POE Práctica 2

Sonia Castro Paniello
Eduard Ramon Aliaga

March 2023

Create a comparative table of the studied models with respect to the single-layer transformer baseline, including loss, accuracy, training time, number of parameters, and hyperparameters differences with at least 4 new models or hyperparameters. Include the modified source code and a simple schematic drawing of the model.

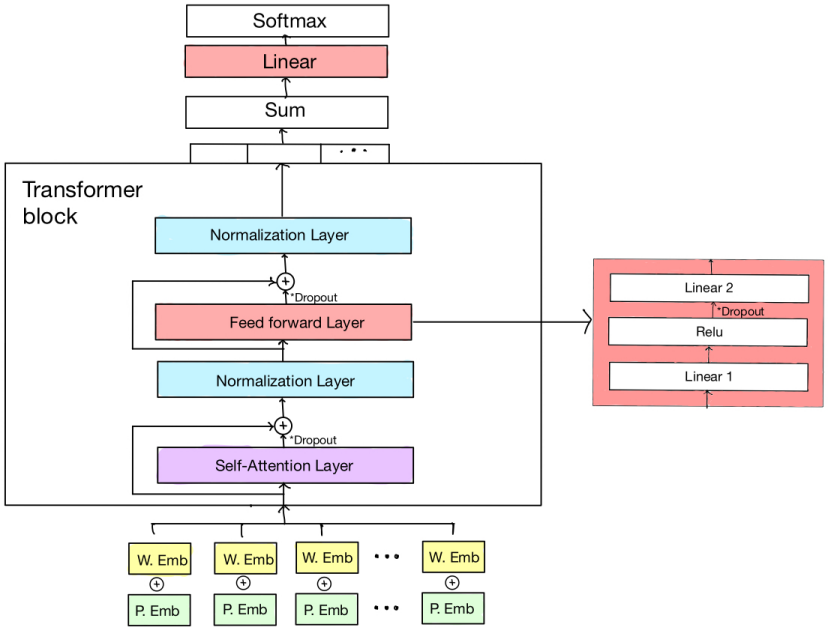

Figure 1: Model architecture.

This is an overview of the model. First, the word embeddings and position embeddings are combined and used as input for the transformer block. The input to the transformer block has a size of (B, W, E), where B is the batch size, W is the number of words in the context, and E is the embedding dimension. The transformer block consists of a self-attention layer, a normalization layer, a feedforward layer, and another normalization layer. The output of the transformer block has the same size as the input.

To obtain a single output that represents the word being predicted, we sum the embeddings of the words in each batch, resulting in a dimension of (B, E). We then use a linear layer to convert this dimension to (B, V), and finally, we use softmax to determine the most probable target word in the vocabulary.

The feedforward layer is composed of a linear layer, ReLU activation function, and another linear layer. This model has already been implemented.

For our experiment, we implemented a multilayer transformer using the Transformer Layer code, allowing us to add the desired number of transformer blocks.

```
class FourLayerTransformer(nn.Module):
    def __init__(self, d_model, dim_feedforward=512, dropout=0.1,
        activation="relu"):
        super().__init__()
        self.layer1 = TransformerLayer(d_model, dim_feedforward, dropout,
            activation)
        self.layer2 = TransformerLayer(d_model, dim_feedforward, dropout,
            activation)
        self.layer3 = TransformerLayer(d_model, dim_feedforward, dropout,
            activation)
        self.layer4 = TransformerLayer(d_model, dim_feedforward, dropout,
            activation)

    def forward(self, src):
        src = self.layer1(src)
        src = self.layer2(src)
        src = self.layer3(src)
        src = self.layer4(src)
        return src
```

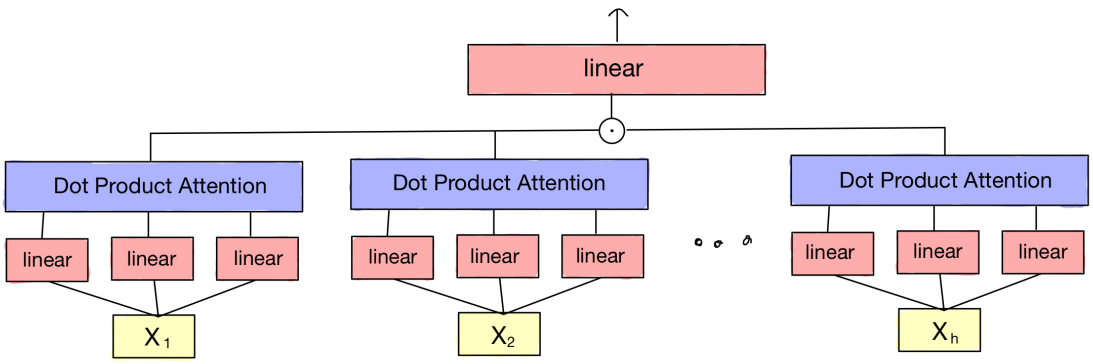

Figure 2: Multihead Attention

We incorporated multi-head attention into our model, as depicted in Figure 2. Initially, we followed this outline but each head was given the entire original input. Subsequently, we implemented a variation in which each head was given a distinct slice of the input. For both implementations, we employed the Scaled Dot Product Attention function, which had already been implemented in the provided code.

