CSE 572: Lab 6

In this lab, you will practice implementing feed-forward network and parameter tuning with k-fold cross valivation.

To execute and make changes to this notebook, click File > Save a copy to save your own version in your Google Drive or Github. Read the step-by-step instructions below carefully. To execute the code, click on each cell below and press the SHIFT-ENTER keys simultaneously or by clicking the Play button.

When you finish executing all code/exercises, save your notebook then download a copy (.ipynb file). Submit the following **three** things:

1. a link to your Colab notebook,
2. the .ipynb file, and
3. a pdf of the executed notebook on Canvas.

To generate a pdf of the notebook, click File > Print > Save as PDF.

## PUT YOUR GROUP INFO HERE

| Group number | August Group XXX | |
|---|---|---|
| Member 1 | NAME | ASURITE ID |
| Member 2 | | |
| Member 3 | | |
| Member 4 | | |

```
# Import libraries
import tensorflow as tf
import numpy as np
import pandas as pd
from tensorflow import keras
from tensorflow.keras import layers

# Set the random seed for reproducibility
seed = 0
tf.random.set_seed(seed)
```

## Feed-forward neural networks

### Load the dataset

For this example, we will use the Cleveland Heart Disease dataset. Review the dataset documentation to learn more about the attributes and other aspects of the dataset. The dataset consists of a CSV file with 303 rows. Each row contains information about a patient. There are 14 attribute columns and one binary class column ( `target` ) that reports whether or not a patient had a heart disease. We will train a feed-forward neural network model to predict whether or not a given patient has a heart disease based on the attribute values.

```
# Load the dataset
data = pd.read_csv("http://storage.googleapis.com/download.tensorflow.org/data/heart.csv")

# Print sample rows
data.head()
```

| | age | sex | cp | trestbps | chol | fbs | restecg | thalach | exang | oldpeak | slope | ca | thal | target |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 63 | 1 | 1 | 145 | 233 | 1 | 2 | 150 | 0 | 2.3 | 3 | 0 | fixed | 0 |
| 1 | 67 | 1 | 4 | 160 | 286 | 0 | 2 | 108 | 1 | 1.5 | 2 | 3 | normal | 1 |
| 2 | 67 | 1 | 4 | 120 | 229 | 0 | 2 | 129 | 1 | 2.6 | 2 | 2 | reversible | 0 |
| 3 | 37 | 1 | 3 | 130 | 250 | 0 | 0 | 187 | 0 | 3.5 | 3 | 0 | normal | 0 |
| 4 | 41 | 0 | 2 | 130 | 204 | 0 | 2 | 172 | 0 | 1.4 | 1 | 0 | normal | 0 |

```
# Print the number of rows and columns
data.shape
```

```
(303, 14)
```

Split the dataset into three randomly-sampled subsets: training (60%), validation (20%), and test (20%). Use the `seed` variable for the `random_state` .

```
train, val, test = np.split(data.sample(frac=1, random_state=seed), [int(.6*len(data)), int(.8*len(data))])
```

Print the number of samples in each of the three subsets and the number of instances from each class. For example, for the training set you might print "The training set has ___ instances (___ heart disease, ___ no disease)".

```
# YOUR CODE HERE
print(f"The training set has, {len(train)}, instances")
print(f"- Heart disease:", train['target'].value_counts()[1])
print(f"- No heart disease:", train['target'].value_counts()[0])

print(f"The validation set has", len(val), "instances")
print(f"- Heart disease:, {val['target'].value_counts()[1]}")
print(f"- No heart disease:, {val['target'].value_counts()[0]}")

print(f"The test set has, {len(test)}, instances")
print(f"Heart disease:, {test['target'].value_counts()[1]}")
print(f"No heart disease:, {test['target'].value_counts()[0]}")
```

```
The training set has, 181, instances
- Heart disease: 48
- No heart disease: 133
The validation set has 61 instances
- Heart disease:, 18
- No heart disease:, 43
The test set has, 61, instances
Heart disease:, 17
No heart disease:, 44
```

### Prepare the dataset

Before we can feed this dataset to our model for training and evaluation, we need to perform a few steps to get it ready:

1. Convert the dataframes to Dataset objects
2. Normalize the numerical feature values
3. Binarize the Categorical features by converting to one-hot encodings

```
# Convert dataframes to Dataset objects
def dataframe_to_dataset(df, shuffle=True):
    df = df.copy()
    # Remove the target column and store in a separate array
    labels = df.pop('target')
    ds = tf.data.Dataset.from_tensor_slices((dict(df), labels))
    if shuffle:
      ds = ds.shuffle(buffer_size=len(df), reshuffle_each_iteration=False)

    return ds


train_ds_original = dataframe_to_dataset(train)
val_ds_original = dataframe_to_dataset(val)
test_ds_original = dataframe_to_dataset(test)
```

