Advanced deep learning project

Report on the practical part about transformers(TP5)

Patrick Costa[1]

[1]Université Côte d'Azur, E-mail: patrick.costa@etu.univ-cotedazur.fr

Contents

1 Introduction

The Transformer, introduced by Vaswani et al. in 2017[13], has revolutionized the field of deep learning, significantly impacting various domains with its innovative approach to processing sequential data. Originally conceived for Natural language processing, as exemplified by its application in groundbreaking models like BERT[2] and GPT[9], the Transformer's utility extends well beyond, influencing a range of other disciplines.

In Computer Vision , the Transformer's architecture has been adapted to address complex tasks traditionally dominated by convolutional networks. This is evident in works such as Parmar et al. (2018)[8], who applied the Transformer to image super-resolution, Carion et al. (2020)[1] in their development of an end-to-end object detection system, and Dosovitskiy et al. (2020)[4], who utilized it for image classification in the Vision Transformer model.

The versatility of the Transformer also extends to the realm of audio processing. Dong et al. (2018)[3] demonstrated its effectiveness in speech recognition tasks, Huang et al.(2018)[6] explored its applications in music and sound processing. These adaptations highlight the Transformer's capability in handling and interpreting complex audio sequences.

Furthermore, the Transformer model has found novel applications in interdisciplinary fields such as chemistry, where Schwaller et al. (2019)[11] utilized it for predicting chemical reactions, and in life sciences, as demonstrated by Rives et al. (2021)[10] in their exploration of protein structure prediction.

This report aims to provide a comprehensive review of the Transformer model, analyzing its core components both from theoretical and practical standpoints. Each section will include an overview and a detailed Python code implementation, offering clear insights into the mechanisms behind key elements like the scaled dot product, multi-head attention, encoder block, and the Transformer predictor. The report will also discuss practical applications of the Transformer, particularly in solving sequence reversal problems and predicting sequences as a whole, underscoring its adaptability and effectiveness in diverse machine learning tasks.

2 Scaled dot product

2.1 Overview

The first important element of the Trasforemr architecture that will be analyzed is the scaled dot product mechanism1.

The scaled dot product attention mechanism is a key component in the architecture of Transformer models. It plays a central role in their capability to efficiently and effectively process sequential data, enabling the dynamic determination of the degree of 'attention' that one element within a sequence should assign to others during processing.

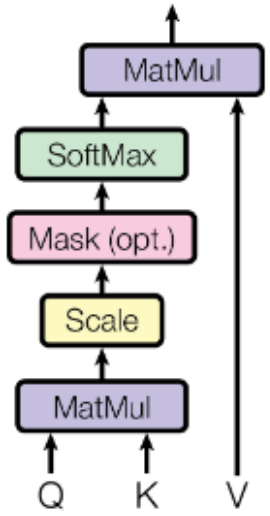

Figure 1: Scaled dot-product attention

What makes this feature essential is how it calculates attention scores, helping to pull out vital context from the sequence

At its core, the scaled dot product attention mechanism relies on three principal sets of vectors: queries (Q), keys (K), and values (V).

These vectors come from the same data but undergo unique transformations, giving them their special roles in the attention process.The main goal here is to figure out how much focus should be given to different parts of the input data for each sequence element.

The computation starts with the calculation of dot products between queries and keys, followed by a scaling operation. This scaling step involves dividing the dot product by a scaling factor (d_k), a critical step aimed at mitigating the potential emergence of excessively large values during the training process. Subsequently, a softmax function is applied to the scaled dot products, yielding a probability distribution that quantifies the degree of attention or relevance each key holds concerning a given query.

Finally, this process leads to a weighted sum of values, where the attention probabilities tell us exactly how to weigh each part.

Mathematically, the scaled dot product attention is expressed as:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right) \cdot V \tag{1}$$

Here, Q represents the query matrix with dimensions $T \times d_k$, where T signifies the sequence length, and d_k represents the hidden dimensionality specific to queries and keys. K stands for the key matrix, sharing identical dimensions with Q, while V designates the value matrix, characterized by dimensions $T \times d_v$, with d_v representing the hidden dimensionality associated with values. The scaling factor $\sqrt{d_k}$ assumes an important role as it ensures the appropriate variance in attention values(because the previous result is divided for the dimension of the embedding).

The matrix multiplication QK^T calculates dot products across all conceivable pairs of queries and keys, resulting in a $T \times T$ matrix. Each row of this matrix embodies the attention logits corresponding to a specific element within the sequence in relation to all other elements. Subsequently, a softmax function is applied to these logits, followed by multiplication with the value matrix V, ultimately yielding a weighted mean. These attention probabilities effectively determine the weighting of each associated value, encapsulating the contextual relationships among words or sub-words within the sequence.

The introduction of the scaling factor $1/\sqrt{d_k}$ It's not a random choice; it's actually crucial, especially in the early stages of model training, .because it helps to tame extremely high values. Such occurrences could otherwise push the softmax function into regions characterized by vanishing gradients, thus mitigating the issue of 'vanishing gradients,' which is essential for effective gradient-based optimization during training.

In practical applications, the scaled dot product attention mechanism has demonstrated remarkable efficiency and effectiveness in capturing intricate long-range dependencies within sequential data. Its exceptional ability to simultaneously consider all components of a sequence has led to significant advancements in deep learning, particularly within the domain of natural language processing. Consequently, the scaled dot product attention mechanism stands as a foundational element within modern deep learning models, providing the means to efficiently process and understand sequential data, making it an invaluable tool for a myriad of applications in artificial intelligence and machine learning.

