Assignment 2

Catalan Word Vectors. Language models

Course: Spoken and Written Language Processing

(POE-GCED)

Jordi Aguilar

Miquel Escobar

March 31, 2020

Contents

1 Introduction

In this homework assignment we have implemented, tested and analyzed a very common architecture used for NLP: the Transformer. The Transformer is a simple yet powerful model to deal with text-based problems such as machine translation, text recognition and text prediction, among others. It avoids the use of complicated and costly recurrences or convolutions, and sets the beginning of a state-of-the-art tool for text processing with faster convergence and better results.

To test the variations of the model, we have modified the given baseline **Kaggle** notebook with the corresponding contributions. We have also proposed models with alternatives architectures tp the Transformer, such as a MLP. These contributions are described and minutiously explained in section **2. Models**.

The analyzed results include a comparison of accuracy, loss, training time and the number of parameters between the different versions and/or models. These comparisons appear in the form of tables and visualizations in order to provide the reader an easier understanding of the obtained results.

The task that the models have been trained for is to predict the missing word in the middle of a sentence (of fixed length, and in Catalan language). The training samples have been extracted from *Wikipedia* and the testing samples from the well-known Spanish newspaper *El Periódico*.

2 Models

2.1 Base model

The base model that we have been given is a Transformer architecture adapted to predict the word of the middle of a sentence. The default input consists of 4 context words, and these are the steps that it follows:

1. **Words to embeddings**: an embedding is assigned to each word (initialized randomly with the us of `torch.nn.Embedding` class). Then a position embedding is added, which helps the model focus on certain areas.

2. **Transformer Layer**:

   (a) **Masked Self-Attention**: an equation that replicates different versions of the input, and performs matrix multiplications between them, in order to relate words with the context.

   (b) **Feed-Forward Network**: A set of fully connected layers to ”understand” the Self-Attention output.

3. **Reduction and Prediction**: the sum of the embeddings, which we could call the ”final embedding”, is passed to a fully connected layer which determines the most likely word out of all the observed ones in the training set.

An schematic approximation of the architecture of the Base model is the following:

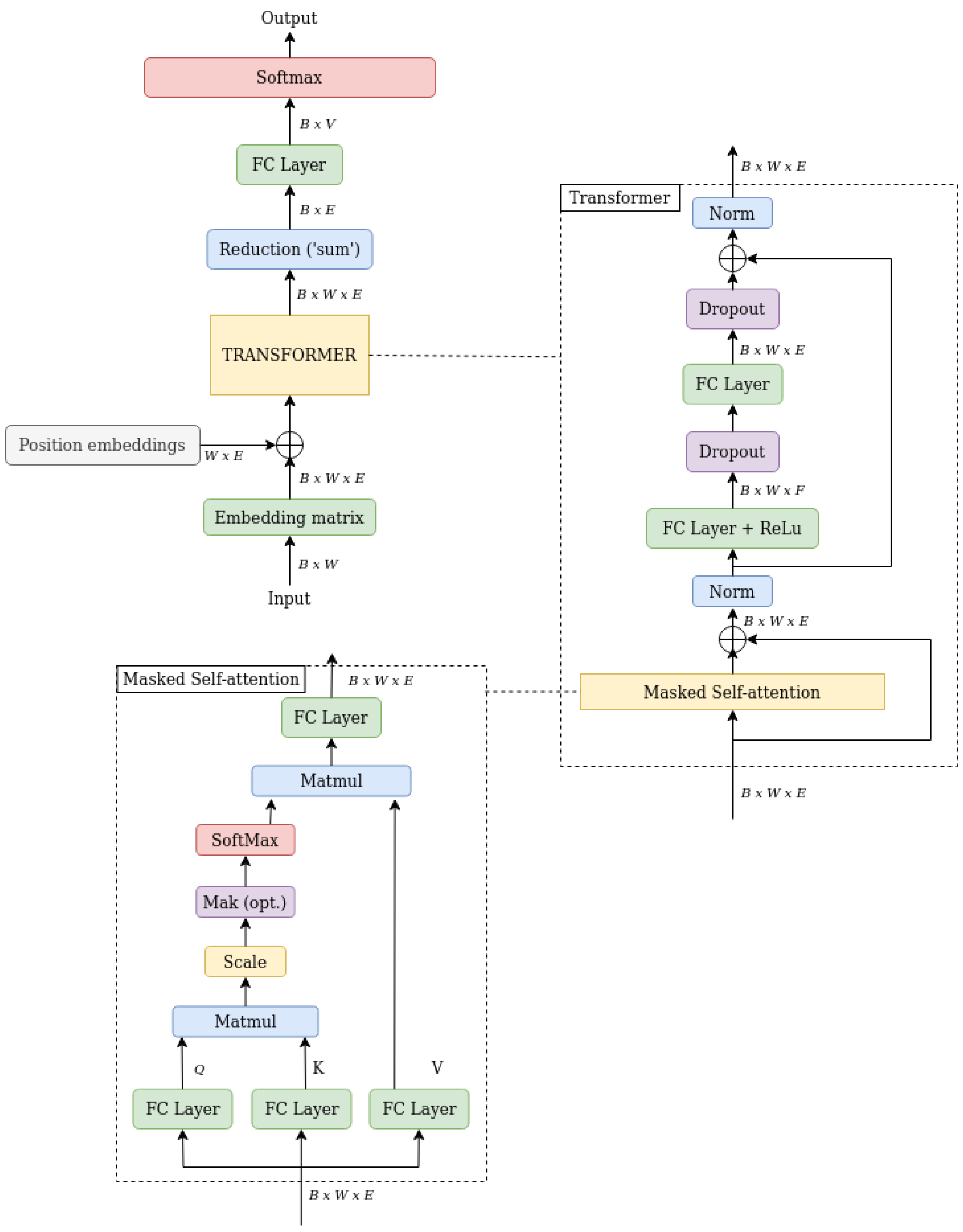


Figure 1: Architecture schema of the Base model

2.2 Increasing the number N of `TransformerLayers`

A way to improve the performance of our model is simply to concatenate N Transformer Layers, one after another.

