```
from sklearn.metrics import accuracy_score
import matplotlib.pyplot as plt
import numpy as np
import struct
import pickle
import time
import os
```

```
root = ''
data_path = os.path.join(root, 'data')
```

Layer Object

```
class Layer:
    def __init__(self, layer_in_size: int, layer_out_size: int, activation: str):
        np.random.seed(42)

        # Drawing samples from LeCun normal distribution
        # Source: https://arxiv.org/pdf/1706.02515.pdf

        self.weights = np.random.normal(loc=0, scale=(1 / layer_in_size), size=(layer_in_size, layer_out_size))
        self.biases = np.random.normal(loc=0, scale=(1 / layer_in_size), size=(layer_out_size, 1))

        if activation == 'tanh':
            self.__activation_function = lambda v: np.tanh(v)
            self.__derivative_function = lambda v: 1 - (np.tanh(v) ** 2)
        elif activation == 'sigmoid':
            self.__activation_function = lambda v: 1 / (1 + np.exp(-v))
            self.__derivative_function = lambda v: self.__sigmoid_derivative(v)
        else:
            self.__activation_function = lambda v: v
            self.__derivative_function = lambda v: 1

    def local_fields(self, data_in):
        return self.weights.T.dot(data_in) + self.biases

    def activations(self, local_fields):
        return self.__activation_function(local_fields)

    def derivatives(self, local_fields):
        return self.__derivative_function(local_fields)

    @staticmethod
    def __sigmoid_derivative(v):
        a = 1 / (1 + np.exp(-v))
        return a * (1 - a)
```

