DL Demystified 11 - Under the Hood - Transformer Architecture Explained

January 29, 2025

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

1 Transformer Architecture: A Comprehensive Tutorial

Author: **Dr. Adnan Masood**

References: - *Speech and Language Processing. Daniel Jurafsky & James H. Martin. Copyright © 2024-2025.* - *Attention Is All You Need (Vaswani et al., 2017)* - Various open-source materials, including PyTorch documentation.

---

2 Building an Intuitive Understanding

We will explain the **Transformer Architecture** in **five levels** of detail. This single notebook walks you from a very light conceptual introduction all the way to an in-depth, mathematically comprehensive discussion. These five levels are:

1. (Intuitive, middle-school-level language)
2. (Slightly deeper, but still accessible)
3. (Clear discussion of the main ideas)
4. (Technical details and bridging to the math)
5. (Full details with equations, derivations, and advanced insights)

Each level adds more detail, culminating in code examples and conceptual expansions. By the end, you will be able to both **explain and implement** a Transformer in **PyTorch**, see how it is used to solve real-world problems, and explore Q&A with code.

Let's begin!

2.0.1 Intuitive Explanation

Imagine you're trying to read a sentence in another language. You want to find the best way to turn it into English so you can understand it. A **Transformer** is like a very clever friend that reads sentences in one language, learns what each word really means by looking at all the other words, and then writes it out in another language.

1. We split the sentence into "pieces" (tokens or words) so it can read them one at a time.
2. The **Transformer** has two main parts:
   - An **Encoder** that reads the words.
   - A **Decoder** that writes the translation.
3. Inside, there's **attention**, a way for the Transformer to focus on the words that matter the most. For example, if we have a sentence "The cat sat on the mat because it was tired," the word **"it"** might need to look closely at **"cat"** to know that "it" means "the cat."

By using **attention**, the Transformer can figure out how different words in a sentence connect to each other and then produce an accurate translation (or do other tasks like summarization or text generation). That's the simplest picture!

2.0.2 More Detail on Concepts

The Transformer is a type of deep learning model that excels in understanding and generating language. It's used in things like **ChatGPT**, **Google Translate**, and many other AI-based language tools.

- **Encoder**: Takes a sentence (like "Il fait beau aujourd'hui" in French) and transforms it into a hidden representation (a bunch of numbers that the computer can understand).
- **Decoder**: Uses that hidden representation to generate an English sentence (e.g., "The weather is nice today").

Key ideas: 1. **Self-Attention**: Each word in the sentence can "attend" to every other word. For the word "it" in the example above, the model looks at all other words to see which word "it" should refer to. 2. **Positional Encoding**: Transformers don't process words one after another in a chain (like older RNNs do). Instead, they process them all at once. We still need to tell the model which word is first, second, etc. *Positional encoding* does this by adding specific patterns of numbers to the word representations. 3. **Multi-Head Attention**: Instead of using just one attention map, we use multiple. Imagine multiple sets of eyes looking at the sentence in different ways, each capturing some nuance of meaning.

Overall, the Transformer model can read the entire sentence at once, figure out important relationships, and then generate a whole new sentence in the target language.

2.0.3 The Architecture Explained

Consider a sentence of length N. We break it up into tokens (words/pieces). The Transformer has:

1. **Input Embeddings**: Convert each token into a vector of real numbers (dimension d).
2. **Positional Encodings**: Add a positional encoding to each token embedding to mark its position in the sequence.
3. **Encoder Stack**: N tokens go into the bottom encoder. Each encoder has:
   - A **Multi-Head Self-Attention** sub-layer: the model learns to pay attention to parts of the input.

- A **Feed-Forward Network (FFN)** sub-layer: a small neural network that processes each position independently.
- **Residual** connections and **Layer Normalization** around each sub-layer.

4. **Decoder Stack**: Takes the previously generated tokens as input and does almost the same steps, but also has an **"encoder-decoder attention"** that looks at the output of the **encoder stack**.
5. **Final Linear + Softmax**: The decoder output is turned into a probability distribution over all possible words in the vocabulary.

2.0.4 Self-Attention in a Nutshell

For each word, we create 3 vectors: - **Query (Q)** - **Key (K)** - **Value (V)**

All words produce Q, K, and V. The attention for a given word is computed by:

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

where d_k is the dimension of the key vectors. This gives us a weighted sum of the values, with weights determined by how well the query matches the keys.

