MSc. Data Science & AI

Class: Advanced Deep Learning
Professors: Frederic Precioso, Rémy Sun, Pierre-Alexandre Mattei

Student: Aglind Reka
Mail: aglind.reka@etu.univ-cotedazur.fr

Report

Contents

1. Introduction

1.1 Context

Transformers, introduced in the 2017 paper "Attention is All You Need" [1], have become a dominant architecture in natural language processing and machine learning. They leverage attention mechanisms for parallelized and scalable training, distinguishing them from traditional sequence models. The self-attention mechanism allows elements in a sequence to prioritize information, enabling the model to capture long-range dependencies efficiently. Transformers commonly employ an encoder-decoder architecture, incorporating positional encoding to handle the sequence order. Their impact extends to various tasks, making them a foundational technology in contemporary deep learning applications.

2. Transformers

2.1 Scaled Dot Product

2.1.1 Code

Code extraction 1: Scaled Dot-Product Attention

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

def scaled_dot_product(q, k, v, mask=None):
    d_k = q.size()[-1]

    ######################
    ### YOUR CODE HERE! ###
    ######################

    # Compute attn_logits
    attn_logits = torch.matmul(q, k.transpose(-2, -1))
    attn_logits /= math.sqrt(d_k)

    # Apply mask if not None
    if mask is not None:
        attn_logits = attn_logits.masked_fill(mask == 0, -
            1e14)

    # Pass through softmax
    attention = F.softmax(attn_logits, dim=-1)

    # Weight values accordingly
    output_values = torch.matmul(attention, v)

```

```python
    #######################
    ###        END        ###
    #######################

    return output_values , attention
```

The purpose of the function is to implement the scaled dot-product attention mechanism, which is commonly used in the context of attention mechanisms in neural network architectures, especially in transformer models.

The input parameters that the function takes:

- `q`: Query vectors.
- `k`: Key vectors.
- `v`: Value vectors.
- `mask`: An optional mask to control which positions should be ignored in the attention computation, by default `None`.

2.1.2 The steps that the function follows (Explanation)

Step 1: Compute Attention Logits

Use matrix multiplication to compute the raw attention scores between the query (`q`) and key (`k`) vectors. This step measures the compatibility between each query and key.

Step 2: Scale Logits

Divide the attention logits by the square root of the dimension of the key vectors (`d_k`). This scaling helps stabilize the training process and prevents the gradients from becoming too small during backpropagation.

Step 3: Apply Mask (Optional)

If a mask is provided, use it to mask certain positions in the attention logits. This step is useful, for example, in sequence-to-sequence tasks where certain positions in the input sequence should be ignored during attention computation.

Step 4: Softmax

Apply the softmax function to obtain normalized attention weights. Softmax converts the attention logits into a probability distribution, ensuring that the weights sum to 1.

Step 5: Weight Values

Use the computed attention weights to weight the values (`v`). This step combines information from the values based on the attention weights. Finally, the function returns the weighted values (`output_values`) and the attention weights (`attention`).

The function is a crucial part of the attention mechanism in transformers, allowing the model to focus on different parts of the input sequence during processing. The attention weights indicate how much attention each element in the input sequence should receive. This mechanism is used to capture long-range dependencies and improve the model's ability to handle sequential data.

2.2 Multi-Head Attention

The scaled dot product attention enables a network to focus on different aspects of a sequence. To address the need for attending to multiple aspects of a sequence element, attention mechanisms are extended to multiple heads. This involves transforming query, key, and value matrices into sub-queries, sub-keys, and sub-values. These are independently processed through scaled dot product attention, and the resulting heads are concatenated and combined using a final weight matrix. This process enhances the network's ability to capture diverse patterns in the input sequence.

2.2.1 Code

Code extraction 2: Multihead Attention Class

```python
import torch
import torch.nn as nn

class MultiheadAttention(nn.Module):
    def __init__(self, input_dim, embed_dim, num_heads):
        super().__init__()
        assert embed_dim % num_heads == 0, "Embedding dimension must be 0 modulo number of heads."

        self.embed_dim = embed_dim  # dimension of concatenated heads
        self.num_heads = num_heads
        self.head_dim = embed_dim // num_heads

        ######################
        ### YOUR CODE HERE! ###
        ######################

        # Create linear layers for both qkv and output
        # TIP: Stack all weight matrices 1...h together for efficiency
        self.o_proj = nn.Linear(embed_dim, embed_dim)
        self.qkv_proj = nn.Linear(input_dim, embed_dim * 3)