2.2.1 Baseline architecture

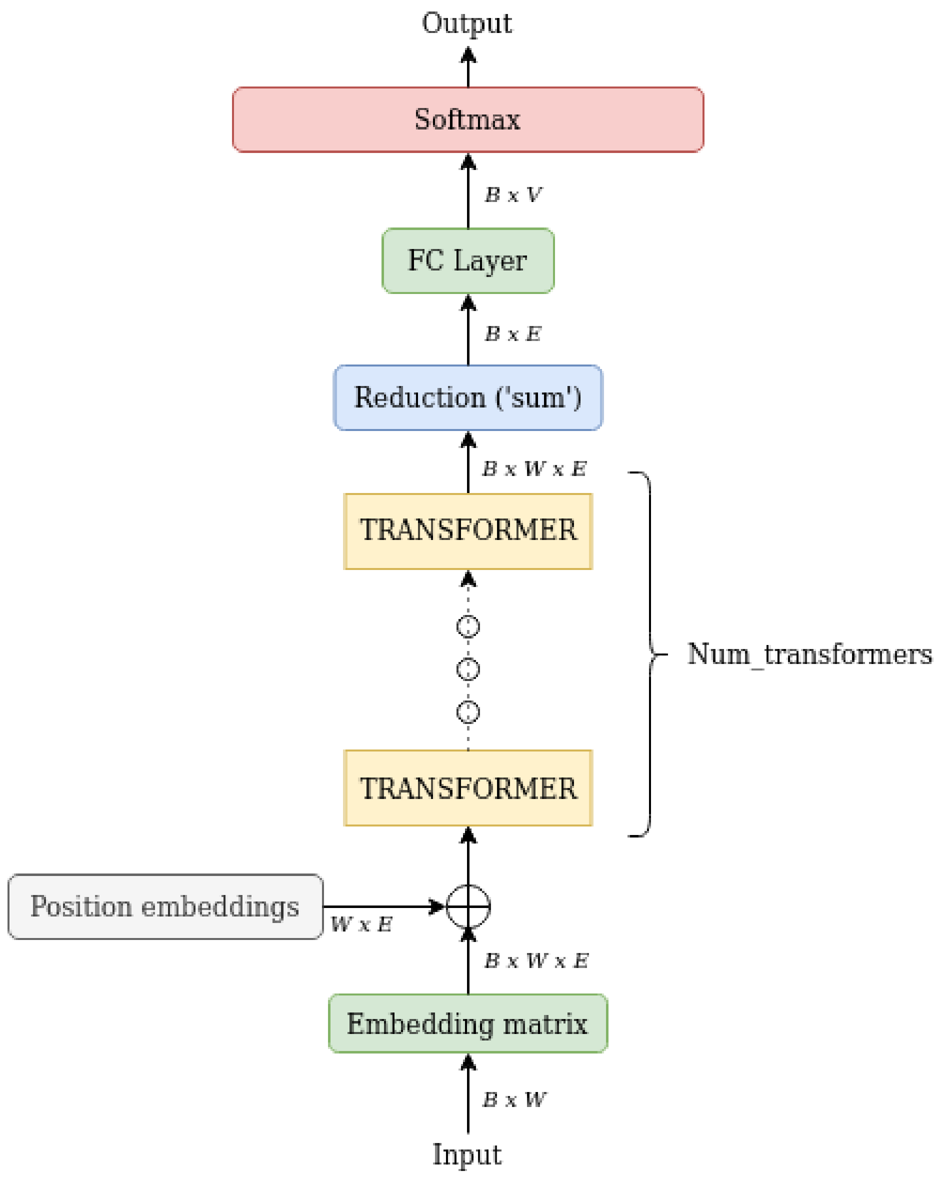


Figure 2: Architecture schema of a model with N transformers

2.2.2 Modified code

```python
class Predictor(nn.Module):
    def __init__(self, num_embeddings, embedding_dim, num_transformers = 1,
    num_heads = 1, 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.transformers = nn.ModuleList([TransformerLayer(embedding_dim,
    num_heads) for i in range(num_transformers)])
        self.position_embedding = nn.Parameter(torch.Tensor(context_words,
    embedding_dim))
        nn.init.xavier_uniform_(self.position_embedding)

    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)
        for t in self.transformers:
            u = t(u)
        # u shape is (B, W, E)
        x = u.sum(dim=1)
        # x shape is (B, E)
        y = self.lin(x)
        return y
```

Listing 1: Model with N `TransformerLayers` implemented using `PyTorch`

2.3 Multilayer perceptron (MLP) over the concatenated input vectors

We could not resist to apply the broadly used Multilayer Perceptron on our data. To do so, the word embeddings have been concatenated into a new vector, which is then applied into a set of fully connected layers.

This model is very simple as it does not take into consideration the context of the words. Even so, we think that is good to compare it with other variations, and it is also pretty fast to train.