2.2 Implementation

Following the theoretical overview of the scaled dot product attention mechanism, we now turn our attention to its practical implementation. This section will provide a detailed walkthrough of the Python code essential for realizing this mechanism. Emphasizing clarity and efficiency, the implementation is designed to encapsulate the core principles.

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

def scaled_dot_product(q, k, v, mask=None):
    d_k = q.size()[-1]  # Get the dimension of the key vectors
```

```
    # Compute attention logits
    attn_logits = torch.matmul(q, k.transpose(-2, -1))  # Step 1: Dot product of Q and K^T
    attn_logits /= math.sqrt(d_k)  # Step 2: Scale the logits by sqrt(d_k)

    # Apply mask
    if mask is not None:
        attn_logits = attn_logits.masked_fill(mask == 0, -1e14)
        # Step 3: Apply mask by replacing masked values with a large negative number

    # Softmax to get attention weights
    attention = F.softmax(attn_logits, dim=-1)  # Step 4: Apply softmax over the last dimension

    # Weight values accordingly
    output_values = torch.matmul(attention, v)  # Step 5: Multiply attention weights with V

    return output_values, attention
```

This code snippet demonstrates the step-by-step implementation of the scaled dot product attention mechanism, including the calculation of attention logits, scaling, masking, softmax, and weighting of values.

2.3 Explanation

In this section, we delve into a comprehensive explanation of the code that implements the scaled dot product attention mechanism. Each line of code will be meticulously dissected to elucidate its function and significance within the broader context of the attention process. This breakdown aims to provide a deeper understanding of how each component contributes to the overall mechanism, thus offering insights into the intricacies of this powerful tool in Transformer models.

`d_k = q.size()[-1]`

This line retrieves the dimensionality of the key vectors (d_k). It extracts the size of the last dimension of the query matrix q, representing the feature or embedding size in the model. d_k is then used as a scaling factor in the attention computation to prevent large values in the attention scores, which could destabilize the softmax operation.

`attn_logits = torch.matmul(q, k.transpose(-2, -1))`

This calculates the raw attention scores between the queries and keys. The matrix multiplication between q and the transposed key matrix k results in unnormalized attention scores, represented by $attn\_logits$. Each element measures the attention level of a query element towards each key element in the sequence.

`attn_logits /= math.sqrt(d_k)`

Here, the attention logits are scaled by the square root of d_k. This step stabilizes the gradients during training by preventing extremely small gradients in the softmax function, thereby enhancing the training efficiency and stability.

```
if mask is not None:
```

Checks if a mask tensor is provided, which is used to prevent certain positions in the input (like padding tokens) from influencing the result, allowing selective focus of the attention mechanism.

```
attn_logits.masked_fill(mask == 0, -1e14)
```

Applies the mask to the attention logits. If the mask is 0 at a position, that position in *attn_logits* is replaced with a large negative number, effectively nullifying it in the softmax step.

```
attention = F.softmax(attn_logits, dim=-1)
```

Applies softmax to the scaled logits, transforming them into a probability distribution. The resulting *attention* matrix determines how much each value in the sequence contributes to the output.

```
output_values = torch.matmul(attention, v)
```

Computes the weighted sum of the values based on the attention weights, resulting in the final output of the attention step. This encapsulates aggregated information from the input sequence based on relevancy determined by the attention mechanism.

3 Multihead attention

3.1 Overview

The second mechanism that will be analyzed in this report is The Multi-Head Attention mechanism, it is an indispensable part in Transformer architectures, represents a significant advancement in deep learning. Its design allows for simultaneous processing of sequences through multiple distinct attention heads(the single-head self-attention is powerful but has limitations in modeling complex dependencies), enriching the model's ability to discern diverse attributes of the input. A visualization of how it works can be observed in the Fig.2

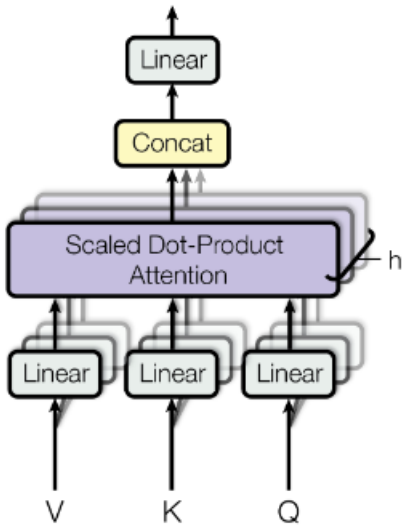

Figure 2: Multi-head attention

At its foundation, Multi-Head Attention builds upon the principles of the scaled dot product attention mechanism, extending its capabilities. It facilitates attention to various aspects of sequence elements, overcoming the constraints imposed by single weighted averages. This is realized by dividing the query, key, and value matrices into h separate sets, named sub-queries, sub-keys, and sub-values.Independently, each set is processed through the scaled dot product attention, leading to a combined output that is subsequently integrated through a final weight matrix. The mathematical formulation of this process can be expressed as:

$$\begin{aligned} \text{Multihead}(Q, K, V) &= \text{Concat}(\text{head}_1, \ldots, \text{head}_h)W^O, \\ \text{where head}_i &= \text{Attention}(QW_i^Q, KW_i^K, VW_i^V). \end{aligned}$$

Crucial to this framework are the learnable parameters $W_{1\ldots h}^Q \in R^{D\times d_k}$, $W_{1\ldots h}^K \in R^{D\times d_k}$, $W_{1\ldots h}^V \in R^{D\times d_v}$, and $W^O \in R^{h\cdot d_k \times d_{out}}$, with D denoting input dimensionality.

A unique attribute of Multi-Head Attention is its permutation equivariance in relation to the inputs, this means that if you change the input, also the output will change but the relationship between the elements will remain constant.

Altering the input sequence order leads to a corresponding shift in the output, effectively perceiving the input as a set rather than a fixed sequence. This trait significantly broadens the Transformer's applicability and effectiveness.

In neural network implementations, Multi-Head Attention commonly involves designating the current feature map $X \in R^{B\times T\times d_{\text{model}}}$ as Q, K, and V. Transformations of X via weight matrices W^Q, W^K, and W^V generate appropriate feature vectors, facilitating the integration of this module into diverse neural network designs.

Moreover, Multi-Head Attention permits the model to jointly attend to distinct representation subspaces at varied positions. With multiple heads, each dimensionally scaled down—the model

achieves a comprehensive understanding of the input. This approach maintains computational efficiency comparable to single-head attention, while significantly enhancing interpretative depth and flexibility.

3.2 Implementation

In this section, the focus shifts from theory to practice as we delve into the practical implementation of the Multihead Attention mechanism. The objective is to provide a detailed code walkthrough that faithfully captures the foundational principles of Multihead Attention.

