Deep Learning 2026: Hand-in Assignment 1

January 20, 2026

Individual assignment **Due: February 8, 2026**

In this individual assignment, you will implement a neural network to classify images. You will consider the so-called MNIST dataset, which is one of the most well-studied datasets within machine learning and image processing. The dataset consists of 60 000 training data points and 10 000 test data points. Each data point consists of a 28×28 pixels grayscale image of a handwritten digit. The digit has been size-normalized and centered within a fixed-sized image. Each image is also labeled with the digit (0,1,...,8, or 9) it is depicting. In Figure 1, a set of 100 random data points from this dataset is displayed.

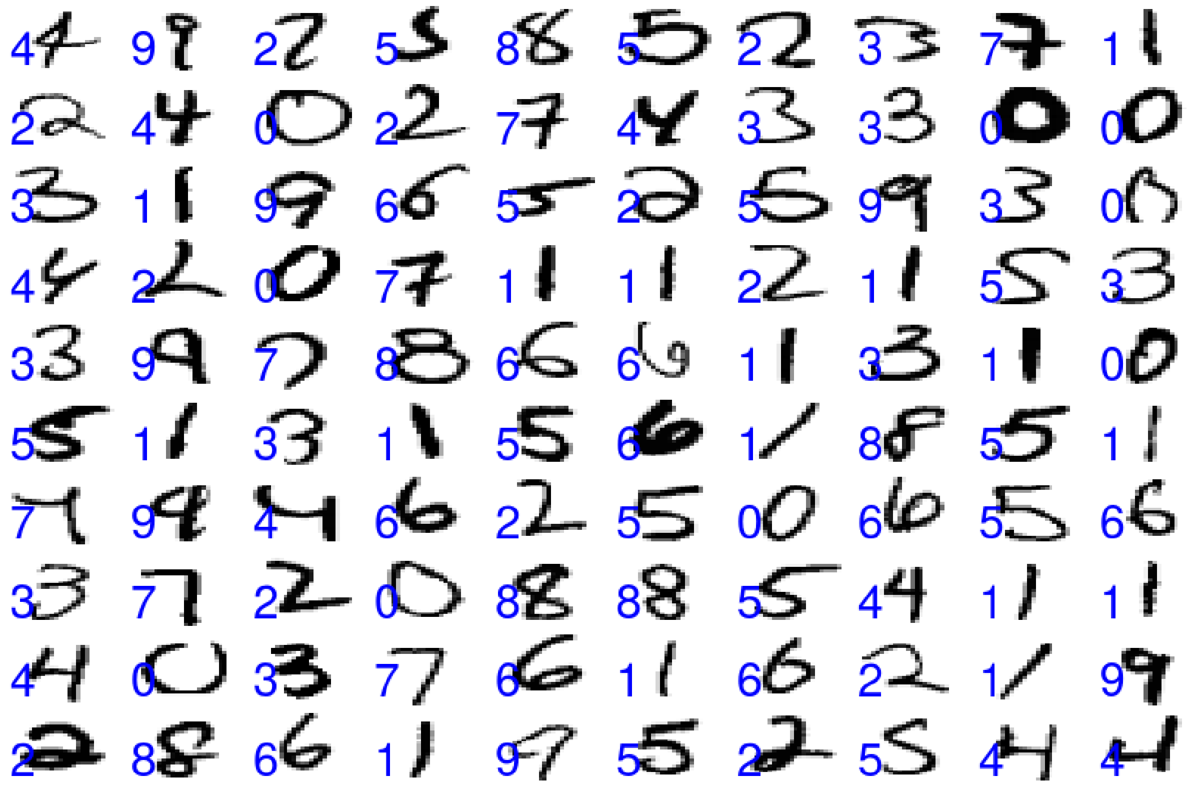

Figure 1: Some samples from the MNIST dataset used in the assignment. The input is the pixel values of an image (grayscale), and the output is the label of the digit depicted in the image (blue).

In this classification task, we consider the image as our input $\mathbf{x} = [x_1, \dots x_{D_i}]^\mathsf{T}$. Each input variable x_j corresponds to a pixel in the image. In total we have $D_i = 28 \cdot 28 = 784$ input variables. The value of each x_j represents the color of that pixel. The color-value is within the interval [0,1], where $x_j = 0$ corresponds to a black pixel and $x_j = 1$ to a white pixel. Anything between 0 and 1 is a gray pixel with corresponding intensity.

The dataset is available in the file `MNIST.zip` from the course homepage. The dataset is divided into subfolders `train` and `test`, and further into subfolders `0-9` containing several images in the form `0000.png-xxxx.png`. (Note: Each class does not have exactly the same number of images.) All images are 28×28 pixels and stored in the png-file format. We provide a script `load_mnist` in `utils.py` for importing the data.

