UNIVERSITY OF WATERLOO

FACULTY OF ENGINEERING

ECE 457B - Assignment 1

Prepared by:

Jianxiang (Jack) Xu [20658861]

12 February 2021

Table of Contents

1 Problem 1: Perceptron [Full Implementation: Code 9]

1.1 (a): Derivation of Δw_i (5 marks)

$$\nabla_w E(w) = \frac{\partial \left[\frac{1}{2}\left(t^{(k)} - s(\sum_i w_i x_i^{(k)})\right)^2\right]}{\partial w} \tag{1}$$

$$\textbf{Let } f(w) = t^{(k)} - s(\sum_i w_i x_i^{(k)}) \tag{2}$$

$$\therefore \quad \nabla_w E(w) = \frac{\partial \left[\frac{1}{2}(f(w))^2\right]}{\partial w} = f(w)\frac{\partial f(w)}{\partial w} \tag{3}$$

$$\textbf{Let } g(w) = \sum_i w_i x_i^{(k)} \tag{4}$$

$$\therefore \quad \frac{\partial f(w)}{\partial w} = \frac{\partial \left[t^{(k)} - s(\sum_i w_i x_i^{(k)})\right]}{\partial w} = \frac{\partial \left[t^{(k)} - s(g(w))\right]}{\partial w} = \frac{\partial\left[-s(g(w))\right]}{\partial w} \tag{5}$$

$$\because \quad s'(x) = s(x)(1-s(x)) \text{ [Sigmoid Derivative]} \tag{6}$$

$$\therefore \quad \frac{\partial f(w)}{\partial w} = -\left[s(g(w))\right]\cdot\left[1 - s(g(w))\right]\cdot\frac{\partial g(w)}{\partial w} \tag{7}$$

$$\because \quad \frac{\partial g(w)}{\partial w} = \frac{\partial\left[\sum_i w_i x_i^{(k)}\right]}{\partial w} = \sum_i x_i^{(k)} \tag{8}$$

$$\therefore \quad \frac{\partial f(w)}{\partial w} = -\left[s(g(w))\right]\cdot\left[1 - s(g(w))\right]\cdot\sum_i x_i^{(k)} \tag{9}$$

$$\therefore \quad \nabla_w E(w) = f(w)\frac{\partial f(w)}{\partial w} \tag{10}$$

$$= \left[t^{(k)} - s(\sum_i w_i x_i^{(k)})\right]\cdot\left\{-\left[s(g(w))\right]\cdot\left[1 - s(g(w))\right]\cdot\sum_i x_i^{(k)}\right\} \tag{11}$$

$$= \left[t^{(k)} - s(\sum_i w_i x_i^{(k)})\right]\cdot\left\{-\left[s(\sum_i w_i x_i^{(k)})\right]\cdot\left[1 - s(\sum_i w_i x_i^{(k)})\right]\cdot\sum_i x_i^{(k)}\right\} \tag{12}$$

$$= \left[t^{(k)} - s(\sum_i w_i x_i^{(k)})\right]\cdot\left\{-\left[\frac{1}{1+e^{-(\sum_i w_i x_i^{(k)})}}\right]\cdot\left[1 - \frac{1}{1+e^{-(\sum_i w_i x_i^{(k)})}}\right]\cdot\sum_i x_i^{(k)}\right\} \tag{13}$$

$$= \left[t^{(k)} - s(\sum_i w_i x_i^{(k)})\right]\cdot\left\{-\left[\frac{e^{-(\sum_i w_i x_i^{(k)})}}{\left[1+e^{-(\sum_i w_i x_i^{(k)})}\right]^2}\right]\cdot\sum_i x_i^{(k)}\right\} \tag{14}$$

$$\textbf{By definitions: } s = s(g(w)) = s(\sum_i w_i x_i^{(k)}) = \frac{1}{1+e^{-(\sum_i w_i x_i^{(k)})}} \tag{15}$$

$$\therefore \quad \nabla_w E(w) = \left[t^{(k)} - s\right]\cdot\left\{-\left[e^{-(\sum_i w_i x_i^{(k)})}\cdot s^2\right]\cdot\sum_i x_i^{(k)}\right\} \tag{16}$$

$$= -\left(t^{(k)} - s\right)\left(s^2\cdot e^{-(\sum_i w_i x_i^{(k)})}\right)\cdot\sum_i x_i^{(k)} \tag{17}$$

As a result:

$$\because \quad \nabla_w E(w) = \sum_i (\nabla_w E(w))_i \tag{18}$$

$$\therefore \quad (\nabla_w E(w))_i = -\left(t^{(k)} - s\right)\left(s^2 \cdot e^{-(\sum_i w_i x_i^{(k)})}\right) \cdot x_i^{(k)} \tag{19}$$

$$\therefore \quad \Delta w_i = -\eta (\Delta_w E(w))_i = \eta \left(t^{(k)} - s\right)\left(s^2 \cdot e^{-(\sum_i w_i x_i^{(k)})}\right) x_i^{(k)} \tag{20}$$

Equation 1.1: Final Derived Equation

$$\Delta w_i = \eta \left(t^{(k)} - s\right)\left(s^2 \cdot e^{\left(-\sum_i w_i x_i^{(k)}\right)}\right) x_i^{(k)} \tag{21}$$

Q.E.D.

1.2 (b): Programs

The implementation can be seen in Code 1 and Code 2:

```
def perceptron(
    X: List[List[float]],
    y: List[int],
    max_pass=500
)-> [List[float], float, List[int]]:
    """
    @param          X: \in R^{nxd}
    @param          y: \in {-1,1}^n
    @param   max_pass: \in N
    """
    # shuffle data
    c = list(zip(X, y))
    np.random.shuffle(c)
    X, y = zip(*c)
    # train
    X = np.array(X)
    y = np.array(y)
    [n, d] = np.shape(X)
    w = np.random.uniform(-1,1,d) # assume x padded with first bias term
    mistake = []
    for t in range(0, max_pass): # max passes / iterations
        mistake.append(0)
        for i in range(0, n): # iterate through all dataset
            x_i = X[i, :]
            if (y[i] * (np.dot(x_i, w))) <= 0:
                w = w + y[i] * x_i
                mistake[t] += 1

        if (t >= 1) and (mistake[t] == mistake[t-1]):
            break # Converged

    return w, mistake
```

Code 1: Perceptron Implementation

```
def adaline(
    X: List[List[float]],
    y: List[float],
    # Configuration with Default Settings
    max_pass: int       = 500,
    eta: float          = 4e-3,
    error_tol: float    = 1e-5,
)-> [List[float], float, Dict]:
    """
    @param          X: \in R^{nxd}
    @param          y: \in R^n
    @param   max_pass: \in N
    @param        eta: \in [0,1]    (learning rate)
    @param  error_tol: \sim 0       (tolerance for steady state)
    """
    # shuffle data
    c = list(zip(X, y))
    np.random.shuffle(c)
    X, y = zip(*c)
    # train
    X = np.array(X)
    XT = np.transpose(X)
    y = np.array(y)
    [n, d] = np.shape(X)
    w = np.random.uniform(-1,1,d) # assume x padded with first bias term
    mistake = []
    # logger to track the progress
    training_log = {
        "t" : [],
        "w" : [],
        "training_error": [],
    }
    # training
```

```
    for t in range(0, max_pass): # max passes / iterations
        pw = copy.deepcopy(w)
        # update:
        f_err = ( np.dot(X, w) - y ) # pred - y
        dw = np.dot(XT, f_err)
        w = w - eta * dw
        # compute loss and error:
        error = 1 / 2 * (np.linalg.norm(f_err) ** 2)
        # log progress:
        training_log["t"].append(t)
        training_log["w"].append(w)
        training_log["training_error"].append(error)
        # stopping criteria:
        if np.linalg.norm((pw - w), ord=1) <= error_tol: # L1 Diff
            break # STOPPING

    return w, training_log
```

Code 2: Adaline Implementation

Alert 1.1: Assumption

For the perceptron program, the assumption on binary data label is $\in \{-1, 1\}$, hence, there would be data augmentation before feeding in training dataset if there is a need. In addition, the weight vector has been padded with the bias term as w_0, hence, there is an prior assumption on the dataset for an increased dimension of $(1+d)$, with d in terms of the data dimension.

In addition, to better tune the training parameters and monitor the progress of the algorithm, their progress would also be plotted with Code 3.

```
def print_report_adaline(
    training_log: Dict,
    tag: str,
):
    # plot status
    fig1 = plt.gcf()
    ax1 = plt.subplot(111)
    plt.plot(training_log["t"], training_log["training_error"])
    plt.title("Training Progress")
    plt.ylabel("Training Error")

    plt.xlabel("iteration")
    plt.show()
    fig1.savefig("fig/p1/p1_adaline_training_progress_{tag}.png".format(
        tag = tag
    ), bbox_inches = 'tight')

    print(training_log["t"][-1])
    print("> [{tag:8s}] T: {itr:3d} | Training Error: {train_err:.5f} ".format(
        tag        = tag,
        itr        = training_log["t"][-1],
        train_err  = training_log["training_error"][-1],
    ))

def print_report_perceptron(
    mistake: List[int],
    tag: str,
):
    fig1, ax1 = plt.subplots()
    ax1.plot((mistake))
    ax1.set_title("Mistakes vs. Passes")
    ax1.set_xlabel("number of passes")
    ax1.set_ylabel("number of mistakes")
    fig1.savefig("fig/p1/p1_perceptron_progress_{tag}.png".format(
        tag = tag
    ), bbox_inches = 'tight')
```

Code 3: Progress Report Implementation For Perceptron and Adaline

1.3 (c): Training Result

1.3.1 Perceptron Training Result

Training Progress (where we can see the convergence of the algorithm):

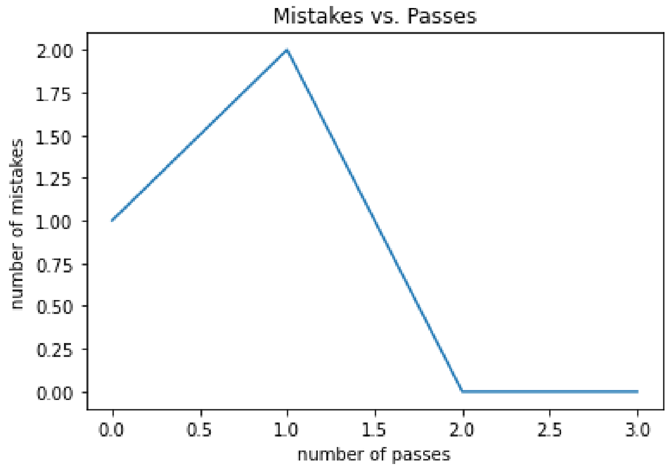


Figure 1-1. Perceptron Training Progress

Equation 1.2: Hyperplane Equation

$$1.818x + -2.540y + -0.132z = 1.634 \tag{22}$$

The resultant hyperplane:

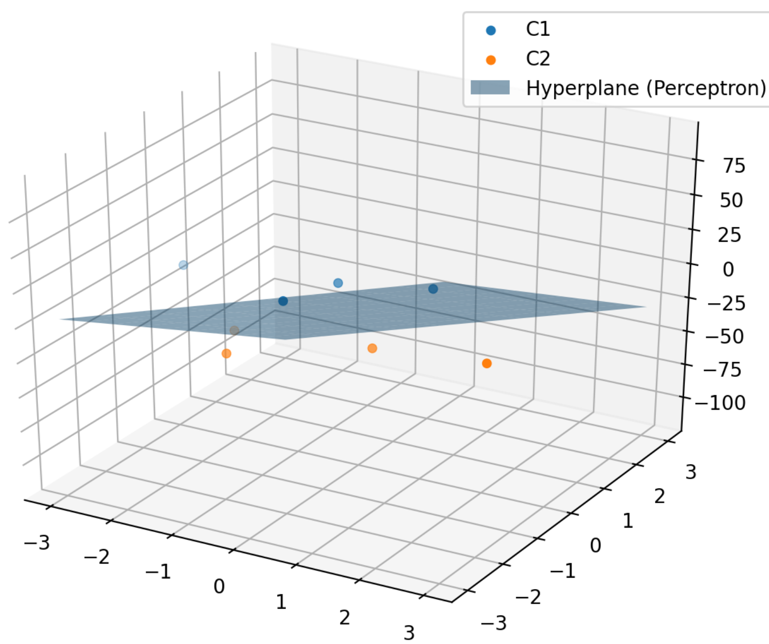


Figure 1-2. Perceptron Hyperplanes Result in 3D

Alert 1.2: Label Modification Needed

The given class label $\in \{0,1\}$ may not lead to a convergence of the perceptron, and the label shall be transformed into $\in \{-1,1\}$. Here is showing the modified result.

