SS 2025
© Prof. Georgia Koppe

Course Coordinator: Andre Herz
`andre.herz@iwr.uni-heidelberg.de`

Machine Learning Essentials

Exercise Sheet 04

Due: 02.06.2025 11:15

This sheet covers the **backpropagation** algorithm for a classification task by reinforcing the necessary calculations and guiding you through implementing it for a nonlinear dataset. The last exercise aims to give you some intuition about the training process of simple feedforward neural networks, as well as the influence of hyperparameters and regularization.

Regulations

Please submit your solutions via Moodle in teams of **3** students. The coding tasks must be completed using the template `MLE25_sheet04.ipynb`. Each submission must include **exactly** one file:

- Upload a `.pdf` file containing both your Jupyter notebook and solutions to analytical exercises.

The Jupyter notebook can be exported to pdf by selecting **File** → **Download as** → **pdf** in JupyterLab. If this method does not work, you may print the notebook as a pdf instead. Your analytical solutions can be either scanned handwritten solutions or created using LaTeX.

Exercise 1: Backpropagation for a Classification Task

Consider again the "transition" from single-layer Perceptrons to MLPs we've discussed on the last sheet. At the time of Minsky and Papert's book (1969), research on neural networks primarily focused on single-layer Perceptrons, which are limited to learning linearly separable functions. Although (linear) MLPs were known, the crucial importance of **nonlinear activation functions** in enabling networks to learn complex functions (like XOR) was not widely appreciated. This changed with the re-discovery of the **backpropagation** algorithm (1986[1]), which, alongside more capable hardware, provided an efficient and systematic way to compute the gradient of a loss function with respect to all network parameters $\boldsymbol{\theta}$, even in deep(er) networks. This then allowed MLPs to be trained effectively

[1]The seminal paper here was "Learning representations by back-propagating errors" by Rumelhart, Hinton and Williams.

using **gradient descent** optimization. Last sheet, we've manually selected parameters $\boldsymbol{\theta} = \{\boldsymbol{W}^{(1)}, \boldsymbol{b}^{(1)}, \boldsymbol{w}^{(2)}, b^{(2)}\}$ that solve XOR. In practice, we'd want the network to **learn** these parameters from data. Let's consider how backpropagation works based on a general feedforward network for multinomial classification.

Network Definition

An L-layer MLP[2] maps an input $\boldsymbol{x} \in \mathbb{R}^{d_0}$ to a vector of class probabilities $\hat{\boldsymbol{y}} \in [0,1]^C$:

$$f_{\boldsymbol{\theta}} : \mathbb{R}^{d_0} \to [0,1]^C, \quad \boldsymbol{x} \mapsto \hat{\boldsymbol{y}}.$$

The parameters $\boldsymbol{\theta}$ consist of weight matrices and bias vectors for each transformation layer:

$$\boldsymbol{\theta} = \{(\boldsymbol{W}^{(l)} \in \mathbb{R}^{d_{l+1} \times d_l}, \boldsymbol{b}^{(l)} \in \mathbb{R}^{d_{l+1}}) \mid l = 0, \dots, L-1\}.$$

Now the forward pass proceeds as follows. For the input layer:

$$\boldsymbol{z}^{(0)} = \boldsymbol{x}.$$

For the hidden layers ($k = 1, \dots, L-1$):

$$\begin{aligned} \tilde{\boldsymbol{z}}^{(k)} &= \boldsymbol{W}^{(k-1)} \boldsymbol{z}^{(k-1)} + \boldsymbol{b}^{(k-1)} && \text{(Preactivation)} \\ \boldsymbol{z}^{(k)} &= \boldsymbol{\phi}(\tilde{\boldsymbol{z}}^{(k)}) && \text{(Activation)} \end{aligned}$$

For the output layer ($k = L$):

$$\begin{aligned} \tilde{\boldsymbol{z}}^{(L)} &= \boldsymbol{W}^{(L-1)} \boldsymbol{z}^{(L-1)} + \boldsymbol{b}^{(L-1)} && \text{(Preactivation)} \\ \hat{\boldsymbol{y}} = \boldsymbol{z}^{(L)} &= \operatorname{softmax}(\tilde{\boldsymbol{z}}^{(L)}) = \boldsymbol{\sigma}(\tilde{\boldsymbol{z}}^{(L)}) && \text{(Output)} \end{aligned}$$

