In [185…
```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
import time
```

1) Regression

Implement the neural network for regression by using the energy efficiency dataset. There are 2 simulation energy loads and 8 different features in this dataset. Shuffle the dataset then use 75% of data samples for training and 25% for testing. Note that for the categorical features (orientation, glazing area distribution), you need to encode them into one-hotvectors.

(a) Please try to predict the heating load of buildings by minimizing the sum-of-squares error function.

Import "energy_efficiency_data.csv" dataset

In [186…
```python
data1_raw = pd.read_csv("D:\研究所課程\深度\DL_HW1\energy_efficiency_data.csv")
data1_raw
```

Out[186]:

	# Relative Compactness	Surface Area	Wall Area	Roof Area	Overall Height	Orientation	Glazing Area	Glazing Area Distribution	Heating Load	Cooling Load
0	0.98	514.5	294.0	110.25	7.0	2.0	0.0	0.0	15.55	21.33
1	0.98	514.5	294.0	110.25	7.0	4.0	0.0	0.0	15.55	21.33
2	0.98	514.5	294.0	110.25	7.0	5.0	0.0	0.0	15.55	21.33
3	0.90	563.5	318.5	122.50	7.0	2.0	0.0	0.0	20.84	28.28
4	0.90	563.5	318.5	122.50	7.0	3.0	0.0	0.0	21.46	25.38
...	...	...	...	...	...	...	...	...	...	...
763	0.71	710.5	269.5	220.50	3.5	2.0	0.4	5.0	12.43	15.59
764	0.69	735.0	294.0	220.50	3.5	3.0	0.4	5.0	14.28	15.87
765	0.66	759.5	318.5	220.50	3.5	4.0	0.4	5.0	14.92	17.55
766	0.64	784.0	343.0	220.50	3.5	3.0	0.4	5.0	18.19	20.21
767	0.62	808.5	367.5	220.50	3.5	4.0	0.4	5.0	16.48	16.61

768 rows × 10 columns

We encode categorical features(orientation, glazing area distribution) into one-hotvectors.

In [187…
```python
data1 = pd.get_dummies(data1_raw, columns=['Orientation', 'Glazing Area Distribution'])
```

In [188…
```python
data1
```

Out[188]:

	# Relative Compactness	Surface Area	Wall Area	Roof Area	Overall Height	Glazing Area	Heating Load	Cooling Load	Orientation_2.0	Orientation_3.0	Orientation_4.0
0	0.98	514.5	294.0	110.25	7.0	0.0	15.55	21.33	1	0	0
1	0.98	514.5	294.0	110.25	7.0	0.0	15.55	21.33	0	0	1
2	0.98	514.5	294.0	110.25	7.0	0.0	15.55	21.33	0	0	0
3	0.90	563.5	318.5	122.50	7.0	0.0	20.84	28.28	1	0	0
4	0.90	563.5	318.5	122.50	7.0	0.0	21.46	25.38	0	1	0
...	...	...	...	...	...	...	...	...	...	...	...
763	0.71	710.5	269.5	220.50	3.5	0.4	12.43	15.59	1	0	0
764	0.69	735.0	294.0	220.50	3.5	0.4	14.28	15.87	0	1	0
765	0.66	759.5	318.5	220.50	3.5	0.4	14.92	17.55	0	0	1
766	0.64	784.0	343.0	220.50	3.5	0.4	18.19	20.21	0	1	0
767	0.62	808.5	367.5	220.50	3.5	0.4	16.48	16.61	0	0	1

768 rows × 18 columns

Split the data to X and Y

In [189…

```
# Load data
x = data1.drop(["Heating Load"], axis = 1)
x= x.drop(["Orientation_5.0"], axis = 1) #drop baseline column
x=x.drop(["Glazing Area Distribution_5.0"], axis = 1)
y = pd.DataFrame(data1.iloc[:, 6])
x
```

Out[189]:

	# Relative Compactness	Surface Area	Wall Area	Roof Area	Overall Height	Glazing Area	Cooling Load	Orientation_2.0	Orientation_3.0	Orientation_4.0	Glazir Distribut
0	0.98	514.5	294.0	110.25	7.0	0.0	21.33	1	0	0	
1	0.98	514.5	294.0	110.25	7.0	0.0	21.33	0	0	1	
2	0.98	514.5	294.0	110.25	7.0	0.0	21.33	0	0	0	
3	0.90	563.5	318.5	122.50	7.0	0.0	28.28	1	0	0	
4	0.90	563.5	318.5	122.50	7.0	0.0	25.38	0	1	0	
...	...	...	...	...	...	...	...	...	...	...	
763	0.71	710.5	269.5	220.50	3.5	0.4	15.59	1	0	0	
764	0.69	735.0	294.0	220.50	3.5	0.4	15.87	0	1	0	
765	0.66	759.5	318.5	220.50	3.5	0.4	17.55	0	0	1	
766	0.64	784.0	343.0	220.50	3.5	0.4	20.21	0	1	0	
767	0.62	808.5	367.5	220.50	3.5	0.4	16.61	0	0	1	

768 rows × 15 columns

In [190…

```
y
```

Out[190]:

	Heating Load
0	15.55
1	15.55
2	15.55
3	20.84
4	21.46
...	...
763	12.43
764	14.28
765	14.92
766	18.19
767	16.48

768 rows × 1 columns

Shuffle the dataset then use 75% of data samples for training and 25% for testing.

In [191…

```
# Split, reshape, shuffle
x_train, x_test = train_test_split(x, random_state=111024520, train_size=0.75)
y_train, y_test = train_test_split(y, random_state=111024520, train_size=0.75)
```

Normalize the continuous variables of X_{train}, X_{test} except the first column, because its values are either 0 or 1.

$$normalize(X) = \frac{X - mean(X_{train})}{sd(X_{train})}$$

In [192…

```
#Normalize
mean_train = np.mean(x_train, axis = 0)
sd_train = np.std(x_train, axis = 0)

for i in range(7):
    x_train.iloc[:,i] = (x_train.iloc[:,i]-mean_train[i]) / (sd_train[i])
    x_test.iloc[:,i] = (x_test.iloc[:,i]-mean_train[i]) / (sd_train[i])

#轉成array
x_train, x_test =x_train.values, x_test.values
y_train, y_test =y_train.values, y_test.values
print("Training data: X={}, Y={}".format(x_train.shape, y_train.shape))
print("Test data: X={}, Y={}".format(x_test.shape, y_test.shape))
```

```
Training data: X=(576, 15), Y=(576, 1)
Test data: X=(192, 15), Y=(192, 1)
```

