Assignment 2. Language models

Alex Carrillo and Robert Tura
Spoken and Written Language Processing

March 31, 2020
Universitat Politècnica de Catalunya

1 Introduction

In the last assignment we studied a *word2vec* technique that generated continuous dense vector representations of words, which captured contextual and semantic similarity. We focused on the *CBOW* model, which learned word representations by predicting a word according to its context defined as a symmetric window containing all the surrounding words.

This time, we study a concept which aims to improve the performance of that specific predicting model, and gives rise to other extends like neural machine translation applications. The principal motivation of this assignment is to explore different model architectures and try to give an intuition of why they provide better or worse results.

2 The Transformer

Putting things into context, we should briefly comment *seq2seq* models in NLP which convert sequences of type x into sequences of type y. For example, translating Catalan sentences to Spanish is one of these sequence-to-sequence tasks. Some of these models are based on *recurrent neural networks* (*RNNs*) and are enhanced with the addition of an *attention mechanism*, which we will talk about later on. So, although seq2seq models are pretty versatile and useful, they suffer certain limitations, for instance, they find it hard to deal with long-range language dependencies.

The *Transformer* is a new architecture that aims to solve seq2seq tasks like ours (that of predicting the center word of a 5-gram) while handling long-range dependencies with ease. The idea behind it is to rely on a *self-attention mechanism* to compute representations of its input and output without using RNNs.

Without going into details, in this report we consider the Transformer as a language model, in the sense that we do not use the original Transformer architecture (with both *encoder* and *decoder*), but the *Transformer decoder model* which discards the encoder part. In this way, there is only one single input sentence rather than two separate source and target sequences.

2.1 Self-Attention

Opening a parenthesis on attention is necessary to understand the why of our chosen models. In a nutshell, attention allows the model to focus on the relevant parts of the input sequence as needed, that is, to get a better understanding of a certain word in the sequence.

To do this, the mathematical intuition behind is to look at other positions in the input sequences for clues that can help lead to a better encoding of a given word. So, self-attention creates three vectors Q (*query*), K (*key*) and V (*value*) (although in practice it is done in matrix form for faster processing) from each of the embeddings of each word, by multiplying the embedding by three trained matrices W_Q, W_K and W_V.

These vectors (matrices) are abstractions for computing an output vector (matrix) for each word that captures which of the other words have more weight and which must be drown-out because they are irrelevant. Without much more details, the output self-attention matrix can be calculated by:

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

The code we used for `attention` is the provided in the Kaggle competition:

```
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, -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
```

3 Models

3.1 Self-Attention Transformer

3.1.1 Baseline model

Firstly, we studied the provided baseline model in order to understand the architecture seen in class. The following is a simple schematic of that baseline model:

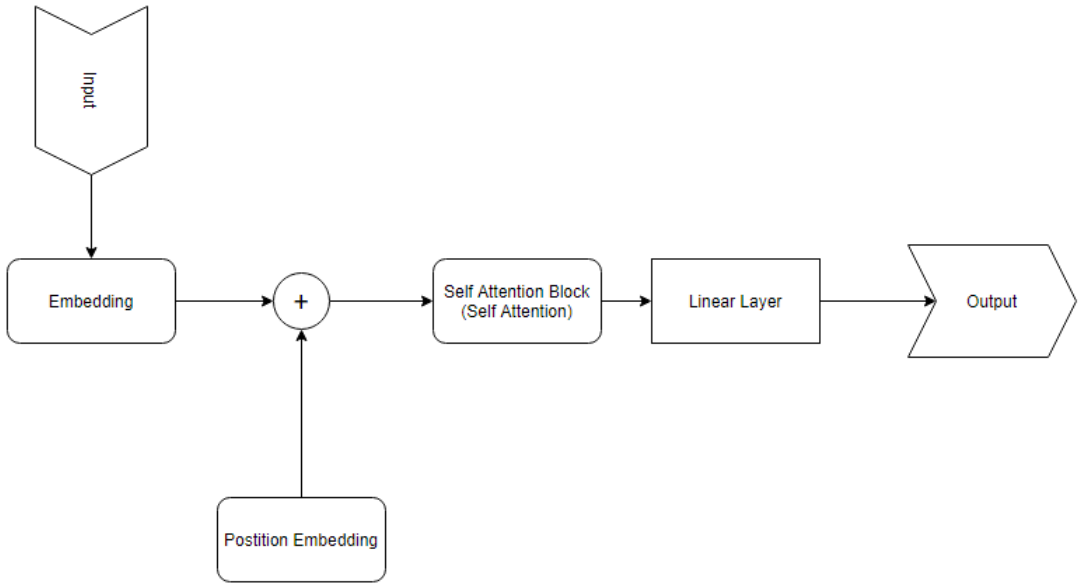

Figure 1: Self-Attention baseline model architecture.

Let's take a look at the `SelfAttention` block code to comprehend the mathematical intuition that we briefly discussed in the previous section:

```
class SelfAttention(nn.Module):
    def __init__(self, embed_dim, bias=True):
        super().__init__()
        self.k_proj = nn.Linear(embed_dim, embed_dim, bias=bias)
        self.v_proj = nn.Linear(embed_dim, embed_dim, bias=bias)
        self.q_proj = nn.Linear(embed_dim, embed_dim, bias=bias)
        self.out_proj = nn.Linear(embed_dim, embed_dim, bias=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))
        nn.init.xavier_uniform_(self.v_proj.weight, gain=1 / math.sqrt(2))
        nn.init.xavier_uniform_(self.q_proj.weight, gain=1 / math.sqrt(2))
        nn.init.xavier_uniform_(self.out_proj.weight)
        if self.out_proj.bias is not None:
            nn.init.constant_(self.out_proj.bias, 0.)

    # B = Batch size
    # W = Number of context words (left + right)
    # E = embedding_dim
    def forward(self, x):
        # x shape is (B, W, E)
        q = self.q_proj(x)
        # q shape is (B, W, E)
        k = self.k_proj(x)
        # k shape is (B, W, E)
        v = self.v_proj(x)
        # k shape is (B, W, E)
        y, _ = attention(q, k, v)
        # y shape is (B, W, E)
        y = self.out_proj(y)
        # y shape is (B, W, E)
        return y
```

To get started, this preliminary architecture uses only 1 Transformer layer with Self-Attention. The model is simple but effective and it is a very good base to start testing new improvements. The `TransformerLayer` and `Predictor` classes of the network can be found in Appendix A.

3.1.2 Sharing two Transformer layers

An early test was to increase the number of layer to two, but with the nuance of sharing those Transformer layer in the sense of using the same weights. As we will see in the results, this approach is not effective: basically, because the runtime is increased no only without any accuracy gain, but lowering the accuracy below the baseline. The `Predictor` class of the network can be found in Appendix B.

3.1.3 Using three Transformer layers

With the above in mind, we bet on not only increasing the number of layers, but to use different ones and not to share their weights.

Referring to the Figure 1, the architecture would be the same but adding three Self-Attention blocks one after another. This approach seemed to have better results and the increase in time was not that concerning. The `Predictor` class of the network can be found in Appendix C.

3.2 Multi-Headed Self-Attention Transformer

Now, we focus on what we think will improve the model: a refinement in the Self-Attention mechanism, what is called *Multi-Headed Self-Attention*. This new architecture helps the model to focus on different parts of the input sequence and, one way or another, now the network can have some more like a global vision.

Regarding to the mathematical intuition, and in conjunction with what we explained about Self-Attention, now in Multi-Headed we have several sets of matrices W_Q, W_K and W_V (*query*, *key* an *value* matrices); that is what we call the number of *heads* of the architecture. As we did in base Self-Attention, the fact of projecting the input embeddings by means of these matrices implies representing each word embedding in another representation subspace (that one where the attention is focused on some other words). Hence, the essence of Multi-Headed Self-Attention is to project those embeddings into several representation subspaces.