Simple Neural Network

```
class NeuralNetwork:
    def __init__(self,
                 data_x: np.ndarray,
                 data_y: np.ndarray,
                 test_x: np.ndarray,
                 test_y: np.ndarray,
                 hidden_layers: tuple,
                 learning_rate: float,
                 lr_decay_factor: float = 0.9):
        self.data = data_x
        self.labels = data_y

        self.test_x = test_x
        self.test_y = test_y

        self.learning_rate = learning_rate
        self.decay_factor = lr_decay_factor

        self.n_features = self.data.shape[0]
        self.n_outputs = self.labels.shape[0]

        self.n_samples = self.data.shape[1]

        # Layers of neural network
```

```
        self.nn_layers = list()
        self.nn_layers.append(Layer(self.n_features, hidden_layers[0]['num_nodes'], hidden_layers[0]['activation']))
        for i in range(1, len(hidden_layers)):
            self.nn_layers.append(Layer(hidden_layers[i - 1]['num_nodes'], hidden_layers[i]['num_nodes'],
                                        hidden_layers[i]['activation']))
        self.nn_layers.append(Layer(hidden_layers[-1]['num_nodes'], self.n_outputs, 'sigmoid'))

    def calc_stats(self, data, labels):
        predictions = self.predict(data)
        mse = np.sum(((predictions - labels) ** 2) / labels.shape[1])
        acc = accuracy_score(np.argmax(labels, axis=0), np.argmax(predictions, axis=0))
        return mse, acc

    def predict(self, data):
        local_fields, activations = self.__forward(data)
        return activations[-1]

    def __forward(self, x_i):
        local_fields = list()
        activations = list()
        current_input = x_i
        for layer in self.nn_layers:
            z = layer.local_fields(current_input)
            local_fields.append(z)
            a = layer.activations(z)
            activations.append(a)
            current_input = a
        return local_fields, activations

    def __backward(self, initial_delta, local_fields):
        current_delta = initial_delta
        layer_delta = list()
        for i in reversed(range(len(self.nn_layers))):
            if i == len(self.nn_layers) - 1:
                delta = current_delta * self.nn_layers[i].derivatives(local_fields[i])
            else:
                delta = self.nn_layers[i + 1].weights.dot(current_delta) * self.nn_layers[i].derivatives(
                    local_fields[i])
            layer_delta.insert(0, delta)
            current_delta = delta
        return layer_delta

    def __update_layer_params(self, x_i, layer_delta, activations):
        current_input = x_i
        for layer, activation, delta in zip(self.nn_layers, activations, layer_delta):
            layer.weights = layer.weights - self.learning_rate * current_input.dot(delta.T)
            layer.biases = layer.biases - self.learning_rate * delta
            current_input = activation

    def train(self):
        train_epoch_stats = list()
        test_epoch_stats = list()
        train_mse, train_acc = self.calc_stats(self.data, self.labels)
        train_epoch_stats.append([0, train_mse, train_acc])
        test_mse, test_acc = self.calc_stats(self.test_x, self.test_y)
        test_epoch_stats.append([0, test_mse, test_acc])
        epoch_cnt = 1
        while test_epoch_stats[-1][2] < 0.955:
            start_time = time.time()
            for i in range(self.n_samples):
                x_i = self.data[:, i].reshape((self.n_features, 1))
                d_i = self.labels[:, i].reshape((self.n_outputs, 1))
                local_fields, activations = self.__forward(x_i)
                y_i = activations[-1]
                initial_delta = 2 * (y_i - d_i) / self.n_samples
                delta_list = self.__backward(initial_delta, local_fields)
                self.__update_layer_params(x_i, delta_list, activations)
            train_mse, train_acc = self.calc_stats(self.data, self.labels)
            test_mse, test_acc = self.calc_stats(self.test_x, self.test_y)
            print(
                '[Epoch: {}] => Train MSE: {:.4f}, Train Accuracy: {:.4f}, Test Accuracy: {:.4f}, Epoch Duration: {:.4f} S'.format
                    epoch_cnt, train_mse, train_acc, test_acc, time.time() - start_time))

            if test_acc <= test_epoch_stats[-1][2]:
                self.learning_rate = self.learning_rate * self.decay_factor

            train_epoch_stats.append([epoch_cnt, train_mse, train_acc])
            test_epoch_stats.append([epoch_cnt, test_mse, test_acc])
            epoch_cnt += 1
        return np.array(train_epoch_stats), np.array(test_epoch_stats)
```

```
    def save_params(self, save_path):
        nn_params = list()
        for layer in self.nn_layers:
            nn_params.append({
                'weight': layer.weights,
                'bias': layer.biases
            })
        with open(save_path, 'wb') as model_params:
            pickle.dump(nn_params, model_params)

    def load_params(self, params_path):
        nn_params = pickle.load(open(params_path, 'rb'))
        assert len(self.nn_layers) == len(nn_params)
        for layer, params in zip(self.nn_layers, nn_params):
            layer.weights = params['weight']
            layer.biases = params['bias']
```

Reading Data

```
def read_idx(filename):
    with open(filename, 'rb') as f:
        zero, data_type, dims = struct.unpack('>HBB', f.read(4))
        shape = tuple(struct.unpack('>I', f.read(4))[0] for d in range(dims))
        return np.frombuffer(f.read(), dtype=np.uint8).reshape(shape)
```

```
train_images = read_idx(os.path.join(data_path, 'train-images-idx3-ubyte'))
train_images = train_images.reshape((train_images.shape[0], train_images.shape[1] * train_images.shape[2])).T

train_labels = read_idx(os.path.join(data_path, 'train-labels-idx1-ubyte'))
train_labels = np.eye(10)[train_labels].T

test_images = read_idx(os.path.join(data_path, 't10k-images-idx3-ubyte'))
test_images = test_images.reshape((test_images.shape[0], test_images.shape[1] * test_images.shape[2])).T

test_labels = read_idx(os.path.join(data_path, 't10k-labels-idx1-ubyte'))
test_labels_norm = test_labels.copy()
test_labels = np.eye(10)[test_labels].T
```

