EBERHARD KARLS UNIVERSITÄT TÜBINGEN

Computer Science Department
Neuro-Cognitive Modeling

Summer Term 2024

Recurrent and Generative Neural Networks

Exercise Sheet 02

Release: November 04, 2024 Deadline: November 18, 2024 (12:00 pm)

General remarks:

- Download the file `exercisesheet02.zip` from the lecture site (ILIAS). This archive contains files (Python scripts), which are required for the exercises.
- All relevant equations can be found in the corresponding lecture slides. Ideally, the exercises should be completed in teams of two students. Larger teams are not allowed.
- **Add a brief documentation (pdf) to your submission**. The documentation should contain your **names**, **matriculation numbers**, and **email adresses** as well as protocols of your experiments, parameter choices, and a discussion of the results.
- Set up a Conda environment using the provided `environment.yml` file (included in the `exercisesheet02.zip` archive).
- Exercises that require GPU training include both `train.sbatch` and `train.sh` scripts for use on the TCML cluster. To run these, copy all source code files to your home directory on the cluster. Start training by executing:

```
sbatch train.sbatch
```

- When questions arise start a forum discussion in ILIAS or contact us via email:
  Fedor Scholz (`fedor.scholz@uni-tuebingen.de`)
  Manuel Traub (`manuel.traub@uni-tuebingen.de`)

Exercise 1: Time Series [50 points]

In this exercise, we will implement a simple RNN, LSTM, and Temporal Convolution Network (TCN) without pytorch's built-in classes for these architecture. Use only `Linear` layers to build your models.

We will compare the performance of these models on two variations of a memory task. The model receives a sequence containing mostly 0s, with a single 1, 2, or 3 placed randomly within. At the sequence's end, a "request token" is shown, after which the model should output the observed 1, 2, or 3.

In the `memory` task, train and validation sets have similar sequences. In the `memorygen` task, however, the training set includes sequences with tokens in the first half, while the validation set has them in the second half.

To train and validate your models, run `python main.py`. You can modify the lists of tasks and models at the end of the file.

(a) Implementing a Vanilla Recurrent Neural Network [10 points]

1. **RNN:** Implement the `RNN` class in `net.py` by making use of pytorch's `RNNCell` class. This class's forward method receives a whole sequence as input. Therefore, the hidden state needs to be initialized first. The class should output the model's last output only.
2. **RNNCell:** Implement the `RNNCell` class in `net.py` and use it in `RNN` instead of pytorch's version. This class receives only single steps of a sequence together with the corresponding hidden state. It should output the next hidden state.

(b) Implementing a Long Short-Term Memory [15 points]

1. **LSTM:** Implement the `LSTM` class in `net.py` by making use of pytorch's `LSTMCell` class. This class's forward method receives a whole sequence as input. Therefore, the hidden and cell states need to be initialized first. The class should output the model's last output only.
2. **LSTMCell:** Implement the `LSTMCell` class in `net.py` and use it in `LSTM` instead of pytorch's version. This class receives only single steps of a sequence together with the corresponding hidden and cell state. It should output the next hidden and cell state. You will get full points if you implement `LSTMCell` with a single `Linear` layer.

(c) Implementing a Temporal Convolution Network [15 points]

For simplicity, we here assume `kernel_size == stride == 3` and do not use any pooling, as presented in slides. How many layers do you need? If necessary, pad the sequence on the fly with 0s at the end.

1. **TCN:** Implement the `TCN` class in `net.py` by making use of pytorch's `Conv1d` class. This class's forward method receives a whole sequence as input. Use the figure in the slides for guidance.
2. **Conv1d:** Implement the `Conv1d` class in `net.py` and use it in `TCN` instead of pytorch's version. You will get full points if you implement `Conv1d` without a loop.

(d) Comparison [10 points]

Review the generated figures to analyze your models. The training and validation losses over time are saved in `*_losses.png`, while `*_prediction.png` shows an individual sequence alongside the model's prediction.

