An Accessible Explanation of the Groundbreaking Paper: "Attention Is All You Need"

Ela Uçan

23/07/2025

Abstract

This document presents an approachable interpretation of the paper "Attention Is All You Need" by Vaswani *et al.*, outlining the motivations, mechanisms, and architecture of the Transformer model without the complexity of recurrence or convolution. An example PyTorch implementation of the Transformer decoder is also provided.

1 Introduction

"Attention Is All You Need" is a game-changing research paper written by 7 Google employees (Ashish Vaswani et al.) and a research personality from the University of Toronto. This paper serves as a milestone in neural network systems for certain sequence-related tasks, specifically those addressed by the Transformers model. The main compartment of this "transformer" system is the so-called transformers. This is a highly complex article that requires a vast amount of brainstorming and analysis while grasping the knowledge covered inside the paper. This article will be an interpretation and explanation in simpler terms of each chapter discussed. As mentioned, this paper is the solid proof of how transformers were found. Before transformers (equally, before the publishing of this paper), models that were handling sequence data like transactions from, for instance, Turkish to English were built on RNNs and CNNs, recurrent neural networks and convolutional neural networks. These two networks both consist of two parts: an encoder and a decoder. The encoder reads the input sentence (Turkish), and the decoder generates the output sequence (English).

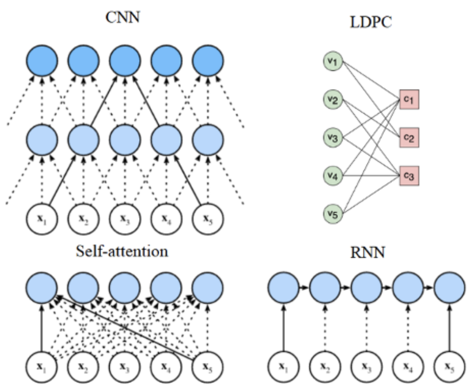

Figure 1: Networks Visualization

2 Background

Both networks use attention mechanisms to focus on the correct parts of the input while generating the output. However, this system is costly, as it takes more effort, and slower because it has step-by-step recurrence. However, authors of this research paper have proposed a new architecture called transformers in which recurrences (RNNs) and convolutions (CNNs) are completely removed and a self-attention & encoder-decoder attention system is used. This method is more optimized as it's faster, parallelizable, and it performs better. Another benefit provided by Transformers is that it works better with large and limited training data.

In the Background section, the authors describe the limitations of previous neural networks and how transformers are able to solve such limitations. On traditional sequence transduction models, at each time step, the model's computation depends completely on the results from the previous time step. This sequence dependency limits the degree to which computations can be parallelized across the sequence. Which is relatively costly. However, with transformers, the process is reduced to a constant number of operations with the usage of self-attention.

3 Self-Attention Mechanism

Self-attention is an attention mechanism that allows the model to focus on several parts of the single sequence when computing results on each compartment of the model. So self-attention builds interconnections between each part of the model. The mechanism does this by computing a weighted sum of all the other tokens in the sequence, and the weights are learned dynamically based on how relevant they are to the current one. How this mechanism works will be explained in further detail. Other mechanisms previously used include end-to-end memory networks and recurrent attention mechanisms. With end-to-end memory networks, the mechanism heavily uses stored information; using attention mechanisms, it retrieves relevant parts of the memory and optimizes the final task without needing to perform sequence-aligned recurrences. The process can be explained in simpler terms as follows:

```
+ Where are Çağla's keys?
- Çağla took the keys
- Çağla went to the kitchen
- Çağla left the kitchen
```

With this mechanism, we find relevant facts and infer the answer: Kitchen

In end-to-end memory networks, recurrent attention mechanisms are used instead of sequence attention mechanisms. However, with Transformer, everything is completely reliant on self-attention.

4 Encoder–Decoder Mechanism

In the transformer, the mechanism adopts the encoder-decoder model for sequence transduction tasks. The encoder takes the input

$$(x_1, x_2, x_3, \ldots, x_n)$$

and then maps it into a series of continuous representations

$$(z_1, z_2, z_3, \ldots, z_n).$$

Finally, the decoder takes these continuous representations and produces the target output

$$(y_1, y_2, y_3, \ldots, y_n).$$

This is done in an auto-regressive manner, meaning at each step, it consumes the previously generated tools while computing the next token using layers of operations that are computed with self-attention networks and no recurrences.