The Dataset object yields a tuple containing the input feature vector and target (class value): `(input, target)` . `input` is a dictionary of features and `target` is the value 0 or 1. The code below prints an example instance drawn from the training Dataset object.

```
for x, y in train_ds_original.take(1):
    print("Input:", x)
    print("\nTarget:", y)
```

```
Input: {'age': <tf.Tensor: shape=(), dtype=int64, numpy=63>, 'sex': <tf.Tensor: shape=(), dtype=int64, numpy=0>, 'cp': <tf.Tensor: shape=(), dtype=int64, numpy=3>, 'trestbps': <tf.Tensor: shape=(), dtype=int64, numpy=135>, 'chol': <tf.Tensor: shape=(), dtype=int64, numpy=252>, 'fbs': <tf.Tensor: shape=(), dtype=int64, numpy=0>, 'restec

Target: tf.Tensor(0, shape=(), dtype=int64)
```

We can use the batch() function in keras to create batches from the full dataset for passing to the model. For the training dataset, we'll define a hyperparameter `batch_size` that we will set. For the validation and test sets, we will make the batch size equivalent to the size of the subset so all samples in that subset are evaluated each time the dataset is evaluated by the model.

```
batch_size = 32
train_ds = train_ds_original.batch(batch_size)
val_ds = val_ds_original.batch(val.shape[0])
test_ds = test_ds_original.batch(test.shape[0])
```

There are seven categorical features in the dataset: `sex`, `cp`, `fbs`, `restecg`, `exang`, `ca`, and `thal`. You can read more about what these features mean in the dataset documentation. All of them except `thal` have integer data type while `thal` has String data type. Below we define a function to encode these feature values as one-hot encodings using the IntegerLookup() and StringLookup() layers. These layers create look-up tables for mapping a set of arbitrary integers or strings to a one-hot encoding. We use an `is_string` argument to indicate whether we should use the `StringLookup()` for `thal` or the `IntegerLookup()` for the remaining features.

```
def encode_categorical_feature(feature, name, dataset, is_string):
    from tensorflow.keras.layers import IntegerLookup
    from tensorflow.keras.layers import StringLookup

    # Create lookup layer to turn categorical features into 1-hot integer encodings
    if is_string:
        lookup = StringLookup(output_mode="binary")
    else:
        lookup = IntegerLookup(output_mode="binary")

    # Prepare a Dataset that only yields the feature of interest
    feature_ds = dataset.map(lambda x, y: x[name])
    feature_ds = feature_ds.map(lambda x: tf.expand_dims(x, -1))

    # Find the set of possible values and assign them a fixed integer index
    lookup.adapt(feature_ds)

    # Turn the input into integer indices
    encoded_feature = lookup(feature)
    return encoded_feature
```

The remaining features in the dataset ( `age`, `trestbps`, `chol`, `thalach`, `oldpeak`, and `slope` ) are all numerical measurements. You can read more about what these features mean in the dataset documentation. We don't need to encode the numerical features, but we do want to scale

them to the same range of values (e.g., using standardization or normalization). Below we define a function that uses the Normalization() layer to standardize the data (subtract the mean and divide by the standard deviation for each feature).

```
def normalize(feature, name, dataset):
    from tensorflow.keras.layers import Normalization

    # Create a Normalization layer for our feature
    normalizer = Normalization()

    # Prepare a Dataset that only yields the feature of interest
    feature_ds = dataset.map(lambda x, y: x[name])
    feature_ds = feature_ds.map(lambda x: tf.expand_dims(x, -1))

    # Learn the statistics of the data
    normalizer.adapt(feature_ds)

    # Normalize the input feature
    norm_feature = normalizer(feature)
    return norm_feature
```

