Deep Learning PhD course: Hand-in assignment 1B

Niklas Wahlström

March 25, 2019

Due: 26th of April 2019, 23:59

1 Classification of hand-written digits

In this assignment we will implement a neural network to classify images. We will consider the so-called MNIST dataset[1], 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 batch of 100 data points from this dataset is displayed.

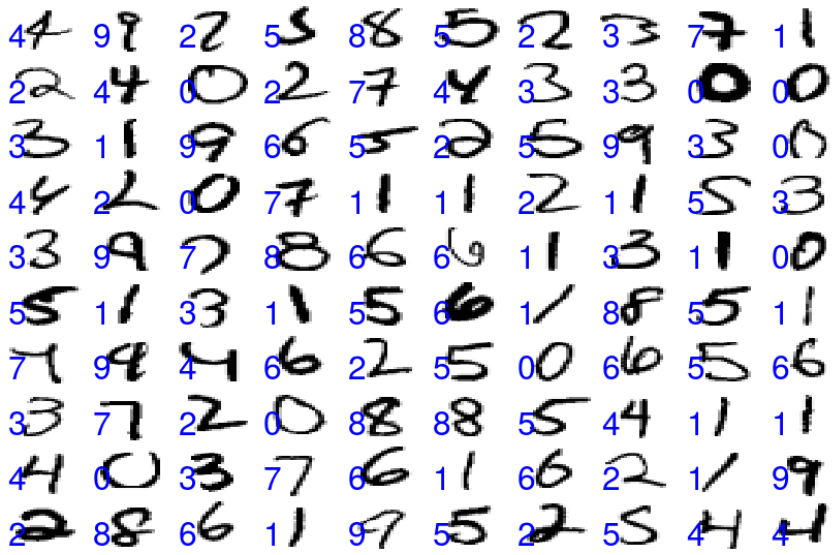

Figure 1: Some samples form the MNIST dataset used in the assignment. The input is the pixels values of an image (grayscale), and the output is the label of the digit depicted in the image (blue).

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] `http://yann.lecun.com/exdb/mnist/`

2 The tasks

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 hand-in assignment 1A in three aspects:

1. **Softmax output** Replace the sigmoid output with a softmax output to handle $K > 2$ classes. Use the cross-entropy as loss function.
2. **Mini-batch training** Replace the gradient descent with mini-batch gradient descent to be able to handle larger datasets.
3. **Multiple layers** Extend the model to include multiple layers with intermediate activation functions.

Exercise 2.1 *Implement the first two extensions and train your model on the MNIST training dataset. 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 2.2 *Implement the third extension, train your model on the MNIST training dataset, and evaluate it on the MNIST test dataset. Implement the option to use either the sigmoid or the ReLU activation functions. Adjust your design choices (mainly the number of layers, number of hidden units, the learning rate and choice of activation function) to increase your classification accuracy on test data. In the report, include 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. For the training data, evaluate on the current mini-batch instead of the whole training dataset.*

Some advice/strong encouragements/comments regarding the implementation:

- It is recommended to implement the three extensions in the order presented above since that allows you to verify your implementation after each these three subtasks.
  - After implementing the softmax output you are able to train on the MNIST training dataset. The training will be slow, but the accuracy on the test data should start approaching 90%.
  - After implementing the mini-batch gradient descent, the training will be significantly faster and you should quickly reach 92% accuracy.
  - After implementing the full neural network you should be able to get up to almost 98% accuracy on test data after playing around a bit with the design choices.
- 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 by sampling from $\mathcal{N}\left(0, \sigma^2\right)$ where $\sigma = 0.01$ will do the job. The offset vectors can still be initialized with zeros.
- It is strongly encouraged for you to 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[2] (for example Matlab, R and 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 the data points. For example, for a linear model

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

the vectorized version would be

$$\begin{bmatrix} \mathbf{z}_1^\mathsf{T} \\ \vdots \\ \mathbf{z}_n^\mathsf{T} \end{bmatrix} = \begin{bmatrix} \mathbf{x}_1^\mathsf{T} \\ \vdots \\ \mathbf{x}_n^\mathsf{T} \end{bmatrix} \mathbf{W}^\mathsf{T} + \mathbf{b}^\mathsf{T} \tag{2}$$

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

, where $\mathbf{b}^\top$ is added to each row (called broadcasting in python-language). For this implementation you might want to consider implementing the transposed version of $\mathbf{W}$ and $\mathbf{b}$ to avoid the transposes, see Section 7.1.5 in the lecture notes for this convention. Finally, note that in the main optimization loop we do still need a `for`-loop over the number of epochs/iterations.

- To structure your code, assign each submodule of the code to its own function/method/class. As one option, you might want to divide your code into the submodules listed below. Do not necessarily take this list literately, there are of course many options for how to structure your code.

`initialize_parameters`	Initializes all parameters in the model
`linear_forward`	Linear part of a layer's forward propagation
`sigmoid`	Sigmoid activation
`relu`	ReLU activation
`linear_activation_forward`	Forward propagation for one layer
`L_model_forward`	Forward propagation for all L layers
`softmax`	Softmax activation function
`compute_cost`	Cost function computed from the logits for numerical stability
`linear_backward`	Linear portion of backward propagation for a single layer
`sigmoid_backward`	Backward propagation for a single sigmoid unit
`relu_backward`	Backward propagation for a single ReLU unit
`linear_activation_backward`	Backward propagation for one layer
`L_model_backward`	Backward propagation for all L layers
`update_parameters`	Update parameters using gradient descent
`random_mini_batches`	Create a list of random minibatches
`predict`	Predict the results of a L-layer neural network
`train_L_layer_model`	Main loop for training a L-layer neural network
`load_mnist`	Load the MNIST data

- If you want (but not required), try to implement the extensions to mini-batch gradient descent that Thomas talked about in the lecture, like RMSprop and Adam.