Language Modeling

Spoken and Written Language Processing

Data Science and Engineering
Polytechnic University of Catalonia

Claudia Len Manero - 53869554
Alina Castell Blasco - 49188689S

March 2024

1 Introduction

This assignment consists on studying, developing, and comparing various models for non-causal language modeling, focusing on predicting the central word from a context of three previous and three next words. The goal is to improve the performance of the baseline TransformerLayer notebook in the competition test set by achieving improved perplexity and accuracy.

We have explored various models:

- No Attention model.
- x TransformerLayers model.
- Multi-Head Attention model.
- Domain Adaptation model.
- Sharing Input/Output embedding model.
- Hyperparameter Tuning model.

Subsequent sections detail the implementation, evaluation, and selection (if needed) of the most effective model among each model in specific.

Note that in every accuracy and loss evaluation we have taken into account the values obtained from the validation dataset 'El Periódico', not the training set nor the validation 'wikipedia' set.

The baseline model consists of an input of the predicted context words embeddings and their positional embedding, which are passed to a TransformerLayer where Attention is computed representing the dependencies between input and output, results are normalized and projected to a non-lineal space with MLP. The output of this layer is passed through a Pooling layer which groups the embeddings creating a mean for each different batch and, finally, this goes through a Lineal+Softmax layer to obtain the probability distributions of our results.

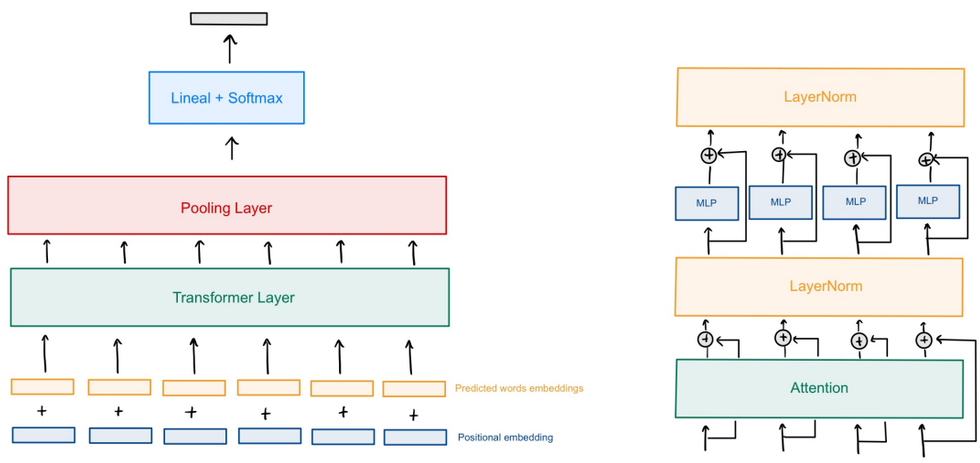

Figure 1: Baseline Model Architecture

2 No Attention model

The modifications on the baseline model to erase the attention used are composed of the removal of the self-attention layer on the Transformer. The code modifications are made on the *forward* function

of the class *Transformer*. We have commented the self-attention code lines in order to easily see the changes:

```
# This transformer layer does not contain a self-attention layer
class TransformerLayer(nn.Module):
    def __init__(self, d_model, dim_feedforward=512, dropout=0.1, activation="relu"):
        ...
    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
```

The evaluation metrics obtained from this model will be further analysed in the last section, despite that, for now we can state that the model with no self attention achieves an accuracy of 32.9% and a loss of 4.31, which are worse values than the ones achieved with the baseline model. Applying self Attention to the model allows it to focus on certain parts of the input that are more relevant and provide more context to calculation of the solution, thus, it obtains an output with more accuracy and robustness. This is the reason why a model without Attention worsens with respect to one with an Attention mechanism.

Taking into consideration that a model without attention performs worse, we will be applying attention to all the following models; even though it may not be mentioned, they all contain at least one head of attention.

3 TransformerLayers addition model

This model consists on adding two or more transformer layers to the baseline model and observe the metrics obtained for each number of layers. Then, evaluate and select the model with the number of layers that achieves best performance.