Here, $\boldsymbol{\phi}$ is an element-wise nonlinear activation function (e.g. sigmoid, ReLU, tanh) applied to the hidden layers. The softmax function normalizes the final preactivations into probabilities for each class:

$$\hat{y}_i = (\boldsymbol{z}^{(L)})_i = \frac{\exp\left(\tilde{z}_i^{(L)}\right)}{\sum_{m=1}^{C} \exp\left(\tilde{z}_m^{(L)}\right)} \quad \text{for } i = 1, \dots, C.$$

To train this network, we minimize the **cross-entropy loss** for a given training sample $(\boldsymbol{x}, \boldsymbol{y})$, where $\boldsymbol{y} \in \{0,1\}^C$ is the one-hot encoded true label:

$$\mathcal{L}_{\boldsymbol{\theta}}(\boldsymbol{x}, \boldsymbol{y}) = -\sum_{i=1}^{C} y_i \ln(\hat{y}_i).$$

[2]Here, L denotes the number of layers involving computation (transformations). An L-layer network has an input layer (layer 0), $L-1$ hidden layers (layers 1 to $L-1$), and an output layer (layer L). It has L sets of weights and biases, indexed $l = 0, \dots, L-1$.

We then apply gradient descent with learning rate γ to update the parameters:

$$\boldsymbol{W}^{(l)}_{\text{new}} = \boldsymbol{W}^{(l)}_{\text{old}} - \gamma \nabla_{\boldsymbol{W}^{(l)}} \mathcal{L}, \quad \boldsymbol{b}^{(l)}_{\text{new}} = \boldsymbol{b}^{(l)}_{\text{old}} - \gamma \nabla_{\boldsymbol{b}^{(l)}} \mathcal{L}.$$

In the lecture, it was shown that a key quantity for the backpropagation algorithm is the gradient of the loss w.r.t. the preactivation of each layer,

$$\boldsymbol{\delta}^{(l)} := \nabla_{\tilde{\boldsymbol{z}}^{(l)}} \mathcal{L} = \frac{\partial \mathcal{L}}{\partial \tilde{\boldsymbol{z}}^{(l)}}.$$

The crucial recursion of the backpropagation algorithm then is obtained by relating $\boldsymbol{\delta}^{(l)}$ to $\boldsymbol{\delta}^{(l+1)}$, using the chain rule. For multivariate functions, this means: if the **(scalar)** loss $\mathcal{L}$ depends on $\boldsymbol{y}$, and $\boldsymbol{y}$ depends on $\boldsymbol{x}$, then $\nabla_{\boldsymbol{x}} \mathcal{L} = (\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}})^\top \nabla_{\boldsymbol{y}} \mathcal{L}$, where $\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}$ is the **Jacobian** matrix of $\boldsymbol{y}$ w.r.t. $\boldsymbol{x}$ [3]. Applying this for the network's hidden layers gives the **general recursion formula for backpropagation**:

$$\boldsymbol{\delta}^{(l)} = ((\boldsymbol{W}^{(l)})^\top \boldsymbol{\delta}^{(l+1)}) \odot \boldsymbol{\phi}'(\tilde{\boldsymbol{z}}^{(l)}).$$

Here, $\odot$ denotes the element-wise product, also called Hadamard product, and $\boldsymbol{\phi}'(\tilde{\boldsymbol{z}}^{(l)})$ represents the vector that contains the diagonal elements of the activation's Jacobian matrix $\boldsymbol{J}_{\boldsymbol{\phi}} = \frac{\partial \boldsymbol{z}^{(l)}}{\partial \tilde{\boldsymbol{z}}^{(l)}} = \text{diag}(\boldsymbol{\phi}'(\tilde{\boldsymbol{z}}^{(l)}))$. This term appears via the chain rule:

$$\nabla_{\tilde{\boldsymbol{z}}^{(l)}} \mathcal{L} = \left(\frac{\partial \boldsymbol{z}^{(l)}}{\partial \tilde{\boldsymbol{z}}^{(l)}} \right)^\top \nabla_{\boldsymbol{z}^{(l)}} \mathcal{L} = \boldsymbol{J}_{\boldsymbol{\phi}}^\top \nabla_{\boldsymbol{z}^{(l)}} \mathcal{L}.$$

