# CO395 - Neural Networks Coursework

February 2019

## 1 Overview

In this assignment, you will implement a mini neural network library and then use it (or an alternate high-level library of your choice) to solve two practical problems inspired by industrial applications. The goals of this assignment are as follows:

- create a low-level implementation of a multi-layered neural network, including a basic implementation of the backpropagation algorithm,
- implement data preprocessing, training, and evaluation utilities,
- use the mini-library you have just developed (or a high-level library such as TensorFlow or PyTorch) to solve a regression task and a classification task, documenting your work in a report.

The mini-library in Part 1 must be designed using NumPy, and will require you to implement a linear layer class, a multi-layer network class, a trainer class, and a data preprocessing class.

For Part 2 and 3, you are free to use any library / approach of your preference when attempting the questions. We will provide support for both TensorFlow and PyTorch if you choose to use one of these libraries. These two parts of the assignment will be primarily be assessed on the basis of your report - we will not be assessing your code, unlike for Part 1 (although you will still be required to submit your code for parts 2 and 3, and we will run some of your code on a hidden dataset).

All implementations should be completed in Python 3. We do NOT provide any code or support for any other programming languages.

In addition to the instructions provided in this document, we provide a suite of tests for you to test your code prior to submission using the LabTS platform. To test your code, push to your Gitlab coursework repo (that will be generated for your group) and access the tests through the LabTS portal,

`https://teaching.doc.ic.ac.uk/labts/` .

Please note that the test suite is expected to evolve between the release and the submission dates of this coursework. **The tests are only an indication of the status of your code, and may not reflect your final mark in any part of this coursework.**

Read this manual THOROUGHLY.

**Deadline: Friday - 1 March, 2019 (7pm) on CATE.**

# 2 Setup

## 2.1 Working on DoC lab workstations (recommended)

You can also work from home and use the lab workstations. See this list `https://www.doc.ic.ac.uk/csg/facilities/lab/workstations` to ssh into one of the machines. In order to load all the packages that you might need for the course work, you can run the following command:

```
export PYTHONUSERBASE=/vol/bitbucket/nuric/pypi
```

We have installed any packages you might need for all the courseworks in CO395. If you don't want to type that command every time that you connect to your lab machine, you can add it to your bashrc:

```
echo "export PYTHONUSERBASE=/vol/bitbucket/nuric/pypi" >> ~/.bashrc
```

This way, every terminal you open will have that environment variable set. It is recommended to use "python3" exclusively. The current python3 version in lab machines is 3.6.7. To test the configuration:

```
python3 -c "import numpy as np; print(np)"
```

This should print:

```
<module 'numpy' from '/vol/bitbucket/nuric/pypi/lib/python3.6/site-packages/numpy/__init__.py>
```

## 2.2 Working on your own system

If you decide to work locally on your machine, then you must make sure that your code also runs on the lab machines using the above environment. Anything we cannot run will result in marks for the question being reduced by 30%.

Python>= 3.5: All provided code has been tested on Python versions 3.5 or 3.6. Make sure to install Python version 3.5 or 3.6 on your local machine, otherwise you might encounter errors! If you are working on Mac OSX, you can use Homebrew to brew install python3.

# Part 1: Creating a neural network mini-library

In this part, you will implement your own modular neural network mini-library using NumPy. This will require you to complete a number of classes in *src/nn_lib.py*.

A simple dataset (`https://en.wikipedia.org/wiki/Iris_flower_data_set`, provided in the file `iris.dat`) and sample code demonstrating intended usage is provided for debugging purposes as you work through this part of the coursework.

## Q1.1: Implement a linear layer:

For this question, you must implement a linear layer (which performs an affine transformation $XW + B$ on a batch of inputs $X$) by completing the following methods of the `LinearLayer` class:

**Constructor:** Define the following attributes in the constructor:

- NumPy arrays representing the learnable parameters of the layer, initialized in a sensible manner (hint: you can use the provided `xavier_init` function). Use the attributes `_W, _b` to refer to your weights matrix and bias respectively.

**Forward pass method:** Implement the `forward` method to do the following:

- Return the outputs of the layer given a NumPy array representing a batch of inputs.
- Store any data necessary for computing the gradients when later performing the backward pass in the `_cache_current` attribute.
- You are NOT allowed to use Python loops in this method (for efficiency reasons); use vectorized operations instead.

