Spoken and Written Language Processing - POE

Assignment 2. Language Modeling.

Gerard Martín Pey

Jose Àngel Mola Audí

2nd April 2023

Assignment 2. Language Modeling.

Before explaining the practice, we would like to make clear the problem we had, and which we already explained in an email, but we also want to record it in this report.

When we made the Predictor class, in the __init__ function, we only defined self.att once, so we were only training one layer and therefore not training the number of layers we wanted. So, what we've been doing is having fewer parameters to train since we've trained the same layer multiple times. The purpose of this practice was to have and train different layers.

We caught this mistake when we had already consumed both GPU hours Kaggle provides per user, so given the delivery date of the practice, we couldn't re-run everything again.

We did add the number of parameters we would have if we had done it correctly, but we couldn't train the different models.

The code attached in the file is the one that should have been used. Notice that the experiments have not been run with this code. The initial code reused the same layer.

The objective is now to obtain a good model in terms of perplexity and accuracy, improving the performance of the baseline TransformerLayer notebook in the competition test set.

Suggestions:

- **Increase the number of TransformerLayers (2 or more)**
- **TransformerLayer with multi-head attention**
- **Hyperparameter optimization: embedding size, batch size, pooling layer (mean, max, first, ...), optimizer, learning rate/scheduler, number of epochs, etc.**

The report should include the modified source code, a simple schematic drawing of the model, your results, and conclusions.

Create a comparative table of the studied models with respect to the single-layer transformer baseline, including loss, accuracy, training time, number of parameters, and hyperparameters differences with at least 4 new models or hyperparameters.

Considering the limitations of time that Kaggle poses regarding the use of GPU, it has been decided to take the following approach: The two members of the team have decided to run simultaneously some experiments tuning different hyperparameters and modifying the architecture in different ways, starting from the baseline, aiming at finally combining the best results of each approach into a final best model. Notice that we are fully aware that this procedure does not guarantee at all that the mentioned final model will be the best one and, in fact, it could turn out to be worse than most of the others.

Architecture

First, we have decided to check how the baseline model behaves when adding transformer layers. Therefore, we have added up to 3 extra layers by concatenation. The hyperparameters of these models remain the same as the ones from the initial model:

Assignment 2. Language Modeling.

Model 0 (Baseline): 1Transform Layer, emb_size=256, batch_size=2048

Model1: 2 Transform Layers, emb_size=256, batch_size=2048

Model2: 3 Transform Layers, emb_size=256, batch_size=2048

Model3: 4 Transform Layers, emb_size=256, batch_size=2048

The results obtained have been the following:

Metrics	Model 0	Model 1	Model 2	Model 3
Train accuracy	46.6%	47.0%	46.9%	47.0%
Train loss	3.06	3.03	3.05	3.04
Valid accuracy Wikipedia	45.4%	46.0%	46.0%	45.8%
Valid loss Wikipedia	3.21	3.16	3.16	3.17
Valid accuracy Periódico	34.9%	35.3%	35.7%	35.4%
Valid Loss Periódico	4.12	4.07	4.06	4.07
Training time	3h 44min	4h 11min	4h 38min	5h 4min
#parameters	51.729.664	51.729.664	51.729.664	51.729.664
#parameters with correct layers	51.729.664	51.961.856	52.341.504	52.721.152

It can be appreciated that adding transform layers slightly enhances the accuracy for validation and training sets, despite taking longer time for the network to be trained. Specifically, the model with 3 layers is the one that seems to provide the best results.

However, it can be observed that the number of parameters does not change. This is not intuitive, and it is obviously wrong. We have not declared new transformer layers in the Predictor but reused the same layer, which results in the network learning which parameters optimize the model when applied num_layers times rather than learning different parameters in each layer. Given the lack of GPU time we are not able to rerun the experiments, so the results presented next are also 'defective'.

Simultaneously, we have implemented multihead attention to the baseline model (1 transform layer). This mechanism is expected to improve models' performance as it has been stated in papers such the one introducing it. In this case we have tried models with 4 and 8 heads:

Metrics	Baseline	4 heads	8 heads
Train accuracy	46.6%	47.5%	47.4%
Train loss	3.06	3.00	3.00
Valid accuracy Wikipedia	45.4%	46.2%	46.1%
Valid loss Wikipedia	3.21	3.16	3.17
Valid accuracy Periódico	34.9%	35.7%	35.6%
Valid Loss Periódico	4.12	4.07	4.08
Training time	3h 44min	3h 46min	3h 57min
#parameters	51.729.664	51.582.208	51.557.632

As observed, the model with 4 heads is the one that seems to provide the best output. Moreover, it does not suppose a huge increase in training time. We observe that the number of parameters decrease when increasing the number of heads, which may not seem intuitive before the experiment.

