BTT004_week7_Assignment-solution

June 10, 2024

1 Assignment 7: Implement a Neural Network Using Keras

[1]:
```
import pandas as pd
import numpy as np
import os
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix
import tensorflow.keras as keras
import time
```

In this assignment, you will implement a feedforward neural network using Keras for a binary classification problem. You will complete the following tasks:

1. Build your DataFrame and define your ML problem:
    - Load the Airbnb "listings" data set
    - Define the label - what are you predicting?
    - Identify the features
2. Prepare your data so that it is ready for modeling.
3. Create labeled examples from the data set.
4. Split the data into training and test data sets.
5. Construct a neural network.
6. Train the neural network.
7. Evaluate the neural network model's performance on the training, validation and test data.
8. Experiment with ways to improve the model's performance.

For this assignment, use the demo Implementing a Neural Network in Keras that is contained in this unit as a reference.

Note: some of the code cells in this notebook may take a while to run

1.1 Part 1. Build Your DataFrame and Define Your ML Problem

Load a Data Set and Save it as a Pandas DataFrame We will work with the data set `airbnbData_train`.

Task: In the code cell below, use the same method you have been using to load the data using `pd.read_csv()` and save it to DataFrame `df`.

You will be working with the file named "airbnbData_train.csv" that is located in a folder named "data_NN".

[2]:
```
# YOUR CODE HERE

#SOLUTION:

filename = os.path.join(os.getcwd(), "data_NN", "airbnbData_train.csv")
df = pd.read_csv(filename, header=0)
```

Define the Label Your goal is to train a machine learning model that predicts whether an Airbnb host is a 'super host'. This is an example of supervised learning and is a binary classification problem. In our dataset, our label will be the `host_is_superhost` column and the label will either contain the value `True` or `False`.

Identify Features Our features will be all of the remaining columns in the dataset.

1.2 Part 2. Prepare Your Data

Many data preparation techniques have already been performed and the data is almost ready for modeling; the data set has one-hot encoded categorical variables, scaled numerical values, and imputed missing values. However, the data set has a few features that have boolean values. When working with Keras, features should have floating point values.

Let's convert these features from booleans to floats.

Task: Using the Pandas `astype()` method, convert any boolean columns in DataFrame `df` to floating point columns. Use the online documentation as a reference.

Note that there are a few different ways that you can accomplish this task. You can convert one boolean column at a time, or you can use the Pandas `select_dtypes()` method to find and return all boolean columns in DataFrame `df` and then convert the columns as a group. Use the online documentation as a reference.

[3]:
```
# YOUR CODE HERE

#SOLUTION:

column_names = df.select_dtypes(bool).columns
df[column_names] = df[column_names].astype(float)
```

Let's inspect the columns after the conversion.

[4]:
```
df.head()
```

[4]:
```
   host_is_superhost  host_has_profile_pic  host_identity_verified  \
0                0.0                   1.0                     1.0
1                0.0                   1.0                     1.0
2                0.0                   1.0                     1.0
3                0.0                   1.0                     0.0
4                0.0                   1.0                     1.0

   has_availability  instant_bookable  host_response_rate  \
```

```
0              1.0              0.0          -0.578829
1              1.0              0.0          -4.685756
2              1.0              0.0           0.578052
3              1.0              0.0           0.578052
4              1.0              0.0          -0.054002

   host_acceptance_rate  host_listings_count  host_total_listings_count  \
0             -2.845589            -0.054298                  -0.054298
1             -0.430024            -0.112284                  -0.112284
2             -2.473964            -0.112284                  -0.112284
3              1.010024            -0.112284                  -0.112284
4             -0.066308            -0.112284                  -0.112284

   accommodates  ...  n_host_verifications  \
0     -1.007673  ...              1.888373
1      0.067470  ...              0.409419
2      0.605041  ...             -1.069535
3     -0.470102  ...             -0.576550
4     -1.007673  ...              0.902404

   neighbourhood_group_cleansed_Bronx  neighbourhood_group_cleansed_Brooklyn  \
0                                 0.0                                    0.0
1                                 0.0                                    1.0
2                                 0.0                                    1.0
3                                 0.0                                    0.0
4                                 0.0                                    0.0

   neighbourhood_group_cleansed_Manhattan  \
0                                     1.0
1                                     0.0
2                                     0.0
3                                     1.0
4                                     1.0

   neighbourhood_group_cleansed_Queens  \
0                                  0.0
1                                  0.0
2                                  0.0
3                                  0.0
4                                  0.0

   neighbourhood_group_cleansed_Staten Island  room_type_Entire home/apt  \
0                                         0.0                        1.0
1                                         0.0                        1.0
2                                         0.0                        1.0
3                                         0.0                        0.0
4                                         0.0                        0.0
```

```
   room_type_Hotel room  room_type_Private room  room_type_Shared room
0                   0.0                     0.0                    0.0
1                   0.0                     0.0                    0.0
2                   0.0                     0.0                    0.0
3                   0.0                     1.0                    0.0
4                   0.0                     1.0                    0.0

[5 rows x 50 columns]
```

