LAB 2

LANGUAGE MODELING

POE

DATA SCIENCE AND ENGINEERING
UNIVERSITAT POLITÈCNICA DE CATALUNYA

Adrián Cerezuela Hernández - 48222010A

Ramon Ventura Navarro - 21785256R

March-April 2024

1 Introduction and baseline model

Language modeling is crucial in understanding and processing natural language. It helps predict the next word in a sequence based on the words before and after it. In this assignment, we will focus on a specific type of language modeling called non-causal language modeling. Here, the task is to predict the middle word in a sequence given the words around it.

Our main goal is to study, develop and compare the effectiveness of different model architectures for non-causal language modeling. We will evaluate these models based on key metrics such as training accuracy, validation accuracy and loss. By examining the strengths and limitations of each architecture, we aim to gain insights into their applicability and efficacy in capturing complex linguistic patterns.

We have been given a baseline model, which we will try to improve by trying different things such as:

- Experimenting with a Feedforward Neural Network Language Model (without attention)
- Scaling up by adding more Transformer Layers
- Implementing TransformerLayer with multi-head attention
- Investigating domain adaptation techniques
- Exploring shared input/output embedding approaches
- Conducting hyperparameter optimization

Before we begin exploring other architectures, let's understand how the baseline model works. As we can see in the following images, this is the architecture of our starting point.

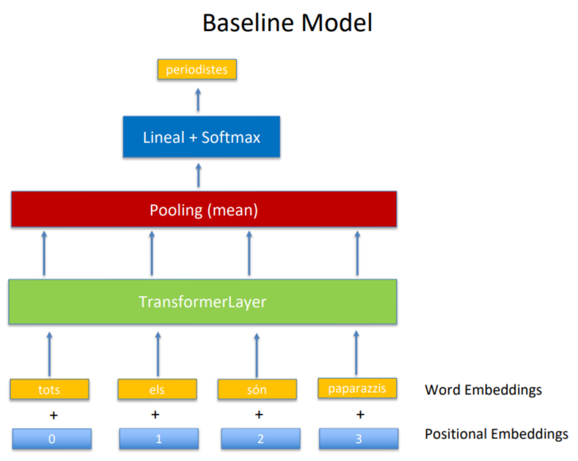


Figure 1: Baseline Model

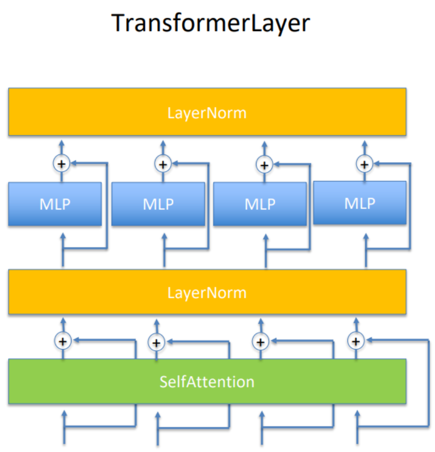


Figure 2: Transformer Layer

This first model takes a group of words around the word we want to predict. Then, these words are turned into embeddings, obtaining an embedding matrix. This matrix is passed as the input of the Transformer Layer. Inside this layer, there's a Self-Attention mechanism that calculates scores showing how much each word in the sequence is related to the others. After that, the output is normalized using layer normalization and fed into a MultiLayer Perceptron. Finally, we can find another normalization step and a series of steps including pooling, linear transformation, and softmax activation help in generating the probability distribution for predicting the central word.

We assess the performance of the models using key metrics such as training accuracy, training loss, validation accuracy, and validation loss. The table below presents the metrics obtained during model training and validation.

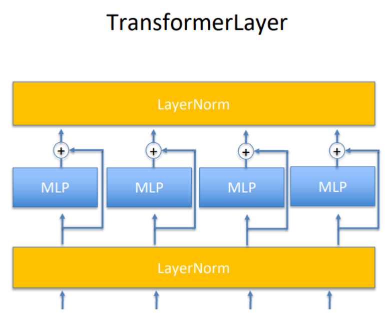
Metrics	**Accuracy (%)**	**Loss**	**Parameters**	**Time**
Training	46.6	3.06	51729664	13042
Validation (Wikipedia)	45.6	3.20		
Validation (El Periódico)	34.9	4.13		

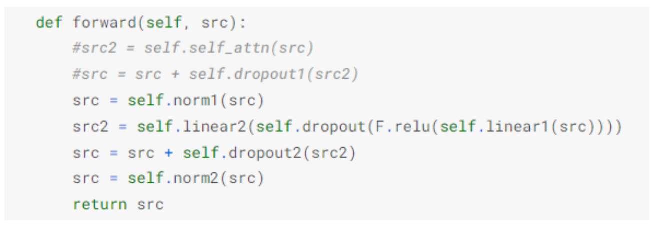
After having reviewed the baseline model let's try to improve it with the measures explained earlier.