Training Process

```
hidden_layer_params = (
    {'num_nodes': 256,
     'activation': 'tanh'},
    {'num_nodes': 16,
     'activation': 'tanh'}
)

nn_regressor = NeuralNetwork(train_images,
                             train_labels,
                             test_images,
                             test_labels,
                             hidden_layers=hidden_layer_params,
                             learning_rate=12,
                             lr_decay_factor=0.7)
train_epoch_stats, test_epoch_stats = nn_regressor.train()
nn_regressor.save_params(os.path.join(root, 'model_params/256_16_sigmoid_params.pkl'))

# nn_regressor.load_params(os.path.join(root, 'model_params/256_16_sigmoid_params.pkl'))
# with open(os.path.join(root, 'model_stats/256_16_sigmoid_stats.pkl'), 'wb') as stats_file:
#     pickle.dump((train_epoch_stats, test_epoch_stats), stats_file)
```

```
[Epoch: 1] => Train MSE: 0.8954, Train Accuracy: 0.2083, Test Accuracy: 0.2112, Epoch Duration: 70.6459 S
[Epoch: 2] => Train MSE: 0.8654, Train Accuracy: 0.2059, Test Accuracy: 0.2081, Epoch Duration: 71.3460 S
[Epoch: 3] => Train MSE: 0.8501, Train Accuracy: 0.2122, Test Accuracy: 0.2106, Epoch Duration: 69.2662 S
[Epoch: 4] => Train MSE: 0.8395, Train Accuracy: 0.2127, Test Accuracy: 0.2110, Epoch Duration: 73.9732 S
[Epoch: 5] => Train MSE: 0.8259, Train Accuracy: 0.2136, Test Accuracy: 0.2116, Epoch Duration: 70.0234 S
[Epoch: 6] => Train MSE: 0.7818, Train Accuracy: 0.3125, Test Accuracy: 0.3109, Epoch Duration: 69.6082 S
[Epoch: 7] => Train MSE: 0.7173, Train Accuracy: 0.5504, Test Accuracy: 0.5507, Epoch Duration: 70.4475 S
[Epoch: 8] => Train MSE: 0.6012, Train Accuracy: 0.6069, Test Accuracy: 0.6019, Epoch Duration: 68.0290 S
[Epoch: 9] => Train MSE: 0.4829, Train Accuracy: 0.7888, Test Accuracy: 0.7924, Epoch Duration: 72.5066 S
[Epoch: 10] => Train MSE: 0.3897, Train Accuracy: 0.8565, Test Accuracy: 0.8570, Epoch Duration: 68.1072 S
[Epoch: 11] => Train MSE: 0.3233, Train Accuracy: 0.8991, Test Accuracy: 0.8964, Epoch Duration: 72.3832 S
[Epoch: 12] => Train MSE: 0.2719, Train Accuracy: 0.9138, Test Accuracy: 0.9123, Epoch Duration: 68.2074 S
[Epoch: 13] => Train MSE: 0.2379, Train Accuracy: 0.9222, Test Accuracy: 0.9162, Epoch Duration: 66.5742 S
[Epoch: 14] => Train MSE: 0.2119, Train Accuracy: 0.9279, Test Accuracy: 0.9279, Epoch Duration: 69.1238 S
[Epoch: 15] => Train MSE: 0.1976, Train Accuracy: 0.9279, Test Accuracy: 0.9273, Epoch Duration: 66.5518 S
[Epoch: 16] => Train MSE: 0.1723, Train Accuracy: 0.9391, Test Accuracy: 0.9376, Epoch Duration: 70.3178 S
[Epoch: 17] => Train MSE: 0.1588, Train Accuracy: 0.9438, Test Accuracy: 0.9384, Epoch Duration: 66.7519 S
[Epoch: 18] => Train MSE: 0.1509, Train Accuracy: 0.9457, Test Accuracy: 0.9447, Epoch Duration: 70.9285 S
[Epoch: 19] => Train MSE: 0.1420, Train Accuracy: 0.9479, Test Accuracy: 0.9426, Epoch Duration: 67.6233 S
[Epoch: 20] => Train MSE: 0.1298, Train Accuracy: 0.9539, Test Accuracy: 0.9450, Epoch Duration: 65.8898 S
[Epoch: 21] => Train MSE: 0.1270, Train Accuracy: 0.9543, Test Accuracy: 0.9485, Epoch Duration: 67.8939 S
[Epoch: 22] => Train MSE: 0.1187, Train Accuracy: 0.9575, Test Accuracy: 0.9503, Epoch Duration: 65.7243 S
[Epoch: 23] => Train MSE: 0.1173, Train Accuracy: 0.9576, Test Accuracy: 0.9508, Epoch Duration: 68.5207 S
[Epoch: 24] => Train MSE: 0.1121, Train Accuracy: 0.9602, Test Accuracy: 0.9530, Epoch Duration: 65.5310 S
[Epoch: 25] => Train MSE: 0.1091, Train Accuracy: 0.9597, Test Accuracy: 0.9522, Epoch Duration: 70.2115 S
[Epoch: 26] => Train MSE: 0.1033, Train Accuracy: 0.9631, Test Accuracy: 0.9528, Epoch Duration: 65.8000 S
[Epoch: 27] => Train MSE: 0.0988, Train Accuracy: 0.9650, Test Accuracy: 0.9548, Epoch Duration: 65.7438 S
[Epoch: 28] => Train MSE: 0.0955, Train Accuracy: 0.9666, Test Accuracy: 0.9559, Epoch Duration: 66.9296 S
```