The encoder consists of $N = 6$ identical layers. Each layer contains two sub-layers:

- A multi-head self-attention mechanism
- A position-wise feed-forward network

In between the two sublayers, a layer normalization (and residual connection) is deployed.

In the decoder part, a 3rd sublayer is inserted, which acts as a connection to the multi-head attention from the output of the encoder. Like the encoder part, there exists a residual connection and layer normalization. To ensure appropriate autoregressive behavior during training, the self-attention in the decoder employs masking so that the positionings in producing the output are fixed, meaning that predictions of position k can only attend to positions $< k$.

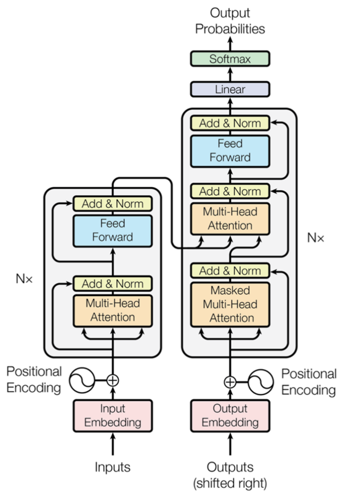

Figure 2: Encoder–Decoder attention sublayer

Attention is a mechanism in neural networks that helps the model to focus on the important parts of the input while generating the output. Attention assigns each token a value, which is then converted to a weight to make the model work properly; therefore, in the end, attention creates a weighted sum of all inputs to produce its context vector (that is composed of Query, Key, and Value.) Transformers use queries, keys, and values in their attention to find the relationship between tokens and combine that information with other layers to find the output. The transformer uses scaled dot product attention. The first step is to compute the dot product between query and key vectors, then scale by the square root of the dimensionality to avoid errors from having large values. This can also cause the softmax function function to have high differences between other vectors and hinder learning.

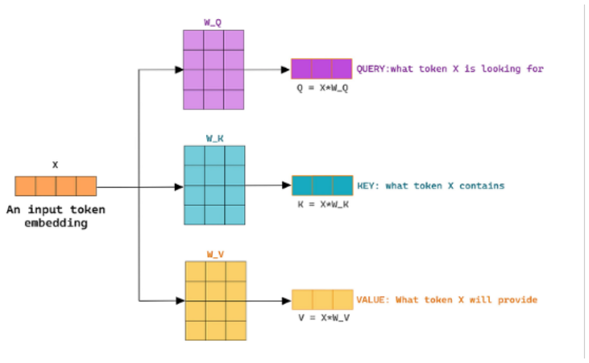
W_Q
QUERY:what token X is looking for
Q = X*W_Q
W_K
x
KEY: what token X contains
An input token embedding
K = X*W_K
W_V
VALUE: What token X will provide
V = X*W_V

Figure 3: Input Token Embedding

5 Multi-Head Attention

The next step in the attention mechanism is multi-head attention. Here, queries, keys, and values are linearly projected h times. In each projected layer, attention is performed in parallel, which is another indicator of how practical self-attention is. With this model, information can be easily shared from different subspaces to different positions. This is a benefit of self-attention. Parallel attention can be mixed and passed through a final linear transformation. This way, the model is able to capture relationships simultaneously, which is something that can't be obtained with a single attention head. In addition to the attention sub-layers, there also exists a fully connected feed-forward network, which can also be interpreted as an encoder-decoder network with a ReLU system (activation) in between. This is applied to each position separately but identically. A feedforward network consists of a linear transformation followed by a ReLU activation then another linear transformation.

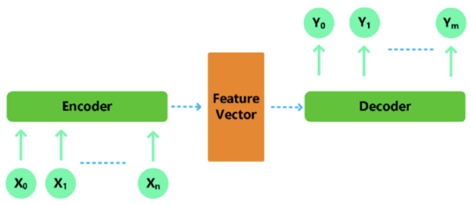

Figure 4: Encoder-Decoder Mechanism

6 Feed-Forward Network

In the first step of the feed-forward network, with the 1st linear transformation, we expand the dimensionality of the token representation from the model's size (for instance, from n = 512 to n = 1024). This gives the model more capacity and allows it to learn more complex patterns. This is followed by ReLU, in which we zero out all negative values of token representations and make the network model nonlinear. The reason why non-linearity is important is that if we were to move on to the second linear transformation without ReLU, we'd derive a completely linear network, which wouldn't make it possible for the model to learn complex patterns. In the second linear transformation, the new dimensioned token representations are reduced back to their original dimension so they can be passed onto the next layer. The reason why this is applied is because this network teaches the model to learn complexity, so setting the dimension back to its original position is crucial because preserving compatibility with residual connections needs to be maintained. To propose a solution to the lack of inherent sequential structure in self-attention, the transformer proposes a positional encoding. Such encoding inject information about the relative or absolute positions of tokens in a sequence. In this article, fixed sinusoidal functions are used.

7 Why Self-Attention?

The fifth section of the article is about the reasoning behind why self-attention is used. 1) Total Computational Complexity per layer. 2) Amount of computation that can be parallelized. 3) Path length between long-range dependencies in the network. The path length refers to the number of sequential steps required for information from one step to reach another. In evaluations, self-attention successfully meets all three criteria. It achieves parallelization across all tokens. In many sequence transduction tasks. Learning long-range dependencies is a hard task; the shorter the paths between any combinations of inputs, the easier it is to learn long dependency tasks. As self-attention layers connect all positions with a constant number of sequentially executed operations, unlike recurrent layers, it's much faster to teach the model about long-range dependencies.

8 PyTorch Implementation

The figure below demonstrates how transformers are built. Each code is clarified briefly by the explanations right next to the codes in columns.

Listing 1: Transformer Decoder Implementation

```
import torch
import torch.nn as nn
import torch.nn.functional as F

class SelfAttention(nn.Module): # heart of the transformer lets every token
    look at every other token in the sequence
    def __init__(self, embed_size, heads): # init initializes the attributes
        that define how the multi-head self-attention works
        super(SelfAttention, self).__init__() # This calls the parent class (nn
            .Module) s  constructor. PyTorch needs this call to properly
            register parameters, submodules, etc.
        self.embed_size = embed_size # Without self., the variables only exist inside the function (__init__). They disappear after the function ends.
        self.heads = heads  # with self It attaches the variable to the object itself
        self.head_dim = embed_size // heads

        assert (
            self.head_dim * heads == embed_size
        ), "Embed size must be divisible by heads"

        # Linear layers to get Queries, Keys, and Values
        self.query = nn.Linear(embed_size, embed_size)
        self.key = nn.Linear(embed_size, embed_size)
        self.value = nn.Linear(embed_size, embed_size)

        # Final fully connected layer
        self.fc_out = nn.Linear(embed_size, embed_size)

    def forward(self, x):
        N, seq_length, embed_size = x.shape

        # Split embedding into heads
        Q = self.query(x).view(N, seq_length, self.heads, self.head_dim) # we change x.shape into this. (2, 8, 10, 32)
        K = self.key(x).view(N, seq_length, self.heads, self.head_dim)
        V = self.value(x).view(N, seq_length, self.heads, self.head_dim)

        # Transpose for multi-head attention: (N, heads, seq_len, head_dim)
        Q = Q.transpose(1, 2) #we do this so that we can easily compute the attention per head
        K = K.transpose(1, 2)
        V = V.transpose(1, 2)

        # Scaled Dot-Product Attention
        energy = torch.matmul(Q, K.transpose(-1, -2)) / (self.embed_size ** 0.5) # scaled dot-product attention
        attention = torch.softmax(energy, dim=-1) # For each tokens Query vector, you want to measure similarity with every other tokens Key vector.
        # we need to turn K into a matrix vector to compute a dot product. so we take the transpose and swap the last two dimensions.
        # which gives us the attention matrix
        # Multiply attention scores with V
        out = torch.matmul(attention, V) # weighted sum of the value vectors for each token

        # Merge heads back together
        out = out.transpose(1, 2).contiguous().view(N, seq_length, embed_size) # flatten heads and head_dim

        # Final linear layer
        out = self.fc_out(out) # until here you glued outputs side by side. but now you need to build the
        # interactions between heads. this means the model can't combine these different "views" effectively

        return out

class TransformerBlock(nn.Module):
    def __init__(self, embed_size, heads, dropout, forward_expansion):
        super(TransformerBlock, self).__init__()
```

```python
        self.attention = SelfAttention(embed_size, heads) # multi-head self-attention mechanism. context aware representation
        # layer normalization normalizing outputs to stabilize and speed up training.
        self.norm1 = nn.LayerNorm(embed_size) # one for after attention
        self.norm2 = nn.LayerNorm(embed_size) # one for after feed-forward
        self.feed_forward = nn.Sequential(  # feed forward network
            nn.Linear(embed_size, forward_expansion * embed_size),
            nn.ReLU(), # expanded the embedding size by f. e. then applyed ReLU activation
            nn.Linear(forward_expansion * embed_size, embed_size) # and finally reduced back to embed size.
        )
        self.dropout = nn.Dropout(dropout) # randomly zeros out elements during training for regulatization

    def forward(self, x):
        # Residual connection + LayerNorm after attention
        attention = self.attention(x) # every tokens learn from all others
        x = self.norm1(attention + x) # we add original input back which is residual connection to normalize for stability.
        x = self.dropout(x)

        # Residual connection + LayerNorm after feed-forward
        forward = self.feed_forward(x) # each tokens embedding is refined individually
        out = self.norm2(forward + x) # we again add input back and normalize
        out = self.dropout(out) # randomly zero-out parts for regularization during training.

        return out

class TransformerDecoder(nn.Module):
    def __init__(
        self,
        vocab_size,
        embed_size,
        num_layers,
        heads,
        dropout,
        forward_expansion,
        max_length
    ):
        super(TransformerDecoder, self).__init__()
        self.embed_size = embed_size
        self.word_embedding = nn.Embedding(vocab_size, embed_size)
        self.position_embedding = nn.Embedding(max_length, embed_size)

        self.layers = nn.ModuleList([ # This creates a list of TransformerBlock layers. and it stores them in a ModuleList.
            TransformerBlock(embed_size, heads, dropout, forward_expansion)
            for _ in range(num_layers)
        ])

        self.fc_out = nn.Linear(embed_size, vocab_size) # This is a fully connected (linear) layer that maps:
        # From: embed_size (like 512)     To: vocab_size (like 30,000)
        # After the stack of TransformerBlocks, each token is represented as a vector of size embed_size. But we want to predict a word.
        # So we need to turn that vector into a probability distribution over all vocabulary words.
        self.dropout = nn.Dropout(dropout)

    def forward(self, x):
        N, seq_length = x.shape
        positions = torch.arange(0, seq_length).unsqueeze(0).expand(N, seq_length)
        # in positions we are creating position IDs for every token in every sequence.
        out = self.dropout(self.word_embedding(x) + self.position_embedding(positions))

        for layer in self.layers: # self.layers is a list of stacked TransformerBlocks.
            out = layer(out)  # For each TransformerBlock, you pass the current out through it:
        # vectors now contain more context about the whole sequence.

        out = self.fc_out(out)
        return out
# Hyperparameters
vocab_size = 10000    # size of vocabulary (number of unique tokens)
embed_size = 256      # size of embedding vector
num_layers = 6        # number of Transformer blocks
heads = 8             # number of attention heads
dropout = 0.1         # dropout probability
forward_expansion = 4 # expand hidden size in feed-forward
max_length = 100      # max length of input sequence

# Model
model = TransformerDecoder(
    vocab_size,
    embed_size,
    num_layers,
    heads,
    dropout,
    forward_expansion,
    max_length
) # builds a stack of 6 transformerblocks.

# Dummy input (batch=2, sequence length=10)
x = torch.randint(0, vocab_size, (2, 10)) # creates a random tensor of shape 2,10.
# each value is a random integer from 0 to 9999
out = model(x) # this runs the forward method of the model.

print(out.shape)  # (2, 10, vocab_size)
```

9 Conclusion

In conclusion, the Transformer architecture, explained by self-attention mechanisms, represents a very significant advancement in sequence modeling; it is offering efficiency, parallelization, and enhanced performance on various tasks. The elimination of recurrent and convolutional dependencies marks a recognizable shift in neural network design, facilitating faster training and inference times. Additionally, the flexibility and modularity of Transformers have opened up new possibilities for a broad range of applications, from language translation and text summarization to computer vision and beyond. The success of the Transformer model has been an inspiration to further research and development in neural networks, leading to the emergence of large-scale pre-trained models like BERT, GPT, and others. These models have reshaped how we approach machine learning problems, setting a new benchmark for performance and versatility across different tasks. The foundational principles established in the "Attention Is All You Need" article continue to spread innovations, affirming the transformative impact of this groundbreaking research.

References

[1] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, L., Polosukhin, I. (2017). *Attention Is All You Need*. In Proceedings of NeurIPS.

[2] Wu, H., et al. (2021). Not All Attention Is All You Need. `arXiv:2104.04692`.

[3] Gheini, M., et al. (2021). Cross-Attention Is All You Need: Adapting Pretrained Transformers for Machine Translation. `arXiv:2104.08771`.

[4] Salehinejad, Hojjat, et al. “Recent Advances in Recurrent Neural Networks.” arXiv.Org, 22 Feb. 2018, arxiv.org/abs/1801.01078.