ML hw3

107081028 常安彥

1. Implementation
   a. Preprocessing

Data Loading

```
##### Preprocessing
classes = ['Carambula', 'Lychee', 'Pear']
label = 0
train_data, train_label, test_data, test_label = [], [], [], []
for class_name in classes:
    for filename in os.listdir('./Data_train/' + class_name):
        image = (np.array(cv2.imread('Data_train/' + class_name + '/' + filename))[:,:,0] / 255.0)
        train_data.append(image.reshape(image.shape[0] * image.shape[1]))
        train_label.append(label)
    for filename in os.listdir('./Data_test/' + class_name):
        image = (np.array(cv2.imread('Data_test/' + class_name + '/' + filename))[:,:,0] / 255.0)
        test_data.append(image.reshape(image.shape[0] * image.shape[1]))
        test_label.append(label)
    label += 1
```

I read the images with cv2 package and discarded the alpha channel of them. Also, I normalized the input images and flatten the two-dimensional data with reshape function.

PCA

```
# PCA
pca = PCA(2)                                    # 2 principal components as required
train_X = pca.fit_transform(train_data)
test_X = pca.transform(test_data)
train_label = np.array(train_label)
test_label = np.array(test_label)
print(train_label.shape)
```

Using the PCA function imported from scikit learn, I extracted the input data into two features. The PCA function was fit with training data and transformed both the training and testing data.

One Hot

```
# One Hot label
test_y = np.array(test_label.astype('int32')[:, None] == np.arange(3), dtype=np.float32)
train_y = np.array(train_label.astype('int32')[:, None] == np.arange(3), dtype=np.float32)
```

I turned the training and testing label into one hot code in order to feed into the deep neural network. Here, the number of classes is 3, so I constructed a numpy array with three objects for each label.

b. Deep Neural Network

```
##### Train
DNN = NeuralNetwork(input_size = 2, hidden_size = 16, output_size = 3)
layer2_test_acc, layer2_test_loss, layer2_train_loss_curve, layer2_test_loss_curve \
    = DNN.train(train_X, train_y, test_X, test_y, batch_size=32, lr=0.1, beta=0.9)

DNN3 = NeuralNetwork_3layer(input_size = 2, hidden_size = 16, output_size = 3)
layer3_test_acc, layer3_test_loss, layer3_train_loss_curve, layer3_test_loss_curve \
    = DNN3.train(train_X, train_y, test_X, test_y, batch_size=32, lr=0.1, beta=0.9)
```

This initializes the neural network class by init function.

```
# Deep Neural Network 2 Layer
class NeuralNetwork():
    def __init__(self, input_size, hidden_size, output_size):
        self.in_size = input_size
        self.hidden_size = hidden_size
        self.out_size = output_size
        self.parameters = self.initialize()
        self.cache = {}
```

Initialize

```
def initialize(self):
    input_layer = self.in_size
    hidden_layer = self.hidden_size
    output_layer = self.out_size

    param = {
        "W1": np.random.randn(hidden_layer, input_layer) * np.sqrt(1./input_layer),
        "b1": np.zeros((hidden_layer, 1)) * np.sqrt(1./input_layer),
        "W2": np.random.randn(output_layer, hidden_layer) * np.sqrt(1./hidden_layer),
        "b2": np.zeros((output_layer, 1)) * np.sqrt(1./hidden_layer)
    }
    return param
```

(Two layers)

I used randn() function in random package in numpy to receive the initial weights, drawn from the standard normal distribution. Also, initialize the bias with zeros function. I set the variance for each layer to 1/n, where n is the number of inputs feeding into the very layer, by dividing it by n^0.5.

```
param = {
    "W0": np.random.randn(hidden_layer, input_layer) * np.sqrt(1./input_layer),
    "b0": np.zeros((hidden_layer, 1)) * np.sqrt(1./input_layer),
    "W1": np.random.randn(hidden_layer*2, hidden_layer) * np.sqrt(1./hidden_layer),
    "b1": np.zeros((hidden_layer*2, 1)) * np.sqrt(1./hidden_layer),
    "W2": np.random.randn(output_layer, (hidden_layer*2)) * np.sqrt(1./(hidden_layer*2)),
    "b2": np.zeros((output_layer, 1)) * np.sqrt(1./(hidden_layer*2))
}
return param
```