In the first implementation, we created a new function called AttentionHead, which featured three linear projections (for the keys, queries, and values) from the embedding dimension to the head dimension, allowing each head to use all of the input X. The attention function was also invoked in this function. We also implemented the SelfAttention multihead by using a nn.ModuleList to iterate over the AttentionHeads. Since we projected the last dimension of the embedding size to the head dimension in AttentionHead, we concatenated the outputs of the different heads to restore the last dimension to the embedding size before applying the final linear layer.

Later, we discovered that each head should only use a portion of the input. However, we included the code and results obtained from the initial implementation in our report, as they may be of interest.

```
def attention(query, key, value, mask=None, dropout=None):
    "Compute 'Scaled Dot Product Attention'"
    d_k = query.size(-1)
    scores = torch.matmul(query, key.transpose(-2, -1)) \
             / math.sqrt(d_k)
    if mask is not None:
        scores = scores.masked_fill(mask == 0, float('-Inf'))
    p_attn = F.softmax(scores, dim = -1)
    if dropout is not None:
        p_attn = dropout(p_attn)
    return torch.matmul(p_attn, value), p_attn

class AttentionHead(nn.Module):
    def __init__(self, embed_dim, head_dim, bias=True):
        super().__init__()
        self.head_dim = head_dim
        self.k_proj = nn.Linear(embed_dim, self.head_dim, bias)
        self.v_proj = nn.Linear(embed_dim,self.head_dim, bias)
        self.q_proj = nn.Linear(embed_dim, self.head_dim, bias)
        self.reset_parameters()

    def reset_parameters(self):
        # Empirically observed the convergence to be much better with the scaled
            initialization
        nn.init.xavier_uniform_(self.k_proj.weight, gain=1 / math.sqrt(2.0))
        nn.init.xavier_uniform_(self.v_proj.weight, gain=1 / math.sqrt(2.0))
        nn.init.xavier_uniform_(self.q_proj.weight, gain=1 / math.sqrt(2.0))

    def forward(self, x):
        y, _ = attention(self.q_proj(x), self.k_proj(x), self.v_proj(x))
        return y

    class SelfAttention_multihead(nn.Module):
```

```
def __init__(self, embed_dim, num_heads = 8, bias=True):
      super().__init__()
      self.head_dim = embed_dim // num_heads
      self.attn_heads = nn.ModuleList([AttentionHead(embed_dim, self.head_dim,
          bias) for _ in range(num_heads)])
      self.out_proj = nn.Linear(embed_dim, embed_dim, bias)
      self.reset_parameters()

def reset_parameters(self):
    if self.out_proj.bias is not None:
          nn.init.xavier_uniform_(self.out_proj.weight)
          nn.init.constant_(self.out_proj.bias, 0.)
def forward(self, x):
      output = torch.cat([head(x) for head in self.attn_heads], dim=-1)
      return self.out_proj(output)
```

If no modifications are specified, the default parameters used include an embedding dimension of 256, a window size of 7, a batch size of 2048, 4 epochs, and an Adam optimizer. We conducted a total of 6 experiments using the provided code that are summarized in the following two tables:

	Tain accuracy	Val. accuracy	Test accuracy	Train loss	Val. loss	Test loss
Trial 1	47.9	46.9	36.1	2.96	3.09	4.00
Trial 2	35.7	34.5	26.7	4.19	4.23	5.00
Trial 3	48.80	47.60	37.00	2.89	3.03	3.93
Trial 4	48.80	47.40	36.90	2.89	3.04	3.95
Trial 5	48.80	47.90	37.70	2.88	3.01	3.90
Trial 6	49.00	47.90	37.50	2.87	3.00	3.90

Table 1: Train, validation and test accuracy and loss of the las epoch.

	Training time	Num. Param.	Layers	Heads	Epochs	Scheduler
Trial 1	4h 40min	53310976	4	1	4	No
Trial 2	3h 40min	52783872	3	4	4	Yes
Trial 3	5h 4min	52783872	3	4	4	No
Trial 4	5h 41min	52783872	3	8	4	No
Trial 5	7h 4min	53310976	4	4	5	No
Trial 6	8h 15min	53310976	4	8	5	No

Table 2: Trainning time, number of parameters, and hyperparameters chosen in each trial.

For our initial trial, we only modified the number of layers to four while keeping all other parameters constant. We decided to do this because based on previous research and intuition, we believed that increasing the number of layers would lead to higher accuracy. So we didn't find necessary to spend hours of execution on it rather than having a baseline with 4 layers.

In the second trial, we attempted to increase the accuracy by using a OneCycleLR scheduler with lr=0.01, as we had previously done with CBOW. However, the results were unexpected, and the scheduler seemed to have a negative effect on the process. Due to

execution time limitations, we decided to remove the scheduler and work without it.

Subsequently, we conducted experiments with three Transformer layers, modifying the number of heads to four and eight (trial 3 and trial 4). As seen in the tables, both trials yielded better results than not using different heads, although using eight heads resulted in a worse performance than using four.