Tips

- **Debugging**: If something does not work, try to isolate the problem. Comment out code that possibly has bugs, do plenty of printouts and plots. Simplify the task until it works and then work from there. Work in small iterations.
- **One-sample training**: Can the network converge if you only have one sample in the train set and overfit to that?
- **Training time**: These are very small images; the network should converge in minutes when training. If it takes longer, you probably have a bug or your code is not vectorized (you're not using numpy matrix multiplication or broadcasting).

Classification of handwritten digits

In this hand-in assignment, you will implement and train a fully connected neural network for solving a classification problem with multiple classes. To solve this assignment, you can use, combine, and extend the notebooks you produced in Lab 1.

It is strongly encouraged that you write your code in a *vectorized* manner (as you did in Lab 1), meaning that you should not use `for` or `while` loops over data points or hidden units, but rather use the equivalent matrix/vector operations. In modern languages that support array processing[1] (in this case `numpy` in Python), the code will run much faster if you vectorize. This is also how modern software for deep learning works, which we will be using later in the course. Finally, vectorized code will require fewer lines of code and will be easier to read and debug. To vectorize your code, choose which dimension represents your data points, and be consistent with your choice. In machine learning and Python, it is common to reserve the first dimension for indexing the data points, i.e., one row equals one sample. For example, for a linear model

$$\mathbf{f}_i = \boldsymbol{\Omega}\mathbf{x}_i + \boldsymbol{\beta}, \qquad i = 1, \dots, I_b,$$

the vectorized version (transposing the expression above to get each $\mathbf{f}_i$ as a row output) for a mini-batch with I_b data points would be

$$\begin{bmatrix} \mathbf{f}_1^\top \\ \vdots \\ \mathbf{f}_{I_b}^\top \end{bmatrix} = \begin{bmatrix} \mathbf{x}_1^\top \\ \vdots \\ \mathbf{x}_{I_b}^\top \end{bmatrix} \boldsymbol{\Omega}^\top + \boldsymbol{\beta}^\top \tag{1}$$

where $\boldsymbol{\beta}^\top$ is added to each row (called broadcasting in Python).[2] Note that in the main optimization loop, we still need a `for`-loop over the number of epochs/iterations.

The recommended loss function ℓ_i for a multi-output classification problem is the cross-entropy loss, which, for numerical issues when $f \ll 0$, is computed *together* with the softmax function. The cost L, for a D_o-class problem, is computed by summing the loss ℓ_i over all the training points $\mathbf{x}_i$

$$\ell_i = \log\left(\sum_{k=1}^{D_o} \exp[f_{ik}]\right) - \sum_{k=1}^{D_o} \widetilde{y}_{ik} f_{ik} \qquad L = \frac{1}{I}\sum_{i=1}^{I} \ell_i \tag{2}$$

where $\widetilde{y}_i$ is the one-hot encoding of the true label y_i for data point i

$$\widetilde{y}_{ik} = \begin{cases} 1, & \text{if } y_i = k \\ 0, & \text{if } y_i \neq k \end{cases} \qquad \text{for } k = 1, \dots, D_o$$

[1] `https://en.wikipedia.org/wiki/Array_programming`

[2] You might want to consider implementing the transposed version of $\boldsymbol{\Omega}$ and $\boldsymbol{\beta}$ to avoid the transposes in this vectorized model.

Exercise 1. Implement a multi-layer neural network:

In Python, using `numpy`, implement a fully connected neural network for solving the classification problem. It should involve the following functionalities, some of which have already been implemented in Lab 1:

1. **Initialize.** A function `initialize` that initializes $\boldsymbol{\Omega}_k, \boldsymbol{\beta}_k$ for each layer in the model. Note that when adding more layers, it is important that the elements in the weight matrices are initialized randomly (why?). Initialize each element in $\boldsymbol{\Omega}_k$ using He initialization[3]. The offset vectors $\boldsymbol{\beta}_k$ can be initialized with zeros.

2. **Activation functions.** *Two* functions `sigmoid` and `ReLU` that implement each of the corresponding activation functions, as well as their derivatives. Note, only the second one was implemented in Lab 1.

$$\texttt{sigmoid} : a[x] = \frac{1}{1+\exp[-x]}, \qquad \texttt{ReLU} : a[x] = max(0, x). \tag{3}$$

   Also, implement the derivative with respect to the input for each of the two activation functions.

3. **Forward propagation.** A function `forward_pass` that does the forward propagation for all the layers in the model. It is recommended that the function also returns all pre-activations and activations for all layers. This will make the backward propagation more efficient.

4. **Softmax and cost.** Write a function `softmax` that computes the softmax activation function as

$$p_k = \frac{\exp[f_k]}{\sum_{k'=1}^{D_o} \exp[f_{k'}]}, \quad k = 1, \ldots, D_o. \tag{4}$$

   Also, write a function `compute_cost` that computes the cost from the last layer linear output `f`. Preferably, `compute_cost` should compute softmax and the cross-entropy loss combined in its simplified form, as shown in (2). Finally, write a function `d_cost_d_output` that computes the derivative of the cost with respect to the last layer linear output. See also Task 3.6 in Lab 1 for additional guidance on these functions.