        ######################
        ###       END       ###
        ######################

        self._reset_parameters()

```

```python
    def _reset_parameters(self):
        # Original Transformer initialization, see PyTorch documentation
        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):

        ######################
        ### YOUR CODE HERE! ###
        ######################

        batch_dim, seq_length, input_dim = x.shape

        # Compute linear projection for qkv and separate heads
        # QKV: [Batch, Head, SeqLen, Dims]
        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
        else:
            return o

        ######################
        ###      END       ###
        ######################
```

2.2.2 Class Explanation

The class `MultiheadAttention` is a neural network module representing a multihead attention mechanism.

Initialization

The constructor initializes the module's parameters, including the embedding dimension, the number of attention heads, and the input dimension. It ensures that the embedding dimension is divisible evenly by the number of heads.

Linear Layers

Two linear layers are created – `o_proj` for output projection and `qkv_proj` for the linear projection of queries, keys, and values.

Parameter Initialization

The `_reset_parameters` method initializes the linear layer weights and biases using Xavier initialization. Xavier uniform initialization for the weight parameters of two linear layers (self.qkv_proj and self.o_proj). Additionally, it sets the bias parameters of these layers to zero. This choice aims to establish a neutral starting point for learning and simplifies the initial learning dynamics.

Forward Method

The `forward` method executes the forward pass of the attention mechanism.

Linear Projection and Head Separation

The input sequence x undergoes linear projection (self.qkv_proj) to obtain the query (q), key (k), and value (v) matrices. These are then reshaped and permuted to separate the heads.

Dot Product Attention

The scaled dot product attention is applied to the separated query, key, and value matrices, producing attention values and attention weights.

Concatenation and Output Projection

The attention values from different heads are concatenated and reshaped to the desired output dimensions. The concatenated attention values are linearly projected using self.o_proj to produce the final output.

Return Statement

The method returns either the output values or both the output values and attention weights based on the `return_attention` parameter.

Overall Purpose

The class provides a modular implementation of a multihead attention module, following the design principles of the original Transformer architecture. Its primary utility lies in integration within neural network models, especially those dealing with sequential data. The code prioritizes modularity for flexibility and efficiency in handling attention computations across multiple heads. One crucial property of multi-head attention is its permutation-equivariance with respect to inputs. This implies that when swapping two elements in the input sequence (e.g., X1-X2, ignoring the batch dimension), the output remains the same except for the interchange of elements 1 and

2. Consequently, multi-head attention processes the input as a set of elements rather than a fixed sequence.

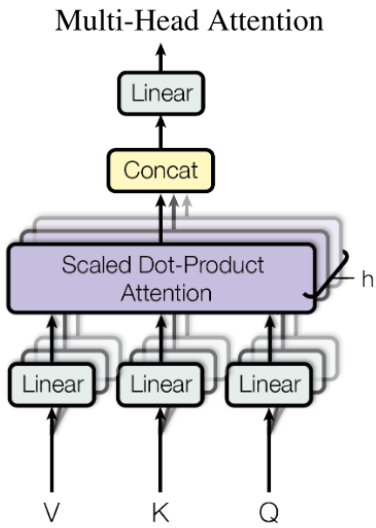

Figure 1: Multi-Head Attention

2.3 Encoder Block

The encoder is composed of N identical blocks applied sequentially. Given input x, it undergoes a Multi-Head Attention block, as implemented previously. The output is then combined with the original input using a residual connection, and a subsequent Layer Normalization is applied to the sum. In summary, it computes LayerNorm(x + Multihead(x, x, x)), with x as the input to the attention layer (Q, K, and V).

2.3.1 Code

Code extraction 3: EncoderBlock Class

```python
import torch.nn as nn

class EncoderBlock(nn.Module):
    def __init__(self, input_dim, num_heads, dim_feedforward, dropout=0.0):
        """
        Args:
            input_dim: Dimensionality of the input
            num_heads: Number of heads to use in the attention block
            dim_feedforward: Dimensionality of the hidden layer in the MLP
            dropout: Dropout probability to use in the dropout layers
        """
        super().__init__()

        ######################
        ### YOUR CODE HERE! ###
        ######################

        # Create Attention layer
        self.self_attn = MultiheadAttention(input_dim, input_dim, num_heads)

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

        # Layers to apply in between the main layers (Layer Norm and Dropout)
        self.norm = nn.Sequential(
            nn.LayerNorm(input_dim),
            nn.Dropout(dropout)
        )
```

```python

        ######################
        ###       END      ###
        ######################

    def forward(self, x, mask=None):
        # Compute Attention part
        attn = self.self_attn(x)
        x = self.norm(x + attn)

