Universitat Politècnica de Catalunya

Ciència i Enginyeria de Dades

Processament Oral i Escrit
Word Vectors

Gerard Comas Quiles & Miquel Lopez Escoriza

March 2024

1 Introducción

The aim of this lab is to enhance a model's performance in terms of perplexity and accuracy, outperforming the baseline provided by the TransformerLayer notebook on the competition's test set. Our approach is structured as follows:

We will explore various NLP techniques across different sections of the lab to strengthen our network's effectiveness. The measure of success will be the improvement in test accuracy over the baseline. Nevertheless other metrics are going to be taken into account such as Test Loss, Training time(Tt.) and the number of parameters(#Par.). Ultimately, we will integrate these enhancements to devise the most accurate network configuration possible.

Subsequently, we will conduct hyperparameter tuning on this optimized network. It's important to note that the initial set of hyperparameters will remain constant to ensure a fair comparison between network iterations.

Hyperparameter	**Value**
Batch size (bs)	2048
Learning Rate (lr)	0.01
Embedding size (es)	256
Pooling layer (pl)	Sum
Optimizer (o)	Adam's
Number of epochs (ne)	4

Table 1: Initial Hyperparameters

2 Feed Forward

To evaluate whether a feedforward network could enhance our architecture, we've placed the feedforward layer right after the Transformer Layer. To ease coding we have created a FeedForward class.

```python
class FeedForward(nn.Module):
    def __init__(self, d_model, dim_feedforward=512, dropout=0.1,
    ↪   activation="relu"):
        super().__init__()
        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):
        src = self.norm1(src)
        src = self.linear2(self.dropout(F.relu(self.linear1(src))))
        src = self.norm2(src)
        return src
```

The results are the following ones:

Model	**Loss**	**Acc. (%)**	**Tt. (s)**	**# Par.**
Base model	4.13	34.9	3h 40min.	51.7M
Feedforward	4.06	35.5	4h 5min.	52M

Table 2: Results obtained from the Feedforward model

We can see that the feedforward model outperforms the base model.

3 Addition of transformers

Incorporating multiple transformer layers can significantly enhance model performance by enabling deeper comprehension of sequential data. This approach allows the model to capture increasingly complex dependencies and patterns within the data, which might be missed with a smaller architecture. Each transformer layer builds on the insights of the previous one, refining the model's ability to process and interpret the input.

Model	**Loss**	**Acc. (%)**	**Tt. (s)**	**# Par.**
1 Transformer	4.03	35.8	4h 3min.	52M
2 Transformer	4.03	35.8	4h 3min.	52M
3 Transformer	3.98	36.2	4h 30min.	53M
4 Transformer	3.93	37	4h 56min.	53M
5 Transformer	3.95	36.9	5h 23min.	54M
6 Transformer	3.94	37.1	5h 52 min.	54M
7 Transformer	3.95	36.9	6h 18min.	55M

Table 3: Comparison of performance with number of transformers

As the number of transformer layers increases, the model demonstrates enhanced generalization capabilities, evidenced by reduced test loss and improved accuracy. However, an interesting observation emerges when the model is expanded to include five or more transformer layers; here, we see a convergence or even decline in accuracy.

This deterioration could be attributed to overfitting, where the model becomes too complex, capturing noise in the training data rather than learning the underlying pattern. It might also be a sign of the model's difficulty in effectively training or optimizing with increased depth.

4 TransformerLayer with multi-head attention

A possible reason that could explain why the model does not learn effectively when increasing size is because it lacks information. Transformers use self-attention which 'states' that the word in the context are related, but in a unique way. The reality is that the different words in a sentence can relate to each other in many different ways simultaneously. We can capture the different relations amongst words with multi-head attention.