2.3.1 Baseline architecture

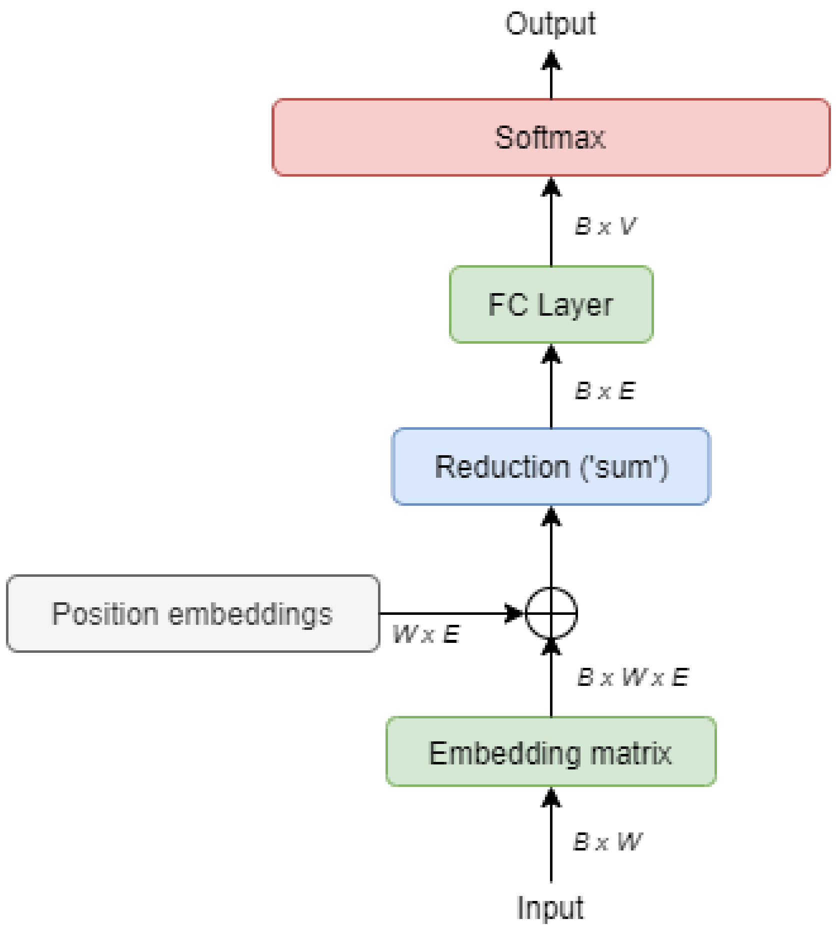


Figure 3: Architecture schema of a model with N transformers

2.3.2 Modified code

```python
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(context_words*embedding_dim, num_embeddings, bias
=False)
        self.position_embedding = nn.Parameter(torch.Tensor(context_words,
embedding_dim))
        nn.init.xavier_uniform_(self.position_embedding)

    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)
        x = u.view(params.batch_size, -1)
        # x shape is (B, WxE)
        y = self.lin(x)
        # y shape is (B, V)
        return y
```

Listing 2: MLP over concatenated embeddings model implemented using `PyTorch`

2.4 `TransformerLayer` with multihead attention

This model is based on a mechanism named ”Multi-head attention” [1]. It helps the model focus on different positions of the embeddings and it performs different Scaled Dot-Products Attention, which hopefully improves the accuracy as it is a more customized fit to the given problem.

2.4.1 Baseline architecture

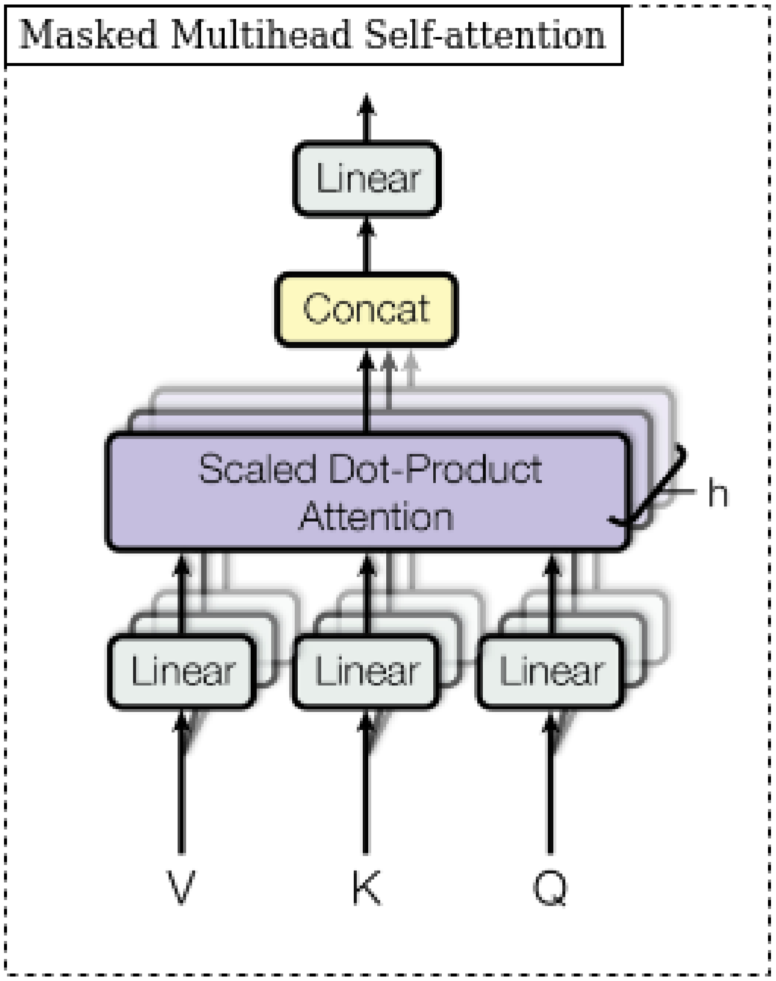


Figure 4: Architecture schema of model with h heads (only Self-Attention chunk)

2.4.2 Modified code

```python
class SelfAttention(nn.Module):
    def __init__(self, d_model, num_heads, dropout = 0.2):
        super().__init__()

        self.d_model = d_model
        self.d_k = d_model // num_heads
        self.h = num_heads

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

    def forward(self, x, mask=None):

        bs = x.size(0)

        # perform linear operation and split into h heads

        k = self.k_proj(x).view(bs, -1, self.h, self.d_k)
        q = self.q_proj(x).view(bs, -1, self.h, self.d_k)
        v = self.v_proj(x).view(bs, -1, self.h, self.d_k)
        #k, q, v, shape is (B, W, H, 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)
        #k, q, v, shape is (B, H, W, D_K)

        scores, _ = attention(q, k, v, self.d_k, mask, self.dropout)
        #scores shape is (B, H, W, D_K)

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

        output = self.out(concat)
        #output shape is (B, W, E)

        return output
```

Listing 3: `TransformerLayer` with multihead attention model implemented using `PyTorch`

2.5 Replacing the `SoftmaxLayer` with `AdaptiveSoftmax`

When dealing with Natural Language Processing problems, we often have to predict one word over an enormous set of words (100.002 words in our case, which is the Catalan language), which implies that there is going to be a very large outputs. This is translated into a very expensive Softmax and a last layer with an enormous size (256×100002). *AdaptiveSoftmax* solve us both problems. Firstly, it performs a Softmax operation only to the most frequent outputs, and to the other outputs only when strictly necessary. Furthermore, it reduces the number of parameters of the less frequent outputs by adding an intermediate layer of smaller size. For example, if 80,000 words are part of the third group, the amount of parameters for the third group will be $(256 \times 16) + (16 \times 80,000) = 1,284,096$ instead of $(256 \times 80,000) = 20,480,000$, which is a reduction of about ≈ 20 times. The loss in terms of accuracy is minimal considering the obtained reduction of computation costs and training time.