2 Model Trials

2.1 Feedforward Neural Network Language Model

The first architecture is a Feedforward Neural Network (FNN) Language Model without incorporating attention mechanisms. Below can be seen the changes made:





```python
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
```

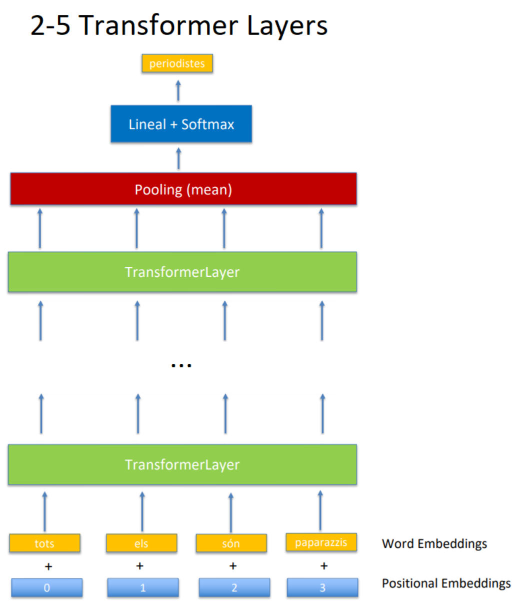
After its implementation, and as we initially could expect, the results indicate that this model with no attention performs poorer than the baseline model.

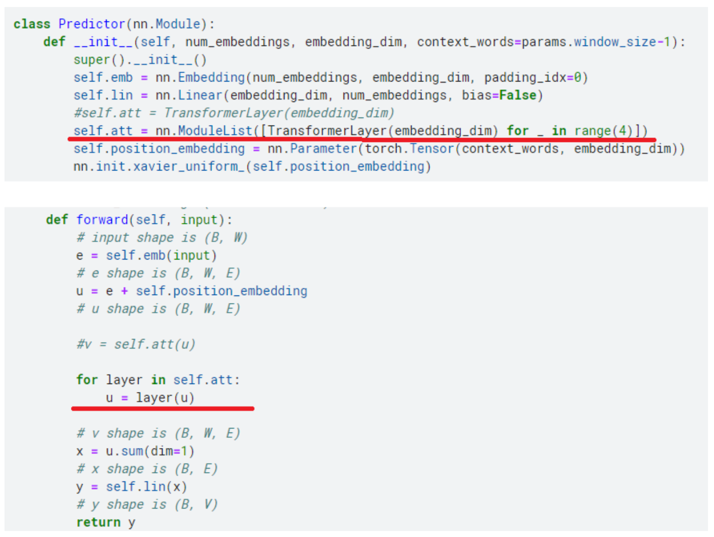
Model	Metrics	Accuracy (%)	Loss	Parameters	Time (s)
Baseline	Training	46.6	3.06	51729664	13042
	Validation (Wikipedia)	45.6	3.20		
	Validation (El Periódico)	34.9	4.13		
FNN	Training	44.6	3.22	51729664	12134
	Validation (Wikipedia)	43.4	3.39		
	Validation (El Periódico)	32.7	4.31		

Looking at the provided metrics, we observe that the FNN model without attention achieves lower accuracies and higher losses. This highlights how important are attention mechanisms to help the model focus on important contextual information, making it better at predicting the next word in a sequence and improving the model's ability to understand and work with complex language. Despite the model's simplicity, it serves as a useful starting point for comparison.

