Deep Learning for Image Analysis 2021: Assignment 2

Partly adapted from Niklas Wahlström

March 27, 2023

Due: April 14, 2023

In this assignment we will implement a neural network to classify images. We will consider the so-called MNIST dataset, which is one of the most well studied datasets within machine learning and image processing; do checkout the `http://yann.lecun.com/exdb/mnist/` page for some context. 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.

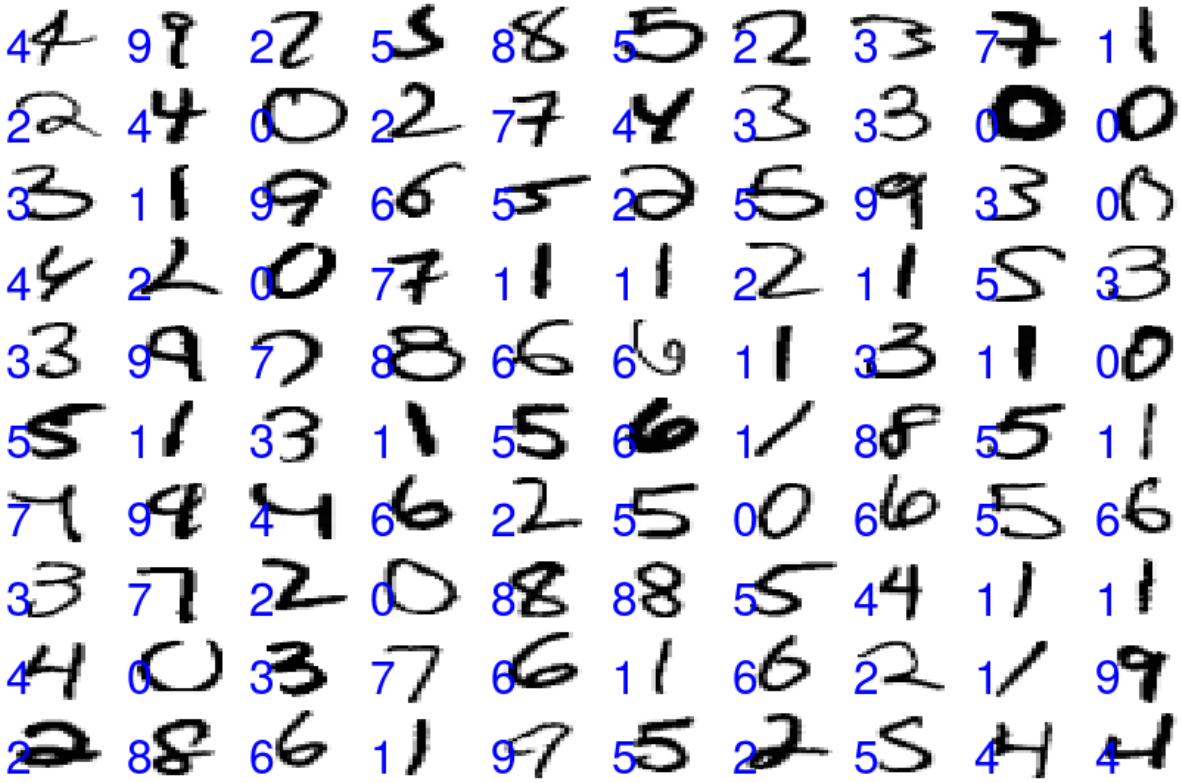

Figure 1: Some samples form the MNIST dataset used in the assignment. The input is the pixels values of an image (grayscale), and the output is the label of the digit depicted in the image (blue).

In this classification task we consider the image as our input $\mathbf{x} = [x_1, \dots x_p]^\mathsf{T}$. Each input variable x_j corresponds to a pixel in the image. In total we have $p = 28 \cdot 28 = 784$ 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 a number of images on 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. If you are running python we provide a script `load_mnist.py` for importing the data, which is also available on the course homepage.

1 Classification of handwritten digits

In this hand-in assignment you will implement and train a feed forward neural network for solving a classification problem with multiple classes. To solve this assignment you need to extend the code from Assignment 1 in three aspects:

- **Softmax output** Add a softmax function to the output to handle this as a classification problem. Use the cross-entropy as loss function instead of the squared error loss.
- **Mini-batch training** Replace the gradient descent with mini-batch gradient descent to be able to handle larger datasets.
- **Multiple layers** Extend the model to include multiple layers with intermediate activation functions.

Do the implementation in steps and test your code along the way, to simplify bug hunting. Observe, e.g., that for solving Exercise 1.3, not all parts of 1.2 have to be completed (i.e., read the whole assignment description before starting coding).