```
class MultiheadAttention(nn.Module):
    def __init__(self, input_dim, embed_dim, num_heads):
        super().__init__()

        # Ensure that the embedding dimension is divisible by the number of heads
        assert embed_dim % num_heads == 0, "Embedding dimension
        must be divisible by the number of heads."

        # Define class attributes
        self.embed_dim = embed_dim  # Dimension of concatenated heads
        self.num_heads = num_heads  # Number of attention heads
        self.head_dim = embed_dim // num_heads  # Dimension of each attention head

        # Linear projections for query, key, and value vectors
        self.o_proj = nn.Linear(embed_dim, embed_dim)
        self.qkv_proj = nn.Linear(input_dim, embed_dim * 3)

        # Initialize model parameters
        self._reset_parameters()

    def _reset_parameters(self):
        # Initialize model weights and biases as per the original Transformer
        nn.init.xavier_uniform_(self.qkv_proj.weight)
        self.qkv_proj.bias.data.fill_(0)
        nn.init.xavier_uniform_(self.o_proj.weight)
        self.o_proj.bias.data.fill_(0)

    def forward(self, x, mask=None, return_attention=False):
        batch_dim, seq_length, input_dim = x.shape

        # Compute linear projection for qkv and separate heads
        qkv = self.qkv_proj(x)
        qkv = qkv.reshape(batch_dim, seq_length, self.num_heads, 3 * self.head_dim)
        qkv = qkv.permute(0, 2, 1, 3)
        q, k, v = torch.chunk(qkv, 3, dim=-1)

        # Apply Dot Product Attention to qkv
```

```
attention_values, attention = scaled_dot_product(q, k, v)

# Concatenate heads to [Batch, SeqLen, Embed Dim]
attention_values = attention_values.permute(0, 2, 1, 3)
attention_values = attention_values.reshape(batch_dim, seq_length, self.embed_dim)

# Output projection
o = self.o_proj(attention_values)

if return_attention:
    return o, attention  # Return attention scores if requested
else:
    return o  # Return the output of multihead attention
```

The code emonstrates the step-by-step implementation of the multihead attention mechanism, from the definition of its characteristics(i.e. possibility to choose heads) until the final output of attention values.

3.3 Explanation

In this section, we present a meticulous and comprehensive exposition of the code that embodies the Multihead Attention mechanism. Our objective is to provide a rigorous analysis of each line of code, elucidating its precise function and its overarching significance within the broader context of the attention mechanism. This in-depth scrutiny aims to cultivate a profound comprehension of how each constituent element harmoniously contributes to the overall mechanism, thereby offering profound insights into the intricacies of this potent tool within the Transformer framework.

```
class MultiheadAttention(nn.Module):
```

This line declares 'MultiheadAttention' as a subclass of 'nn.Module', integrating it into the PyTorch neural network framework. This subclassing is essential for leveraging PyTorch's built-in functionalities .

```
def __init__(self, input_dim, embed_dim, num_heads):
```

The `MultiheadAttention` class is initialized with its primary configuration parameters. `input_dim` specifies the dimension of each input feature, defining the input data's dimensionality. `embed_dim` determines the dimension for embeddings in each attention head, shaping the representation in a lower-dimensional space. `num_heads` represents the total number of attention heads in the mechanism.

```
super().__init__()
```

This line calls the constructor of the superclass ('nn.Module'). This initialization is for correctly setting up the PyTorch module, ensuring that all underlying mechanisms, like parameter registration and module hierarchy, are properly configured.

```
assert embed_dim % num_heads == 0
```

This line ensures that the embedding dimension (`embed_dim`) is divisible by the number of heads (`num_heads`). This check is for split the embedding dimension across all attention heads, ensuring that each head receives an equal portion of the embedding space.

```
self.embed_dim = embed_dim
```

Assigns the provided embedding dimension to the instance variable `self.embed_dim`. Storing this as an instance variable allows the model to use the embedding dimension size in various calculations and operations throughout the class, maintaining consistency in the model's architecture.

```
self.num_heads = num_heads
```

Sets the number of attention heads to the instance variable `self.num_heads`. This variable is used for the parallel processing of the attention mechanism, deciding how many separate attention calculations the model performs at each step.

```
self.head_dim = embed_dim // num_heads
```

Calculates and stores the dimension of each attention head by dividing the embedding dimension by the number of heads. This ensures that each attention head works with a segment of the embedding vector of equal size, facilitating equal distribution of the model's capacity across all heads.

```
self.o_proj = nn.Linear(embed_dim, embed_dim)
```

Initializes a linear transformation (‘nn.Linear‘) for the output projection. This layer is responsible for projecting the concatenated outputs of all attention heads back to the original embedding dimension.