Predicting random samples from test data just to make sure

```
random_indices = np.random.randint(0, 10000, size=(100,))
test_preds = nn_regressor.predict(test_images[:, random_indices])
test_preds = np.argmax(test_preds, axis=0)
print('Test accuracy: {:.4f}'.format(accuracy_score(test_labels_norm[random_indices], test_preds)))
```

```
Test accuracy: 0.9600
```

Network Architecture

The network contains two hidden layers with 256 and 16 neurons respectively. Activation function for hidden layers is 'tanh' while the output layer uses a 'sigmoid' activation.

Output Representation

The output is represented as one-hot encoded vectors corresponding to a single digit.
For example, the digit 3 is represented as [0, 0, 0, 1, 0, 0, 0, 0, 0, 0] and 9 is represented as [0, 0, 0, 0, 0, 0, 0, 0, 0, 1]

Activation and Learning Rate

Hidden layer neurons use 'tanh' activation while output layer uses 'sigmoid' activation.
The initial learning rate is 12 for all neurons and it is decayed by a factor of 0.7 when there is a decrease in test accuracy from the previous epoch.

Energy Function

The energy function used is the Mean Squared Error because of its convexity. As a result, there is only one global optimum, a smooth loss landscape and gradients can be found easily.
$\frac{1}{n}\sum_{i=1}^{n}(d_i - y_i)^2$

Hyperparameters

Hyperparameter	Value	Reason
Learning rate	12	Moderately large learning rates can help faster convergence in the initial training steps. Such learning rates are usually used in combination with a decay rate less than 1
Decay rate	0.7	Learning rate is decayed at a rate of 0.7 so as to avoid divergence

Number of hidden layers	2	Having more hidden layer increases neural network complexity allowing it to represent more complex functions. Our application requires only moderate complexity and thus 2 hidden layers. This also helps avoid drastic reduction in the size of inputs to subsequent layers.
Number of Hidden layer Neurons	Powers of 2 256 and 16	Having powers-of-two neurons is a general recommendation. Some sources state this helps easier compiler optimizations.
Activations	tanh, tanh and sigmoid	In Efficient BackProp the authors argue that convergence is faster if the average of inputs to each layer is close to zero. Hyperbolic tangent function has this property where the outputs ∈ (-1, 1) thus pushing the average close to zero. The output activation is sigmoid because it more closely represents probabilities. However, tanh also gives good performance.

Design Details

Hidden Layers - Initially the network contained one hidden layer with 64 neurons and tanh activations for all layers. However, this network plateaued at 90% training accuracy indicating a more complex model could perform better.
Learning Rates - A couple of learning rates have also been tried allowing me to conclude lower learning rates caused the initial convergence to be slow thus taking large number of epochs to converge. Learning rates greater than 20 caused divergence
Decay Rates - Higher decay rates cause the network to oscillate around a single point (error) while lower decay rates resulted in slow convergence.
Activation Functions - Tanh activation for output layer resulted in better performance and higher convergence rate. However, sigmoid activation allowed the network to train in a more stable manner. Also, sigmoid is a better representation for probablities than a tanh.

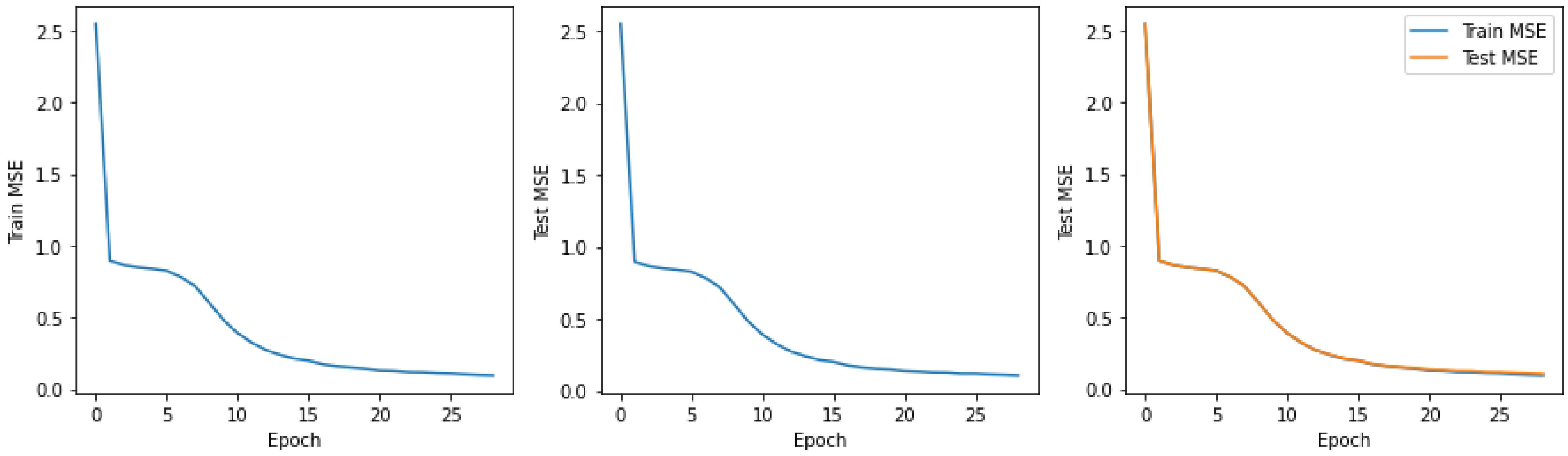
Final Design
Two hidden layers with 256 and 16 neurons respectively and tanh activation. Output layer with sigmoid activation. Initial learning rate of 12 and a decay factor of 0.7.

Plots

MSE Plots

```
fig = plt.figure(figsize=(15, 4))
plt.subplot(1, 3, 1)
plt.plot(train_epoch_stats[:, 0], train_epoch_stats[:, 1])
plt.xlabel('Epoch')
plt.ylabel('Train MSE')
plt.subplot(1, 3, 2)
plt.plot(test_epoch_stats[:, 0], test_epoch_stats[:, 1])
plt.xlabel('Epoch')
plt.ylabel('Test MSE')
plt.subplot(1, 3, 3)
plt.plot(train_epoch_stats[:, 0], train_epoch_stats[:, 1], label='Train MSE')
plt.plot(test_epoch_stats[:, 0], test_epoch_stats[:, 1], label='Test MSE')
plt.legend(loc='best')
plt.xlabel('Epoch')
plt.ylabel('Test MSE')
```

```
Text(0, 0.5, 'Test MSE')
```

