UNIVERSITAT POLITÈCNICA
DE CATALUNYA
BARCELONATECH

Bachelor's degree in Data Science and Engineering

Spoken and Written Language Processing - 270225

Assignment 2 - Language Modeling

Sílvia Fàbregas
Eduard Morillo

Professors:
José Adrián Rodríguez
Carlos Escolano

May 23, 2024

Contents

Motivation

In the previous assignment, we aimed at constructing meaningful word embeddings. In order to do so, we trained various models in the task of predicting words given a context or predicting a context given a word.

The objective now is to obtain a good model in terms of accuracy, increasing the performance of a baseline Predictor model. To study and compare the trained models, a comparative table is included in this study, covering **loss, accuracy, training time, number of parameters and hyperparameters**, together with the description and diagram of the architectures.

Theoretical background

Causal language models, such as Recurrent Neural Networks (RNNs) and Convolutional Neural Networks (CNNs), have been foundational in natural language processing tasks. RNNs, in particular, are adept at sequential data processing due to their recurrent connections, allowing them to capture temporal dependencies in sequences. Similarly, CNNs have been successful in capturing local patterns in sequential data.

However, these models come with limitations, especially when dealing with long-range dependencies in sequences. RNNs suffer from vanishing and exploding gradients, limiting their ability to capture long-term dependencies effectively. CNNs, on the other hand, have fixed receptive fields, which can restrict their ability to capture global context in sequences.

Non-causal language models

Non-causal language models, also known as masked language models, address the limitations of causal models by allowing bidirectional processing of sequences. In these models, each token in the sequence can attend to all other tokens, regardless of their position in the sequence. This bidirectional processing enables the model to capture long-range dependencies more effectively.

The transformer architecture

The Transformer model [1] revolutionized natural language processing by proposing a purely attention-based architecture for sequence transduction tasks. Unlike RNNs and CNNs, the Transformer model relies solely on self-attention mechanisms to capture dependencies between input and output tokens in parallel

Self-attention mechanism

The self-attention mechanism is the cornerstone of the Transformer model. It allows each word in the sequence to attend to all other words, capturing dependencies across the entire sequence.

Given an input sequence $X = \{x_1, x_2, ..., x_n\}$, where x_i represents the i-th token embedding, the self-attention mechanism computes attention scores between each pair of tokens. These scores determine the importance of other tokens with respect to a particular token.
Let's denote the attention score between token x_i and token x_j as $A(x_i, x_j)$. The attention score is computed using a compatibility function, often a dot product:

$$A(x_i, x_j) = \text{softmax}\left(\frac{Q(x_i) \cdot K(x_j)}{\sqrt{d_k}}\right)$$

Here, $Q(x_i)$ and $K(x_j)$ are query and key vectors computed from token embeddings x_i and x_j respectively. The division by $\sqrt{d_k}$ is a scaling factor to stabilize gradients. with d_k the dimensionality of the key vectors. Next, the attention scores are used to compute weighted sums of the value vectors $V(x_j)$, which represent information associated with each token:

$$\text{Attention}(X) = \sum_{j=1}^{n} A(x_i, x_j) \cdot V(x_j)$$

This operation is performed for each token in the sequence, producing the attention output.

Multi-head attention

The Transformer model *can* is able to train multiple attention heads in parallel. This paradigm is referred to multi-head attention, and because its ability to attend to different parts of the input sequence simultaneously, it is able to capture diverse patterns and relationships within sequences. This not only improves the model's ability to handle long-range dependencies but also enhances its robustness and generalization capabilities.

Let's denote h as the number of attention heads. For each head i, we compute independent query, key, and value vectors:

$$Q_i = XW_{Qi},\ K_i = XW_{Ki},\ V_i = XW_{Vi}$$

Where W_{Qi}, W_{Ki}, and W_{Vi} are learnable weight matrices specific to each head. The attention outputs of all heads are concatenated and linearly transformed:

$$\text{MultiHead}(X) = \text{Concat}(\text{head}_1, \text{head}_2, ..., \text{head}_h)W_O$$

Where W_O is a learnable weight matrix.

