```python
def forward_prop(x, y, weightsAndBiases):
    Ws, bs = unpack(weightsAndBiases)
    num_examples = x.shape[0]
    zs = []
    hs = []
    h = x.T
    hs.append(h)
    for i in range(NUM_HIDDEN_LAYERS):
        z = np.dot(Ws[i], h) + bs[i].reshape(-1, 1)
        zs.append(z)
        h = np.maximum(0, z)
        hs.append(h)
    z = np.dot(Ws[-1], h) + bs[-1].reshape(-1, 1)
    zs.append(z)
    yhat = np.exp(z) / np.sum(np.exp(z), axis=0)
    hs.append(yhat)
    loss = -np.sum(y * np.log(yhat.T)) / num_examples
    return loss, zs, hs, yhat

def back_prop(x, y, weightsAndBiases):
    loss, zs, hs, yhat = forward_prop(x, y, weightsAndBiases)
    num_examples = x.shape[0]
    Ws, bs = unpack(weightsAndBiases)
    dJdWs = []
    dJdbs = []
    dJdz_output = yhat - y.T
    dJdWs_output = np.dot(dJdz_output, hs[-2].T) / num_examples
    dJdbs_output = np.sum(dJdz_output, axis=1) / num_examples
    dJdWs.append(dJdWs_output)
    dJdbs.append(dJdbs_output)
    for i in range(NUM_HIDDEN_LAYERS - 1, -1, -1):
        dJdh = np.dot(Ws[i + 1].T, dJdz_output)
        dJdz = dJdh * (zs[i] > 0)
        dJdWs_hidden = np.dot(dJdz, hs[i].T) / num_examples
        dJdbs_hidden = np.sum(dJdz, axis=1) / num_examples
        dJdWs.insert(0, dJdWs_hidden)
        dJdbs.insert(0, dJdbs_hidden)
        dJdz_output = dJdz
    return np.hstack([ dJdW.flatten() for dJdW in dJdWs ] + [ dJdb.flatten() for dJdb in dJdbs ])

def train(trainX, trainY, weightsAndBiases, testX, testY):
    NUM_EPOCHS = 100
    trajectory = []
    for epoch in range(NUM_EPOCHS):
        for i in range(trainX.shape[0]):
            x = np.atleast_2d(trainX[i])
            y = np.atleast_2d(trainY[i])
            gradients = back_prop(x, y, weightsAndBiases)
            weightsAndBiases -= gradients
            trajectory.append(weightsAndBiases.copy())
    return weightsAndBiases, trajectory
```

```
100
101 def back_prop(x, y, weightsAndBiases):
102     loss, zs, hs, yhat = forward_prop(x, y, weightsAndBiases)
103     num_examples = x.shape[0]
104     Ws, bs = unpack(weightsAndBiases)
105     dJdWs = []
106     dJdbs = []
107     dJdz_output = yhat - y.T
108     dJdWs_output = np.dot(dJdz_output, hs[-2].T) / num_examples
109     dJdbs_output = np.sum(dJdz_output, axis=1) / num_examples