Now we can apply these functions to each of our features and return an encoded/preprocessed input layer. We first create tensor variables for each of the inputs, then apply the appropriate function, then concatenate all of these input layers for each feature together to form a single input feature layer.

```
all_inputs = [
    keras.Input(shape=(1,), name="sex", dtype="int64"),
    keras.Input(shape=(1,), name="cp", dtype="int64"),
    keras.Input(shape=(1,), name="fbs", dtype="int64"),
    keras.Input(shape=(1,), name="restecg", dtype="int64"),
    keras.Input(shape=(1,), name="exang", dtype="int64"),
    keras.Input(shape=(1,), name="ca", dtype="int64"),
    keras.Input(shape=(1,), name="thal", dtype="string"),
    keras.Input(shape=(1,), name="age"),
    keras.Input(shape=(1,), name="trestbps"),
    keras.Input(shape=(1,), name="chol"),
    keras.Input(shape=(1,), name="thalach"),
    keras.Input(shape=(1,), name="oldpeak"),
    keras.Input(shape=(1,), name="slope"),
]

feature_layer = layers.concatenate(
    [
        encode_categorical_feature(all_inputs[0], "sex", train_ds, False),
        encode_categorical_feature(all_inputs[1], "cp", train_ds, False),
        encode_categorical_feature(all_inputs[2], "fbs", train_ds, False),
        encode_categorical_feature(all_inputs[3], "restecg", train_ds, False),
        encode_categorical_feature(all_inputs[4], "exang", train_ds, False),
        encode_categorical_feature(all_inputs[5], "ca", train_ds, False),
        encode_categorical_feature(all_inputs[6], "thal", train_ds, True),
        normalize(all_inputs[7], "age", train_ds),
        normalize(all_inputs[8], "trestbps", train_ds),
        normalize(all_inputs[9], "chol", train_ds),
        normalize(all_inputs[10], "thalach", train_ds),
        normalize(all_inputs[11], "oldpeak", train_ds),
        normalize(all_inputs[12], "slope", train_ds)
    ]
)
```

## Build the model

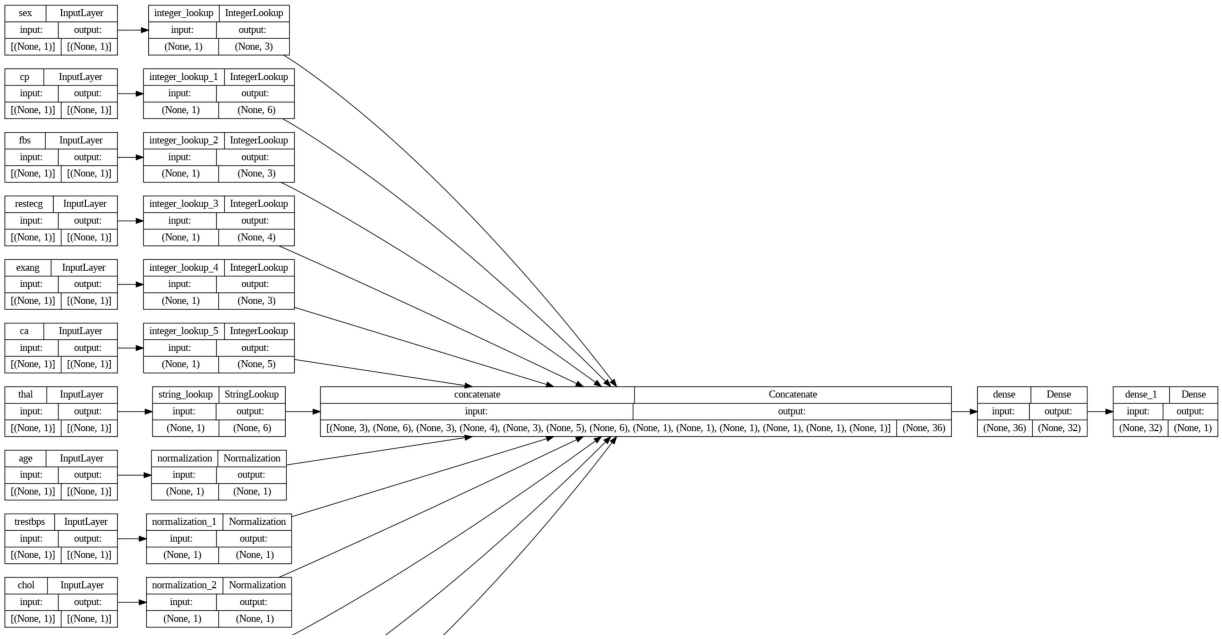
Now that we've prepared our dataset, we can construct our neural network model. We construct the model by composing Layer objects starting with the input layer (which we've already defined as `feature_layer`) and ending with the output layer (which will be the final output of the model). In this example, we will only use Dense() layers which are simple fully-connected feed-forward layers (i.e., each output from layer `i-1` is connected by a weight variable to every neuron in layer `i`). We will create a network with only one hidden layer (between the input and output layers).