2.2 Increasing the number of Transformer Layers

Next up, in our efforts to improve non-causal language modeling, we have focused on adding more Transformer Layers to our baseline model to boost accuracy and reduce loss, so they count with attention mechanisms. Below is shown the new architecture and how to increase the number of layers through changing the Predictor class and the forward method (4-layer example):





```python
class Predictor(nn.Module):
    def __init__(self, num_embeddings, embedding_dim, context_words=params.window_size-1):
        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 = nn.ModuleList([TransformerLayer(embedding_dim) for _ in range(4)])
        self.position_embedding = nn.Parameter(torch.Tensor(context_words, embedding_dim))
        nn.init.xavier_uniform_(self.position_embedding)
```

```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)

        for layer in self.att:
            u = layer(u)

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

The metrics for the models studied in this section are the following:

Model	Metrics	Accuracy (%)	Loss	Parameters	Time (s)
2 Transformer Layers	Training	47.2	3.00	52256768	14600
	Validation (Wikipedia)	46.5	3.11		
	Validation (El Periódico)	36.1	4.01		
3 Transformer Layers	Training	47.6	2.98	52783872	16195
	Validation (Wikipedia)	46.9	3.08		
	Validation (El Periódico)	36.4	3.98		
4 Transformer Layers	Training	48.0	2.96	53310976	17819
	Validation (Wikipedia)	46.8	3.08		
	Validation (El Periódico)	36.4	3.97		
5 Transformer Layers	Training	48.0	2.96	53838080	19404
	Validation (Wikipedia)	47.2	3.07		
	Validation (El Periódico)	36.5	3.96		

As we added more Transformer Layers, ranging from two to five, we have noticed improvements in accuracy and loss. Among these, the 5-layer model stood out for its better performance in validation tests across different datasets. However, according to our own arbitrary criteria, we've chosen to go with the 4-layer model instead of the 5-layer one. The main reason is that while the 5-layer model shows some slight improvements in the metrics, it takes significantly longer to run (around 19400s, versus 17800s for 4 layers), which affects severely to the time execution of posterior models that will count with more than one Transformer Layer. We believe that finding the right balance between model complexity and practicality is crucial due to having limited GPU execution time.

After choosing the best model for this section, this is the update of the global metrics table:

Model	Metrics	Accuracy (%)	Loss	Parameters	Time (s)
Baseline	Training	46.6	3.06	51729664	13042
	Validation (Wikipedia)	45.6	3.20		
	Validation (El Periódico)	34.9	4.13		
FNN	Training	44.6	3.22	51729664	12134
	Validation (Wikipedia)	43.4	3.39		
	Validation (El Periódico)	32.7	4.31		
4 Transformer Layers	Training	48.0	2.96	53310976	17819
	Validation (Wikipedia)	46.8	3.08		
	Validation (El Periódico)	36.4	3.97		

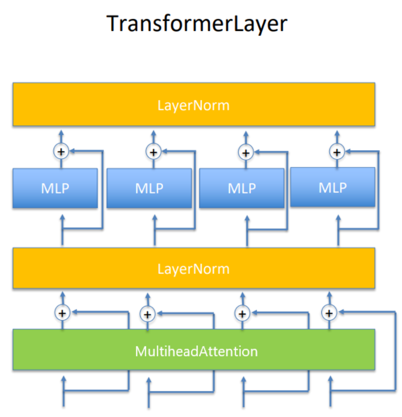
We can observe a notable improvement in both training and validation accuracy as we increase the number of Transformer Layers from the baseline model. Furthermore, in addition with the reduction in loss metrics (prediction are more precise and closer to the ground truth) we reach to the conclusion that the additional layers, and consequently parameters, allow the model to capture more intricate patterns and dependencies within the input data, and therefore enhance its predictive performance.

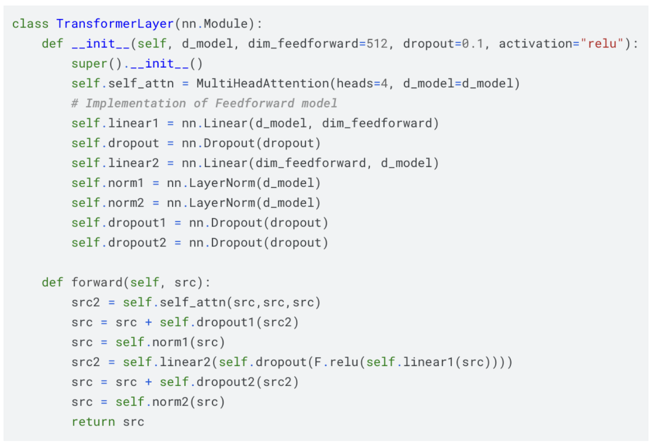
2.3 TransformerLayer with multi-head attention

From now on, both in this section and in the following ones, all proposed changes will be implemented with 4 Transformer Layers.

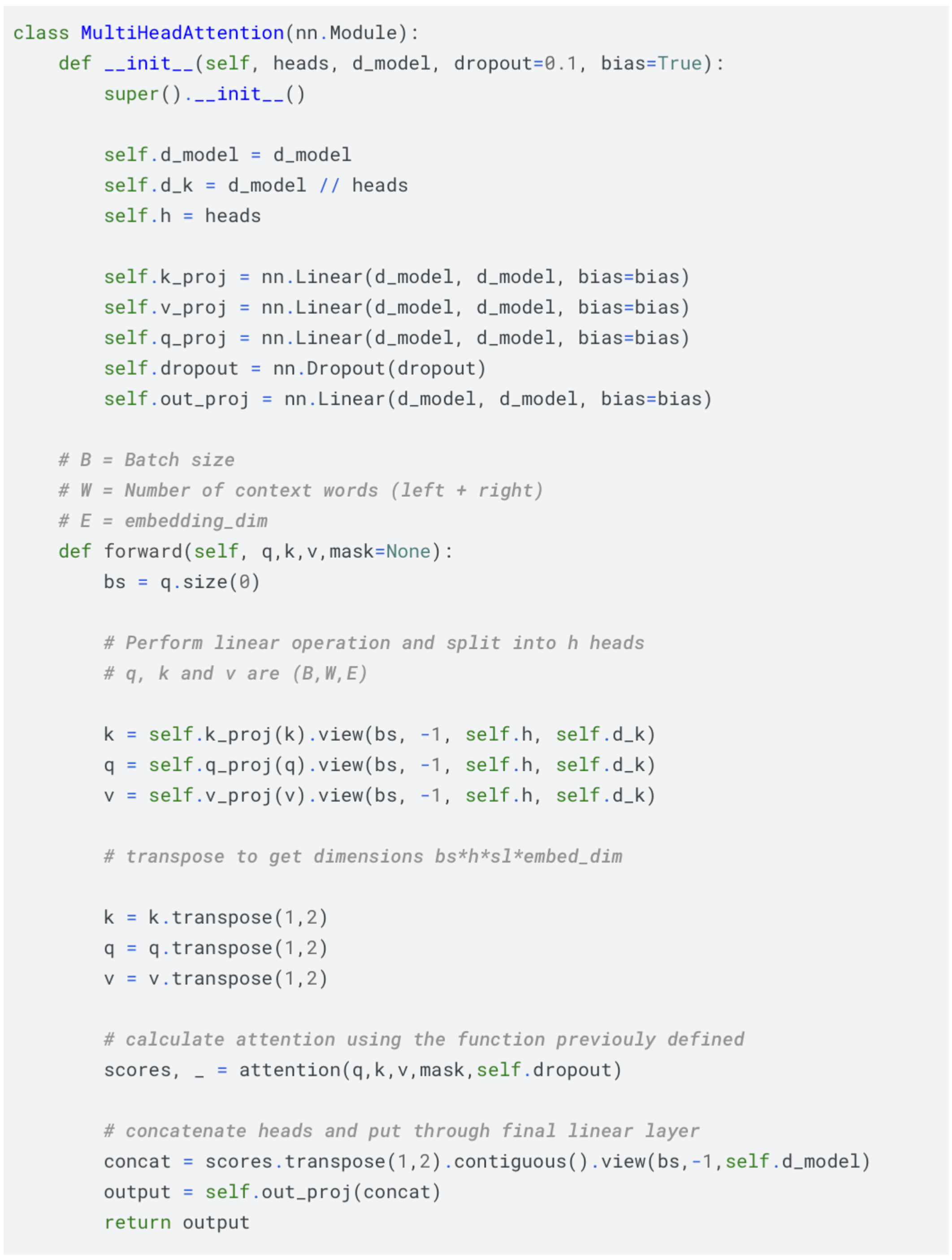
As mentioned before, we've tried to make the TransformerLayers better by adding Multi-head to the Self-attention part. This way, we hope the model will work better and be more flexible.

To do this, we made a new class. You can see it below. In this class, we tell it how many heads to use, and then we divide the embedding dimension by that number to get the head dimension. Also, we made the q, k, v vectors smaller, matching the head dimension:





```
class TransformerLayer(nn.Module):
    def __init__(self, d_model, dim_feedforward=512, dropout=0.1, activation="relu"):
        super().__init__()
        self.self_attn = MultiHeadAttention(heads=4, d_model=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)
        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 MultiHeadAttention(nn.Module):
    def __init__(self, heads, d_model, dropout=0.1, bias=True):
        super().__init__()

        self.d_model = d_model
        self.d_k = d_model // heads
        self.h = heads

        self.k_proj = nn.Linear(d_model, d_model, bias=bias)
        self.v_proj = nn.Linear(d_model, d_model, bias=bias)
        self.q_proj = nn.Linear(d_model, d_model, bias=bias)
        self.dropout = nn.Dropout(dropout)
        self.out_proj = nn.Linear(d_model, d_model, bias=bias)

    # B = Batch size
    # W = Number of context words (left + right)
    # E = embedding_dim
    def forward(self, q,k,v,mask=None):
        bs = q.size(0)

        # Perform linear operation and split into h heads
        # q, k and v are (B,W,E)

        k = self.k_proj(k).view(bs, -1, self.h, self.d_k)
        q = self.q_proj(q).view(bs, -1, self.h, self.d_k)
        v = self.v_proj(v).view(bs, -1, self.h, self.d_k)