2.5.1 Baseline architecture

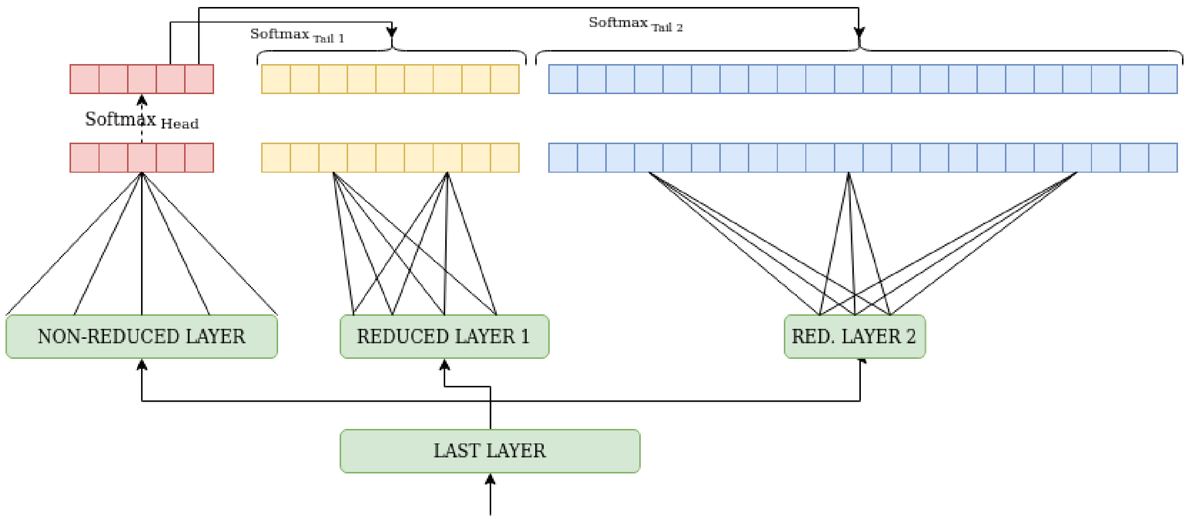


Figure 5: Architecture schema of the "`AdaptiveSoftmaxLayer`" model

2.5.2 Modified code

It is worth noting that due to the computation of the softmax on the *forward* function, we must change the criterion function of `CrossEntropyLoss`, which computes the softmax operatin, to the `NLLLoss`.

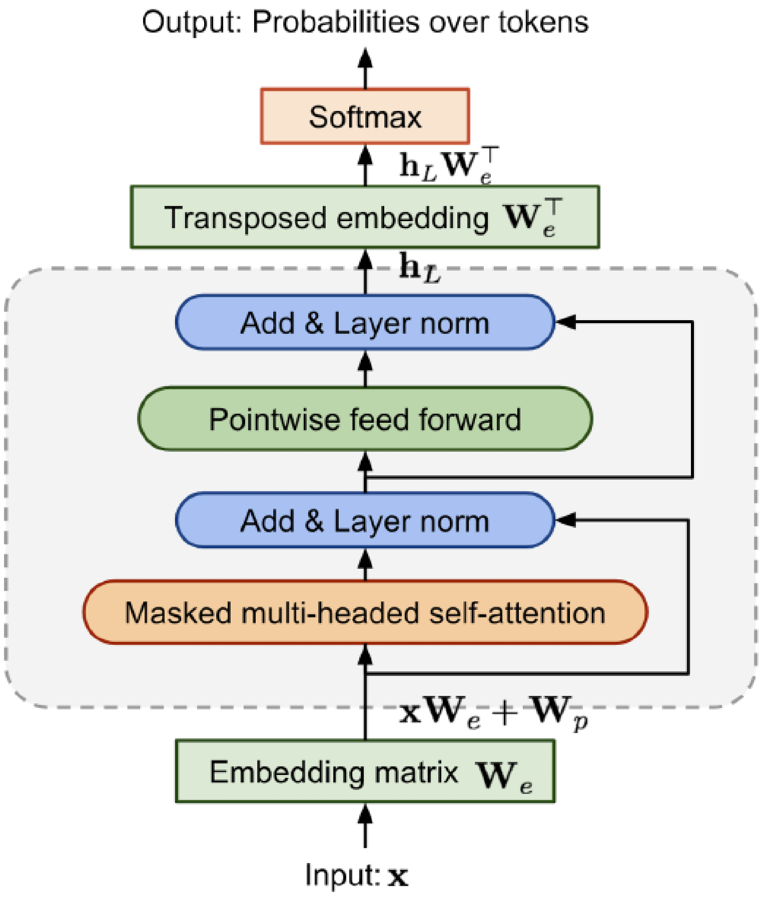
```python
class Predictor(nn.Module):
    def __init__(self, num_embeddings, embedding_dim, num_transformers = 1,
    num_heads = 1, context_words=4):
        super().__init__()
        self.emb = nn.Embedding(num_embeddings, embedding_dim, padding_idx=0)
        self.transformers = nn.ModuleList([TransformerLayer(embedding_dim,
    num_heads) for i in range(num_transformers)])
        self.position_embedding = nn.Parameter(torch.Tensor(context_words,
    embedding_dim))
        nn.init.xavier_uniform_(self.position_embedding)

        self.out = nn.AdaptiveLogSoftmaxWithLoss(embedding_dim, num_embeddings
    , cutoffs=[500, 2000, 8000],div_value=4)

    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)
        for t in self.transformers:
            u = t(u)
        # u shape is (B, W, E)
        x = u.sum(dim=1)
        # x shape is (B, E)

        y = self.out.log_prob(x)
        # y shape is (B, V)
        return y
```