2.0.5 Multi-Head Attention

We do that self-attention multiple times in parallel (heads). Each head learns to pay attention to different relationships or patterns in the sentence.

2.0.6 Why Transformers?

- They allow parallel processing of the entire sentence.
- Self-attention can capture long-range dependencies better than older RNN-based approaches.
- Achieves **state-of-the-art** results on many language tasks.

2.0.7 A Bit More Math and Detail

The model dimension is d, typically 512 or higher. Suppose we have an input sequence (tokens) of length N. We embed them into a matrix $X \in \mathbb{R}^{N \times d}$.

We add **positional encodings** P to get the final input to the first encoder:

$$X_0 = X + P, \quad X_0 \in \mathbb{R}^{N \times d}.$$

Encoder:

Each encoder layer ℓ (for $\ell = 1, ..., L$) has:

1. **Layer Normalization** on the input:

$$T_1 = \text{LayerNorm}(X_{\ell-1}).$$

2. **Multi-Head Self-Attention**:

$$\text{MHA}(T_1) = \left(\sum_{h=1}^{H} \text{softmax} \left(\frac{Q^h {K^h}^T}{\sqrt{d_k}} \right) V^h \right) W^O,$$

where each head h has its own projection matrices W_Q^h, W_K^h, W_V^h to get Q, K, V.

3. **Add & Norm** (a residual connection):

$$Z = X_{\ell-1} + \text{MHA}(T_1).$$

4. Another **Layer Normalization**:

$$T_2 = \text{LayerNorm}(Z).$$

5. **Feed-Forward Network** (FFN):

$$\text{FFN}(T_2) = \max(0, T_2 W_1 + b_1) W_2 + b_2.$$

6. **Add & Norm** again:

$$X_\ell = Z + \text{FFN}(T_2).$$

Hence, we get the next layer's input. After L layers, the final output of the encoder stack is X_L.

Decoder has a similar structure, with the modification: - We apply **Masked** Multi-Head Self-Attention (so the decoder cannot peek at future tokens in the output sequence). - We have an extra multi-head attention that uses the **encoder output** as the keys and values, and the decoder hidden states as queries.

2.0.8 Training

We use a **cross-entropy loss** across the entire sequence. For each position in the output sentence, we want the correct word to have the highest probability.

2.0.9 Complexity

- Self-attention is $O(N^2 \times d)$. This is why it's sometimes expensive for large sequences, but faster in practice than very deep recurrent networks for many tasks.

2.0.10 Summary So Far

- The Transformer is a stack of encoders and decoders.
- Each uses (1) multi-head self-attention, (2) feed-forward, (3) residual connections, (4) layer normalization.
- The decoder is also conditioned on the encoder outputs and prevents attending to future output words.

2.0.11 Under-the-Hood Details

1. **Learned Embeddings** E: Typically dimension d. For a vocabulary size $|V|$, $E \in \mathbb{R}^{|V| \times d}$. Input tokens become rows in E.
2. **Positional Encoding**: Original paper uses:

$$PE(pos, 2i) = \sin\left(pos/10^{4i/d}\right), \quad PE(pos, 2i+1) = \cos\left(pos/10^{4i/d}\right).$$

3. **Scaled Dot-Product**:

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

   - $Q = XW^Q$, $K = XW^K$, $V = XW^V$.
   - $Q, K, V \in \mathbb{R}^{N \times d_k}$
   - Then we do multi-head by replicating this with separate parameters for each head.
4. **Masking**: For decoding, we apply a causal mask so the output at time t can only attend to earlier times (1 to $t-1$), not future times.
5. **Final Linear**: We often share embedding weights with the final linear ("unembedding") step (weight tying), ensuring a large matrix doesn't blow up parameters.
6. **Optimization**: Often we use the **Adam** or **Adafactor** optimizer with warm-up learning rates.
7. **Implementation**: In large language models, we sometimes have many modifications, but the central concept remains the same.

That's the fundamental architecture from a deeper perspective.

---

3 Intuition and Illustrative Examples

3.0.1 Example: Short Sentence Translation

Suppose we want to translate from French to English:

1. We have **Input**: "Le chat dort" (meaning "The cat sleeps").
2. The **Encoder** transforms ["Le", "chat", "dort"] into some high-level representation.
3. The **Decoder** uses these representations to generate ["The", "cat", "is", "sleeping"] or something close.