        # transpose to get dimensions bs*h*sl*embed_dim

        k = k.transpose(1,2)
        q = q.transpose(1,2)
        v = v.transpose(1,2)

        # calculate attention using the function previouly defined
        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_proj(concat)
        return output
```

Once shown how the code has been modified, metrics are displayed below. An arbitrary choice has been made to default to 4 heads in the attention definition process.

Model	**Metrics**	**Accuracy (%)**	**Loss**	**Parameters**	**Time (s)**
Baseline	Training	46.6	3.06	51729664	13042
	Validation (Wikipedia)	45.6	3.20		
	Validation (El Periódico)	34.9	4.13		
FNN	Training	44.6	3.22	51729664	12134
	Validation (Wikipedia)	43.4	3.39		
	Validation (El Periódico)	32.7	4.31		
4 Transformer Layers	Training	48.0	2.96	53310976	17819
	Validation (Wikipedia)	46.8	3.08		
	Validation (El Periódico)	36.4	3.97		
Multi-Head Attention	Training	48.1	2.94	53310976	18713
	Validation (Wikipedia)	47.5	3.04		
	Validation (El Periódico)	37.0	3.94		

We can observe that even though we use Multi-Head instead of Self-Attention, the number of parameters remains the same, while the validation metrics are slightly higher compared to the previous model. Additionally, it can be noted that the execution time does not deviate significantly from what we had with a Self-Attention mechanism. Based on this, we conclude that a model with Multi-Head Attention will perform better, and therefore, it will be the attention mechanism used in subsequent models after this section.

2.4 Domain adaptation

The next step in the search for the best model for this practice involves adapting the domain in which the training process is executed. In addition to using the vocabulary generated in the *Vocabulary()* class, in this step, a small sample of the dataset from *El Periódico* is added. Therefore, by introducing more training data as a sample of the validation set and thus being able to expect that it resembles the vocabulary that will be used to validate the model, the expectation is that the model performs better than the baseline one.

It has been verified that the dataset from *El Periódico* consists of 20.000 instances. From the total, a random sample of 10% will be taken, which is *num_samples = 2000*. It is taken as an hyperparameter at the beginning of the notebook.

Below you can find firstly how the code has been modified in the epoch loop, and secondly the metric board displayed, as done before.

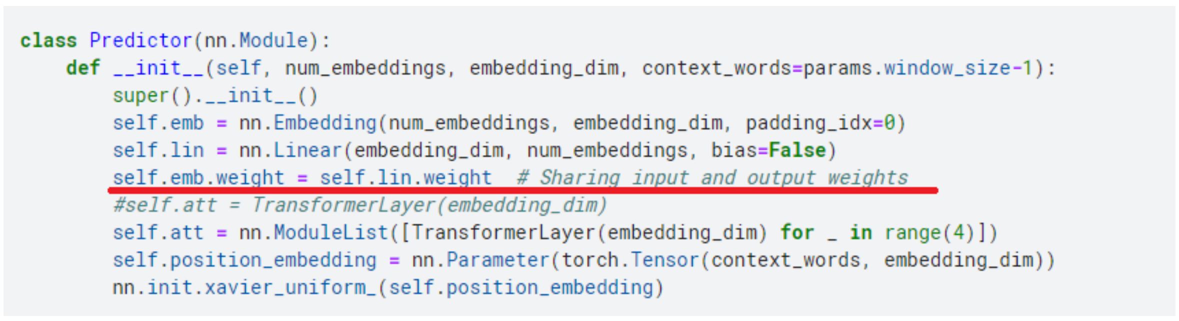
```
import random

optimizer = torch.optim.Adam(model.parameters())
criterion = nn.CrossEntropyLoss(reduction='sum')

train_accuracy = []
wiki_accuracy = []
valid_accuracy = []
for epoch in range(params.epochs):
    # Add a random subsample of 'El Periódico' validation data to the training subset
    random_indices = random.sample(range(len(valid_x)), k=params.num_samples)
    train_x = np.concatenate((data[0][0], valid_x[random_indices]), axis=0)
    train_y = np.concatenate((data[0][1], valid_y[random_indices]), axis=0)

    acc, loss = train(model, criterion, optimizer, train_x, train_y, params.batch_size, device, log=True)
    train_accuracy.append(acc)
    print(f'| epoch {epoch:03d} | train accuracy={acc:.1f}%, train loss={loss:.2f}')
    acc, loss = validate(model, criterion, data[1][0], data[1][1], params.batch_size, device)
    wiki_accuracy.append(acc)
    print(f'| epoch {epoch:03d} | valid accuracy={acc:.1f}%, valid loss={loss:.2f} (wikipedia)')
    acc, loss = validate(model, criterion, valid_x, valid_y, params.batch_size, device)
    valid_accuracy.append(acc)
    print(f'| epoch {epoch:03d} | valid accuracy={acc:.1f}%, valid loss={loss:.2f} (El Periódico)')