Listing 4: Model with `AdaptiveSoftmaxLayer` implemented using `PyTorch`

2.6 Sharing input & output embeddings

This variation takes advantage of the fact that we have the same number of inputs and outputs. Although we only give to the model 4 indices (by default) indicating the input words, the Embedding matrix has size $(vocab\_dim \times embed\_dim)$. Knowing that the output size is also $Vocab\_dim$, we can transpose the embedding matrix to obtain the size $(embed\_dim \times vocab_d im)$, getting the matrix of a Fully Connected Layer of input size *embed_dim* and output size *vocab_dim*. This reduces considerably the number of parameters and thus, the computation costs and training time, while it remains solid in terms of accuracy.

2.6.1 Baseline architecture



Output: Probabilities over tokens
Softmax
$\mathbf{h}_L \mathbf{W}_e^\top$
Transposed embedding $\mathbf{W}_e^\top$
$\mathbf{h}_L$
Add & Layer norm
Pointwise feed forward
Add & Layer norm
Masked multi-headed self-attention
$\mathbf{x}\mathbf{W}_e + \mathbf{W}_p$
Embedding matrix $\mathbf{W}_e$
Input: $\mathbf{x}$

Figure 6: Architecture schema of the model "Sharing input & output embeddings"

2.6.2 Modified code

```python
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)

    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)
        W = self.emb.weight
        # W shape is (V, E)
        y = torch.matmul(x, W.transpose(0,1))
        # y shape is (B, V)
        return y
```

Listing 5: Sharing input & output embeddings model implemented using `PyTorch`

2.7 Hyperparameters tuning

When we talk about hyperparameter optimization, it often comes to our mind the question *The more the better?*, which insinuates that an increase in the number of layers, parameters, epochs, parameter updates,... is going to improve our model performance.

After reading *Attention is all you need* [2] , where they took a base model and evaluated model variants, we can observe that the BLEU metric on the English-to-German translation is correlated with the total number of parameters. The perplexity is also lower with more parameters. Moreover, to set a new accuracy record they basically went deeper: doubling the size of the embeddings, the size the feed forward layer and the number of heads, increasing the train steps (it lasted 3and a half days to train it) and increasing dropout to avoid overfitting.

With this in mind, we took two of the best performing models and modified the hyperparameters to try to obtain even better results. These are the models with the chosen hyperparameters:

T	H	Embed_dim	d_{ff}	d_k	P_{drop}	Adaptive Softmax	#Params
4	4	512	1024	128	0.2	Yes	65×10^6
2	8	256	2048	128	0.15	No	55×10^6

The evaluation and conclusions of these models are in section **3.5** where all trained models are compared.

3 Models performance evaluation

In this section we show the performance of each of the models by means of tables and visualizations. For visualization and aesthetic purposes, we have ”encoded” the model names into a more compressed form:

- **2T-1H**: model with 2 transformers and no multihead attention.
- **4T-1H**: model with 4 transformers and no multihead attention.
- **4T-1H-A**: model with 4 transformers, no multihead attention and an `AdaptiveSoftmaxLayer`.
- **1T-4H**: model with 1 transformer and multihead attention with 4 heads.
- **4T-4H-A**: model with 4 transformers, multihead attention with 4 heads and an `AdaptiveSoftmaxLayer`.
- **2T-8H**: model with 2 transformers and multihead attention with 8 heads.
- **MLP**: Multilayer Perceptron over the concatenated input vectors.
- **SIOE**: model sharing input & output embeddings.

The plots and tables have been generated by the notebook `models-metrics.ipynb`. The data has been stored during the training of the models. It is important to note that the granularity of the data depends on the metric as for instance, the training accuracy is available for all generated batches while the validation accuracy only for each of the epochs.

3.1 Loss

The loss function is used for the training - it is indeed what allows the application of the backpropagation algorithm. Even though it does not measure the quality of the predictions themselves, it is a good indicator of the progress made during the training.