The multi-head attention layer is constructed by modifying the self-attention layer and refining the attention score computation to accommodate the distinct scores from each attention head.

```

def attention(query, key, value,d_k, mask=None, dropout=None):
    "Compute 'Scaled Dot Product Attention'"
    scores = torch.matmul(query, key.transpose(-2, -1)) \
             / math.sqrt(d_k)
    if mask is not None:
        scores = scores.masked_fill(mask == 0, float('-Inf'))
    p_attn = F.softmax(scores, dim = -1)
    if dropout is not None:
        p_attn = dropout(p_attn)
    return torch.matmul(p_attn, value)

class MultiHeadAttention(nn.Module):
    def __init__(self, embed_dim,heads, bias=True):
        super().__init__()
        self.k_proj = nn.Linear(embed_dim, embed_dim, bias=bias)
        self.v_proj = nn.Linear(embed_dim, embed_dim, bias=bias)
        self.q_proj = nn.Linear(embed_dim, embed_dim, bias=bias)
        self.out_proj = nn.Linear(embed_dim, embed_dim, bias=bias)
        self.reset_parameters()

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

    def forward(self, x):
        bs = x.size(0)
        # x shape is (B, W, E)
        q = self.q_proj(x).view(bs, -1, self.h, self.d_k)
        # q shape is (B, W, E)
        k = self.k_proj(x).view(bs, -1, self.h, self.d_k)
        # k shape is (B, W, E)
        v = self.v_proj(x).view(bs, -1, self.h, self.d_k)
        # k shape is (B, W, E)
        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)
        # 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
```

Let's now compare the results of multi-head attention against the self-attention. Note that the number of parameters used is the same as for every head the same weights are used.

Model	**Loss**	**Acc. (%)**	**Tt. (s)**	**# Par.**
4 Transformer self-attention	3.93	37	4h 56min.	53M
4 Transformer 4-multihead-attention	3.92	37.2	5h 12min.	53M
5 Transformer self-attention	3.95	36.9	5h 23min.	54M
5 Transformer 4-multihead-attention	3.91	37.3	5h 42min.	54M

Table 4: Comparison of performance between multihead and self attention

We can see that there is a substantial improvement from self-attention to multi-head attention. This can enable the model to train and learn well when increased in depth. Nevertheless, the number of heads used for the multihead attention model plays a crucial part.

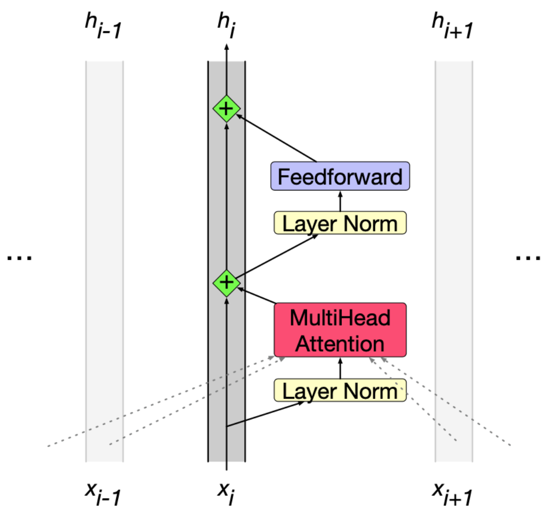
Model	**Loss**	**Acc. (%)**	**Tt. (s)**	**# Par.**
5 Transformer 4-multihead-attention	3.91	37.3	5h 42min.	54M
5 Transformer 8-multihead-attention	3.89	37.4	5h 58min.	54M

Table 5: Comparison of performance between number of heads for a multihead transformer

By segmenting the attention process into multiple heads, the model can simultaneously attend to various aspects of the input data, enhancing its ability to capture a wide range of dependencies and features. This is the reason why the model has better performance when the number of heads is increased. However is not that noticeable.

5 Prenorm

There is an alternative form of the transformer architecture that is commonly used because it performs better in many cases. In this prenorm transformer architecture, the layer norm happens in a slightly different place: before the attention layer and before the feedforward layer, rather than afterwards.