# Save model
torch.save(model.state_dict(), params.modelname)
```

Model	Metrics	Accuracy (%)	Loss	Parameters	Time (s)
Baseline	Training	46.6	3.06	51729664	13042
	Validation (Wikipedia)	45.6	3.20		
	Validation (El Periódico)	34.9	4.13		
FNN	Training	44.6	3.22	51729664	12134
	Validation (Wikipedia)	43.4	3.39		
	Validation (El Periódico)	32.7	4.31		
4 Transformer Layers	Training	48.0	2.96	53310976	17819
	Validation (Wikipedia)	46.8	3.08		
	Validation (El Periódico)	36.4	3.97		
Multi-Head Attention	Training	48.1	2.94	53310976	18713
	Validation (Wikipedia)	47.5	3.04		
	Validation (El Periódico)	37.0	3.94		
Domain Adaptation	Training	48.0	2.96	53310976	17813
	Validation (Wikipedia)	46.8	3.08		
	Validation (El Periódico)	36.5	3.94		

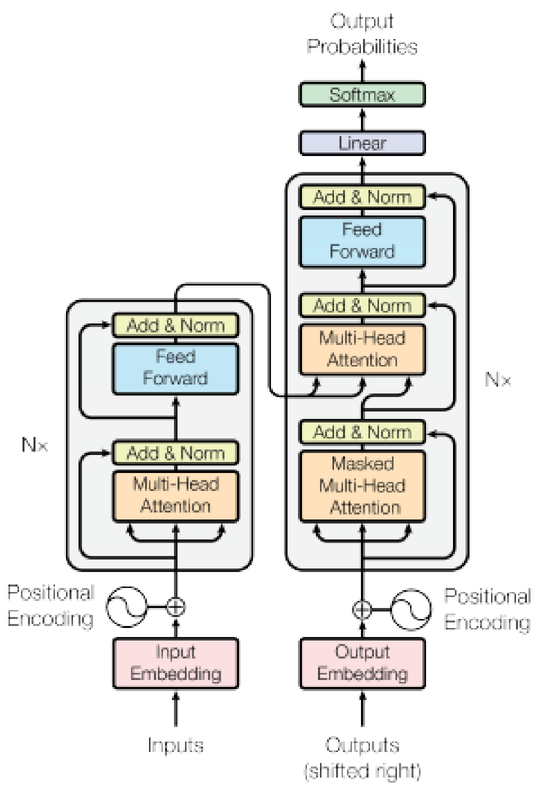
As shown in the table, this model performs slightly worse than the model without domain adaptation, despite taking a similar amount of time and having the same number of parameters. Therefore, this model will be automatically discarded, and we will proceed with the one seen in the previous section.

2.5 Sharing input/output embeddings

Another approach worth studying is sharing input and output embeddings. This means using the same set of weights for multiple parts of the model. Considering the 4 Transformer Layers Multi-Head Attention is the best model at the moment, we have implemented the mentioned approach to it with the help of the following changes in the Predictor class:



```
class Predictor(nn.Module):
    def __init__(self, num_embeddings, embedding_dim, context_words=params.window_size-1):
        super().__init__()
        self.emb = nn.Embedding(num_embeddings, embedding_dim, padding_idx=0)
        self.lin = nn.Linear(embedding_dim, num_embeddings, bias=False)
        self.emb.weight = self.lin.weight  # Sharing input and output weights
        #self.att = TransformerLayer(embedding_dim)
        self.att = nn.ModuleList([TransformerLayer(embedding_dim) for _ in range(4)])
        self.position_embedding = nn.Parameter(torch.Tensor(context_words, embedding_dim))
        nn.init.xavier_uniform_(self.position_embedding)
```



Model	Metrics	Accuracy (%)	Loss	Parameters	Time (s)
Baseline	Training	46.6	3.06	51729664	13042
	Validation (Wikipedia)	45.6	3.20		
	Validation (El Periódico)	34.9	4.13		
FNN	Training	44.6	3.22	51729664	12134
	Validation (Wikipedia)	43.4	3.39		
	Validation (El Periódico)	32.7	4.31		
4 Transformer Layers	Training	48.0	2.96	53310976	17819
	Validation (Wikipedia)	46.8	3.08		
	Validation (El Periódico)	36.4	3.97		
Multi-Head Attention	Training	48.1	2.94	53310976	18713
	Validation (Wikipedia)	47.5	3.04		
	Validation (El Periódico)	37.0	3.94		
Domain Adaptation	Training	48.0	2.96	53310976	17813
	Validation (Wikipedia)	46.8	3.08		
	Validation (El Periódico)	36.5	3.94		
Sharing Input/Output Embedding (MH Att.)	Training	50.0	2.73	27710464	18259
	Validation (Wikipedia)	48.4	2.95		
	Validation (El Periódico)	37.8	3.87		

We see that this approach has several benefits. Firstly, it reduces to almost half the total number of parameters in the model. Instead of having separate sets of parameters for input and output embeddings, the same set is used for both, saving memory and computational resources, also making the model converge faster.

Secondly, sharing parameters allows the model to transfer knowledge across different parts of the network. By using the same learned representations for input and output, the model can capture relationships and patterns in the data more effectively. This clearly leads to better generalization and performance, counting with the best overall metrics of all the models considered in this study. We can see an significant improvement in all of the metrics, with an increase of the accuracy on the validation set (El Periódico) of 0.8% respect the second best model (Multi-Head Attention).

Additionally, we also considered trying this approach using Self Attention instead of Multi-Head Attention, even though the latter model gave better results. The metrics found were:

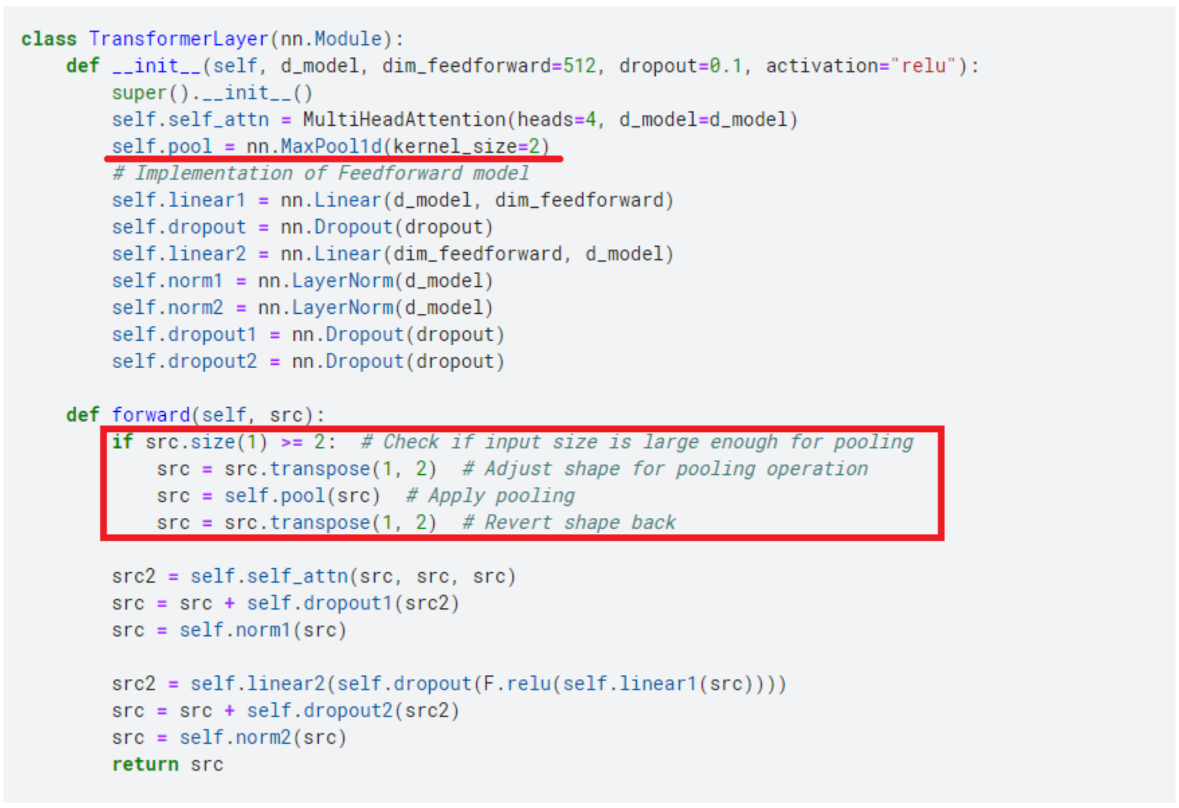
Model	Metrics	Accuracy (%)	Loss	Parameters	Time (s)
Sharing Input/Output Embedding (MH Att.)	Training	50.0	2.73	27710464	18259
	Validation (Wikipedia)	48.4	2.95		
	Validation (El Periódico)	37.8	3.87		
Sharing Input/Output Embedding (Self Att.)	Training	49.9	2.74	27710464	17337
	Validation (Wikipedia)	48.1	2.97		
	Validation (El Periódico)	37.5	3.87		

Which reaffirmed our first intuition, making us discard the model with Self Attention as, even though it performed faster, the results seem a little bit worse.

2.6 Hyperparameter optimization

Once we have the model architecture defined, we will perform a small tuning of certain hyperparameters that could improve our results. Our initial expectation with this testing phase is to observe how the model metrics change based on these adjustments. The modification will be done simply by changing the corresponding value in the initial variable *params*.

The hyperparameters to tune will be the *embedding size*, the *learning rate* with different *optimizers*, the *batch size*, and additionally, experimenting with adding a *max pooling* layer with a kernel size of 2 to the model. This last modification is shown below. The results can be found following the modified code:



```python
class TransformerLayer(nn.Module):
    def __init__(self, d_model, dim_feedforward=512, dropout=0.1, activation="relu"):
        super().__init__()
        self.self_attn = MultiHeadAttention(heads=4, d_model=d_model)
        self.pool = nn.MaxPool1d(kernel_size=2)
        # 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):
        if src.size(1) >= 2:  # Check if input size is large enough for pooling
            src = src.transpose(1, 2)  # Adjust shape for pooling operation
            src = self.pool(src)  # Apply pooling
            src = src.transpose(1, 2)  # Revert shape back

        src2 = self.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
```

Model	Metrics	Accuracy (%)	Loss	Parameters	Time (s)
Sharing Input/Output Embedding (MH Att.)	Training	50.0	2.73	27710464	18259
	Validation (Wikipedia)	48.4	2.95		
	Validation (El Periódico)	37.8	3.87		
Embedding Dim 128	Training	48.4	2.90	13594112	11898
	Validation (Wikipedia)	47.2	3.05		
	Validation (El Periódico)	36.9	3.95		
Embedding Dim 512	Training	49.9	2.72	57516032	32236
	Validation (Wikipedia)	48.1	2.98		
	Validation (El Periódico)	37.0	3.93		
Adam lr=0.005	Training	6.1	7.36	27710464	18244
	Validation (Wikipedia)	6.2	7.40		
	Validation (El Periódico)	4.4	7.55		
RMSProp lr=0.005	Training	15.7	6.12	27710464	18099
	Validation (Wikipedia)	16.9	6.07		
	Validation (El Periódico)	13.4	6.44		
Max Pooling (kernel-size = 2)	Training	48.1	2.92	27710464	13279
	Validation (Wikipedia)	46.9	3.07		
	Validation (El Periódico)	36.4	3.97		
Batch Size 4096	Training	50.5	2.68	27710464	17269
	Validation (Wikipedia)	49.0	2.90		
	Validation (El Periódico)	38.1	3.82		

Firstly, and as the most notable finding, we can see that the optimizer and learning rate provided by default in the initial notebook yield the best results by a large margin compared to others tested. Therefore, the model will use an *Adam* optimizer with a learning rate of 0.001, which is the default value in the *torch.optim.Adam* function.

Next, we observe that although increasing the *embedding size* increases the number of parameters and execution time, a model with a larger embedding size provides better results. However, the tradeoff between time, number of parameters, and prediction accuracy is not significant enough to favor a larger embedding size.

Regarding the *batch size*, it can be seen that increasing the batch size tends to improve computational efficiency, as it allows processing more samples at once, which could accelerate the training process. We also see a slight improvement in metrics obtained, without even increasing the number of parameters. It can be concluded that this model generalizes slightly better than the one taken as the baseline.

Finally, when including a max pooling layer with kernel-size = 2, we observe that the prediction accuracy is slightly lower. The execution time is also significantly longer. However, due to the loss of accuracy, this model cannot be considered better than the one taken at the beginning of the section.

3 Final Conclusions

After exploring different architectures for the proposed model to solve this problem, we have opted for one in which the embeddings are shared between input and output, with the following hyperparameters:

embedding_dim	256	**Transformer Layers**	4
window_size	7	**Attention**	*Multi-Head*
batch_size	4096	**Domain Adaptation**	No
epochs	4		
optimizer	*Adam*		
learning rate	0.001		

Thus obtaining the following metrics:

Metrics	**Accuracy (%)**	**Loss**
Training	50.5	2.68
Validation (Wikipedia)	49.0	2.90
Validation (El Periódico)	38.1	3.82

The next model that presents better results would be this same one with a lower batch size, as proposed in the initial notebook of the practice.