KerasNeuralNetwork

September 29, 2021

1 Import packages

[2]:
```python
# import packages
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MinMaxScaler
from sklearn.preprocessing import StandardScaler
# packages for building the model
from tensorflow import keras
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Activation, Dense
from tensorflow.keras import losses
from tensorflow.keras import metrics
# packages for training model
from tensorflow.keras import optimizers
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.metrics import categorical_crossentropy
# misc package
import category_encoders as category_encoder
```

2 Import and view raw data

[3]:
```python
data = pd.read_csv('heart.csv')
```

[4]:
```python
data.columns
```

[4]:
```
Index(['Age', 'Sex', 'ChestPainType', 'RestingBP', 'Cholesterol', 'FastingBS',
       'RestingECG', 'MaxHR', 'ExerciseAngina', 'Oldpeak', 'ST_Slope',
       'HeartDisease'],
      dtype='object')
```

[5]:
```python
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 918 entries, 0 to 917
Data columns (total 12 columns):
```

```
 #   Column          Non-Null Count  Dtype
---  ------          --------------  -----
 0   Age             918 non-null    int64
 1   Sex             918 non-null    object
 2   ChestPainType   918 non-null    object
 3   RestingBP       918 non-null    int64
 4   Cholesterol     918 non-null    int64
 5   FastingBS       918 non-null    int64
 6   RestingECG      918 non-null    object
 7   MaxHR           918 non-null    int64
 8   ExerciseAngina  918 non-null    object
 9   Oldpeak         918 non-null    float64
 10  ST_Slope        918 non-null    object
 11  HeartDisease    918 non-null    int64
dtypes: float64(1), int64(6), object(5)
memory usage: 86.2+ KB
```

[6]:
```
# 918 samples, 12 features
data.shape
```

[6]:
```
(918, 12)
```

3 Preprocessing of Data

- Check for null values
- Check for duplicated values
- convert categorical data to numerical data
- Normalize data
- Split data into train and test samples/labels
- Convert datasets to numpy arrays to be able to pass data to Keras models

3.1 Preprocessing data: Check for null and/or duplicated values

[7]:
```
# check for missing values - there is no missing data
data.isnull().sum(axis=0)
```

[7]:
```
Age               0
Sex               0
ChestPainType     0
RestingBP         0
Cholesterol       0
FastingBS         0
RestingECG        0
MaxHR             0
ExerciseAngina    0
Oldpeak           0
ST_Slope          0
```

```
HeartDisease      0
dtype: int64
```

[8]:
```python
# check for duplicate values - there is no duplicated data
data.duplicated().sum(axis=0)
```

[8]: 0

3.2 Preprocessing data: View unique values for categorical features

[9]:
```python
category_columns = []

# viewing the unique values, number of dimensions, and shape of each column in
↪the data frame
for col in data.columns:
    if data[col].dtype == 'object':
        print(f'{col}')
        print(f'Values: {data[col].unique()}')
        print('\n')
        category_columns.append(col)
print(f'The following categories: {category_columns} shall be converted to
↪numerical data via pd.get_dummies')
```

```
Sex
Values: ['M' 'F']


ChestPainType
Values: ['ATA' 'NAP' 'ASY' 'TA']


RestingECG
Values: ['Normal' 'ST' 'LVH']


ExerciseAngina
Values: ['N' 'Y']


ST_Slope
Values: ['Up' 'Flat' 'Down']


The following categories: ['Sex', 'ChestPainType', 'RestingECG',
'ExerciseAngina', 'ST_Slope'] shall be converted to numerical data via
pd.get_dummies
```

[10]:
```
# labels - 0 does not have heart disease, 1 does have heart disease
data['HeartDisease'].unique()
```

[10]:
```
array([0, 1])
```

3.3 Preprocessing data: Converting categorical values into numerical values

[11]:
```
# Convert the nomimnal categorical label values to numerical values

# get_dummies method (one hot encoding technique for multi-value categorical␣
↪variables)
mod_data = data.copy()
mod_data = pd.get_dummies(mod_data, columns=category_columns)
mod_data.head(2)
```

[11]:
```
   Age  RestingBP  Cholesterol  FastingBS  MaxHR  Oldpeak  HeartDisease  \
0   40        140          289          0    172      0.0             0
1   49        160          180          0    156      1.0             1

   Sex_F  Sex_M  ChestPainType_ASY  …  ChestPainType_NAP  ChestPainType_TA  \
0      0      1                  0  …                  0                 0
1      1      0                  0  …                  1                 0

   RestingECG_LVH  RestingECG_Normal  RestingECG_ST  ExerciseAngina_N  \
0               0                  1              0                 1
1               0                  1              0                 1

   ExerciseAngina_Y  ST_Slope_Down  ST_Slope_Flat  ST_Slope_Up
0                 0              0              0            1
1                 0              0              1            0

[2 rows x 21 columns]
```