   Optional: For numerical stability when $f \gg 0$, reduce the magnitude of f by subtracting its maximum value before computing the loss (verify that adding a constant to f does not change the loss).

5. **Backward propagation.** A function `backward_pass` that computes backward propagation for all the layers in the model.

6. **Take a step.** Write a function `update_parameters` that updates the parameters for every layer based on provided gradients. Scale the update step by the *learning rate* α, which will determine the size of the steps you take in each iteration.

7. **Predict.** A function `predict` which, using the trained model, and a data set (can be either train or test data), predicts softmax output based on corresponding inputs $\mathbf{x}_i$. Also, compute the accuracy by counting the fraction of times your prediction, obtained as the maximum index in the softmax output $\mathbf{p}_i$, matches the correct class label. As the last function output, return the cost for the data set.

8. **Mini-batch generation.** A function `random_mini_batches` that randomly partitions the training data into several mini-batches `(x_mini, y_mini)`.

9. **Model training.** A final function `train_model` that iteratively calls the above functions that you have defined. We expect that it should have as inputs, at least:
   `x_train`: train data
   `y_train`: train labels
   `model`: defining the model architecture in some way, e.g., a vector with the number of nodes in each layer
   `num_epochs`: number of epochs to be used during training
   `learning_rate`: learning rate α that determines the step size
   `batch_size`: number of training examples to use for each step

   To monitor the training, you may also wish to provide test data (not used for training!):
   `x_test`: test data
   `y_test`: test labels
   and call the `performance` function every k:th iteration. We recommend that you save and return the training and test costs and accuracies at every k:th iteration in a vector, and possibly also print or plot the results live during training.

[3]See section 7.5 in the course book

Note 1: The above function names are suggestions; you are allowed to structure your code in another way if you feel that it suits you more.

Note 2: If you use columns for data point dimension, remember to transpose the data matrices after loading the data via `load_mnist` in `utils.py`.

Exercise 2. Evaluate a linear model

Evaluate your code on the MNIST dataset by first considering a **linear model**, i.e., a neural network with zero layers of hidden units[4]. Use the provided code to extract the train and test data and labels in the desired format. You should be able to reach over 90% accuracy on the test data.

(a) Using `matplotlib`, produce a plot of the cost, both on training and test data, with epochs on the x-axis. Also, include a plot with the classification accuracy, also evaluated on both test and training data, with epochs on the x-axis. For the training data, evaluate on the current mini-batch instead of the full training dataset. As a help, you are strongly suggested to use the function `training_curve_plot` in `utils.py`. It plots a training curve using arrays of training and test accuracies/costs that you stored during your training.

(b) Extract each of the ten rows (or columns, depending on your implementation) of your weight matrix, reshape each of them into size 28×28, and visualize them as 10 images. What is your interpretation of these images? Include a few of these images in the report.

Exercise 3. Evaluate your multi-layer neural network

Evaluate your **full neural network** with several layers. Try both of the two activation functions. You should be able to get up to almost 98% accuracy on the test data after playing around a bit with the design choices. Train the model until convergence, i.e., until the point that you don't see any clear improvement. Provide two plots, one plot for each of the two activation functions, using similar plotting code as you used in Exercise 2a).

Exercise 4. Implement batch gradient descent with momentum

Voluntary extra task (for your own learning pleasure): Implement extensions to mini-batch gradient descent, like use of *momentum* (easy) and *ADAM* (a little bit more work).

Submission instructions: Submit a PDF with answers to Exercise 1, Exercise 2, Exercise 3, and possibly Exercise 4. Also include your well-commented code as an appendix in your PDF **and** as a separate zip-file. Submit both files in one submission. Use the Assignment template for compiling the PDF.

[4]If your code is based on the notebooks in Lab 1, use $K = 0$ to get a linear model.

Checklist

Please check these things before handing in:

- ☐ **No miss**: Make sure you didn't miss answering any task.
- ☐ **Repeat the problem**: Start the answer to each task with the problem formulation (a short sentence where you write with your own words what you did). For example, "Exercise 1: In Exercise 1, we implement ... using ... since ..." instead of just "Exercise 1: Answer:..". Also, provide details about learning rate, batch size, and number of nodes in each layer when applicable.
- ☐ **Concise answers**: Still, be short and concise in your answers. You don't need to write long explanations for each exercise.
- ☐ **Plots available**: All requested plots are available and have proper figure captions, legends, and axis labels. The plots are visible (not too small text), and you comment on what can be seen in the plots.
- ☐ **Reasonable accuracy**: The accuracy on the test-set of the one-layer network should be at least 90% and the multi-layer at least 95% if you implemented everything correctly. If you choose the learning rate, number of hidden nodes and layers in a good way, you can get above 98% in Exercise 3. Try to experiment a bit.
- ☐ **Generative AI**: You have commented if you have used any generative AI, and if so, how.[a]
- ☐ **Code**: Code is both attached as an appendix to the PDF and submitted as a separate zip-file.

[a]See on Studium regarding use of generative AI.