1.3 Part 3. Create Labeled Examples from the Data Set

Task: In the code cell below, create labeled examples from DataFrame `df`.

[5]:
```
# YOUR CODE HERE

#SOLUTION:

y = df['host_is_superhost']
X = df.drop(columns = 'host_is_superhost', axis=1)
```

1.4 Part 4. Create Training and Test Data Sets

Task: In the code cell below, create training and test sets out of the labeled examples. Create a test set that is 25 percent of the size of the data set. Save the results to variables `X_train, X_test, y_train, y_test`.

[6]:
```
# YOUR CODE HERE

# solution
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25,␣
 ↪random_state=1234)
```

[7]:
```
X_train.shape
```

[7]:
```
(21016, 49)
```

1.5 Part 5. Construct the Neural Network

1.5.1 Step 1. Define Model Structure

Next we will create our neural network structure. We will create an input layer, three hidden layers and an output layer:

- Input layer: The input layer will have the input shape corresponding to the number of features.
- Hidden layers: We will create three hidden layers of widths (number of nodes) 64, 32, and 16. They will utilize the ReLu activation function.

- Output layer: The output layer will have a width of 1. The output layer will utilize the sigmoid activation function. Since we are working with binary classification, we will be using the sigmoid activation function to map the output to a probability between 0.0 and 1.0. We can later set a threshold and assume that the prediction is class 1 if the probability is larger than or equal to our threshold, or class 0 if it is lower than our threshold.

To construct the neural network model using Keras, we will do the following: * We will use the Keras `Sequential` class to group a stack of layers. This will be our neural network model object. For more information, consult the Keras online documentation. * We will use the `InputLayer` class to create the input layer. For more information, consult the Keras online documentation. * We will use the `Dense` class to create each hidden layer and the output layer. For more information, consult the Keras online documentation. * We will add each layer to the neural network model object.

Task: Follow these steps to complete the code in the cell below:

1. Create the neural network model object.
    - Use `keras.Sequential()` to create a model object, and assign the result to the variable `nn_model`.
2. Create the input layer:
    - Call `keras.layers.InputLayer()` with the argument `input_shape` to specify the dimensions of the input. In this case, the dimensions will be the number of features (coumns) in `X_train`. Assign the number of features to the argument `input_shape`.
    - Assign the results to the variable `input_layer`.
    - Use `nn_model.add(input_layer)` to add the layer `input_layer` to the neural network model object.
3. Create the first hidden layer:
    - Call `keras.layers.Dense()` with the arguments `units=64` and `activation='relu'`.
    - Assign the results to the variable `hidden_layer_1`.
    - Use `nn_model.add(hidden_layer_1)` to add the layer `hidden_layer_1` to the neural network model object.
4. Create the second hidden layer using the same approach that you used to create the first hidden layer, specifying 32 units and the `relu` activation function.
    - Assign the results to the variable `hidden_layer_2`.
    - Add the layer to the neural network model object.
5. Create the third hidden layer using the same approach that you used to create the first two hidden layers, specifying 16 units and the `relu` activation function.
    - Assign the results to the variable `hidden_layer_3`.
    - Add the layer to the neural network model object.
6. Create the output layer using the same approach that you used to create the hidden layers, specifying 1 unit and the `sigmoid` activation function.

- Assign the results to the variable `output_layer`.
- Add the layer to the neural network model object.

