Deep Learning (IST, 2023-24)

Homework 2

André Martins, Fábio Vital, João Santinha

Deadline: Saturday, January 6, 2024.

Please turn in the answers to the questions below in a PDF file, together with the code you implemented to solve them (when applicable).

IMPORTANT: Please write 1 paragraph indicating clearly what was the contribution of each member of the group in this project. A penalization of 10 points will be applied if this information is missing.

Please submit **a single zip file** in Fenix under your group's name.

Question 1 (30 points)

1. (5 points) Consider the self-attention layer of a transformer with a single attention head, which performs the computation $\boldsymbol{Z} = \mathrm{Softmax}(\boldsymbol{Q}\boldsymbol{K}^\top)\boldsymbol{V}$, where $\boldsymbol{Q} \in \mathbb{R}^{L\times D}$, $\boldsymbol{K} \in \mathbb{R}^{L\times D}$, and $\boldsymbol{V} \in \mathbb{R}^{L\times D}$, for a sequence length L and hidden size D. What is the computational complexity, in terms of L, of computing $\boldsymbol{Z}$? Briefly explain why this could be problematic for long sequences.

2. (10 points) We will show that by suitably approximating the softmax transformation the computational complexity above can be reduced. Consider the McLaurin series expansion for the exponential function,

$$\exp(t) = \sum_{n=0}^{\infty} \frac{t^n}{n!} = 1 + t + \frac{t^2}{2} + \frac{t^3}{6} + \dots$$

   Using only the first three terms in this series, we obtain $\exp(t) \approx 1 + t + \frac{t^2}{2}$. Use this approximation to obtain a feature map $\phi : \mathbb{R}^D \to \mathbb{R}^M$ such that, for arbitrary $\boldsymbol{q} \in \mathbb{R}^D$ and $\boldsymbol{k} \in \mathbb{R}^D$ we have $\exp(\boldsymbol{q}^\top \boldsymbol{k}) \approx \phi(\boldsymbol{q})^\top \phi(\boldsymbol{k})$. Express the dimensionality of the feature space M as a function of D. What would be the dimensionality if you used $K \geq 3$ terms in the McLaurin series expansion (as a function of D and K)?

3. (10 points) Using the approximation $\exp(\boldsymbol{q}^\top \boldsymbol{k}) \approx \phi(\boldsymbol{q})^\top \phi(\boldsymbol{k})$, and denoting by $\Phi(\boldsymbol{Q}) \in \mathbb{R}^{L\times M}$ and $\Phi(\boldsymbol{K}) \in \mathbb{R}^{L\times M}$ the matrices whose rows are (respectively) $\phi(\boldsymbol{q}_i)$ and $\phi(\boldsymbol{k}_i)$, where $\boldsymbol{q}_i$ and $\boldsymbol{k}_i$ denote (also respectively) the i^{th} rows of the original matrices $\boldsymbol{Q}$ and $\boldsymbol{K}$, show that the self-attention operation can be approximated as $\boldsymbol{Z} \approx \boldsymbol{D}^{-1}\Phi(\boldsymbol{Q})\Phi(\boldsymbol{K})^\top \boldsymbol{V}$, where $\boldsymbol{D} = \mathbf{Diag}(\Phi(\boldsymbol{Q})\Phi(\boldsymbol{K})^\top \mathbf{1}_L)$ (here, $\mathbf{Diag}(\boldsymbol{v})$ denotes a matrix with the entries of vector $\boldsymbol{v}$ in the diagonal, and $\mathbf{1}_L$ denotes a vectors of ones with size L).

4. (5 points) Show how we can exploit the above approximation to obtain a computational complexity which is linear in L. How does the computational complexity depend on M and D?

Question 2 (35 points)

Image classification with CNNs. In this exercise, you will implement a convolutional neural network to perform classification using the OCTMNIST dataset.

As previously done in Homework 1, you will need to download the OCTMNIST dataset. You can do this by running the following command in the homework directory:

```
python download_octmnist.py
```

Python skeleton code is provided (`hw2-q2.py`). You will now try out convolutional networks. For this exercise, we recommend you use a deep learning framework with automatic differentiation (suggested: Pytorch).