The implementation of the prenorm model only changes the transformer layer base code:

```
def forward(self, src):
    src_norm1 = self.norm1(src)
    src2 = self.self_attn(src_norm1)
    src2 = src + self.dropout1(src2)
    src3 = self.norm2(src2)
    src3 = self.linear2(self.dropout(F.relu(self.linear1(src3))))
    src = src2 + self.dropout2(src3)
    return src
```

Model	**Loss**	**Acc. (%)**	**Tt. (s)**	**# Par.**
4 Transformer 8-multihead + Prenorm	3.80	38.5	5h 26min.	53M
5 Transformer 8-multihead-attention	3.89	37.4	5h 58min.	54M

Table 6: Comparison of performance with prenorm

We can see in the table above how the prenorm model outperforms a model without prenorm but with more transformers. We think this has been the biggest performance improvement he have encountered as it improves drastically the base model without using more parameters and without augmenting training time. This is the reason why we tried this model with far more transformers to see if it performed well when increasing its depth and the results are astonishing:

Model	**Loss**	**Acc. (%)**	**Tt. (s)**	**# Par.**
4 Transformer 8-multihead + Prenorm	3.80	38.5	5h 26min.	53M
6 Transformer 8-multihead + Prenorm	3.74	39	6h 33min.	55M
8 Transformer 8-multihead + Prenorm	3.72	39.4	7h 40min.	55.5M
10 Transformer 8-multihead + Prenorm	3.69	39.5	8h 46min.	54M

Table 7: Comparison of performance with prenorm and number of transformers

6 Sharing output embedding

in weight tying, we use the same weights for two different matrices in the model. Thus at the input stage of the transformer the embedding matrix E is used to map from a one-hot vector over the vocabulary to an embedding . And then in the language model head, E^T, the transpose of the embedding matrix is used to map back from an embedding to a vector over the vocabulary. In the learning process, E will be optimized to be good at doing both of these mappings.

We can tye the weights of the embedding and linear layer the following way:

```
    self.lin.weight = self.emb.weight
```

Nevertheless, the results were not the expected ones as the test loss was increased. A reason to explain this phenomenon can be that the number of parameters is reduced to it makes the convergence of the model more difficult(even though it can generalize better). And as we only have 12 hours to train the model did not converge.

7 Domain adaption

So far, we've only considered the data from the previous practice for training. However, when evaluating its performance and accuracy on the test data, it's important to understand the probability distribution of the domain we're trying to predict. This understanding could enhance the model's functionality.

However, the challenge lies in the fact that we don't have the answers for the test data, so we consider them unlabeled. Our approach to this issue will involve working with semi-labeled data, gradually labeling them iteratively as the model learns. This approach will aid in improving the model's performance over time.

The implementation of this solutuion for Domain Adaptation is:

```
for epoch in range(params.epochs):
    # Domain Adaptation
    valid_x_df = pd.read_csv(f'{COMPETITION_ROOT}/x_test.csv')
    test_x = valid_x_df[tokens].apply(vocab.get_index).to_numpy(dtype='int32')
    test_y = validate(model, None, test_x, None, params.batch_size, device)
    data_y = np.concatenate((data[0][1], test_y))
    concatenated_data = np.concatenate((data[0][0], test_x), axis=0)
    data_x = np.reshape(concatenated_data, (len(data_y), -1), order='F')

    acc, loss = train(model, criterion, optimizer, data[0][0], data[0][1],
    ↪  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)')
```

Let's now compare the results obtained with and without domain adaptation for a 4 Transformer Layers with self-attention:

Model	**Loss**	**Acc. (%)**	**Subm. Acc. (%)**	**Tt. (s)**	**# Par.**
Without Domain Adaptation	3.93	37.0	35.1	4h 56min.	53M
With Domain Adaptation	3.89	37.1	35.9	5h 03min.	53M

Table 8: Comparison of performance with and without domain adaptation

We can observe that, overall, the two solutions share many similarities, with the notable exception being the submission accuracy, which is significantly higher in the case of Domain Adaptation. It enables the model to better adjust to the specific characteristics of the new data domain, leading to a notable improvement in prediction accuracy.