The implementation basically modifies the class *Predictor* by removing the method *self.attention* and adding a method called *self.layers*; as well as adding a loop iteration in the *forward* function. Here we present the code modifications:

```
class Predictor(nn.Module):
    # Modifiy the x variable with the corresponding number of layers
    def __init__(self, num_embeddings, embedding_dim, num_layers=x, context_words=
    params.window_size-1):
        ...
        self.layers = nn.ModuleList([TransformerLayer(embedding_dim) for _ in range(
    num_layers)])
        ...
    def forward(self, input):
        ...
        # u shape is (B, W, E)
        for layer in self.layers:
            u = layer(u)
        # v shape is (B, W, E)
        v = u.sum(dim=1)
        # x shape is (B, E)
        y = self.lin(v)
        # y shape is (B, V)
```

```
17          return y
```

We have tried adding 2, 3, 4, and 5 additional transformer layers, the reason why is explained along with the result interpretation. In order to evaluate their performances we have taken into consideration their accuracy, loss and training time. It is reminded that, at all times the metrics evaluated are the ones presented by the validation set of *El Periódico*.

We have to take into account that the addition of Transformer Layers changes the model's architecture, for example if we add 4 layers, the architecture will change to the following:

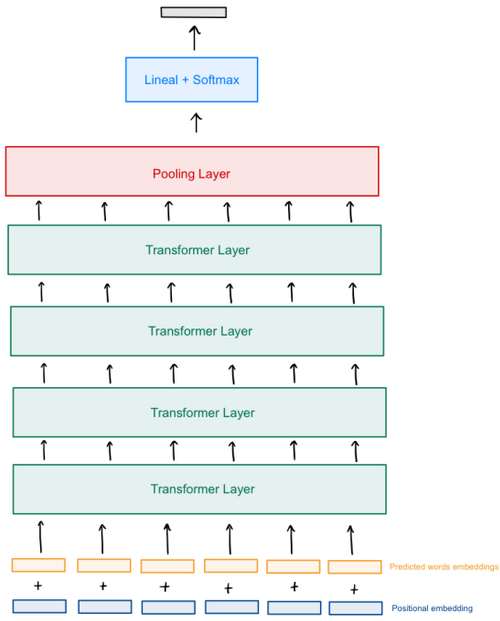

Figure 2: 4 Transformer Layers Architecture

We present a visual analysis of these metrics in the following table:

Layers	2	3	4	5
Accuracy	36.1	36.4	36.4	36.5
Loss	4.01	3.98	3.97	3.96

Table 1: Results of models with different transformer layers.

Both metrics improve with the number of layers, accuracy increases by 4 tenths from the model with least layers to the one with most, while maintaining the same accuracy from the 3 layers model to the 4 layers one. The loss metric does reflect and improvement from these last mentioned models. Still, the results seem to be asymptotic.

As for the time, the model trained on 5 layers took the longest to execute, achieving a time of 5 hours and 23 minutes. Followed by the one with 4 layers, completed in 4 hours 57 minutes and the 3 layer model, completed in 4 hours and 30 minutes. The one that took less time was the model with 2 layers, which was completed in 4 hours and 3 minutes. These facts clearly state that the more layers added the more time it takes the model to complete the training and validation. In fact, it can be seen that approximately half an hour of execution time is increased per layer, making the network costlier computationally.

To sum up, considering both the accuracy and loss metrics, and the execution time we conclude that

the improvement on the performance of the model is not worth all the execution time it takes per layer increased. Thus, we finish on with 5 transformer layers to ensure a tradeoff between accuracy and computational cost.

4 Multi-Head Attention model

For this model we changed the SelfAttention for MultiHeadAttention, creating a new class for it and then using it as the attention in the TransformerLayer.

```
def attention(query, key, value, d_k, mask, dropout):
    "Compute 'Scaled Dot Product Attention'"
    #d_k = query.size(-1)
    scores = torch.matmul(query, key.transpose(-2, -1)) / math.sqrt(d_k)
    ...
    return torch.matmul(p_attn, value), p_attn
```

```
class MultiHeadAttention(nn.Module):
    def __init__(self, heads, d_model, 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 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, 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, self.d_k, mask, self.dropout)
        # concatenate heads and put through final linear layer
        concat = scores.permute(1,2,0,3).contiguous()\
        .view(bs, -1, self.d_model)
        output = self.out(concat)
        return output
```

```
class TransformerLayer(nn.Module):
```

```
    def __init__(self, d_model, heads = 8, dim_feedforward=512, dropout=0.1,
    activation="relu"):
        super().__init__()
        self.multi_attn = MultiHeadAttention(heads, d_model)
        ...
```

For the study, we trained a model with Multi-head attention and 4 Transformer Layers, alongside another model featuring only Multi-head. Despite the potential for better performance with 5 transformer layers, we opted for 4 due to time constraints, given the high computational cost of training with 8 attention heads. See the results in the following table:

Transformer Layers	Accuracy	Loss	Time	Params
1	32.7%	4.31	3h 45m	51729664
4	37.0%	3.94	5h 12m	53310976

Table 2: Multi-head attention models comparison

These results prove that the model with Multi-head attention does not show any improvement with respect to the baseline, in fact, the results are far worse. In contrast, it works notably better when using Multi-Head Attention with 4 transformer layers. The method applies attention multiple times in parallel allowing each parallel implementation to develop its own parameters which allows the model to capture different patterns in the data by attending to different parts of the input sequence simultaneously.