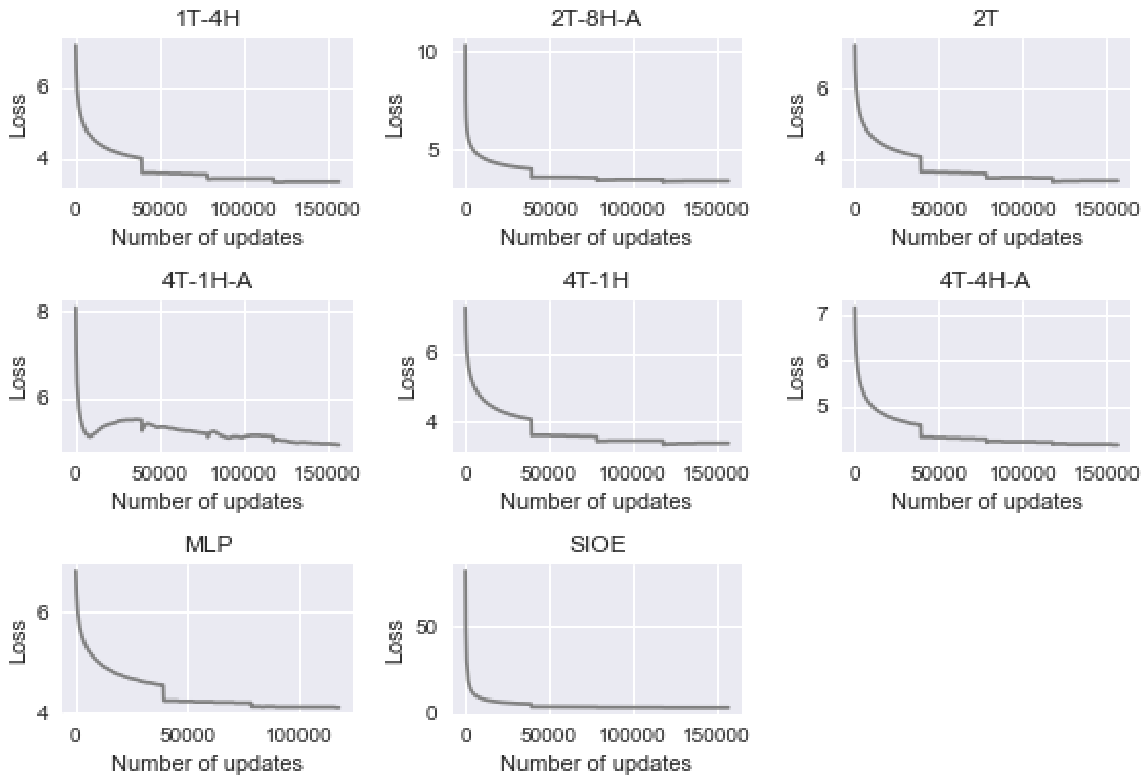


Figure 7: Loss function value evolution during training for each of the models.

In **Figure 7** we observe how most of the models did not reduce the loss function value significantly after the second or third epoch. The **4T-1H-A** model plot stands out as the value does grow in some prolongated periods, which might be indicating that the learning rate l_r was set too high for its training.

On the other side of the spectrum, the **4T-4H-A** model seems to have room for improvement - as the loss function value was still descending significantly in the last epoch - and probably could have been trained for two or three more epochs, resulting in a better model.

3.2 Accuracy

The accuracy indicates the proportion of good predictions made by the model. Of course, as the training advances we expect the accuracy to improve. This is what we observe in all models (for both training and validation accuracy), except for the **MLP** model - which confirms our suspicions that a lower learning rate l_r would have been more appropiate.

It is also worth noting that the accuracy for the *Wikipedia* dataset is always higher than for the *El Periódico* dataset, and that is because, as mentioned earlier in this report, the *Wikipedia* dataset is actually the one used for training.

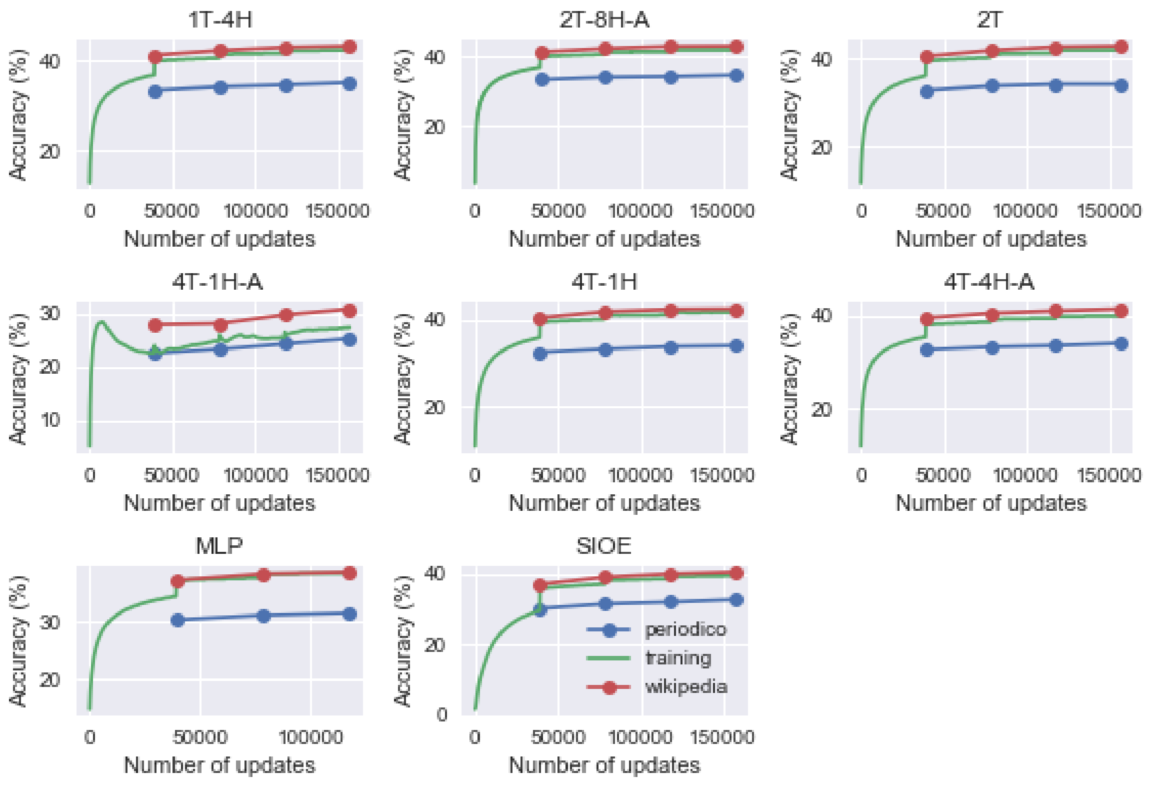


Figure 8: Accuracy value evolution during training for each of the models.

After training the models, we obtained the test accuracy for each of the models (by submitting their predicitons to the *Kaggle* competition). Based on this metric, the best performing model is **4T-1H**, and only $\frac{4}{8}$ models improve the baseline (corresponding to the given model).