1. (20 points) Implement a simple convolutional network with the following structure:

   - A convolution layer with 8 output channels, a kernel of size 3x3, stride of 1, and padding chosen to preserve the original image size. Note: Assign convolution layer initialization to a variable called `self.conv1`.
   - A rectified linear unit activation function.
   - A max pooling with kernel size 2x2 and stride of 2.
   - A convolution layer with 16 output channels, a kernel of size 3x3, stride of 1, and padding of zero.
   - A rectified linear unit activation function.
   - A max pooling with kernel size 2x2 and stride of 2.
   - An affine transformation with 320 output features (to determine the number of input features use the number of channels, width and height of the output of the second block. Hint: The number of input features = *number of output channels* $\times$ *output width* $\times$ *output height*).
   - A rectified linear unit activation function.
   - A dropout layer with a dropout probability of 0.7.
   - An affine transformation with 120 output features.
   - A rectified linear unit activation function.
   - An affine transformation with the number of classes followed by an output LogSoftmax layer.

   Hint: use the functions `nn.Conv2d` and `nn.MaxPool2d`.

   Train your model for 15 epochs using `SGD` tuning only the learning rate on your validation data, using the following values: 0.1, 0.01, 0.001. Report the learning rate of best configuration and plot two things: the training loss and the validation accuracy, both as a function of the epoch number.

2. (10 points) Implement and asses a similar network but where the max-pooling layers were removed and `self.conv1` and `self.conv2` are different. `self.conv1` is a convolution layer with 8 output channels, a kernel of size 3x3, padding of 1 and stride of 2, and `self.conv2` is a convolution layer with 16 output channels, a kernel of size 3x3, padding of 0, and stride of 2. Modify the `__init__` and the `forward` functions to use the `no_maxpool` variable and ensure the ability to switch between current and previous definitions of the `self.conv1` and `self.conv2` layers and application or not of the max-pooling layer. Report the performance of this network using the optimal hyper-parameters defined in the previous question.

3. (5 points) Implement the function get_number_trainable_params to determine the number of trainable parameters of CNNs from the two previous questions. What justifies the difference in terms of performance between the networks?

Question 3 (35 points)

Automatic Speech Recognition Automatic speech recognition (ASR) is a deep-learning application that enables the automatic transcription and translation of spoken language into text. The task you will address is the transcription of the LJ Speech dataset (v 1.1). This small dataset has around 13000 short audio clips and the corresponding transcription of reading passages from several books. You will use and compare two different encoder-decoder architectures to solve this problem. Figure 1 depicts a general overview of an encoder-decoder architecture for the ASR application.

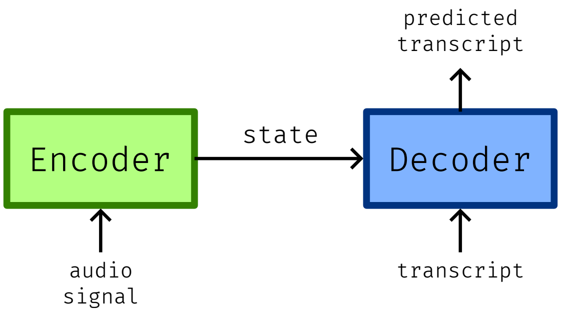


Figure 1: Encoder-decoder architecture to solve ASR task. The encoder receives a processed version of an audio signal and outputs a hidden representation. The decoder receives and merges the received transcript (sequence of tokens) and the encoder's output, predicting the next token of each input token.

The challenge proposed in questions 1 and 2 aims to implement the forward pass of different decoder implementations and compare their performance. We will skip the definition of the encoder's architecture since the entire implementation is already present in the code provided (`hw2-q3.ipynb`). For completeness, the encoder's architecture appears as supplementary material in Appendix B. In question 1, the decoder employs a recurrent-based approach to process the input, whereas an attention-based mechanism is used in question 2. Both decoders receive as input the text to decode as a sequence of tokens and the encoder's output. For each input token, the decoder's output will be a the logits corresponding to the probability of choosing each token in the vocabulary as the next token in the sequence.

1. (10 points) The decoder structure to employ is the following (see also Figure 2a):

    1. Embedding layer to convert the input text tokens into dense vectors, also called token embeddings.
    2. The token embeddings pass through a normalization layer followed by an LSTM.
    3. A residual cross-attention block will merge and attend the LSTM's output with the encoder's output. The query input will receive the LSTM's output, and the key and value inputs receive the encoder's output. For more details, refer to Appendix C.
    4. Normalize the residual block output using a normalization layer.
    5. The output of the attention mechanism passes through a single linear layer (classifier) to scale the hidden representation to the desired output dimension, in this case, the vocabulary size $|V| = 35$.