$TRAIN_Y = [-1,-1,-1,-1,1,1,1,1]$

1.3.2 Adaline Training Result

Training Progress (where we can see the convergence of the algorithm):

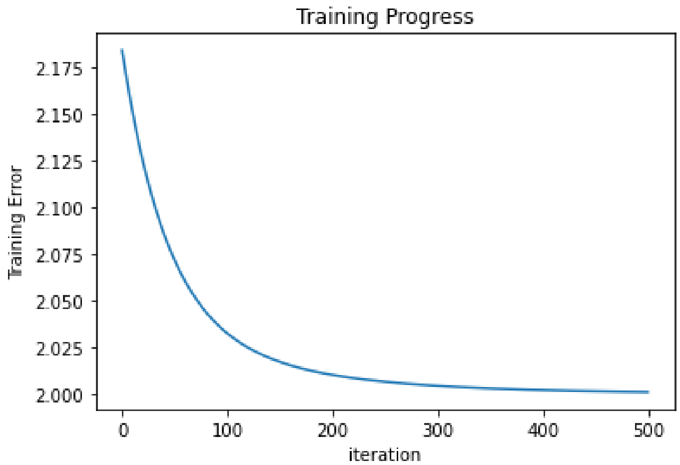
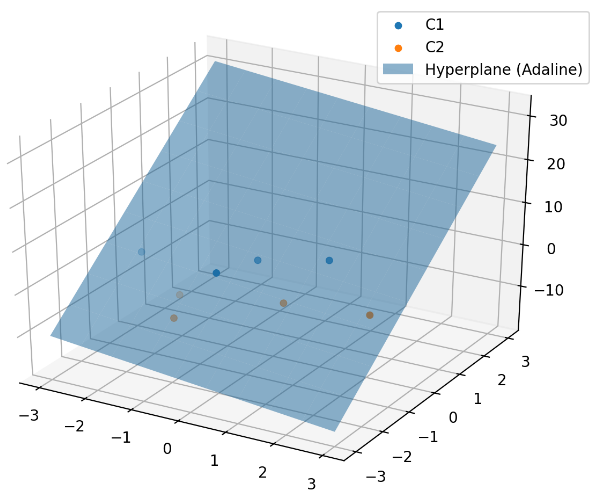


Figure 1-3. Adaline Training Progress

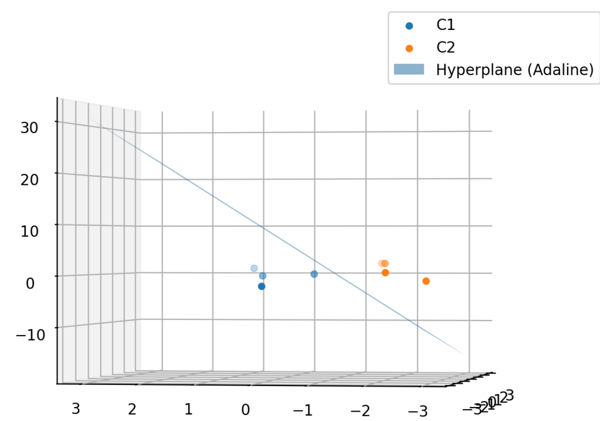
Equation 1.3: Hyperplane Equation

$$0.143x + -0.757y + 0.095z = 0.740 \tag{23}$$

The resultant hyperplane:



(a) Nominal View Angle



(b) Modified View Angle

Figure 1-4. Adaline Hyperplanes Result in 3D

Remark 1.1: Label Modification Not Required

The given class label $\in \{0,1\}$ would also lead to a convergence of the adaline, but for a fair comparison, the label here shall be $\in \{-1,1\}$. Here is showing the modified result.

$TRAIN_Y = [-1,-1,-1,-1,1,1,1,1]$

1.4 (d) Test Data

Since I define "C1" as -1 label for computation optimization and efficiency.

Let's redefine the testing data as follow:

```
y_test = -1
x_test = [-1, -1.3, -1.5, 2]
```

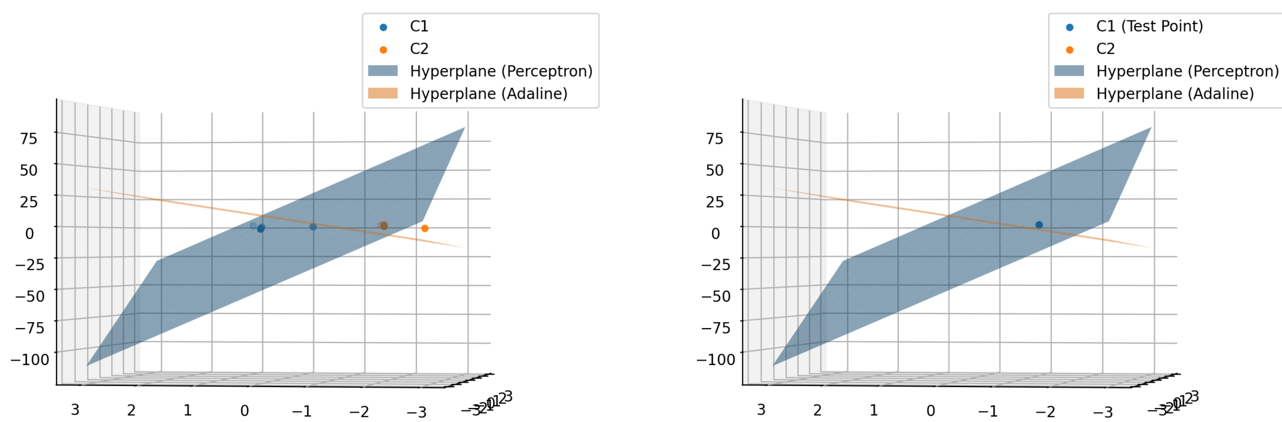
Hence, as long as the dot product with the plane norm vector agrees with the label, it would indicate the plane is a valid plane for the test data point, with $(y(\mathbf{x} \cdot \mathbf{w}) > 0)$:

```
is_perceptron = (y_test * (np.dot(x_test, w1))) > 0
is_adaline = (y_test * (np.dot(x_test, w2))) > 0
```

With result:

```
[ Valid Plane ] Perceptron: True | Adaline: False
```

Hence, the perceptron plane is good enough for the test point, whereas the hyperplane is not good for the test point. This can also be observed in Figure 1-5 below:



(a) Training Dataset (b) Test Dataset

Figure 1-5. Hyperplanes visualization with training dataset [Left] and test point [Right] in 3D

1.5 (e) Why Adaline has better capabilities than perceptron in terms of learning rule

They are different from the loss function, one with LMS and one with Hebbian learning rule. The LMS learning rule is a continuous and progressive learning step, whereas Hebbian learning rule is rather more discrete.

Specifically, the iterations of Adaline networks do not stop at arbitrary hyperplane, instead, it converges by reducing the least mean square error.

Whereas, the perceptron may stop with an arbitrary hyperplane depending on the order of the sampled point, whereas adaline will result an optimal and deterministic hyperplane, where its margin is the best (closest) possible geometrical plane. Since adaline utilizes the gradient decent with mean squared error, it will result a hyperplane that is close to each cluster of the dataset. In comparison, the perceptron would result an arbitrary margin with an arbitrary hyperplane, may result a non-ideal hyperplane, leading to a bias to a specific class (more close to one class).

Lastly, the perceptron requires the label to be $\in -1, 1$ with opposite sign, whereas, the adaline does not care much about the label sign, as long as the label is different.

2 Problem 2: Madaline [Full Implementation: Code 10]

The madaline structure would be consist of 3 fixed weights and bias:

```
# Hard-coded weights [biasi, wi1, wi2]
w1 = [-1, 1, 1]
w2 = [1, 1, 1]
w3 = [0, -1, 1]
```

Code 4: Madaline Weights

As Figure 2-1 shown, the hyperplanes with prescribed weight is able to separate the two output class (1 and -1):

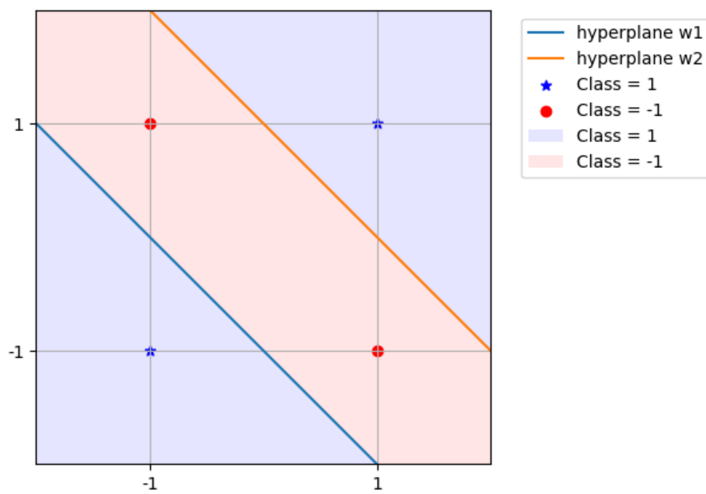


Figure 2-1. Hyperplanes visualization for Madaline XNOR gate in 2D

3 Problem 3: BPL [Full Implementation: Code 11]

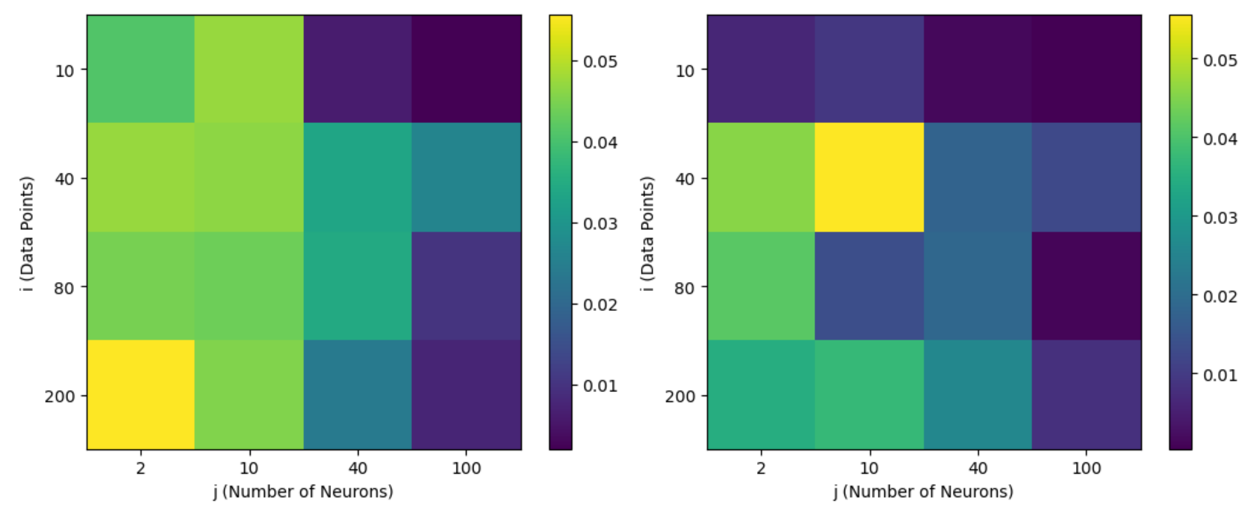
3.1 (a) Hyperparameter Tuning

For the dataset, the program would generate an arbitrary large enough (`MAX_DATA_SIZE`=500) dataset randomly in a uniform distribution. The data is then split into two portions: training data and testing data. In every combination, it will downsample the training data to a prescribed need by i consistently. In every trial, it will shuffle such data to randomize the order. Within the 10-fold operation, it will split the training data into 10 pieces, and one piece would be used as validation in each iteration of the k-fold operation.