Since each activation corresponds only to the corresponding preactivation, this Jacobian matrix is diagonal and, as a diagonal matrix, $\boldsymbol{J}_{\boldsymbol{\phi}} = \boldsymbol{J}_{\boldsymbol{\phi}}^\top$. Multiplying by this diagonal matrix on the left is equivalent to the elementwise multiplication $\boldsymbol{\phi}'(\tilde{\boldsymbol{z}}^{(l)}) \odot \nabla_{\boldsymbol{z}^{(l)}} \mathcal{L}$, leading to the final recursion formula.

Tasks

1. Show that the gradient of the loss with respect to the output layer's activation has the simple form

$$\boldsymbol{\delta}^{(L)} := \frac{\partial \mathcal{L}}{\partial \tilde{\boldsymbol{z}}^{(L)}} = \nabla_{\tilde{\boldsymbol{z}}^{(L)}} \mathcal{L} = \hat{\boldsymbol{y}} - \boldsymbol{y}.$$

[3] The particular form of the chain rule/the backpropagation formulas depends on convention. We use the (arguably most common) **column vector convention** for gradients and the **numerator layout** for Jacobians ($J_{ij} = \frac{\partial x_i}{\partial x_j}$), which aligns with typical $\boldsymbol{y} = \boldsymbol{A}\boldsymbol{x}$ linear algebra. Using row vectors for gradients would alter the chain rule and shift transposes in backpropagation formulas. Both conventions are fine, the key is to stay consistent.

This is the "initialization" of the backward pass (the first step of the recursion).
Hint: When computing the inner derivative, you should obtain $\frac{\partial \hat{y}_i}{\partial \tilde{z}_j^{(L)}} = \hat{y}_i(\delta_{ij} - \hat{y}_j)$, where δ_{ij} is the Kronecker delta.

(4 pts.)

2. Using the result from Task 1., compute the gradient of the loss w.r.t. the parameters of the last hidden layer:

   (a) $\nabla_{\boldsymbol{W}^{(L-1)}} \mathcal{L}$,

   (b) $\nabla_{\boldsymbol{b}^{(L-1)}} \mathcal{L}$.

   What are the dimensions of these gradients? Make sure they match the dimensions of $\boldsymbol{b}^{(L-1)}$ and $\boldsymbol{W}^{(L-1)}$, respectively.
   Hint: You can check your results using

$$\nabla_{\boldsymbol{b}^{(l)}} \mathcal{L} = \boldsymbol{\delta}^{(l+1)}$$
$$\nabla_{\boldsymbol{W}^{(l)}} \mathcal{L} = \boldsymbol{\delta}^{(l+1)} (\boldsymbol{z}^{(l)})^\top.$$

(3 pts.)

3. Now we take one additional step backwards. To find $\boldsymbol{\delta}^{(L-1)}$, we set $l = L - 1$, using the general backpropagation recursion formula provided:

$$\boldsymbol{\delta}^{(L-1)} = ((\boldsymbol{W}^{(L-1)})^\top \boldsymbol{\delta}^{(L)}) \odot \boldsymbol{\phi}'(\tilde{\boldsymbol{z}}^{(L-1)}).$$

   From Task 1, we know that for our network here, $\boldsymbol{\delta}^{(L)} = \hat{\boldsymbol{y}} - \boldsymbol{y}$. Substituting this in, we get:

$$\boldsymbol{\delta}^{(L-1)} = ((\boldsymbol{W}^{(L-1)})^\top (\hat{\boldsymbol{y}} - \boldsymbol{y})) \odot \boldsymbol{\phi}'(\tilde{\boldsymbol{z}}^{(L-1)}).$$

   Let the activation function be the **Rectified Linear Unit (ReLU)**, defined as $\phi(x) = \max(0, x)$. Using its derivative $\phi'(x)$, explain what happens to the j-th component of the error signal, $\boldsymbol{\delta}_j^{(L-1)}$, if the preactivation $\tilde{z}_j^{(L-1)}$ for neuron j in layer $L - 1$ was negative during the forward pass. Pay attention to the point $x = 0$. Here you can adopt a common convention: setting $\phi'(0) = 0$. Briefly describe how choosing a ReLU activation influences the flow of gradients through the network during backpropagation based on this observation (you might also want to check your result here "in practice" using the web app from Task 3).

