Language Modeling with Transformers

Co-authors: Jordi Vaccher[1], *Daniel Navarrete*[1]

[1]Master in Advanced Telecommunication Technologies
ETSETB - Universitat Politècnica de Catalunya
jordi.vaccher@estudiantat.upc.edu, daniel.navarrete.jimenez@estudiantat.upc.edu

1. Introduction

This report shows the study and development of different models for non-causal language modeling. We are comparing the performance of a simple Feed Forward Neural Network with a Transformer Layer. Therefore we are increasing the number of Transformer Layers of the system and also using Multihead Attention. Finally we have applyed a normalization of Input/Output Shared Embeddings.

This work follows the previous study about Continuous Bag of Words (CBOW), so the same code structure is used.

2. Try a Feedforward Neural Network Language Model

In this section we fed a Feed Forward Neural Network (FFNN) with the input embeddings so we have obtained our baseline result. The FFNN has the following structure.

Table 1: *FFNN structure*

Linear - (Embedding dim x context words) x 512
Tanh - 512
Linear2 - 512 x num Embeddings
Softmax - Output

From this model we have obtained worst results than using our first system based in CBOW. Despite this results it has a nice performance considering we are feeding a network with the embeddings.

Table 2: *Train and validation accuracy results*

Accuracy
Train acc = 0.241
Dev acc Wikipedia = 0.241
Dev acc El Periodico = 0.192

3. Increase the number of TransformerLayers

The provided code contains a Transformer Layer and it has a greater performance than the obtained using a FFNN. The Transformer Layer uses attention in order to make more relevant the needed processed word. It takes advantage from long-range dependencies and it is parallelizable. So an attention layer and a FFNN is contained into the Transformer Layer.
We have achieved the following results:

Table 3: *Train and validation accuracy results*

Accuracy
Train acc = 0.450
Dev acc Wikipedia = 0.438
Dev acc El Periodico = 0.335

In this section we have increased the number of Transformers used in the system from 1 to 3, obtaining the following result.

Table 4: *Train and validation accuracy results*

Accuracy
Train acc = 0.453
Dev acc Wikipedia = 0.441
Dev acc El Periodico = 0.336

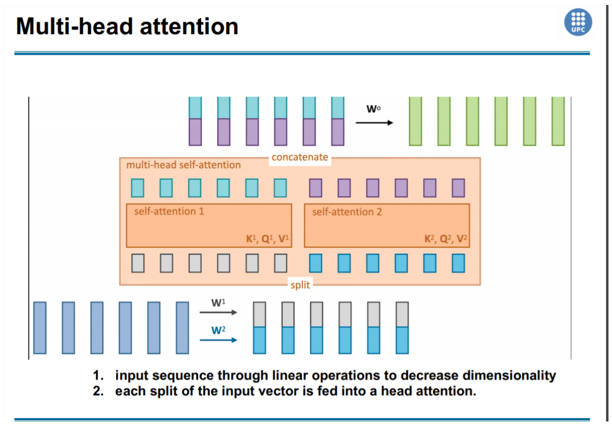
As we can see it outperforms slightly the baseline. The attention of each Layer is not paralleled so it does not take advantage at all. In order to improve this results we are using Muti-Head Attention in the next section.

4. TransformerLayer with Multi-Head attention

In this section we are using Multi-Head Attention so we want to parallelize several Transformer Layers in order to expand the ability to focus on different positions and give the Attention Layer multiple representation subspaces.
To do it we have splitted the embedding tensors as the Figure 1 shows.

We have splited into 4 tensors of dimension 1/4 x Embedding Dimension. We have chose this dimension due to the number 4 is multiple of the Embedd Dimension and it fill completly every new tensor.

This new configuration has obtained the following results.


Multi-head attention

1. input sequence through linear operations to decrease dimensionality
2. each split of the input vector is fed into a head attention.

Figure 1: *Multi-head attention concept from UPC slides*

Table 5: *Train and validation accuracy results*

Accuracy
Train acc = 0.465
Dev acc Wikipedia = 0.438
Dev acc El Periodico = 0.334

5. Sharing input/output embedding

In this section it is proposed to use a Normalization of Input-output Shared Embeddings. By applying normalization methods on embedding weight matrices, the bias of estimations for output scores is eliminated.