As suggested in the assignment, 16 combinations of nodes and data points have been evaluated through 5 times of the 10-fold cross validation. The best out of 5 of the averaged loss from 10-fold evaluation has been recorded for each model. The max hard stop epochs were set to 1000 iterations with early stopping condition based on validation error. The resulted matrix of validation error and training error are tabulated in Table 3-1 and Table 3-2 and graphed in Figure 3-1 and Figure 3-2 respectively.

	Lowest Average k-Fold Training Error				Lowest Average k-Fold Validation Error			
	j=2	j=10	j=40	j=100	j=2	j=10	j=40	j=100
i=10	0.04101	0.04728	0.00614	0.00209	0.00637	0.00940	0.00151	0.00035
i=40	0.04729	0.04643	0.03344	0.02613	0.04572	0.05555	0.01800	0.01259
i=80	0.04453	0.04385	0.03457	0.01004	0.04141	0.01372	0.01871	0.00102
i=200	0.05555	0.04532	0.02384	0.00768	0.03471	0.03724	0.02599	0.00791

Table 3-1. Lowest average k-fold training and validation errors for $f_1(x)$

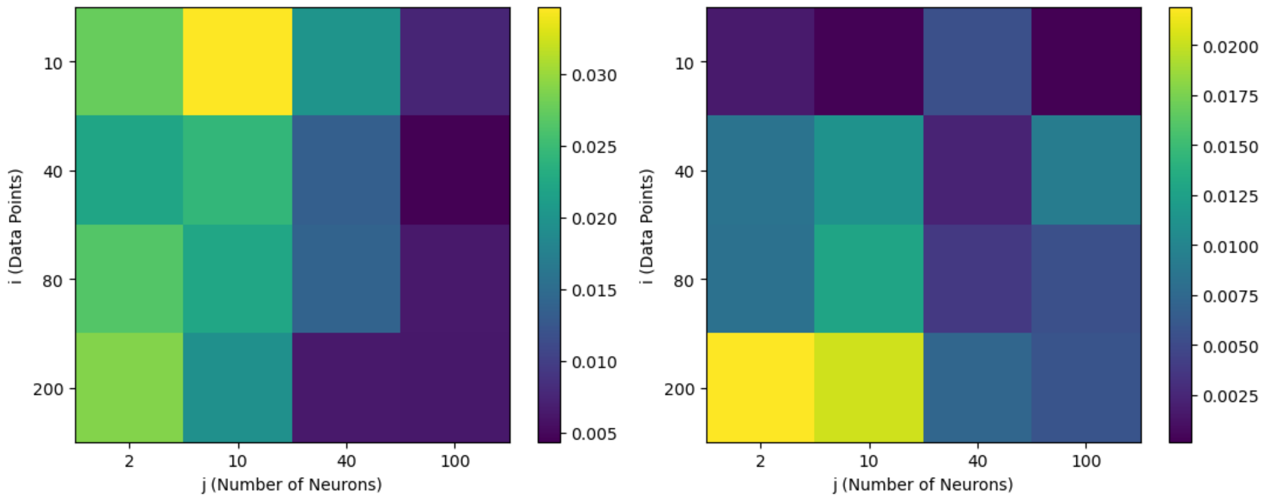


(a) Best Training Error (b) Best Validation Error

Figure 3-1. Visualized lowest average k-fold errors for $f_1(x)$

	Lowest Average k-Fold Training Error				Lowest Average k-Fold Validation Error			
	j=2	j=10	j=40	j=100	j=2	j=10	j=40	j=100
i=10	0.02760	0.03464	0.02003	0.00742	0.00169	0.00029	0.00544	0.00015
i=40	0.02215	0.02436	0.01346	0.00432	0.00839	0.01126	0.00232	0.00930
i=80	0.02652	0.02221	0.01397	0.00634	0.00829	0.01284	0.00376	0.00549
i=200	0.02898	0.01951	0.00642	0.00629	0.02190	0.02035	0.00737	0.00580

Table 3-2. Lowest average k-fold training and validation errors for $f_2(x)$

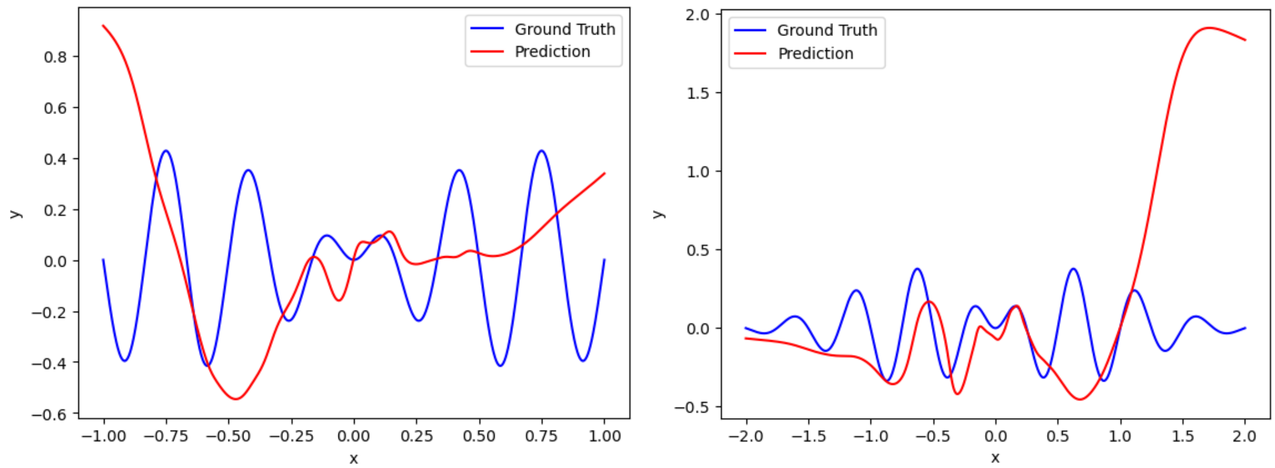


(a) Best Training Error (b) Best Validation Error

Figure 3-2. Visualized lowest average k-fold errors for $f_2(x)$

3.2 (b & c) Observations and Best Fitting Results

Based on the minimum validation error, it concludes the lowest error model is 10 data points and 100 neurons for both $f_1(x)$ and $f_2(x)$. However, this assumption quite limited, since the dataset is too small to conclude the validation. As we may see the training error is actually significantly larger than the validation error, which leads to an under-fitting model (as Figure 3-3 suggested as what we have expected). Ideally, we expect the validation error of a good fit model is higher than training error. Hence, we may need a better model and more generalized model that has best overall errors, where the validation error does not deviate much from the training error.



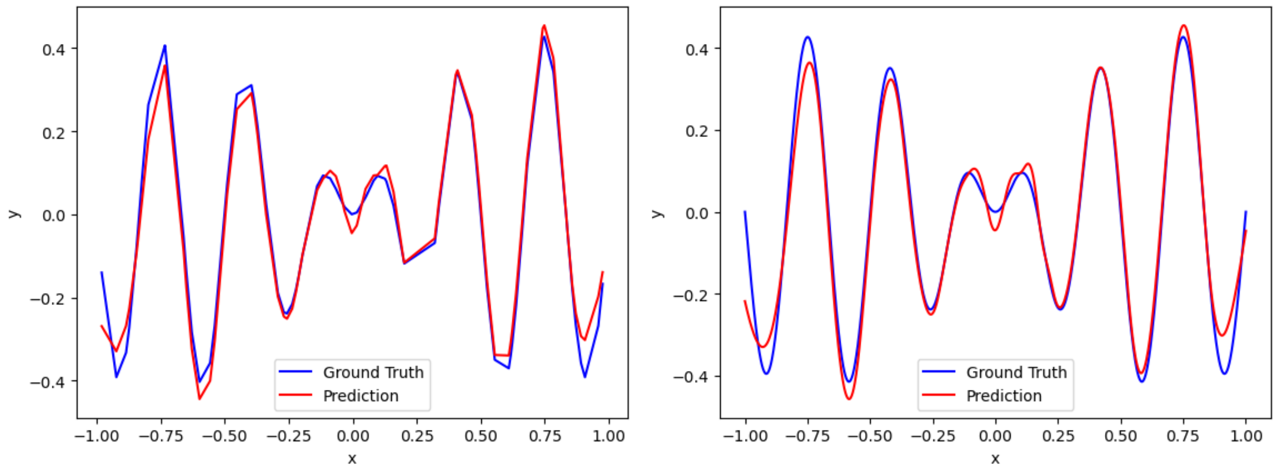
(a) Underfitting model of $f_1(x)$ (b) Underfitting model of $f_2(x)$

Figure 3-3. Underfitting models

Recall from the lecture contents, as model complexity increases (with higher j), it would be overfitting if the validation error also increases. The model right before such increase would be a good model. If we keep complexity unchanged, and if we observe there is an increase in validation error as the data size increases (with higher i), the model would be also overfitting. Hence, the one before such overfitting model would be a good model in that column. In addition, with consideration on the deviation between the training and validation error, a good model should exhibits all three characteristics.

