COSC343/AIML402 S2 2022

# Tutorial 10: Multilayer perceptron (MLP)

**Tutorial objectives:**

- **Practice with solving regression and classification machine learning problems using the MLP model**
- **Develop a bit of intuitive feel for choosing MLP hyper-parameters (those aspects of the model that have to be set before training)**

## Regression vs. classification

1. Answer the following:
   (a) What's the difference between regression and classification?
   (b) Given a dataset of images, the task is to determine if a face is present in the image or not. Is this regression or classification?
   (c) Given a set of measurements from an array of sensors from industrial machinery, the task is to estimate of the remaining time of the machine before next maintenance cycle is due. Is this regression or classification?
   (d) Given a set of true or false answers for variety of symptoms, the task is to estimate the probability of patient having a flu? Is this regression or classification?

## Multilayer perceptron in Sklearn

For the following exercises you will be using sklearn libraries' implementatoon of an MLPs, which provides two classes: `MLPRegressor` and `MLPClassifier`. These function in a similar way to the learner classes you implemented in Tutorials 6,7 and 8 - you instantiate them, passing arguments for configuration to the initialiser, then call the `fit` method to train the model. The initialisers for each class have similar arguments, many of which we'll leave at their defaults. However, there are some that you'll definitely will need to set:

- `hidden_layer_sizes` – this is a tuple of numbers that dictates the architecture of the MLP – the length of the tuple is the number of hidden layers, and the value of each item is the number of neurons per layer;

- `activation` – this is a string specifying the activation function of the hidden neurons – you have a choice of 'identity' (which in lectures was referred to as *linear*, $\sigma_l$), 'logistic' (which in lecture was referred to as *sigmoid*, $\sigma_s$), 'tanh' and 'relu';
- `learning_rate_init` – this is essentially the learning rate, except in Sklearn you can have learning rate that change over the course of training (can you think why that would be desired?) and so the argument `learning_parameter` is a string that specifies the schedule (default values 'constant', which is fine for us) and the actual learning rate is specified in the initial value of learning rate, `learning_rate_init`...which doesn't change when the schedule is specified to be 'constant';
- `max_iter` – the total number of epochs to train for.

The number of neurons in the output layer is deduced from the training data; the output activation and the choice of loss function for training is fixed for each class:

- `MLPRegressor` uses identity (linear) output neuron(s) and Mean Squared Error (MSE) loss for training
- `MLPClassifier` uses a sigmoid output neuron when only two classes are detected in the target used for training, or softmax output neurons when more than two classes aroudn found in the target labels, and the and Cross-Entropy (CE) loss for training.

There are three other arguments of the initialisers of both classes that you should set (and keep fixed for all the exercises) to the following values:

- `solver='sgd'` – this setting indicates that you want to train the model with Steepest Gradient Descent; we want to stick with 'sgd' since that's the only optimisation solver we covered in the paper;
- `tol=1e-5` – this parameter, along with `n_iter_no_change` (listed next), controls the stopping criteria of the training – if the loss does not improve more than `tol` for `n_iter_no_change` epochs, the training is halted before reaching `max_iter` epochs; the 'sgd' solver is a bit slow in the flatter areas of the loss surface (the other solvers improve on that, but may become a bit unstable in other circumstances) and so the tolerance must be set to allow continuation of training even with very tiny changes over many iterations with little change;
- `n_iter_no_change=10000` – see the explanation of `tol` above.

**Exercise 1:** In this exercise you train an MLP neural network model on the same regression problem you worked with in the previous lab. Create a new PyCharm project (select Anaconda's cosc343 environment for the interpreter), and download `helper.py` from Blackboard.

Create a new python file and at the top load the dataset like so:

```
from helper import *

X, y = load_reg1dataset()
y = y[:,0]
```

That last line converts an N×1 Numpy matrix `y` to an N-dimensional vector (literally, it selects the first column of the matrix). The reason for this is that in the previous exercise it was helpful to have `y` represented as a single column matrix, but (for some reason) Sklearn's MLP classes don't like that - they require the target labels to be a single vector.

The rest is up to you - figure out whether `MLPRegressor` or `MLPClassifier` is appropriate to use. Make sure to set the `solver`, `tol` and `n_iter_no_change` as specified above. The rest of the parameters that you need to set, are up to you.