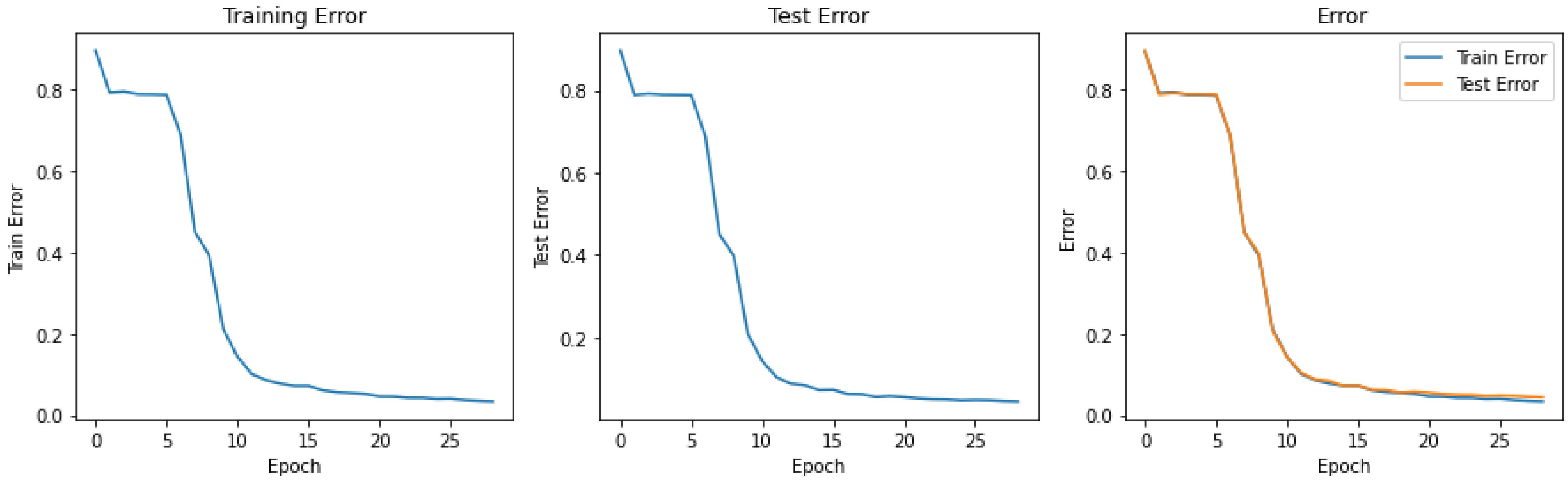


Error Plots

```
fig = plt.figure(figsize=(15, 4))
plt.subplot(1, 3, 1)
plt.plot(train_epoch_stats[:, 0], 1-train_epoch_stats[:, 2])
plt.xlabel('Epoch')
plt.ylabel('Train Error')
plt.title('Training Error')
plt.subplot(1, 3, 2)
```

```
plt.subplot(1, 3, 2)
plt.plot(test_epoch_stats[:, 0], 1-test_epoch_stats[:, 2])
plt.xlabel('Epoch')
plt.ylabel('Test Error')
plt.title('Test Error')
plt.subplot(1, 3, 3)
plt.plot(train_epoch_stats[:, 0], 1-train_epoch_stats[:, 2], label='Train Error')
plt.plot(test_epoch_stats[:, 0], 1-test_epoch_stats[:, 2], label='Test Error')
plt.legend(loc='best')
plt.xlabel('Epoch')
plt.ylabel('Error')
plt.title('Error')
```