After that, we have selected the best model for each of both of the previous experiments, which are 3 layers and 4 heads. To that architecture, we have added layers of pooling (kernel_size=stride_size=window_size) between the last transform layer and the linear one. These have been the results:

Metrics	Sum	Max pooling	Avg pooling
Train accuracy	47.4%	47.2%	47.5%
Train loss	3.00	3.04	3.01
Valid accuracy Wikipedia	46.1%	46.0%	46.2%
Valid loss Wikipedia	3.16	3.18	3.15
Valid accuracy Periódico	35.5%	35.6%	35.8%
Valid Loss Periódico	4.08	4.07	4.05
Training time	4h 12min	5h 11min	5h 11min
#parameters	51.582.208	51.582.208	51.582.208

As expected, the mean pooling, which is known to be useful in NLP models, performs better than the addition. In contrast, the max pooling, despite being quite useful in image processing, barely improves the performance of the model.

Although not adding parameters, pooling layers imply and increase in the training time of almost an hour.

Hyperparameters

Apart from changes in the architecture, we have tried to modify some parameters to see how the performance of the model changes. This has been the models trained, with 4 layers and multihead attention using 4 heads:

Model1: embedding_dim = 300, batch_size=2048

Model2: embedding =100, batch_size =2048

Model3: embedding =400, batch_size =1024

Model4: embedding =300, batch_size =1024

Metrics	Model 1	Model 2	Model 3	Model 4
Train accuracy	47.8%	44.9%	47.5%	47.3%
Train loss	2.987	3.28	3.05	3.07
Valid accuracy Wikipedia	46.4%	44.0%	46.0%	46.0%
Valid loss Wikipedia	3.15	3.37	3.19	3.22
Valid accuracy Periódico	35.8%	33.9%	35.3%	35.5%
Valid Loss Periódico	4.07	4.26	4.12	4.15
Training time	6h	3h 22min	85 45min	7h 28min
#parameters	60.470.912	20.122.312	80.697.712	60.470.912
#parameters with correct layers	61.874.648	20.486.248	82.778.848	61.874.648

Looking at these results, we observe that using 300 as embedding size and keeping the batch size as the baseline seems to work better than the rest of models. Therefore, we add the avg pooling layer to this network and run which is expected to be the best model:

Metrics	Final model
Train accuracy	47.9%
Train loss	2.96
Valid accuracy Wikipedia	46.6%
Valid loss Wikipedia	3.11
Valid accuracy Periódico	36.0%
Valid Loss Periódico	4.03
Training time	5h 26min
#parameters	60.470.912
#parameters with correct layers	61.406.736

Conclusion

In this practise, we have been introduced to the concept of transformer and we have observed how it improves the performance of the CBOW model. We done that by experiencing with different architectures and hyperparameters. We have got closer to multihead attention concept and which effects does it have to the layers, contributing to reach slightly better performances. At the same time, we have discovered how beneficial can mean pooling layers be in NLP tasks. Despite not having done properly some experiments concerning the number of layers, given that we have not trained different layers but the same several times, we have been able to see the potential of this architecture, leading to better results that the previous networks.

Assignment 2. Language Modeling.

Codes

```python
class MultiHeadAttention(nn.Module):

    def __init__(self, embed_dim, num_heads=4, bias=True):
        super().__init__()

        assert(embed_dim%num_heads==0)
        self.num_heads = num_heads
        self.head_dim = embed_dim // num_heads

        self.k_proj = nn.ModuleList([nn.Linear(self.head_dim, self.head_dim, bias=bias) for i in range (num_heads)])
        self.v_proj = nn.ModuleList([nn.Linear(self.head_dim, self.head_dim, bias=bias) for i in range (num_heads)])
        self.q_proj = nn.ModuleList([nn.Linear(self.head_dim, self.head_dim, bias=bias) for i in range (num_heads)])
        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
        for i in range (self.num_heads):
            nn.init.xavier_uniform_(self.k_proj[i].weight, gain=1 / math.sqrt(2))
            nn.init.xavier_uniform_(self.q_proj[i].weight, gain=1 / math.sqrt(2))
            nn.init.xavier_uniform_(self.v_proj[i].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):
        # x shape is (B, W, E)
        x = torch.split(x, self.head_dim, dim=2)
        q = [self.q_proj[i](x[i]) for i in range(self.num_heads)]
        k = [self.k_proj[i](x[i]) for i in range(self.num_heads)]
        v = [self.v_proj[i](x[i]) for i in range(self.num_heads)]

        # Determine output
        y = [attention(q[i], k[i], v[i]) for i in range(self.num_heads)]
        # Concatenar aquí
        y = torch.cat(y, dim=2)
        y = self.out_proj(y)
        return y
```

```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.att1 = TransformerLayer(embedding_dim)
        self.att2 = TransformerLayer(embedding_dim)
        self.att3 = TransformerLayer(embedding_dim)
        self.meanpool = nn.AvgPool1d(kernel_size=context_words, stride=context_words)
        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.att1(u)
        v2 = self.att2(v)
        v3 = self.att3(v2)

        x = self.meanpool(v3.transpose(1,2)).squeeze()
        # x shape is (B, E)

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