self.ff = nn.Sequential(
            nn.Linear(d_model, dim_feedforward),
            nn.ReLU(),
            nn.Linear(dim_feedforward, d_model)
        )

        self.norm1 = nn.LayerNorm(d_model)
```

```
        self.norm2 = nn.LayerNorm(d_model)

    def forward(self, x):
        # x shape: (batch_size, seq_len, d_model)
        # Self-attention
        attn_out, _ = self.attn(x, x, x)  # (batch_size, seq_len, d_model)
        x = x + attn_out
        x = self.norm1(x)

        # Feedforward
        ff_out = self.ff(x)
        x = x + ff_out
        x = self.norm2(x)
        return x

########################################
# MINI TRANSFORMER LANGUAGE MODEL
########################################
class MiniTransformerLM(nn.Module):
    def __init__(self, num_tokens, d_model, max_seq_len, num_heads=2,␣
 ↪num_layers=2):
        super().__init__()
        self.token_embed = nn.Embedding(num_tokens, d_model)
        self.pos_embed = PositionalEncoding(d_model, max_len=max_seq_len)

        self.layers = nn.ModuleList([
            TransformerBlock(d_model, num_heads) for _ in range(num_layers)
        ])

        # Final linear layer to map d_model -> vocab size
        self.output_lin = nn.Linear(d_model, num_tokens)

    def forward(self, x):
        # x shape: (batch_size, seq_len)
        embed = self.token_embed(x)  # shape: (batch_size, seq_len, d_model)
        out = self.pos_embed(embed)

        for layer in self.layers:
            out = layer(out)

        logits = self.output_lin(out)  # (batch_size, seq_len, num_tokens)
        return logits

# Create model
model = MiniTransformerLM(num_tokens, d_model, max_seq_len, num_heads,␣
 ↪num_layers)
print(model)
```

```python

# Run a forward pass
logits = model(dummy_inputs)
print('Logits shape:', logits.shape)

# Compare with targets
loss_fn = nn.CrossEntropyLoss()
# We'll flatten batch and sequence dims to compare with vocab
logits_2d = logits.view(-1, num_tokens)
targets_1d = dummy_targets.view(-1)
loss = loss_fn(logits_2d, targets_1d)
print('Loss:', loss.item())

# Basic training step: backward
loss.backward()
# In real training, you'd update model params with an optimizer step
# e.g.: optimizer.step()
```

```
MiniTransformerLM(
  (token_embed): Embedding(50, 16)
  (pos_embed): PositionalEncoding()
  (layers): ModuleList(
    (0-1): 2 x TransformerBlock(
      (attn): MultiheadAttention(
        (out_proj): NonDynamicallyQuantizableLinear(in_features=16,
out_features=16, bias=True)
      )
      (ff): Sequential(
        (0): Linear(in_features=16, out_features=64, bias=True)
        (1): ReLU()
        (2): Linear(in_features=64, out_features=16, bias=True)
      )
      (norm1): LayerNorm((16,), eps=1e-05, elementwise_affine=True)
      (norm2): LayerNorm((16,), eps=1e-05, elementwise_affine=True)
    )
  )
  (output_lin): Linear(in_features=16, out_features=50, bias=True)
)
Logits shape: torch.Size([4, 10, 50])
Loss: 4.222940444946289
```

14.0.1 TODO Items & Hints

- **TODO**: Implement a causal mask to ensure the model only attends to previous positions. (Hint: in PyTorch's `nn.MultiheadAttention`, you can pass an `attn_mask`.)
- **TODO**: Add a proper training loop with an optimizer and multiple epochs.
- **TODO**: Try a real text dataset, e.g. small samples of Wikipedia.
- **HINT**: Investigate how you can tie the weights in the `output_lin` layer with the `token_embed`

matrix if you want weight tying.

[5]:
```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim

# HYPERPARAMETERS
d_model = 16  # Embedding dimension
num_tokens = 50  # Just a small vocab for demonstration
max_seq_len = 10  # We'll limit sequence length for simplicity
num_heads = 2   # Multi-head attention heads
num_layers = 2  # Number of Transformer blocks
batch_size = 4  # We'll process 4 sequences at a time
epochs = 5      # Number of training epochs
learning_rate = 0.01

# SAMPLE DATA (RANDOM)
torch.manual_seed(42)
dummy_inputs = torch.randint(0, num_tokens, (batch_size, max_seq_len))
dummy_targets = torch.randint(0, num_tokens, (batch_size, max_seq_len))

# POS EMBEDDING (LEARNED)
class PositionalEncoding(nn.Module):
    def __init__(self, d_model, max_len=5000):
        super().__init__()
        pe = torch.zeros(max_len, d_model)
        position = torch.arange(0, max_len, dtype=torch.float).unsqueeze(1)
        div_term = torch.exp(torch.arange(0, d_model, 2).float() * -(torch.
 ↪log(torch.tensor(10000.0)) / d_model))
        pe[:, 0::2] = torch.sin(position * div_term)
        pe[:, 1::2] = torch.cos(position * div_term)
        self.register_buffer('pe', pe)

    def forward(self, x):
        seq_len = x.size(1)
        x = x + self.pe[:seq_len, :]
        return x

# BASIC TRANSFORMER BLOCK WITH CAUSAL MASK
class TransformerBlock(nn.Module):
    def __init__(self, d_model, num_heads, dim_feedforward=64):
        super().__init__()
        self.attn = nn.MultiheadAttention(embed_dim=d_model,␣
 ↪num_heads=num_heads, batch_first=True)
        self.ff = nn.Sequential(
            nn.Linear(d_model, dim_feedforward),
```

```
            nn.ReLU(),
            nn.Linear(dim_feedforward, d_model)
        )
        self.norm1 = nn.LayerNorm(d_model)
        self.norm2 = nn.LayerNorm(d_model)

    def forward(self, x, attn_mask=None):
        attn_out, _ = self.attn(x, x, x, attn_mask=attn_mask)
        x = x + attn_out
        x = self.norm1(x)
        ff_out = self.ff(x)
        x = x + ff_out
        x = self.norm2(x)
        return x

# MINI TRANSFORMER LANGUAGE MODEL
class MiniTransformerLM(nn.Module):
    def __init__(self, num_tokens, d_model, max_seq_len, num_heads=2,␣
 ↪num_layers=2):
        super().__init__()
        self.token_embed = nn.Embedding(num_tokens, d_model)
        self.pos_embed = PositionalEncoding(d_model, max_len=max_seq_len)
        self.layers = nn.ModuleList([
            TransformerBlock(d_model, num_heads) for _ in range(num_layers)
        ])
        self.output_lin = nn.Linear(d_model, num_tokens)

        # Weight tying
        self.output_lin.weight = self.token_embed.weight

    def generate_causal_mask(self, seq_len):
        return torch.triu(torch.ones(seq_len, seq_len) * float('-inf'),␣
 ↪diagonal=1).to(next(self.parameters()).device)

    def forward(self, x):
        embed = self.token_embed(x)
        out = self.pos_embed(embed)

        attn_mask = self.generate_causal_mask(out.size(1))
        for layer in self.layers:
            out = layer(out, attn_mask=attn_mask)

        logits = self.output_lin(out)
        return logits

model = MiniTransformerLM(num_tokens, d_model, max_seq_len, num_heads,␣
 ↪num_layers)
```

```python
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model = model.to(device)

# Loss and optimizer
loss_fn = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=learning_rate)

# Move data to device
dummy_inputs, dummy_targets = dummy_inputs.to(device), dummy_targets.to(device)

# Training loop
for epoch in range(epochs):
    model.train()
    optimizer.zero_grad()

    # Forward pass
    logits = model(dummy_inputs)

    # Flatten logits and targets for CrossEntropyLoss
    logits_2d = logits.view(-1, num_tokens)
    targets_1d = dummy_targets.view(-1)

    loss = loss_fn(logits_2d, targets_1d)