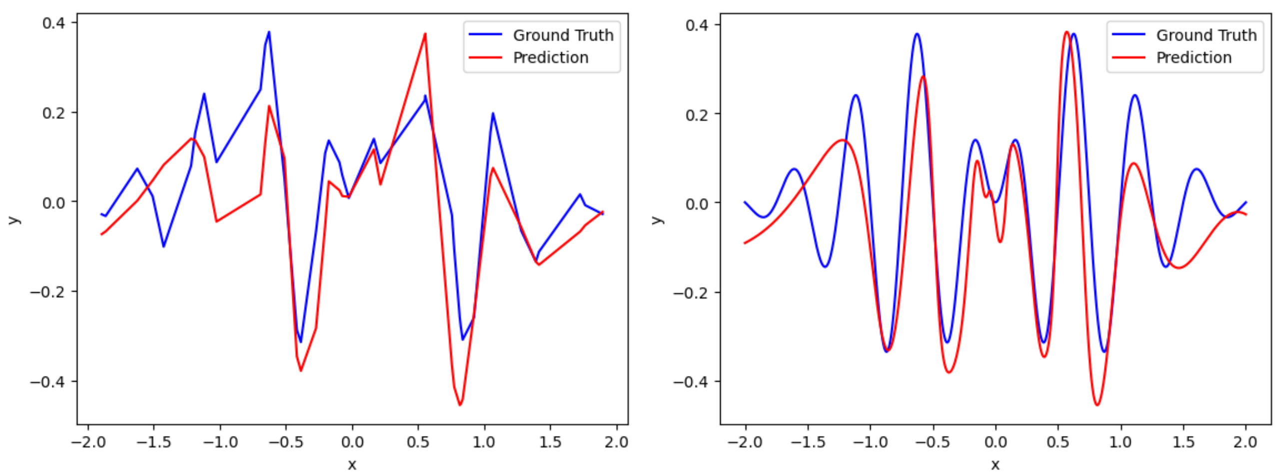
Hence, we may conclude a good model for $f_1(x)$ based on loss matrices in Table 3-1 and Figure 3-1 would be $(i = 80, j = 100)$, while a good model for $f_2(x)$ based on Table 3-2 and Figure 3-2 would be $(i = 40, j = 40)$.

As observed from Figure 3-4 and Figure 3-5 below, we can see these models are good model, that is neither overfitting nor underfitting the original function. They are much more generalized.



(a) Test Data Fitting (b) Overall Fitting

Figure 3-4. Final fitting of a good model $f_1(x)$ with $(i = 80, j = 100)$

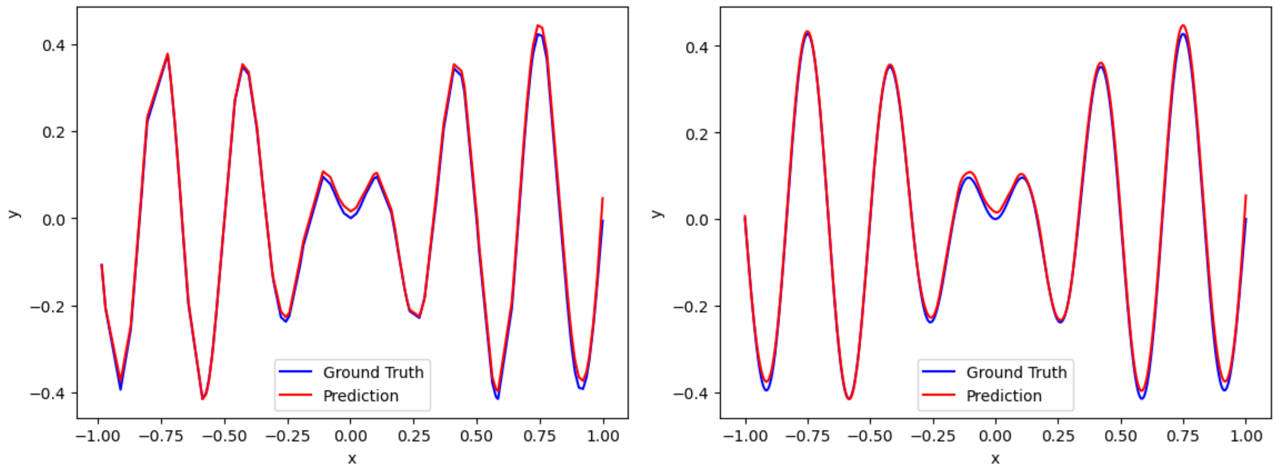


(a) Test Data Fitting (b) Overall Fitting

Figure 3-5. Final fitting of a good model $f_2(x)$ with $(i = 40, j = 40)$

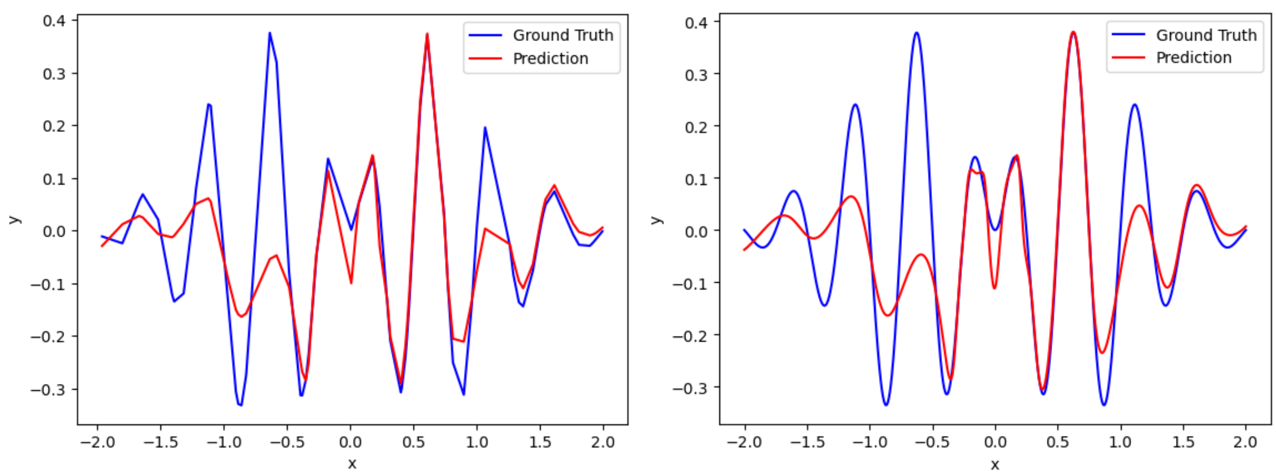
So, what if we choose the model that uses more data points?

As matrix suggested, we may expect an overfitting behaviour for both $f_1(x)$ and $f_2(x)$, as shown in both Figure 3-6 and Figure 3-7 for an increased dataset with $(i = 200, j = 100)$ and $(i = 80, j = 40)$ respectively.



(a) Test Data Fitting (b) Overall Fitting

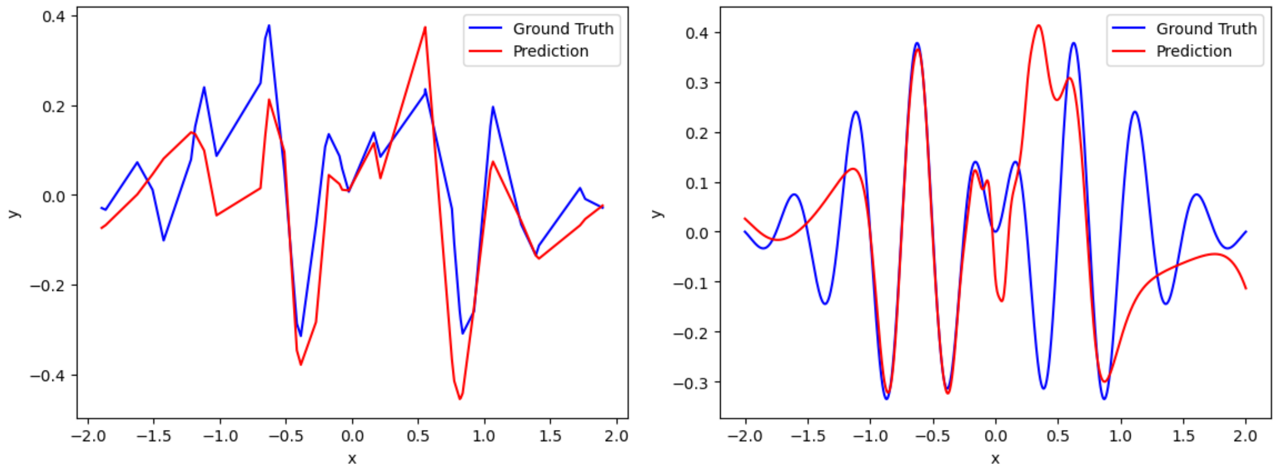
Figure 3-6. Overfitting model $f_1(x)$ with an increased population with $(i = 200, j = 100)$



(a) Test Data Fitting (b) Overall Fitting

Figure 3-7. Overfitting model $f_2(x)$ with an increased population with $(i = 80, j = 40)$

For $f_2(x)$, we may also try to increase the model complexity, we may also observe a regional over-fitting behaviour, which ruins the overall fitting as both hyperparameter matrices (Figure 3-2) and Figure 3-8 below suggested.



(a) Test Data Fitting (b) Overall Fitting

Figure 3-8. Overfitting model $f_2(x)$ with an increased complexity with $(i = 40, j = 100)$

3.3 (c*) Further Conclusion

In light of lecture materials, as we observed in this experiment as discussed in Section 3.2, a good model would have a good balance of variance and bias, where it can perform well in both training and test data. Overfitting often happens when the model is too complex for the given problem or excessive amount of training dataset. In addition, excessive amount of training would also lead to an overfitting of the model, which is handled by the early stopping callbacks in our case. And k-fold validation methods indeed help us to find a suitable good model based on the error matrices.

4 Problem 4: Neural Network Classifier [Full Implementation: Code 12]

4.1 MLP Classifier Training Result

To train the model, we shall first normalize the given dataset. 75% of dataset are used for training and 25% are used to evaluate the trained model. The best performed model with highest possible testing score would be used to predict the dataset for Section 4.2 later.

In addition, the label of the dataset is augmented into binary terms, with index of the bit as the label. For instance, label 3 would be represented by an array of $[0,0,1]$. We uses `'softmax'` to make elements of output vector in range(0,1) and sum up to 1, so that the output vector is a class probability identifier. The element with highest probability shall be the class of the dataset.

