Heidelberg University
Institute of Computer Engineering
HAWAII Lab

Hendrik Borras
Xiao Wang

Neural Networks from Scratch
Summer Term 2026

Exercise 1

- **Return electronically until Wednesday, April 22, 2026, 09:00**
- **Include your names on the top sheet. Hand in only one PDF.**
- **A maximum of three students are allowed to work jointly in a group.**
- **When you include plots add a short explanation of what you expected and observed.**
- **Hand in source code if the exercise required programming. You can bundle the source code along with the PDF in a .zip file.**

1.1 Polynomial curve fitting

This exercise serves as a gentle introduction to Python and the phenomenon of overfitting.

- **Setup.** Install `numpy` and `matplotlib` on your machine. If you do not have Python (and/or work on Windows), we recommend using `uv`. See here: uv docs, make sure to familiarize yourself with the workings of `uv`. With the exercises we also provide a reference `pyproject.toml`, to get you started. Alternatively you can also use pip directly like this: `uv pip install numpy matplotlib`
- **Check the template.** Start with the template `ex11_template.py` in the "template/" folder. The file should already produce two plots, with as of yet incorrect data.
- **Data generation.** Extend the `ground_truth_function` to generate data of the shape $h(x) = \sin(2\pi x)$ as in the lecture.
- **Error function.** Extend the `error_function` to reflect the non-regularized error function from the lecture.
- **Perform the polynomial fit.** For this exercise, we will use function `polynomial.Polynomial.fit` from `numpy` to fit the polynomial for us. If everything is set up correctly, you should by now obtain a plot for a polynomial fit of third degree, similar to the one shown in the lecture.
- **Investigate overfitting.** Create a new plot for an overfitted polynomial of 11th degree. Now vary the degree of the polynomial from 0 to 11. With this data, reproduce the plot of "Polynomial degree against the train and test error" from the lecture. Make use of the already created data and ensure to use the RMS error instead of the direct error.
- **Investigate dataset size.** Next, vary the size of the data while keeping the degree of the polynomial constant. As a sensible starting point, choose a 10th-degree polynomial. Then vary the sample size of the dataset from 10 until the RMS error of the train and test datasets are reasonably close, i.e., the function no longer overfits. Note that the required sample size can easily be on the order of 100s to 1000s of datapoints for this to occur. Make a plot of how train and test error change with the number of samples in the dataset.

1.2 Polynomial curve fitting with Stochastic Gradient Descent (SGD)

In this second part, you will implement and train the same polynomial model using your own implementation of stochastic gradient descent (SGD). We will use a fixed dataset size of $N = 200$ samples.

- **Choose the model degree.** Open the script `ex12_template.py`. In the main section, add the degree you found in Part 1 to balance accuracy and overfitting.
- **Implement the model function.** Implement `poly_predict(x, w)` that computes

$$\hat{y}(x; \mathbf{w}) = \sum_{i=0}^{M} w_i x^i.$$

Heidelberg University
Institute of Computer Engineering
HAWAII Lab

Hendrik Borras
Xiao Wang

- **Implement the loss.** Implement `mse_loss(y_pred, y_true)` that returns the mean squared error (MSE):

$$E(\mathbf{w}) = \frac{1}{2N} \sum_{n=1}^{N} (h(x_n, \mathbf{w}) - t_n)^2$$

- **Implement the SGD update rule.** Implement `sgd_update(x_b, y_b, w, η)` that:
  1. computes the gradient $\frac{\partial L}{\partial w_j}$ of the MSE loss with respect to each weight (polynomial coefficient) w_j,
  2. returns the updated weights $w \leftarrow w - \eta \nabla_w L$.
- **Train with gradient descent.** Implement `train_fullbatch(x, y, degree, η, n_epochs)` that initializes w, then for each epoch calls `sgd_update` on the full dataset and records the training loss.
- **Extend to mini-batch training.** Implement `train_minibatch(x, y, degree, η, b, n_epochs)` that shuffles the data each epoch, splits it into batches of size b, applies `sgd_update` per batch, and records the training loss once per epoch.
- **Learning-rate sweep (fixed batch).** For a fixed batch size (e.g. $b = 16$), train for **2000 epochs** with several learning rates (try out values for $\eta \in [10^{-4}, 1]$). Create a plot of training loss vs. epoch for each learning rate. Briefly discuss the observed influence of the learning rate on the training.
- **Batch-size sweep (fixed learning rate).** For a fixed learning rate (e.g. $\eta = 0.03$), train for **2000 epochs** with several batch sizes (try out values for $b \in [1, N]$). Create a plot of training loss vs. epoch for each batch size. Briefly discuss the observed influence of the batch size on the training.
- **It learns!** Choose a well-performing (η, b) and create an animation showing how the predicted polynomial changes over the course of training (e.g., every few epochs). To achieve this, implement a variant of the training loop that takes a snapshot of your polynomial every few epochs by collecting and returning the weights. You may adjust the number of epochs to obtain a satisfactory animation. An example `create_gif` function is provided in the template, feel free to adapt it to your liking.