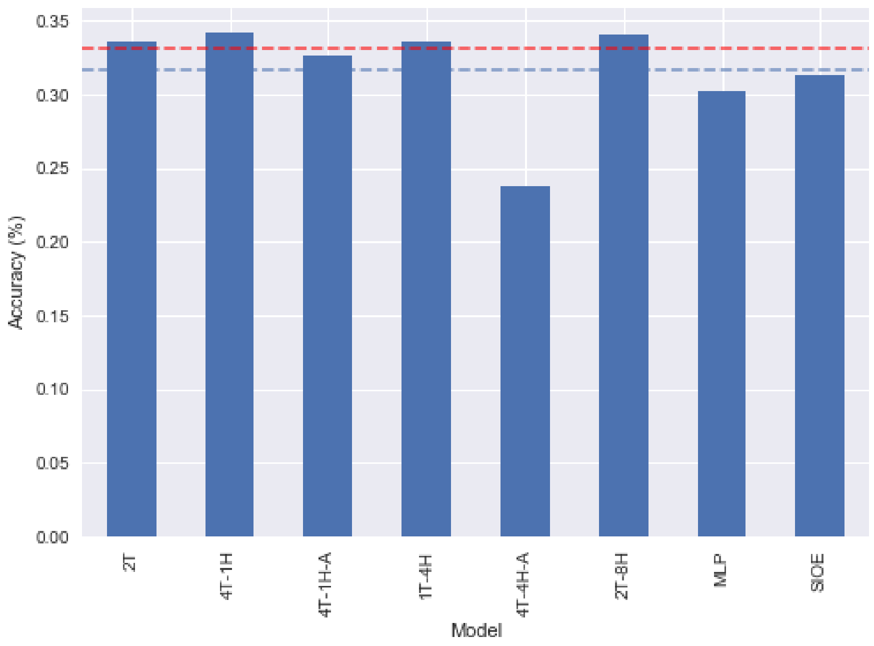


Figure 9: Test accuracy value for each of the models. The dashed blue line indicates the average. The red dashed line indicates the baseline accuracy.

3.3 Training time

The training time helps to identify the computational costs and scalability of the model and its training. A model might perform well with predictions, but might have a huge training duration, which would imply it is not the best fit for a production environment.

In this case, each of the models was trained only once (as we are in a testing/development environment), reason for which it is not a critical metric to choose the best overall-performing model, but it is always interesting to observe what relation there is between the accuracy, performance and training time.

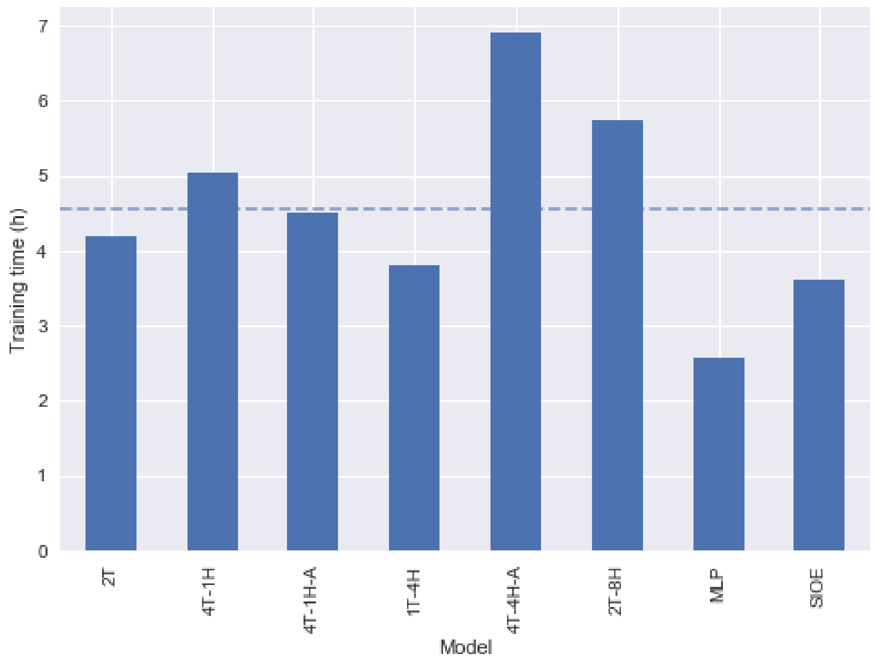


Figure 10: Training time for each of the models. The dashed blue line indicates the average.

We observe that the model with a longer training duration is, indeed, the second best in terms of accuracy (**2T-8H**). We also observe that the duration is highly correlated to the number of parameters of each model, as could be expected and can be seen in the **3.4** section below.

We can also analyze the use of the `AdaptiveSoftmaxLayer` comparing the metrics for the models **4T-1H** and **4T-1H-A**, given that the only difference between both is the presence of this `AdaptiveSoftmaxLayer`. We observe what we would expect: the model using adaptive softmax takes around $\frac{3}{4}$ of the training time of the model without it, while obtaining a slightly worse accuracy (around 1.5% less).

3.4 Number of parameters

The number of parameters is a very interesting metric as it somewhat indicates the ”complexity” and size of the model and, together with the training time mentioned above, is a good indicator of the computational costs and scalability of the models and their training.

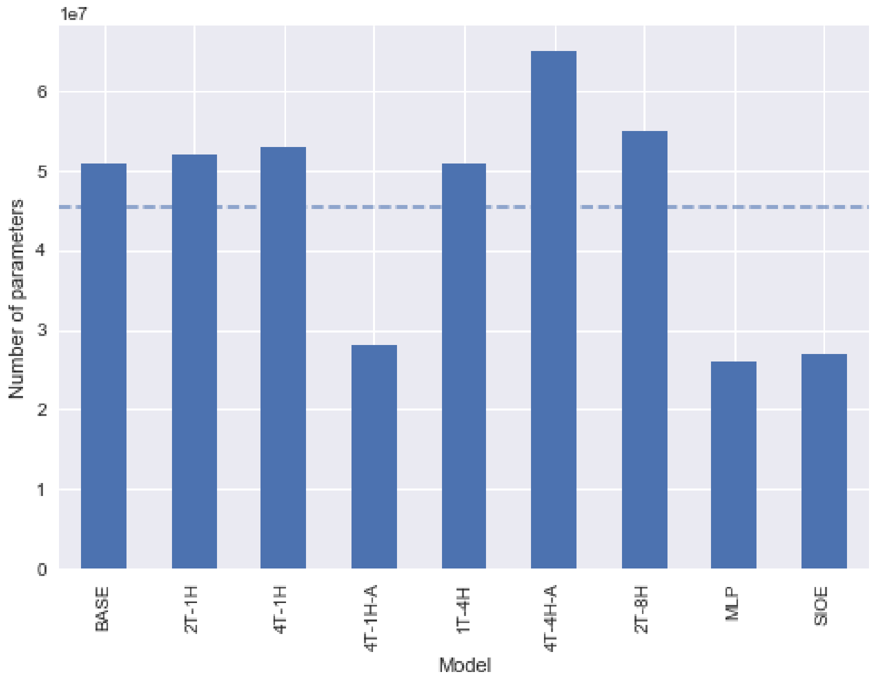


Figure 11: Number of parameters for each of the models. The dashed blue line indicates the average.