In [193…

```
class DeepNeuralNetwork_reg():
    def __init__(self, sizes, activation='relu'):
        self.sizes = sizes
        # Choose activation function
        if activation == 'relu':
            self.activation = self.relu
        # Save all weights
        self.params = self.initialize()
        # Save all intermediate values, i.e. activations
        self.cache = {}

    def relu(self, x, derivative=False):
        '''
            Derivative of ReLU is a bit more complicated since it is not differentiable at x = 0

            Forward path:
            relu(x) = max(0, x)
            In other word,
            relu(x) = 0, if x < 0
                    = x, if x >= 0

            Backward path:
```

```
        ∇relu(x) = 0, if x < 0
                 = 1, if x >=0
    '''
    if derivative:
        x = np.where(x < 0, 0, x)
        x = np.where(x >= 0, 1, x)
        return x
    else:
        return np.maximum(0, x)



def initialize(self):
    # number of nodes in each layer
    input_layer=self.sizes[0]
    hidden_layer1=self.sizes[1]
    hidden_layer2=self.sizes[2]
    output_layer=self.sizes[3]

    params = {
        "W1": np.random.randn(hidden_layer1, input_layer) * np.sqrt(1./input_layer),
        "b1": np.zeros((hidden_layer1, 1)),
        "W2": np.random.randn(hidden_layer2, hidden_layer1) * np.sqrt(1./hidden_layer1),
        "b2": np.zeros((hidden_layer2, 1)),
        "W3": np.random.randn(output_layer, hidden_layer2) * np.sqrt(1./hidden_layer2),
        "b3": np.zeros((output_layer, 1))
    }
    return params

def sum_of_square_loss(self, y, output,derivative=False):
    '''
        sse = ∑(y-ŷ)^2.
    '''
    if derivative:
        sse= 2*(output-y.T)
    else:
        sse = np.sum((output-y.T)**2)

    return sse

def feed_forward(self, x):
    '''
        y = σ(wX + b)
    '''
    self.cache["X"] = x
    self.cache["Z1"] = np.matmul(self.params["W1"], self.cache["X"].T) + self.params["b1"]
    self.cache["A1"] = self.activation(self.cache["Z1"])
    self.cache["Z2"] = np.matmul(self.params["W2"], self.cache["A1"]) + self.params["b2"]
    self.cache["A2"] = self.activation(self.cache["Z2"])
    self.cache["Z3"] = np.matmul(self.params["W3"], self.cache["A2"]) + self.params["b3"]
    self.cache["A3"] = self.cache["Z3"]

    return self.cache["A3"]

def back_propagate(self, y, output):
    '''
        This is the backpropagation algorithm, for calculating the updates
        of the neural network's parameters.

    '''
    current_batch_size = y.shape[0]

    dA3 = self.sum_of_square_loss(y, output, derivative = True) #(1, 576)
    dZ3 = dA3*1  # delta3

    dW3 =  (1./current_batch_size) * np.matmul(dZ3, self.cache["A2"].T)
    db3 =  (1./current_batch_size) *np.sum(dZ3, axis=1, keepdims=True)

    dA2 = np.matmul(self.params["W3"].T, dZ3)
    dZ2 = dA2 * self.activation(self.cache["Z2"], derivative=True) #delta2
    dW2 =  (1./current_batch_size) *np.matmul(dZ2, self.cache["A1"].T)
    db2 =  (1./current_batch_size) *np.sum(dZ2, axis=1, keepdims=True)

    dA1 = np.matmul(self.params["W2"].T, dZ2)
    dZ1 = dA1 * self.activation(self.cache["Z1"], derivative=True)
    dW1 =  (1./current_batch_size) *np.matmul(dZ1, self.cache["X"])
    db1 =  (1./current_batch_size) *np.sum(dZ1, axis=1, keepdims=True)

    self.grads = {"W1": dW1, "b1": db1, "W2": dW2, "b2": db2,"W3": dW3, "b3": db3}
```

```python
        return self.grads



    def optimize(self, l_rate):
        '''
            Stochatic Gradient Descent (SGD):
            θ^(t+1) <- θ^t - η∇L(y, ŷ)
        '''
        if self.optimizer == "sgd":
            for key in self.params:
                self.params[key] = self.params[key] - l_rate * self.grads[key]
        else:
            raise ValueError("Optimizer is currently not support, please use 'sgd' instead.")


    def train(self, x_train, y_train, x_test, y_test, epochs, batch_size, optimizer='sgd', l_rate=0.05, show

        self.train_loss = []
        self.test_loss = []
        # Hyperparameters
        self.epochs = epochs
        self.batch_size = batch_size
        num_batches = -(-x_train.shape[0] // self.batch_size)

        # Initialize optimizer
        self.optimizer = optimizer
        if self.optimizer == 'sgd':
            self.params = self.initialize()

        start_time = time.time()
        template = "Epoch {}: {:.2f}s, train loss={:.2f}, test loss={:.2f}"

        # Train
        np.random.seed(111024520)
        for i in range(self.epochs):
            # Shuffle
            permutation = np.random.permutation(x_train.shape[0])
            x_train_shuffled = x_train[permutation]
            y_train_shuffled = y_train[permutation]

            for j in range(num_batches):
                # Batch
                begin = j * self.batch_size
                end = min(begin + self.batch_size, x_train.shape[0])
                x = x_train_shuffled[begin:end]
                y = y_train_shuffled[begin:end]

                # Forward
                output = self.feed_forward(x)
                # Backprop
                grad = self.back_propagate(y, output)
                # Optimize
                self.optimize(l_rate=l_rate)

            # Evaluate performance
            # Training data
            output = self.feed_forward(x_train)
            train_loss = self.sum_of_square_loss(y_train, output) #sse
            self.train_loss.append(train_loss)

            # Test data
            output = self.feed_forward(x_test)
            test_loss = self.sum_of_square_loss(y_test, output) #sse
            self.test_loss.append(test_loss)
            if show:
                if (i+1) % 500 == 0:
                    print(template.format(i+1, time.time()-start_time, train_loss, test_loss))
```

We set epochs= 15000, batch_size= 20, optimizer='sgd', l_rate= 0.00001, and print the result every 500 epochs.

The output of loss is the value of sum-of-squares error (SSE).

In [194…

```python
dnn = DeepNeuralNetwork_reg(sizes=[15,10,5,1], activation='relu')
np.random.seed(111024520)
dnn.train(x_train, y_train, x_test, y_test, epochs=15000,batch_size=20, optimizer='sgd', l_rate=0.00001)
```

```
Epoch 500: 1.94s, train loss=2740.44, test loss=882.35
Epoch 1000: 3.68s, train loss=1990.00, test loss=628.86
Epoch 1500: 5.66s, train loss=1781.50, test loss=561.08
Epoch 2000: 7.55s, train loss=1677.06, test loss=527.40
Epoch 2500: 9.56s, train loss=1621.87, test loss=510.26
Epoch 3000: 11.76s, train loss=1592.09, test loss=501.83
Epoch 3500: 13.69s, train loss=1575.31, test loss=497.50
Epoch 4000: 15.54s, train loss=1565.22, test loss=494.95
Epoch 4500: 17.42s, train loss=1558.61, test loss=493.43
Epoch 5000: 19.25s, train loss=1553.77, test loss=492.25
Epoch 5500: 21.08s, train loss=1549.91, test loss=491.04
Epoch 6000: 23.04s, train loss=1546.62, test loss=490.31
Epoch 6500: 24.83s, train loss=1543.69, test loss=489.45
Epoch 7000: 27.02s, train loss=1541.04, test loss=488.87
Epoch 7500: 29.42s, train loss=1538.56, test loss=488.01
Epoch 8000: 31.39s, train loss=1536.27, test loss=487.18
Epoch 8500: 33.35s, train loss=1534.15, test loss=486.59
Epoch 9000: 35.43s, train loss=1532.14, test loss=486.00
Epoch 9500: 37.58s, train loss=1530.26, test loss=485.48
Epoch 10000: 39.72s, train loss=1528.49, test loss=484.69
Epoch 10500: 42.04s, train loss=1526.82, test loss=484.39
Epoch 11000: 44.57s, train loss=1525.25, test loss=483.96
Epoch 11500: 47.06s, train loss=1523.78, test loss=483.41
Epoch 12000: 49.61s, train loss=1522.36, test loss=483.09
Epoch 12500: 52.05s, train loss=1521.03, test loss=482.68
Epoch 13000: 54.39s, train loss=1519.77, test loss=482.39
Epoch 13500: 56.43s, train loss=1518.59, test loss=482.01
Epoch 14000: 58.61s, train loss=1517.45, test loss=481.86
Epoch 14500: 60.98s, train loss=1516.38, test loss=481.54
Epoch 15000: 63.31s, train loss=1515.36, test loss=481.24
```

(b)

(1) network architecture (number of hidden layers and neurons)

In [195…

```python
df_1 =pd.DataFrame({'number of neurons': [17, 10, 5, 1],'activation function': ['', 'relu', 'relu', ''],
                    },index=['input layer', 'hidden layer1', 'hidden layer2','output'])
df_1
```

Out[195]:

	number of neurons	activation function
input layer	17	
hidden layer1	10	relu
hidden layer2	5	relu
output	1	

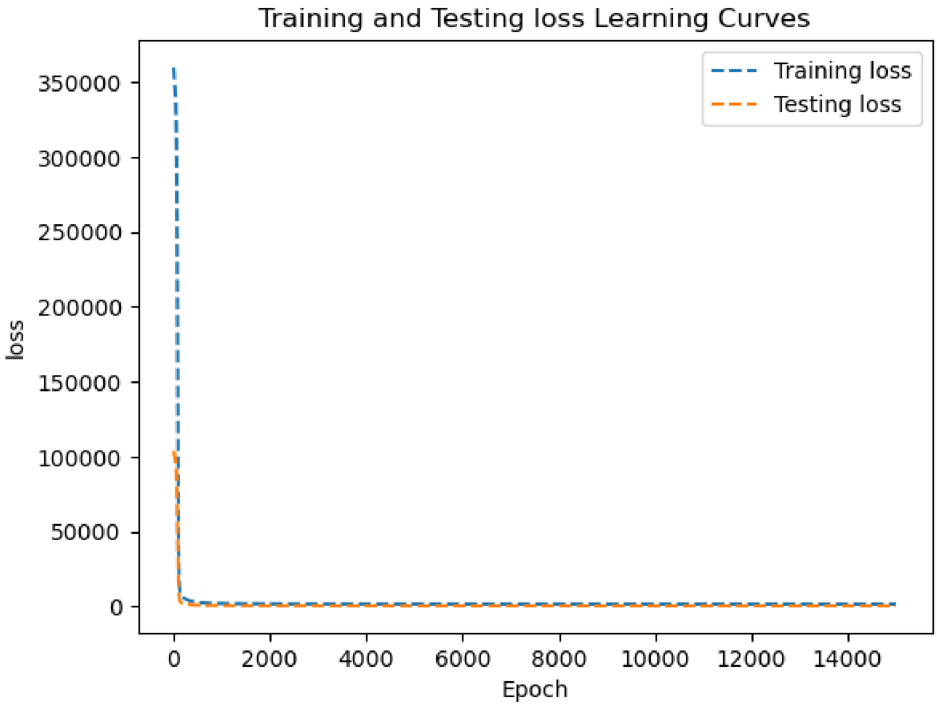
(2) learning curve

We draw the learning curve of loss function: sum-of-squares error function

$$SSE = \sum_{n=1}^{N}(y_{ntrue} - \hat{y_n})^2$$

In [196…

```python
epochs = list(range(1, len(dnn.train_loss) + 1))

# 绘制训练RMSE曲线
plt.plot(epochs, dnn.train_loss, label='Training loss', linestyle='--')

# 绘制测试RMSE曲线
plt.plot(epochs, dnn.test_loss, label='Testing loss',linestyle='--')

# 设置图例和标签
plt.legend()
plt.xlabel('Epoch')
plt.ylabel('loss')
plt.title('Training and Testing loss Learning Curves')

# 显示图形
plt.show()
```



(3) training RMS error

In [197...

```
print("Training RMSE: {} ".format(np.sqrt(dnn.train_loss[14999]/len(y_train))))
```

```
Training RMSE: 1.6219817892689972
```

(4) testing RMS error

In [198...

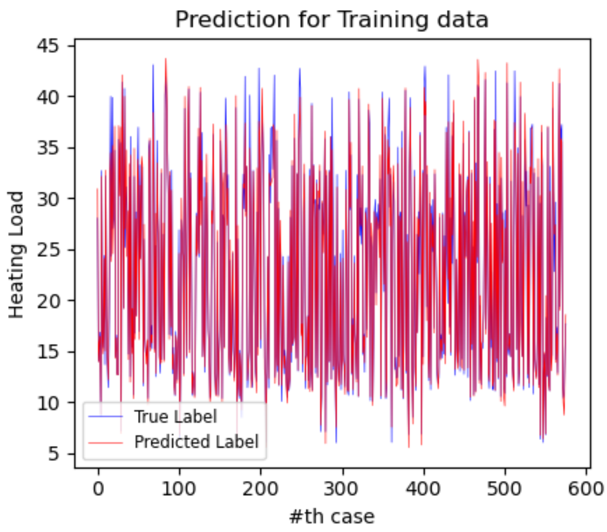
```
print("Testing RMSE: {} ".format(np.sqrt(dnn.test_loss[14999]/len(y_test))))
```

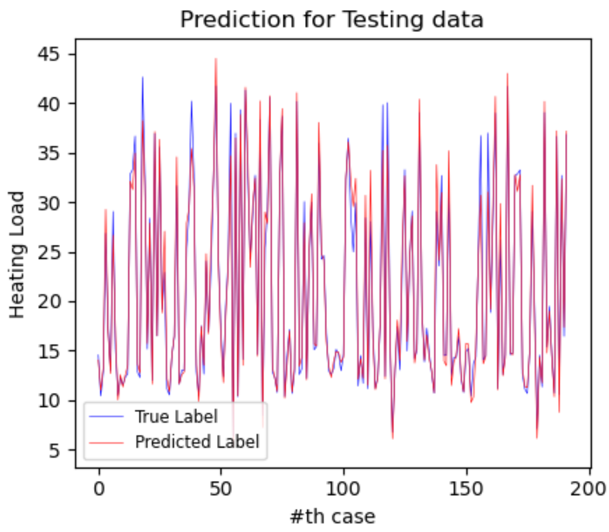
```
Testing RMSE: 1.5831853429682958
```

(5) regression result with training labels & test labels

In [199...

```
training_labels = y_train
predicted_labels = dnn.feed_forward(x_train).T

x = np.arange(len(training_labels))

plt.figure(figsize=(9, 4))
plt.subplot(1, 2, 1)
plt.plot(x, training_labels, color='blue', label='True Label', linewidth=0.4)
plt.plot(x, predicted_labels, color='red', label='Predicted Label', linewidth=0.4)
plt.xlabel('#th case')
plt.ylabel('Heating Load')
plt.legend(loc='lower left',fontsize='small')
plt.title('Prediction for Training data')

testing_labels = y_test
predicted_testing = dnn.feed_forward(x_test).T

x = np.arange(len(testing_labels))

plt.subplot(1, 2, 2)
plt.plot(x, testing_labels, color='blue', label='True Label', linewidth=0.4)
plt.plot(x, predicted_testing, color='red', label='Predicted Label', linewidth=0.4)
plt.xlabel('#th case')
plt.ylabel('Heating Load')
plt.legend(loc='lower left',fontsize='small')
plt.title('Prediction for Testing data')

plt.tight_layout()

# Show the plot
plt.show()
```





(c) Design a feature selection procedure to find out which input features influence the energy load significantly and explain why it works. You may compare the performance of choosing different features.

We use backward selection to start with all available features and iteratively removes one feature at a time, then evaluate the performance (RMSE) of the model after each removal.

Steps for feature selection using backward selection:

1. Start with all features.
2. Enter a loop and remove one feature from the feature list at a time.
3. Calculate the RMSE for each modified feature set.
4. Determine which feature's removal results in the smallest increase in RMSE.
5. Repeat steps 2-4 until only remain one feature.

This process selectively removes features until the optimal subset of features that significantly affect the model's performance is found.

In [201…

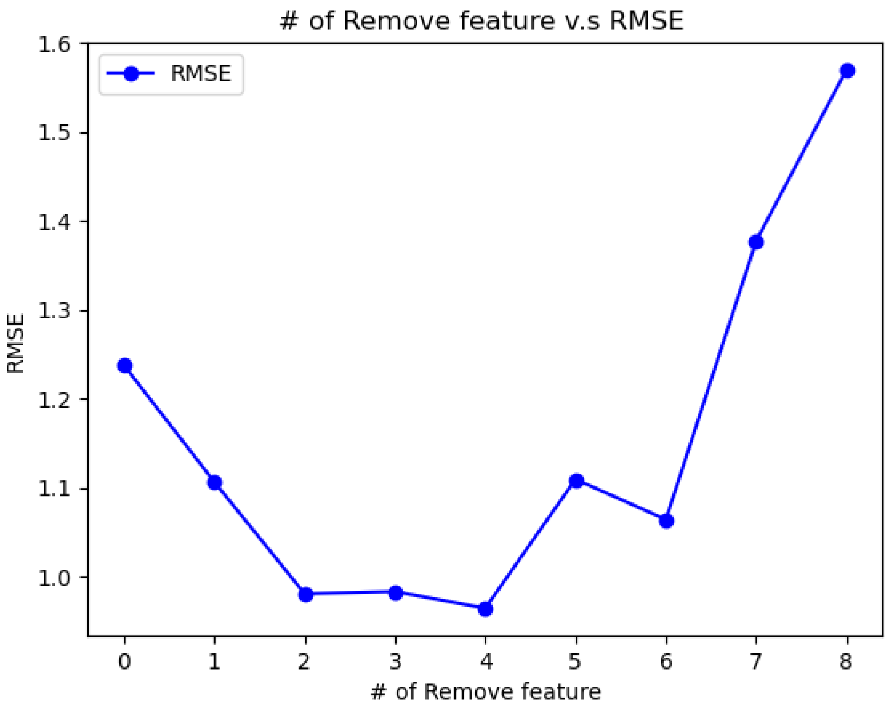
```
selected_features = np.array([[0],[1],[2],[3],[4],[5],[6],[7,8,9],[10,11,12,13,14]] ,dtype=object)
#data1.columns.tolist()  # 從所有特徵開始
colnames=['# Relative Compactness', 'Surface Area', 'Wall Area', 'Roof Area', 'Overall Height','Glazing Area
          'Orientation', 'Glazing Area Distribution']

remove_list = []
# 初始化所有特徵的基準 RMSE
dnn_1=DeepNeuralNetwork_reg(sizes=[15,10,5,1], activation='relu')
np.random.seed(111024520)
dnn_1.train(x_train, y_train, x_test, y_test, epochs=500,batch_size=20, optimizer='sgd', l_rate=0.00001,show
baseline_rmse =np.sqrt(dnn_1.test_loss[499]/len(y_train))
print('All feature:','RMSE=',baseline_rmse)

# 初始化存儲RMSE的列表
rmse_values = [baseline_rmse]

while len(selected_features) > 1:
    rmses = []

    for k in range(len(selected_features)):
        # 刪除當前特徵的副本

        X_train_temp = np.delete(x_train,selected_features[k]+remove_list, axis=1)
        X_test_temp = np.delete(x_test, selected_features[k]+remove_list, axis=1)

        dnn_temp=DeepNeuralNetwork_reg(sizes=[X_train_temp.shape[1],10,5,1], activation='relu')
        np.random.seed(111024520)
        dnn_temp.train(X_train_temp, y_train, X_test_temp, y_test, epochs=500,batch_size=20, optimizer='sgd'
                       show=False)

        # 計算當前特徵組合的RMSE
        rmse =np.sqrt(dnn_temp.test_loss[499]/len(y_train))
        rmses.append(rmse)

    min_rmse_index = np.argmin(rmses)
```

```
    feature_to_remove = colnames[min_rmse_index]
    print('Remove feature:',feature_to_remove,',RMSE=',np.min(rmses))

    #names.append(colnames1[index])
    #colnames1 = np.delete(colnames1, index)
    remove_list = remove_list + selected_features[min_rmse_index]
    selected_features = np.delete(selected_features, min_rmse_index, 0)
    colnames=np.delete(colnames, min_rmse_index, 0)
    # 更新RMSE值列表
    rmse_values.append(np.min(rmses))
```

```
All feature: RMSE= 1.237684079059977
Remove feature: Orientation ,RMSE= 1.1064678311878857
Remove feature: # Relative Compactness ,RMSE= 0.9809013573468406
Remove feature: Surface Area ,RMSE= 0.9833206336188115
Remove feature: Roof Area ,RMSE= 0.9645689266031343
Remove feature: Glazing Area ,RMSE= 1.1094116973747612
Remove feature: Glazing Area Distribution ,RMSE= 1.0641900666953403
Remove feature: Wall Area ,RMSE= 1.3773809444614784
Remove feature: Overall Height ,RMSE= 1.56978656602934
```

In [184…

```
x_values = np.arange(9)

plt.plot(x_values, rmse_values, marker='o', linestyle='-', color='b', label='RMSE')

plt.xlabel('# of Remove feature')
plt.ylabel('RMSE')
plt.title('# of Remove feature v.s RMSE ')
plt.legend()
plt.show()
```



當移除變數個數=4 (Orientation, # Relative Compactness, Surface Area, Roof Area)時,Mean Squared Error (MSE) 達到最小。\ 也就是剩下： Wall Area, Overall Height, Glazing Area, Glazing Area Distribution, Cooling Load 此5個變數為important features.

2) Classification

Implement the neural network for binary classification by using the Ionosphere dataset. There are 34 different features and 2 classes. The last column represents their corresponding labels:“g” for good and “b” for bad. Use 80% of data samples for training and 20% for testing.

(a) Please try to classify the Ionosphere data by minimizing the cross-entropy error function.

In [202…

```
data2 = pd.read_csv("D:\研究所課程\深度\DL_HW1\ionosphere_data.csv",header=None)
data2
```

Out[202]:

	0	1	2	3	4	5	6	7	8	9	...	25	26	27	28
0	1	0	0.99539	-0.05889	0.85243	0.02306	0.83398	-0.37708	1.00000	0.03760	...	-0.51171	0.41078	-0.46168	0.21266
1	1	0	1.00000	-0.18829	0.93035	-0.36156	-0.10868	-0.93597	1.00000	-0.04549	...	-0.26569	-0.20468	-0.18401	-0.19040
2	1	0	1.00000	-0.03365	1.00000	0.00485	1.00000	-0.12062	0.88965	0.01198	...	-0.40220	0.58984	-0.22145	0.43100
3	1	0	1.00000	-0.45161	1.00000	1.00000	0.71216	-1.00000	0.00000	0.00000	...	0.90695	0.51613	1.00000	1.00000
4	1	0	1.00000	-0.02401	0.94140	0.06531	0.92106	-0.23255	0.77152	-0.16399	...	-0.65158	0.13290	-0.53206	0.02431
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
346	1	0	0.83508	0.08298	0.73739	-0.14706	0.84349	-0.05567	0.90441	-0.04622	...	-0.04202	0.83479	0.00123	1.00000
347	1	0	0.95113	0.00419	0.95183	-0.02723	0.93438	-0.01920	0.94590	0.01606	...	0.01361	0.93522	0.04925	0.93159
348	1	0	0.94701	-0.00034	0.93207	-0.03227	0.95177	-0.03431	0.95584	0.02446	...	0.03193	0.92489	0.02542	0.92120
349	1	0	0.90608	-0.01657	0.98122	-0.01989	0.95691	-0.03646	0.85746	0.00110	...	-0.02099	0.89147	-0.07760	0.82983
350	1	0	0.84710	0.13533	0.73638	-0.06151	0.87873	0.08260	0.88928	-0.09139	...	-0.15114	0.81147	-0.04822	0.78207

351 rows × 35 columns

We transform the labels of the last column: 'g' to '1' and 'b' to '0'.

In [6]:

```
label_mapping = {'g': 1, 'b': 0}
# Assuming y and output are loaded as strings or in a non-numeric format

data2.iloc[:, -1] = data2.iloc[:, -1].map(label_mapping)
```

```
C:\Users\李昕\AppData\Local\Temp\ipykernel_22052\3367824845.py:4: DeprecationWarning: In a future version, `
df.iloc[:, i] = newvals` will attempt to set the values inplace instead of always setting a new array. To r
etain the old behavior, use either `df[df.columns[i]] = newvals` or, if columns are non-unique, `df.isetite
m(i, newvals)`
  data2.iloc[:, -1] = data2.iloc[:, -1].map(label_mapping)
```

Split the data to X and Y

In [7]:

```
# Load data
x2 = data2.drop(1, axis=1)
x2 = x2.drop( 34 ,axis=1)
y2 = pd.DataFrame(data2.iloc[:, 34])
```

In [8]:

```
x2
```

Out[8]:

	0	2	3	4	5	6	7	8	9	10	...	24	25	26	
0	1	0.99539	-0.05889	0.85243	0.02306	0.83398	-0.37708	1.00000	0.03760	0.85243	...	0.56811	-0.51171	0.41078	-0.4
1	1	1.00000	-0.18829	0.93035	-0.36156	-0.10868	-0.93597	1.00000	-0.04549	0.50874	...	-0.20332	-0.26569	-0.20468	-0.1
2	1	1.00000	-0.03365	1.00000	0.00485	1.00000	-0.12062	0.88965	0.01198	0.73082	...	0.57528	-0.40220	0.58984	-0.2
3	1	1.00000	-0.45161	1.00000	1.00000	0.71216	-1.00000	0.00000	0.00000	0.00000	...	1.00000	0.90695	0.51613	1.0
4	1	1.00000	-0.02401	0.94140	0.06531	0.92106	-0.23255	0.77152	-0.16399	0.52798	...	0.03286	-0.65158	0.13290	-0.5
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	
346	1	0.83508	0.08298	0.73739	-0.14706	0.84349	-0.05567	0.90441	-0.04622	0.89391	...	0.95378	-0.04202	0.83479	0.0
347	1	0.95113	0.00419	0.95183	-0.02723	0.93438	-0.01920	0.94590	0.01606	0.96510	...	0.94520	0.01361	0.93522	0.0
348	1	0.94701	-0.00034	0.93207	-0.03227	0.95177	-0.03431	0.95584	0.02446	0.94124	...	0.93988	0.03193	0.92489	0.0
349	1	0.90608	-0.01657	0.98122	-0.01989	0.95691	-0.03646	0.85746	0.00110	0.89724	...	0.91050	-0.02099	0.89147	-0.0
350	1	0.84710	0.13533	0.73638	-0.06151	0.87873	0.08260	0.88928	-0.09139	0.78735	...	0.86467	-0.15114	0.81147	-0.0

351 rows × 33 columns

In [9]:

```
y2
```

Out[9]:

	34
0	1
1	0
2	1
3	0
4	1
...	...
346	1
347	1
348	1
349	1
350	1

351 rows × 1 columns

Shuffle the dataset then use 80% of data samples for training and 20% for testing.

In [203…

```
# Split, reshape, shuffle
x2_train, x2_test = train_test_split(x2, random_state=111024520, train_size=0.8)
y2_train, y2_test = train_test_split(y2, random_state=111024520, train_size=0.8)
```

Normalize the variables of X

Normalize the continuous variables of X_{train}, X_{test} except the first column, because its values are either 0 or 1.

$$normalize(X) = \frac{X - mean(X_{train})}{sd(X_{train})}$$

In [204…

```
#Normalize
mean_train = np.mean(x2_train, axis = 0)
sd_train = np.std(x2_train, axis = 0)

for i in  range(1, 33):
    x2_train.iloc[:,i] = (x2_train.iloc[:,i]-mean_train[i+1]) / (sd_train[i+1])
    x2_test.iloc[:,i] = (x2_test.iloc[:,i]-mean_train[i+1]) / (sd_train[i+1])

#轉成array
x2_train, x2_test =x2_train.values, x2_test.values
y2_train, y2_test =y2_train.values, y2_test.values

print("Training data: X={}, Y={}".format(x2_train.shape, y2_train.shape))
print("Test data: X={}, Y={}".format(x2_test.shape, y2_test.shape))
```

```
Training data: X=(280, 33), Y=(280, 1)
Test data: X=(71, 33), Y=(71, 1)
```

In [205…

```
class DeepNeuralNetwork_binary():
    def __init__(self, sizes, activation='sigmoid'):
        self.sizes = sizes

        # Choose activation function
        if activation == 'relu':
            self.activation = self.relu
        elif activation == 'sigmoid':
            self.activation = self.sigmoid
        else:
            raise ValueError("Activation function is currently not support, please use 'relu' or 'sigmoid' i

        # Save all weights
        self.params = self.initialize()
        # Save all intermediate values, i.e. activations
        self.cache = {}

    def relu(self, x, derivative=False):
        '''
            Derivative of ReLU is a bit more complicated since it is not differentiable at x = 0
```

```
            Forward path:
            relu(x) = max(0, x)
            In other word,
            relu(x) = 0, if x < 0
                    = x, if x >= 0

            Backward path:
            ∇relu(x) = 0, if x < 0
                     = 1, if x >=0
        '''
        if derivative:
            x = np.where(x < 0, 0, x)
            x = np.where(x >= 0, 1, x)
            return x
        return np.maximum(0, x)

    def sigmoid(self, x, derivative=False):
        '''
            Forward path:
            σ(x) = 1 / 1+exp(-z)

            Backward path:
            ∇σ(x) = exp(-z) / (1+exp(-z))^2
        '''
        if derivative:
            return (np.exp(-x))/((np.exp(-x)+1)**2)

        return 1/(1 + np.exp(-x))

    def initialize(self):
        # number of nodes in each layer
        input_layer=self.sizes[0]
        hidden_layer1=self.sizes[1]
        hidden_layer2=self.sizes[2]
        output_layer=self.sizes[3]

        params = {
            "W1": np.random.randn(hidden_layer1, input_layer) * np.sqrt(1./input_layer),
            "b1": np.zeros((hidden_layer1, 1)),
            "W2": np.random.randn(hidden_layer2, hidden_layer1) * np.sqrt(1./hidden_layer1),
            "b2": np.zeros((hidden_layer2, 1)),
            "W3": np.random.randn(output_layer, hidden_layer2) * np.sqrt(1./hidden_layer2),
            "b3": np.zeros((output_layer, 1))
        }
        return params


    def feed_forward(self, x):
        '''
            y = σ(wX + b)
        '''
        self.cache["X"] = x
        self.cache["Z1"] = np.matmul(self.params["W1"], self.cache["X"].T) + self.params["b1"]
        self.cache["A1"] = self.activation(self.cache["Z1"])
        self.cache["Z2"] = np.matmul(self.params["W2"], self.cache["A1"]) + self.params["b2"]
        self.cache["A2"] = self.activation(self.cache["Z2"])
        self.cache["Z3"] = np.matmul(self.params["W3"], self.cache["A2"]) + self.params["b3"]
        self.cache["A3"] = self.sigmoid(self.cache["Z3"])

        return self.cache["A3"]

    def back_propagate(self, y, output):
        '''
            This is the backpropagation algorithm, for calculating the updates
            of the neural network's parameters.

            Note: There is a stability issue that causes warnings. This is
                  caused by the dot and multiply operations on the huge arrays.

                  RuntimeWarning: invalid value encountered in true_divide
                  RuntimeWarning: overflow encountered in exp
                  RuntimeWarning: overflow encountered in square
        '''
        current_batch_size = y.shape[0]

        dA3 = self.cross_entropy_loss(y, output, derivative = True)
        dZ3 = dA3 * self.sigmoid(self.cache["Z3"], derivative = True) # delta3

        dW3 = (1./current_batch_size) * np.matmul(dZ3, self.cache["A2"].T)
```

```
        db3 = (1./current_batch_size) * np.sum(dZ3, axis=1, keepdims=True)

        dA2 = np.matmul(self.params["W3"].T, dZ3)
        dZ2 = dA2 * self.activation(self.cache["Z2"], derivative=True) #delta2
        dW2 = (1./current_batch_size) * np.matmul(dZ2, self.cache["A1"].T)
        db2 = (1./current_batch_size) * np.sum(dZ2, axis=1, keepdims=True)

        dA1 = np.matmul(self.params["W2"].T, dZ2)
        dZ1 = dA1 * self.activation(self.cache["Z1"], derivative=True)
        dW1 = (1./current_batch_size) * np.matmul(dZ1, self.cache["X"])
        db1 = (1./current_batch_size) * np.sum(dZ1, axis=1, keepdims=True)

        self.grads = {"W1": dW1, "b1": db1, "W2": dW2, "b2": db2,"W3": dW3, "b3": db3}
        return self.grads


    def cross_entropy_loss(self, y, output,derivative = False):
        '''
            L(y, ŷ) = −∑ylog(ŷ)+(1-y)log(1-ŷ).
        '''
        if derivative:
            return (-(y.T / (output + 1e-15)) + ((1 - y.T) / (1 - output + 1e-15))) / len(y)
        return -(y.T * np.log(output + 1e-15) + (1 - y.T) * np.log(1 - output + 1e-15)).mean()


        if derivative:
            l_sum = -(y.T/output -(1- y.T)/(1-output))
        else:
            #epsilon = 1e-15  # 避免log(0)
            #output = np.clip(output, epsilon, 1 - epsilon)  # 限制在 (epsilon, 1 - epsilon)
            l_sum  = - np.sum(y.T * np.log(output) + (1 - y.T) * np.log(1 - output))

        return l_sum


    def optimize(self, l_rate):
        '''
            Stochastic Gradient Descent (SGD):
            θ^(t+1) <- θ^t - η∇L(y, ŷ)
        '''
        if self.optimizer == "sgd":
            for key in self.params:
                self.params[key] = self.params[key] - l_rate * self.grads[key]
        else:
            raise ValueError("Optimizer is currently not support, please use 'sgd' instead.")

    def accuracy(self, y, output):

        predicted = (output.T > 0.5).astype(int)
        correct_predictions = (predicted == y).sum()
        total_samples = len(y)
        accuracy = correct_predictions / total_samples

        return accuracy

    def train(self, x_train, y_train, x_test, y_test, epochs,
              batch_size, optimizer='sgd', l_rate=0.1):

        self.train_loss = []
        self.test_loss = []
        self.train_acc=[]
        self.test_acc=[]
        self.A2 = []

        # Hyperparameters
        self.epochs = epochs
        self.batch_size = batch_size
        num_batches = -(-x_train.shape[0] // self.batch_size)

        # Initialize optimizer
        self.optimizer = optimizer
        if self.optimizer == 'sgd':
            self.params = self.initialize()

        start_time = time.time()
        template = "Epoch {}: {:.2f}s, train acc={:.2f}, train loss={:.2f}, test acc={:.2f}, test loss={:.2f
        
        # Train
        np.random.seed(111024520)
        for i in range(self.epochs):
```

```
            # Shuffle
            permutation = np.random.permutation(x_train.shape[0])
            x_train_shuffled = x_train[permutation]
            y_train_shuffled = y_train[permutation]

            for j in range(num_batches):
                # Batch
                begin = j * self.batch_size
                end = min(begin + self.batch_size, x_train.shape[0])
                x = x_train_shuffled[begin:end]
                y = y_train_shuffled[begin:end]

                # Forward
                output = self.feed_forward(x)
                # Backprop
                grad = self.back_propagate(y, output)
                # Optimize
                self.optimize(l_rate=l_rate)

            # Evaluate performance
            # Training data
            output = self.feed_forward(x_train)
            train_acc = self.accuracy(y_train, output)
            train_loss = self.cross_entropy_loss(y_train, output)
            self.train_loss.append(train_loss)
            self.train_acc.append(train_acc)
            self.A2.append(self.cache["A2"])

            # Test data
            output = self.feed_forward(x_test)
            test_acc = self.accuracy(y_test, output)
            test_loss = self.cross_entropy_loss(y_test, output)
            self.test_loss.append(test_loss)
            self.test_acc.append(test_acc)

            if (i+1) % 500 == 0:
                print(template.format(i+1, time.time()-start_time, train_acc, train_loss, test_acc, test_lo
```

We set epochs$=15000$, batch size$=50$, optimizer='sgd', learning rate $=0.005$, and print the result every 500 epochs.

The output of loss is the value of cross-entropy error.

In [210…

```
dnn2 = DeepNeuralNetwork_binary(sizes=[33,20,10,1], activation='relu')
np.random.seed(111024520)
dnn2.train(x2_train, y2_train, x2_test, y2_test, epochs=15000,batch_size=50, optimizer='sgd', l_rate=0.005)
```

```
Epoch 500: 0.74s, train acc=0.39, train loss=0.68, test acc=0.42, test loss=0.67
Epoch 1000: 1.51s, train acc=0.70, train loss=0.63, test acc=0.79, test loss=0.63
Epoch 1500: 2.25s, train acc=0.78, train loss=0.60, test acc=0.85, test loss=0.60
Epoch 2000: 3.01s, train acc=0.83, train loss=0.58, test acc=0.87, test loss=0.57
Epoch 2500: 3.70s, train acc=0.86, train loss=0.55, test acc=0.86, test loss=0.54
Epoch 3000: 4.38s, train acc=0.86, train loss=0.53, test acc=0.85, test loss=0.52
Epoch 3500: 5.11s, train acc=0.86, train loss=0.51, test acc=0.85, test loss=0.50
Epoch 4000: 5.83s, train acc=0.85, train loss=0.50, test acc=0.85, test loss=0.48
Epoch 4500: 6.47s, train acc=0.85, train loss=0.49, test acc=0.85, test loss=0.47
Epoch 5000: 7.13s, train acc=0.85, train loss=0.48, test acc=0.85, test loss=0.46
Epoch 5500: 7.78s, train acc=0.84, train loss=0.47, test acc=0.83, test loss=0.45
Epoch 6000: 8.45s, train acc=0.84, train loss=0.46, test acc=0.83, test loss=0.45
Epoch 6500: 9.11s, train acc=0.84, train loss=0.45, test acc=0.83, test loss=0.44
Epoch 7000: 9.76s, train acc=0.84, train loss=0.45, test acc=0.83, test loss=0.44
Epoch 7500: 10.39s, train acc=0.84, train loss=0.44, test acc=0.83, test loss=0.44
Epoch 8000: 11.06s, train acc=0.84, train loss=0.44, test acc=0.83, test loss=0.44
Epoch 8500: 11.73s, train acc=0.84, train loss=0.43, test acc=0.83, test loss=0.43
Epoch 9000: 12.45s, train acc=0.84, train loss=0.43, test acc=0.83, test loss=0.43
Epoch 9500: 13.16s, train acc=0.85, train loss=0.42, test acc=0.83, test loss=0.43
Epoch 10000: 13.86s, train acc=0.85, train loss=0.41, test acc=0.83, test loss=0.43
Epoch 10500: 14.75s, train acc=0.86, train loss=0.40, test acc=0.83, test loss=0.43
Epoch 11000: 15.69s, train acc=0.86, train loss=0.39, test acc=0.83, test loss=0.42
Epoch 11500: 16.65s, train acc=0.87, train loss=0.39, test acc=0.83, test loss=0.42
Epoch 12000: 17.55s, train acc=0.88, train loss=0.38, test acc=0.85, test loss=0.41
Epoch 12500: 18.59s, train acc=0.87, train loss=0.38, test acc=0.85, test loss=0.41
Epoch 13000: 19.56s, train acc=0.88, train loss=0.38, test acc=0.85, test loss=0.40
Epoch 13500: 20.31s, train acc=0.88, train loss=0.37, test acc=0.85, test loss=0.40
Epoch 14000: 21.04s, train acc=0.87, train loss=0.37, test acc=0.85, test loss=0.40
Epoch 14500: 21.81s, train acc=0.87, train loss=0.37, test acc=0.85, test loss=0.40
Epoch 15000: 22.55s, train acc=0.88, train loss=0.36, test acc=0.85, test loss=0.39
```

(b)

(1) network architecture (number of hidden layers and neurons)

In [211...
```
df_2 =pd.DataFrame({'number of neurons': [33,20,10,1],'activation function': ['', 'relu', 'relu', 'sigmoid']
                   },index=['input layer', 'hidden layer1', 'hidden layer2','output layer'])
df_2
```

Out[211]:

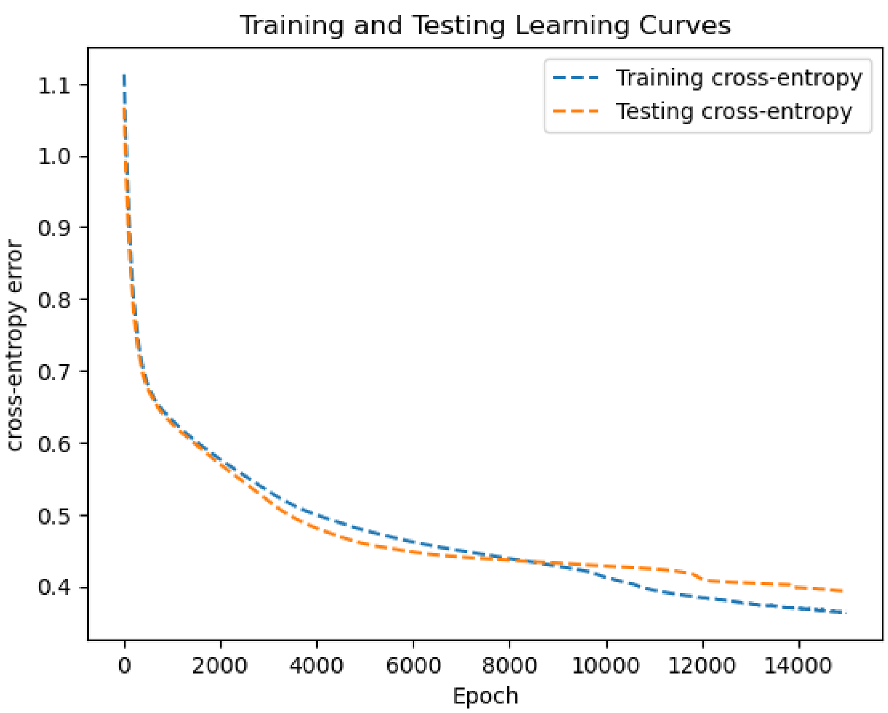
	number of neurons	activation function
input layer	33	
hidden layer1	20	relu
hidden layer2	10	relu
output layer	1	sigmoid

(2) learning curve

We draw the learning curve of loss function: cross-entropy error function.

$$cross-entropy\ error = \sum_{n=1}^{N}\sum_{k=1}^{K} y_{nk} log\hat{y}_k$$

In [212...
```
epochs = list(range(1, len(dnn2.train_loss) + 1))

plt.plot(epochs, dnn2.train_loss, label='Training cross-entropy', linestyle='--')
plt.plot(epochs, dnn2.test_loss, label='Testing cross-entropy',linestyle='--')
plt.legend()
plt.xlabel('Epoch')
plt.ylabel('cross-entropy error')
plt.title('Training and Testing Learning Curves')
plt.show()
```



We can found that when Epoch>9000，the Training cross-entropy will be much lower than Testing cross-entropy.

(3) training error rate

$$

error\ rate*{training}=1-accuracy*{training} $$

In [213...
```
print("Training error rate: {} ".format(round(1-dnn2.train_acc[14999],3)))
```

```
Training error rate: 0.125
```

(4) testing error rate

$$

error\ rate*{testing}=1-accuracy*{testing} $$

In [214…
```python
print("Testing error rate: {} ".format(round(1-dnn2.test_acc[14999],3)))
```

```
Testing error rate: 0.155
```

(c) Compare the results of choosing different numbers of nodes in the layer before the output layer by plotting the distribution of latent features at different training stage.

Model 1: numbers of nodes in the layer before the output layer = $\color{red}{2}$

In [215…
```python
df_node2 =pd.DataFrame({'number of neurons': [33,20,2,1],'activation function': ['', 'relu', 'relu', 'sigmoi
                   },index=['input layer', 'hidden layer1', 'hidden layer2','output layer'])
df_node2
```

Out[215]:

	number of neurons	activation function
input layer	33	
hidden layer1	20	relu
hidden layer2	2	relu
output layer	1	sigmoid

We set epochs$=15000$, batch size$=50$, optimizer='sgd', learning rate $=0.008$, and print the result every 500 epochs.

In [216…
```python
dnn2_node2 = DeepNeuralNetwork_binary(sizes=[33,20,2,1], activation='relu')
np.random.seed(111024520)
dnn2_node2.train(x2_train, y2_train, x2_test, y2_test, epochs=15000,batch_size=50, optimizer='sgd', l_rate=0
```

```
Epoch 500: 0.76s, train acc=0.62, train loss=0.71, test acc=0.58, test loss=0.71
Epoch 1000: 1.36s, train acc=0.69, train loss=0.66, test acc=0.63, test loss=0.67
Epoch 1500: 2.01s, train acc=0.75, train loss=0.62, test acc=0.75, test loss=0.62
Epoch 2000: 2.64s, train acc=0.78, train loss=0.59, test acc=0.79, test loss=0.58
Epoch 2500: 3.31s, train acc=0.79, train loss=0.56, test acc=0.83, test loss=0.53
Epoch 3000: 3.96s, train acc=0.81, train loss=0.53, test acc=0.86, test loss=0.49
Epoch 3500: 4.57s, train acc=0.82, train loss=0.50, test acc=0.87, test loss=0.46
Epoch 4000: 5.16s, train acc=0.83, train loss=0.47, test acc=0.87, test loss=0.43
Epoch 4500: 5.77s, train acc=0.83, train loss=0.45, test acc=0.87, test loss=0.41
Epoch 5000: 6.37s, train acc=0.84, train loss=0.43, test acc=0.87, test loss=0.40
Epoch 5500: 6.99s, train acc=0.84, train loss=0.42, test acc=0.87, test loss=0.38
Epoch 6000: 7.79s, train acc=0.84, train loss=0.40, test acc=0.87, test loss=0.37
Epoch 6500: 8.65s, train acc=0.84, train loss=0.39, test acc=0.89, test loss=0.36
Epoch 7000: 9.34s, train acc=0.85, train loss=0.37, test acc=0.87, test loss=0.36
Epoch 7500: 10.16s, train acc=0.86, train loss=0.36, test acc=0.86, test loss=0.35
Epoch 8000: 10.96s, train acc=0.87, train loss=0.34, test acc=0.86, test loss=0.35
Epoch 8500: 11.71s, train acc=0.88, train loss=0.33, test acc=0.86, test loss=0.34
Epoch 9000: 12.39s, train acc=0.87, train loss=0.32, test acc=0.86, test loss=0.34
Epoch 9500: 13.18s, train acc=0.88, train loss=0.32, test acc=0.85, test loss=0.34
Epoch 10000: 13.82s, train acc=0.89, train loss=0.31, test acc=0.85, test loss=0.33
Epoch 10500: 14.43s, train acc=0.89, train loss=0.30, test acc=0.87, test loss=0.32
Epoch 11000: 15.03s, train acc=0.89, train loss=0.29, test acc=0.87, test loss=0.32
Epoch 11500: 15.66s, train acc=0.89, train loss=0.28, test acc=0.87, test loss=0.32
Epoch 12000: 16.34s, train acc=0.90, train loss=0.28, test acc=0.89, test loss=0.31
Epoch 12500: 17.03s, train acc=0.90, train loss=0.27, test acc=0.89, test loss=0.31
Epoch 13000: 17.65s, train acc=0.90, train loss=0.27, test acc=0.89, test loss=0.31
Epoch 13500: 18.32s, train acc=0.90, train loss=0.27, test acc=0.89, test loss=0.31
Epoch 14000: 18.90s, train acc=0.91, train loss=0.26, test acc=0.89, test loss=0.30
Epoch 14500: 19.53s, train acc=0.91, train loss=0.26, test acc=0.89, test loss=0.30
Epoch 15000: 20.21s, train acc=0.91, train loss=0.26, test acc=0.89, test loss=0.30
```

In this case, the last hidden layer is a layer with 2 neurons, and their outputs are considered latent features.

$$\text{latent features: } a_2 = activation(z_2)$$

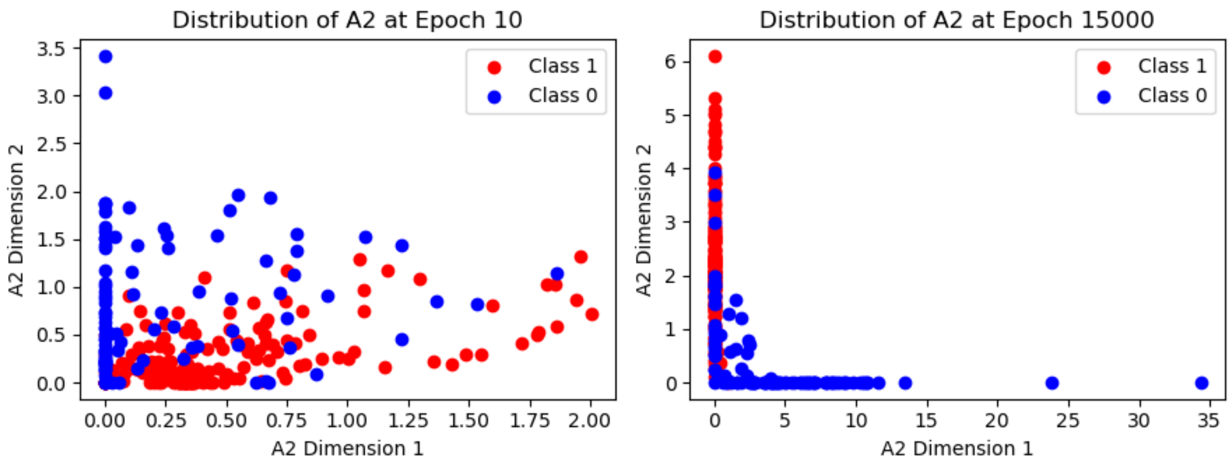
\ Latent features: $a2$ are $2-dim$ because hidden layer2 has 2 neurons.\ Then plotting the distribution of latent features at different training stage: Epoch 10, Epoch 15000

In [217…

```
class_1_idx = np.where(y2_train == 1)
class_0_idx = np.where(y2_train == 0)
```

In [218…

```
#epoch=10
red_points =dnn2_node2.A2[9].T[class_1_idx[0]]  #good
blue_points = dnn2_node2.A2[9].T[class_0_idx[0]]  #bad

plt.figure(figsize=(9, 3.5))
plt.subplot(1, 2, 1)
plt.scatter(red_points[:, 0], red_points[:, 1], c='red', label='Class 1')
plt.scatter(blue_points[:, 0], blue_points[:, 1], c='blue', label='Class 0')

plt.title('Distribution of A2 at Epoch 10')
plt.xlabel('A2 Dimension 1')
plt.ylabel('A2 Dimension 2')
plt.legend()

#epoch=10000
red_points2 =dnn2_node2.A2[14999].T[class_1_idx[0]]  #good
blue_points2 = dnn2_node2.A2[14999].T[class_0_idx[0]]  #bad


plt.subplot(1, 2, 2)
plt.scatter(red_points2[:, 0], red_points2[:, 1], c='red', label='Class 1')
plt.scatter(blue_points2[:, 0], blue_points2[:, 1], c='blue', label='Class 0')

plt.title('Distribution of A2 at Epoch 15000')
plt.xlabel('A2 Dimension 1')
plt.ylabel('A2 Dimension 2')
plt.legend()

plt.tight_layout()
plt.show()
```



觀察上圖:\ Epoch 10：藍點與紅點重疊程度高，混雜在圖中。\ Epoch 15000：紅點集中在左側，藍點較集中在右側，兩者區分程度相較Epoch 10的圖明顯許多。

Model 2: numbers of nodes in the layer before the output layer = 3

In [219…

```
df_node3 =pd.DataFrame({'number of neurons': [33,20,3,1],'activation function': ['', 'relu', 'relu', 'sigmoi
                   },index=['input layer', 'hidden layer1', 'hidden layer2','output layer'])
df_node3
```