(Parameters for three layers) Same initialization method

Feed Forward

$$y_k(\mathbf{x}, \mathbf{w}) = \sigma\left(\sum_{j=1}^{M} w_{kj}^{(2)}\, h\left(\sum_{i=1}^{D} w_{ji}^{(1)} x_i + w_{j0}^{(1)}\right) + w_{k0}^{(2)}\right)$$

```
def feed_forward(self, x):
    self.cache["X"] = x
    self.cache["Z1"] = np.matmul(self.parameters["W1"], self.cache["X"].T) + self.parameters["b1"]
    self.cache["A1"] = self.sigmoid(self.cache["Z1"])
    self.cache["Z2"] = np.matmul(self.parameters["W2"], self.cache["A1"]) + self.parameters["b2"]
    self.cache["A2"] = self.sigmoid(self.cache["Z2"])
    return self.cache["A2"]
```

I multiplied the weights by the activations of the previous layer, and apply the activation function to the outcome. In order to get through each layer, I sequentially apply dot operation, which turned out to be matrix multiplication in this situation, followed by sigmoid activation.

y = σ (wX + b)

Forward pass:

$\boldsymbol{A}^{(0)} \leftarrow \left[\, \boldsymbol{a}^{(0,1)} \quad \cdots \quad \boldsymbol{a}^{(0,M)} \,\right]^{\top}$;

for $k \leftarrow 1$ *to* L **do**

$\quad \boldsymbol{Z}^{(k)} \leftarrow \boldsymbol{A}^{(k-1)} \boldsymbol{W}^{(k)}$;

$\quad \boldsymbol{A}^{(k)} \leftarrow \mathrm{act}(\boldsymbol{Z}^{(k)})$;

end

```
def feed_forward(self, x):
    self.cache["X"] = x
    self.cache["Z0"] = np.matmul(self.parameters["W0"], self.cache["X"].T) + self.parameters["b0"]
    self.cache["A0"] = self.sigmoid(self.cache["Z0"])
    self.cache["Z1"] = np.matmul(self.parameters["W1"], self.cache["A0"]) + self.parameters["b1"]
    self.cache["A1"] = self.sigmoid(self.cache["Z1"])
    self.cache["Z2"] = np.matmul(self.parameters["W2"], self.cache["A1"]) + self.parameters["b2"]
    self.cache["A2"] = self.sigmoid(self.cache["Z2"])
    return self.cache["A2"]
```

(Feed forward function for three layers) Same implementation method

Backpropagation

Recall that, in general, each unit computes a weighted sum:

$a_j = \sum_i w_{ji} z_i$ with activation $z_j = h(a_j)$

For each error term: $\frac{\partial E_n}{\partial w_{ji}} = \underbrace{\frac{\partial E_n}{\partial a_j}}_{\equiv \delta_j} \frac{\partial a_j}{\partial w_{ji}}$

Therefore $\frac{\partial E_n}{\partial w_{ji}} = \delta_j z_i$

In the network $\delta_j \equiv \frac{\partial E_n}{\partial a_j} = \sum_k \frac{\partial E_n}{\partial a_k} \frac{\partial a_k}{\partial a_j}$ where $j \to \{k\}$

Algorithm: $\delta_j = h'(a_j) \sum_k w_{kj} \delta_k$ as $\frac{\partial a_k}{\partial a_j} = \frac{\partial a_k}{\partial z_j} \frac{\partial z_j}{\partial a_j}$

```
def back_propagate(self, y, output):
    batch_size = y.shape[0]
    dZ2 = output - y.T
    dW2 = (1./batch_size) * np.matmul(dZ2, self.cache["A1"].T)
    db2 = (1./batch_size) * np.sum(dZ2, axis=1, keepdims=True)

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

    self.grads = {"W1": dW1, "b1": db1, "W2": dW2, "b2": db2}
    return self.grads
```

First, calculate the error output by the link. Then, multiply with the input to the link. Finally sum up over the data points.

Backward pass:

Compute error signals

$$\Delta^{(L)} = \begin{bmatrix} \delta^{(L,0)} & \dots & \delta^{(L,M)} \end{bmatrix}^{\top}$$

for $k \leftarrow L-1$ *to* 1 **do**

$$\Delta^{(k)} \leftarrow \mathrm{act}'(\boldsymbol{Z}^{(k)}) \odot (\Delta^{(k+1)} \boldsymbol{W}^{(k+1)\top}) \; ;$$

end

Return $\frac{\partial c^{(n)}}{\partial \boldsymbol{W}^{(k)}} = \sum_{n=1}^{M} \boldsymbol{a}^{(k-1,n)} \otimes \boldsymbol{\delta}^{(k,n)}$ for all k

```
def back_propagate(self, y, output):
    batch_size = y.shape[0]
    dZ2 = output - y.T
    dW2 = (1./batch_size) * np.matmul(dZ2, self.cache["A1"].T)
    db2 = (1./batch_size) * np.sum(dZ2, axis=1, keepdims=True)

    dA1 = np.matmul(self.parameters["W2"].T, dZ2)
    dZ1 = dA1 * self.sigmoid(self.cache["Z1"], derivative=True)
    dW1 = (1./batch_size) * np.matmul(dZ1, self.cache["A0"].T)
    db1 = (1./batch_size) * np.sum(dZ1, axis=1, keepdims=True)

    dA0 = np.matmul(self.parameters["W1"].T, dZ1)
    dZ0 = dA0 * self.sigmoid(self.cache["Z0"], derivative=True)
    dW0 = (1./batch_size) * np.matmul(dZ0, self.cache["X"])
    db0 = (1./batch_size) * np.sum(dZ0, axis=1, keepdims=True)

    self.grads = {"W0": dW0, "b0": db0, "W1": dW1, "b1": db1, "W2": dW2, "b2": db2}
    return self.grads
```

(Backpropagation function for three layers) Same implementation method

Sigmoid

$$s(a) = \frac{1}{1 + \exp(-a)}$$

```
def sigmoid(self, x, derivative = False):
    if derivative:
        return (np.exp(-x)) / ((np.exp(-x)+1) ** 2)
    return 1 / (1 + np.exp(-x))
```

Sigmoid function is implemented in two ways.

Forward path: σ(x) = 1 / [1 + exp(-z)]

Backward path: ∇σ(x) = exp(-z) / [1 + exp(-z)] ^ 2

Optimization

```
def SGD(self, lr = 0.1, beta = 0.9):
    for key in self.parameters:
        self.parameters[key] = self.parameters[key] - lr * self.grads[key]
```

As required, stochastic gradient descent is implemented. Beta is used for momentum, which is not used in this assignment.

θ^(t + 1) <- θ^t - η∇L(y, ý)

Cross Entropy Loss

- Empirical loss function

$$L(\boldsymbol{w}) = \frac{1}{n}\sum_{i} l(z_i, f_{\boldsymbol{w}}(\boldsymbol{x_i}))$$

```
def cross_entropy_loss(self, y, output):
    l_sum = np.sum(np.multiply(y.T, np.log(output)))
    m = y.shape[0]
    l = -(1./m) * l_sum
    return l
```

Cross entropy loss is used in this case of classification.

L(y, ý) = - Σy log(ý)

Training

```
for i in range(self.epochs):
    shuffled = np.random.permutation(x_train.shape[0])
    x_train_shuffled = x_train[shuffled]
    y_train_shuffled = y_train[shuffled]
```

First of all, do permutation to the training data set in order to shuffle the input. Reorder the data set with the shuffled index.

```
    for j in range(num_batches):
        # Setting Batch
        begin = j * self.batch_size
        end = min(begin + self.batch_size, x_train.shape[0]-1)
        x = x_train_shuffled[begin:end]
        y = y_train_shuffled[begin:end]

        output = self.feed_forward(x)              # Forward
        grad = self.back_propagate(y, output)      # Backprop
        self.SGD(lr=lr, beta=beta)                 # Optimize
```

Next, following the order of the shuffled data, do feed forward, backpropagation and optimization on the data in batch size.

```
# Training Evaluation
output = self.feed_forward(x_train)
train_acc = self.accuracy(y_train, output)
train_loss = self.cross_entropy_loss(y_train, output)
train_loss_curve.append(train_loss)
# Test data Evaluation
output = self.feed_forward(x_test)
test_acc = self.accuracy(y_test, output)
test_loss = self.cross_entropy_loss(y_test, output)
test_loss_curve.append(test_loss)
print(f"Epoch {i + 1}: train acc={train_acc:.2f}, train loss=
```

Finally, evaluate the training process and prediction on testing data. Print the training and testing loss and accuracy. Part of the training log:

```
Epoch 78: train acc=0.88, train loss=0.34, test acc=0.84, test loss=0.33
Epoch 79: train acc=0.88, train loss=0.33, test acc=0.87, test loss=0.30
Epoch 80: train acc=0.88, train loss=0.33, test acc=0.82, test loss=0.32
Epoch 81: train acc=0.88, train loss=0.33, test acc=0.91, test loss=0.28
Epoch 82: train acc=0.88, train loss=0.33, test acc=0.87, test loss=0.30
Epoch 83: train acc=0.88, train loss=0.33, test acc=0.90, test loss=0.28
Epoch 84: train acc=0.88, train loss=0.32, test acc=0.87, test loss=0.29
Epoch 85: train acc=0.88, train loss=0.32, test acc=0.91, test loss=0.27
Epoch 86: train acc=0.88, train loss=0.32, test acc=0.87, test loss=0.28
Epoch 87: train acc=0.88, train loss=0.32, test acc=0.90, test loss=0.26
Epoch 88: train acc=0.88, train loss=0.32, test acc=0.85, test loss=0.31
Epoch 89: train acc=0.88, train loss=0.31, test acc=0.87, test loss=0.27
Epoch 90: train acc=0.88, train loss=0.31, test acc=0.87, test loss=0.28
Epoch 91: train acc=0.89, train loss=0.31, test acc=0.88, test loss=0.27
Epoch 92: train acc=0.89, train loss=0.31, test acc=0.88, test loss=0.26
Epoch 93: train acc=0.89, train loss=0.31, test acc=0.88, test loss=0.26
Epoch 94: train acc=0.88, train loss=0.31, test acc=0.85, test loss=0.29
Epoch 95: train acc=0.88, train loss=0.31, test acc=0.84, test loss=0.31
Epoch 96: train acc=0.88, train loss=0.30, test acc=0.86, test loss=0.26
Epoch 97: train acc=0.88, train loss=0.30, test acc=0.88, test loss=0.25
Epoch 98: train acc=0.89, train loss=0.31, test acc=0.89, test loss=0.25
Epoch 99: train acc=0.89, train loss=0.30, test acc=0.88, test loss=0.25
Epoch 100: train acc=0.88, train loss=0.30, test acc=0.87, test loss=0.26
```

2. Result with discussion of 2-layer and 3-layer

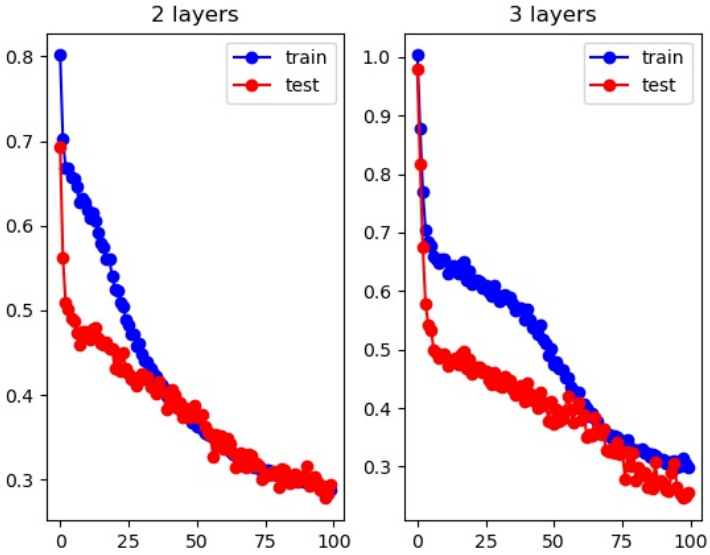
a. Test Accuracy

```
def accuracy(self, y, output):
    return np.mean(np.argmax(y, axis=-1) == np.argmax(output.T, axis=-1))
```

Accuracy function is implemented with argmax function finding the highest value through the one hot code outputs and labels.

```
2 Layer: test acc=0.8433734939759037, test loss=0.29461498569438704
3 Layer: test acc=0.8714859437751004, test loss=0.25627918604181155
```

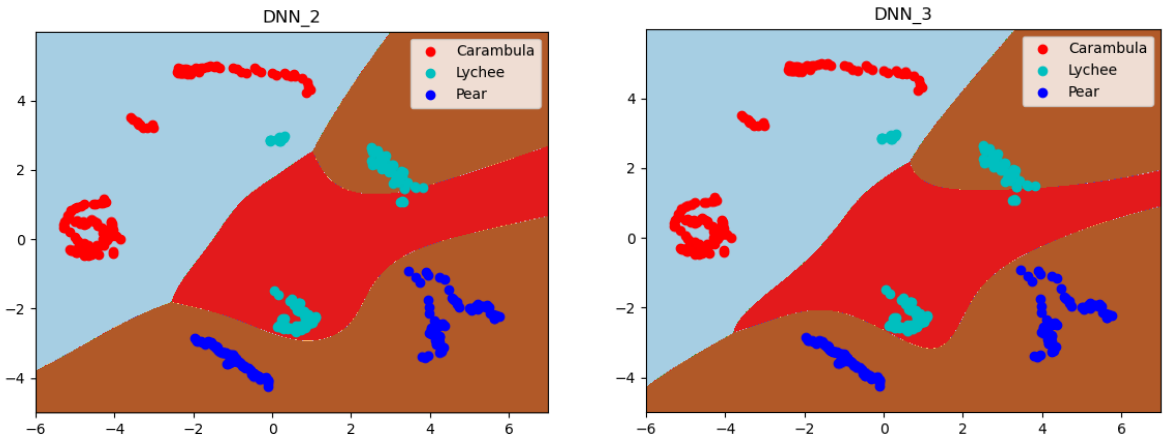
b. Loss Curves



According to the loss curve, the loss of the three-layer model decreases more rapidly than the two-layer model. Nevertheless, the loss jitters in a bigger range when the loss converges, which means that it is affected in a bigger amplitude.

c. Decision Regions

3-layer model slightly better set the boundary between lychee and pear.