The Dense() layer object allows us to specify the activation function to use using the `activation` argument. In class, we talked about several activation functions including sigmoid, sign, and tanh. Another commonly used activation function is the rectified linear unit, or "ReLU" function. Another commonly used activation function is the rectified linear unit, or "ReLU" function, which has the equation $a(z) = max(0, z)$. We will use `relu` as our activation function in this example for all layers except the final layer, which will use a `sigmoid` activation.

```
# Create a variable for the number of units/neurons in the layer
h1_units = 32
# Create a Dense layer and append it to the input layer
h1_layer = layers.Dense(h1_units, activation='relu')(feature_layer)

# Create an output layer with one output representing the likelihood of
# heart disease and append it to the hidden layer
output_layer = layers.Dense(1, activation="sigmoid")(h1_layer)

# Build the model specifying the input and output layer
model = keras.Model(inputs=all_inputs, outputs=output_layer)
```

We can plot our completed model to visualize the input, hidden, and output layers.

```
# `rankdir='LR'` is to make the graph horizontal
keras.utils.plot_model(model, show_shapes=True, rankdir="LR")
```




Next we compile the model by specifying the optimization technique and loss function to be used in model training. We can also specify the metric(s) that will be logged during training. We will use stochastic gradient descent (`sgd`) for the optimizer and binary cross entropy (log loss) as the loss function. We will log the accuracy metric during training. We can also specify the learning rate hyperparameter here.

```
learning_rate = 0.01
model.compile(keras.optimizers.SGD(learning_rate=learning_rate), "binary_crossentropy", metrics=["accuracy"])
```

## Train the model

Now that we've constructed and compiled our model, we can train the model using our training dataset. This is done in keras using the `fit()` function, which also gives us an option to provide the validation dataset which will be used to evaluate validation accuracy after every epoch.

```
model_result = model.fit(train_ds, epochs=100, validation_data=val_ds)
```

```
Epoch 1/100
6/6 [==============================] - 1s 71ms/step - loss: 0.5801 - accuracy: 0.7238 - val_loss: 0.6214 - val_accuracy: 0.6066
Epoch 2/100
6/6 [==============================] - 0s 5ms/step - loss: 0.5635 - accuracy: 0.7293 - val_loss: 0.6068 - val_accuracy: 0.6393
Epoch 3/100
6/6 [==============================] - 0s 5ms/step - loss: 0.5491 - accuracy: 0.7403 - val_loss: 0.5941 - val_accuracy: 0.6721
Epoch 4/100
6/6 [==============================] - 0s 5ms/step - loss: 0.5365 - accuracy: 0.7348 - val_loss: 0.5827 - val_accuracy: 0.6721
Epoch 5/100
6/6 [==============================] - 0s 5ms/step - loss: 0.5253 - accuracy: 0.7514 - val_loss: 0.5727 - val_accuracy: 0.6885
Epoch 6/100
6/6 [==============================] - 0s 5ms/step - loss: 0.5152 - accuracy: 0.7569 - val_loss: 0.5635 - val_accuracy: 0.6885
Epoch 7/100
6/6 [==============================] - 0s 5ms/step - loss: 0.5062 - accuracy: 0.7569 - val_loss: 0.5553 - val_accuracy: 0.6885
Epoch 8/100
6/6 [==============================] - 0s 5ms/step - loss: 0.4979 - accuracy: 0.7569 - val_loss: 0.5476 - val_accuracy: 0.6885
Epoch 9/100
6/6 [==============================] - 0s 5ms/step - loss: 0.4904 - accuracy: 0.7624 - val_loss: 0.5406 - val_accuracy: 0.7049
Epoch 10/100
6/6 [==============================] - 0s 5ms/step - loss: 0.4833 - accuracy: 0.7624 - val_loss: 0.5340 - val_accuracy: 0.7049
Epoch 11/100
6/6 [==============================] - 0s 5ms/step - loss: 0.4768 - accuracy: 0.7624 - val_loss: 0.5280 - val_accuracy: 0.7049
Epoch 12/100
6/6 [==============================] - 0s 5ms/step - loss: 0.4708 - accuracy: 0.7624 - val_loss: 0.5223 - val_accuracy: 0.7049
Epoch 13/100
6/6 [==============================] - 0s 5ms/step - loss: 0.4651 - accuracy: 0.7624 - val_loss: 0.5169 - val_accuracy: 0.7049
Epoch 14/100
```

```
6/6 [==============================] - 0s 5ms/step - loss: 0.4598 - accuracy: 0.7624 - val_loss: 0.5120 - val_accuracy: 0.7049
Epoch 15/100
6/6 [==============================] - 0s 5ms/step - loss: 0.4548 - accuracy: 0.7624 - val_loss: 0.5072 - val_accuracy: 0.7049
Epoch 16/100
6/6 [==============================] - 0s 5ms/step - loss: 0.4501 - accuracy: 0.7680 - val_loss: 0.5028 - val_accuracy: 0.7049
Epoch 17/100
6/6 [==============================] - 0s 5ms/step - loss: 0.4456 - accuracy: 0.7790 - val_loss: 0.4985 - val_accuracy: 0.6885
Epoch 18/100
6/6 [==============================] - 0s 5ms/step - loss: 0.4413 - accuracy: 0.7790 - val_loss: 0.4945 - val_accuracy: 0.7213
Epoch 19/100
6/6 [==============================] - 0s 6ms/step - loss: 0.4372 - accuracy: 0.7790 - val_loss: 0.4906 - val_accuracy: 0.7213
Epoch 20/100
6/6 [==============================] - 0s 5ms/step - loss: 0.4333 - accuracy: 0.7735 - val_loss: 0.4869 - val_accuracy: 0.7213
Epoch 21/100
6/6 [==============================] - 0s 5ms/step - loss: 0.4295 - accuracy: 0.7790 - val_loss: 0.4833 - val_accuracy: 0.7213
Epoch 22/100
6/6 [==============================] - 0s 5ms/step - loss: 0.4258 - accuracy: 0.7790 - val_loss: 0.4798 - val_accuracy: 0.7213
Epoch 23/100
6/6 [==============================] - 0s 5ms/step - loss: 0.4223 - accuracy: 0.7790 - val_loss: 0.4764 - val_accuracy: 0.7213
Epoch 24/100
6/6 [==============================] - 0s 5ms/step - loss: 0.4189 - accuracy: 0.7845 - val_loss: 0.4732 - val_accuracy: 0.7213
Epoch 25/100
6/6 [==============================] - 0s 8ms/step - loss: 0.4156 - accuracy: 0.7901 - val_loss: 0.4700 - val_accuracy: 0.7541
Epoch 26/100
6/6 [==============================] - 0s 8ms/step - loss: 0.4124 - accuracy: 0.7901 - val_loss: 0.4669 - val_accuracy: 0.7541
Epoch 27/100
6/6 [==============================] - 0s 5ms/step - loss: 0.4094 - accuracy: 0.7956 - val_loss: 0.4639 - val_accuracy: 0.7541
Epoch 28/100
6/6 [==============================] - 0s 5ms/step - loss: 0.4064 - accuracy: 0.7956 - val_loss: 0.4611 - val_accuracy: 0.7541
Epoch 29/100
6/6 [==============================] - 0s 5ms/step - loss: 0.4036 - accuracy: 0.7901 - val_loss: 0.4583 - val_accuracy: 0.7541
```

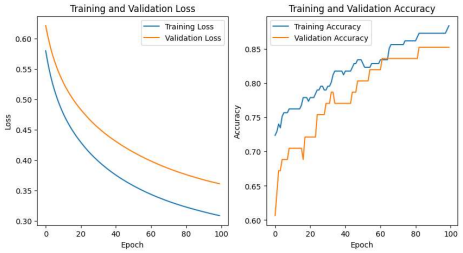
The `fit()` function returns a history attribute that gives the metrics recorded during training as a dictionary. We can print the dictionary keys to see which metrics were stored:

```
model_result.history.keys()
```

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

Create a figure with two subplots. The first subplot should plot the training and validation loss (`loss` and `val_loss`) and the second subplot should plot the training and validation accuracy (`accuracy` and `val_accuracy`). Make sure you include the axis labels and a legend in each plot.

```
import matplotlib.pyplot as plt



# Create a figure with two subplots
fig, (ax1, ax2) = plt.subplots(ncols=2, figsize=(10,5))


ax1.plot(model_result.history['loss'], label='Training Loss')
ax1.plot(model_result.history['val_loss'], label='Validation Loss')
ax1.set_xlabel('Epoch')
ax1.set_ylabel('Loss')
ax1.set_title('Training and Validation Loss')
ax1.legend()


ax2.plot(model_result.history['accuracy'], label='Training Accuracy')
ax2.plot(model_result.history['val_accuracy'], label='Validation Accuracy')
ax2.set_xlabel('Epoch')
ax2.set_ylabel('Accuracy')
ax2.set_title('Training and Validation Accuracy')
ax2.legend()

plt.show()
```




## Test the model

Finally, we evaluate our trained model on the held-out test set. First we predict the outputs for the test data.

```
preds = model.predict(test_ds)
```

```
1/1 [==============================] - 0s 254ms/step
```

The model output from the final sigmoid layer is a value between 0 and 1 representing the likelihood that a given sample patient has heart disease. To get the predicted classes, we predict 1 if the output was >= 0.5 and 0 otherwise.

```
pred_classes = [1 if p >= 0.5 else 0 for p in preds]
```