Here is the overall flow: 1. Each French word is turned into an embedding: `[[LE_vec], [CHAT_vec], [DORT_vec]]`. 2. The encoder layers apply **self-attention** so each word's encoding can see the other words. 3. The final encoder output is passed to the decoder. 4. The decoder does masked self-attention on the partial English phrase it has generated, plus attends to the encoder's output. 5. Finally, we get a distribution from the last linear+softmax. The highest probability word is selected as the next word.

3.0.2 A Brief History

- **Before Transformers**: We had RNNs (like LSTM, GRU) for machine translation and sequence tasks. They struggled with long sequences.
- **The Attention Mechanism** in RNN-based seq2seq improved translation quality a lot.
- **Transformers (Vaswani et al. 2017)** removed the recurrence entirely and used only attention (plus feed-forward networks, etc.).
- Has led to large language models like BERT, GPT, T5, etc.

3.0.3 Real-World Problem Solved

- **Machine Translation**: The Transformer provides superior translations while training faster in parallel.
- **Other**: Summarization, question-answering, text classification, code generation, and more. Essentially, Transformers are the backbone of modern LLMs.

---

4 Example Calculations

In this mini example, let's do a small attention calculation by hand.

Assume we have **2 words** in the input (for simplicity), and each embedding is 2-dimensional. Let the words be `W1=[0.5, 1.0]` and `W2=[-0.5, 0.5]`.

1. We define weight matrices for Q, K, V each as 2x2 to keep it small. For instance,

$$W^Q = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, \quad W^K = \begin{pmatrix} 0.5 & 0 \\ 0 & 0.5 \end{pmatrix}, \quad W^V = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

2. For word1: Q1 = W^Q * W1 = [0.5, 1.0], K1 = W^K * W1 = [0.25, 0.5], V1 = W^V * W1 = [0.5, 1.0]. For word2: Q2 = W^Q * W2 = [-0.5, 0.5], K2 = W^K * W2 = [-0.25, 0.25], V2 = W^V * W2 = [-0.5, 0.5].

3. Suppose we compute attention for word1 on both words:

   - Dot product(Q1, K1) = 0.5*0.25 + 1.0*0.5 = 0.125 + 0.5 = 0.625
   - Dot product(Q1, K2) = 0.5*(-0.25) + 1.0*0.25 = -0.125 + 0.25 = 0.125
   - Suppose $d_k = 2$, so we divide by sqrt(2) ~ 1.414. Then:
     - score1 = 0.625/1.414 ~ 0.442
     - score2 = 0.125/1.414 ~ 0.088
   - Softmax: e^{0.442} ~ 1.556, e^{0.088} ~ 1.092 sum=2.648
     - alpha1=1.556/2.648 ~0.587, alpha2=1.092/2.648 ~0.413
   - Weighted sum of values: alpha1*V1 + alpha2*V2 = 0.587*[0.5,1.0] + 0.413*[-0.5,0.5] ~ [0.2935 -0.2065, 0.587 +0.2065] = [0.087, 0.7935]

That final vector [0.087, 0.7935] is the self-attention output for the first word. This is an example of how the final vector might incorporate some of the second word's representation.

---

5 Step by Step Example: Building a Tiny Transformer from Scratch

5.0.1 1. Tokenize Input

- We assume you have some data, e.g. pairs `("je suis étudiant", "i am a student")`. ### 2. Create Embedding Layer
- `nn.Embedding(vocab_size, d_model)` in PyTorch. ### 3. Positional Encoding
- Implement a function that returns a matrix of shape `(max_len, d_model)` with sine/cosine patterns. ### 4. Build an Encoder

1. Self-Attention sub-layer with multi-head.
2. Feed-Forward sub-layer.
3. Residual + LayerNorm. ### 5. Build a Decoder
4. Masked Self-Attention.
5. Encoder-Decoder Attention.
6. Feed-Forward.
7. Residual + LayerNorm. ### 6. Final Linear + Softmax

- For each decoder output position, produce distribution over vocabulary. ### 7. Loss Function (Cross Entropy) ### 8. Optimizer (Adam) ### 9. Train
- For each batch, run forward pass, compute loss, backprop, update parameters. ### 10. Inference
- Greedy or beam search for generating translations.

