Part 3 Report Scott Peters 10102681

I decided to split my data into an 80/20 split, where 80% of my data was training and 20% was the test data. The code I used to do this was:

```
# Taken from https://stackoverflow.com/questions/43697240/how-can-i-split-a-dataset-from-a-csv-file-for-training-and-testing
train = data.sample(frac=0.8, random_state = np.random.RandomState())
test = data.loc[~data.index.isin(train.index)]

train.to_csv('heart_train.csv', index=False)
test.to_csv('heart_test.csv', index=False)

print("--Get data--")
y_test = test.chd
x_test = test.drop('chd', axis=1)

y_train = train.chd
x_train = train.drop('chd', axis=1)

print(x_train.shape, x_test.shape, y_train.shape, y_test.shape)
```

This split allowed me to have a high number of datapoints (370) to train my model, while still having a somewhat high number of datapoints to test on (92).

The way I decided to handle the non-numerical input column was to simply map the data to 0/1; more precisely, map Present to 1, and map Absent to 0.

```
data['famhist'] = data['famhist'].map({'Present': '1', 'Absent': '0'})
```

I then had to adjust the dtype of the columns, since all other columns had a dtype of float32, but the famhist column had a dtype of string.

```
# famhist still has dtype string need to convert to float

data['famhist'] = data['famhist'].astype(float)

inputs = {}

for name, column in data.items():
  dtype = column.dtype
  if dtype == object:
    dtype = tf.string
  else:
    dtype = tf.float32

  inputs[name] = tf.keras.Input(shape=(1,), name=name, dtype=dtype)

inputs
```

This allowed me to use a similar method of getting train/test data as before, and inputting that to my model. I am not sure if this is the best or ideal method to get this data to run, but it seemed to work fine for me.

To deal with overfitting/underfitting I first tried to implement dropout, using a dropout rate of 0.2 as well as 0.5. Both of these values seemed to make my predictions A LOT worse, for example an accuracy of 60% compared to 70% without. Although this did make my training/testing accuracy closer to each other, I felt like this was not the direction to go in since it was hurting my predictions so much. I then played with my layers neurons, trying everything from 8 to 128 while also adding in l2 regularizers. I settled on using 2 hidden layers with relu activation, each with 16 neurons, as well as using l2 regularizers.

```
print("--Make model--")
model = tf.keras.models.Sequential([
  tf.keras.layers.Dense(16, input_shape=(9,), activation="relu", kernel_regularizer=tf.keras.regularizers.l2(0.0001)),
  tf.keras.layers.Dense(16, activation="relu", kernel_regularizer=tf.keras.regularizers.l2(0.0001)),
  #tf.keras.layers.Dense(32, activation="relu", kernel_regularizer=tf.keras.regularizers.l2(0.001)),
  tf.keras.layers.Dense(1, activation = "sigmoid")
])
```

As you can see I played around with a third hidden layer, however this mostly just hampered my results so I gave up on it. These things combined gave me a respectable 76% train accuracy and a 72% test accuracy. These values are much closer together than they were before the changes I put in place, which makes me believe I have (somewhat) fixed the overfitting/underfitting.

I used a similar model and hyperparameters as my other parts:

```
initial_learning_rate = 1e-2
lr_schedule = tf.keras.optimizers.schedules.ExponentialDecay(
  initial_learning_rate, decay_steps=100000, decay_rate=0.9, staircase=True)

optimizer = tf.keras.optimizers.Adamax(learning_rate=lr_schedule)

model.compile(optimizer=optimizer, loss='binary_crossentropy', metrics=['accuracy'])

print("--Fit model--")
model.fit(x_train, y_train, epochs=75, verbose=2, batch_size = 5)

print("--Evaluate model--")
model_loss1, model_acc1 = model.evaluate(x_train,  y_train, verbose=2)
model_loss2, model_acc2 = model.evaluate(x_test,  y_test, verbose=2)
print(f"Train / Test Accuracy: {model_acc1*100:.1f}% / {model_acc2*100:.1f}%")
```

However, in this case since our output is binary, I used sigmoid activation on my last layer, as well as binary cross entropy for loss. These are the standard values to use when our output is binary, however I also added a learning rate schedule using the Adamax optimizer to help aid my accuracies. My final results were a 76% train accuracy and a 72% test accuracy.