```
self.qkv_proj = nn.Linear(input_dim, embed_dim * 3)
```

Another linear layer is initialized to project the input data into queries, keys, and values (collectively abbreviated as qkv). The layer's output dimension is three times the embedding dimension to accommodate separate linear projections for queries, keys, and values within a single matrix multiplication.

```
nn.init.xavier_uniform_(self.qkv_proj.weight)
```

This line applies the Xavier uniform initialization to the weights of the qkv projection layer. Xavier initialization is used in deep learning models as it helps in maintaining a consistent scale of gradients, reducing the risk of vanishing or exploding gradients.

```
self.qkv_proj.bias.data.fill_(0)
```

Sets the bias values in the qkv projection layer to zero. Initializing the biases to zero is a common practice as it starts the training process without any preconceived bias, allowing the learning process to adjust these values appropriately. This helps in starting the training from a neutral point, avoiding any initial skew in the decision-making process of the model.

```
nn.init.xavier_uniform_(self.o_proj.weight)
```

Applies the Xavier uniform initialization to the weights of the output projection layer. This choice of initialization ensures that the weights of this layer are set to values that help in maintaining the scale of the gradients across the network, promoting stable and efficient learning, especially in the early stages of training.

```
self.o_proj.bias.data.fill_(0)
```

This line initializes the biases of the output projection layer to zero. This initialization strategy is used to start with a non-prejudiced model, allowing the training process to learn and adjust these biases based on the data and the task at hand.

```
batch_dim, seq_length, input_dim = x.shape
```

This line extracts the dimensions of the input tensor `x`. `batch_dim` represents the size of each batch, `seq_length` denotes the length of the sequence in each batch, and `input_dim` indicates the feature size of each element in the sequence.

```
qkv = self.qkv_proj(x)
```

Applies the qkv linear projection to the input tensor ‘x‘. This projection is designed to generate query, key, and value representations from the input data. It’s a key operation in the attention mechanism, enabling the model to prepare the necessary components for calculating attention scores.

```
qkv = qkv.reshape(batch_dim, seq_length, self.num_heads, 3 * self.head_dim)
```

Reshapes the qkv tensor to prepare it for splitting into separate query, key, and value tensors. This reshaping is essential for enabling parallel processing across the multiple heads in the attention mechanism, ensuring that each head receives an appropriately sized segment of the qkv tensor.

```
qkv = qkv.permute(0, 2, 1, 3)
```

Rearranges the dimensions of the qkv tensor to align them correctly for the subsequent operations. This permutation is vital for ensuring that the tensor is structured correctly for splitting into queries, keys, and values in a way that aligns with the requirements of the multi-head attention mechanism.

```
q, k, v = torch.chunk(qkv, 3, dim=-1)
```

Splits the qkv tensor into three separate tensors: queries (q), keys (k), and values (v). This operation is central to the attention mechanism, as it provides the distinct components required to compute the attention scores. Each of these components plays a unique role in the attention calculation.

```
attention_values, attention = scaled_dot_product(q, k, v)
```

Computes the scaled dot-product attention using the queries, keys, and values. This function is the heart of the attention mechanism, determining the degree of focus that should be placed on different parts of the input sequence. The attention values indicate how much each part of the sequence should contribute to the final output.

```
attention_values = attention_values.permute(0, 2, 1, 3)
```

Permutes the attention values tensor to bring it back to the original order. This reordering is necessary post-computation to align the attention values correctly for further processing, ensuring that the model's output maintains the proper sequence structure.

```
attention_values = attention_values.reshape(batch_dim, seq_length, self.embed_dim)
```

Reshapes the attention values tensor to concatenate the outputs from all attention heads. This operation combines the separately computed attention values from each head into a unified representation, crucial for integrating the diverse perspectives captured by different heads.

```
o = self.o_proj(attention_values)
```

This line applies the output projection to the concatenated attention values, synthesizing information from each attention head. The `self.o_proj` operation combines and transforms the outputs from all attention heads to produce the final output tensor.

"'latex