6 What Problems Does This Solve?

- **Machine Translation**: High-quality, fast translation.
- **Text Summarization**: Summarize long documents.
- **Chatbots / Language Modeling**: Big LLMs are built upon Transformer blocks.
- **Any sequence-based prediction** problem: code completion, music generation, text classification, Q&A, etc.

7 How to Solve a Real-World Problem

1. **Obtain Data**: For example, a parallel corpus of text in two languages.
2. **Preprocess and Tokenize**: Convert each sentence into tokens, build vocabulary.
3. **Create/Load a Transformer**: Implementation from scratch or from libraries like PyTorch.
4. **Training**:
   - Write a training loop.
   - Use Adam optimizer, cross-entropy loss.
   - Possibly use GPUs.
5. **Evaluate**: Compare with known references. For translation, measure BLEU or similar metrics.
6. **Deployment**: Save the trained model. In your application, encode input text, run the decoder to generate output.

8 Points to Ponder (Questions)

1. How do we handle words outside of the vocabulary? (Answer: subword tokenization, or use special tokens.)
2. What about very long sequences? (Answer: Transformers can be extended with special techniques to handle longer contexts, e.g. sparse attention, or efficient variants.)
3. Does the Transformer only do machine translation? (Answer: No, it can do many tasks: summarization, question answering, text generation, etc.)
4. How do we pick the number of heads, layers, etc.? (Answer: Typically a hyperparameter search or by referencing large standard models like GPT, T5, BERT which have known configurations.)
5. How is training speed improved over RNNs? (Answer: Parallelization. The entire sequence is processed at once, as opposed to step-by-step in RNNs.)

9 Answers to the Points to Ponder

1. **Handling Out-of-Vocabulary**: We use subword approaches (e.g., Byte Pair Encoding) that break words into smaller subwords, so almost all tokens are covered.
2. **Very Long Sequences**: We can employ specialized architectures or chunking. Recent research addresses memory overhead (e.g., "Longformer", "Big Bird", etc.).
3. **Beyond MT**: The same architecture is used for chatbots, code generation, summarization, and so on.
4. **Hyperparameters**: Often chosen empirically. Common dimension sizes are 256, 512, or 1024, with multiple heads (e.g., 8 or 16 heads) and anywhere from 6 to 96 layers.
5. **Parallelization**: Self-attention operates across the entire input in parallel, leading to faster training on modern hardware.

10 A Sample Exercise

Here is an **easy PyTorch code sample** to illustrate the main concepts. This example is not a fully optimized production code, but a conceptual illustration. We'll define a small Transformer and run it on synthetic data.

10.1 TODO Items:

1. Modify the dimension sizes (e.g., `d_model` or `nhead`), see how that changes the model parameters.
2. Experiment with the sequence length to see the effect on training time.
3. Try adding a small positional encoding function.

Let's proceed!