The PyTorch code we used to try out this new architecture for the model is the following (with some similarities with the `Self-Attention` class and with some tweaks like the concatenation of all the attention heads):

```
class MultiHeadAttention(nn.Module):
    def __init__(self, d_model, heads = 8, dropout = 0.1):
        super().__init__()
        self.d_model = d_model
        self.d_k = d_model // heads
        self.h = heads
        self.q_linear = nn.Linear(d_model, d_model)
        self.v_linear = nn.Linear(d_model, d_model)
        self.k_linear = nn.Linear(d_model, d_model)
        self.dropout = nn.Dropout(dropout)
        self.out = nn.Linear(d_model, d_model)

    def forward(self, q, k, v, mask=None):
        bs = q.size(0)
        # perform linear operation and split into h heads
        k = self.k_linear(k).view(bs, -1, self.h, self.d_k)
        q = self.q_linear(q).view(bs, -1, self.h, self.d_k)
        v = self.v_linear(v).view(bs, -1, self.h, self.d_k)

        # transpose to get dimensions bs * h * sl * d_model
        k = k.transpose(1,2)
        q = q.transpose(1,2)
        v = v.transpose(1,2)
        # calculate attention using function we will define next
        scores, _ = attention(q, k, v, mask, self.dropout)

        # concatenate heads and put through final linear layer
        concat = scores.transpose(1,2).contiguous()\
        .view(bs, -1, self.d_model)

        output = self.out(concat)

        return output
```

3.2.1 One Transformer layer

So, with the implementation of the Multi-Headed Self-Attention block, we start with only one Transformer layer to see how long does it takes and to notice the effects it creates. The `TransformerLayer` and `Predictor` classes of the network can be found in Appendix D. Again, all the information regarding the specific results of the models are summarized in a final table at the end of the sections.

Just say, the scheme of the network is the same as for Figure 2, but considering one Multi-Headed attention block. Moreover, two configurations are tested: one with 8 heads and another one with 16 heads (with which a slight improvement in accuracy is obtained).

3.2.2 Using two Transformer layers

The same Multi-Headed Self-Attention approach is tested now with two Transformer layers. We expected to improve the 1 layer architecture, and indeed we achieved it. However, for the moment, we have not obtained better results than with the Self-Attention mechanism. We will have to see how to tune the model.

Moreover, two configurations are tested: one with 8 heads and another one with 16 heads (with which a slight improvement in accuracy is obtained). The `Predictor` class of the network can be found in Appendix E.

3.2.3 Using three Transformer layers

With the above in mind, we keep betting on increasing the number of layer in order to enhance the performance of the model. Now, we add up to three Transformer layers. The architecture of the current network is mainly the same stated in the previous cases, just adding three Multi-Headed blocks one after another:

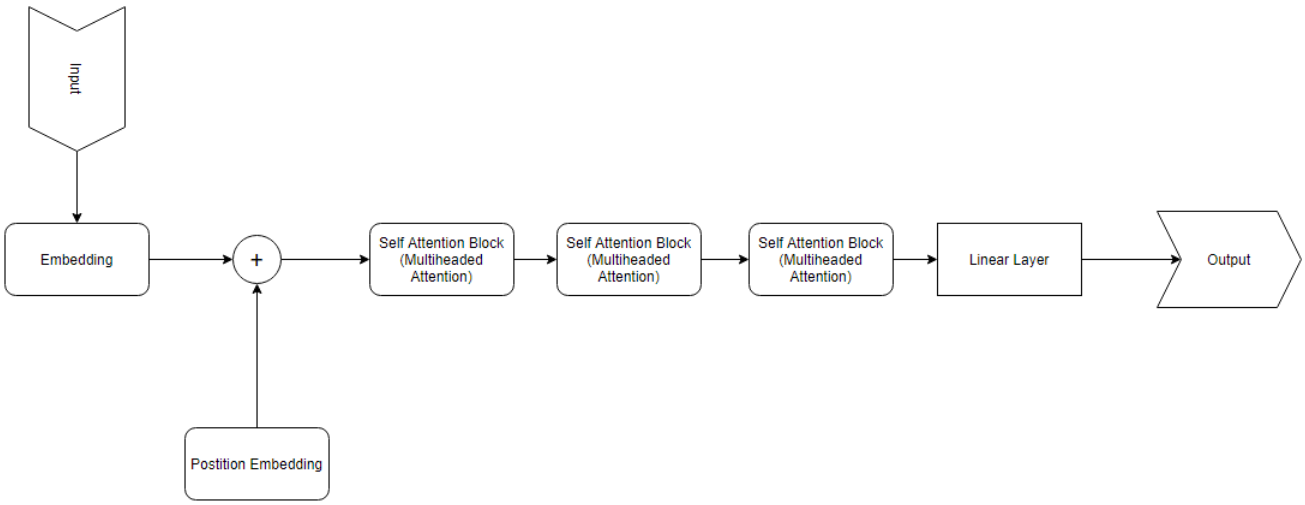

Figure 2: Multi-Headed Self-Attention with three Transformer layer model architecture.

This time, we are only testing a configuration with 8 heads because, as we have been observing in the previous sets ups, we expect an increase in the execution time (as seen in the case of 8 heads), so no test is performed with 16 heads. It should be noted that with this network we have achieved our highest accuracy and, although we have tried new things, none of them has overcome this model, therefore, it becomes the best accuracy obtained at this assignment: **0.34250**. The `Predictor` class of the network can be found in Appendix F.

3.3 Sharing input and output embeddings

All our previous models had an initial embedding layer which ensured that the model is fed with proper data. Regardless, in the end of the model the data should be turned into the initial form. The first approach to this aspect of the model was simple, leave it to the model to train them as non related layers. But then we thought that this two layers have connection between them and maybe their parameters could be trained, so this next model focuses on having less parameters for its embedding layers as are trained as a whole.

This is the code that allows us to share the embedding at the input and the output:

```
class Predictor(nn.Module):
    def __init__(self, num_embeddings, embedding_dim, context_words=4):
        super().__init__()
        self.emb = nn.Embedding(num_embeddings, embedding_dim, padding_idx=0)
        self.lin = nn.Linear(embedding_dim, num_embeddings, bias=False)
        #self.att = TransformerLayer(embedding_dim)
        self.att_1 = TransformerLayer(embedding_dim)
        self.att_2 = TransformerLayer(embedding_dim)
        self.att_3 = TransformerLayer(embedding_dim)
        self.position_embedding = nn.Parameter(torch.Tensor(context_words, embedding_dim))
        nn.init.xavier_uniform_(self.position_embedding)

    # B = Batch size
    # W = Number of context words (left + right)
    # E = embedding_dim
    # V = num_embeddings (number of words)
    def forward(self, input):
        # input shape is (B, W)
        e = self.emb(input)
        # e shape is (B, W, E)
        u = e + self.position_embedding
        # u shape is (B, W, E)
        #v = self.att(u)
        v = self.att_1(self.att_2(self.att_3(u)))
        # v shape is (B, W, E)
        x = v.sum(dim=1)
        # x shape is (B, E)
        #y = self.lin(x)
        y = x.matmul(self.emb.weight.t())
        # y shape is (B, V)
        return y
```

As it is clearly shown in the results table this model did not give us a great performance with respect to the other contestants: the lack of accuracy was about 1 percent worse, but compared with its competitor that did not share its embedding we see that time-wise the new model needed roughly half the time. This method is known for its improvement in time and maintains better the semantic structure of the output space word but in this case seems that it felt a little bit short (maybe it needed some further hyperparameter tweaking) in comparison with others.

3.4 Multilayer perceptron (MLP) over the concatenated input vectors

The last trick up our sleeve was a simple multi layer perceptron just after the embedding of our word sets, in order to extract new features and not just the words itself, and maybe new ones as: noun, adverb, adjective, etc. This is an schematic of the model with 3 transformer layers, using Multi-Headed Self-Attention and with a couple of linear layers:

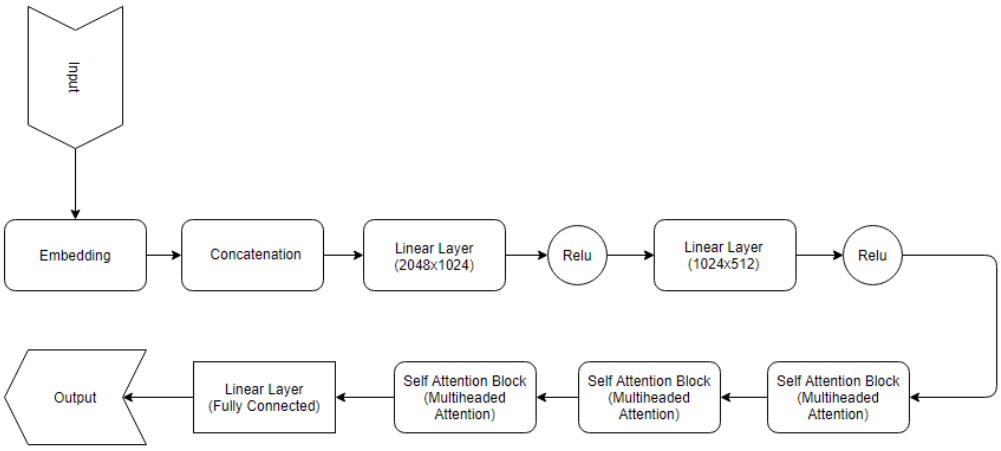

Figure 3: Multi-Headed Self-Attention with a MLP model architecture.

At first we had problems with this model for our GPU VRAM which was being drained and was not enough for the model to run; after tweaking and simplifying the model and its parts we got to the model that is shown, featuring 2 linear layers of sizes (2048x1024) and (1024x512), respectively. There is lots of things we would add to it, like dropout layers and other, but this is the best model that was able to run. In these terms, as this was kind of hard to get to work, we focused more on the other options for the model tweaking. This is the code used to make the model possible:

```
class Predictor(nn.Module):
    def __init__(self, num_embeddings, embedding_dim, target, context_words=4):
        super().__init__()
        self.target = target
        self.emb = nn.Embedding(num_embeddings, embedding_dim, padding_idx=0)
        self.linear1 = nn.Linear(2048, 1024, bias=False)
        self.linear2 = nn.Linear(1024, 512, bias=False)
        self.lin = nn.Linear(embedding_dim, num_embeddings, bias=False)
        self.att_1 = TransformerLayer(embedding_dim)
        self.att_2 = TransformerLayer(embedding_dim)
        self.att_3 = TransformerLayer(embedding_dim)
        self.position_embedding = nn.Parameter(torch.Tensor(context_words, embedding_dim))
        nn.init.xavier_uniform_(self.position_embedding)

    # B = Batch size, W = # context words, E = embedding_dim, V = num_embeddings
    def forward(self, input):
        # input shape is (B, W)
        e = self.emb(input)
        # e shape is (B, W, E)
        u = e.view(e.size(0), -1)
        u = F.relu(self.linear1(u))
        u = F.relu(self.linear2(u))
        # u shape is (B, W, E)
        v = self.att_1(self.att_2(self.att_3(u)))
        # v shape is (B, W, E)
        y = self.lin(v)
        # y shape is (B, V)
        return y
```

3.5 Hyperparameter optimization

Finally here, after trying lots of different models with more or less difficulty, we ended up with 4 different architectures. Then we went back to the first lab of POE and try to apply the results we found interesting and all the hyperparameter tweaking information we took there. Unfortunately, this did not end being a perfect guide on how to improve our models but at least helped us understand what the parameters meant and why should we change ones.

Unluckily, our best model was not any of the ones that have been tweaked, which means that there is no mandatory correlations between models; for this time, increasing the embedding dimension did not improve the accuracy of the model and did just affect the computation times. Also with the Multi-Headed Self-Attention we got a new parameter to tweak as the amount of heads that the model takes into account showed a little increase in the run time and little to no improve in the model's accuracy.

In the end we ended with a fairly simple model that did not have much change from its first iteration. Which is not bad at all, but a warning that maybe we focused on the wrong parameters and maybe some parts of the code are not properly coded for some results, like the MLP one, were deceiving.

