CPSC 501- Assignment 2 Part 3: Build a logistic regression model to predict coronary heart disease

For part 3 of assignment, I've mainly used the tutorial on loading panda dataframe
https://www.tensorflow.org/tutorials/load_data/pandas_dataframe
And on overfitting and underfitting data
https://www.tensorflow.org/tutorials/keras/overfit_and_underfit

Data-splitting decisions:

```
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split

# Read the csv file
data = pd.read_csv("heart.csv")

# Split data into a training set and testing set
train, test = train_test_split(data, test_size=0.15, random_state=

# Save the training and testing data to a new csv file
train.to_csv("heart_train.csv", index=False)
test.to_csv("heart_test.csv", index=False)
```

Initially, I split the provided heart.csv file into two files, heart_train.csv and heart_test.csv. Heart_train.csv contains 85% of the initial data and heart_test.csv contains the 15% of the initial data. Since provided data is quite small, I thought I need at least 85% of data to train the model.

Non-numerical input data:

Since categorical inputs are unordered and they are inappropriate to feed directly to the model; the model would interpret them as being ordered. To use these inputs I needed to encode them, either as one-hot vectors or embedding vectors. For binary features on the other hand do not generally need to be encoded or normalized.

For categorical features, I first needed to encode them into either binary vectors or embeddings. Since all these features only contain a small number of categories, I convert the inputs directly to one-hot vectors using the output_mode='one_hot' option. To determine the vocabulary for each input, I create a layer to convert that vocabulary to a one-hot vector

```
preprocessed = []

for name in binary_feature_names:
  vocab = sorted(set(raw_train_dataset[name]))
  print(f'name: {name}')
  print(f'vocab: {vocab}\n')

  if type(vocab[0]) is str:
    lookup = tf.keras.layers.StringLookup(vocabulary=vocab, output_mode='one_hot')

  x = inputs[name][:, tf.newaxis]
  x = lookup(x)
  preprocessed.append(x)
```

Overfitting/underfitting:

```
print("--Make model--")
# Build the Keras model
# Model based off TensorFlow 2 tutorial on overfitting and underfitting
# https://www.tensorflow.org/tutorials/keras/overfit_and_underfit
body = tf.keras.Sequential([
  tf.keras.layers.Flatten(),
  tf.keras.layers.Dense(512, kernel_regularizer=tf.keras.regularizers.l2(0.01), activation='relu'),
  tf.keras.layers.Dropout(rate=0.2),
  tf.keras.layers.Dense(128, kernel_regularizer=tf.keras.regularizers.l2(0.01), activation='relu'),
  tf.keras.layers.Dense(1, activation = 'sigmoid')
])
```

A common way to mitigate overfitting is to put constraints on the complexity of a network by forcing its weights only to take small values, which makes the distribution of weight values more “regular”. This is called “weight regularization”, and it is done by adding to the loss function of the network a cost associated with having large weights. I added weight regularization by passing weight regularizer instances to layers as keyword arguments to mitigate overfitting/underfitting.

Also, Dropout is one of the most effective and most commonly used regularization techniques for neural networks. Dropout, applied to a layer, consist of randomly “dropping out” a number of output features of the layer during training. I added dropout layer to my network to prevent overfitting/underfitting.

Hyper-parameters/ Results:

```
--Evaluate model--
13/13 - 0s - loss: 0.9932 - accuracy: 0.7423 - 243ms/epoch - 19ms/step
3/3 - 0s - loss: 1.0602 - accuracy: 0.7571 - 29ms/epoch - 10ms/step
Train / Test Accuracy: 74.2% / 75.7%
```

To compile the model, I used model.complie(optimizer=‘Adam’, loss=‘binary_crossentropy’, metrics=[‘accuracy’]). Finally, to fit the model, I used 15 epochs, and 128 batch size. Results came out to be 74.2% for train data and 75.5% for test data.