EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

Computer Science Department
Neuro-Cognitive Modeling

Summer Term 2024

Recurrent and Generative Neural Networks

Exercise Sheet 01

Release: October 21, 2024 Deadline: November 4, 2024 (12:00 pm)

General remarks:

- Download the file `exercisesheet01.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 `exercisesheet01.zip` archive).
- When questions arise start a forum discussion in ILIAS or contact us via email:
  Manuel Traub (`manuel.traub@uni-tuebingen.de`)

Exercise 1: Implementing Backpropagation [50 points]

(a) Backward Pass Implementation [30 points]

In this exercise, you are provided with a class for a simple feed-forward neural network in `Python-FeedForward/net.py`. The goal is to complete the `backward` method to implement backpropagation, computing gradients for each layer in the network.

1. **Gradient Calculation for Output Layer:** Using the Binary Cross-Entropy loss, derive the gradients at the output layer. For each output neuron, compute the error term $\delta_j^{(L)}$, given by

$$\delta_j^{(L)} = \frac{\partial E}{\partial z_j^{(L)}},$$

   where E is the error, and $z_j^{(L)}$ is the pre-activation output of neuron j in the final layer.

2. **Backpropagation through Hidden Layers:** Propagate the gradients backward to compute the error terms for the hidden layer neurons, $\delta_j^{(l)}$. For each layer l, the error term $\delta_j^{(l)}$ is computed recursively from the next layer's error terms.
3. **Weight Gradient Computation:** Using the error terms, calculate the partial derivatives of the weights:
$$\frac{\partial E}{\partial w_{ij}} = a_i^{(l-1)} \delta_j^{(l)},$$
where $a_i^{(l-1)}$ is the activation of the neuron i in the previous layer.

After implementing the gradients, ensure the `backward` method aggregates these gradients for each batch. Then, verify your implementation by observing the loss trend in `main.py` during training.

(b) Cross-Entropy and Softmax Activation [20 points]

In multi-class classification, the Cross-Entropy loss is commonly used together with the softmax activation function in the output layer. Answer the following questions:

1. **Why is Cross-Entropy typically paired with the Softmax activation function?**
2. **Derive the Gradient for Cross-Entropy with Softmax:** Given the softmax function,
$$\text{softmax}(z_j) = \frac{e^{z_j}}{\sum_k e^{z_k}},$$
and the Cross-Entropy loss,
$$L = -\sum_j y_j \log(\text{softmax}(z_j)),$$
derive the gradient of L with respect to z_j.
3. **Alternative Normalization Without Exponentiation:** Consider an alternative normalization function that produces probability-like outputs without exponentiation:
$$\text{normalized}(z_j) = \frac{z_j}{\sum_k z_k}.$$
Derive the gradient of the Cross-Entropy loss with this alternative function:
$$L = -\sum_j y_j \log\left(\frac{z_j}{\sum_k z_k}\right).$$
Compare your result to the gradient derived with softmax. Discuss how the absence of the exponential function affects the gradient structure and why the combination of Cross-Entropy with Softmax might be more effective in gradient-based optimization.

Exercise 2: Getting familiar with PyTorch [30 points]

In this exercise, we are going to work with PyTorch, a widely-used deep learning framework that provides automatic differentiation and efficient GPU computation, making it ideal for neural network development and experimentation. To successfully train the network, you must complete the following two parts of this exercise: implementing the backward pass (part a) and the Adam optimizer (part c).

(a) Backward Pass Implementation [10 points]

In this exercise, you are provided with a PyTorch implementation of a feed-forward neural network in `PyTorch-FeedForward/net.py`. The network uses custom autograd functions `LinearFunction` and `ReLUFunction`, where the `forward` methods are implemented, but the `backward` methods are incomplete. Your task is to complete these `backward` methods to enable gradient computation for training.

(b) Gradient Test Case Implementation [10 points]

To verify your backward pass implementations, complete the test case implementation in `test_gradients.py`, which compares the custom gradients computed by `LinearFunction` and `ReLUFunction` against PyTorch's automatic differentiation.

(c) Implementing the Adam Optimizer [20 points]

An incomplete implementation of the Adam optimizer is provided in `optimizer.py`. Your task is to complete the `step` method to perform parameter updates according to the Adam optimization algorithm.

Exercise 3: Exploring the Decision Boundary [10 points]

For this task, you need to figure out a configuration of the feed-forward network that results in a hexagon-shaped decision boundary when classifying the points. In particular, consider modifying the number of hidden units and activation functions.

1. **Describe the Network Configuration:** Specify the exact configuration you found that produces the hexagonal decision boundary.
2. **Explain the Hexagonal Boundary:** Provide a short theoretical explanation of why this specific configuration leads to a hexagonal pattern in the decision boundary.