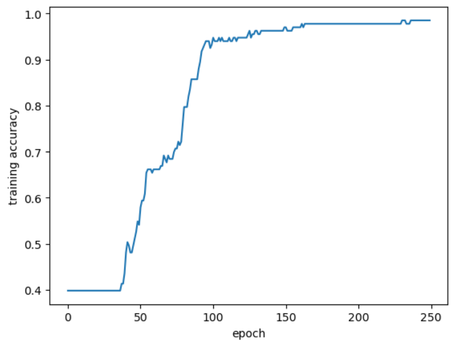
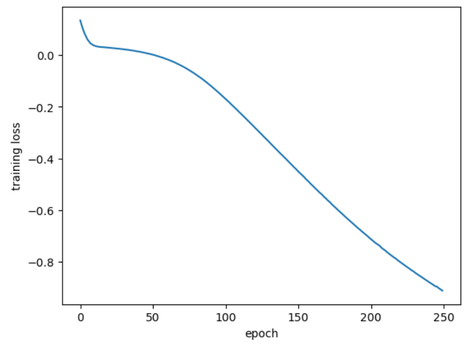
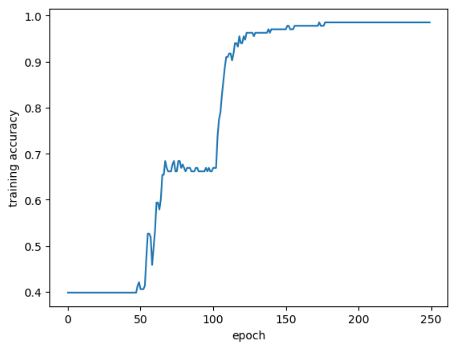
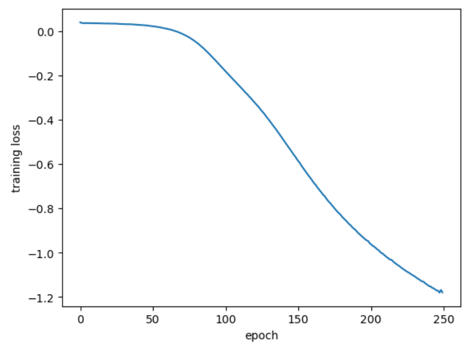
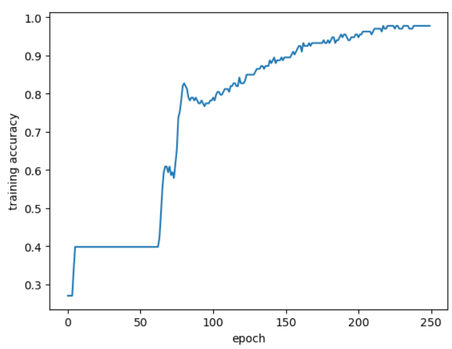
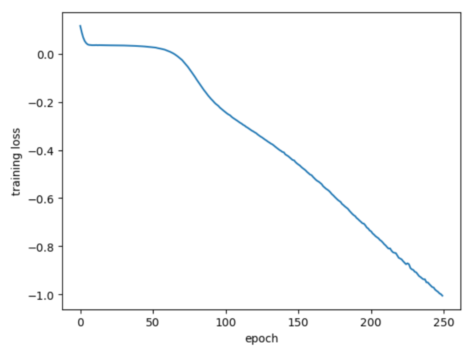
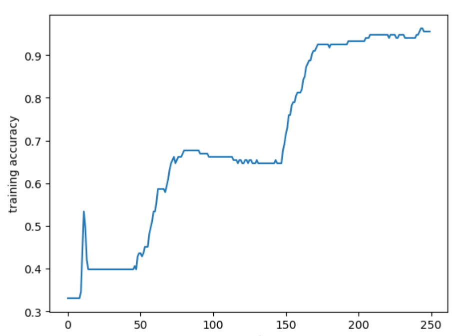
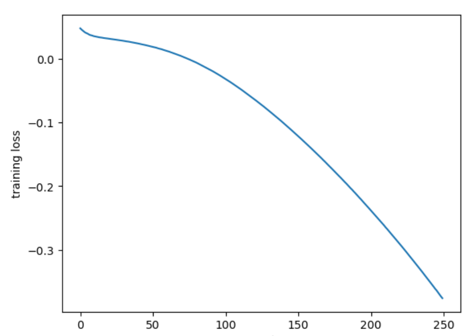
To find best possible classification model, we augmented various combinations of hidden layers and nodes as stated below, with an arbitrary max-epoch as hard stop (early-termination):

```
    # construct mlp test models
    MLP_DICT = {
        "t1": {
            "mlp": keras.models.Sequential([
                Dense(10, activation='sigmoid', input_shape=(13,)),
                Dense(20, activation='sigmoid'),
                Dense(3, activation='softmax')
            ]),
            "max_epoch": 250,
        },
        "t2": {
            "mlp": keras.models.Sequential([
                Dense(10, activation='sigmoid', input_shape=(13,)),
                Dense(20, activation='sigmoid'),
                Dense(20, activation='sigmoid'),
                Dense(3, activation='softmax')
            ]),
            "max_epoch": 250,
        },
        "t3": {
            "mlp": keras.models.Sequential([
                Dense(10, activation='sigmoid', input_shape=(13,)),
                Dense(20, activation='sigmoid'),
                Dense(20, activation='sigmoid'),
                Dense(20, activation='sigmoid'),
                Dense(3, activation='softmax')
            ]),
            "max_epoch": 250,
        },
        "t4": {
            "mlp": keras.models.Sequential([
                Dense(5, activation='sigmoid', input_shape=(13,)),
                Dense(5, activation='sigmoid'),
                Dense(3, activation='softmax')
            ]),
            "max_epoch": 250,
        },
        "t5": {
            "mlp": keras.models.Sequential([
                Dense(5, activation='sigmoid', input_shape=(13,)),
                Dense(5, activation='sigmoid'),
                Dense(5, activation='sigmoid'),
                Dense(3, activation='softmax')
            ]),
            "max_epoch": 250,
        },
        "t6": {
            "mlp": keras.models.Sequential([
                Dense(5, activation='sigmoid', input_shape=(13,)),
                Dense(15, activation='sigmoid'),
                Dense(15, activation='sigmoid'),
                Dense(3, activation='softmax')
            ]),
            "max_epoch": 250,
```

```
        }
    }
```

Code 5: Testing Models

The performance results are tabulated below:

Model	Train Accuracy	Test Accuracy	Training Accuracy Progress	Training Loss Progress
t1	98.50%	97.78%		
t2	98.50%	97.78%		
t3	97.74%	88.89%		
t4	95.49%	91.11%		

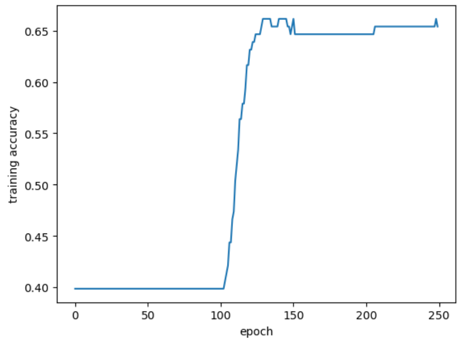
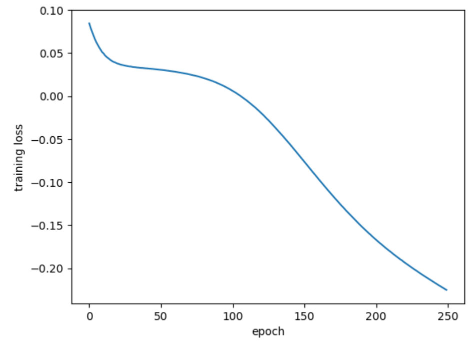
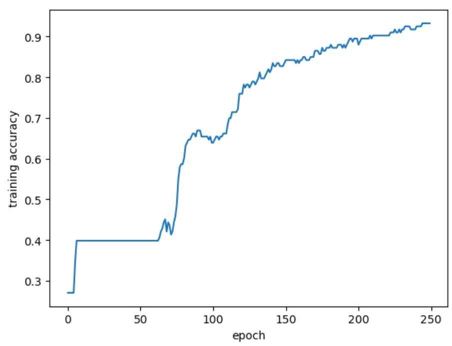
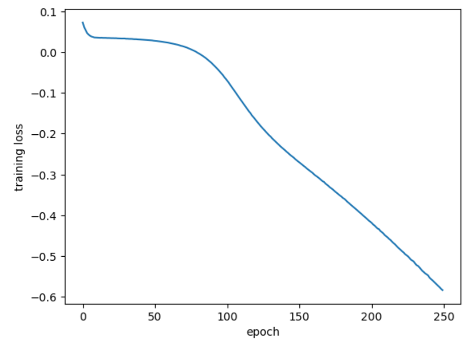
t5	65.41%	64.44%		
t6	93.23%	88.89%		

Table 4-1. Model Performance Summary Table

As Table 4-1 shown, and also seen in output Code 7 below, we may find the best performing model is the first model (Code 6) with 2 hidden layers and 10 and 20 neurons in the first and second layer respectively.

```
        "t1": {
            "mlp": keras.models.Sequential([
                Dense(10, activation='sigmoid', input_shape=(13,)),
                Dense(20, activation='sigmoid'),
                Dense(3, activation='softmax')
            ]),
            "max_epoch": 250,
        },
```

Code 6: Best Model

```
=== P4.1 ===
==== TEST [t1          ] ====
Train accuracy: 98.50 %
Test accuracy: 97.78 %
==== TEST [t2          ] ====
Train accuracy: 98.50 %
Test accuracy: 97.78 %
==== TEST [t3          ] ====
Train accuracy: 97.74 %
Test accuracy: 88.89 %
==== TEST [t4          ] ====
Train accuracy: 95.49 %
Test accuracy: 91.11 %
==== TEST [t5          ] ====
Train accuracy: 65.41 %
Test accuracy: 64.44 %
==== TEST [t6          ] ====
Train accuracy: 93.23 %
Test accuracy: 88.89 %
t1
{'mlp': <tensorflow.python.keras.engine.sequential.Sequential object at 0x7f8a36ff8a30>, 'max_epoch':
    250, 'train_accuracy': 98.49624037742615, 'test_accuracy': 97.77777791023254}
```

Code 7: Python Output

4.2 Classification Test Result

The classification result is as shown in Code 8 below:

```
=== P4.2 ===
[13.72, 1.43, 2.5, 16.7, 108, 3.4, 3.67, 0.19, 2.04, 6.8, 0.89, 2.87, 1285]
Normalized: [[0.70789474 0.13636364 0.60962567 0.31443299 0.41304348 0.83448276
  0.70253165 0.11320755 0.51419558 0.47098976 0.33333333 0.58608059
  0.71825963]]
[test_a    ]: Predicted ranking array: [[0.97562927 0.02270135 0.00166926]], Classified as: 1
[12.04, 4.3, 2.38, 22, 80, 2.1, 1.75, 0.42, 1.35, 2.6, 0.79, 2.57, 580]
Normalized: [[0.26578947 0.70355731 0.54545455 0.58762887 0.10869565 0.3862069
  0.29746835 0.54716981 0.29652997 0.11262799 0.25203252 0.47619048
  0.21540656]]
[test_b    ]: Predicted ranking array: [[0.01448519 0.8093967  0.17611817]], Classified as: 2
[14.13, 4.1, 2.74, 24.5, 96, 2.05, 0.76, 0.56, 1.35, 9.2, 0.61, 1.6, 560]
Normalized: [[0.81578947 0.66403162 0.73796791 0.71649485 0.2826087  0.36896552
  0.08860759 0.81132075 0.29652997 0.67576792 0.10569106 0.12087912
  0.20114123]]
[test_c    ]: Predicted ranking array: [[0.0041227  0.02048865 0.9753887 ]], Classified as: 3
```

Code 8: Test Result

The `‘a’` dataset is classified as class 1, `‘b’` dataset is classified as class 2, `‘c’` dataset is classified as class 3.