110     dJdWs.append(dJdWs_output)
111     dJdbs.append(dJdbs_output)
112     for i in range(NUM_HIDDEN_LAYERS - 1, -1, -1):
113         dJdh = np.dot(Ws[i + 1].T, dJdz_output)
114         dJdz = dJdh * (zs[i] > 0)
115         dJdWs_hidden = np.dot(dJdz, hs[i].T) / num_examples
116         dJdbs_hidden = np.sum(dJdz, axis=1) / num_examples
117         dJdWs.insert(0, dJdWs_hidden)
118         dJdbs.insert(0, dJdbs_hidden)
119         dJdz_output = dJdz
120     return np.hstack([ dJdW.flatten() for dJdW in dJdWs ] + [ dJdb.flatten() for dJdb in dJdbs ])
121
122 def train(trainX, trainY, weightsAndBiases, testX, testY):
123     NUM_EPOCHS = 100
124     trajectory = []
125     for epoch in range(NUM_EPOCHS):
126         for i in range(trainX.shape[0]):
127             x = np.atleast_2d(trainX[i])
128             y = np.atleast_2d(trainY[i])
129             gradients = back_prop(x, y, weightsAndBiases)
130             weightsAndBiases -= gradients
131             trajectory.append(weightsAndBiases.copy())
132     return weightsAndBiases, trajectory
133
134 # problem_1_a_b
135 # Initialize weights and biases randomly
136 weightsAndBiases = initWeightsAndBiases()
137
138 # Perform gradient check on 5 training examples
139 print(scipy.optimize.check_grad(lambda wab: forward_prop(np.atleast_2d(trainX[0:5,:]), np.atleast_2d(trainY[0:5,:]), wab)[0], \
140                                 lambda wab: back_prop(np.atleast_2d(trainX[0:5,:]), np.atleast_2d(trainY[0:5,:]), wab), \
141                                 weightsAndBiases))
```

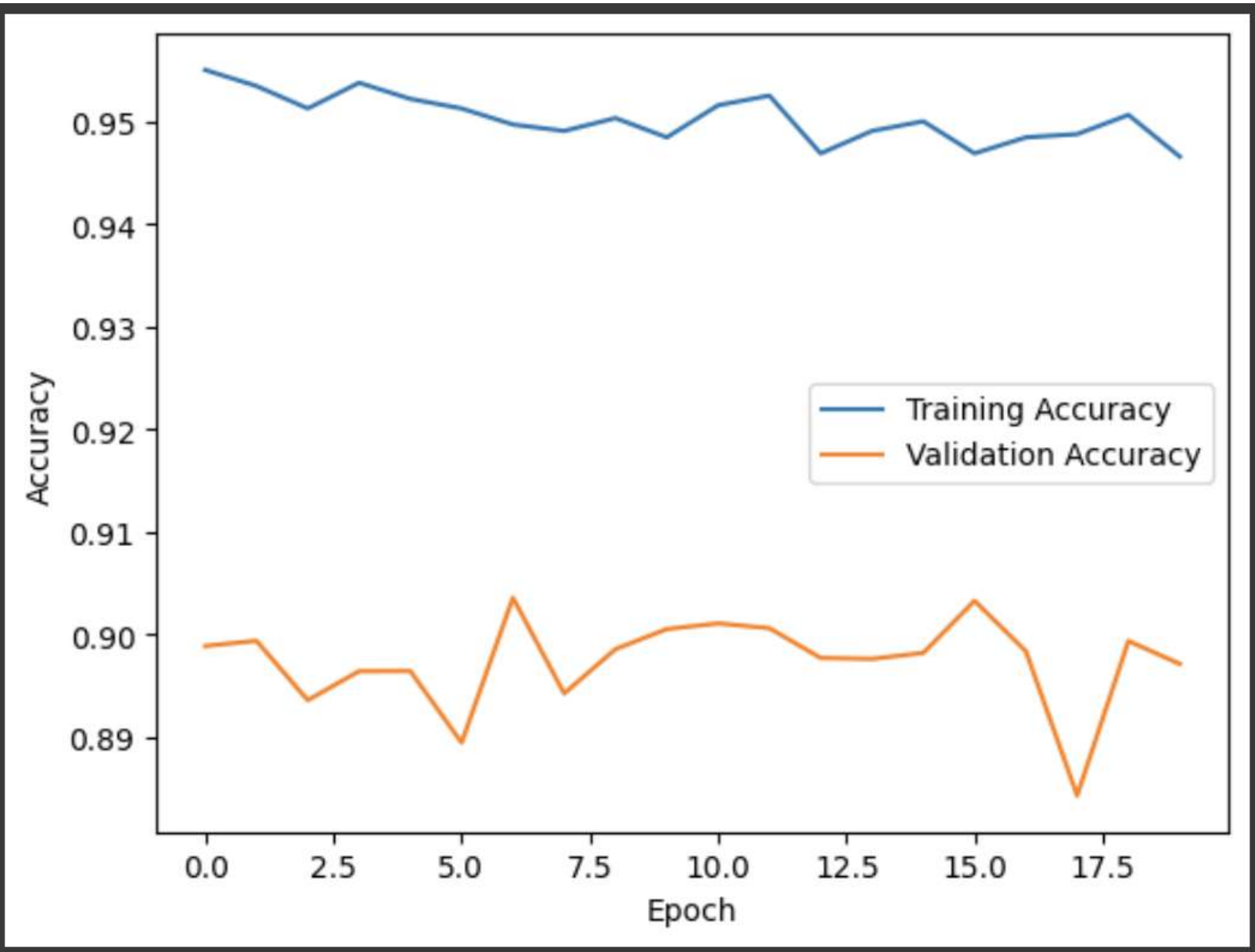
```
1.4577590509876156e-06
```

```python
def problem_2_a():
    NUM_HIDDEN_LAYERS = 3
    LEARNING_RATE = 0.001
    MINIBATCH_SIZE = 128
    NUM_EPOCHS = 30
    L2_REGULARIZATION_STRENGTH = 0.1
    NUM_INPUT = 784
    NUM_HIDDEN = 10
    NUM_OUTPUT = 10
    model = Sequential()
    model.add(Dense(NUM_HIDDEN, input_shape=(NUM_INPUT,), activation='relu', kernel_regularizer=regularizers.l2(L2_REGULARIZATION_STRENGTH)))
    for i in range(NUM_HIDDEN_LAYERS - 1):
        model.add(Dense(NUM_HIDDEN, activation='relu', kernel_regularizer=regularizers.l2(L2_REGULARIZATION_STRENGTH)))
    model.add(Dense(NUM_OUTPUT, activation='softmax'))
    model.compile(loss='categorical_crossentropy', optimizer=tf.keras.optimizers.SGD(learning_rate=LEARNING_RATE), metrics=['accuracy'])
    history = model.fit(trainX, trainY, batch_size=MINIBATCH_SIZE, epochs=NUM_EPOCHS, verbose=2, validation_data=(valX, valY))
    test_loss, test_accuracy = model.evaluate(testX, testY)
    print(f'Test Loss: {test_loss:.4f}, Test Accuracy: {test_accuracy:.2%}')
    plt.plot(history.history['accuracy'], label='Training Accuracy')
    plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
    plt.xlabel('Epoch')
    plt.ylabel('Accuracy')
    plt.legend()
    plt.show()
```

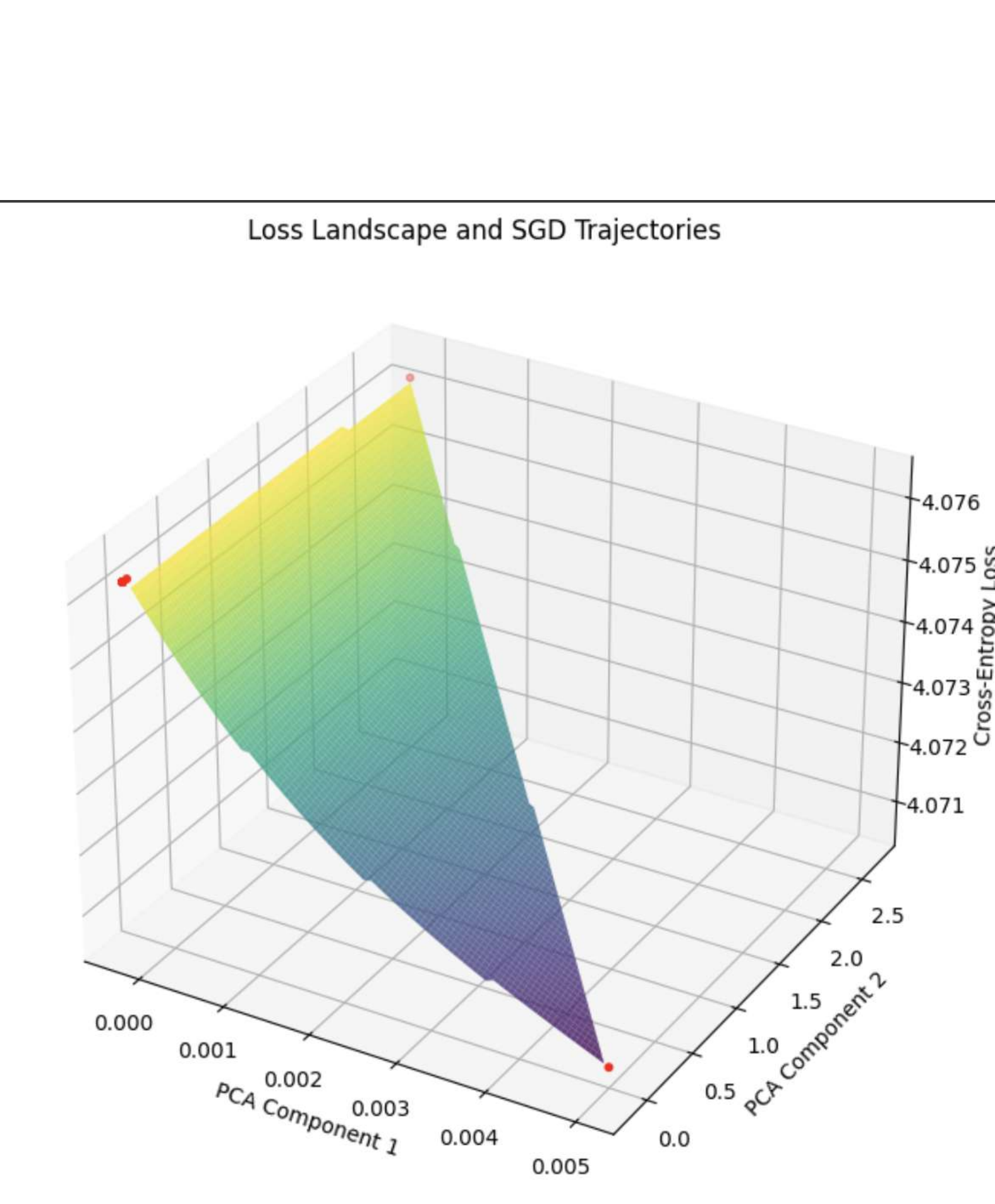
```python
def findBestHyperparameters(trainX, trainY, testX, testY, valX, valY):
    grid_search_num_epochs_values : list = [60, 80, 100, 120, 140, 160, 180, 200]
    grid_search_minibatch_size_values : list = [16, 32, 64, 80, 100, 128, 160, 180, 200, 256,]
    grid_search_learning_rate_values : list = [5e-2, 3e-1, 0.001, 0.005, 0.01, 0.05, 0.1, 0.5]
    grid_search_l2_regularization_strength_values : list = [0.001, 0.01, 0.1, 1.0]
    grid_search_num_hidden_layers_values : list = [3,4,5,6]
    grid_search_num_units_per_layer_values : list = [10,20,30,40,50]
    best_accuracy = 0.0
    best_hyperparameters = {}
    for MINIBATCH_SIZE in grid_search_minibatch_size_values:
        for LEARNING_RATE in grid_search_learning_rate_values:
            for L2_REGULARIZATION_STRENGTH in grid_search_l2_regularization_strength_values:
                for NUM_HIDDEN_LAYERS in grid_search_num_hidden_layers_values:
                    for NUM_HIDDEN in grid_search_num_units_per_layer_values:
                        for NUM_EPOCHS in grid_search_num_epochs_values:
                            model = Sequential()
                            model.add(Dense(NUM_HIDDEN, input_shape=(784,), activation='relu', kernel_regularizer=regularizers.l2(L2_REGULARIZATION_STRENGTH)))
                            for _ in range(NUM_HIDDEN_LAYERS - 1):
                                model.add(Dense(NUM_HIDDEN, activation='relu', kernel_regularizer=regularizers.l2(L2_REGULARIZATION_STRENGTH)))
                            model.add(Dense(10, activation='softmax'))
                            model.compile(loss='categorical_crossentropy', optimizer=tf.keras.optimizers.SGD(learning_rate=LEARNING_RATE), metrics=['accuracy'])
                            history = model.fit(trainX, trainY, batch_size=MINIBATCH_SIZE, epochs=NUM_EPOCHS, verbose=0, validation_data=(valX, valY))
                            test_loss, test_accuracy = model.evaluate(testX, testY)
                            print(f"Test Loss: {test_loss:.4f}, Test Accuracy: {test_accuracy:.2%}")
                            if history.history['val_accuracy'][-1] > best_accuracy:
                                best_accuracy = history.history['val_accuracy'][-1]
                                best_hyperparameters = {
                                    'EPOCH': NUM_EPOCHS,
                                    'MINIBATCH_SIZE': MINIBATCH_SIZE,
                                    'LEARNING_RATE': LEARNING_RATE,
                                    'L2_REGULARIZATION_STRENGTH': L2_REGULARIZATION_STRENGTH,
                                    'NUM_HIDDEN_LAYERS': NUM_HIDDEN_LAYERS,
                                    'NUM_HIDDEN': NUM_HIDDEN,
                                    'TEST_LOSS': test_loss,
                                    'VAL_LOSS': best_accuracy,
                                }
    print(f'Best Hyperparameters : {best_hyperparameters}, Best Validation Accuracy : {best_accuracy * 100:.2f}%')
    return None
```

```
133
134  # problem_1_a_b
135  # Initialize weights and biases randomly
136  weightsAndBiases = initWeightsAndBiases()
137
138  # Perform gradient check on 5 training examples
139  print(scipy.optimize.check_grad(lambda wab: forward_prop(np.atleast_2d(trainX[0:5,:]), np.atleast_2d(trainY[0:5,:]), wab)[0], \
140                                  lambda wab: back_prop(np.atleast_2d(trainX[0:5,:]), np.atleast_2d(trainY[0:5,:]), wab), \
141                                  weightsAndBiases))
```

```
1.4577590509876156e-06
```




```
# Final Hyperparameters
NUM_HIDDEN_LAYERS = 3
LEARNING_RATE = 3e-3
MINIBATCH_SIZE = 64
NUM_EPOCHS = 600
L2_REGULARIZATION_STRENGTH = 2e-3
NUM_INPUT = 784
NUM_HIDDEN = 40
NUM_OUTPUT = 10
```


Loss Landscape and SGD Trajectories
4.076
4.075
4.074
4.073
4.072
4.071
Cross-Entropy Loss
0.000
0.001
0.002
0.003
0.004
0.005
PCA Component 1
0.0
0.5
1.0
1.5
2.0
2.5
PCA Component 2