Assignment 4

marko.ristin@nu.edu.kz, siamac.fazli@nu.edu.kz

Due date: **13 October 2024 23:55**

In this assignment, we will build a fully-connected neural network from scratch using the NumPy library.

We will work with the MNIST handwritten digit dataset.

Task 1: Set up the environment

Make sure that you have a Python environment set up and that you installed numpy and can access it. There are multiple ways to set up a Python environment. For example, you can create a virtual environment with a command line:

```
python3 -m venv venv
```

Then you need to activate it, again in the command line (in Linux):

```
source venv/bin/activate
```

Or in Windows:

```
venv/Scripts/activate.bat
```

Finally, you can install numpy with pip:

```
pip3 install "numpy==2.1.0"
```

Please search on the Internet for other ways to set up a Python environment if you prefer Conda, Anaconda *etc.*

Let's also install matplotlib for visualization:

```
pip3 install "matplotlib==3.9.2"
```

Task 2: Obtain and read the dataset

Download the dataset from:

~~https://yann.lecun.com/exdb/mnist/~~ (Marko: the link is temporarily unavailable)

the Moodle.

Reading the training data is a hard part of almost every machine learning project. There's no difference with the MNIST digit dataset. It is an old dataset from the 80's stored in an arcane format.

The instructions about the format for the labels and images can be read at the bottom of https://yann.lecun.com/exdb/mnist/.

Use the `struct` package from the standard library to read binary data (https://docs.python.org/3/library/struct.html).

Read the images into numpy matrices. For example, one image per row.

You probably want to make separate functions for loading the data. For example, `load_labels` and `load_images` which accept the path to a file and give back the loaded data.

Usually, you need to preprocess the data. To that end, pick the data processing of your choice and estimate any statistics on the training set, then apply it both on the training and on the test set.

You probably want to store the pre-processed data to save some computation (*i.e.,* processing time).

Task 3: Re-shape and visualize the images

Find out using numpy documentation how to reshape the images from a large matrix to a list of individual 2D matrices for visualization.

Make this a separate function, say, `reshape_matrix_to_images`, so that you can use it both for the training and testing images.

Visualize them with `matplotlib`, *e.g.,* using `matplotlib.pyplog.imshow` (https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.imshow.html).

Check the images against the labels to make sure that everything fits together.

Task 4: Structure the neural network

Write a class or multiple classes to represent the neural network.

Coming up with suitable software architecture is another integral part of machine learning projects, so we deliberately leave it up to you to pick the structure that you find best. Make sure you do not overthink it.

Some people prefer to keep it simple and create a single class (*e.g.,* `NeuralNetwork`) and keep all its state with different attributes typed as numpy matrices and scalar numbers. Other people prefer to structure the network using multiple classes (*e.g.,* one class for every kind of layer), many methods *etc.* It is up to you to pick your style.

You probably want to structure the weights as matrices (*e.g.,* rows as number of layer's inputs and columns as number of layer's outputs). Also, don't forget to include the bias with the weights!

We will play later with different numbers of layers each consisting with a different number of nodes. Make sure your architecture supports such parameters as well as different kinds of nodes (input nodes, different activation functions, output nodes).

For now, let's try the following structure:

- 784 input nodes (for all 28x28 pixels),
- 512 hidden nodes, and
- 10 output nodes (one for every digit).

Let's use:

- ReLU as activation function for the hidden nodes (see https://en.wikipedia.org/wiki/Rectifier_(neural_networks)), and
- Softmax for the output nodes (see https://en.wikipedia.org/wiki/Softmax_function).

If you feel lost: try to implement a toy network first with a single input layer and a single output node. How can you add another layer between the input and the output?

If you are lost with numpy, try first to use Python lists (https://docs.python.org/3/tutorial/datastructures.html#more-on-lists) and implement all the mathematical operations in pure Python, and only then move to numpy.

Task 5: Initialize your network

Write a function (or a constructor method, if you prefer object-oriented programming) to initialize the network.

For now, let's just initialize weights with normally distributed values in [0, 1] range. Later, we will have a better initialization.

Task 6: Implement a forward pass

Write the code to perform a forward pass through your network given an image as a 1D vector. You probably want to use a dot-product function supplied with numpy (`numpy.dot`).

For the ReLU, consider using `numpy.maximum` (https://numpy.org/doc/stable/reference/generated/numpy.maximum.html).

Run it on one of the training samples. The output will be meaningless as we haven't done any training yet. However, make sure you have no exceptions and everything works as expected.

Now initialize the network with zero weights, and apply the network on one training sample. Does the output make sense?

What happens with the output if you initialize the weights with very large numbers?

Task 7: Implement a loss function

Implement the softmax function.

Compute the initial total loss of your network against the training labels.

Does the loss make sense given that your network has been initialized randomly?

Task 8: Implement backpropagation

Write down the derivatives for the loss function as well as all the hidden layers on paper. Once you have that, figure out how to compute the change for each individual node given one training sample in code.

Now generalize this to the whole network using matrix operations.

Run a backpropagation for a single training sample. No exceptions and no problems?

Finally, implement a stochastic gradient descent for your network, and train it for a while. Does the loss of your network go down?

You probably have to play with the batch size and learning rate a bit (remember the linear decay!).

To avoid overfitting on the test data, use only 75% of your training set for training and use the remaining 25% for the validation, *i. e.,* to select the parameters (such as batch size and learning rate).

Make sure you store the state of the training (*e.g.,* using pickle, https://docs.python.org/3/library/pickle.html) so that you can resume if there is a crash.

Store the loss over training iterations, *e.g.,* epochs, for future comparisons.

Task 9: Implement better initialization

We saw in the lecture a couple of possible initializations.

Since we are using ReLU, search on the Internet or read the paper He et al., “Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification”, CVPR 2015 to figure out how to initialize the weights. If you are too lost, implement the Xavier initialization.

Now plot the loss over training iterations, and compare it against the training of the network with normally-distributed random initialization. Did your training improve?

Task 10: Implement Momentum

Adapt the learning rate using the Momentum method. What happens with the training loss?

Task 11: Overfitting *versus* underfitting

Finally, plot the training loss against the validation loss and decide when to stop the training.

To round it all up, run the network on the test set. What does the confusion matrix on the test data look like? (https://en.wikipedia.org/wiki/Confusion_matrix; note that we have multiple classes, one for each digit)

If you are really, really lost

There are many resources on the Internet, apart from the literature we have already recommended in the lecture.

For example, we found this series of videos to be very instructive: https://www.youtube.com/watch?v=lGLto9Xd7bU&list=PLQVvvaa0QuDcjD5BAw2DxE6OF2tius3V3