Position-wise *feedforward* network

After computing attention, each token's representation is passed through a position-wise feedforward network. This network consists of two linear transformations with a ReLU activation function in between:

$$\text{FFN}(X) = \text{ReLU}(XW_1 + b_1)W_2 + b_2$$

Where W_1, W_2 are weight matrices, and b_1, b_2 are bias vectors.

Layer normalization and Residual connections

Layer normalization and residual connections are applied around each sub-layer (self-attention and position-wise feedforward network) to aid training stability and enable the model to effectively learn deep representations.

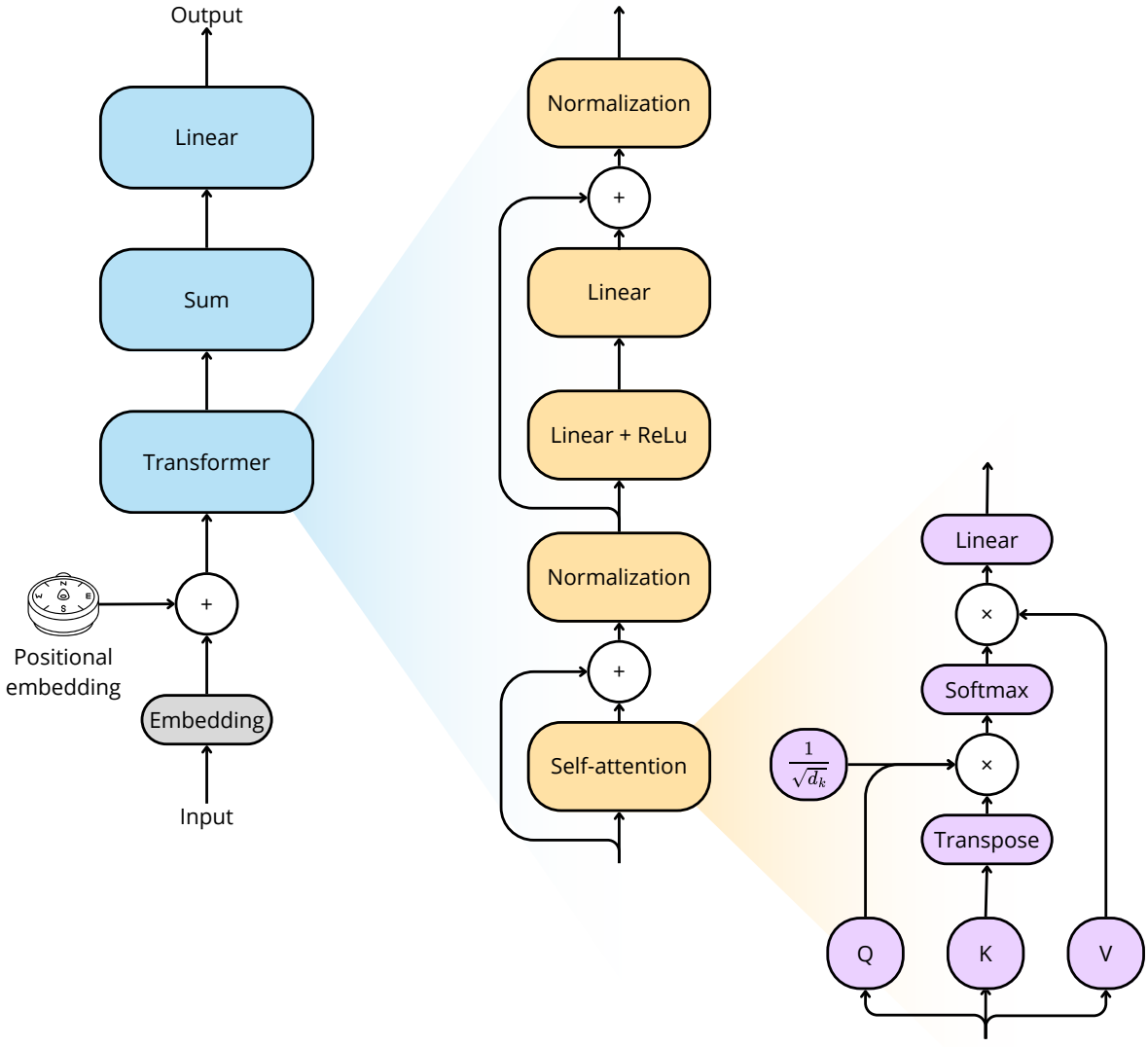

Figure 1: Schematic representation of the transformer architecture.

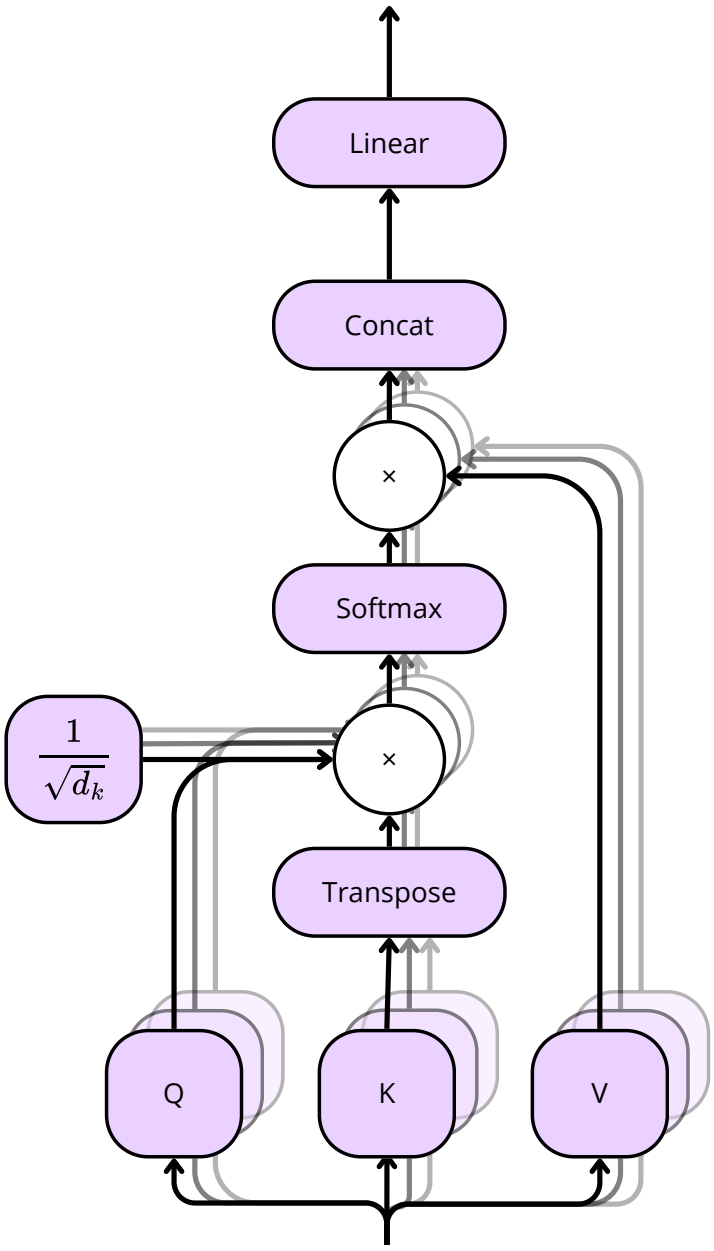

Figure 2: Schematic representation of the self-attention mechanism with multi-head attention.