Go to the notebook and implement the `_forward` method of the `TextDecoderRecurrent` class. Note that every layer is already initialized in the `TextDecoderRecurrent`'s `__init__` method. Next, run the cell under the markdown cell `(a)`. Plot the comparison of the training and validation loss over epochs and the string similarity scores obtained in the validation set. Also, report the final test loss and the string similarity scores obtained with the test set.

2. (10 points) The decoder structure to employ is the following (see also Figure 2b):

    1. Embedding layer to convert the input text tokens into dense vectors.
    2. Add a position embeddings matrix to the token embeddings.
    3. The token embeddings pass through a residual attention layer as input for the queries, keys, and values. Additionally, this layer receives a masking matrix to ensure that the prediction of the current token does not attend to future tokens in the input. For more details, refer to Appendix A.
    4. A residual cross-attention block will merge and attend the previous attention's output with the encoder's output. The query input will receive the attention layer output, and the key and value inputs receive the encoder's output. For more details, refer to Appendix C.
    5. Normalize the residual block output using a normalization layer.
    6. The output of the normalization layer passes through a single linear layer (classifier) to scale the hidden representation to the desired output dimension, in this case, the vocabulary size $|V| = 35$.

    Go to the notebook and implement the `forward` method of the `TextDecoderTramsformer` class. Note that every layer is already initialized in the `TextDecoderTransformer`'s `__init__` method. Next, run the cell under the markdown cell `(a)`. Plot the comparison of the training and validation loss over epochs and the string similarity scores obtained in the validation set. Also, report the final test loss and the string similarity scores obtained with the test set.

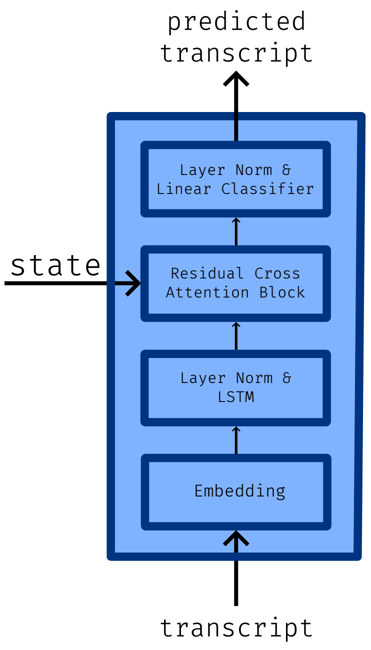
3. (10 points) Comment on the differences in the test results obtained using the decoder architectures in questions 1 and 2. *Note: start by illustrating how the LSTM (in question 1) and the attention mechanism (in question 2) process the text input.*

4. (5 points) Comment on the score values obtained by each string similarity score used. Why each score has different values based on what each similarity tries to evaluate?

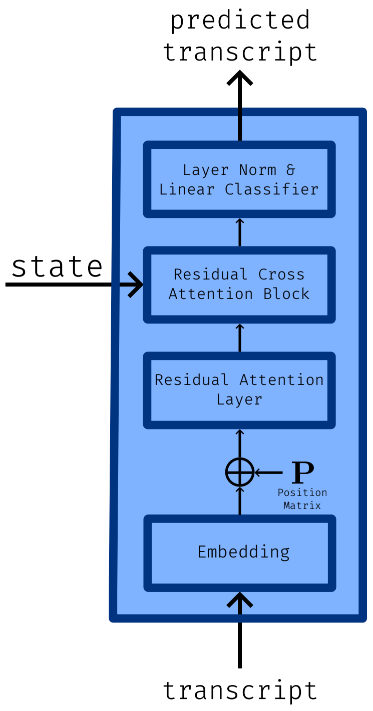
A Multi-Head Attention Mechanism & Residual Attention Layer

The attention mechanism used in both encoder and decoder architectures is the *scaled dot-product* attention, defined as $\text{Attention}(\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}) := \text{softmax}(\frac{\boldsymbol{QK}}{\sqrt{d_k}})\boldsymbol{V}$. Normally, the queries, keys, and values are packed together into matrices $\boldsymbol{Q}$, $\boldsymbol{K}$, and $\boldsymbol{V}$, respectively. Furthermore, instead of applying a single attention pass, this function is parallelized k times, where each attention head computes a different pass. To maintain the same output dimension, the output of each head is concatenated and projected using a single linear layer into the desired dimension.

$$\text{MultiHeadAttention}(\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V} := \text{Concat}(\text{head}_1, \ldots, \text{head}_k)\boldsymbol{W}^O$$
$$\text{where head}_i = \text{Attention}(\boldsymbol{Q}\boldsymbol{W}_i^Q, \boldsymbol{K}\boldsymbol{W}_i^K, \boldsymbol{V}\boldsymbol{W}_i^V).$$



(a) RNN based architecture to process text input - Question 1.