Appendix A P1 - Code

```python
# To add a new cell, type '# %%'
# To add a new markdown cell, type '# %% [markdown]'
# %%
from IPython import get_ipython

# %% [markdown]
# ## P1 - b & c & d) Program

# %%
# lib
import numpy as np
import matplotlib.pyplot as plt
import copy
from typing import List, Dict, Optional


# %%
# Perceptron
def perceptron(
    X: List[List[float]],
    y: List[int],
    max_pass=500
)-> [List[float], float, List[int]]:
    """
    @param          X: \in R^{nxd}
    @param          y: \in {-1,1}^n
    @param   max_pass: \in N
    """
    # shuffle data
    c = list(zip(X, y))
    np.random.shuffle(c)
    X, y = zip(*c)
    # train
    X = np.array(X)
    y = np.array(y)
    [n, d] = np.shape(X)
    w = np.random.uniform(-1,1,d) # assume x padded with first bias term
    mistake = []
    for t in range(0, max_pass): # max passes / iterations
        mistake.append(0)
        for i in range(0, n): # iterate through all dataset
            x_i = X[i, :]
            if (y[i] * (np.dot(x_i, w))) <= 0:
                w = w + y[i] * x_i
                mistake[t] += 1

        if (t >= 1) and (mistake[t] == mistake[t-1]):
            break # Converged

    return w, mistake

# Adaline
def adaline(
    X: List[List[float]],
    y: List[float],
    # Configuration with Default Settings
    max_pass: int       = 500,
    eta: float          = 4e-3,
    error_tol: float    = 1e-5,
)-> [List[float], float, Dict]:
    """
    @param          X: \in R^{nxd}
    @param          y: \in R^n
    @param   max_pass: \in N
    @param        eta: \in [0,1]    (learning rate)
    @param  error_tol: \sim 0       (tolerance for steady state)
    """
    # shuffle data
    c = list(zip(X, y))
    np.random.shuffle(c)
```

```
    X, y = zip(*c)
    # train
    X = np.array(X)
    XT = np.transpose(X)
    y = np.array(y)
    [n, d] = np.shape(X)
    w = np.random.uniform(-1,1,d) # assume x padded with first bias term
    mistake = []
    # logger to track the progress
    training_log = {
        "t" : [],
        "w" : [],
        "training_error": [],
    }
    # training
    for t in range(0, max_pass): # max passes / iterations
        pw = copy.deepcopy(w)
        # update:
        f_err = ( np.dot(X, w) - y ) # pred - y
        dw = np.dot(XT, f_err)
        w = w - eta * dw
        # compute loss and error:
        error = 1 / 2 * (np.linalg.norm(f_err) ** 2)
        # log progress:
        training_log["t"].append(t)
        training_log["w"].append(w)
        training_log["training_error"].append(error)
        # stopping criteria:
        if np.linalg.norm((pw - w), ord=1) <= error_tol: # L1 Diff
            break # STOPPING

    return w, training_log

def print_report_adaline(
    training_log: Dict,
    tag: str,
):
    # plot status
    fig1 = plt.gcf()
    ax1 = plt.subplot(111)
    plt.plot(training_log["t"], training_log["training_error"])
    plt.title("Training Progress")
    plt.ylabel("Training Error")

    plt.xlabel("iteration")
    plt.show()
    fig1.savefig("fig/p1/p1_adaline_training_progress_{tag}.png".format(
        tag = tag
    ), bbox_inches = 'tight')

    print(training_log["t"][-1])
    print("> [{tag:8s}] T: {itr:3d} | Training Error: {train_err:.5f} ".format(
        tag       = tag,
        itr       = training_log["t"][-1],
        train_err = training_log["training_error"][-1],
    ))

def print_report_perceptron(
    mistake: List[int],
    tag: str,
):
    fig1, ax1 = plt.subplots()
    ax1.plot((mistake))
    ax1.set_title("Mistakes vs. Passes")
    ax1.set_xlabel("number of passes")
    ax1.set_ylabel("number of mistakes")
    fig1.savefig("fig/p1/p1_perceptron_progress_{tag}.png".format(
        tag = tag
    ), bbox_inches = 'tight')

def w2str(w:List[float])->str:
```

```python
    return (" {A:.3f} x + {B:.3f} y + {C:.3f} z = {D:.3f}"        .format(A=w[1], B=w[2], C=w[3], D=w[0]))


# %%
def plot3d(
    X: List[List[float]],
    y: List[int],
    y_label: Dict[int, str],
    ws: Dict[str, List[float]],
    tag: str,
    view_opt: Optional[List[float]]=None
)-> None:
    X,y = np.array(X),np.array(y)
    # pre-processing
    x_min, x_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    y_min, y_max = X[:, 2].min() - 1, X[:, 2].max() + 1
    z_min, z_max = X[:, 3].min() - 1, X[:, 3].max() + 1
    px3d = np.transpose(X[:, 1:4])
    yt = np.transpose(y)

    # plot init
    fig1 = plt.gcf()
    ax = plt.axes(projection='3d')
    # ax.set_aspect(1)
    ax.grid()

    # Plot Points
    for y_val, label in y_label.items():
        ax.scatter(px3d[0][yt == y_val], px3d[1][yt == y_val], px3d[2][yt == y_val],            label=label, linewidth=0.5)

    # Plot Hyper-planes
    xx = np.linspace(-3,3,10)
    yy = np.linspace(-3,3,10)
    XX,YY = np.meshgrid(xx,yy)
    for name, w in ws.items():
        w = np.array(w)
        ZZ = (w[0] - w[1] * XX - w[2] * YY) / w[3]
        ax.w_xaxis.set_pane_color((1.0, 1.0, 1.0, 1.0))
        surf1 = ax.plot_surface(XX, YY, ZZ,alpha=0.5, label=name)
        surf1._facecolors2d=surf1._facecolors3d
        surf1._edgecolors2d=surf1._edgecolors3d
    if view_opt is not None:
        ax.view_init(elev=view_opt[0], azim=view_opt[1])
    ax.legend()
    plt.show()
    fig1.set_size_inches(8,6)
    fig1.savefig("fig/p1/p1_plot3d_{tag}.png".format(
        tag = tag
    ), bbox_inches = 'tight', dpi=200)


# %%
TRAIN_X = [[-1, 0.8, 0.7, 1.2], [-1, -0.8,- 0.7, 0.2], [-1, -0.5,0.3,- 0.2], [-1, -2.8, -0.1, -2], [-1, 1.2,- 1.7, 2.2], [-1, -0.8,-2, 0.5], [-1, -0.5,-2.7,- 1.2], [-1, 2.8, -1.4, 2.1]]
TRAIN_Y = [-1,-1,-1,-1,1,1,1,1]


# %%
get_ipython().run_line_magic('matplotlib', 'inline')
np.random.seed(341)
w1, mistake = perceptron(X=TRAIN_X, y=TRAIN_Y, max_pass=30)
print_report_perceptron(mistake, tag="")
print(w2str(w1))


# %%
get_ipython().run_line_magic('matplotlib', 'inline')
w2, log = adaline(X=TRAIN_X, y=TRAIN_Y)
print_report_adaline(log, tag="")
print(w2str(w2))
```

```


# %%
get_ipython().run_line_magic('matplotlib', 'inline')
plot3d(X=TRAIN_X, y=TRAIN_Y,
    y_label={
        -1  : "C1",
        1   : "C2"
    },
    ws={
        "Hyperplane (Perceptron)"   : w1,
    },
    tag="perceptron"
)
plot3d(X=TRAIN_X, y=TRAIN_Y,
    y_label={
        -1  : "C1",
        1   : "C2"
    },
    ws={
        "Hyperplane (Adaline)"      : w2
    },
    tag="Adaline"
)
plot3d(X=TRAIN_X, y=TRAIN_Y,
    y_label={
        -1  : "C1",
        1   : "C2"
    },
    ws={
        "Hyperplane (Adaline)"      : w2
    },
    tag="Adaline(div)",
    view_opt=[0, 190]
)
plot3d(X=TRAIN_X, y=TRAIN_Y,
    y_label={
        -1  : "C1",
        1   : "C2"
    },
    ws={
        "Hyperplane (Perceptron)"   : w1,
        "Hyperplane (Adaline)"      : w2
    },
    tag="both",
    view_opt=[0, 190]
)


# %%



# %%
## Qd) Plane Validation for test points
y_test = -1
x_test = [-1, -1.3, -1.5, 2]

is_perceptron = (y_test * (np.dot(x_test, w1))) > 0
is_adaline = (y_test * (np.dot(x_test, w2))) > 0

print("[ Valid Plane ] Perceptron: {0} | Adaline: {1}".format(is_perceptron, is_adaline))

# plot the point as well
plot3d(X=[x_test], y=[y_test],
    y_label={
        -1  : "C1 (Test Point)",
        1   : "C2"
    },
    ws={
        "Hyperplane (Perceptron)"   : w1,
        "Hyperplane (Adaline)"      : w2
```

```
    },
    tag="test_point",
    view_opt=[0, 190]
)


# %%
w3, log3 = adaline(X=XNOR_TRAIN_X, y=XNOR_TRAIN_Y)
print_report_adaline(log3, tag="")
print((w3))
```

Code 9: Perceptron and Adaline Implementation

Appendix B P2 - Code

```
# lib
import numpy as np
import matplotlib.pyplot as plt
import copy
from typing import List, Dict, Optional

train_x = np.array([[-1, 1], [1, -1], [1, 1], [-1,-1]])
train_y = np.array([-1, -1, 1, 1])

# Hard-coded weights [biasi, wi1, wi2]
w1 = [-1, 1, 1]
w2 = [1, 1, 1]
w3 = [0, -1, 1]

f_hyper_plane = lambda x, w: (w[0] - w[1] * x)/w[2]

# plot
fig = plt.figure()
ax = plt.axes()
ax.grid(True)
MIN, MAX = -2, 2
ax.set_xlim(MIN, MAX)
ax.set_ylim(MIN, MAX)

CLR_LUT = {-1:'r', 1:'b'}
MRK_LUT = {-1:'o', 1:'*'}
CLS_LABEL = {-1:"Class = -1", 1:"Class = 1"}

train_x_1 = train_x[np.where(train_y == 1)][:]
train_x_2 = train_x[np.where(train_y == -1)][:]

ax.scatter(train_x_1[:, 0], train_x_1[:, 1], c=CLR_LUT[1], marker=MRK_LUT[1], label=CLS_LABEL[1])
ax.scatter(train_x_2[:, 0], train_x_2[:, 1], c=CLR_LUT[-1], marker=MRK_LUT[-1], label=CLS_LABEL[-1])

X = np.linspace(MIN, MAX, 10)
Y1 = f_hyper_plane(X, w1)
Y2 = f_hyper_plane(X, w2)

plt.plot(X, Y1, label="hyperplane w1")
plt.plot(X, Y2, label="hyperplane w2")

ax.fill_between(X, MIN, Y1, facecolor='blue', alpha=0.1,label=CLS_LABEL[1])
ax.fill_between(X, Y2, MAX, facecolor='blue', alpha=0.1)
ax.fill_between(X, Y1, Y2, facecolor='red', alpha=0.1, label=CLS_LABEL[-1])
ax.legend(loc='upper left', bbox_to_anchor=(1.05, 1))
ax.set_aspect(1)
plt.xticks([-1, 1], [-1, 1])
plt.yticks([-1, 1], [-1, 1])
fig.savefig("fig/madaline.png", bbox_inches = 'tight')

print("Please find result @ fig/madaline.png")
```