Results

We tested different models by altering the following characteristics:

- Networks with two or more transformers (up to 4). Adding new transformers to the predictor improved the overall performance. However, we attempted to train a predictor with 6 transformers and it improved very slowly. An increase in the complexity of the model also means an increase in the number of epochs and thus an increase in the running time.
- Employing multi-head attention with both four heads and eight heads. Using multi-head seem to increase the accuracy.
- Adding domain adaptation at the end of training. The validation set was split into training and validation in order to slightly adapt the model to the *El Periódico* dataset.
- Sharing input and output embeddings. It made the accuracy drop but also shortened the training time.
- Increasing the embedding size. A model with an embedding size of 512 instead of 256 was trained, but in the last two epochs the learning rate was probably too high and the accuracy kept systematically decreasing.
- Having seen that when training more complex architectures, after the first one or two epochs the models seemed not to be able to keep improving, we decided to change the static learning rate to an adaptive learning rate (see [1]). The adaptive learning rate started at about 0.06, increased during the first 40000 iterations and then linearly decreased. It did not seem to perform as well because the best learning rate for the first epoch seemed to be the learning rate by default 0.001 and so the overall accuracy dwindled. We found that, in general, choosing the right learning rate was key in order to train our models. The code for the adaptive learning rate can be found in the next page.
- Changing the optimizer. Doing so produced worse results and so the trained models were not submitted to the competition to check their accuracy. The `Adam` optimizer outperformed all the rest. For that reason, all of the models included in this assignement employed the `Adam` optimizer.

In page 8, a table displaying the trained models and their corresponding loss and accuracy (both validation accuracy at the ending of training and the test accuracy procured by the submissions). Note that "-" means it uses the default configuration, ant that the number of parameters represents the difference with respect to the default value, to ease the comparison. The models without an associated test accuracy were not submitted but are still listed to perform an accurate comparison of models.

For `modA`, one line of the source code was changed to share embeddings.

```
1          # change in the Predictor class
2          self.lin.weight = self.emb.weight
```

The class `MultiheadAttention` was extracted from [2] and the `Transformer` class was altered to use multi-head attention, see the code below.

```python
class TransformerLayer(nn.Module):
    def __init__(self, d_model, dim_feedforward=512, dropout=0.1, /
    activation="relu", heads=1):
        super().__init__()
#         self.self_attn = SelfAttention(d_model)
        self.mhattn = MultiHeadAttention(heads, 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.mhattn(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
```

Finally, in order to perform domain adaptation, a train-test split of 0.80/0.20 was performed and once the training finished, some models underwent some fine-tuning, with a learning rate of 10^{-5}.

The following is the code for the learning rate adaptation, extracted from [3].

```python
class ScheduledOptim():
    '''Class for learning rate scheduling'''

    def __init__(self, optimizer, d_model, n_warmup_steps):
        self._optimizer = optimizer
        self.n_warmup_steps = n_warmup_steps
        self.n_current_steps = 0
        self.init_lr = np.power(d_model, -0.5)

    def step_and_update_lr(self):
        "Step with the inner optimizer"
        self._update_learning_rate()
        self._optimizer.step()

    def zero_grad(self):
```

```
        "Zero out the gradients by the inner optimizer"
        self._optimizer.zero_grad()

    def _get_lr_scale(self):
        return np.min([
            np.power(self.n_current_steps, -0.5),
            np.power(self.n_warmup_steps, -1.5) * self.n_current_steps])

    def _update_learning_rate(self):
        ''' Learning rate scheduling per step '''

        self.n_current_steps += 1
        lr = self.init_lr * self._get_lr_scale()

        for param_group in self._optimizer.param_groups:
            param_group['lr'] = lr
```

The scheduled optimizer was created with the following hyperparameters, inspired by the values of the original paper.

```
optimizer = torch.optim.Adam(model.parameters())
scheduledoptim = ScheduledOptim(optimizer, 256*4, 4000)
```