Compute and print the test accuracy.

```
# Compute and print the test accuracy
from sklearn.metrics import accuracy_score
for x,y in test_ds.take(1):
    y_acc = y

test_acc = accuracy_score(y_acc,pred_classes)

# Extract the test accuracy from the results
test_res = model.evaluate(test_ds)
train_res = model.evaluate(train_ds)
val_res = model.evaluate(val_ds)

## Extract the training and validation accuracies from the results
train_acc = train_res[1]
val_acc = val_res[1]
test_acc = test_res[1]

print("Test Accuracy: {:.2f}".format(test_acc))
print("Train Accuracy: {:.2f}".format(train_acc))
print("Validation Accuracy: {:.2f}".format(val_acc))
```

```
1/1 [==============================] - 0s 31ms/step - loss: 0.4122 - accuracy: 0.7705
6/6 [==============================] - 0s 6ms/step - loss: 0.3879 - accuracy: 0.8895
1/1 [==============================] - 0s 25ms/step - loss: 0.3612 - accuracy: 0.8525
Test Accuracy: 0.77
Train Accuracy: 0.89
Validation Accuracy: 0.85
```

**Question 1: There is a large difference between the training and validation accuracies and the test accuracy. What do you think could explain this difference?**

**Answer:**

YOUR ANSWER HERE

- Small validation/test set size

## Hyperparameter tuning with K-fold Cross Validation

### k-fold Cross validation

We will use 5-fold cross validation to train and evaluate our classifier. We will not do any model selection/hyperparameter tuning in this step, so we need to split our data into a training and test set.

To split the data into 3 folds we will shuffle the rows and then split them into $k$ equal groups.

```
k = 3

# Note: np.split raises error if indices_or_sections is
# an integer and doesn't result in equal size splits
folds = np.split(data.sample(frac=1, random_state=seed), indices_or_sections=k)
```

Use a for loop to print the number of samples and number of samples from each class in each fold.

```
folds[0][folds[0]['target']==0]
```

| | age | sex | cp | trestbps | chol | fbs | restecg | thalach | exang | oldpeak | slope | ca | thal | target |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **225** | 58 | 0 | 4 | 130 | 197 | 0 | 0 | 131 | 0 | 0.6 | 2 | 0 | normal | 0 |
| **201** | 43 | 1 | 4 | 115 | 303 | 0 | 0 | 181 | 0 | 1.2 | 2 | 0 | normal | 0 |
| **52** | 53 | 0 | 4 | 130 | 264 | 0 | 2 | 143 | 0 | 0.4 | 2 | 0 | normal | 0 |
| **168** | 54 | 0 | 3 | 108 | 267 | 0 | 2 | 167 | 0 | 0.0 | 1 | 0 | normal | 0 |
| **223** | 39 | 0 | 3 | 138 | 220 | 0 | 0 | 152 | 0 | 0.0 | 2 | 0 | normal | 0 |
| **...** | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... |
| **247** | 59 | 1 | 0 | 164 | 176 | 1 | 0 | 90 | 0 | 1.0 | 1 | 2 | 1 | 0 |
| **288** | 57 | 0 | 4 | 128 | 303 | 0 | 2 | 159 | 0 | 0.0 | 1 | 1 | normal | 0 |
| **156** | 62 | 0 | 4 | 150 | 244 | 0 | 0 | 154 | 1 | 1.4 | 2 | 0 | normal | 0 |
| **297** | 56 | 1 | 4 | 125 | 249 | 1 | 2 | 144 | 1 | 1.2 | 2 | 1 | normal | 0 |
| **252** | 57 | 0 | 1 | 130 | 236 | 0 | 0 | 174 | 0 | 0.0 | 1 | 1 | 2 | 0 |

75 rows × 14 columns

```
# YOUR CODE HERE

for i, fold in enumerate(folds):
    has_disease = fold[fold['target'] == 1].shape[0]
    no_disease = fold[fold['target'] == 0].shape[0]
    print(f"Fold, {i+1}, has, {len(fold)}, instances (, {has_disease}, heart disease, {no_disease} no disease)")
```

```
Fold. 1. has. 101. instances (. 26. heart disease. 75 no disease)
```

## Train a Neural Network classifier

We will use the Neural Network (NN) implemented in previous section for our classification model. Use cross validation to train and evaluate the model. Set hyperparameters to `h1_units=32` and `h1_activation='relu'`, and tune the hypaerparameter `learning_rate` with candidate values `[0.01, 0.03]`.