4 Results

As a final recap we took all the information of each run of the models and we present it in the form of a table:

Architecture	Dim.	Tf. layers	Heads	Params.	Time	Accuracy	Loss
Self-Attention[1]	256	1	-	51,729,152	3.81	0.33133	3.42
	256	(shared) 2	-	55,404,800	4.17	0.32783	3.40
	256	3	-	55,404,800	4.59	0.34100	3.37
[1]**multilayer perceptron (MLP)**	256	3	-	109,759,488	8.78	0.04100	7.39
Multi-Headed Self-Attention[2]	256	1	8	55,404,800	3.86	0.33516	3.39
	256	1	16	55,404,800	3.94	0.33783	3.34
	256	2	8	55,404,800	4.32	0.33766	3.36
	256	2	16	55,404,800	4.50	0.33783	3.39
	256	3	8	55,404,800	4.79	→ 0.34250	3.35
	512	2	8	109,759,488	7.58	0.33466	3.34
	576	3	12	123,593,920	7.80	0.34050	3.30
[2]**sharing I/O embeddings**	256	3	8	52,783,360	4.63	0.33100	3.50

Table 1: Summarized results of all our model architectures.

As a final comment, we wanted to keep trying models but our quota in Kaggle and the restriction of 2 submissions per day did not help, so we tried 4 different architectures which we think is fine given we have not much experience, but we would have wanted to keep tweaking the models further.

For instance, we want to comment on a point that has not been emphasized throughout the report. This has to do with the replacement of the softmax layer with AdaptativeSoftmax, which speeds up our deep learning language model according to what we have read. The solution of using `log_prob()` in the AdaptiveSoftMax is not optimal with regard to training time but it allows to use it without much change (you just have to change the loss to use this output with the logarithm of the probability). For future research related to this area, it should be interesting to study this in more depth.

Appendix

A Baseline model

```
class TransformerLayer(nn.Module):
    def __init__(self, d_model, dim_feedforward=512, dropout=0.1, activation="relu"):
        super().__init__()
        self.self_attn = SelfAttention(d_model)
        # Implementation of Feedforward model
        self.linear1 = nn.Linear(d_model, dim_feedforward)
        self.dropout = nn.Dropout(dropout)
        self.linear2 = nn.Linear(dim_feedforward, d_model)
        self.norm1 = nn.LayerNorm(d_model)
        self.norm2 = nn.LayerNorm(d_model)
        self.dropout1 = nn.Dropout(dropout)
        self.dropout2 = nn.Dropout(dropout)

    def forward(self, src):
        src2 = self.self_attn(src)
        src = src + self.dropout1(src2)
        src = self.norm1(src)
        src2 = self.linear2(self.dropout(F.relu(self.linear1(src))))
        src = src + self.dropout2(src2)
        src = self.norm2(src)
        return src

class Predictor(nn.Module):
    def __init__(self, num_embeddings, embedding_dim, context_words=4):
        super().__init__()
        self.emb = nn.Embedding(num_embeddings, embedding_dim, padding_idx=0)
        self.lin = nn.Linear(embedding_dim, num_embeddings, bias=False)
        self.att = TransformerLayer(embedding_dim)
        self.position_embedding = nn.Parameter(torch.Tensor(context_words, embedding_dim))
        nn.init.xavier_uniform_(self.position_embedding)

    # B = Batch size
    # W = Number of context words (left + right)
    # E = embedding_dim
    # V = num_embeddings (number of words)
    def forward(self, input):
        # input shape is (B, W)
        e = self.emb(input)
        # e shape is (B, W, E)
        u = e + self.position_embedding
        # u shape is (B, W, E)
        v = self.att(u)
        # v shape is (B, W, E)
        x = v.sum(dim=1)
        # x shape is (B, E)
        y = self.lin(x)
        # y shape is (B, V)
        return y
```

B Self-Attention sharing two Transformer layers

```
class Predictor(nn.Module):
    def __init__(self, num_embeddings, embedding_dim, context_words=4):
        super().__init__()
        self.emb = nn.Embedding(num_embeddings, embedding_dim, padding_idx=0)
        self.lin = nn.Linear(embedding_dim, num_embeddings, bias=False)
        self.att = TransformerLayer(embedding_dim)
        self.position_embedding = nn.Parameter(torch.Tensor(context_words, embedding_dim))
        nn.init.xavier_uniform_(self.position_embedding)

    # B = Batch size
    # W = Number of context words (left + right)
    # E = embedding_dim
    # V = num_embeddings (number of words)
    def forward(self, input):
        # input shape is (B, W)
        e = self.emb(input)
        # e shape is (B, W, E)
        u = e + self.position_embedding
        # u shape is (B, W, E)
        v = self.att(self.att(u))
        # v shape is (B, W, E)
        x = v.sum(dim=1)
        # x shape is (B, E)
        y = self.lin(x)
        # y shape is (B, V)
        return y
```

