CO395 coursework 2 report

Part 1 Neural Network Library Implementation

See codes in src/nn_lib.py

Part 2 Robot Arm Position Prediction

2.1 **Comments**

2.1.1 Network Architecture

The initial architecture we adopt is a **fully connected network**.

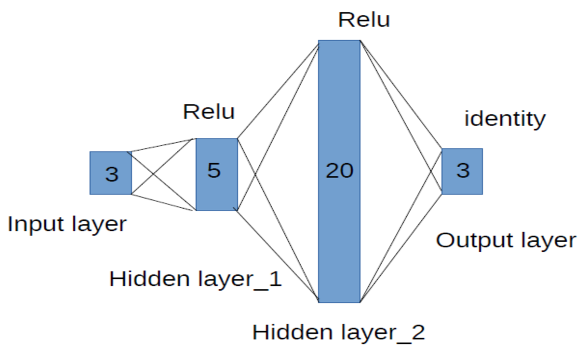


Figure 1: initial network architecture

The input dimension is 3. There are 3 fully connected layers, 5 neurons in the first hidden layer, 20 neurons in the second hidden layer and 3 neurons in the output layer. The internal layers adopt ReLU activation function to avoid exploding and vanishing gradient issue, and the output layer uses identity activation because this is not a classification problem.

2.1.2 Interesting Findings

We carried out comparisons between extreme wide network and extreme deep network:

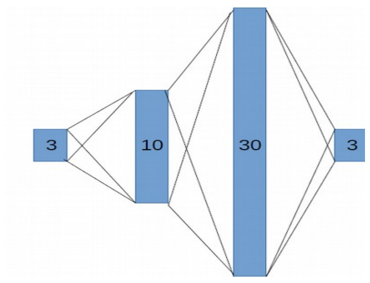


Figure 2: shallow and wide network

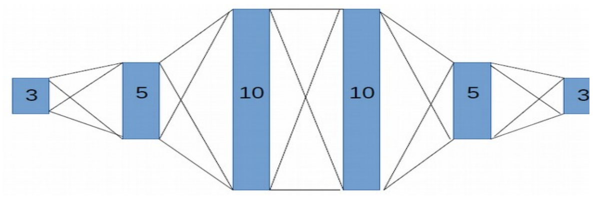


Figure 3: thin and deep network

The experiment results show that the wide but shallow networks generally have better performances than the thin but deep neural network. In addition, the wide networks present a more stable performance among tests. Therefore, in the following tuning process, we explore more on shallow and wide network structures.

2.2 **Evaluation Methodology**

The robot simulator is adopted to test the network. The function evaluate_architecture() samples 1000 random angles as input. Robot simulator calculates the targets position and the best network is loaded to predict the _position_. The average Euclidean distance is calculated as the evaluation criteria of the current network.

See codes in src/learn_FM.py: evaluate_architecture(network, prep, nb_pos=1000)
input argument prep is a Preprocessor object containing the training data information used to normalize the input data and revert the actual output position.

2.3 **Hyper-parameter Search Methodology**

- Learning Rate

To search for the proper learning rate, codes are added to monitor the loss of each epoch. Learning rate of 0.01, 0.01, 0.1 and 0.2 are compared. The observation shows that the loss decreases very slowly when using learning rate 0.001 and 0.01; learning rate 0.1 and 0.2 provides comparatively better performance but the difference between them is not obvious.

Therefore, to avoid oscillation during learning process, we have learning rate = 0.1 for later tuning.

- Activation Function

Experiments are carried out to compare the performance of using ReLU and sigmoid function, the results show that:

Architecture	Activation function	Learning rate	Average distance to target position
Initial architecture	All Relu in internal layer	0.1	36
Initial architecture	All sigmoid in internal layer	0.1	25 & nan

Sigmoid presents a better but unstable performance (there are cases when the gradients become too small and nan appears – the vanishing gradient issue of sigmoid). Therefore, we finally decide using activation function = ReLU

- Architecture

As mentioned in 2.1.2, we found shallow and wide networks tend to have better performance, thus in this part we mainly explore the number of neurons in the first and second layer and keep the number of layers as 3.

Architecture	Activation function	Learning rate	Average distance to target position
3, 10, 30, 3	All ReLU in internal layer	0.1	36
3, 10, 40, 3	All ReLU in internal layer	0.1	132
3, 20, 30, 3	All ReLU in internal layer	0.1	16
3, 30, 100, 3	All ReLU in internal layer	0.1	8
3, 80, 200, 3	All ReLU in internal layer	0.1	7

Therefore, the best model we have is:

Architecture	Activation function	Learning rate	Average distance to

			target position
3, 80, 200, 3	All ReLU in internal layer	0.1	7

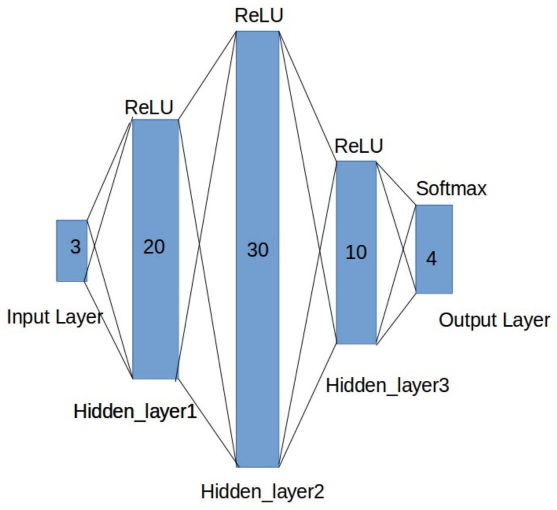
Part 3 Learning a “region of interest” detector

3.1 **Comments**

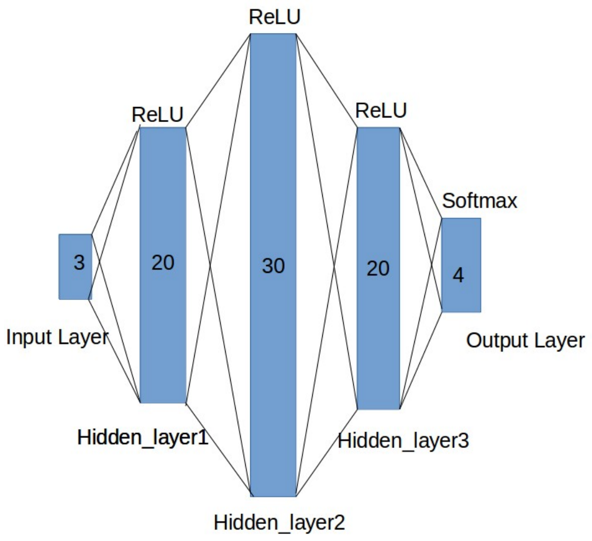
3.1.1 Network Architecture

The initial architecture we adopt is a **fully connected network**.

The input dimension is 3. There are 4 fully connected layers, 20 neurons in the first hidden layer, 30 neurons in the second hidden layer, 10 neurons in the third hidden layer and 4 neurons in the output layer. The internal layers adopt ‘ReLU’ activation function to avoid exploding and vanishing gradient issue, and the output layer uses sigmoid activation because this is a classification problem.



After tuning of the model, we found that change the number of neurons in third hidden layer to 20 could achieve better performance. Therefore, the final architecture is



3.1.2 Interesting Findings

After exploring the whole dataset, we found the number of data samples in each of the 4 classes given as:

Class No.	Class 0: Ground	Class 1: Zone 1	Class 2: Zone 2	Class 3: Unlabelled area
Data No.	1,325	1,522	144	12,634

As shown in the table above, the dataset gives an imbalanced distribution on the 4 classes, where Class 2 has only 144 data samples while Class 3 has 12,634 data samples. This implies us that if the dataset is used directly for training, it is very likely that the trained model would be biased to classify more Class 3, and possibly no Class 2 in the test dataset. In order to tackle this issue, there are 3 possible methods we could choose:

a. Adjusting weights for different classes
   During training, the model could be set to have specific weights for different classes. Usually, the weights are particularly chosen to be proportional to the number of samples for each class. In this case, we could set a large weight for Class 2, and a small weight for Class 3.

b. Up-sampling the minority class
   Up-sampling is the process of randomly duplicating samples from the minority class in order to reinforce its signal. In this case, we could choose to up-sample Class 2 to make the dataset balanced.

c. Down-sampling the majority class
   Down-sampling involves randomly removing samples from the majority class to prevent its signal from dominating the learning algorithm. In this case, we could down-sample Class 3 to make the dataset balanced.

Furthermore, it should be taken care that these methods could only be applied on training set solely, but not validation set or test set. Therefore, in terms of implementation, we first randomly split the whole dataset into train and validation set by a given proportion, and then a combination method of both up-sampling and down-sampling, where the 4 classes having 6000 samples each, is used to balance the training data.

3.2 **Evaluation Methodology**

The loss function used for the multi-class classification problem is cross entropy loss. Cross entropy loss measures the performance of a classification model whose output is a probability value between 0 and 1. Cross entropy equation is *-y_true*(log(y_predict)).* The cross-entropy loss for a multi-class model is the sum of the cross entropy of all classes. The objective is to minimize the cross-entropy loss

The dataset is splited into training and testing dataset. The testing data is used then to evaluate model. The train_test_split is performed before balancing the dataset. With a well-trained model, the unbalanced test data could still be classified well. The function evaluate_architecture() takes angular positions of motors as input. The trained model predicts the corresponding zone. The weighted average of the f1 score of 4 classes and confusion matrix generated by the prediction and ground truth of the corresponding zone are used as evalution criteria of the current network.

See codes in src/learn_ROI.py: evaluate_architecture(network, prep, xtest, ytest).
Input argument network is the trained network.

Input argument xtest, ytest are the test data splited before.
Input argument prep is a Preprocessor object containing the training data information used to normalize the input data. The output classes are in one hot representation thus does not need to be pre-processed.

The function will print out the confusion matrix and the error rate (Total number of wrong prediction/Total number of prediction). The returned value of evaluation function is the weighted average of f1 score.

3.3 **Hyper-parameter Search Methodology**

- Activation Function

As the objective is to predict which region of interest the robot arm will be in among the 4 different zones (classes), therefore the neural network we implemented should perform classification task. This suggests that the expected outputs would at least give a probability ranged in [0,1] for each class, so the output layer should be designed to be sigmoid function.

For activation functions used in the hidden layers, in order to prevent the vanishing gradient issue of sigmoid when the gradients in the hidden layers becomes too small, we choose to use 'ReLU' activation function.

- Learning Rate

To search for the proper learning rate, codes are added to monitor the loss of each epoch. Learning rate of 0.001, 0.01, 0.1 and 0.2 are compared. The observation shows that the loss decreases very slowly when using learning rates 0.001 and 0.01; learning rates 0.1 and 0.2 provide comparatively better performance but the larger loss for learning rate 0.2 might suggest that the learning rate has already reached the optimal value. Here we use average F1 score as an evaluation on the model performance, the model is said to classify well on the task if the average F1 score is closer to 1.

Learning rate	0.001	0.01	0.1	0.2
Average F1 score	0.0738	0.1846	0.9601	0.8075

Therefore, to avoid oscillation during learning process and achieve good performance, we have learning rate = 0.1 for later tuning.

- Architecture

Architecture	Activation function	Learning rate	Average F1 score
3, 20, 30, 4	ReLU – hidden layers, Sigmoid – output layer	0.1	0.9290
3, 20, 30, 10, 4	ReLU – hidden layers, Sigmoid – output layer	0.1	0.9601
3, 20, 30, 20, 4	ReLU – hidden layers, Sigmoid – output layer	0.1	0.9847

Therefore, the best model we have is:

Architecture	Activation function	Learning rate	Average F1 score
3, 20, 30, 20, 4	ReLU – hidden layers,	0.1	0. 9847

	Sigmoid – output layer		