Other approaches can be considered:

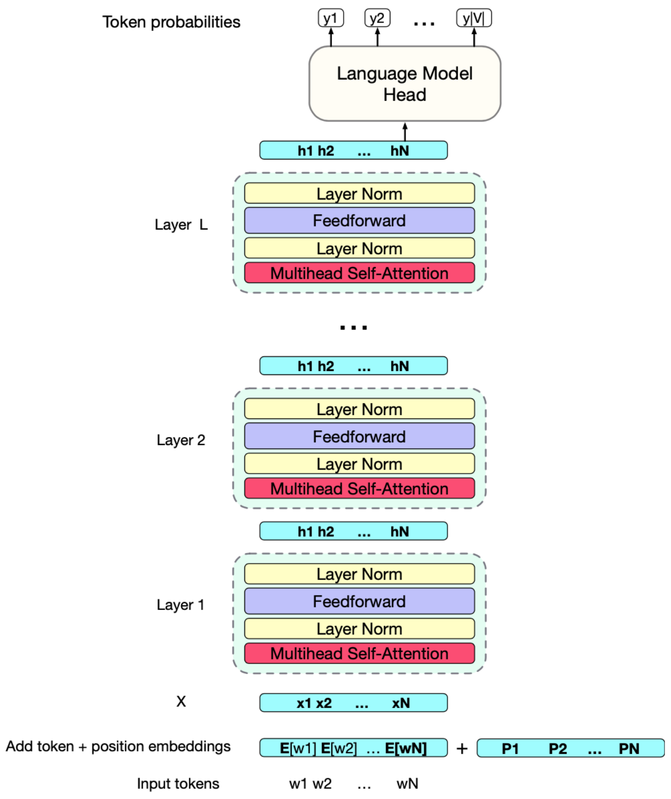
- Label the target data before each epoch instead of after.
- Remove the previously pseudo-labeled target data from the training dataset.
- Employ transfer learning by training the model with target data while freezing certain layers.
- Retrain the model for the target data but with a reduced learning rate.

However, due to the time constraints of Kaggle's execution, we will not consider these approaches, although it would be interesting to study each one in depth.

8 Final Model

To enhance the clarity and presentation of your explanation regarding the final model implementation and its comparison to the BERT model, consider the following revised version:

Our final model is inspired by BERT, with specific modifications tailored to our objectives. The standard BERT model architecture can be visualized as follows:



Characteristics of BERT include:

- Hidden layers, each with a dimensionality of 768.
- A total of 12 transformer blocks, each incorporating 12 multihead attention layers.

To better suit our project's needs, we've introduced several key alterations to the original BERT architecture:

- Prenormalization: We've implemented prenormalization layers within the architecture. This modification was adopted after observing improved performance in our preliminary tests, likely due to enhanced stability and training dynamics.

- Feedforward Network Dimensionality: The size of the feedforward networks within each transformer block has been adjusted to 512. This alteration aims to optimize the model's complexity and efficiency, potentially enhancing its ability to learn from our specific dataset.
- Transformer and Attention Configuration: While maintaining the original count of 12 transformer blocks, we've modified the multihead attention mechanism within each block to contain 8 heads instead of 12. This change is based on our experiments, which suggested that a reduced number of attention heads, in this context, provides a beneficial trade-off between model complexity and performance.

Model	**Loss**	**Acc. (%)**	**Subm. Acc. (%)**	**Tt. (s)**	**# Par.**
Final Model	3.68	40.0	38.0	12h.	58M

Table 9: Final model performance

9 Ensemble Learning

Combine predictions from multiple models trained with different hyperparameters or architectures to improve overall performance. Ensemble methods often yield better results than individual models.