What observations can you make about each model's performance on the different tasks? Discuss and try to explain any differences you notice.

Exercise 2: Vision Transformers [50 points]

In this exercise, we will explore a predictive Vision Transformer (ViT) model designed to predict the next frame in a video sequence, given the previous $n-1$ frames. The model aims to capture the temporal progression and dynamics of a training video, consisting of gameplay footage collected from participants in our last team project. We will evaluate the model's ability to learn and predict complex temporal patterns effectively.

(a) Implementing the Self-Attention Mechanism [10 points]

To begin, complete the self-attention mechanism in the `SelfAttentionLayer` class found in `nn/attention.py`. The self-attention mechanism is crucial in ViTs, allowing the model to communicate between different image regions to build representations. Your tasks are:

1. **Compute Attention Scores:** Implement the query-key-value matrix multiplications to generate the attention scores.
2. **Apply Softmax Normalization:** Normalize the attention scores using the softmax function to obtain the attention weights.
3. **Compute Weighted Sum of Values:** Use the attention weights to compute a weighted sum of the values for each query, resulting in the self-attention layer's output.

(b) Implementing Patching and Positional Embeddings [10 points]

Vision Transformers operate on sequences of image patches instead of entire images. In this section, complete the patch embedding implementation within the `PatchEmbedding` class in `nn/attention.py`. You will also add positional embeddings to supply location-specific information to each patch. Specifically, your tasks are:

1. **Patch Generation:** Implement the function that divides the input image into non-overlapping patches.
2. **Linear Projection of Patches:** Flatten each patch and project it into a fixed-dimensional vector space.
3. **Add Positional Embeddings:** Add positional embeddings to each patch embedding to maintain spatial information across patches.

Once you have completed parts (a) and (b), you can train the model using `python -m model.main -cfg config/vit.json --train`. After training for a few epochs, you can generate predictions with `python -m model.scripts.predict -cfg config/vit.json --load out/vit.device0/quicksave.ckpt`.

(c) Analyzing Predictive ViT Performance [10 points]

Briefly discuss the predictive ViT model's performance after training. In which areas does the model perform well, and where does it struggle? Why does it fail to predict certain elements of the video?

(d) Exploring the Role of Self-Attention [10 points]

In this section, we will examine the role of the self-attention mechanism. Train the predictive ViT model without self-attention and observe its impact on performance. Your tasks are:

1. **Train the Model without Self-Attention:** Replace self-attention in the encoder with feed-forward network layers and attempt the masked reconstruction task. Observe if the network achieves reasonable performance without self-attention.
2. **Discuss Performance and Interpretability:** After training, analyze and discuss whether the model achieves meaningful results without self-attention. If it succeeds, provide reasons why feed-forward layers alone were sufficient. If it fails, explain why omitting self-attention negatively affected performance.

(e) Exploring the Impact of Positional Embeddings [10 points]

In this part, investigate how positional embeddings affect the model's performance. Your tasks are:

1. **Train the Model without Positional Embeddings:** Train the predictive ViT model without positional embeddings and assess the impact on its performance.
2. **Discuss Performance and Interpretability:** Analyze and discuss the model's performance without positional embeddings. If the model struggles to learn effectively, explain why omitting positional embeddings negatively impacted performance. If the model succeeds, discuss why positional embeddings may not have been necessary for this task.

(f) Bonus: Hyperparameter Tuning [20 points]

In this bonus section, try to identify optimal hyperparameters for the predictive ViT model (adjusting `num_frames`, `channels`, `num_heads`, `num_layers`, `patch_size`, etc., in the configuration file). Aim for a configuration that allows the model to either perfectly memorize the video or make predictions without noticeable artifacts. Submit a configuration file and model checkpoint demonstrating the best performance you achieved.

Provide a summary of your findings and insights in the documentation submitted with your code.