Deep Learning (IST, 2024-25)

Homework 2

André Martins, Mario Figueiredo, Chryssa Zerva, Ben Peters,
Tomás Soares da Costa, Pavlo Vasylenko, André Duarte, Emmanouil Zaranis

Deadline: Friday, January 10, 2025.

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 (25 points)

Modern Hopfield networks, the Convex-Concave Procedure, and Transformers Hopfield networks are a type of networks that exhibit associative memory capabilities (Hopfield, 1982). Let $\boldsymbol{X} \in \mathbb{R}^{N\times D}$ be a matrix whose rows hold a set of examples $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_N$ ("memory patterns"), where each $\boldsymbol{x}_i \in \mathbb{R}^D$, and let $\boldsymbol{q}_0 \in \mathbb{R}^D$ be a query vector (called a "state pattern"). The Hopfield network iteratively builds updates $\boldsymbol{q}_t \mapsto \boldsymbol{q}_{t+1}$ for $t \in \{0, 1, \ldots\}$ according to a certain rule and, under certain conditions, these dynamical trajectories converge to a fixed point attractor state $\boldsymbol{q}^*$ which corresponds to one of the memorized examples.

The update rule corresponds to the minimization of an "energy" function of the form $E(\boldsymbol{q}) = -\sum_{i=1}^N F(\boldsymbol{q}^\top \boldsymbol{x}_i)$, where $F : \mathbb{R} \to \mathbb{R}$. Classic Hopfield networks use $F(u) = u^2/2$, so the energy function can be written as $E(\boldsymbol{q}) = (1/2)\, \boldsymbol{q}^\top \boldsymbol{W} \boldsymbol{q}$, where $\boldsymbol{W} = \boldsymbol{X}^\top \boldsymbol{X} \in \mathbb{R}^{D\times D}$ are called the Hopfield network parameters; when $D \ll N$, $\boldsymbol{W}$ can be seen as a "compressed memory". Intuitively, each element $W_{i,j}$ of $\boldsymbol{W}$ can be seen as a connection weight between elements i and j of the network, which is proportional to the sum of the products of components i and j of all the memory patterns: $W_{i,j} = \sum_{n=1}^N x_{n,i}\, x_{n,j}$, where $x_{n,i}$ is the i-th component of the n-th pattern. In the classic Hopfield network, the state vector $\boldsymbol{q}$ is further constrained to be binary ($\boldsymbol{q}_t \in \{-1, 1\}^D$), and the update rule is $\boldsymbol{q}_{t+1} = \mathrm{sign}(\boldsymbol{W}\boldsymbol{q}_t)$.

In more recent work, Krotov and Hopfield (2016) proposed other energy functions, leading to so-called "modern Hopfield networks", in which the state vector $\boldsymbol{q}_t \in \mathbb{R}^D$ is unconstrained. One of the new variants (Ramsauer et al., 2020) uses

$$E(\boldsymbol{q}) = -\mathrm{lse}(\beta, \boldsymbol{X}\boldsymbol{q}) + \beta^{-1}\log N + +\frac{1}{2}\boldsymbol{q}^\top \boldsymbol{q} + \frac{1}{2}M^2, \tag{1}$$

where $M = \max_i \|\boldsymbol{x}_i\|$ and $\mathrm{lse}(\beta, \boldsymbol{z})$ is the log-sum-exp function with "temperature" β^{-1}:

$$\mathrm{lse}(\beta, \boldsymbol{z}) = \beta^{-1}\log \sum_{i=1}^N \exp(\beta z_i). \tag{2}$$

The log-sum-exp function is the soft version of the vector maximum function, $\text{vecmax}(\boldsymbol{z}) = \max\{z_1, ..., z_N\}$, in the sense that

$$\text{vecmax}(\boldsymbol{z}) \leq \text{lse}(\beta, \boldsymbol{q}) \leq \text{vecmax}(\boldsymbol{z}) + \frac{1}{\beta} \log N,$$

thus $\lim_{\beta \to \infty} \text{lse}(\beta, \boldsymbol{z}) = \text{vecmax}(\boldsymbol{z})$.