[2]:
```
# TODO: Modify code as needed.
import torch
import torch.nn as nn
import torch.optim as optim

# For demonstration: tiny example with minimal code.
```

```
class SimpleTransformer(nn.Module):
    def __init__(self, vocab_size=20, d_model=16, nhead=2,␣
 ↪num_encoder_layers=2, num_decoder_layers=2):
        super(SimpleTransformer, self).__init__()

        self.d_model = d_model
        # Embedding for tokens
        self.embedding = nn.Embedding(vocab_size, d_model)

        # We will use PyTorch's built-in Transformer
        self.transformer = nn.Transformer(
            d_model=d_model,
            nhead=nhead,
            num_encoder_layers=num_encoder_layers,
            num_decoder_layers=num_decoder_layers,
            dim_feedforward=64, # smaller feed-forward for demonstration
            dropout=0.1
        )

        # Final linear layer
        self.fc_out = nn.Linear(d_model, vocab_size)

    def forward(self, src, tgt):
        # src: (S, N) shape [sequence_len, batch_size]
        # tgt: (T, N)
        # Let's embed them
        embedded_src = self.embedding(src) * (self.d_model ** 0.5)
        embedded_tgt = self.embedding(tgt) * (self.d_model ** 0.5)

        # For the built-in Transformer, we want shape [sequence_len,␣
 ↪batch_size, d_model]
        # embedded_src and embedded_tgt are [S, N, d_model]
        embedded_src = embedded_src
        embedded_tgt = embedded_tgt

        # We can create source and target masks (e.g. causal mask) if we want
        # But let's skip that for the simplest demonstration

        # pass through the transformer
        out = self.transformer(
            src=embedded_src,
            tgt=embedded_tgt
        )  # out shape [T, N, d_model]

        # final linear
        logits = self.fc_out(out)  # shape [T, N, vocab_size]
```

```
        return logits

# Let's run a tiny example training loop with synthetic data.
def generate_fake_data(num_samples=64, seq_len=5, vocab_size=20):
    # returns (src, tgt), each shape (seq_len, num_samples)
    src = torch.randint(0, vocab_size, (seq_len, num_samples))
    tgt = torch.randint(0, vocab_size, (seq_len, num_samples))
    return src, tgt

def train_tiny_transformer(model, vocab_size=20, steps=10):
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.Adam(model.parameters(), lr=0.01)

    for step in range(steps):
        src, tgt = generate_fake_data()
        # let's say we want the model to shift the target by one, just to have␣
 ↪a training objective
        # typical of sequence to sequence. We'll feed the entire tgt but␣
 ↪compare with next-step.
        # For the sake of demonstration, we simply do a random shift.
        # In reality, you'd set up your training data properly.
        optimizer.zero_grad()

        logits = model(src, tgt[:-1, :])  # input all but last token, predict␣
 ↪next token
        # logits shape: [T, N, vocab_size], T=seq_len-1
        # we want to compare with the 'gold' next token, which is: tgt[1:,:]

        # flatten them for cross entropy
        # logits => [ (T*N), vocab_size ]
        # target => [ T*N ]
        Tdim, Ndim, _ = logits.shape
        loss = criterion(logits.view(Tdim*Ndim, -1), tgt[1:,:].view(-1))

        loss.backward()
        optimizer.step()

        if (step+1) % 2 == 0:
            print(f"Step {step+1}/{steps}, Loss={loss.item():.4f}")

# Instantiate and train a tiny model
vocab_size = 20
model = SimpleTransformer(vocab_size=vocab_size, d_model=16, nhead=2,␣
 ↪num_encoder_layers=1, num_decoder_layers=1)

train_tiny_transformer(model, vocab_size=vocab_size, steps=10)
```

```
print("\nTraining complete. This is a demonstration of how to set up a PyTorch␣
  ↪Transformer.")
```

```
/home/nightwing/.local/lib/python3.12/site-
packages/torch/nn/modules/transformer.py:379: UserWarning: enable_nested_tensor
is True, but self.use_nested_tensor is False because
encoder_layer.self_attn.batch_first was not True(use batch_first for better
inference performance)
  warnings.warn(
```

```
Step 2/10, Loss=3.1810
Step 4/10, Loss=3.0239
Step 6/10, Loss=3.0087
Step 8/10, Loss=3.0242
Step 10/10, Loss=2.9732

Training complete. This is a demonstration of how to set up a PyTorch
Transformer.
```

11 Glossary

- **Attention**: A method by which a model can dynamically focus on certain parts of its input.
- **Self-Attention**: The attention mechanism where a sequence element attends to other positions in the same sequence.
- **Multi-Head Attention**: Using multiple sets (heads) of attention in parallel, which allows the model to capture different types of relationships.
- **Embedding**: The mapping from discrete tokens (e.g. words) to continuous vectors.
- **Positional Encoding**: A method to inject sequence order information into the model.
- **Feed-Forward Layer**: A fully connected network that applies transformations position-wise.
- **Residual Connection**: A shortcut connection that adds an input to the output of a layer, which helps training deep networks.
- **Layer Normalization**: A method to normalize the input across features in each sample, improving training stability.
- **Encoder**: The part of the model that processes the input tokens.
- **Decoder**: The part of the model that produces the output tokens.
- **Masked Self-Attention**: Self-attention in which future tokens are not visible, ensuring causality.
- **Cross Entropy Loss**: A common loss function for classification, comparing predicted probabilities to the target distribution.

12 Conclusion

We've explored the Transformer architecture in five incremental levels of detail, from a child's perspective to a PhD-level deep dive. This model revolutionized NLP and is widely used in modern large language models.

In the code example, you see a small PyTorch demonstration of how the core ideas can be put into practice. Feel free to extend the code with more layers, heads, embeddings, positional en-

codings, etc. The Transformer is extremely versatile and has led to breakthroughs in translation, summarization, question answering, code generation, and beyond.

Happy Transforming!

[3]:
```
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: 2025-01-29 01:31:05.772474
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
```

[ ]: