Task 1: Neural Network for Handwritten Digit Recognition.

```
from scipy.io import loadmat
import numpy as np
import scipy.optimize as opt
import pandas as pd
import matplotlib.pyplot as plt
# reading the data
data = loadmat('ex4data1.mat')
X = data['X']
y = data['y']
# visualizing the data
#import matplotlib.image as mpimg
fig, axis = plt.subplots(10,10,figsize=(8,8))
for i in range(10):
    for j in range(10):
        axis[i,j].imshow(X[np.random.randint(0,5001),:].reshape(20,20,order="F"), cmap="hot")
        axis[i,j].axis("off")

plt.show()

# Loading already trained parameters
mat2=loadmat("ex4weights.mat")
Theta1=mat2["Theta1"] # Theta1 has size 25 x 401
Theta2=mat2["Theta2"] # Theta2 has size 10 x 26
print(Theta1)
print("*****Above is Theta1  and below one is Theta2*******")
print(Theta2)

# Forward propagation
def sigmoid(z):
    """
    return the sigmoid of z
    """

    return 1 / (1 + np.exp(-z))
```

```python
def predict(Theta1, Theta2, X):
    """
    Predict the label of an input given a trained neural network
    """
    m = X.shape[0]
    X = np.hstack((np.ones((m, 1)), X))

    a1 = sigmoid(X @ Theta1.T)
    a1 = np.hstack((np.ones((m, 1)), a1))  # hidden layer
    a2 = sigmoid(a1 @ Theta2.T)  # output layer

    return np.argmax(a2, axis=1) + 1

pred2 = predict(Theta1, Theta2, X)
print("Training Set Accuracy:",sum(pred2[:,np.newaxis]==y)[0]/5000*100,"%")
```

```python
# Compute Neural Network Cost Function

def nnCostFunction(nn_params, input_layer_size, hidden_layer_size, num_labels, X, y, Lambda):
    """
    nn_params contains the parameters unrolled into a vector

    compute the cost and gradient of the neural network
    """
    # Reshape nn_params back into the parameters Theta1 and Theta2
    Theta1 = nn_params[:((input_layer_size + 1) * hidden_layer_size)].reshape(hidden_layer_size, input_layer_size + 1)
    Theta2 = nn_params[((input_layer_size + 1) * hidden_layer_size):].reshape(num_labels, hidden_layer_size + 1)

    m = X.shape[0]
    J = 0
    X = np.hstack((np.ones((m, 1)), X))
    y10 = np.zeros((m, num_labels))

    a1 = sigmoid(X @ Theta1.T)
    a1 = np.hstack((np.ones((m, 1)), a1))  # hidden layer
    a2 = sigmoid(a1 @ Theta2.T)  # output layer

    for i in range(1, num_labels + 1):
        y10[:, i - 1][:, np.newaxis] = np.where(y == i, 1, 0)
    for j in range(num_labels):
        J = J + sum(-y10[:, j] * np.log(a2[:, j]) - (1 - y10[:, j]) * np.log(1 - a2[:, j]))

    cost = 1 / m * J
    reg_J = cost + Lambda / (2 * m) * (np.sum(Theta1[:, 1:] ** 2) + np.sum(Theta2[:, 1:] ** 2))
```

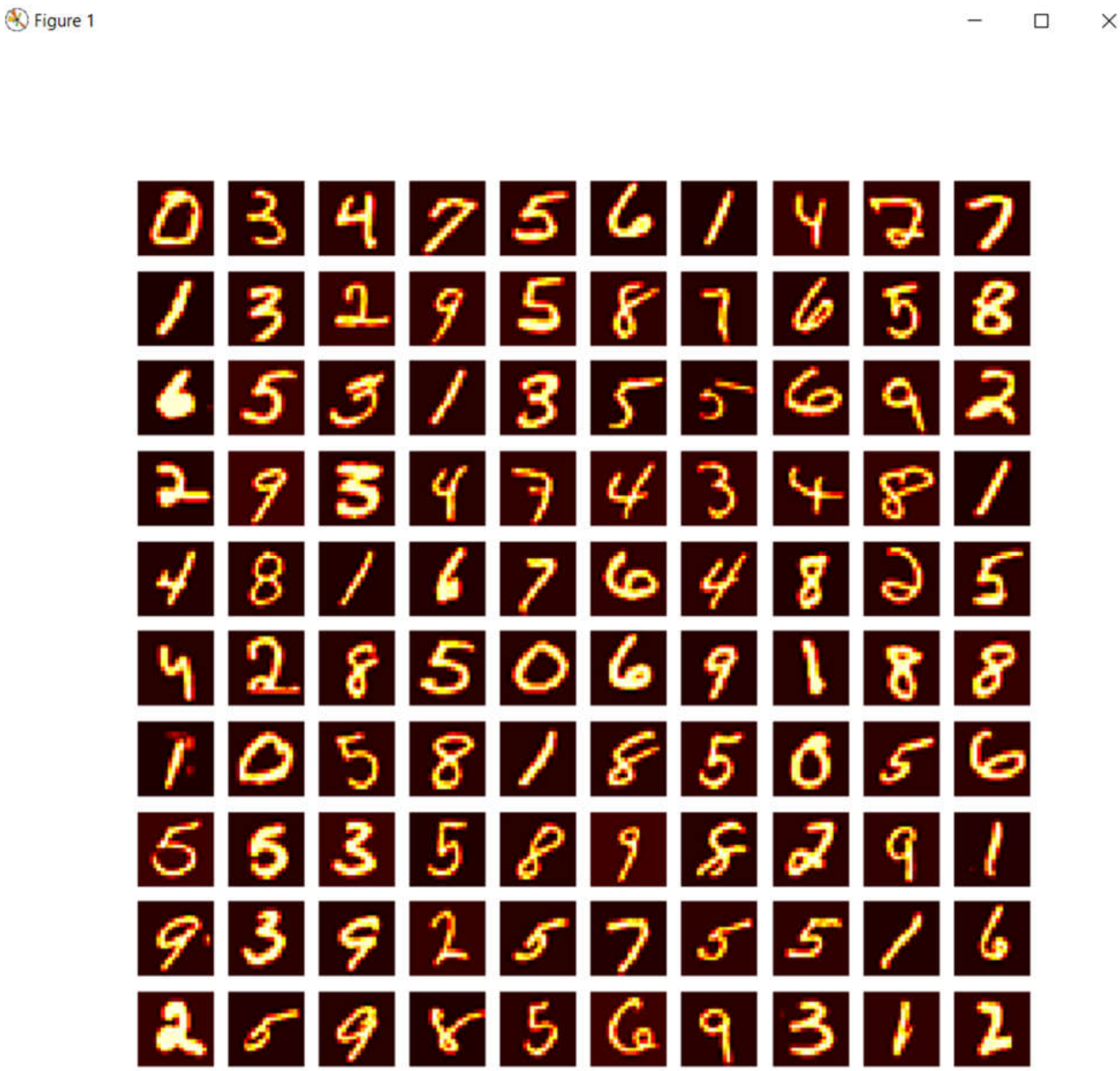
Source code credit: B. Lau

```python
    # Implement the backpropagation algorithm to compute the gradients

    grad1 = np.zeros((Theta1.shape))
    grad2 = np.zeros((Theta2.shape))

    for i in range(m):
        xi = X[i, :]  # 1 X 401
        a1i = a1[i, :]  # 1 X 26
        a2i = a2[i, :]  # 1 X 10
        d2 = a2i - y10[i, :]
        d1 = Theta2.T @ d2.T * sigmoidGradient(np.hstack((1, xi @ Theta1.T)))
        grad1 = grad1 + d1[1:][:, np.newaxis] @ xi[:, np.newaxis].T
        grad2 = grad2 + d2.T[:, np.newaxis] @ a1i[:, np.newaxis].T

    grad1 = 1 / m * grad1
    grad2 = 1 / m * grad2

    grad1_reg = grad1 + (Lambda / m) * np.hstack((np.zeros((Theta1.shape[0], 1)), Theta1[:, 1:]))
    grad2_reg = grad2 + (Lambda / m) * np.hstack((np.zeros((Theta2.shape[0], 1)), Theta2[:, 1:]))

    return cost, grad1, grad2, reg_J, grad1_reg, grad2_reg

# backprop Starting
def sigmoidGradient(z):
    """
    computes the gradient of the sigmoid function
    """
    sigmoid = 1 / (1 + np.exp(-z))

    return sigmoid * (1 - sigmoid)

# Neural Network Hyper Parameters.
input_layer_size  = 400
hidden_layer_size = 25
num_labels = 10
nn_params = np.append(Theta1.flatten(),Theta2.flatten())
J,reg_J = nnCostFunction(nn_params, input_layer_size, hidden_layer_size, num_labels, X, y, 1)[0:4:3]
print("Cost at parameters (non-regularized):",J,"\nCost at parameters (Regularized):",reg_J)
```

Source code credit: B. Lau

```python
# Random Initialization
def randInitializeWeights(L_in, L_out):
    """
    randomly initializes the weights of a layer with L_in incoming connections and L_out outgoing connections.
    """

    epi = (6 ** 1 / 2) / (L_in + L_out) ** 1 / 2

    W = np.random.rand(L_out, L_in + 1) * (2 * epi) - epi

    return W

initial_Theta1 = randInitializeWeights(input_layer_size, hidden_layer_size)
initial_Theta2 = randInitializeWeights(hidden_layer_size, num_labels)
initial_nn_params = np.append(initial_Theta1.flatten(),initial_Theta2.flatten())


debug_J  = nnCostFunction(nn_params, input_layer_size, hidden_layer_size, num_labels, X, y, 3)
print("Cost at (fixed) debugging parameters (w/ lambda = 3):",debug_J[3])

# Learning Parameters through Gradient Descent
def gradientDescentnn(X, y, initial_nn_params, alpha, num_iters, Lambda, input_layer_size, hidden_layer_size,
                      num_labels):
    """
    Take in numpy array X, y and theta and update theta by taking num_iters gradient steps
    with learning rate of alpha

    return theta and the list of the cost of theta during each iteration
    """
    Theta1 = initial_nn_params[:((input_layer_size + 1) * hidden_layer_size)].reshape(hidden_layer_size,
                                                                                       input_layer_size + 1)
    Theta2 = initial_nn_params[((input_layer_size + 1) * hidden_layer_size):].reshape(num_labels, hidden_layer_size + 1)

    m = len(y)
    J_history = []

    for i in range(num_iters):
        nn_params = np.append(Theta1.flatten(), Theta2.flatten())
        cost, grad1, grad2 = nnCostFunction(nn_params, input_layer_size, hidden_layer_size, num_labels, X, y, Lambda)[
                             3:]
        Theta1 = Theta1 - (alpha * grad1)
        Theta2 = Theta2 - (alpha * grad2)
        J_history.append(cost)
```

```
    nn_paramsFinal = np.append(Theta1.flatten(), Theta2.flatten())
    return nn_paramsFinal, J_history

nnTheta, nnJ_history = gradientDescentnn(X,y,initial_nn_params,0.8,800,1,input_layer_size, hidden_layer_size, num_labels)
Theta1 = nnTheta[:((input_layer_size+1) * hidden_layer_size)].reshape(hidden_layer_size,input_layer_size+1)
Theta2 = nnTheta[((input_layer_size +1)* hidden_layer_size ):].reshape(num_labels,hidden_layer_size+1)

pred3 = predict(Theta1, Theta2, X)
print("Training Set Accuracy:",sum(pred3[:,np.newaxis]==y)[0]/5000*100,"%")
```



Task 2: You need to find out the working of various functions (from different libraries) used in the above program.