In this question, you will show that there is an interesting relation between the updates in this modern Hopfield network and the attention layers in transformers.[1]

1. (10 points) First, you will show that the energy (1) can be written as $E(\boldsymbol{q}) = E_1(\boldsymbol{q}) + E_2(\boldsymbol{q})$, where E_1 is a convex function and E_2 is a concave function. To do this, define $E_1(\boldsymbol{q}) = \frac{1}{2}\boldsymbol{q}^\top \boldsymbol{q} + \beta^{-1} \log N + \frac{1}{2}M^2$ and $E_2(\boldsymbol{q}) = -\text{lse}(\beta, \boldsymbol{X}\boldsymbol{q})$. Show that the gradients of E_1 and $-E_2$ are, respectively,

$$\nabla E_1(\boldsymbol{q}) = \boldsymbol{q}, \qquad \nabla(-E_2(\boldsymbol{q})) = \boldsymbol{X}^T \text{softmax}(\beta \boldsymbol{X}\, \boldsymbol{q}),$$

compute the Hessians of E_1 and $-E_2$, and show that they are positive semi-definite [2] (psd).

2. (10 points) There is an iterative algorithm known as *convex-concave procedure* (CCCP), which under certain mild conditions is guaranteed to find local minima of functions that are written as a sum of a convex and a concave function (Yuille and Rangarajan, 2003). The algorithm works as follows: at each iteration t,

   - linearize the concave function E_2 using a first-order Taylor approximation around $\boldsymbol{q}_t$,

   $$E_2(\boldsymbol{q}) \approx \tilde{E}_2(\boldsymbol{q}) := E_2(\boldsymbol{q}_t) + \left(\nabla E_2(\boldsymbol{q}_t)\right)^T (\boldsymbol{q} - \boldsymbol{q}_t);$$

   - computes a new iterate by solving the convex optimization problem

   $$\boldsymbol{q}_{t+1} = \arg\min_{\boldsymbol{q}} E_1(\boldsymbol{q}) + \tilde{E}_2(\boldsymbol{q}).$$

   Show that applying the CCCP algorithm with the Hopfield energy function (1) yields the updates

   $$\boldsymbol{q}_{t+1} = \boldsymbol{X}^\top \text{softmax}(\beta \boldsymbol{X} \boldsymbol{q}_t). \tag{3}$$

3. (5 points) When $\beta = \frac{1}{\sqrt{D}}$, compare the first update $\boldsymbol{q}_t \mapsto \boldsymbol{q}_{t+1}$ with the computation performed in the cross-attention layer of a transformer with a single attention head, identity projection matrices $\boldsymbol{W}_K = \boldsymbol{W}_V = \boldsymbol{I}$ with input $\boldsymbol{X} \in \mathbb{R}^{N \times D}$. Assume the query matrix $\boldsymbol{Q} = [\boldsymbol{q}_t^{(1)}, \ldots, \boldsymbol{q}_t^{(N)}]^T \in \mathbb{R}^{N \times D}$ is already computed.

[1] If you are interested in this topic, see the long blog post in `https://ml-jku.github.io/hopfield-layers/` for more information.

[2] One of the two following facts may be useful.

- For a convex function $g : \mathbb{R} \to \mathbb{R}$, Jensen's inequality states that, for any L, any set of non-negative numbers $w_1, ..., w_L$, such that $\sum_{i=1}^{L} w_i = 1$, and any $x_1, ..., x_L$, the following holds: $g(\sum_{i=1}^{L} w_i x_i) \leq \sum_{i=1}^{L} w_i\, g(x_i)$.
- A square matrix $\boldsymbol{A}$ is said to be diagonally dominant if $|A_{ii}| \geq \sum_{j \neq i} |A_{i,j}|$, for any i. A symmetric diagonally dominant matrix with non-negative diagonal elements is positive semi-definite.

Note on GPUs for programming exercises Q2 & Q3

If you are interested in running your programs for the following questions (Q2 and Q3) on a GPU, you can try to use Google Colaboratory or Lightning Studio keeping in mind the guidelines here: `https://tinyurl.com/gpudlguide`. Do not forget to move your model and data to CUDA [3].

Question 2 (35 points)

Image classification with CNNs. In this exercise, you will implement a convolutional neural network to perform classification using the same Intel Image classification dataset as used previously in Homework 1.

As previously done in Homework 1, you will need to download the preprocessed Intel Image Classification dataset that can be found in:

`https://drive.google.com/file/d/16AzJIrmra4qWdY7wU9N1ew4bwadmNt4j`

You will now try out convolutional networks. For this exercise, we recommend you use a deep learning framework with automatic differentiation (suggested: Pytorch). We also provide Python skeleton code which supports Pytorch implementations (`hw2-q2.py`). Hints on suggested functions to use also assume use of Pytorch modules.

1. (15 points) Implement a simple convolutional network with the following structure:

   IMPORTANT: This may take a while to run. Think about question 2.4 or the theoretical ones in between.

   - Implement a convolutional block by editing `ConvBlock` class to add:
     - A convolution layer that takes as parameters input and output channels, a kernel of size, stride, and padding. For the kernel size use 3x3, for stride use 1 and for padding use 1.
     - A rectified linear unit activation function.
     - A max pooling with kernel size 2x2 and stride 2.
     - A dropout layer with a dropout probability of 0.1.

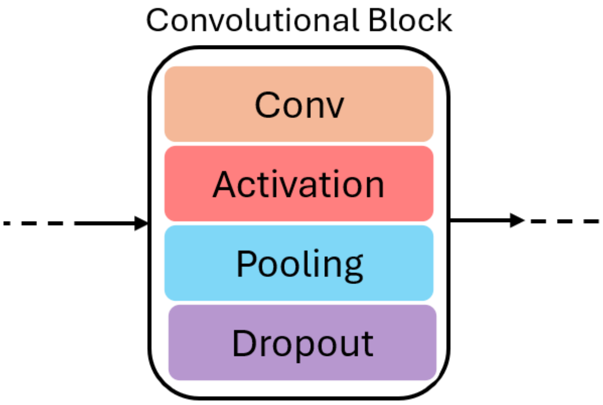

Figure 1: Convolutional Block. Pooling - max pooling

   - Implement a sequence of 3 convolutional blocks with the following output channel sizes: 32, 64, and 128.

[3] Refer to the pytorch tutorial from Practical 5 for more information.

- Flatten the output of the convolution part of the CNN. Hint: The number of input features = *number of output channels* × *output width* × *output height*.
- Multilayer Perceptron (MLP) block using the flattened vector as input:
    - An affine transformation with 1024 output features.
    - A rectified linear unit activation function.
    - A dropout layer with a dropout probability of 0.1.
    - An affine transformation with 512 output features.
    - A rectified linear unit activation function.
    - An affine transformation with the number of classes followed by an output LogSoftmax layer.

-device

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

Train your model for 40 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 with the following changes:

    1. Implement batch normalization in each convolutional block (Hint: use the `nn.BatchNorm2d` function). Figure 2 shows the forward pass of the modified convolution block.

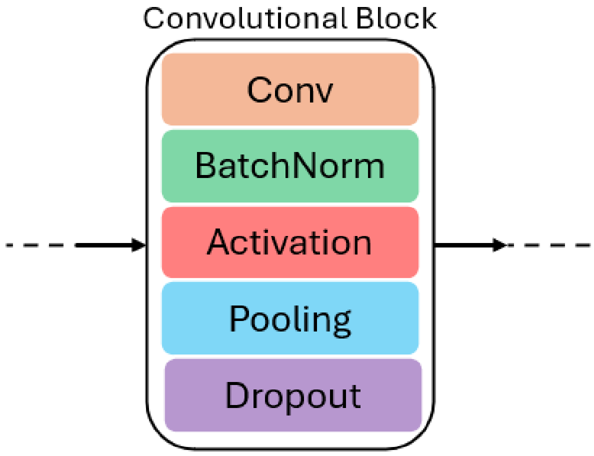

Figure 2: Convolutional Block with batch normalization.

    2. Replace transformation of flattening the output of the convolutional blocks with a global average pooling layer, i.e., right after the last convolution block. Hint: use the `nn.AdaptiveAvgPool2d` function with (1,1) kernel. Alternative, you can just average the output from the last convolutional block over the *height* and *width* dimensions together.
    3. Incorporate a batch normalization layer in the MLP block. before the dropout layer. Hint: use the `nn.BatchNorm1d` function.

Incorporate `batchnorm` variable to ensure the ability to switch between current and previous definitions of the blocks of the CNN (Hint: use the `nn.Identity` function). Report the performance of this network using the optimal configuration 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 number of parameters and performance between the networks?

4. (5 points) Why do we use small kernels in the convolution layers instead of kernel of ($width \times height$). What effect do the pooling layers have?

Question 3 (40 points)

Phoneme-to-Grapheme Conversion Phoneme-to-Grapheme Conversion (P2G)[4] is the task of predicting how a word should be spelled based on its pronunciation in a transcription system such as the International Phonetic Alphabet. P2G, in common with many other tasks such as machine translation or transliteration, is often handled using encoder-decoder models based on either self-attention on recurrent neural networks. In such a model, the encoder module produces a contextual representation for each position in the source sequence; then, the decoder uses these representations to autoregressively generate the target sequence one token at a time.

1. In the following, you will implement a character-level recurrent sequence-to-sequence model that can solve the P2G task for a small dataset of English pronunciation extracted from WikiPron. This task uses two error metrics:

   - Character Error Rate (CER) is the levenshtein distance between the predicted sequences and the true spellings, divided by the length of the true spellings.
   - Word Error Rate (WER) is the percentage of examples for which the predicted sequence does not exactly match the true spelling.

   (a) (10 points) The dataset is provided in the `data/` folder. Implement a vanilla character-level encoder-decoder model with a bidirectional LSTM encoder an autoregressive LSTM decoder. Specifically, in the skeleton code, you will need to implement the method `forward()` of both the `Encoder` and the `Decoder` in `models.py` and then run `python hw2-q3.py`.

   After you have implemented these methods, train the model for 20 epochs using a learning rate of 0.003, a dropout rate of 0.3, a hidden size of 128, and a batch size of 64. The dropout layer should only be applied to the embeddings and the final outputs (in both the encoder and the decoder). You can run the training script with

   ```
   python hw2-q3.py train --checkpoint_name no-attn.pt
   ```

   Plot the validation CER at the end of each epoch.

   The code will automatically save the model checkpoint that records the minimum CER to the file `no-attn.pt`. Now run this checkpoint on the test set with the following command and report the CER and WER:

   ```
   python hw2-q3.py test --checkpoint_name no-attn.pt
   ```

   Hint: At each time-step, the LSTM decoder receives as input the embedding of the current word (`self.embedding`), then each output at each timestep has dropout applied (`self.dropout`) and the resulting decoder state is used in the next timestep. In the end, all decoder outputs are concatenated to form an output sequence.

[4] Not to be confused with Grapheme-to-Phoneme Conversion (G2P), which is the same task but with the source and target flipped.

(b) (20 points) In sequence-to-sequence models, the Bahdanau Attention mechanism allows the decoder to selectively focus on specific parts of the input sequence during decoding. It computes a weighted sum of encoder outputs, where the weights indicate the relevance of each encoder hidden state to the current decoding step. The mechanism can be described as follows:

- Let $\mathbf{h}_t \in \mathbb{R}^H$ be the encoder hidden state at time step t, and $\mathbf{s}_{t-1} \in \mathbb{R}^H$ be the decoder hidden state from the previous time step, where H is the hidden size.
- The alignment score e_{ti} for each encoder hidden state $\mathbf{h}_i$ with respect to the decoder hidden state $\mathbf{s}_{t-1}$ is given by:

$$e_{ti} = \mathbf{v}^\top \tanh(\mathbf{W}_s \mathbf{s}_{t-1} + \mathbf{W}_h \mathbf{h}_i),$$

  where:
  - $\mathbf{W}_s \in \mathbb{R}^{H \times H}$ is the weight matrix for the decoder hidden state,
  - $\mathbf{W}_h \in \mathbb{R}^{H \times H}$ is the weight matrix for the encoder hidden states,
  - $\mathbf{v} \in \mathbb{R}^H$ is the scoring vector.
- The alignment scores are normalized using a softmax function to compute the attention weights α_{ti}:

$$\alpha_{ti} = \frac{\exp(e_{ti})}{\sum_{j=1}^{n} \exp(e_{tj})},$$

  where n is the number of encoder time steps.
- The context vector $\mathbf{c}_t \in \mathbb{R}^H$ is computed as a weighted sum of encoder hidden states:

$$\mathbf{c}_t = \sum_{i=1}^{n} \alpha_{ti} \mathbf{h}_i.$$

- The context vector $\mathbf{c}_t$ is then combined with the decoder hidden state $\mathbf{s}_{t-1}$ and passed through a linear transformation to produce the attention-enhanced decoder state $\tilde{\mathbf{s}}_t$:

$$\tilde{\mathbf{s}}_t = \tanh(\mathbf{W}_{\text{out}}[\mathbf{c}_t; \mathbf{s}_{t-1}]),$$

  where:
  - $\mathbf{W}_{\text{out}} \in \mathbb{R}^{H \times 2H}$ is a learnable weight matrix, and
  - $[\mathbf{c}_t; \mathbf{s}_{t-1}]$ denotes the concatenation of $\mathbf{c}_t$ and $\mathbf{s}_{t-1}$.

Your tasks are to implement the `forward()` method of the `BahdanauAttention` class in `models.py` and slightly modify your `Decoder` implementation to account for the attention mechanism (it should modify the output of the LSTM at each timestep).

After finishing the implementation, train and test the model with the same hyperparameters as in the previous exercise, remembering to set the `--use_attn` flag and to select an informative `--checkpoint_name` such as `attn.pt`. Again, plot the validation CER for each epoch and report the best checkpoint's CER and WER on the test set.

(c) (10 points) Up to this point, we have used greedy decoding to generate hypotheses from our models: at each time step, the next token is the one that has the highest probability, conditioned on the source and previously generated tokens. Although this procedure is not guaranteed to find the *globally* most likely sequence, it is simple to implement and very effective in practice. However, in this exercise you will implement **nucleus sampling**, a technique which is commonly used with LLMs to generate text that is both high-quality and varied.

Nucleus sampling, also known as top-p sampling, dynamically selects tokens from a subset of the vocabulary based on their cumulative probability mass. The key idea is to use the shape of the probability distribution of the vocabulary to determine the set of tokens to sample from. Given a distribution $P(x \mid x_{1:t-1})$ over the vocabulary V at timestep t of the decoding process, we define the top-p vocabulary $V^{(p)} \subset V$ as the smallest set of tokens that their cumulative probability mass equals or exceeds a probability threshold p. More formally:

$$\sum_{x \in V^{(p)}} P(x \mid x_{1:t-1}) \geq p \tag{6}$$

In practice, this is equivalent to selecting the tokens with highest probabilities, whose cumulative probability mass exceeds or is equal to p. After selecting the tokens that belong to $V^{(P)}$, the original distribution is re-scaled to a new distribution, from which the next token is sampled:

$$P'(x \mid x_{1:i-1}) = \begin{cases} \frac{P(x|x_{1:t-1})}{p'}, & \text{if } x \in V^{(p)} \\ 0, & \text{otherwise.} \end{cases} \tag{7}$$

where $p' = \sum\limits_{x \in V^{(p)}} P(x \mid x_{1:t-1})$.

Your task is to implement the `nucleus_sampling` function in `hw2-q3.py`. After you have done so, reuse the model with attention from part (b) to generate predictions from the test set:

```
python hw2-q3.py test --checkpoint_name attn.pt --topp 0.8
```

When you run this code, first it will generate CER and WER as before. But then it will sample predictions 3 more times and show a few examples of cases where the samples are varied. It will also report an extra metric called WER@3: this is the percentage of examples for which *none* of the model's 3 predictions are correct. Report the test CER, WER, and WER@3.

References

Hopfield, J. J. (1982). Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the national academy of sciences*, 79(8):2554–2558.

Krotov, D. and Hopfield, J. J. (2016). Dense associative memory for pattern recognition. *Advances in neural information processing systems*, 29.

Ramsauer, H., Schäfl, B., Lehner, J., Seidl, P., Widrich, M., Adler, T., Gruber, L., Holzleitner, M., Pavlović, M., Sandve, G. K., et al. (2020). Hopfield networks is all you need. *arXiv preprint arXiv:2008.02217.*

Yuille, A. L. and Rangarajan, A. (2003). The concave-convex procedure. *Neural computation*, 15(4):915–936.