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

    print(f"Epoch [{epoch+1}/{epochs}], Loss: {loss.item():.4f}")

# TESTING LOOP (USING RANDOM DATA)
model.eval()
with torch.no_grad():
    logits = model(dummy_inputs)
    _, predicted = torch.max(logits, dim=-1)
    print("Predictions:", predicted)
    print("Targets:    ", dummy_targets)
```

```
Epoch [1/5], Loss: 12.4078
Epoch [2/5], Loss: 8.8194
Epoch [3/5], Loss: 6.9047
Epoch [4/5], Loss: 5.6511
Epoch [5/5], Loss: 4.8191
Predictions: tensor([[11,  3, 19, 19, 19, 35, 19, 11, 43, 43],
        [19, 19, 11, 35, 11, 19, 11, 11, 11, 19],
        [11, 11, 11, 11, 11, 11, 35, 11, 11, 11],
        [ 4,  4, 44,  3, 44, 35, 44, 44,  8, 11]], device='cuda:0')
Targets:     tensor([[36, 22, 27, 19,  7, 23, 43, 44, 43, 27],
```

```
        [20,  9, 20, 39, 36, 49, 15, 34, 28,  8],
        [16, 30, 40, 10, 20, 11, 23, 20, 11, 11],
        [27, 39, 34,  3, 48,  9, 33,  7,  8, 41]], device='cuda:0')
```

15 Brief Glossary of Related Terms

- **Attention**: A mechanism that computes a weighted combination of other elements in a sequence, based on similarity.
- **Self-Attention**: Attention applied among tokens within the same sequence, allowing each token to attend to others.
- **Multi-Head Attention**: Multiple parallel attention heads that each learn different relational patterns.
- **Residual Connection**: Adding the input of a layer to its output to help training deeper models.
- **Layer Normalization**: A method of normalizing the hidden layer for more stable and faster training.
- **Feedforward Network (FFN)**: A small MLP (2-layer typically) inside each Transformer block.
- **Causal Mask**: A mask preventing the model from attending to future tokens in language modeling.
- **Embedding Matrix**: A matrix that maps token IDs to dense vectors.
- **Weight Tying**: Reusing the same weight matrix for embedding and output layers (via transposition).
- **Positional Embeddings**: Extra vectors added to token embeddings to encode position/order.
- **Vocabulary (V)**: The set of all possible tokens (words, subwords) the model can output.
- **Logits**: The raw, unnormalized scores over the vocabulary, transformed by a softmax to become probabilities.
- **Causal (Decoder-Only) Transformer**: A Transformer that only looks backward in the sequence (past tokens) for language modeling tasks.

[6]:
```python
import os, sys, platform, datetime, uuid, socket

def signoff(name="Anonymous"):
    colab_check = "Yes" if 'google.colab' in sys.modules else "No"
    mac_addr = ':'.join(format((uuid.getnode() >> i) & 0xff, '02x') for i in␣
 ↪reversed(range(0, 48, 8)))
    print("+++ Acknowledgement +++")
    print(f"Executed on: {datetime.datetime.now()}")
    print(f"In Google Colab: {colab_check}")
    print(f"System info: {platform.system()} {platform.release()}")
    print(f"Node name: {platform.node()}")
    print(f"MAC address: {mac_addr}")
    try:
        print(f"IP address: {socket.gethostbyname(socket.gethostname())}")
    except:
        print("IP address: Unknown")
```

```
    print(f"Signing off, {name}")

signoff("Ali Muhammad Asad")
```

```
+++ Acknowledgement +++
Executed on: 2025-01-28 18:27:18.494543
In Google Colab: No
System info: Linux 6.8.0-51-generic
Node name: alimuhammad-Inspiron-7559
MAC address: 20:47:47:74:94:05
IP address: 127.0.1.1
Signing off, Ali Muhammad Asad
```