5 Domain Adaptation

The modifications of this model consist on adding a subset of the validation data, in this case from *El Periódico*, to the training data. This is done in order to fine-tune our model to the task at hand, in other words, make our model adapt to a specific domain. The implementation of said modifications is showed on the next code snippet:

```
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 random data subset to the training set, fine-tune with input data
    # Number of samples of 'el Periodico' data subset modified on params
    rnd_ind = random.sample(range(len(valid_x)), k=params.num_samples)
    train_x = np.concatenate((data[0][0], valid_x[rnd_ind]), axis=0)
    train_y = np.concatenate((data[0][1], valid_y[rnd_ind]), axis=0)
    # Train model
    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}')
    # Validate model
    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
     Periodico)')
# Save model
torch.save(model.state_dict(), params.modelname)
```

The main characteristic of this model is the addition of a subset of samples from the validation set into the training set, so, the number of samples added is of high importance. In order to obtain the best performing model, we tried different versions which imply modifications of the number of transformer layers and the number of samples taken as a subset of 'el Periódico'. The size of the training set obtained from preprocessed data is of 82284341 samples, while that of the validation dataset of 'el Periódico' is of 20000 samples; we tried the fine-tuning with a 25% and a 10% of this size in order to see if the size of the subset did make a real difference. The results are shown in the next table:

Transformer Layers	NumSamples	Accuracy	Loss	Time	Params
1	5000	35.2%	4.05	3h 36m	51729664
5	5000	36.7%	3.91	5h 23m	53838080
4	2000	36.5%	3.94	4h 56m	53310976

Table 3: Domain Adaptation models comparison

First, we ran the model adding only the domain adaptation feature and 5.000 samples, and obtained a low accuracy. Then, we added to it 5 transformer layers, and observed that the model's performance improved considerably, as well as the time. On the last attempt, a subset of 2.000 samples was added to the training; while, as for the transformer layers, we considered the high computational cost that comes with the 5 layer training and decided to train with 4 layers. As it can be observed in the accuracy values, the model's performance remains almost consistent compared to the one with 5 layers and 5000 samples, suggesting that the number of Transformer Layers influences performance more significantly than the number of samples. This is attributed to the huge size difference between the training and validation datasets (from 82.284.341 to 20.000), where an increase from 2.000 to 5.000 validation samples has a small impact due to the large number of samples in the training dataset.

To sum up, the domain adapted model that performs the best while maintaining a reasonable execution time is the one that adds 2000 samples to the training set and has a 4 transformer layers architecture. This model is the one that will be considered on the final table comparison.

6 Shared Input/Output

In this section we present a model characterised by sharing input and output embedding. This approach implies that the embeddings learned for input tokens are also used as the embeddings for the corresponding output tokens in the model, which makes the transfer of knowledge from input to output easier. Note that we are able to apply this technique because both belong to a common vocabulary. The implementation of the code is the next:

```
class Predictor(nn.Module):
    ...
    self.emb.weight = self.lin.weight  # Sharing input and output weights
```

4 . . .

In order to analyse different model configurations we have chosen other modifications to add to the sharing input/output modification. The results of the different models are shown in the next table:

Transformer Layers	Multi-Head	Accuracy	Loss	Time	Params
1	No	34.7%	4.09	3h 28m	26129152
4	No	37.5%	3.87	4h 49m	27710464
4	Yes	37.8%	3.87	5h 4m	27710464

Table 4: Sharing Input/output embedding models comparison

It can be observed at first sight, that the number of parameters has been reduced to half the number used for the previous models. This is clearly due to the sharing of embeddings from input to output, and is a great characteristic since the less parameters the less complex the model is and the easier to train. From the results of the models it can be extracted that applying only the sharing input/output architecture does not improve the baseline model, in fact it has slightly lower accuracy. Nevertheless, if applied along with 4 Transformer Layers and with Multi-Head Attention it improves greatly. We have computed two different models, one with Multi-Head attention and one with self-Attention in order to observe the real effect of Multi-Head. We have seen that Multi-Head has no effect over the loss of the model but that it does improve its accuracy, it also consumes more execution time. Thus, we consider that this last model configuration is the best performing within the sharing input/output embedding ones; since the difference on computational cost between the second best is minor and the improvement on accuracy is substantial.