We have implemented a Python code to explore this concept, but further examination is warranted as significant advancements have been made with only a few simplistic models.

```python
def ensemble_voting(vectors):
    vector_length = len(vectors[0])
    new_vector = []

    for i in range(vector_length):
        elements = [vector[i] for vector in vectors]
        counter = Counter(elements)
        most_common_element = counter.most_common(1)[0][0]
        new_vector.append(most_common_element)

    return new_vector
```

To showcase its capabilities, we achieved a 38.45% accuracy using six models, each with an individual accuracy of approximately 37%. This propelled us to the first place in Kaggle's competition leaderboard (at least with the 30% of the test data)

10 Hyperparameter optimization

Hyperparameter optimization involves determining the optimal settings for parameters that are not learned during training. We haven't focused much on this aspect because our best model took 12 hours to execute!

First, we're exploring different values for the embedding dimension while keeping other settings constant (one epoch and 5 transformer layers with 8 multi-head attention). Here are the results for the embedding dimension:

Embedding dim.	**Loss**	**Acc. (%)**	**Tt. (s)**	**# Par.**
128	4.23	34.4	59min.	26M
256	4.14	35.3	1h 29min	54M
512	5.73	19.6	2h 37min.	110M

Table 10: Performance for different embedding dim

From this table, we observe changes in loss, accuracy, training time, and number of parameters as we vary the embedding dimension. This exploration helps us understand that a smaller embedding size results in fewer trainable parameters and faster training times. However, we would expect a model with more parameters to perform better. Contrary to this expectation, a model with a larger embedding dimension performs worse, likely because it struggles to generalize with such a large amount of stored information.

Related to the number of parameters in the network, let's see what happens if we increase the size of the transformer networks. It is recommended that they be larger than the embedding size and also multiples of it. We will work with the original embedding dimension of 256:

Transformer dim.	**Loss**	**Acc. (%)**	**Tt. (s)**	**# Par.**
256	4.16	34.8	1h 25min.	53M
512	4.14	35.3	1h 29min.	54M
1024	4.13	35.2	1h 40min.	55M
2048	4.47	31.6	2h	57M

Table 11: Performance for different transformer dim

The accuracy results among the models were largely consistent, except for the largest one, which performed considerably worse. Moreover, there was minimal variance in both execution time and the number of parameters across the models. Nevertheless, if we were to select the optimal model, it would be the one with a transformer dimension of 256. This model attains the highest accuracy without significantly prolonging training time.

Other parameters that we could consider studying are:

- Batch size
- Learning Rate
- Optimizer
- Loss Function
- Activation Function
- Dropout Rate

11 Further possible improvements

This section outlines potential enhancements to refine our model's performance further. These improvements are aimed at leveraging advanced techniques and pre-existing knowledge within the field to boost the model's understanding and processing of data.

11.1 Utilizing Pre-trained Word2Vec Embeddings

One promising avenue for improvement involves incorporating pre-trained Word2Vec embeddings into our model. Word2Vec provides a rich representation of word meanings by mapping them into high-dimensional vectors. By starting with these pre-trained embeddings, our model can benefit from a foundational understanding of word relationships and semantics learned from vast amounts of text. This could significantly improve the model's ability to grasp the nuances of language, leading to more accurate interpretations of the input data.

11.2 Adapting Training Techniques from BERT

Another potential enhancement is to modify our training approach to mirror techniques used by BERT (Bidirectional Encoder Representations from Transformers), particularly its masking strategy. BERT's training method involves randomly masking some of the tokens from the input and then predicting these masked tokens. This approach encourages a deep contextual understanding of the text, as the model must infer the missing words based on the surrounding context. Adopting a similar strategy could greatly enhance our model's ability to understand and process language data, promoting a more nuanced and context-aware interpretation of the input.