Implement a for loop to iterate hyperparameters, and then implement another for loop to iterate through each fold, training a new NN model each iteration with one fold assigned to validation and the remaining folds assigned to training. Compute the validation accuracy for each iteration and append it to the `accuracies` list.

```
from sklearn.metrics import accuracy_score

def create_model(h1_units, learning_rate, h1_activation):
  all_inputs = [
    keras.Input(shape=(1,), name="sex", dtype="int64"),
    keras.Input(shape=(1,), name="cp", dtype="int64"),
    keras.Input(shape=(1,), name="fbs", dtype="int64"),
    keras.Input(shape=(1,), name="restecg", dtype="int64"),
    keras.Input(shape=(1,), name="exang", dtype="int64"),
    keras.Input(shape=(1,), name="ca", dtype="int64"),
    keras.Input(shape=(1,), name="thal", dtype="string"),
    keras.Input(shape=(1,), name="age"),
    keras.Input(shape=(1,), name="trestbps"),
    keras.Input(shape=(1,), name="chol"),
    keras.Input(shape=(1,), name="thalach"),
    keras.Input(shape=(1,), name="oldpeak"),
    keras.Input(shape=(1,), name="slope"),
  ]
  feature_layer = layers.concatenate(
    [
        encode_categorical_feature(all_inputs[0], "sex", train_ds, False),
        encode_categorical_feature(all_inputs[1], "cp", train_ds, False),
        encode_categorical_feature(all_inputs[2], "fbs", train_ds, False),
        encode_categorical_feature(all_inputs[3], "restecg", train_ds, False),
        encode_categorical_feature(all_inputs[4], "exang", train_ds, False),
        encode_categorical_feature(all_inputs[5], "ca", train_ds, False),
        encode_categorical_feature(all_inputs[6], "thal", train_ds, True),
        normalize(all_inputs[7], "age", train_ds),
        normalize(all_inputs[8], "trestbps", train_ds),
        normalize(all_inputs[9], "chol", train_ds),
        normalize(all_inputs[10], "thalach", train_ds),
        normalize(all_inputs[11], "oldpeak", train_ds),
        normalize(all_inputs[12], "slope", train_ds)
    ]
  )
  h1_layer = layers.Dense(h1_units, activation=h1_activation)(feature_layer)
  output_layer = layers.Dense(1, activation="sigmoid")(h1_layer)
  model = keras.Model(inputs=all_inputs, outputs=output_layer)
  model.compile(keras.optimizers.SGD(learning_rate=learning_rate), "binary_crossentropy", metrics=["accuracy"])
  return model


h1_units=32
learning_rate=[0.01, 0.03]
h1_activation='relu'

accuracies = []

for lr in learning_rate:
  accuracies_per_param = []

  for i in range(len(folds)):
      # assign the folds to training and validation
      train = pd.concat(folds[0:i] + folds[i+1:])
      val = folds[i]
      train_ds_original = dataframe_to_dataset(train)
      val_ds_original = dataframe_to_dataset(val, shuffle=False)

      train_ds = train_ds_original.batch(32)
      val_ds = val_ds_original.batch(val.shape[0])

      model = create_model(h1_units, lr, h1_activation)

      model_result = model.fit(train_ds, epochs=10, validation_data=val_ds)

      # predict test set, the output is a probability, not an integer
      pred_val = model.predict(val_ds)

      # YOUR CODE HERE

      pred_val = (pred_val >= 0.5).astype(int)

      accuracies_per_param.append(accuracy_score(val['target'].values,pred_val))

  accuracies.append(accuracies_per_param)
```

```
Epoch 1/10
7/7 [==============================] - 1s 54ms/step - loss: 0.8852 - accuracy: 0.2921 - val_loss: 0.8221 - val_accuracy: 0.3564
Epoch 2/10
7/7 [==============================] - 0s 5ms/step - loss: 0.8289 - accuracy: 0.3218 - val_loss: 0.7723 - val_accuracy: 0.4257
Epoch 3/10
7/7 [==============================] - 0s 4ms/step - loss: 0.7829 - accuracy: 0.3762 - val_loss: 0.7310 - val_accuracy: 0.4851
Epoch 4/10
7/7 [==============================] - 0s 22ms/step - loss: 0.7447 - accuracy: 0.4455 - val_loss: 0.6965 - val_accuracy: 0.5842
Epoch 5/10
7/7 [==============================] - 0s 13ms/step - loss: 0.7129 - accuracy: 0.5099 - val_loss: 0.6677 - val_accuracy: 0.6238
Epoch 6/10
7/7 [==============================] - 0s 5ms/step - loss: 0.6861 - accuracy: 0.5743 - val_loss: 0.6429 - val_accuracy: 0.6535
Epoch 7/10
7/7 [==============================] - 0s 4ms/step - loss: 0.6630 - accuracy: 0.6089 - val_loss: 0.6216 - val_accuracy: 0.7129