Out[219]:

	number of neurons	activation function
input layer	33	
hidden layer1	20	relu
hidden layer2	3	relu
output layer	1	sigmoid

In [220…

```
dnn2_node3 = DeepNeuralNetwork_binary(sizes=[33,20,3,1], activation='relu')
np.random.seed(111024520)
dnn2_node3.train(x2_train, y2_train, x2_test, y2_test, epochs=15000,batch_size=50, optimizer='sgd', l_rate=0
```

```
Epoch 500: 0.75s, train acc=0.65, train loss=0.66, test acc=0.63, test loss=0.65
Epoch 1000: 1.45s, train acc=0.65, train loss=0.63, test acc=0.62, test loss=0.62
Epoch 1500: 2.17s, train acc=0.66, train loss=0.60, test acc=0.62, test loss=0.59
Epoch 2000: 2.87s, train acc=0.66, train loss=0.58, test acc=0.62, test loss=0.57
Epoch 2500: 3.56s, train acc=0.66, train loss=0.56, test acc=0.62, test loss=0.56
Epoch 3000: 4.18s, train acc=0.69, train loss=0.54, test acc=0.65, test loss=0.55
Epoch 3500: 4.80s, train acc=0.72, train loss=0.53, test acc=0.69, test loss=0.54
Epoch 4000: 5.44s, train acc=0.76, train loss=0.52, test acc=0.76, test loss=0.53
Epoch 4500: 6.05s, train acc=0.78, train loss=0.51, test acc=0.83, test loss=0.52
Epoch 5000: 6.77s, train acc=0.81, train loss=0.49, test acc=0.83, test loss=0.50
Epoch 5500: 7.46s, train acc=0.82, train loss=0.47, test acc=0.83, test loss=0.48
Epoch 6000: 8.06s, train acc=0.82, train loss=0.45, test acc=0.85, test loss=0.45
Epoch 6500: 8.65s, train acc=0.83, train loss=0.42, test acc=0.85, test loss=0.42
Epoch 7000: 9.41s, train acc=0.84, train loss=0.40, test acc=0.86, test loss=0.39
Epoch 7500: 10.14s, train acc=0.86, train loss=0.37, test acc=0.87, test loss=0.37
Epoch 8000: 10.83s, train acc=0.86, train loss=0.35, test acc=0.89, test loss=0.35
Epoch 8500: 11.53s, train acc=0.86, train loss=0.33, test acc=0.90, test loss=0.34
Epoch 9000: 12.17s, train acc=0.86, train loss=0.32, test acc=0.90, test loss=0.32
Epoch 9500: 12.87s, train acc=0.89, train loss=0.30, test acc=0.90, test loss=0.31
Epoch 10000: 13.55s, train acc=0.89, train loss=0.29, test acc=0.90, test loss=0.30
Epoch 10500: 14.15s, train acc=0.90, train loss=0.28, test acc=0.89, test loss=0.30
Epoch 11000: 14.77s, train acc=0.90, train loss=0.27, test acc=0.90, test loss=0.29
Epoch 11500: 15.39s, train acc=0.91, train loss=0.26, test acc=0.90, test loss=0.29
Epoch 12000: 16.08s, train acc=0.91, train loss=0.25, test acc=0.93, test loss=0.28
Epoch 12500: 16.72s, train acc=0.91, train loss=0.25, test acc=0.93, test loss=0.28
Epoch 13000: 17.39s, train acc=0.91, train loss=0.24, test acc=0.93, test loss=0.27
Epoch 13500: 18.01s, train acc=0.92, train loss=0.24, test acc=0.93, test loss=0.27
Epoch 14000: 18.69s, train acc=0.92, train loss=0.23, test acc=0.93, test loss=0.26
Epoch 14500: 19.39s, train acc=0.93, train loss=0.22, test acc=0.93, test loss=0.26
Epoch 15000: 20.04s, train acc=0.93, train loss=0.22, test acc=0.92, test loss=0.25
```

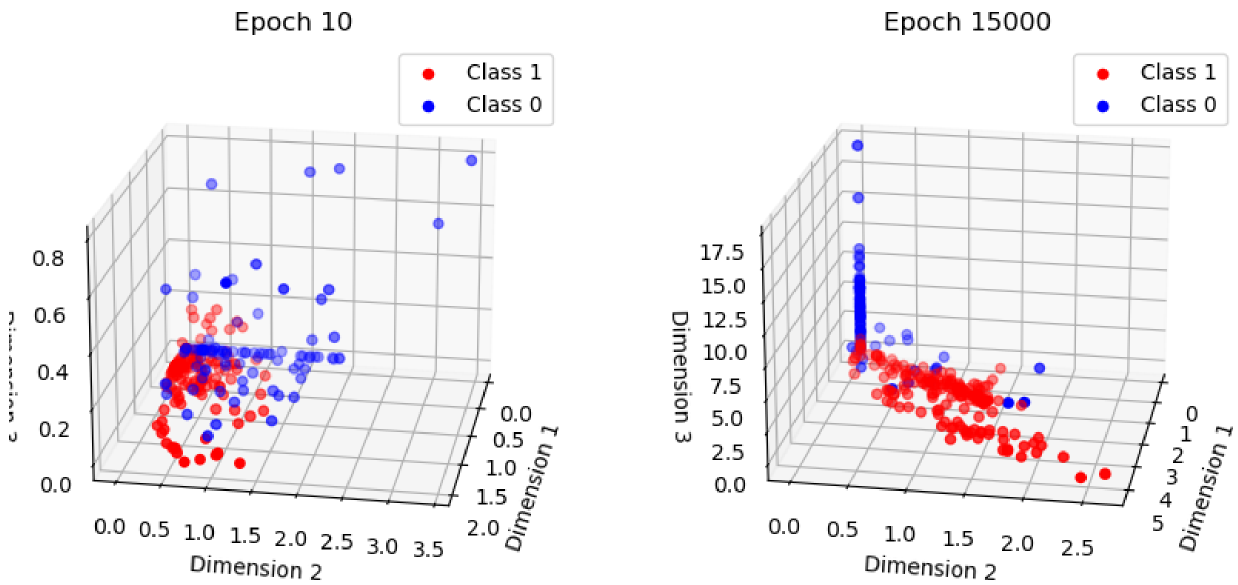
In this case, the last hidden layer is a layer with 3 neurons, and their outputs are considered latent features.

$$\text{latent features: } a_2 = activation(z_2)$$

\ Latent features: a_2 are $3-dim$ because hidden layer2 has 3 neurons.\ Then making the 3D-plot of the distribution of latent features at different training stage: Epoch 10, Epoch 15000

In [221…

```
from mpl_toolkits.mplot3d import Axes3D

# Data for Epoch 10
red_points = dnn2_node3.A2[9].T[class_1_idx[0]]
blue_points = dnn2_node3.A2[9].T[class_0_idx[0]]

# Create the first 3D plot for Epoch 10
fig = plt.figure(figsize=(9, 4))
ax = fig.add_subplot(121, projection='3d')  # Create a subplot with 1 row and 2 columns, and use the first c

ax.scatter(red_points[:, 0], red_points[:, 1], red_points[:, 2], c='red', label='Class 1')
ax.scatter(blue_points[:, 0], blue_points[:, 1], blue_points[:, 2], c='blue', label='Class 0')
ax.set_xlabel('Dimension 1')
ax.set_ylabel('Dimension 2')
ax.set_zlabel('Dimension 3')
ax.view_init(elev=20, azim=10)
ax.set_title('Epoch 10')
ax.legend()

# Data for Epoch 15000
red_points2 = dnn2_node3.A2[14999].T[class_1_idx[0]]
blue_points2 = dnn2_node3.A2[14999].T[class_0_idx[0]]

# Create the second 3D plot for Epoch 15000

ax2 = fig.add_subplot(122, projection='3d')  # Create a subplot with 1 row and 2 columns, and use the second

ax2.scatter(red_points2[:, 0], red_points2[:, 1], red_points2[:, 2], c='red', label='Class 1')
ax2.scatter(blue_points2[:, 0], blue_points2[:, 1], blue_points2[:, 2], c='blue', label='Class 0')
ax2.set_xlabel('Dimension 1')
ax2.set_ylabel('Dimension 2')
ax2.set_zlabel('Dimension 3')
ax2.view_init(elev=20, azim=10)
ax2.set_title('Epoch 15000')
ax2.legend()

plt.tight_layout()
plt.show()
```



觀察上圖:\ Epoch 10：藍點與紅點有部分重疊，無法區分開來。\ Epoch 15000：紅點集中在前方，藍點較集中在後方，兩者分布方向不同，可大致區分開來。