Code 10: Madaline Implementation

Appendix C P3 - Code

```python
# python typical
import numpy as np
import matplotlib.pyplot as plt
import copy
from typing import List, Dict, Optional
from enum import Enum, IntEnum, auto
from dataclasses import dataclass
import os
from datetime import datetime
import operator

# sklearn
from sklearn.model_selection import train_test_split

# tensor flow
import tensorflow.keras as keras
from tensorflow.keras.layers import Dense
from tensorflow.keras.wrappers.scikit_learn import KerasRegressor

## SYS HELPER ##
def mkdir(directory):
    if not os.path.exists(directory):
        os.makedirs(directory)

class P3_Env:
    _train_data_x: List[float]  = None
    _train_data_y: List[float]  = None
    _test_data_x: List[float]   = None
    _test_data_y: List[float]   = None

    @staticmethod
    def print(content: str):
        print("[ P3_Env ] > {}".format(content))

    def __init__(
        self,
        f_data_function,
        x_range: List[float],
        env_name: str,
        # common configuration
        data_pts_i: List[int],
        hidden_nodes_j: List[int],
        N_eval_per_model: int,
        MAX_DATA_SIZE: int,
        TRAIN_SIZE: float
    )->None:
        # create folders
        mkdir("fig")
        mkdir("fig/p3")
        mkdir("fig/p3/{}".format(env_name))

        # store configs
        self._f_data_function = f_data_function
        self._x_range = x_range
        self._data_pts_i = data_pts_i
        self._hidden_nodes_j = hidden_nodes_j
        self._N_eval_per_model = N_eval_per_model
        self._MAX_DATA_SIZE = MAX_DATA_SIZE
        self._TRAIN_SIZE = TRAIN_SIZE
        self._env_name = env_name

        # generate model:
        self._generate_mlp_models()

        # generate data:
        self._generate_data_set()

        # report
        self.print("Data Size: [ Train: {train} | Test: {test} ] #Models: {n_model}"\
```

```
            .format(train=np.shape(self._train_data_x), test=np.shape(self._test_data_x), n_model=len(
    self._dict_of_mlps)))

    def _generate_data_set(self):
        # generate data
        data_x = np.random.uniform(self._x_range[0], self._x_range[1], self._MAX_DATA_SIZE)
        data_y = self._f_data_function(data_x)

        # split train and test data
        self._train_data_x, self._test_data_x, self._train_data_y, self._test_data_y = \
            train_test_split(data_x, data_y, train_size=self._TRAIN_SIZE, shuffle=True)

    def _generate_mlp_models(self):
        self._dict_of_mlps = {}
        for n_pts in self._data_pts_i:
            for n_nodes in self._hidden_nodes_j:
                tag = "i={}-j={}".format(n_pts, n_nodes)
                self._dict_of_mlps[tag] = {
                    "n_pts": n_pts,
                    "n_nodes": n_nodes,
                    "avg_training_errors": [],
                    "avg_validation_errors": [],
                    "lowest_training_error": 1.0,
                    "lowest_validation_error": 1.0,
                }

    def plot_progress(
        self,
        hs,
        tag:str,
    ):
        plt.figure()
        fig2 = plt.gcf()
        for h in hs:
            plt.plot(np.log10(h.history['loss']), 'b')
            if "val_loss" in h.history:
                plt.plot(np.log10(h.history['val_loss']), 'r')
                plt.ylabel("Loss")
            else:
                plt.ylabel("Training Loss")
            plt.xlabel("epoch")
        if 'val_loss' in h.history:
            plt.legend(["Training", "Validation"])
        fig2.savefig("fig/p3/{env}/train_loss_{tag}.png".format(env=self._env_name,tag=tag), bbox_inches
     = 'tight')
        plt.close(fig2)

    def plot_fitness_result(self, mlp, tag, mlp_index):
        plt.figure()
        fig2 = plt.gcf()

        if mlp_index is not None:
            instance = self._dict_of_mlps[mlp_index]
            tx = self._test_data_x[0:instance["n_pts"]]
            ty = self._test_data_y[0:instance["n_pts"]]
            C = sorted(zip(tx,ty), key=operator.itemgetter(0))
            new_x, new_y = zip(*C)
            x = new_x
            y_true = new_y
        else:
            x = np.linspace(self._x_range[0], self._x_range[1], num=1000)
            y_true = self._f_data_function(x)

        y_pred = mlp.predict(x)
        plt.plot(x, y_true, 'b')
        plt.plot(x, y_pred, 'r')
        plt.ylabel("y")
        plt.xlabel("x")
        plt.legend(["Ground Truth", "Prediction"])
        fig2.savefig("fig/p3/{env}/final_fit_{tag}.png".format(env=self._env_name,tag=tag), bbox_inches
    = 'tight')
        plt.close(fig2)
```

```

    def plot_and_print_loss_matrix(self):
        entries = ["lowest_training_error","lowest_validation_error"]
        for topic in entries:
            mat = np.zeros((len(self._data_pts_i), len(self._hidden_nodes_j)))
            # extract result & matrify the result
            for i, n_pts in enumerate(self._data_pts_i):
                for j, n_nodes in enumerate(self._hidden_nodes_j):
                    tag = "i={}-j={}".format(n_pts, n_nodes)
                    mat[i][j] = self._dict_of_mlps[tag][topic]
            # print result
            print("== Matrix {} ===".format(topic))
            print(mat)
            # plot result
            plt.figure()
            fig2 = plt.gcf()
            plt.imshow(mat)
            plt.yticks(list(range(len(self._data_pts_i))), self._data_pts_i)
            plt.xticks(list(range(len(self._hidden_nodes_j))), self._hidden_nodes_j)
            plt.ylabel("i (Data Points)")
            plt.xlabel("j (Number of Neurons)")
            plt.colorbar()
            file_name = "fig/p3/{env}/matrix_{tag}.png".format(env=self._env_name, tag=topic)
            print(file_name)
            fig2.savefig(file_name, bbox_inches = 'tight')
            plt.close(fig2)

    def train_at(
        self,
        mlp_index: str,
        N_epoch: int, # Early stopping
    ):
        # train only one instance
        instance = self._dict_of_mlps[mlp_index]
        # reset session:
        keras.backend.clear_session()
        # build mlp
        mlp = keras.models.Sequential([
            Dense(instance["n_nodes"], activation='sigmoid', input_shape=(1,),
                kernel_initializer=keras.initializers.RandomNormal(mean=0., stddev=30),
                bias_initializer=keras.initializers.RandomNormal(mean=0., stddev=10)
            ),
            Dense(1, activation='linear')
        ])
        mlp.compile(loss='mean_squared_error', optimizer='adam')
        # train the model fully
        h = mlp.fit(
            self._train_data_x[0:instance["n_pts"]],# n_pts training
            self._train_data_y[0:instance["n_pts"]],# n_pts training
            epochs=N_epoch,
            batch_size=1,
            verbose=0
        )
        # save model
        mlp.save('fig/p3/{}/best_mlp'.format(self._env_name))
        return h, mlp

    def evaluate_test_data(
        self,
        mlp_index: str,
        mlp
    ):
        instance = self._dict_of_mlps[mlp_index]
        tx = self._test_data_x[0:instance["n_pts"]]
        ty = self._test_data_y[0:instance["n_pts"]]
        result = mlp.evaluate(tx, ty)
        return result


    def run_hyperParam_optimization(
        self,
        callback_termination,
```

```
        k_fold:int,
        N_epoch:int, # Early stopping
        N_trial:int,
        plot:bool
    ):
        min_key = None
        ind = 0
        tot = N_trial * len(self._dict_of_mlps)
        for tag, instance in self._dict_of_mlps.items():
            # data selection
            training_pair = list(zip(self._train_data_x, self._train_data_y))
            # down sample to limited number of data for training
            training_pair = training_pair[0:instance["n_pts"]]

            ### t-TRIAL ========== ========== ==========
            for t in range(N_trial):
                ind += 1
                print("==== TEST [{tag:10s}] : Trial [{t}/{nt}] : [{ind}/{tot}]====".format(
                    tag=tag, t=(t+1), nt=N_trial, ind=ind, tot=tot
                ))

                # data shuffling per trial
                np.random.shuffle(training_pair)

                ### K-FOLD  ========== ========== ==========
                # divide data into k-portions
                data_pool = np.array_split(training_pair, k_fold)

                ### MLP     ========== ========== ==========
                # create storage:
                subfold_memory = {
                    "training_error"      : np.zeros(k_fold),
                    "validation_error"    : np.zeros(k_fold),
                    "training_history"    : [],
                }

                ## apply k-fold    ========== ==========
                for kf in range(k_fold):
                    # reset session:
                    keras.backend.clear_session()
                    # build mlp
                    mlp = keras.models.Sequential([
                        Dense(instance["n_nodes"], activation='sigmoid', input_shape=(1,),
                            kernel_initializer=keras.initializers.RandomNormal(mean=0., stddev=30),
                            bias_initializer=keras.initializers.RandomNormal(mean=0., stddev=10)
                        ),
                        Dense(1, activation='linear')
                    ])
                    mlp.compile(loss='mean_squared_error', optimizer='adam')
                    # construct training set
                    train_set = np.concatenate((data_pool[0:kf] + data_pool[kf+1:k_fold]), axis=0)
                    # construct validation set
                    valid_set = data_pool[kf]
                    # decode training data
                    tx,ty = zip(*train_set)
                    tx,ty = np.array(tx), np.array(ty)
                    # decode validation data
                    vx,vy = zip(*valid_set)
                    vx,vy = np.array(vx), np.array(vy)
                    # train the model
                    h = mlp.fit(
                        tx,
                        ty,
                        epochs=N_epoch,
                        batch_size=1,
                        verbose=0,
                        validation_data=(vx, vy),
                        callbacks = [callback_termination]
                    )
                    # store:
                    if plot:
                        subfold_memory["training_history"   ].append(h)
```

```
                    subfold_memory["training_error"     ] = h.history['loss'][-1]
                    subfold_memory["validation_error"   ] = h.history['val_loss'][-1]

                # create plots
                if plot:
                    self.plot_progress(hs=subfold_memory["training_history"  ], tag=tag)
                ## compute final average losses
                instance["avg_training_errors"     ].append(np.average(subfold_memory["training_error"]))
                instance["avg_validation_errors"   ].append(np.average(subfold_memory["validation_error"]))

            # capture the best out of N_trial
            instance["lowest_training_error"]   = min(instance["avg_training_errors"])
            instance["lowest_validation_error"] = min(instance["avg_validation_errors"])

            if (min_key is None) or (self._dict_of_mlps[min_key]["lowest_validation_error"] > instance["lowest_validation_error"]):
                min_key = tag

        return min_key, self._dict_of_mlps[min_key]


def main_env(
    f_func,
    tag,
    x_range,
    plot_progress           :bool,
    min_key                 :Optional[str], # None: 3.activation
    N_EPOCH_HARD_STOP       :int,
):

    ## INIT Environment Engine
    env = P3_Env(
        f_data_function     = f_func,
        x_range             = x_range,
        data_pts_i          = [10,40,80,200],
        hidden_nodes_j      = [2,10,40,100],
        env_name            = tag,
        N_eval_per_model    = 5, # repeat the process 5 times by shuffling the data generated randomly
        MAX_DATA_SIZE       = 500,
        TRAIN_SIZE          = 0.8 # 80 % for training by default
    )

    if min_key is None:
        # Run Engine
        val_err_callback = keras.callbacks.EarlyStopping(monitor='val_loss', baseline=0.001, patience=100, mode="min")
        min_key, min_instance = env.run_hyperParam_optimization(
            k_fold                  = 10,
            N_epoch                 = N_EPOCH_HARD_STOP,
            plot                    = plot_progress,
            callback_termination    = val_err_callback,
            N_trial                 = 5
        )

        ## Print Engine Result
        print("== SUMMARY ==")
        env.plot_and_print_loss_matrix()
        print(env._dict_of_mlps)

        print("== OPTIMAL MODEL ==")
        print(min_instance)

    print(min_key)
    ## Retrain the best model
    h, best_mlp_new = env.train_at(
        mlp_index=min_key,
        N_epoch=N_EPOCH_HARD_STOP
    )
    env.plot_fitness_result(mlp=best_mlp_new, tag="Best[Test Data]{}".format(min_key), mlp_index=min_key)
```

```
    env.plot_fitness_result(mlp=best_mlp_new, tag="Best[Overall]{}".format(min_key), mlp_index=None)
    env.plot_progress(hs=[h], tag="Best{}".format(min_key))
    ## hard evaluation:
    result = env.evaluate_test_data(mlp=best_mlp_new, mlp_index=min_key)
    print("Final Training Loss: {}".format(h.history['loss'][-1]))
    print("Final Test Loss: {}".format(result))

def main():
    ENABLE_P3_2 = True # Else RUN: P3-1 to generate hyperparam matrices
    ##
    if ENABLE_P3_2:
        min_key_f1 = "i=80-j=100"
        min_key_f2 = "i=40-j=40"
    else:
        min_key_f1 = None
        min_key_f2 = None

    # f1
    main_env(
        f_func              = (lambda x: x * np.sin(6 * np.pi * x) * np.exp(- x ** 2)),
        x_range             = [-1, 1],
        tag                 = "F1",
        plot_progress       = False,
        min_key             = min_key_f1,#None, # None : for auto-tuning
        N_EPOCH_HARD_STOP   = 1000,
    )
    # f2
    main_env(
        f_func              = (lambda x: np.exp(- x ** 2) * np.arctan(x) * np.sin(4 * np.pi * x)),
        x_range             = [-2, 2],
        tag                 = "F2",
        plot_progress       = False,
        min_key             = min_key_f2, # None : for auto-tuning
        N_EPOCH_HARD_STOP   = 1000,
    )

if __name__ == "__main__":
    start_time = datetime.now()
    main()
    end_time = datetime.now()
    print("=== END of P3 [ Time Elapse: {} ] ===".format(str(end_time-start_time)))
```

Code 11: MLP k-Fold Hyperparameter Optimization

Appendix D P4 - Code

```
# python typical
import numpy as np
import matplotlib.pyplot as plt
import copy
from typing import List, Dict, Optional
from enum import Enum, IntEnum, auto
from dataclasses import dataclass

# tensor flow
import tensorflow.keras as keras
from tensorflow.keras.layers import Dense

# python debugger
from icecream import ic # Debugger

#####################
### DATA LABEL DEF ###
#####################
class Constituents(IntEnum):
    Ethanol                         = 1
    Malic_acid                      = 2
    Ash                             = 3
    Alcalinity_of_ash               = 4
    Magnesium                       = 5
    Total_phenols                   = 6
```

```
    Flavanoids                    = 7
    Nonflavanoid_phenols          = 8
    Proanthocyanins               = 9
    Color_intensity               = 10
    Hue                           = 11
    OD280_OD315_of_diluted_liquid = 12
    Proline                       = 13

class Product(IntEnum):
    P1 = 1
    P2 = 2
    P3 = 3

class DataType(Enum):
    TRAIN_X = "train_x"
    TRAIN_Y = "train_y"
    TEST_X = "test_x"
    TEST_Y = "test_y"


class P4_ENV:
    def __init__(
        self,
        DATA: List[List[float]],
        training_percent: float = 0.75, # 75% of data used for training
    ):
        self._data_extraction_and_preparation(
            DATA=DATA,
            Eta_train=training_percent
        )

    def _data_extraction_and_preparation(
        self,
        DATA: List[List[float]],
        Eta_train: float,
    ):
        # find normalization bound
        DATA_MIN_MAX = []
        for entry in Constituents:
            DATA_MIN_MAX.append([min(DATA[:, entry]), max(DATA[:, entry])])
        DATA_MIN_MAX = np.array(DATA_MIN_MAX)

        # categorize by product class
        DATA_SET = {
            DataType.TRAIN_X  : [],
            DataType.TRAIN_Y  : [],
            DataType.TEST_X   : [],
            DataType.TEST_Y   : []
        }

        for prod in Product:
            # dataset categorization
            data = (DATA[DATA[:,0] == prod, 1:(len(Constituents)+1)])
            # dataset normalization
            data_norm = (data - DATA_MIN_MAX[:,0])/(DATA_MIN_MAX[:,1] - DATA_MIN_MAX[:,0])
            # dataset divide
            n, m = np.shape(data_norm)
            n_train = round(n * Eta_train)
            # gen labels [ k , # Products ]
            Y_label = np.zeros((n, len(Product)))
            Y_label[:, prod - 1] = 1
            # store inputs
            DATA_SET[DataType.TRAIN_X].append( data_norm[0:n_train , :] )
            DATA_SET[DataType.TEST_X ].append( data_norm[n_train:n , :] )
            # store label
            DATA_SET[DataType.TRAIN_Y].append( Y_label[0:n_train , :] )
            DATA_SET[DataType.TEST_Y ].append( Y_label[n_train:n , :] )

        DATA_SET[DataType.TRAIN_X] = np.concatenate(DATA_SET[DataType.TRAIN_X])
        DATA_SET[DataType.TRAIN_Y] = np.concatenate(DATA_SET[DataType.TRAIN_Y])
        DATA_SET[DataType.TEST_X ] = np.concatenate(DATA_SET[DataType.TEST_X ])
        DATA_SET[DataType.TEST_Y ] = np.concatenate(DATA_SET[DataType.TEST_Y ])
```

```python

        # store
        self.DATA_MIN_MAX = DATA_MIN_MAX
        self.DATA_SET = DATA_SET

        # check data shape
        ic(np.shape(self.DATA_SET[DataType.TRAIN_X]))
        ic(np.shape(self.DATA_SET[DataType.TRAIN_Y]))
        ic(np.shape(self.DATA_SET[DataType.TEST_X]))
        ic(np.shape(self.DATA_SET[DataType.TEST_Y]))

    def normalize_and_predict(
        self,
        mlp,
        data_x
    ):
        data_x = np.array(data_x)
        data_norm = (data_x - self.DATA_MIN_MAX[:,0])/(self.DATA_MIN_MAX[:,1] - self.DATA_MIN_MAX[:,0])
        print("Normalized: {}".format(data_norm))
        predict_array = mlp.predict([data_norm])
        predict_label = predict_array.argmax() + 1
        return predict_array, predict_label

    def train_all_mlps(
        self,
        dict_of_mlp,
    ):
        """ P4.1 Hyper Tuning Process
        """
        best_instance = None
        best_tag = None
        for tag, instance in dict_of_mlp.items():
            print("==== TEST [{tag:10s}] ====".format(tag=tag))
            instance["mlp"].compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
            h = instance["mlp"].fit(
                self.DATA_SET[DataType.TRAIN_X],
                self.DATA_SET[DataType.TRAIN_Y],
                epochs=instance["max_epoch"],
                batch_size=20,
                verbose=0
            )
            train_accuracy = self.plot_progress(h=h, tag=tag)
            test_accuracy = self.validate(self.DATA_SET[DataType.TEST_X], self.DATA_SET[DataType.TEST_Y], mlp=instance["mlp"])

            # record back the result
            instance["train_accuracy"] = train_accuracy
            instance["test_accuracy"] = test_accuracy

            # record best
            if best_instance is None:
                best_instance = instance
                best_tag = tag
            elif best_instance["test_accuracy"] < test_accuracy:
                best_instance = instance
                best_tag = tag

        return best_instance, best_tag

    def test_run(
        self
    ):
        """ Implementation Validation Code
        """
        mlp = keras.models.Sequential([
            Dense(10, activation='sigmoid', input_shape=(13,)),
            Dense(20, activation='sigmoid'),
            Dense(3, activation='softmax')
        ])

        print(mlp.summary())
```

```
        mlp.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])

        h = mlp.fit(
            self.DATA_SET[DataType.TRAIN_X],
            self.DATA_SET[DataType.TRAIN_Y],
            epochs=250,
            batch_size=20,
            verbose=0
        )

        self.plot_progress(h=h, tag="test")
        self.validate(self.DATA_SET[DataType.TEST_X], self.DATA_SET[DataType.TEST_Y], mlp=mlp)

    def validate(
        self,
        test_x,
        test_y,
        mlp
    ):
        test_accuracy = 100*mlp.evaluate(test_x, test_y, verbose=0)[1]
        print('Test accuracy: {test_acc:.2f} %'.format(test_acc=test_accuracy))
        return test_accuracy


    def plot_progress(
        self,
        h,
        tag:str
    ):
        train_acc = h.history['accuracy'][-1]*100
        print('Train accuracy: {train_acc:.2f} %'.format(train_acc=train_acc))
        # Plot
        plt.figure()
        plt.plot(h.history['accuracy'])
        plt.ylabel("training accuracy")
        plt.xlabel("epoch")
        fig2 = plt.gcf()
        fig2.savefig("fig/p4/train_accu_{tag}.png".format(tag=tag), bbox_inches = 'tight')

        plt.figure()
        plt.plot(np.log10(h.history['loss']))
        plt.ylabel("training loss")
        plt.xlabel("epoch")

        fig2 = plt.gcf()
        fig2.savefig("fig/p4/train_loss_{tag}.png".format(tag=tag), bbox_inches = 'tight')
        return train_acc

def main():
    ### IMPORT DATA ###
    DATA = np.loadtxt(open("randomized_data.txt"), delimiter=",")
    env = P4_ENV(DATA=DATA)

    # construct mlp test models
    MLP_DICT = {
        "t1": {
            "mlp": keras.models.Sequential([
                Dense(10, activation='sigmoid', input_shape=(13,)),
                Dense(20, activation='sigmoid'),
                Dense(3, activation='softmax')
            ]),
            "max_epoch": 250,
        },
        "t2": {
            "mlp": keras.models.Sequential([
                Dense(10, activation='sigmoid', input_shape=(13,)),
                Dense(20, activation='sigmoid'),
                Dense(20, activation='sigmoid'),
                Dense(3, activation='softmax')
            ]),
            "max_epoch": 250,
        },
```

```
        "t3": {
            "mlp": keras.models.Sequential([
                Dense(10, activation='sigmoid', input_shape=(13,)),
                Dense(20, activation='sigmoid'),
                Dense(20, activation='sigmoid'),
                Dense(20, activation='sigmoid'),
                Dense(3, activation='softmax')
            ]),
            "max_epoch": 250,
        },
        "t4": {
            "mlp": keras.models.Sequential([
                Dense(5, activation='sigmoid', input_shape=(13,)),
                Dense(5, activation='sigmoid'),
                Dense(3, activation='softmax')
            ]),
            "max_epoch": 250,
        },
        "t5": {
            "mlp": keras.models.Sequential([
                Dense(5, activation='sigmoid', input_shape=(13,)),
                Dense(5, activation='sigmoid'),
                Dense(5, activation='sigmoid'),
                Dense(3, activation='softmax')
            ]),
            "max_epoch": 250,
        },
        "t6": {
            "mlp": keras.models.Sequential([
                Dense(5, activation='sigmoid', input_shape=(13,)),
                Dense(15, activation='sigmoid'),
                Dense(15, activation='sigmoid'),
                Dense(3, activation='softmax')
            ]),
            "max_epoch": 250,
        }
    }

    # Perform Training
    print("\n=== P4.1 ===")
    best_mlp, best_tag = env.train_all_mlps(dict_of_mlp=MLP_DICT)

    # print result
    print("\n=== Summary ===")
    print(MLP_DICT)

    print("\n=== BEST ===")
    print(best_tag)
    print(best_mlp)

    # Perform 4.2 evaluation
    print("\n=== P4.2 ===")
    TEST_DATA = {   "test_a": [13.72, 1.43, 2.5, 16.7, 108, 3.4, 3.67, 0.19, 2.04, 6.8, 0.89, 2.87, 1285],
                    "test_b": [12.04, 4.3, 2.38, 22, 80, 2.1, 1.75, 0.42, 1.35, 2.6, 0.79, 2.57, 580],
                    "test_c": [14.13, 4.1, 2.74, 24.5, 96, 2.05, 0.76, 0.56, 1.35, 9.2, 0.61, 1.6, 560]
        }
    for tag, data in TEST_DATA.items():
        print(data)
        predict_array, predict_class = env.normalize_and_predict(mlp=best_mlp["mlp"], data_x=[data])
        print("[{tag:10s}]: Predicted ranking array: {parray}, Classified as: {pclass}".format(tag=tag, parray=predict_array, pclass=predict_class))

if __name__ == "__main__":
    main()
```

Code 12: MLP Multi-class Classifier