(b) Attention based architecture to process text input - Question 2.

Figure 2: Decoder architectures to implement.

If necessary, the projections $\boldsymbol{W}_i^Q$, $\boldsymbol{W}_i^K$, and $\boldsymbol{W}_i^V$ reduce or increase the queries, keys, and values dimensions, respectively. $\boldsymbol{W}^O$ is the linear layer that reduces the dimensionality of the concatenated outputs from each head.

A residual attention layer is simply the combination of the following layers: $\text{LayerNorm}(\boldsymbol{x} + \text{MultiHeadAttention}(\boldsymbol{x}', \boldsymbol{x}', \boldsymbol{x}'))$, where $\boldsymbol{x}' = \text{LayerNorm}(\boldsymbol{x})$.

B Encoder's Architecture

The encoder's architecture is based on the Transformer, comprising an embedding mechanism followed by a stack of $N_{\text{enc}} = 4$ identical blocks.

Since the encoder's input is an audio signal in the form of a mel-scale conversion of a spectrogram (frequency of a signal), the embedding mechanism starts by applying a sequence of two 1D convolutions, each followed by the GELU activation function. Both convolutions have a kernel size of 3 and padding of 1. Additionally, the second convolution reduces the time dimension (last dimension) by half by using a stride of 2. Afterward, add absolute sinusoidal position embeddings to the hidden representation before giving it to the stack of encoder layers.

Regarding the stack of encoder blocks, each block comprises two consecutive sub-blocks. All sub-blocks have the same design: the input passes through a predefined module, followed by applying layer normalization to the resulting sum of the module's output with a residual connection of the input, $\text{SubBlock}(\boldsymbol{x}) = \text{LayerNorm}(\boldsymbol{x} + \text{Module}(\text{LayerNorm}(\boldsymbol{x})))$. The modules for the first and second sub-layers are:

1. Multi-Head Attention layer, $\text{Module}(\boldsymbol{x}) = \text{MultiHeadAttention}(\boldsymbol{x}, \boldsymbol{x}, \boldsymbol{x})$, where $\boldsymbol{x}$ is given as the layer's query, key, and value inputs, respectively. The hidden dimension is $h_{\text{enc}} = 200$ and the mechanism has $k_{\text{enc}} = 2$ heads.

2. Feed-forward network, $\text{Module}(\boldsymbol{x}) = \text{MLP}(\boldsymbol{x})$, having a single hidden layer, with the GELU function as the activation function. The hidden layer dimension is $h_{\text{enc}} \times 2$, and the output dimension is h_{enc}. GELU is a smooth variation of ReLU, $\text{GELU}(x) = xP(X \leq x) \approx 0.5x(1 + \tanh\left[\sqrt{2/\pi}(x + 0.044715x^3)\right])$, where $X \sim \mathcal{N}(0, 1)$.

C Residual Cross-Attention Block

The residual attention block is a refined version of the attention-based blocks introduced in Appendix B. In the cross-attention design, each block has two consecutive sub-blocks. The second sub-block has the same architecture introduced in Appendix B, $\text{SubBlock}(\boldsymbol{x}) = \text{LayerNorm}(\boldsymbol{x} + \text{Module}(\text{LayerNorm}(\boldsymbol{x})))$. For the first sub-block, we have $\text{SubBlock}(\boldsymbol{x}, \boldsymbol{z}) = \text{LayerNorm}(\boldsymbol{x} + \text{Module}(\text{LayerNorm}(\boldsymbol{x}), \boldsymbol{z}))$. The modules employed for the sequence of two sub-blocks are as follows:

1. Multi-Head Attention layer, $\text{Module}(\boldsymbol{x}, \boldsymbol{z}) = \text{MultiHeadAttention}(\boldsymbol{x}, \boldsymbol{z}, \boldsymbol{z})$, where $\boldsymbol{x}$ is the output of previous sub-block and given as the query. Typically, since the residual block appears inside the decoder, $\boldsymbol{z}$ is often the encoder's output.

2. Feed-forward network, $\text{Module}(\boldsymbol{x}) = \text{MLP}(\boldsymbol{x})$, having a single hidden layer, with the GELU function as the activation function.