# Deep Learning Course: Lab 4

Alex Stergiou, Esther Rodrigo, Fawaz Sammani

24th October 2023

## 1 Artificial Neural Network

### 1.1 Theory

In machine learning, artificial neural networks (ANNs) are learnable models that can solve real-valued, discrete-valued, and vector-valued tasks. We can use ANNs for problems such as learning to interpret complex real-world sensor data. Inspired by the biological neural system in the human brain, ANNs contain densely connected neuron units ($f(x) = \mathbf{w}\mathbf{x} + \mathbf{b}$), where each unit takes a number of real-valued inputs and produces a single real-valued output. Outputs can be then used to approximate a target value (output layers) or be passed to subsequent layers (input or hidden layers).

To train a neural network with an optimization method; e.g, gradient descent, we need to know the partial derivatives of our score function with regard to each network's parameters. For a network with no hidden layer (similar to logistic regression models), this is quite trivial. However, using the same logic with multi-layer networks will only give us the gradient of the last layer. Luckily, backpropagation provides a way to calculate the gradient of all the layers using the chain rule. We will cover backpropagation in detail in the lecture of the subsequent week. But, if you would like to have an initial understanding, we recommend this tutorial.

## 1.2 Python Exercise

### 1.2.1 Exercise 1

In this exercise, you will have to build your neural network and perform a forward computation and a computation of the gradients (backward pass). Then, you will update the network's parameters using gradient descent. In this exercise, we use the digits dataset from `sklearn`, and some samples are illustrated in Fig. 1. You have to edit the `neural_network.ipynb` file. The architecture of our network is: input $\rightarrow$ fully connected $\rightarrow$ sigmoid $\rightarrow$ fully connected $\rightarrow$ sigmoid $\rightarrow$ output.

**Step 1: Load, split data, and convert labels to one-hot vectors.** As in exercise 1, you will load the dataset and split the data into training and test sets. You also need to add a bias term and convert the training output to a one-hot vector.

**Step 2: Forward computation.** In this step, you will implement several building blocks of a neural network, including the fully connected layer, the sigmoid activation function and the mean square error cost function. Afterwards, you need to stack each building block sequentially to construct the network architecture described above.

**Step 3: Backpropagation.** Using backpropagation, you will calculate the gradient for each parameter of the network in this step. First, we need to calculate the gradient for the backward pass for each building block of the network. Afterward, we traverse the network in reverse order to backpropagate the gradient.

**Step 4: Network training.** After calculating the forward and backward pass, we will train the network using batch gradient descent. You need to iterate over your training set several times. For each iteration, we do one forward pass, calculate the loss, and do one backward pass to calculate all the necessary gradients. Then, you update the network's parameters using the gradient descent rule.

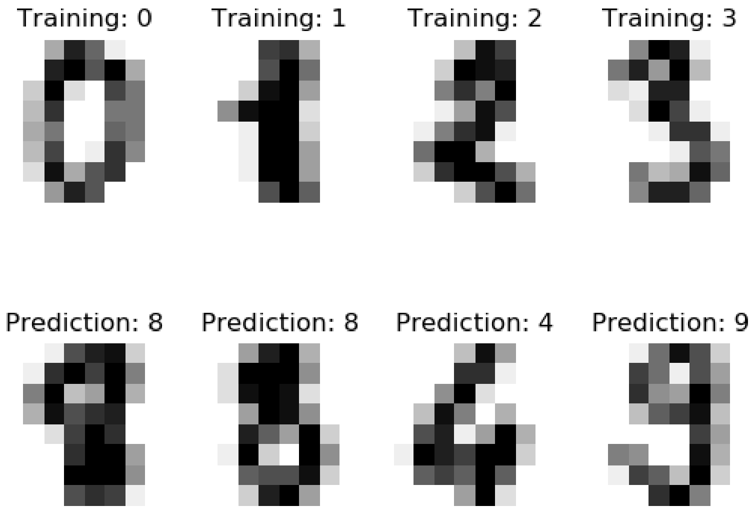

Figure 1: The digit dataset from `sklearn`.

**Step 5: Evaluation.** In this step, you will evaluate the performance of your network on the test set using accuracy as your metric.

## 2 Artificial Neural Network with Pytorch

In this exercise, we will create a similar model but in `PyTorch` instead of `sklearn`. The code for this exercise is in the file `MLP.ipynb`. Note that we have added TODO comments for all the lines that code is required from your side.

**Step 1: Load dataset** Run the code to load the digits dataset, split train/test, and add the bias term.

**Step 2: Plot image** Add code to plot the first image of the training set to visualize what your input looks like.

**Step 3: Build an ANN model using the Pytorch interface** In this step, you need to build an ANN model to classify the digit images by using functions and class instances from `PyTorch`.

- Start with transforming the data into `PyTorch` tensors
- Then, add the required arguments to your model which should be a sequence of linear layers and functions/activations. Use the instructions in the code of the simpleMLP class to construct the network of two linear layers with a ReLU activation function.
- Enable batching of the training data by using the `PyTorch` dataloader (Tip: You should complete the Dataset class for that).

**Step 3: Train the model** Train the model by following the instructions in the comments. Conduct forward passes to update the weights of the model using the cross-entropy loss defined in `PyTorch`. Set the following hyperparameters: use Adam as your optimizer, a learning rate of 0.001, and a hidden layer size of 32. You will train the model for 50 epochs. Try the different optimizer variants, and different learning rates.

**Step 4: Evaluate the model** Fill in the missing code to perform prediction and evaluation.

**Step 5: Visualize the classification result** Run the code to visualize the classification results.

**Step 6: Run the model with different learning rates.** Plot the validation accuracy using different learning rates of [1e-1, 1e-2, 1e-3, 1e-4, 1e-5] by keeping the rest of the hyperparameters fixed as defined in step 3. Note that the e-suffix represents times ten raised to the power. What do you observe?

**Step 7: Run the model with different neural network hidden layer sizes.** Plot the validation accuracy using different neural network layer hidden sizes [16, 32, 64, 128, 256, 512] by keeping the rest of the hyperparameters fixed as defined in step 3. What do you observe?

**Step 8: Run the model with different optimizers** Plot the validation accuracy using different optimizers by keeping the rest of the hyperparameters fixed as defined at step 3. The set of optimizers is defined in the ipython notebook. What do you observe?

**Step 9: Add L2 and L1 regularization to your model** The most popular regularization terms are L2 regularization, which is the sum of squares of all weights in the model, and L1 regularization, which is the sum of the absolute values of all weights in the model. In `PyTorch`, we could implement regularization pretty easily by adding a term to the loss. After computing the loss, whatever the loss function is, we can iterate the parameters of the model, sum their respective square (for L2) or abs (for L1), and backpropagate. For the regularization use a lambda of 0.001.

**Step 10: Repeat steps 3-5** Do this for the newly defined network in the function ComplexMLP, where you should also use the Dropout, and the Batchnorm modules. More details about the network can be found in the ipython notebook.

## References

[1] `https://pytorch.org/assets/deep-learning/Deep-Learning-with-PyTorch.pdf`

[2] `https://pytorch.org/tutorials/beginner/basics/optimization_tutorial.html`

[3] `https://medium.com/unpackai/cross-entropy-loss-in-ml-d9f22fc11fe0`

[4] `https://blog.feedly.com/tricks-of-the-trade-logsumexp/`

[5] `https://gregorygundersen.com/blog/2020/02/09/log-sum-exp/`