It is strongly encouraged that you write your code in a *vectorized* manner, meaning that you should not use `for` or `while` loops over data points or hidden units, but rater 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 that 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

$$\mathbf{z}_i = \mathbf{W}\mathbf{x}_i + \mathbf{b}, \qquad i = 1, \ldots, n,$$

the vectorized version (transposing the expression above to get each $\mathbf{z}_i$ as a row output) for a mini-batch (n_b) would be

$$\begin{bmatrix} \mathbf{z}_1^\mathsf{T} \\ \vdots \\ \mathbf{z}_{n_b}^\mathsf{T} \end{bmatrix} = \begin{bmatrix} \mathbf{x}_1^\mathsf{T} \\ \vdots \\ \mathbf{x}_{n_b}^\mathsf{T} \end{bmatrix} \mathbf{W}^\mathsf{T} + \mathbf{b}^\mathsf{T} \tag{1}$$

where $\mathbf{b}^\mathsf{T}$ 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 chosen loss function L_i for a multi-output classification problem is the cross-entropy loss which, for numerical issues when $z \ll 0$, is computed together with the softmax function. The cost J, for an M-class problem, is computed by summing the loss L_i over all the training points $\mathbf{x}_i$

$$L_i = ln(\sum_{l=1}^{M} e^{z_{il}}) - \sum_{m=1}^{M} \widetilde{y}_{im} z_{im} \qquad J = \frac{1}{n} \sum_{i=1}^{n} L_i \tag{2}$$

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

$$\widetilde{y}_{im} = \begin{cases} 1, & \text{if } y_i = m \\ 0, & \text{if } y_i \neq m \end{cases} \qquad \text{for } m = 1, \ldots, M$$

Exercise 1.1 *As you did in the previous assignment, derive the expressions for $\frac{dJ}{db_m}$ and $\frac{dJ}{dw_{mj}}$ expressed in terms of $\frac{dJ}{dz_{im}}$, $\frac{dz_{im}}{db_m}$ and $\frac{dz_{im}}{dw_{mj}}$, in this case for the multinomial cross-entropy loss presented in Equation 2.*

Exercise 1.2 *Implement in Python, using* `numpy`*, a feed forward neural network for solving the classification problem. It should involve the following functionalities:*

1. **Initialize.** *Write a function* `initialize_parameters` *that initializes $\mathbf{W}, b$ 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 (can you figure out why?). Initializing each element $\mathbf{W}$ by sampling from $\mathcal{N}\left(0, \sigma^2\right)$ where $\sigma = 0.01$ will do the job. The offset vectors b can still be initialized with zeros.*

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

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

2. **Activation functions.** *Write two functions* `sigmoid` *and* `relu` *that implement the activation functions in Equations 3a and 3b, respectivelly*

$$h(x) = \frac{e^x}{1+e^x} = \frac{1}{1+e^{-x}} \tag{3a}$$

$$h(x) = max(0, x) \tag{3b}$$

3. **Forward propagation.** *Write a function* `linear_forward` *that computes the linear part of a layer's forward propagation. Write a function* `activation_forward` *that applies the desired activation function to the linear output for one layer. And finally, write a function* `model_forward` *that produces the whole forward propagation for all the layers in the model. It is recommended that you also compute a* `model_state` *variable where you store linear outputs, activation outputs and parameters for each layer. This will make the backward propagation more efficient.*

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

$$p_m = \frac{e^{z_m}}{\sum_{l=1}^{M} e^{z_l}}, \quad m = 1, ..., M \tag{4}$$

Write a function `compute_loss` *that computes the loss from the last layer linear output z, as shown in Equation 2. For numerical stability when $z \gg 0$, reduce the magnitude of z by subtracting its maximum value before computing the loss (verify that adding a constant to z does not change the loss). Verify that this (stabilized) function gives the same answer (when z is not very large/small) as if computing the NLL loss based on* `softmax(z)`.

5. **Backward propagation.** *Write a function* `linear_backward` *that computes the linear portion of backward propagation for a single layer. Write two functions* `sigmoid_backward` *and* `relu_backward` *that compute the derivatives of the activation functions presented in Equations 3a and 3b. Write a function* `activation_backward` *that computes backward propagation with the desired activation function for one layer. And finally, Write a function* `model_backward` *that computes backward propagation for all the layers in the model. Here the* `model_state` *variable you stored comes in handy.*

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.** *Write a function* `predict` *which, using the trained model, predicts the softmax outputs p_i based on the corresponding inputs $\mathbf{x}_i$. This function will in this task more or less only be a wrapper for* `model_forward`*. Additionally compute the accuracy by counting the percentage of times your prediction —obtained as the maximum index in the softmax output p_i— matches the correct class label.*

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

Write 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 (your choice, e.g. a vector with number of nodes in each layer)*
`iterations`*: number of iterations*
`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 `predict` *function every k-th iteration. We recommended that you save and return the train and test costs and accuracies at every k-th iteration in a vector, and possibly also print or plot the results live during training.*

Exercise 1.3 *Evaluate the code on the MNIST dataset using a **one layer network**. 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 test data. Note that if you implemented the mini-batch generation the training will be much faster.*

1. *Using* `matplotlib`*, produce a plot of the cost, both on training and test data, with iterations on the x-axis. Also, include a plot with the classification accuracy, also evaluated on both test and training data with iterations on the x-axis. For the training data, evaluate on the current mini-batch instead of the whole training dataset.*

2. *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 1.4 *Repeat Exercise 1.3, but this time implementing a **full neural network** with several layers. You should be able to get up to almost 98% accuracy on test data after playing around a bit with the design choices.*

Exercise 1.5 *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).*

Use the 'Assignment template' (on the Course homepage) and submit a pdf with answers to Exercise 1.1, Exercise 1.3 and Exercise 1.4. For Exercise 1.2, submit a zip file with your well commented code.