C Self-Attention using three Transformer layers

```
class Predictor(nn.Module):
    def __init__(self, num_embeddings, embedding_dim, context_words=4):
        super().__init__()
        self.emb = nn.Embedding(num_embeddings, embedding_dim, padding_idx=0)
        self.lin = nn.Linear(embedding_dim, num_embeddings, bias=False)
        self.att_1 = TransformerLayer(embedding_dim)
        self.att_2 = TransformerLayer(embedding_dim)
        self.att_3 = TransformerLayer(embedding_dim)
        self.position_embedding = nn.Parameter(torch.Tensor(context_words, embedding_dim))
        nn.init.xavier_uniform_(self.position_embedding)

    # B = Batch size
    # W = Number of context words (left + right)
    # E = embedding_dim
    # V = num_embeddings (number of words)
    def forward(self, input):
        # input shape is (B, W)
        e = self.emb(input)
        # e shape is (B, W, E)
        u = e + self.position_embedding
        # u shape is (B, W, E)
        v = self.att_1(self.att_2(self.att_3(u)))
        # v shape is (B, W, E)
```

```
        x = v.sum(dim=1)
        # x shape is (B, E)
        y = self.lin(x)
        # y shape is (B, V)
        return y
```

D Multi-Headed Self-Attention using one Transformer layer

```
class TransformerLayer(nn.Module):
    def __init__(self, d_model, dim_feedforward=512, dropout=0.1, activation="relu"):
        super().__init__()
        self.attn = MultiHeadAttention(d_model)
        # Implementation of Feedforward model
        self.linear1 = nn.Linear(d_model, dim_feedforward)
        self.dropout = nn.Dropout(dropout)
        self.linear2 = nn.Linear(dim_feedforward, d_model)
        self.norm1 = nn.LayerNorm(d_model)
        self.norm2 = nn.LayerNorm(d_model)
        self.dropout1 = nn.Dropout(dropout)
        self.dropout2 = nn.Dropout(dropout)

    def forward(self, src):
        src2 = self.attn(src, src, src)
        src = src + self.dropout1(src2)
        src = self.norm1(src)
        src2 = self.linear2(self.dropout(F.relu(self.linear1(src))))
        src = src + self.dropout2(src2)
        src = self.norm2(src)
        return src

class Predictor(nn.Module):
    def __init__(self, num_embeddings, embedding_dim, context_words=4):
        super().__init__()
        self.emb = nn.Embedding(num_embeddings, embedding_dim, padding_idx=0)
        self.lin = nn.Linear(embedding_dim, num_embeddings, bias=False)
        self.att = TransformerLayer(embedding_dim)
        self.position_embedding = nn.Parameter(torch.Tensor(context_words, embedding_dim))
        nn.init.xavier_uniform_(self.position_embedding)

    # B = Batch size
    # W = Number of context words (left + right)
    # E = embedding_dim
    # V = num_embeddings (number of words)
    def forward(self, input):
        # input shape is (B, W)
        e = self.emb(input)
        # e shape is (B, W, E)
        u = e + self.position_embedding
        # u shape is (B, W, E)
        v = self.att(u)
        # v shape is (B, W, E)
        x = v.sum(dim=1)
        # x shape is (B, E)
        y = self.lin(x)
```

```
        # y shape is (B, V)
        return y
```