7 Hyperparameter Tuning model

Our last implementation consisted on optimizing the hyperparameters to find the best fitted for training this model.

We tried to optimize the embedding_dim parameter, the batch_size, the number of epochs, the optimizer, the scheduler, the activation function, the window_size and the dropout value.

We will discuss each of these optimizations one by one and comment the final model containing all optimized parameters.

For reference, our baseline model had embedding_dim=256, batch_size=2048, epochs=4. We won't be comparing these parameter values in the table for that reason as we already know the results for them.

Hyperparameter	Train time	Train acc	Train loss	Wiki acc	Wiki loss	Per acc	Per loss
Baseline model	**3h 37min**	**46.4%**	**3.08**	**45.2%**	**3.23**	**34.9%**	**4.13**
emb_dim=128	2h 23min	45.1%	3.24	44.3%	3.33	34.2%	4.21
emb_dim=512	5h 48min	46.8%	3.00	45.3%	3.21	34.5%	4.15
batch_size=1024	4h 1min	46.4%	3.14	45.1%	3.26	**34.9%**	4.17
batch_size=4096	**3h 18min**	**46.6%**	**3.05**	**45.5%**	**3.20**	**35.1%**	**4.13**
epochs=8	7h 49min	**48.1%**	**2.92**	**46%**	**3.16**	**35.5%**	**4.08**
optimizer='rmsprop'	3h 30min	28.1%	5.04	23.4%	5.24	17%	5.92

Table 5: Hyperparameter optimization metrics

Looking at the table 5 we can analyze which are the optimal parameters for the efficency of the model. For the *embedding_dim* parameter, even though $embedding\_dim = 512$ has better training results, it has worse results in the validation step, which makes us conclude that our original *embedding_dim* parameter was the best one.

For the *batch_size* hyperparameter, results for $batch\_size = 1024$ were very close to the baseline model metrics, but despite training accuracy and el Periodico validation accuracy being equal to the baseline, we considered the default parameter was better as it was faster to train and gave smaller loss values. We then tried with $batch\_size = 4096$ and had good results, giving less training time and better validation accuracy for el Periodico dataset.

The *epochs* parameter was the one we hoped would cause a bigger improvement in resulting metrics as most of the times a low accuracy is indicator of too little training time. This hypothesis was confirmed by the metrics of the training of the model with $epochs = 8$ as we observe an improvement of 0.6% in accuracy and a 0.05 reduction in loss from the baseline model. The only problem is the training time as it doubled, but we think although it is slower, results are worth.

We also tried using a different optimizer, but results were underwhelming, giving half of the accuracy we had with our baseline model.

We could conclude that a better implementation of the model would be to change *batch_size* to 4096 and *epochs* to 8.

8 Models Comparison

The next comparison table summarizes the key features of selected models across the previous sections, which contain the reason behind their selection and the incorporation of certain features from one model to the next. In general, the addition of 4 transformation layers to several model variations resulted in performance improvements compared to those with only 1 transformer layer. The hyperparameters section is not specified in the table as the values remain consistent across all models, except for the Hyperparameter Tuning model. The common hyperparameters employed for all models include a batch size of 2048, embedding dimension of 256, window size of 7, Adam optimizer, dropout rate of 0.1, and bias, except for the hyperparameter optimized model which uses different values.

		Accuracy	Loss	Train Time	Parameters	Transformer Layers
	Baseline	**34.9%**	**4.13**	**3h 37min**	**51729664**	**1**
Language Model Comparison	No Attention	32.9%	4.31	3h 22min	51729664	1
	5 Transformer Layers	36.5%	3.96	5h 23min	53838080	5
	Multi-Head Attention	36.5%	3.94	4h 56min	53310976	4
	Domain Adaptation	36.5%	3.94	4h 56min	53310976	4
	Sharing embed	**37.8%**	**3.87**	**5h 4min**	**27710464**	**4**
	Batch Size tuning	35.1%	4.13	3h 18min	51729664	1
	Epochs tuning	35.5%	4.08	7h 49min	51729664	1

Table 6: Model Evaluation

Comparing all models used in this project, we can conclude that the improvement task has been successful as we only found one model with worse performance than the baseline.

We should highlight the Shared embedding strategy, which used 4 Transformer Layers and Multi-Head

Attention, resulting in a validation accuracy of 37.8% and a loss of 3.87, outperforming the baseline by 2.9% and 0.26 respectively. All details and explanations regarding each model are provided in their corresponding sections, concluding in the presentation of the best-performing model from each section in this table. To sum up, after observing the features of each of the tested models we decided to choose the mentioned model since it presents the best performance, and it contains the least number of parameters which makes it the least complex.