Id	*Layers*	*Multi-head* (n°of heads)	*Domain adaptation*	*Shared embed.*	*Embed. size*	*Learning rate*	*N° of params*	*Training time*	*Valid. Loss*	*Valid. Accuracy*	*Test Accuracy*
`def`	1	NO	NO	NO	256	0.001	51,729,664	3:37 h	4.12	35.2%	33.70%
`modA`	1	-	-	YES	-	-	-25,600,512	3:29 h	4.27	33.7%	31.87%
`modB`	1	8	YES	-	-	-	+0	3:44 h	3.97	36.2%	33.45%
`modC`	2	-	-	-	-	-	+527,104	4:03 h	4.00	36.3%	32.76%
`modD`	2	4	YES	-	-	-	+527,104	4:10 h	4.11	37.5%	34.00%
`modE`	1	4	YES	-	512	-	+52,253,440	5:54 h	4.15	35.5%	32.48%
`modF`	4	4	-	-	-	-	+1,581,312	5:12 h	3.95	36.9%	32.90%
`modG`	4	4	YES	-	-	-	+1,581,312	5:18 h	3.73	39.1%	34.61%
`modH`	4	4	YES	-	-	10^{-3} / 10^{-5}	+1,581,312	7:12 h	3.33	41.9%	37.95%
`modI`	3	4	-	-	-	adaptive	+1,054,208	3:38 h	4.09	35.8%	-
`modJ`	1	8	YES	-	-	0.01	+0	3:43 h	4.35	32.9%	-
`modK`	6	8	-	-	-	-	+1,581,312	11:37 h	5.80	18.7	-

Table 1: Accuracies obtained with the different models.

Conclusions

The best model architecture turned out to be the predictor composed by four transformers, each trained for four epochs initially. Following this, two additional epochs were conducted with a reduced learning rate; however, the second epoch failed to show improvement. To refine the model further, domain adaptation techniques were applied for a total of 10 epochs with a significantly lower learning rate of 0.00001 and there was a notable increase in the accuracy of the *El Periódico* dataset, until surpassing 40%.

Exploring the space of hyper-parameters and altering the architecture of the model allowed for these results. Some other observations were also noted, and are described below.
Regarding the number of transformers, optimal performance is achieved with four transformers. Models with only two transformers have a somewhat worse performance, while the one with six prove overly complex and resource-intensive: it probably should have been trained for more epochs and with a higher learning rate at the beginning. A predictor with 5 transformer was not tested, which could be a step to implement and further improve the performance.

An adaptive learning rate strategy, despite its theoretical appeal, yielded inferior results in practice. It would probably had worked better with a more complex architecture. Increasing the number of training epochs with a lower learning rate may be a more effective approach. Moreover, determining the optimal learning rate at the initial training phase is critical for model performance. Different values of learning rate were tested and $lr = 0.001$, the default value, seemed to be the top choice.

Domain adaptation helps in enhancing the accuracy of the models. At the beginning, we tried fine tuning with the default learning rate, but it seemed not to improve the performance of the models and sometimes it even reduced it. For that reason, we opted for a smaller learning rate for this final stage, and the results were quite positive.

The use of multi-head attention boosted the models, still not showing much difference between the choice of 4 heads or 8 heads.

Finally, sharing input and output embedding gave the worst results. Increasing the size of the embedding may be a way to counteract this.

References

[1] Ashish Vaswani et al. "Attention is all you need". In: *Advances in neural information processing systems* 30 (2017).

[2] Samuel Lynn Evans. *Transformer/Sublayers.py*. https://github.com/SamLynnEvans/Transformer/blob/master/Sublayers.py. 2023.

[3] Chan CheeKean. *DataScience/13 - NLP/C04 - BERT (Pytorch Scratch)*. https://github.com/ChanCheeKean/DataScience/blob/main/13%20-%20NLP/C04%20-%20BERT%20(Pytorch%20Scratch).ipynb. 2024.