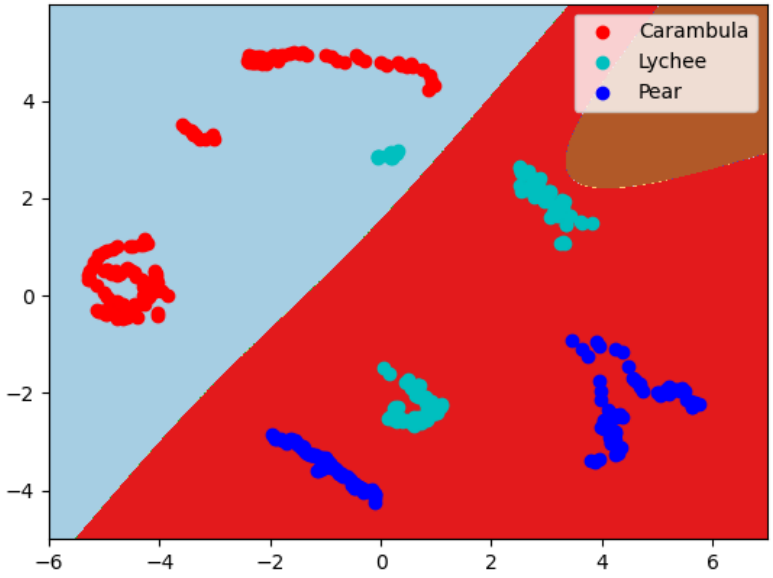


3. Discussion of numbers of hidden size

I implemented the 3-layer model with hidden size equals to 5 and hidden size equals to 16 in order to make a comparison.

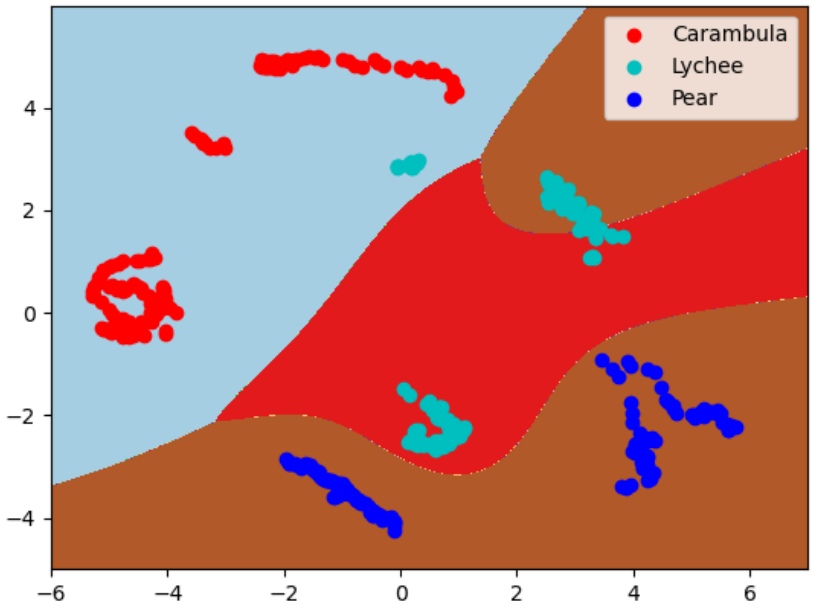
```
5 hidden size: test acc=0.6325301204819277, test loss=0.5349732908520926
16 hidden size: test acc=0.8554216867469879, test loss=0.24116509293627023
```

The accuracy obviously shows that a model with too small amount of hidden size cannot handle the classification of this case. The number of weights is not enough to calculate the relation among the data set.



(5 hidden size)

Furthermore, through the plot of decision region made by 5-hidden size model, we can see that it can't properly draw the boundaries between three classes.



(16 hidden size)

4. References
    a. https://www.coursera.org/lecture/deep-neural-network/weight-initialization-for-deep-networks-RwqYe
    b. https://mlfromscratch.com/neural-networks-explained/#/
    c. Teacher's slides