**Backward pass method:** Implement the `backward` method to do the following:

- Given the gradient of some scalar function with respect to the outputs of the layer as input, compute the gradient of the function with respect to the parameters of the layer and store them in the relevant attributes defined in the constructor (`_grad_W_current` and `_grad_b_current`).
- Compute and return the gradient of the function with respect to the inputs of the layer.
- You are NOT allowed to use Python loops in this method (for efficiency reasons); use vectorized operations instead.

**Parameter update method:** Implement the `update_params` method to perform one step of gradient descent on the parameters of the layer (using the stored gradients and the learning rate provided as argument).

Here is an example of how this class can be used:

```
layer = LinearLayer(n_in=3, n_out=42)
# `inputs` shape: (batch_size, 3)
# `outputs` shape: (batch_size, 42)
outputs = layer(inputs)
```

```
# `grad_loss_wrt_outputs` shape: (batch_size, 42)
# `grad_loss_wrt_inputs` shape: (batch_size, 3)
grad_loss_wrt_inputs = layer.backward(grad_loss_wrt_outputs)
layer.update_params(learning_rate)
```

## Q1.2: Implement activation function classes:

For this question, you must implement the `SigmoidLayer` and `ReluLayer` activation function classes (note: linear activation can be achieved without an activation class). In each case, complete the following:

**Forward pass method:** Implement the `forward` method to do the following:

- Returns the elementwise transformation of the inputs using the activation function.
- Store any data necessary for computing the gradients when later performing the backward pass in the `_cache_current` attribute.
- You are NOT allowed to use Python loops in this method (for efficiency reasons); use vectorized operations instead.

**Backward pass method:** Implement the `backward` method to do the following:

- Compute and return the gradient of the function with respect to the inputs of the layer.
- You are NOT allowed to use Python loops in this method (for efficiency reasons); use vectorized operations instead.

## Q1.3: Implement a multi-layer network:

For this question, you must implement a multi-layer network (consisting of stacked linear layers and activation functions) by completing the following methods of the `MultiLayerNetwork` class:

**Constructor:** Define the following in the constructor:

- Attribute/s containing instances of the `LinearLayer` class and activation classes, as specified by the arguments:
  - `input_dim`: an integer specifying the number of input neurons in the first linear layer,
  - `neurons`: a list specifying the number of output neurons in each linear layer,
  - `activations`: a list specifying which activation functions to apply to the output of each linear layer.
- Store your layer instances in the `_layers` attribute.

**Forward pass method:** Implement the `forward` method to do the following:

- Return the outputs of the network given a NumPy array representing a batch of inputs (note: the instances of the `LinearLayer` classes created in the constructor should automatically handle any storage of data needed to compute gradients).

**Backward pass method:** Implement the `backward` method to do the following:

- Given the gradient of some scalar function with respect to the outputs of the network as input, compute the gradient of the function with respect to the parameters of the network. (note: the instances of the `LinearLayer` classes created in the constructor should automatically handle any storage of computed gradients).
- Return the gradient of the function with respect to the inputs of the network.

**Parameter update method:** Implement the `update_params` method to perform one step of gradient descent on the parameters of the network (using the stored gradients and the learning rate provided as argument).

Here is an example of how this class can be used:

```
# The following command will create a MultiLayerNetwork object
# consisting of the following stack of layers:
#   - LinearLayer(4, 16)
#   - ReluLayer()
#   - LinearLayer(16, 2)
#   - SigmoidLayer()
network = MultiLayerNetwork(
    input_dim=4, neurons=[16, 2], activations=["relu", "identity"]
)
# `inputs` shape: (batch_size, 4)
# `outputs` shape: (batch_size, 2)
outputs = network(inputs)
# `grad_loss_wrt_outputs` shape: (batch_size, 2)
# `grad_loss_wrt_inputs` shape: (batch_size, 4)
grad_loss_wrt_inputs = network.backward(grad_loss_wrt_outputs)
network.update_params(learning_rate)
```

## Q1.4: Implement a trainer:

For this question, you must implement a "Trainer" class which handles data shuffling, training a given network using minibatch gradient descent on a given training dataset, as well as computing the loss on a validation dataset. To do so, complete the following methods of the `Trainer` class:

**Constructor:** Define the following in the constructor:

- An attribute (`_loss_layer`) referencing an instance of a loss layer class as specified by the `loss_fun` argument (which can take values `"mse"` or `"cross_entropy"`, corresponding to the mean-squrared error and binary cross-entropy losses respectively.).

**Data shuffling:** Implement the `shuffle` method to return a randomly reordered version of the data observations provided as arguments.

**Main training loop:** Implement the `train` method to carry out the training loop for the network. It should loop over the following `nb_epoch` times:

- If `shuffle_flag = True`, shuffle the provided dataset using the `shuffle` method.

- Split the provided dataset into minibatches of size `batch_size` and train the network using minibatch gradient descent.

**Computing evaluation loss:** Implement the `eval_loss` method to compute and return the loss on the provided evaluation dataset.

Here is an example of how this class can be used:

```
trainer = Trainer(
    network=network,
    batch_size=32,
    nb_epoch=10,
    learning_rate=1.0e-3,
    shuffle_flag=True,
    loss_fun="mse",
)
trainer.train(train_inputs, train_targets)
print("Validation loss = ", trainer.eval_loss(val_inputs, val_targets))
```

## Q1.5: Implement a preprocessor:

Data normalization can be crucial for effectively training neural networks. For this question, you will need to implement a "Preprocessor" class which performs min-max scaling such that the data is scaled to lie in the interval $[0, 1]$. Same as before, complete the methods of the `Preprocessor` class.

**Constructor:** Should compute and store data normalization parameters according to the provided dataset. This function does **not** modify the provided dataset.

**Apply method:** Complete the `apply` method such that it applies the pre-processing operations to the provided dataset and returns the preprocessed version.

**Revert method:** Complete the `revert` method such that it reverts the pre-processing operations that have been applied to the provided dataset and returns the reverted dataset. For any dataset `A`, `prep.revert(prep.apply(A))` should return `A`.

Here is an example of how this class can be used:

```
prep = Preprocessor(dataset)
normalized_dataset = prep.apply(dataset)
original_dataset = prep.revert(normalized_dataset)
```

## Part 2: Learning the Forward model of a robot

Now that we have a mini-library, we can use it to train a deep neural network to solve some tasks (alternatively, you are free to use another library of your choice to implement and train your models in the rest of the assignment). The first task consists in using a neural network to predict the position of a industrial robot given the motor commands (this is what we usually call a forward model). This coursework considers the ABB IRB 120, which is a very popular industrial robot (see figure 1), used in many assembly lines.

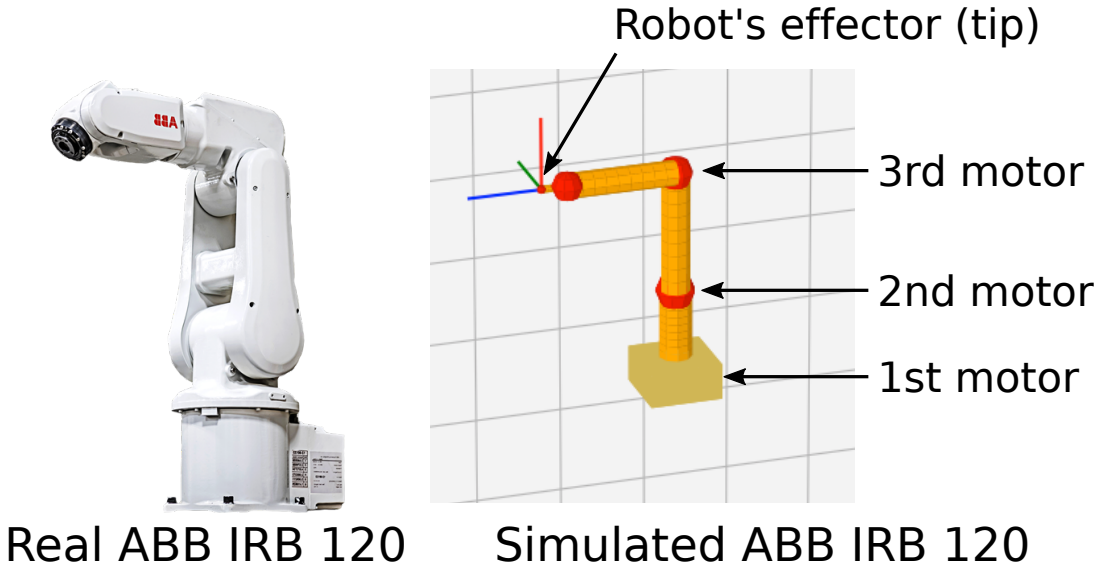


Figure 1: ABB IRB 120

While this robot has 6 degrees of freedom (i.e., 6 inputs), we will only consider the first three degrees of freedom, which control the final position of the robot's tip (end-effector).

The objective of this part of the coursework is to create a deep neural network that predicts the Cartesian position (x/y/z) of the robot's tip given the angular position of the first three motors ($\theta_1, \theta_2, \theta_3$). Such a model is usually called in robotics the "forward model" of the robot.

A (very simplistic) simulator is provided to allow you judge by yourself of the accuracy of your network, without the need of heavy libraries.

### Q2.1: Implement the forward model of the ABB IRB 120

The dataset *FM_dataset.dat* contains a matrix in which each line is a different sample. The first three columns correspond to the inputs (angular positions of the motors) and the last 3 ones correspond to the Cartesian position of the robot's tip (X/Y/Z position in mm).

For this question, open the file *src/learn_FM.py* and complete the main function to solve the task.

If you have used the mini-library developed in Part 1 of this coursework, you can visualize the accuracy of your model using the function `illustrate_results_FM` which randomly generates 10 angular positions, animates the simulated robot and displays the predictions of the network in the same space.

Among other comments that you find interesting, report the initial architecture used (number of layers, number of neurons per layer, activation function etc. ) and the obtained performance.

### Q2.2: Evaluate your architecture

In the file *src/learn_FM.py* add and implement a function named `evaluate_architecture` (which can take the arguments you think are necessary) and which prints or returns indicators about the performance of your network. Explain in your report the methodology that you use. You are allowed to use utilites from libraries such as `scikit-learn` to evaluate your model.

### Q2.3: Fine tune your architecture

Using the tools you have developed so far, perform a hyper-parameter search using a well thought methodology that you will detail in your report. You are allowed to use utilites from libraries such as `scikit-learn`.

Save your best performing model (hint: use the provided `save_network` function, if you have used the mini-library developed in Part 1).

Finally, write a function called `predict_hidden` in *src/learn_FM.py* which does the following:

- Accepts only one argument, `dataset`, a NumPy array containing data in the same format as that loaded from *FM_dataset.dat*.
- Performs any preprocessing steps on the dataset.
- Loads your best performing model (hint: use the provided `load_network` function, if you have used the mini-library developed in Part 1).
- Returns your best performing model's predictions on the provided dataset as a NumPy array of shape $(N, M)$, where $N$ is the number of observations in the dataset, and $M$ is the number of outputs.

We will use `predict_hidden` to evaluate your model's performance on a hidden dataset. Please make sure that your code runs in the provided lab environment. Any code that does not run or does not precisely adhere to the above specification will result in a loss of marks.

# Part 3: Learning a "region of interest" detector

For the last part of this coursework, the objective is to create neural network that predicts in which zone (or region of interest, ROI) the robot's gripper will be depending on the angular position of the first three motors $(\theta_1, \theta_2, \theta_3)$. Four zones are defined: 1) Zone 1, which represents the work-space in front of the robot, 2) zone 2, which represents two small lateral work spaces (this can for instance symbolize 2 small containers), 3) the Ground, which is an area you would like to avoid any damage on the robot, and 4) the rest of the work-space. An illustration of the different zones is given in figure 2

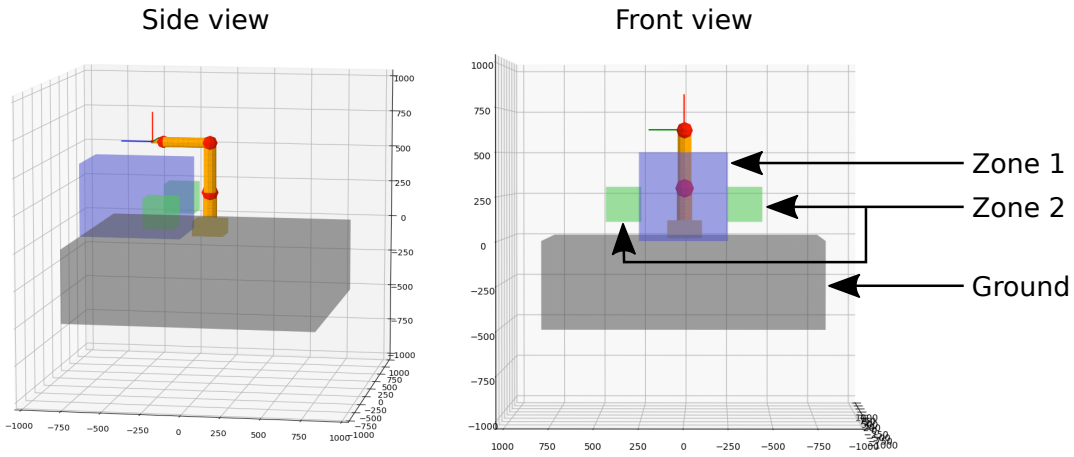


Figure 2: Regions of interest for the ABB IRB 120

## Q3.1: Implement the ROI detector

The dataset *ROI_dataset.dat* contains a matrix in which each line is a different sample. Like in the previous section, the first three columns correspond to the inputs (angular positions of the motors). However, for this task the last 4 columns represent a "one hot encoding" of the corresponding zone: A [1 0 0 0] vector means that the gripper is in the zone 1; [0 1 0 0] means that the gripper is in the zone 2, [0 0 1 0] the gripper is in the "ground", and [0 0 0 1] is in the unlabeled zone.

For this question, open the file *src/learn_ROI.py* and complete the main function to solve the task.

Like in the previous section, If you have used the mini-library developed in Part 1 of this coursework, you can visualize the accuracy of your model using the function `illustrate_results_ROI` which randomly generates 10 angular positions, animates the simulated robot and displays the predictions of the network in the same space.

Among other comments that you find interesting, report the initial architecture used (number of layers, number of neurons per layer, activation function etc. ) and the obtained performance.

## Q3.2: Evaluate your architecture

In the file *src/learn_ROI.py* add and implement a function named `evaluate_architecture` (which can take the arguments you think are necessary) and which prints or returns indicators

about the performance of your network. Explain in your report the methodology that you use. You are allowed to use utilites from external libraries to evaluate your model.

### Q3.3: Fine tune your architecture

Using the tools you have developed so far, perform a hyper-parameter search using a well thought methodology that you will detail in your report. You are allowed to use utilites from libraries such as `scikit-learn`.

Save your best performing model (hint: use the provided `save_network` function, if you have used the mini-library developed in Part 1).

Finally, write a function called `predict_hidden` in *src/learn_ROI.py* which does the following:

- Accepts only one argument, `dataset`, a NumPy array containing data in the same format as that loaded from *ROI_dataset.dat*.
- Performs any preprocessing steps on the dataset.
- Loads your best performing model (hint: use the provided `load_network` function, if you have used the mini-library developed in Part 1).
- Returns your best performing model's predictions on the provided dataset as a NumPy array of shape $(N, M)$, where $N$ is the number of observations in the dataset, and $M$ is the number of outputs.

We will use `predict_hidden` to evaluate your model's performance on a hidden dataset. Please make sure that your code runs in the provided lab environment. Any code that does not run or does not precisely adhere to the above specification will result in a loss of marks.

## Deliverables

You will have to submit two things on CATE:

- The SHA1 corresponding to the commit on you gitlab repository (used with LabTS). In this repository, you should have completed and pushed:
  - Completed version of `nn_lib.py`
  - Completed version of `learn_FM.py`
  - Completed version of `learn_ROI.py`
  - (optional) a readme explaining how to run your code, if needed.
- Report (PDF) containing answers to written questions ( Q2.1,Q2.2,Q2.3,Q3.1,Q3.2,Q3.3) for part II and III: for each of the questions.

## Grading scheme

Total = 100 marks.

- Part 1:
  - Linear layer (10 marks)

  - Implement activation function classes (5 marks)
  - Multilayer network (10 marks)
  - Trainer (10 marks)
  - Preprocessor (5 marks)
- Part 2:
  - Implement forward model (10 marks)
  - Evaluate your architecture (5 marks)
  - Fine tune your architecture (5 marks)
  - Evaluation on hidden dataset (5 marks)
- Part 3:
  - Implement forward model (10 marks)
  - Evaluate your architecture (5 marks)
  - Fine tune your architecture (5 marks)
  - Evaluation on hidden dataset (5 marks)
- Report presentation quality (10 marks)