There is a chance you won't get the architecture and/or the learning rate exactly right the first time. And so, it is a good idea to observe monitor the training, to gauge whether the model is learning well. There are two ways of doing that:

- Set `verbose=True` when instantiating your MLP and watch the console after invoking to the `fit` method. You will see printouts of the loss value after each iteration/epoch. The loss should be going down...though it may be going down slowly. Sometimes that means you should increase the learning rate (though not too much, because then the learning might get unstable) and sometimes that means you just have to train for longer...and sometimes that might mean that your model is not powerful enough to fit the data.
- Another way is to train model one epoch at the time and plot the results using visualisation provided in `helper.py`. Though in previous exercises the learner models allowed calling `fit` function many times, Sklearn's MLP models do not – `fit` is a single call that does all the training up to `max_iter` epochs and after that no more training is allowed. However, the API provides the `partial_fit` method, which only does one epoch of training and can be invoked many times. And so, assuming you've stored the referenced to your instantiated MLP object in variable `mlp`, you could train the model one epoch at a time and display the progress of training, like so:

  ```
  for epoch in range(numEpochs):
      mlp.partial_fit(X,y)
      plot_function(mlp, X, y, blocking=False)

  plot_function(mlp, X, y, blocking=True)
  ```

The `plot_function()` is a function implemented in `helper.py` that plots the output of a regression model (`mlp` in the code above) over a single attribute `X`. The `blocking` parameter controls whether to block execution of the script or not – you want no blocking during training, and blocking at the end, so that visualisation updates and lets the training continue, but does not close the figure immediately after training is done. You probably need to disable the ”Show plots in toolwindow” option in PyCharm→Preferences→Tools→Python Scientific for this to work.

And, by the way, if you're working with a classification problem, partial fit requires an argument specifying all the classes in the dataset, which you can pass in as shown in the code below. Also, if you're classifying a two attribute $X$, then `helper.py` provides a function to visualise classification regions as predicted by the classification model (`mlp` in the code below), which you can invoke, say every 20 epochs, like so:

```
for epoch in range(numEpochs):
    mlp.partial_fit(X,y,classes=np.unique(y))
    if epoch % 20 == 0:
        plot_classified_regions(mlp, X, y, blocking=False)

plot_classified_regions(mlp, X, y, blocking=True)
```

Try the visualisation with different activation functions. Do you seen any differnces?

**Exercise 2:** For this task, use an MLP model to solve the spiral problem. You can load the data as follows:

```
from helper import *

X,y = load_spiral()
```

Probably a good idea to watch the visualisation of the output as the network trains. If the network does not perform well on this problem, you might want to add some layers and/or neurons to the model.

**Exercise 3:** Try training an MLP on the MNIST dataset. To load the data do:

```
from helper import *

X, y = load_mnist()
```

The classification region visualisation will not work on this dataset, as the input image in this dataset consists of 784 attributes. The variable `X` is a 7000×28×28

Numpy array of 7000 28×28 pixel grey-scale images. For MLP you need to convert them to 7000 vectors of 28*28=784 dimensions each. You can do this using the Numpy's reshape function:

```
X = np.reshape(X, (70000,784))
```

For this exercise you should split the data into training and testing subsets. There is a good chance your MLP will train to perfect accuracy, but that doesn't necessarily translate into perfect generalisation. So, the accuracy that you need to eventually check for is the test data accuracy. If you want the standard, predefined trained/test split, the first 60000 points of `X` and `y` are the training points, so you can do:

```
X_train = X[:60000]
X_test = X[60000:]
y_train = y[:60000]
y_test = y[60000:]
```

You can also do your own, random training/test split – if you don't remember how, refer back to the code from Tutorial 6 and 7.

Sklearn's models provide a score function, which for classification is the accuracy. So, you can get accuracy of the model after training with the following code:

```
test_accuracy = net.score(X_test,y_test)

print("Test accuracy: %f" % (test_accuracy))
```

**Exercise 4:** Train an MLP network on the Fashion MNIST dataset. You can load the dataset as follows:

```
from helper import *

X, y = load_fashion_mnist()
```

The images are grey-scale, 28x28 pixels, and there is 70000 of them, so you can reshape `X` into a set of vectors and split it into the train and test sets exactly as you did it with the MNIST data.