So to do it we have matched both Tensors: Input Tensor Embeddings and Output Tensor (Self Position) Embeddings.

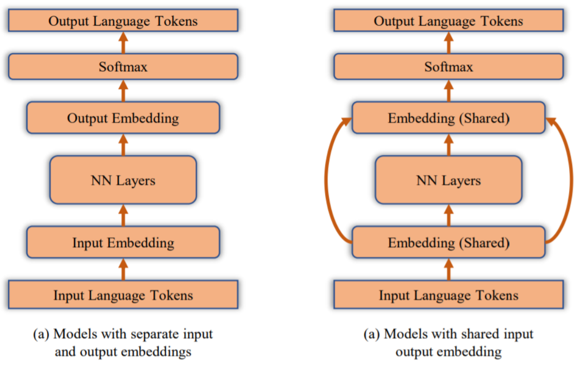


Figure 2: *Sharing In/output embeddings concept from Normalization of Input-output Shared Embeddings in Text Generation Models Paper, University of California, Riverside.*

In order to check the result using a tensor of ones as initial weights, we have trained another model, obtaining the following results:

Table 6: *Train and validation accuracy results*

Accuracy
Train acc = 0.424
Dev acc Wikipedia = 0.435
Dev acc El Periodico = 0.328

6. Hyperparameter optimization

Finally we are tunning the hyperparameters in order to optimize the learning performance.

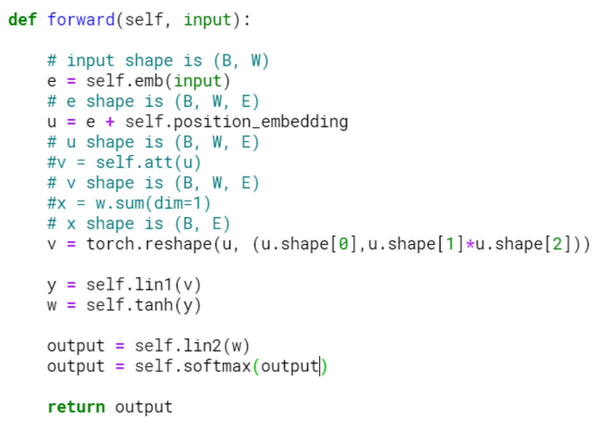
- Firstly we have doubled the batch size in order to increase the learning speed and wondering it could improve the accuracy results. The training time decreases considerably, i.e While an epoch (in Sharing input/output embeddings and with default batch size) can took two hours, setting the batch size as the double we achieve train twice as fast. By the other hand, the results outperform slightly the default hyperparameters, so it is negligible.

7. Conclusions

- To sum up, we have seen that using Transformer layers and multi-head attention the Language Model improve considerably and it outperform not only the CBOW, but also a Feed Forward Neural Network.
- Using several Transformer Layers with no parallelization do not improve an unique Transformer Layer and it does not take advantage of all available data.
- Using the Normalization of Input / Output Shared Embeddings the system outperforms the baseline. The results are similar to the Multi-Head attention but not such greater.
- For future work we could use both methods. It could be very interesting to do that experiment but the training time of other experiments has been so long. So this has made it difficult to try.

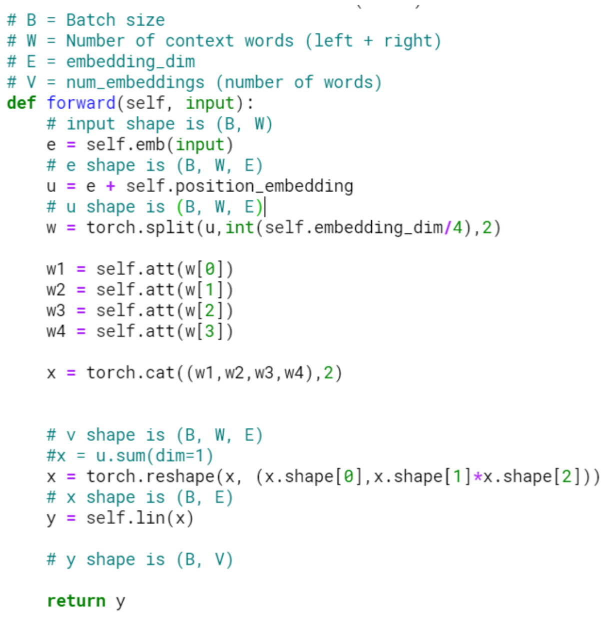
8. ANNEX

In this section we show the code modifications we have done in each section.