(3 pts.)

Exercise 2: Implementing Backpropagation

After exploring some of the inner workings of the backpropagation algorithm, this exercise aims to bridge the theory with practice. You'll implement the backpropagation algorithm for a simple MLP, train it on a classic nonlinear dataset, and visualize the results. The goal is to solidify your understanding about how the derived gradients are used to learn then network's parameters. We'll use a binary classification setting with sigmoid output and **binary cross-entropy loss (BCE)**. However, the principles here directly extend to the multinomial case discussed in Exercise 1. We'll use the "Two Moons" dataset, a standard **benchmark** for binary classification requiring nonlinear decision boundaries. The dataset can be loaded using `sklearn.datasets.make_moons`.

Network Architecture and Loss

- Input Layer (Layer 0): $d_0 = 2$ neurons (for the 2D data $\boldsymbol{x} = \boldsymbol{z}^{(0)} \in \mathbb{R}^2$).
- Hidden Layer (Layer 1): $d_1 = 10$ neurons. Preactivation $\tilde{\boldsymbol{z}}^{(1)} = \boldsymbol{W}^{(0)}\boldsymbol{z}^{(0)} + \boldsymbol{b}^{(0)}$. Activation $\boldsymbol{z}^{(1)} = \phi(\tilde{\boldsymbol{z}}^{(1)})$ using **tanh** activation $\phi(x) = \tanh(x)$.
- Output Layer (Layer 2): $d_2 = 1$ neuron. Preactivation $\tilde{z}^{(2)} = \boldsymbol{W}^{(1)}\boldsymbol{z}^{(1)} + b^{(1)}$ (scalar!). Activation $\hat{y} = z^{(2)} = \sigma(\tilde{z}^{(2)})$ using **sigmoid** activation $\sigma(x) = \frac{1}{1+e^{-x}}$.

Thus the network parameters are $\boldsymbol{\theta} = \{\boldsymbol{W}^{(0)} \in \mathbb{R}^{10\times 2}, \boldsymbol{b}^{(0)} \in \mathbb{R}^{10}, \boldsymbol{W}^{(1)} \in \mathbb{R}^{1\times 10}, b^{(1)} \in \mathbb{R}^1\}$. The (logistic regression-) output $\hat{y} \in (0,1)$ represents $p(y = 1|\boldsymbol{x})$. We use the BCE loss for a single sample $(\boldsymbol{x}, y)$, where $y \in \{0,1\}$ is the true label:

$$\mathcal{L}(\hat{y}, y) = -[y \ln(\hat{y}) + (1-y)\ln(1-\hat{y})]$$

Tasks

1. Before starting to code, briefly derive (as you'll need it for the implementation):
   (a) The derivative of the tanh activation: $\frac{d}{dx}\tanh(x)$. Express it in terms of $\tanh(x)$. (1 pts.)
   (b) The gradient of the BCE loss w.r.t. the output layer's preactivation $\tilde{z}^{(2)}$: $\delta^{(2)} = \frac{\partial \mathcal{L}}{\partial \tilde{z}^{(2)}}$. You should find the simple form $\hat{y} - y$, consistent with the result from Exercise 1.1 (which uses Softmax/Cross-entropy but yielded the same form for $\delta^{(L)}$).
   (1 pts.)
2. By completing the missing code sections in the provided notebook, implement
   (a) the derivative of the `tanh` function, `tanh_prime`.

(b) the `sigmoid` function.

(c) the `binary_cross_entropy` loss function.

(d) the `forward` method of the `MLP` class, which computes the forward pass of the MLP for a batch of data. Remember to store the intermediate values needed for backpropagation in `self.cache`.