We then proceeded to try the same combination of heads (four and eight) with four Transformer layers (trial 5 and trial 6). The accuracy improved when adding more transformer layers, however, it was observed that using 4 heads gave better results than using 8 heads.

The best test accuracy was achieved with the four-layer, four-head combination. In this model, the number of Transformer Layers used seemed to have an impact on the accuracy, as expected. We also increased the number of epochs to observe its impact, but as seen in the logs and the results, it had little significance in the model.

The updated implementation involved dividing the input X into splits, where each split is processed by a separate head, instead of each head processing the entire input projected to a smaller dimension. This modification allows each attention head to focus on a different aspect of the input.

This modification was made to the $SelfAttention\_multihead$ function's forward function. We divide the input tensor X into h splits, where h is the number of attention heads. Each head then processes a different split of the input using the Scaled Dot Product Attention mechanism. After processing the splits, we concatenate the results from each head along the last dimension, which restores the tensor to its original size before the split. Then we apply a linear layer to the concatenated tensor to obtain the final output. Additionally, the projections in the $AttentionHead$ function were updated to be from $head\_dim$ to $head\_dim$. These modifications allowed for further experiments to be conducted.

```
class AttentionHead(nn.Module):
    def __init__(self, head_dim, bias=True):
        super().__init__()
        self.head_dim = head_dim
        self.k_proj = nn.Linear(self.head_dim, self.head_dim, bias)
        self.v_proj = nn.Linear(self.head_dim,self.head_dim, bias)
        self.q_proj = nn.Linear(self.head_dim, self.head_dim, bias)
        self.reset_parameters()

    def reset_parameters(self):
        # Empirically observed the convergence to be much better with the scaled
            initialization
        nn.init.xavier_uniform_(self.k_proj.weight, gain=1 / math.sqrt(2.0))
        nn.init.xavier_uniform_(self.v_proj.weight, gain=1 / math.sqrt(2.0))
        nn.init.xavier_uniform_(self.q_proj.weight, gain=1 / math.sqrt(2.0))

    def forward(self, x):
        y, _ = attention(self.q_proj(x), self.k_proj(x), self.v_proj(x))
        return y
```

```
class SelfAttention_multihead(nn.Module):
 def __init__(self, embed_dim, num_heads = 8, bias=True):
       super().__init__()
       self.num_heads = num_heads
       self.head_dim = embed_dim // num_heads
       self.attn_heads = nn.ModuleList([AttentionHead(self.head_dim, bias) for _
           in range(num_heads)])
       self.out_proj = nn.Linear(embed_dim, embed_dim, bias)
       self.reset_parameters()

 def reset_parameters(self):
    if self.out_proj.bias is not None:
          nn.init.xavier_uniform_(self.out_proj.weight)
          nn.init.constant_(self.out_proj.bias, 0.)

 def forward(self, x):
       batch_size, words_dim, embed_dim = x.size()
       split_size = embed_dim // self.num_heads
       x = x.view(batch_size, words_dim, self.num_heads,
           split_size).transpose(1, 2)

       attn_outputs = [self.attn_heads[i](x[:, i]) for i in
           range(self.num_heads)]

       output = torch.cat(attn_outputs, dim=-1)

       return self.out_proj(output)
```

We wanted to perform additional experiments, but we encountered difficulties when implementing multihead attention, which took up a significant amount of our free execution time. Additionally, we also spent time working on the first implementation shown in this report. As a result, we were not able to conduct as many experiments as we had hoped.

	Tain accuracy	Val. accuracy	Test accuracy	Train loss	Val. loss	Test loss
Trial 1	48.1	47.2	36.8	2.95	3.06	3.94
Trial 2	48.1	47.3	36.9	2.94	3.04	3.92
Trial 3	49.1	47.9	37.50	2.84	3.00	3.91

Table 3: Train, validation and test accuracy and loss of the las epoch.

	Training time	Num. Param.	Layers	Heads	Epochs	Embeding size	Dropout
Trial 1	5h 26min	52721152	4	4	4	256	0.1
Trial 2	6h 20min	52622848	4	8	4	256	0.1
Trial 3	8h 58min	65890688	4	8	5	320	0.15

Table 4: Trainning time, number of parameters, and hyperparameters in each trial.

We attempted models with 4 layers, as previous experiments suggested that more layers would result in higher accuracy. Our results indicate that adding more heads from 4 to

8 in this new implementation slightly improves the accuracy. We conducted a final trial with 4 layers, 8 heads, and increased embedding size, number of epochs, and dropout. We did so because prior experiments suggested that the models did not generalize well. We opted to augment the embedding size as it would enable the model to acquire a greater amount of information. Our last trial achieved the best test accuracy using this modified implementation.

Other options that could have been explored include adding mean or max pooling layers after the transformer blocks, further increasing the embedding dimension as it has yielded good results, increasing the dropout rate to improve generalization, attempting to incorporate a scheduler or make modifications to the learning rate, increasing the batch size, or adding more layers and heads to the model.