In this case, we observe that the best performing model in terms of accuracy (**4T-1H**) is the third model when it comes number of parameters while, surprisingly enough, the worst performing model (maybe due to overfitting) is the one with most parameters.

Finally, another insight we take from this is that the **SIOE** model takes approximately the same time to train as the **Base model**, while having around half of its number of parameters. That is because even though they have a similar architecture with the same amount of layers, one of the layers (word embeddings) is duplicated, which translates into 1. half the amount of paramaters to train and 2. the same training time, as the quantity of operations to perform is somewhat similar.

3.5 Summary

After a detailed exploration into each of the collected metrics, we have summarized the models into the table below. This helps to identify the best performing models taking into account all available information.

	T	H	Embed_dim	d_{ff}	d_k	P_{drop}	Adaptive Softmax	Test Acc	Training time	#Params
baseline	1	1	256	512	256	0.1	No	0.331	13708	51×10^6
	2	1	256	512	256	0.1	No	0.335	15079	52×10^6
	4	1	256	512	256	0.1	No	0.341	18100	53×10^6
	4	1	256	512	256	0.1	Yes	0.326	16178	28×10^6
	1	4	256	512	256	0.1	No	0.335	13709	51×10^6
	4	4	512	1024	128	0.2	Yes	0.237	24840	65×10^6
	2	8	256	2048	128	0.15	No	0.34	20662	55×10^6
MLP								0.302	9255	26×10^6
Sharing Input & Output Embeddings								0.313	13016	27×10^6

In essence, more complex models with a higher number of parameters (and consequently usually a higher training time) seem to perform better, with the exception of (**4T-4H**), which is the model with highest number of parameters. Let's confirm this hypothesis by taking a look at the following scatterplot, where this trend is detectable:

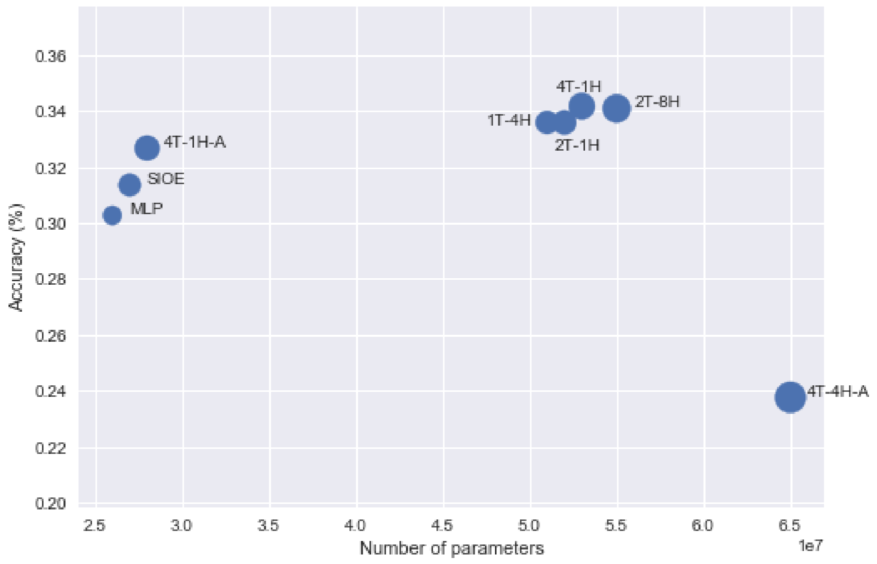


Figure 12: Scatterplot of number of parameters vs accuracy. Size indicates training time.

4 Conclusions

To wrap up this report, we can say that we now understand more deeply the power of Transformers. They stay away from complicated architectures such as recurrences or convolutions, while creating a system able to understand the given context and to find relations between the components of sentences. Its high performance and reduced training time make it one of the best state-of-the-art models.

During the development of this project we have faced one of the bigger challenges of deep learning, which is the tuning and evaluation of a model. What do we have to look for to asses that we finally achieved a great model? Which version do we keep? How much do the training and validation datasets influence the model? How long is going to last the training of this version? Why do some models perform so differently? These are only some of the questions that have come into our minds while the development of the assignment. We have realized that, in fact, deep learning is sometimes closer to trusting and following a certain intuition than an exact science.

On the other hand, we think that one of the reasons why the Self-Attention has not performed outstandingly well - compared to much simpler models such as the MLP - might be the fact that we have always treated with phrases of length 4, which may not be long enough phrases to obtain the optimal representation of relations and behaviour between words. Even so, models with Self-Attention have out-performed models without it.

A final thought is that it would be interesting to go little further into the Transformers by analyzing the self-attention process. We have seen really interesting visualizations that show the links that are formed between the words of the sentence: this is actually the key of the Transformers. Another interesting aspect about Transformers is the ability to analyze problems involving input and output modalities different from text, in order to apply them to other tasks related with sequences.

In general, it has been a great assignment to increase the knowledge acquired in the previous lab and expand our abilities in the world of Natural Language Processing.

References

[1] Samuel Lynn-Evans. *How to code The Transformer in Pytorch*. 2018. URL: `https://towardsdatascience.com/how-to-code-the-transformer-in-pytorch-24db27c8f9ec`.

[2] Ashish Vaswani. *Attention Is All You Need*. 2017. URL: `https://arxiv.org/pdf/1706.03762.pdf`.