E Multi-Headed Self-Attention using two Transformer layers

```
class Predictor(nn.Module):
    def __init__(self, num_embeddings, embedding_dim, context_words=4):
        super().__init__()
        self.emb = nn.Embedding(num_embeddings, embedding_dim, padding_idx=0)
        self.lin = nn.Linear(embedding_dim, num_embeddings, bias=False)
        self.att_1 = TransformerLayer(embedding_dim)
        self.att_2 = TransformerLayer(embedding_dim)
        self.position_embedding = nn.Parameter(torch.Tensor(context_words, embedding_dim))
        nn.init.xavier_uniform_(self.position_embedding)

    # B = Batch size
    # W = Number of context words (left + right)
    # E = embedding_dim
    # V = num_embeddings (number of words)
    def forward(self, input):
        # input shape is (B, W)
        e = self.emb(input)
        # e shape is (B, W, E)
        u = e + self.position_embedding
        # u shape is (B, W, E)
        v = self.att_1(self.att_2(u))
        # v shape is (B, W, E)
        x = v.sum(dim=1)
        # x shape is (B, E)
        y = self.lin(x)
        # y shape is (B, V)
        return y
```

F Multi-Headed Self-Attention using three Transformer layers

```
class Predictor(nn.Module):
    def __init__(self, num_embeddings, embedding_dim, target, context_words=4):
        super().__init__()
        self.target = target
        self.emb = nn.Embedding(num_embeddings, embedding_dim, padding_idx=0)
        self.lin = nn.Linear(embedding_dim, num_embeddings, bias=False)
        self.att_1 = TransformerLayer(embedding_dim)
        self.att_2 = TransformerLayer(embedding_dim)
        self.att_3 = TransformerLayer(embedding_dim)
        self.position_embedding = nn.Parameter(torch.Tensor(context_words, embedding_dim))
        nn.init.xavier_uniform_(self.position_embedding)

    # B = Batch size
    # W = Number of context words (left + right)
    # E = embedding_dim
    # V = num_embeddings (number of words)
    def forward(self, input):
        # input shape is (B, W)
```

```
        e = self.emb(input)
        # e shape is (B, W, E)
        u = e + self.position_embedding
        # u shape is (B, W, E)
        v = self.att_1(self.att_2(self.att_3(u)))
        # v shape is (B, W, E)
        x = v.sum(dim=1)
        # x shape is (B, E)
        y = self.lin(x)
        # y shape is (B, V)
        return y
```

References

[1] Keita Kurita. *Paper Dissected: "Attention is All You Need" Explained [Blog post]*. 2017. URL: https://mlexplained.com/2017/12/29/attention-is-all-you-need-explained/.

[2] Ashish Vaswani et al. *Attention Is All You Need*. 2017. arXiv: 1706.03762 [cs.CL].

[3] Jay Alammar. *The Illustrated Transformer [Blog post]*. 2018. URL: http://jalammar.github.io/illustrated-transformer/.

[4] Jay Alammar. *Visualizing A Neural Machine Translation Model (Mechanics of Seq2seq Models With Attention) [Blog post]*. 2018. URL: https://jalammar.github.io/visualizing-neural-machine-translation-mechanics-of-seq2seq-models-with-attention/.

[5] David Bressler. *Speed up your deep learning language model up to 1000 with the adaptive softmax, Part 1 [Blog post]*. 2018. URL: https://towardsdatascience.com/speed-up-your-deep-learning-language-model-up-to-1000-with-the-adaptive-softmax-part-1-e7cc1f89fcc9.

[6] Giuliano Giacaglia. *How Transformers Work [Blog post]*. 2019. URL: https://towardsdatascience.com/transformers-141e32e69591.

[7] Prateek Joshi. *How do Transformers Work in NLP? A Guide to the Latest State-of-the-Art Models [Blog post]*. 2019. URL: https://www.analyticsvidhya.com/blog/2019/06/understanding-transformers-nlp-state-of-the-art-models/.

[8] Elena Voita et al. "Analyzing Multi-Head Self-Attention: Specialized Heads Do the Heavy Lifting, the Rest Can Be Pruned". In: *CoRR* abs/1905.09418 (2019). arXiv: 1905.09418. URL: http://arxiv.org/abs/1905.09418.

[9] Lilian Weng. *Generalized Language Models [Blog post]*. 2019. URL: https://lilianweng.github.io/lil-log/2019/01/31/generalized-language-models.html#transformer-decoder-as-language-model.