[8]:
```
# 1. Create model object:

# nn_model = YOUR CODE HERE

# solution
nn_model = keras.Sequential()


# 2. Create the input layer and add it to the model object:
# Create input layer:
#input_layer = # YOUR CODE HERE
# SOLUTION:
input_layer = keras.layers.InputLayer(input_shape=(X_train.shape[1],))

# Add input_layer to the model object:
# YOUR CODE HERE
# SOLUTION:
nn_model.add(input_layer)


# 3. Create the first hidden layer and add it to the model object:
# Create hidden layer:
#hidden_layer_1 = # YOUR CODE HERE
# SOLUTION:
hidden_layer_1 = keras.layers.Dense(units=64, activation='relu')

# Add hidden_layer_1 to the model object:
# YOUR CODE HERE
# SOLUTION:
nn_model.add(hidden_layer_1)

# 4. Create the second hidden layer and add it to the model object:
# Create hidden layer:
#hidden_layer_2 = # YOUR CODE HERE
# SOLUTION:
hidden_layer_2 = keras.layers.Dense(units=32, activation='relu')

# Add hidden_layer_2 to the model object:
# YOUR CODE HERE
# SOLUTION:
nn_model.add(hidden_layer_2)


# 5. Create the third hidden layer and add it to the model object:
# Create hidden layer:
#hidden_layer_3 = # YOUR CODE HERE
# SOLUTION:
```

```
hidden_layer_3 = keras.layers.Dense(units=16, activation='relu')

# Add hidden_layer_3 to the model object:
# YOUR CODE HERE
# SOLUTION:
nn_model.add(hidden_layer_3)


# 6. Create the output layer and add it to the model object:
# Create output layer:
#output_layer = # YOUR CODE HERE
# SOLUTION
output_layer = keras.layers.Dense(units=1, activation='sigmoid')

# Add output_layer to the model object:
# YOUR CODE HERE

# SOLUTION
nn_model.add(output_layer)


# Print summary of neural network model structure
nn_model.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense (Dense)                (None, 64)                3200
_________________________________________________________________
dense_1 (Dense)              (None, 32)                2080
_________________________________________________________________
dense_2 (Dense)              (None, 16)                528
_________________________________________________________________
dense_3 (Dense)              (None, 1)                 17
=================================================================
Total params: 5,825
Trainable params: 5,825
Non-trainable params: 0
_________________________________________________________________
```

1.5.2 Step 2. Define the Optimization Function

Task: In the code cell below, create a stochastic gradient descent optimizer using `keras.optimizers.SGD()`. Specify a learning rate of 0.1 using the `learning_rate` parameter. Assign the result to the variables`gd_optimizer`.

[9]:
```
# sgd_optimizer =YOUR CODE HERE


# SOLUTION

sgd_optimizer = keras.optimizers.SGD(learning_rate=0.1)
```

1.5.3 Step 3. Define the Loss Function

Task: In the code cell below, create a binary cross entropy loss function using `keras.losses.BinaryCrossentropy()`. Use the parameter `from_logits=False`. Assign the result to the variable `loss_fn`.

[10]:
```
# loss_fn =YOUR CODE HERE


# SOLUTION

loss_fn = keras.losses.BinaryCrossentropy(from_logits=False)
```

1.5.4 Step 4. Compile the Model

Task: In the code cell below, package the network architecture with the optimizer and the loss function using the `compile()` method.

You will specify the optimizer, loss function and accuracy evaluation metric. Call the `nn_model.compile()` method with the following arguments: * Use the `optimizer` parameter and assign it your optimizer variable:`optimizer=sgd_optimizer` * Use the `loss` parameter and assign it your loss function variable: `loss=loss_fn` * Use the `metrics` parameter and assign it the accuracy evaluation metric: `metrics=['accuracy']`

[11]:
```
# YOUR CODE HERE


# SOLUTION
nn_model.compile(optimizer=sgd_optimizer, loss=loss_fn, metrics=['accuracy'])
```

1.6 Part 6. Fit the Model to the Training Data

We will define our own callback class to output information from our model while it is training. Make sure you execute the code cell below so that it can be used in subsequent cells.