(e) the `backward` method of the `MLP` class. This is the core of the exercise. Follow the backpropagation steps derived in Exercise 1:

- Compute $\delta^{(2)} = \hat{y} - y$.
- Compute $\nabla_{\boldsymbol{W}^{(1)}}\mathcal{L} = \delta^{(2)}(\boldsymbol{z}^{(1)})^\top$ and $\nabla_{\boldsymbol{b}^{(1)}}\mathcal{L} = \delta^{(2)}$.
- Compute $\delta^{(1)} = ((\boldsymbol{W}^{(1)})^\top\delta^{(2)}) \odot \tanh'(\tilde{\boldsymbol{z}}^{(1)})$.
- Compute $\nabla_{\boldsymbol{W}^{(0)}}\mathcal{L} = \boldsymbol{\delta}^{(1)}(\boldsymbol{z}^{(0)})^\top$ and $\nabla_{\boldsymbol{b}^{(0)}}\mathcal{L} = \boldsymbol{\delta}^{(1)}$.

Implement the backward pass using only **vectorized** NumPy operations for efficiency. This means avoiding explicit loops over "neurons" or features. Ensure your gradient calculations correctly average gradients over the training set (i.e. over the "batch dimension").

(8 pts.)

3. Briefly explain why using vectorized operations (like e.g. NumPy's matrix multiplication) is generally preferred in ML.

(1 pts.)

4. Load the data and run the provided training loop in the notebook using the functions you implemented.

(a) Play around with the hyperparameters `learning rate` and `number of epochs` and achieve a sensible setting for both. Use a plot of the evolution of the training loss over the number of trained epochs to guide your decision. Once you are satisfied, create a plot of the evolution of the training loss for your chosen settings.

(2 pts.)

(b) Compute the accuracy of the network on the test set and plot the final learned decision boundary overlaid on the test data. Comment on whether the network successfully classified the data. Does the decision boundary look reasonable?

(2 pts.)

Exercise 3: Neural Playground

In this exercise, we use `http://playground.tensorflow.org` to gain some intuition on what happens throughout the training of a feedforward neural network and how the different pieces interact and work together. It's an interactive web app that allows you to visualize and build intuition about the training process of simple neural networks. You can see how different hyperparameters, architectures and data characteristics influence learning. For each task below, submit screenshots of your trained networks alongside a short discussion of what you've observed.

Tasks

1. Select the **Spiral** dataset. Use the input features X_1 and X_2. Design and train a neural network architecture capable of effectively classifying this nonlinear pattern. "Effectively" here means achieving a low test error (e.g. below 0.1) and a decision boundary that visually matches the spiral structure. You are free to adjust the:

   - Number of hidden layers,
   - Number of neurons per layer,
   - Activation function (ReLU, tanh, sigmoid, linear),
   - Learning rate.

   Leave the other settings at the default. For the discussion: Describe the final architecture you chose. Why did you chose your particular architecture (e.g. what activation function worked best and why might that be the case?)?

   (2 pts.)

2. Now we want to investigate **overfitting** and **regularization**. Use the **Spiral** dataset again, this time selecting all available input features. Configure the largest possible network (corresponding to the largest possible model capacity/hypothesis space): 6 hidden layers, each with 8 neurons, and select the ReLU activation function. To further highlight the overfitting, significantly reduce the amount of training data (e.g. set the Train/Test ratio to 20%). Then, observe the impact of regularization on the network's weights and performance by training the network with:

   (a) No regularization.
   (b) L_1 regularization (choose a regularization rate, adjust as needed to see a clear effect).
   (c) L_2 regularization (choose a regularization rate, adjust as needed to see a clear effect).

   Let each network train for a sufficient number of epochs (until convergence). For the discussion: Compare the three scenarios. How did the test loss differ and why? How did the visual appearance of the learned weights (i.e. the lines connecting neurons) change with L_1 versus L_2 regularization compared to no regularization? Specifically, look for signs of L_1 **promoting sparsity** (some weights becoming zero or very small, indicated by thinner/absent lines) and L_2 **promoting smaller overall weight magnitudes**. Later on in the lecture, when we discuss regression, we will see that this can also be seen formally by analyzing the effects of the additional terms in the loss: L_1 adds a penalty based on the sum of absolute weights ($\lambda \sum ||\boldsymbol{w}||_1$) and L_2 adds a penalty based on the sum of squared weights ($\lambda \sum ||\boldsymbol{w}||_2^2$).

(3 pts.)