```
if return_attention:
    return o, attention
else:
    return o
```

These conditional statements determine the function's return value. If `return_attention` is set to true, the function returns both the output tensor `o` and the attention weights `attention`. This can be useful for visualizing the attention mechanism's behavior and analyzing where the model focuses its attention. If `return_attention` is false, the function returns only the output tensor `o`, which is often the case when using the model for standard predictions or in downstream tasks.

4 Encoder block

4.1 Overview

The Transformer Encoder Block stands as a foundational element in the realm of neural network architectures, especially in the field of natural language processing. Originally developedfor machine translation tasks, the Transformer model exhibits a unique encoder decoder structure. Within this structure, the encoder serves a critical role, processing the input sentence in the source language and transforming it into a rich, attention-based representation, which the decoder then leverages to generate the translated sentence.

The encoder itself can be observed in the left part of the Fig3,it is composed by a series of identical layers, with each layer encapsulating two key sub-layers: a multi-head self-attention mechanism , that has been described in 3.1 and a position-wise fully connected feed-forward network. The multi-head attention mechanism is adept at capturing contextual relationships within the input sequence by processing different parts in parallel. The feed-forward network, applied independently to each position, introduces depth and complexity, enabling the encoder to perform intricate transformations on the sequence data.

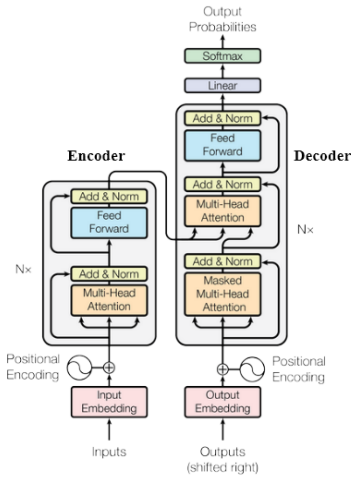

Figure 3: The Transformer model architecture.

A pivotal innovation in the Transformer's design is the use of residual connections[5] around each sub-layer, coupled with layer normalization. These residual connections help avoid the vanishing gradient problem common in training deep networks and maintain the integrity of the input through successive layers. Layer normalization, following each residual connection, ensures consistent training dynamics and offers a form of regularization.

In addition to the architectural components, the Transformer encoder incorporates input embeddings and positional encodings. Input embeddings convert input tokens into high-dimensional vectors, capturing semantic information. Positional encodings,which have the same dimension as the embeddings , are then added to these embeddings to provide the model with sequence order information. The Transformer utilizes a specific pattern for positional encodings, based on sine and cosine functions of different frequencies, as formulated by Vaswani et al.:

$$PE_{(pos,i)} = \begin{cases} \sin\left(\frac{pos}{10000^{i/d_{\text{model}}}}\right) & \text{if} \quad i \bmod 2 = 0 \\ \cos\left(\frac{pos}{10000^{(i-1)/d_{\text{model}}}}\right) & \text{otherwise} \end{cases} \tag{2}$$

These positional encodings, vital for tasks where sequence order is key, are added to the original input features. The distinctive use of sine and cosine functions allows the model to potentially generalize to longer sequences than seen during training. Moreover, this encoding facilitates the model's ability to understand and utilize the relative positions of the tokens in the sequence.

Each Transformer layer concludes with strategically placed dropout layers[12] in both the MLP and after the attention outputs for regularization purposes. This comprehensive architecture, with its self-attention mechanisms, feed-forward networks, makes the Transformer encoder as a powerful tool for handling complex sequential data in diverse applications.

4.2 Implementation

The Python code below implements in detail the encoder block, following the main elements that should be included as described in the original paper.

```
class EncoderBlock(nn.Module):
    # Initialize the Encoder Block with specified parameters
    def __init__(self, input_dim, num_heads, dim_feedforward, dropout=0.0):
        super().__init__() # Initialize the base nn.Module class
        # Set up model parameters
        self.input_dim = input_dim
        self.num_heads = num_heads
        self.dim_feedforward = dim_feedforward
        self.dropout = dropout

        # Multi-head Attention layer
        self.self_attn = MultiheadAttention(input_dim, input_dim, num_heads)

        # Two-layer MLP with dropout
        self.mlp = nn.Sequential(
            nn.Linear(input_dim, input_dim * 2), # First linear layer
            nn.ReLU(),                           # ReLU activation
            nn.Dropout(dropout),                 # Dropout layer
            nn.Linear(input_dim * 2, input_dim) # Second linear layer
        )

        # Layer Norm and Dropout
        self.norm = nn.Sequential(
            nn.LayerNorm(input_dim), # Layer normalization
            nn.Dropout(dropout)      # Dropout
        )

    # Forward pass of the Encoder Block
    def forward(self, x, mask=None):
        attn = self.self_attn(x) # Apply self-attention
        x = self.norm(x + attn)  # Add and normalize

        # Apply MLP and normalize
        x = self.norm(x + self.mlp(x))

        return x # Return the output of the Encoder Block
```

4.3 Explanation

In this section, we present a comprehensive exposition of the code that implement the encoder block, explaining in detail each line of code.

```
class EncoderBlock(nn.Module):
```

This line defines `EncoderBlock` as a subclass of `nn.Module`. This inheritance is crucial for integrating with PyTorch features.

```
def __init__(self, input_dim, num_heads, dim_feedforward, dropout=0.0):
```

The constructor initializes the `EncoderBlock`. `input_dim` specifies the size of input features, `num_heads` defines the number of attention heads, `dim_feedforward` sets the size of the hidden layer in the feedforward network, and `dropout` determines the dropout rate.

```
super().__init__()
```

This line calls the initializer of the superclass `nn.Module`, setting up the necessary configurations for the neural network module within PyTorch.

```
self.self_attn = MultiheadAttention(input_dim, input_dim, num_heads)
```

Here, a multi-head attention mechanism is created. `self.self_attn` is assigned an instance of `MultiheadAttention`, configured with the input dimension and the number of attention heads. This mechanism allows the model to focus on different parts of the input sequence.

```
self.mlp = nn.Sequential(
    nn.Linear(input_dim, input_dim*2),
    nn.ReLU(),
    nn.Dropout(dropout),
    nn.Linear(2*input_dim, input_dim)
)
```

This sequence defines the multi-layer perceptron (MLP) within the encoder. It includes a linear layer to expand the input's dimension, a ReLU activation for non-linearity, a dropout layer for regularization, and another linear layer to project the dimension back to the original size. This MLP adds depth and complexity to the model's processing capability.

```
self.norm = nn.Sequential(
    nn.LayerNorm(input_dim),
    nn.Dropout(dropout)
)
```

Layer normalization (`nn.LayerNorm`) and dropout are setup in sequence (`self.norm`). Layer normalization helps stabilize the learning process, while dropout aids in reducing overfitting by randomly nullifying certain neuron outputs.

```
def forward(self, x, mask=None):
```

Defines the forward pass for `EncoderBlock`. It takes an input tensor `x` and an attention mask `mask`.

```
attn = self.self_attn(x)
x = self.norm(x + attn)
```

Applies self-attention to the input `x`, adds the output of the attention layer to the original input (creating a residual connection), and then normalizes it using the defined `self.norm`. This operation allows the model to integrate information from different parts of the input sequence.