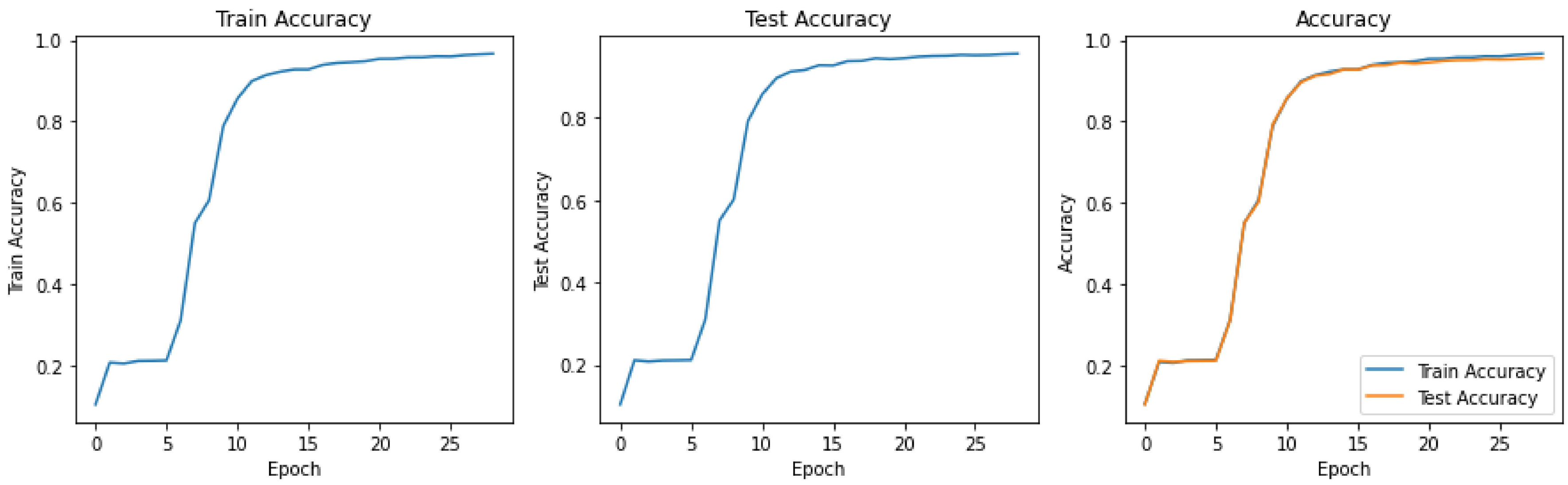
```
Text(0.5, 1.0, 'Error')
```



Accuracy Plots

```
fig = plt.figure(figsize=(15, 4))
plt.subplot(1, 3, 1)
plt.plot(train_epoch_stats[:, 0], train_epoch_stats[:, 2])
plt.xlabel('Epoch')
plt.ylabel('Train Accuracy')
plt.title('Train Accuracy')
plt.subplot(1, 3, 2)
plt.plot(test_epoch_stats[:, 0], test_epoch_stats[:, 2])
plt.xlabel('Epoch')
plt.ylabel('Test Accuracy')
plt.title('Test Accuracy')
plt.subplot(1, 3, 3)
plt.plot(train_epoch_stats[:, 0], train_epoch_stats[:, 2], label='Train Accuracy')
plt.plot(test_epoch_stats[:, 0], test_epoch_stats[:, 2], label='Test Accuracy')
plt.legend(loc='best')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.title('Accuracy')
```

```
Text(0.5, 1.0, 'Accuracy')
```



Pseudocode

while test_accuracy < 0.955:

for each training_example:

for each layer i:

$v^i = {weights^i}^T \cdot activation^{i-1}$ $\{activation^{i-1}$ for first layer is the train example$\}$

$a^i = f^i(v^i)$

(store v and a in an array)
for each layer i:
$\delta^i = (weights^{i+1} \cdot \delta^{i+1}) * f^{i'}(v^i)$
$weights^i \leftarrow weights^i - \eta * (activation^{i-1} \cdot \delta^i)$
$bias^i \leftarrow bias^i - \eta * \delta^i$
mse = $\frac{1}{n}\sum(d_i - y_i)^2$
test_accuracy = $\frac{num\_correct\_test}{num\_samples\_test}$
if test_accuracy < test_accuracy_prev:
$\eta \leftarrow \eta * decay\_factor$

Where,

v^i $\rightarrow$ Local field values of i^{th} layer
w^i, b^i $\rightarrow$ Weights and bias respectively for layer i
δ^i $\rightarrow$ Delta or error w.r.t layer i (Derivative of cost function w.r.t to layer i)
f^i $\rightarrow$ Activation function of layer i
$f^{i'}$ $\rightarrow$ Derivarive of activation function of layer i
a^i $\rightarrow$ Activation of layer i
$\eta \rightarrow$ Learning rate
$\cdot \rightarrow$ Dot product
$* \rightarrow$ Element-wise multiplication