        # Compute MLP part
        x = self.norm(x + self.mlp(x))

        return x
```

2.3.2 Explanation

Class Definition

The `EncoderBlock` class is a neural network module representing an encoder block in a transformer architecture.

Initialization

The constructor initializes the encoder block's parameters, including the input dimension, the number of attention heads, the dimensionality of the hidden layer in the MLP (multi-layer perceptron), and an optional dropout probability.

Attention Layer

An attention layer (`self_attn`) is created using the `MultiheadAttention` class with the specified input dimension, acting as a self-attention mechanism.

MLP (Multi-Layer Perceptron)

A two-layer MLP (`mlp`) is constructed with an initial linear layer, ReLU activation, dropout, and a final linear layer.

Layer Normalization and Dropout

Two sequential layers (`norm`) are defined, consisting of layer normalization followed by dropout. These layers are applied in between the main layers (attention and MLP) to improve training stability.

Forward Method

The `forward` method executes the forward pass of the encoder block.

Attention Part

The attention mechanism is computed by applying the self-attention layer to the input. The result is normalized and added to the original input.

MLP Part

The MLP is applied to the normalized input, and the result is again normalized and added to the original input.

Return Statement

The method returns the final output of the encoder block.

Overall Purpose

The `EncoderBlock` class serves as a container for a transformer encoder block, amalgamating a self-attention mechanism and a multi-layer perceptron. It is engineered to capture intricate patterns and dependencies within input sequences, augmenting the overall capabilities of the transformer model. The code prioritizes modularity and flexibility, emphasizing seamless integration of attention and MLP components.

2.4 Transformer Predictor

The provided template outlines a classifier built upon the Transformer encoder for sequence prediction. In addition to the Transformer architecture, it incorporates:

- A compact input network, responsible for mapping input dimensions to model dimensions.
- The inclusion of positional encoding.
- An output network tasked with transforming output encodings into predictions.

It's essential to note that the output network operates on a 3D tensor (batch samples, sequence length, model dimension) and produces a 2D tensor (batch samples, sequence length). Each output value signifies the prediction for the corresponding reversed number.

2.4.1 Code

Code extraction 4: TransformerPredictor Class

```
import torch.nn as nn

class TransformerPredictor(nn.Module):
    def __init__(
        self,
        input_dim,
        model_dim,
        num_classes,
        num_heads,
        num_layers,
        dropout=0.0,
        input_dropout=0.0,
    ):
        """
        Args:
            input_dim: Hidden dimensionality of the input
            model_dim: Hidden dimensionality to use inside the
               Transformer
```

```
            num_classes: Number of classes to predict per sequence element
            num_heads: Number of heads to use in the Multi-Head Attention blocks
            num_layers: Number of encoder blocks to use.
            lr: Learning rate in the optimizer
            warmup: Number of warmup steps. Usually between 50 and 500
            max_iters: Number of maximum iterations the model is trained for. This is needed for the CosineWarmup scheduler
            dropout: Dropout to apply inside the model
            input_dropout: Dropout to apply on the input features
        """
        super().__init__()
        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

        #######################
        ### YOUR CODE HERE! ###
        #######################

        # Create a Generic Input Encoder Input dim -> Model dim with input dropout
        self.input_net = nn.Sequential(
            nn.Linear(input_dim, model_dim),
            nn.Dropout(input_dropout)
        )

        # Create positional encoding for sequences
        self.positional_encoding = PositionalEncoding(model_dim,max_len=16)

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

        # Create output classifier per sequence element Model_dim -> num_classes
        self.output_net = nn.Linear(model_dim, num_classes)

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

```
        """
        Args:
            x: Input features of shape [Batch, SeqLen, input_dim]
            mask: Mask to apply on the attention outputs (optional)
            add_positional_encoding: If True, we add the positional encoding to the input.
                                      Might not be desired for some tasks.
        """
        x = self.input_net(x)
        if add_positional_encoding:
            x = self.positional_encoding(x)
        x = self.transformer(x, mask=mask)
        x = self.output_net(x)

        #######################
        ###       END       ###
        #######################

        return x

    @torch.no_grad()
    def get_attention_maps(self, x, mask=None, add_positional_encoding=True):
        """Function for extracting the attention matrices of the whole Transformer for a single batch.

        Input arguments same as the forward pass.
        """
        x = self.input_net(x)
        if add_positional_encoding:
            x = self.positional_encoding(x)
        attention_maps = self.transformer.get_attention_maps(x, mask=mask)
        return attention_maps
```

2.4.2 Explanation

Class Definition

The `TransformerPredictor` class is a neural network module designed for sequence prediction tasks using a transformer architecture.