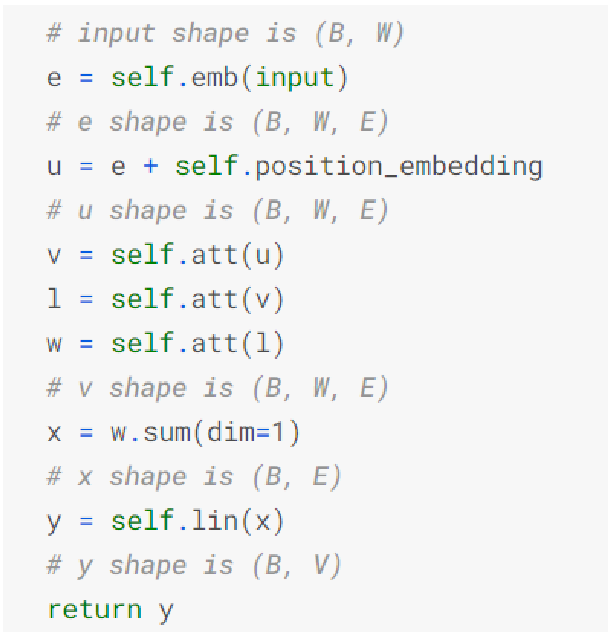
```python
def forward(self, input):

    # 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 = w.sum(dim=1)
    # x shape is (B, E)
    v = torch.reshape(u, (u.shape[0],u.shape[1]*u.shape[2]))

    y = self.lin1(v)
    w = self.tanh(y)

    output = self.lin2(w)
    output = self.softmax(output)

    return output
```

Figure 3: *Feed Forward Neural Network Code.*



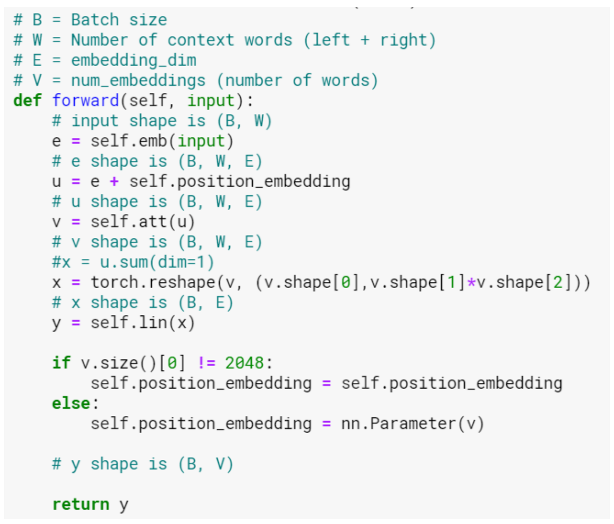
```python
# B = Batch size
# W = Number of context words (left + right)
# E = embedding_dim
# V = num_embeddings (number of words)
def forward(self, input):
    # input shape is (B, W)
    e = self.emb(input)
    # e shape is (B, W, E)
    u = e + self.position_embedding
    # u shape is (B, W, E)
    w = torch.split(u,int(self.embedding_dim/4),2)

    w1 = self.att(w[0])
    w2 = self.att(w[1])
    w3 = self.att(w[2])
    w4 = self.att(w[3])

    x = torch.cat((w1,w2,w3,w4),2)


    # v shape is (B, W, E)
    #x = u.sum(dim=1)
    x = torch.reshape(x, (x.shape[0],x.shape[1]*x.shape[2]))
    # x shape is (B, E)
    y = self.lin(x)

    # y shape is (B, V)

    return y
```

Figure 5: *Multi-Head Attention Code*



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

Figure 4: *Increase the number of TransformerLayers Code.*



```python
# 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 = u.sum(dim=1)
    x = torch.reshape(v, (v.shape[0],v.shape[1]*v.shape[2]))
    # x shape is (B, E)
    y = self.lin(x)

    if v.size()[0] != 2048:
        self.position_embedding = self.position_embedding
    else:
        self.position_embedding = nn.Parameter(v)

    # y shape is (B, V)

    return y
```

Figure 6: *Sharing input / output Embeddings Code*