Epoch 8/10
7/7 [==============================] - 0s 5ms/step - loss: 0.6430 - accuracy: 0.6386 - val_loss: 0.6029 - val_accuracy: 0.7228
Epoch 9/10
7/7 [==============================] - 0s 5ms/step - loss: 0.6254 - accuracy: 0.6782 - val_loss: 0.5863 - val_accuracy: 0.7525
Epoch 10/10
7/7 [==============================] - 0s 5ms/step - loss: 0.6099 - accuracy: 0.7129 - val_loss: 0.5716 - val_accuracy: 0.7624
1/1 [==============================] - 0s 222ms/step
Epoch 1/10
7/7 [==============================] - 1s 56ms/step - loss: 0.7247 - accuracy: 0.5248 - val_loss: 0.7662 - val_accuracy: 0.4653
Epoch 2/10
7/7 [==============================] - 0s 4ms/step - loss: 0.6661 - accuracy: 0.5941 - val_loss: 0.7117 - val_accuracy: 0.5050
Epoch 3/10
7/7 [==============================] - 0s 4ms/step - loss: 0.6207 - accuracy: 0.6485 - val_loss: 0.6690 - val_accuracy: 0.5941
Epoch 4/10
7/7 [==============================] - 0s 8ms/step - loss: 0.5850 - accuracy: 0.6683 - val_loss: 0.6349 - val_accuracy: 0.6238
Epoch 5/10
7/7 [==============================] - 0s 4ms/step - loss: 0.5564 - accuracy: 0.6931 - val_loss: 0.6073 - val_accuracy: 0.6733
Epoch 6/10
7/7 [==============================] - 0s 5ms/step - loss: 0.5329 - accuracy: 0.7178 - val_loss: 0.5846 - val_accuracy: 0.7030
Epoch 7/10
7/7 [==============================] - 0s 5ms/step - loss: 0.5134 - accuracy: 0.7376 - val_loss: 0.5657 - val_accuracy: 0.7129
Epoch 8/10
7/7 [==============================] - 0s 5ms/step - loss: 0.4969 - accuracy: 0.7525 - val_loss: 0.5498 - val_accuracy: 0.7129
Epoch 9/10
7/7 [==============================] - 0s 5ms/step - loss: 0.4828 - accuracy: 0.7673 - val_loss: 0.5361 - val_accuracy: 0.7327
Epoch 10/10
7/7 [==============================] - 0s 5ms/step - loss: 0.4705 - accuracy: 0.7723 - val_loss: 0.5243 - val_accuracy: 0.7624
1/1 [==============================] - 0s 232ms/step
Epoch 1/10
7/7 [==============================] - 2s 83ms/step - loss: 0.7700 - accuracy: 0.3960 - val_loss: 0.7301 - val_accuracy: 0.4257
Epoch 2/10
7/7 [==============================] - 0s 7ms/step - loss: 0.7118 - accuracy: 0.5396 - val_loss: 0.6852 - val_accuracy: 0.5743
Epoch 3/10
7/7 [==============================] - 0s 7ms/step - loss: 0.6663 - accuracy: 0.6188 - val_loss: 0.6490 - val_accuracy: 0.6535
Epoch 4/10
7/7 [==============================] - 0s 6ms/step - loss: 0.6301 - accuracy: 0.6436 - val_loss: 0.6216 - val_accuracy: 0.6931
Epoch 5/10
7/7 [==============================] - 0s 4ms/step - loss: 0.6008 - accuracy: 0.6733 - val_loss: 0.5987 - val_accuracy: 0.7525
Epoch 6/10
7/7 [==============================] - 0s 5ms/step - loss: 0.5769 - accuracy: 0.7327 - val_loss: 0.5798 - val_accuracy: 0.7723
Epoch 7/10
7/7 [==============================] - 0s 4ms/step - loss: 0.5571 - accuracy: 0.7327 - val_loss: 0.5640 - val_accuracy: 0.7624
Epoch 8/10
7/7 [==============================] - 0s 5ms/step - loss: 0.5404 - accuracy: 0.7376 - val_loss: 0.5506 - val_accuracy: 0.7624
```

**Question 2: How many total combinations of the above hyperparameter choices are there?**

**Answer:**

YOUR ANSWER HERE - The learning_rate list has 2 candidate values, And as the other hyperparameters are fixed, we get 2 combinations of hyperparameters.

Print the mean and standard deviation of the accuracy from cross validation for all hyperparams(across all *k* folds).

```
print(accuracies)
acc_mean = np.mean(accuracies, axis=1)
acc_std = np.std(accuracies, axis=1)

print(pd.DataFrame.from_dict(
    { "Params": learning_rate,
    "Accuracy Mean": acc_mean,
    "Accuracy Std ": acc_std,
}) )
```

```
[[0.7623762376237624, 0.7623762376237624, 0.7623762376237624], [0.8019801980198002, 0.7623762376237624, 0.8019801980198002]]
   Params  Accuracy Mean  Accuracy Std
0    0.01       0.762376   1.118223e-16
1    0.03       0.788779   1.866949e-02
```

**Question 3: If you increased the number of folds, do you expect the standard deviation of the accuracy across *k* folds to increase or decrease? Why?**

**Answer:** With increasing the # of folds, size of the training set increases and validation set decreases resulting in an increase in Std. Deviation.