```
x = self.norm(x + self.mlp(x))
```

Processes the output from the attention layer through the MLP, then adds it back to the original input before the MLP and applies normalization. This step enables the model to perform complex transformations on the input data.

```
return x
```

Finally, the processed tensor `x` is returned. This tensor is now enriched with information processed through both the attention mechanism and the feedforward network.

5 Transformer predictor

5.1 Overview

The Transformer Encoder Classifier, designed for classification tasks, enhances the Transformer architecture by integrating additional networks for effective sequential data processing.

The architecture commences with an input network, mapping input dimensions to model dimensions. This network typically consists of linear layers with ReLU activation and dropout for regularization, mathematically represented as:

$$\text{InputNet}(x) = \text{Dropout}(\text{ReLU}(\text{Linear}(x))) \tag{3}$$

Each encoder layer contains a multi-head self-attention mechanism and a feed-forward network. These layers process the input through attention mechanisms and transform it via the feed-forward network, described as:

$$\text{EncoderLayer}(x) = \text{LayerNorm}(x + \text{Sublayer}(x)) \tag{4}$$

The final component is the output network, which transforms the encoder's output into class predictions. This transformation is achieved through a linear layer, mapping the encoded sequence representations to the output classes:

$$\text{OutputNet}(x) = \text{Linear}(x) \tag{5}$$

This comprehensive structure enables the Transformer encoder classifier to handle classification tasks effectively, leveraging the Transformer's sequential data processing capabilities with additional components designed for classification.

5.2 Implementation

In this section the tranformer predictor will be implemented following the ideas discussed in 5.1.

```
class TransformerPredictor(nn.Module):
    # Initialize the Transformer Predictor model
    def __init__(self, input_dim, model_dim, num_classes,
    num_heads, num_layers, dropout=0.0, input_dropout=0.0):
        super().__init__() # Initialize nn.Module
        # Define model parameters
        self.input_dim = input_dim
        self.model_dim = model_dim
        self.num_classes = num_classes
        self.num_heads = num_heads
        self.num_layers = num_layers
        self.dropout = dropout
        self.input_dropout = input_dropout

        # Input layer transformation
        self.input_net = nn.Sequential(
            nn.Linear(input_dim, model_dim), # Linear layer
            nn.Dropout(input_dropout) # Dropout for regularization
        )

        # Positional encoding for input sequences
        self.positional_encoding = PositionalEncoding(model_dim, max_len=16)

        # Transformer Encoder
        self.transformer = TransformerEncoder(num_layers,
        input_dim=model_dim, dim_feedforward=model_dim*2,
        num_heads=num_heads, dropout=dropout)

        # Output layer for class prediction
        self.output_net = nn.Linear(model_dim, num_classes)

    # Forward pass
    def forward(self, x, mask=None, add_positional_encoding=True):
        x = self.input_net(x) # Apply input network
        if add_positional_encoding:
            x = self.positional_encoding(x) # Add positional encoding
        x = self.transformer(x, mask=mask) # Apply Transformer encoder
        x = self.output_net(x) # Apply output network
        return x # Return the final output

    @torch.no_grad() # Disable gradient calculations
    def get_attention_maps(self, x, mask=None, add_positional_encoding=True):
        x = self.input_net(x) # Apply input network
```

```
if add_positional_encoding:
    x = self.positional_encoding(x) # Add positional encoding
attention_maps = self.transformer.get_attention_maps(x, mask=mask) # Get attention maps
return attention_maps # Return attention maps
```

5.3 Explanation

In this section, we offer a thorough explanation of the code that implements the Transformer Predictor class, providing a detailed breakdown of each line of code.

```
class TransformerPredictor(nn.Module):
```

Defines `TransformerPredictor` as a subclass of `nn.Module`, enabling it to utilize PyTorch's neural network functionalities.

```
def __init__(self, input_dim, model_dim,
num_classes, num_heads, num_layers,
dropout=0.0, input_dropout=0.0):
```

Initializes `TransformerPredictor` with parameters for dimensionality, number of classes, heads, layers, and dropout rates. These parameters define the architecture and functionality of the Transformer model.

```
super().__init__()
```

Calls the initializer of the superclass `nn.Module` to set up the module within PyTorch's framework.

```
self.input_net = nn.Sequential(nn.Linear(input_dim, model_dim), nn.Dropout(input_dropout))
```

Creates an input network consisting of a linear transformation from `input_dim` to `model_dim` and applies dropout for regularization.

```
self.positional_encoding = PositionalEncoding(model_dim, max_len=16)
```

Initializes positional encoding to infuse sequence order information into the model, using `model_dim` for dimensionality.

```
self.transformer = TransformerEncoder(num_layers, input_dim=model_dim,

dim_feedforward=model_dim*2, num_heads=num_heads, dropout=dropout)
```

Sets up the Transformer encoder with specified layers, model dimensions, feedforward dimensions, number of heads, and dropout. This encoder is the core component for processing sequences.

```
self.output_net = nn.Linear(model_dim, num_classes)
```

Defines a linear output network to map the Transformer's output to the number of classes, forming the final prediction layer.

```
def forward(self, x, mask=None, add_positional_encoding=True):
```

Specifies the forward pass, handling input `x`, an optional mask, and a TRUE for adding positional encoding.

```
x = self.input_net(x)
```

Processes `x` through the input network, preparing it for the Transformer.

```
if add_positional_encoding:
    x = self.positional_encoding(x)
```

Conditionally adds positional encoding to `x` if add$_positional_encodingisTRUE$,

```
x = self.transformer(x, mask=mask)
```

Feeds `x` into the Transformer encoder, applying attention mechanisms and sequence processing.

```
x = self.output_net(x)
```

Passes the output from the Transformer through the output network, generating class predictions for each sequence element.

```
return x
```

Returns the final predictions, completing the forward pass of the model.

6 The sequence reversal problem

6.1 Definition of the problem

The sequence reversal problem is a pivotal task in the context of deep learning, specifically within sequence processing. This problem entails the inversion of a given sequence of elements, such that the output sequence is a mirror reflection of the input sequence. It's a foundational challenge that tests the efficacy of neural network models in handling and manipulating sequential data.

Consider this example of an input sequence:

$$\text{Input:} \quad [2, 5, 3, 7, 1] \tag{6}$$

For the sequence reversal task, the expected output is:

$$\text{Output:} \quad [1, 7, 3, 5, 2] \tag{7}$$

In traditional programming, reversing a sequence, like an array or list, is relatively simple. But in deep learning, particularly with neural networks such as Recurrent Neural Networks (RNNs) or Transformers, the sequence reversal task becomes a test of a model's proficiency in handling long-term dependencies within sequences.

The difficulty of this task arises because the model needs to correctly connect elements in the sequence that could be far apart from each other. For instance, the first element in the reversed

sequence corresponds to the last in the original. RNNs often face challenges in capturing these long-range dependencies due to issues like vanishing or exploding gradients. In contrast, Transformers are designed with attention mechanisms that adeptly handle such dependencies.

The sequence reversal problem is especially pertinent when exploring Transformer architectures. Unlike RNNs, which process data in a sequential order and might lose context over extended periods, Transformers maintainins a comprehensive context of the entire sequence. This global view is vital for sequence reversal, where a complete understanding of the sequence is necessary to reverse it accurately.

Engaging a Transformer model in this task involves not only evaluating its fundamental sequence processing capabilities but also assessing its proficiency in detecting intricate relationships among elements positioned far apart within a sequence. I will test this concept in section 6.3 with a very simple sequence to gain a better understanding of the underlying intuition.

6.2 Methodology

To Tackle this sequence reversal problem , model was configured with a single encoder block and a single head in the Multi-Head Attention mechanism, a setup chosen due to the simplicity of the sequence reversal task. Each number in the input sequence was represented as a one-hot vector, ensuring a distinct and non-ordinal representation of the sequence elements.

This approach was crucial as it prevented the model from assuming any linear relationship between numerically close categories, such as between 0 and 1, versus 0 and 9. In the implementation, a one-hot vector coupled with an additional linear layer effectively served the same purpose as an embedding layer, mapping the sparse one-hot vector to a denser vector representation. The training regimen encompassed a comprehensive set of steps including training, validation, and testing phases. During training, the input sequence was processed through the model's encoder, and the model was tasked with predicting the output for each input element. The standard Cross-Entropy loss function was employed for this task.

An AdamW [7] optimizer was utilized to adjust and update the model parameters.

The model was subjected to a series of rigorous training and validation cycles, with a keen focus on loss and accuracy. The entire process was characterized by its iterative nature, spanning multiple epochs, with continuous monitoring and feedback on the model's accuracy and loss metrics during both training and validation phases.

6.3 Results

Upon implementing the Transformer model for the sequence reversal problem, the outcomes were highly encouraging. The model achieved a 100% accuracy rate in correctly reversing the input sequences.

This remarkable performance underscores the Transformer's robust capability in sequence processing tasks, even when dealing with potentially complex patterns and long-range dependencies.

The success of the model in this task also brought attention to the critical role of the attention mechanism within the Transformer architecture.
To gain deeper insights into how the model was processing the input sequence, attention maps was generated and analyzed. As it is represented in Fig.4 , this map provided a visual representation of the attention weights assigned by the model to different parts of the sequence during each processing step.

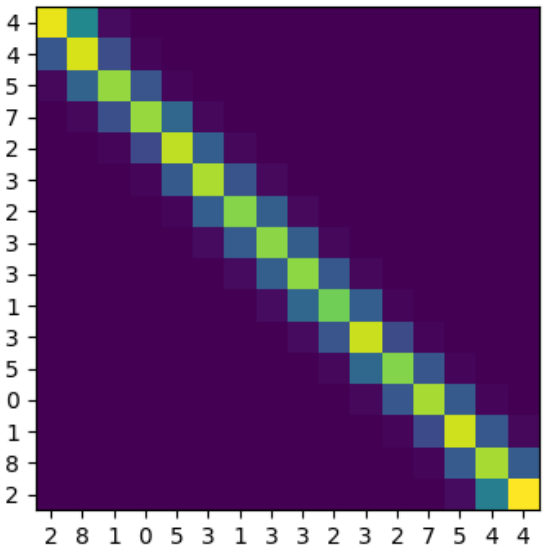

Figure 4: Attention map for the head 1

In these attention maps, each cell corresponds to the weight of attention given from one element of the sequence to another. The intensity of the color in each cell indicates the strength of this attention, with brighter areas signifying higher weights. Such visualization is instrumental in deciphering the inner workings of the Transformer model, revealing how it assigns varying levels of importance to different parts of the input sequence throughout the processing stages.

It is possible to see that the model has acquired the ability to focus on the token that occupies the reversed position within its input sequence. As a result, it performs the desired task as intended. Notably, the model also allocates some of its attention to values located near the reversed index. This behavior arises from the fact that the model doesn't require precise, strict attention to solve this problem but can effectively utilize an approximate and somewhat noisy attention map. The proximity of the indices is a consequence of the intentional design of the positional encoding, aligning with our original intentions for its use.

These results not only validated the Transformer model's effectiveness in the sequence reversal problem but also provided valuable insights into its attention-driven approach to handling sequential data. The ability of the Transformer to maintain high accuracy across different sequence lengths and complexities further exemplifies its adaptability and power in sequence-to-sequence tasks.

7 Predicting about the sequence as a whole

7.1 Definition of the problem

Predicting a sequence as a whole involves analyzing the entire input sequence to generate a single outcome, offering a distinct approach from the element-wise processing typical in sequence-to-sequence tasks. This method is particularly valuable in scenarios where the aggregate properties of a sequence hold more significance than individual elements.

For example, in the field of finance, a model might predict future market trends based on a sequence of historical stock prices, focusing on the overall pattern rather than individual price points. In healthcare, analyzing a sequence of a patient's vital signs could lead to a holistic health assessment or an early warning about potential medical conditions.

In environmental science, models might predict weather patterns or climate trends based on sequences of atmospheric data, where the comprehensive interpretation of data points is crucial for accurate forecasting. In the realm of text analysis, beyond sentiment analysis, this approach can be applied to detect topics or themes in documents or to categorize text based on stylistic features.

Each of these applications requires a deep understanding of how individual elements in a sequence collectively contribute to a broader pattern or trend. The model's task is to distill this complex array of data into a coherent, singular prediction or classification, using the full context of the sequence for decision-making. This comprehensive form of analysis is key to advancements in areas that rely on interpreting extensive sequential data to derive meaningful conclusions.

7.2 Methodology

In transitioning from sequence-to-sequence tasks to whole sequence prediction, the Transformer-Predictor's architecture undergoes pivotal alterations to include the entirety of a sequence in a singular predictive outcome. Central to this modification is the reconfiguration of the output layer. Unlike the original design, where predictions were made for each sequence element, the adapted model integrates the entire sequence information, typically via a global pooling layer or through the output associated with a special token representative of the sequence in its entirety(i.e. the [CLS] token employed in models like BERT).

The forward pass method within the TransformerPredictor is accordingly revised to reflect this shift in focus. Post processing by the transformer encoder, the model applies the chosen aggregation mechanism, culminating in a vector that embodies the entire sequence. This vector is then directed through a fully connected layer, tailored to the specific requirements of the task at hand.

This methodology retains the positional encoding component of the TransformerPredictor. The positional encoding continues to play a crucial role in imparting information about the sequence order, a vital aspect in understanding the sequence as a whole. However, the interpretation and utilization of the attention mechanism undergo a nuanced shift. While the mechanism's core function remains the same, its role is reinterpreted to focus more on discerning the global context of the sequence, rather than on the relationships between individual sequence elements.

The training objective too is redefined to align with the goals of whole-sequence prediction. Corresponding adjustments are made to the loss functions and training strategies to ensure they are compatible with the the task.

In applying these modifications, the TransformerPredictor evolves to solve tasks that demand not just an understanding but a prediction based on the entirety of a sequence, and this shows its applicability beyond mere element-wise sequence processing. This expanded capability is particularly advantageous in fields where the collective interpretation of sequential data is paramount for drawing meaningful and actionable insights.

8 Conclusion

This report has meticulously explored the Transformer architecture, discussing and implementing each component in detail. It has also engaged in a comprehensive discussion of two general tasks: reversing a sequence and predicting a sequence as a whole. These discussions serve as a proof to the Transformer's significant role in the field of deep learning. The model's ability to accurately reverse sequences highlights its proficiency in understanding and manipulating sequential dependencies. Furthermore, its adeptness in predicting entire sequences underscores its capacity for data interpretation and decision-making. Overall, the Transformer model stands out as a robust and versatile tool in deep learning, demonstrating its capability to tackle a wide range of challenges with remarkable efficiency and effectiveness

Bibliography

[1] Nicolas Carion, Francisco Massa, Gabriel Synnaeve, Nicolas Usunier, Alexander Kirillov, and Sergey Zagoruyko. End-to-end object detection with transformers. In *European conference on computer vision*, pages 213–229. Springer, 2020.

[2] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[3] Linhao Dong, Shuang Xu, and Bo Xu. Speech-transformer: a no-recurrence sequence-to-sequence model for speech recognition. In *2018 IEEE international conference on acoustics, speech and signal processing (ICASSP)*, pages 5884–5888. IEEE, 2018.

[4] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, et al. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929*, 2020.

[5] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[6] Cheng-Zhi Anna Huang, Ashish Vaswani, Jakob Uszkoreit, Noam Shazeer, Ian Simon, Curtis Hawthorne, Andrew M Dai, Matthew D Hoffman, Monica Dinculescu, and Douglas Eck. Music transformer. *arXiv preprint arXiv:1809.04281*, 2018.

[7] Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101*, 2017.

[8] Niki Parmar, Ashish Vaswani, Jakob Uszkoreit, Lukasz Kaiser, Noam Shazeer, Alexander Ku, and Dustin Tran. Image transformer. In *International conference on machine learning*, pages 4055–4064. PMLR, 2018.

[9] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

[10] Alexander Rives, Joshua Meier, Tom Sercu, Siddharth Goyal, Zeming Lin, Jason Liu, Demi Guo, Myle Ott, C Lawrence Zitnick, Jerry Ma, et al. Biological structure and function emerge from scaling unsupervised learning to 250 million protein sequences. *Proceedings of the National Academy of Sciences*, 118(15):e2016239118, 2021.

[11] Philippe Schwaller, Teodoro Laino, Théophile Gaudin, Peter Bolgar, Christopher A Hunter, Costas Bekas, and Alpha A Lee. Molecular transformer: a model for uncertainty-calibrated chemical reaction prediction. *ACS central science*, 5(9):1572–1583, 2019.

[12] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958, 2014.

[13] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.