[12]:
```
class ProgBarLoggerNEpochs(keras.callbacks.Callback):

    def __init__(self, num_epochs: int, every_n: int = 50):
        self.num_epochs = num_epochs
        self.every_n = every_n

    def on_epoch_end(self, epoch, logs=None):
        if (epoch + 1) % self.every_n == 0:
```

```
            s = 'Epoch [{}/ {}]'.format(epoch + 1, self.num_epochs)
            logs_s = ['{}: {:.4f}'.format(k.capitalize(), v)
                      for k, v in logs.items()]
            s_list = [s] + logs_s
            print(', '.join(s_list))
```

Task: In the code cell below, fit the neural network model to the training data.

1. Call `nn_model.fit()` with the training data `X_train` and `y_train` as arguments.
2. In addition, specify the following parameters:
    - Use the `epochs` parameter and assign it the variable to `epochs`: `epochs=num_epochs`
    - Use the `verbose` parameter and assign it the value of 0: `verbose=0`
    - Use the `callbacks` parameter and assign it a list containing our logger function: `callbacks=[ProgBarLoggerNEpochs(num_epochs_M, every_n=5)]`
    - We will use a portion of our training data to serve as validation data. Use the `validation_split` parameter and assign it the value `0.2`
3. Save the results to the variable `history`.

Note: This may take a while to run.

[13]:
```
t0 = time.time() # start time

num_epochs = 100 # epochs

# history = # YOUR CODE HERE


# SOLUTION
history = nn_model.fit(
    X_train,
    y_train,
    epochs=num_epochs,
    validation_split=0.2,
    verbose=0, # disable the default progress bar
    callbacks=[ProgBarLoggerNEpochs(num_epochs, every_n=5)],
)



t1 = time.time() # stop time

print('Elapsed time: %.2fs' % (t1-t0))
```

```
Epoch [5/ 100], Loss: 0.3575, Accuracy: 0.8388, Val_loss: 0.3945, Val_accuracy:
0.8185
```

```
Epoch [10/ 100], Loss: 0.3318, Accuracy: 0.8463, Val_loss: 0.3755, Val_accuracy:
0.8202
Epoch [15/ 100], Loss: 0.3163, Accuracy: 0.8573, Val_loss: 0.3669, Val_accuracy:
0.8342
Epoch [20/ 100], Loss: 0.3021, Accuracy: 0.8616, Val_loss: 0.3675, Val_accuracy:
0.8299
Epoch [25/ 100], Loss: 0.2879, Accuracy: 0.8680, Val_loss: 0.3783, Val_accuracy:
0.8368
Epoch [30/ 100], Loss: 0.2782, Accuracy: 0.8750, Val_loss: 0.3910, Val_accuracy:
0.8406
Epoch [35/ 100], Loss: 0.2683, Accuracy: 0.8816, Val_loss: 0.4037, Val_accuracy:
0.8121
Epoch [40/ 100], Loss: 0.2550, Accuracy: 0.8868, Val_loss: 0.4022, Val_accuracy:
0.8375
Epoch [45/ 100], Loss: 0.2487, Accuracy: 0.8885, Val_loss: 0.4137, Val_accuracy:
0.8359
Epoch [50/ 100], Loss: 0.2381, Accuracy: 0.8929, Val_loss: 0.4564, Val_accuracy:
0.8223
Epoch [55/ 100], Loss: 0.2311, Accuracy: 0.8971, Val_loss: 0.4731, Val_accuracy:
0.8292
Epoch [60/ 100], Loss: 0.2200, Accuracy: 0.9007, Val_loss: 0.4773, Val_accuracy:
0.8304
Epoch [65/ 100], Loss: 0.2160, Accuracy: 0.9065, Val_loss: 0.4958, Val_accuracy:
0.8225
Epoch [70/ 100], Loss: 0.2128, Accuracy: 0.9061, Val_loss: 0.5117, Val_accuracy:
0.8121
Epoch [75/ 100], Loss: 0.2001, Accuracy: 0.9110, Val_loss: 0.5534, Val_accuracy:
0.8223
Epoch [80/ 100], Loss: 0.1922, Accuracy: 0.9174, Val_loss: 0.5468, Val_accuracy:
0.8254
Epoch [85/ 100], Loss: 0.1838, Accuracy: 0.9215, Val_loss: 0.5932, Val_accuracy:
0.8192
Epoch [90/ 100], Loss: 0.1884, Accuracy: 0.9198, Val_loss: 0.5917, Val_accuracy:
0.8166
Epoch [95/ 100], Loss: 0.1708, Accuracy: 0.9273, Val_loss: 0.6272, Val_accuracy:
0.8187
Epoch [100/ 100], Loss: 0.1696, Accuracy: 0.9289, Val_loss: 0.6118,
Val_accuracy: 0.8142
Elapsed time: 81.42s
```

[14]:
```
history.history.keys()
```

[14]:
```
dict_keys(['loss', 'accuracy', 'val_loss', 'val_accuracy'])
```

1.6.1 Visualize the Model's Performance Over Time

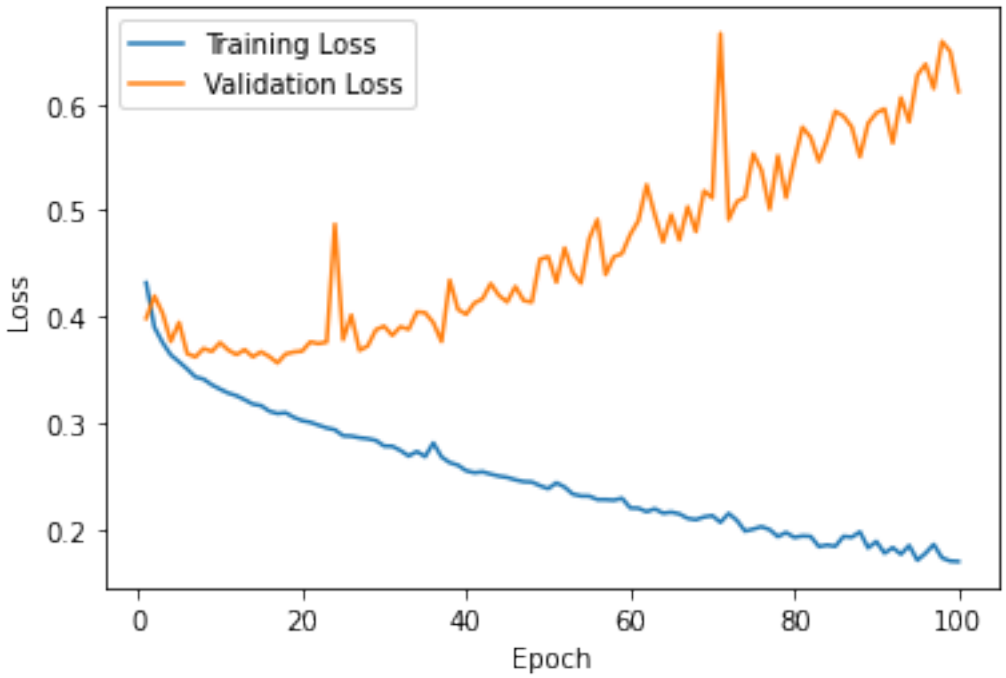
The code below outputs both the training loss and accuracy and the validation loss and accuracy. Let us visualize the model's performance over time:

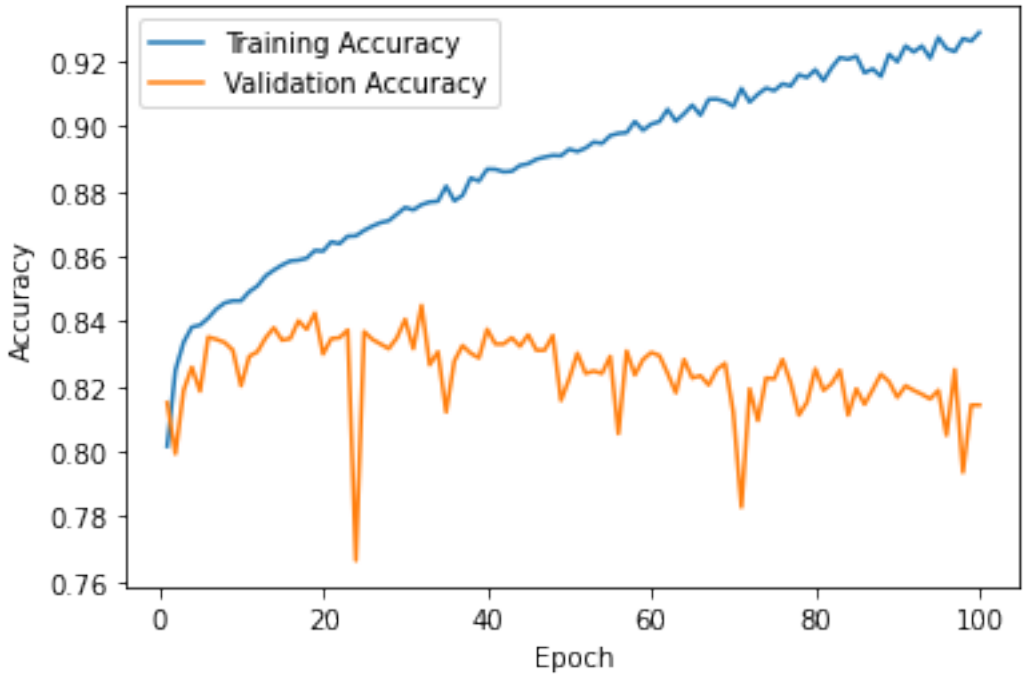
[15]:
```
# Plot training and validation loss
plt.plot(range(1, num_epochs + 1), history.history['loss'], label='Training␣
 ↪Loss')
plt.plot(range(1, num_epochs + 1), history.history['val_loss'],␣
 ↪label='Validation Loss')

plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()
plt.show()


# Plot training and validation accuracy
plt.plot(range(1, num_epochs + 1), history.history['accuracy'], label='Training␣
 ↪Accuracy')
plt.plot(range(1, num_epochs + 1), history.history['val_accuracy'],␣
 ↪label='Validation Accuracy')

plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()
plt.show()
```





1.7 Part 7. Evaluate the Model's Performance

We just evaluated our model's performance on the training and validation data. Let's now evaluate its performance on our test data and compare the results.

Keras makes the process of evaluating our model very easy. Recall that when we compiled the model, we specified the metric that we wanted to use to evaluate the model: accuracy. The Keras method `evaluate()` will return the loss and accuracy score of our model on our test data.

Task: In the code cell below, call `nn_model.evaluate()` with `X_test` and `y_test` as arguments.

Note: The `evaluate()` method returns a list containing two values. The first value is the loss and the second value is the accuracy score.

[16]:
```
#loss, accuracy = # YOUR CODE HERE
# SOLUTION
loss, accuracy = nn_model.evaluate(X_test, y_test)


print('Loss: {0} Accuracy: {1}'.format(loss, accuracy))
```

```
219/219 [==============================] - 0s 1ms/step - loss: 0.6409 -
accuracy: 0.8204
Loss: 0.6408803462982178 Accuracy: 0.8204396367073059
```

Next, for every example in the test set, we will make a prediction using the `predict()` method, receive a probability between 0.0 and 1.0, and then apply a threshold (we will use a threshold of 0.6) to obtain the predicted class. We will save the class label predictions to list `class_label_predictions`.

[17]:
```
# Make predictions on the test set
probability_predictions = nn_model.predict(X_test)
class_label_predictions=[]

for i in range(0,len(y_test)):
    if probability_predictions[i] >= 0.6:
        class_label_predictions.append(1)
    else:
        class_label_predictions.append(0)
```

Task: In the code cell below, create a confusion matrix out of `y_test` and the list `class_label_predictions`.

[18]:
```
# YOUR CODE HERE
# Solution

c_m = confusion_matrix(y_test, class_label_predictions, labels=[True, False])

pd.DataFrame(
c_m,
columns=['Predicted: Terrible Host', 'Predicted: Great Host'],
index=['Actual: Terrible Host', 'Actual: Great Host']
)
```

[18]:

	Predicted: Terrible Host	Predicted: Great Host
Actual: Terrible Host	854	881
Actual: Great Host	370	4901

1.8 Part 8. Analysis

Experiment with the neural network implementation above and compare your results every time you train the network. Pay attention to the time it takes to train the network, and the resulting loss and accuracy on both the training and test data.

Below are some ideas for things you can try:

- Adjust the learning rate.
- Change the number of epochs by experimenting with different values for the variable `num_epochs`.
- Add more hidden layers and/or experiment with different values for the `unit` parameter in the hidden layers to change the number of nodes in the hidden layers.

Record your findings in the cell below.