Initialization

The constructor initializes the predictor's parameters, including the input dimension, model dimension, number of classes to predict per sequence element, number of attention heads, number of encoder blocks, and optional dropout probabilities for the model and input features.

Input Network

A generic input encoder (`input_net`) is created, consisting of a linear layer mapping input dimension to model dimension, followed by dropout on the input features.

Positional Encoding

A positional encoding module is instantiated (`positional_encoding`) to provide positional information to the input sequences.

Transformer Encoder

A transformer encoder (`transformer`) is created with the specified number of layers, input dimension, hidden layer dimension, number of attention heads, and dropout. This component captures complex dependencies in the input sequences.

Output Network

An output classifier (`output_net`) is defined, mapping the model dimension to the number of classes. This layer produces predictions for each sequence element.

Forward Method

The `forward` method executes the forward pass of the predictor.

Input Processing

The input features go through the input encoder (`input_net`) and optionally receive positional encoding.

Transformer Processing

The processed input is fed into the transformer encoder (`transformer`), capturing hierarchical dependencies in the sequence.

Output Generation

The final output is obtained by passing the transformer's output through the output classifier (`output_net`).

Attention Maps Method (get_attention_maps)
get_attention_maps is defined to extract the attention matrices of the entire transformer for a single batch. It applies the input encoder and positional encoding (if specified) to the input and retrieves attention maps using the get_attention_maps method of the transformer.

Overall Purpose

The `TransformerPredictor` class is designed for sequence prediction tasks, employing a transformer architecture to capture both local and global dependencies in input sequences. It prioritizes modularity, enabling users to adjust parameters for diverse tasks. Additionally, the class features attention map extraction, offering insights into the learned attention patterns of the transformer.

3. Questions

3.1 How do we solve the sequence reversal problem at the end of the notebook? Discuss both the method and results.

The sequence reversal problem is tackled through the utilization of a Transformer architecture, employing the following key components:

Dataset Handling

- The `ReverseDataset` class is designed to generate random sequences for training, where labels represent the reversed sequences.

Model Configuration

- The `TransformerPredictor` class is employed, featuring a single encoder block and a single head in the Multi-Head Attention for simplicity.

Training Approach

- Training involves processing sequences through the Transformer encoder.
- Cross-Entropy loss is applied, treating each number as a one-hot vector.
- The AdamW optimizer is utilized.

Training Outcome

- Achieves 100% accuracy on both training and validation sets, as well as on the test set.

Attention Visualization

- Attention maps are extracted post-training, unveiling the model's focus during the sequence reversal task.
- Attention patterns are visually represented using a dedicated plotting function.

Results and Discussion

- The Transformer model effectively solves the sequence reversal task, demonstrating perfect accuracy.
- Attention maps show the model's ability to concentrate on the token at the flipped index, aligning with the task requirements.
- The model adapts to positional encoding patterns, showcasing its flexibility in handling sequential dependencies.

In summary, this solution highlights the Transformer's proficiency in sequential tasks, emphasizing accuracy and providing insight into the model's attention mechanisms through attention map visualization.

3.2 What would be different if we wanted to predict something about the sequence as a whole?

To shift the model's focus from predicting individual sequence elements to making predictions about the sequence as a whole, several adjustments are required:

1. **Output Layer Modification:** The output layer needs to be adapted to consider the entire sequence rather than individual elements. This adjustment involves using appropriate layers or mechanisms, such as pooling layers, recurrent structures, or global attention mechanisms, to aggregate information across the entire sequence.

2. **Choice of Loss Function:** Depending on the nature of the sequence-level prediction task, the loss function must align with the desired output. Common choices include Binary Cross-Entropy, Categorical Cross-Entropy, or Mean Squared Error for tasks involving the sequence as a cohesive entity.

3. **Model Architecture Adjustments:** The Transformer architecture may undergo modifications to better capture global dependencies within the sequence. This could entail changes in the arrangement of Transformer layers or the inclusion of additional components tailored for capturing holistic sequence characteristics.

4. **Input Representation Reevaluation:** The input representation should be scrutinized to ensure it encapsulates comprehensive features relevant to the entire sequence, aligning with the specific requirements of the prediction task.

5. **Enhanced Sequence Encoding:** For tasks focused on the complete sequence, a more sophisticated encoding scheme may be necessary. This could involve the incorporation of intricate encoding mechanisms to better represent the holistic aspects of the sequence.

Bibliography

[1] 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.