[12]:
```
mod_data.columns
```

[12]:
```
Index(['Age', 'RestingBP', 'Cholesterol', 'FastingBS', 'MaxHR', 'Oldpeak',
       'HeartDisease', 'Sex_F', 'Sex_M', 'ChestPainType_ASY',
       'ChestPainType_ATA', 'ChestPainType_NAP', 'ChestPainType_TA',
       'RestingECG_LVH', 'RestingECG_Normal', 'RestingECG_ST',
       'ExerciseAngina_N', 'ExerciseAngina_Y', 'ST_Slope_Down',
       'ST_Slope_Flat', 'ST_Slope_Up'],
      dtype='object')
```

[13]:
```
# separating features from output (HeartDisease)
features = [col for col in mod_data.columns if col != 'HeartDisease']
output = ['HeartDisease']
features, len(features)
```

[13]:
```
(['Age',
  'RestingBP',
  'Cholesterol',
  'FastingBS',
  'MaxHR',
  'Oldpeak',
  'Sex_F',
  'Sex_M',
  'ChestPainType_ASY',
  'ChestPainType_ATA',
  'ChestPainType_NAP',
  'ChestPainType_TA',
  'RestingECG_LVH',
  'RestingECG_Normal',
  'RestingECG_ST',
  'ExerciseAngina_N',
  'ExerciseAngina_Y',
  'ST_Slope_Down',
  'ST_Slope_Flat',
  'ST_Slope_Up'],
 20)
```

3.4 Preprocessing data: Convert data to numpy arrays

[14]:
```
X = np.array(mod_data[features])
y = np.array(mod_data[output])

len(features), X.shape, y.shape
```

[14]:
```
(20, (918, 20), (918, 1))
```

[15]:
```
type(X), type(y), X.dtype, y.dtype
```

[15]:
```
(numpy.ndarray, numpy.ndarray, dtype('float64'), dtype('int64'))
```

3.5 Preprocessing data: Splitting data into train and test samples/labels

[15]:
```
SEED = 2

# split dataframe into train samples/labels and test samples/labels
train_samples, test_samples, train_labels, test_labels = train_test_split(X, y,␣
↪test_size=0.20, random_state=SEED)
# convert the dataframes to numpy arrays (tensors)
train_labels = np.array(train_labels)
train_samples = np.array(train_samples)
test_labels = np.array(test_labels)
```

```
test_samples = np.array(test_samples)

train_samples.shape, train_labels.shape, len(features)
```

[15]: ((734, 20), (734, 1), 20)

4 Preprocessing data: Normalizing (z-scoring) data

[16]:
```
# converts the numerical values to all be within the range of [0, 1]
scaler = MinMaxScaler(feature_range=[0,1])
# fit and transform train data
scaled_train_samples = scaler.fit_transform(train_samples)
# transform but not fit test data to prevent test data bias leakage
scaled_test_samples = scaler.transform(test_samples)
# verify the data was transformed
scaled_train_samples, scaled_train_samples.shape, scaled_test_samples,␣
↪scaled_test_samples.shape
```

[16]:
```
(array([[0.44897959, 0.8       , 0.        , …, 0.        , 1.        ,
         0.        ],
        [0.12244898, 0.75      , 0.35489221, …, 0.        , 0.        ,
         1.        ],
        [0.67346939, 0.705     , 0.48424544, …, 0.        , 1.        ,
         0.        ],
        …,
        [0.46938776, 0.685     , 0.56218905, …, 0.        , 1.        ,
         0.        ],
        [0.67346939, 0.695     , 0.46932007, …, 0.        , 0.        ,
         1.        ],
        [0.6122449 , 0.675     , 0.3681592 , …, 0.        , 0.        ,
         1.        ]]),
 (734, 20),
 array([[0.48979592, 0.64      , 0.        , …, 0.        , 0.        ,
         1.        ],
        [0.30612245, 0.575     , 0.        , …, 0.        , 1.        ,
         0.        ],
        [0.87755102, 0.65      , 0.36650083, …, 0.        , 1.        ,
         0.        ],
        …,
        [0.71428571, 0.65      , 0.4212272 , …, 0.        , 1.        ,
         0.        ],
        [0.67346939, 0.65      , 0.        , …, 0.        , 1.        ,
         0.        ],
        [0.59183673, 0.6       , 0.58706468, …, 0.        , 0.        ,
         1.        ]]),
 (184, 20))
```

5 Build the Neural Network Model with Keras Sequential class

[17]:
```
# building a Sequential model - linear stack of layers
# init the model
model = Sequential()
# add Dense layers to the models
# units = number of nodes, input_shape = tensor shape the input layer expect␣
↪(inits weights); activation - activation function
model.add(Dense(units=16, activation='relu', input_shape=(20, )))
model.add(Dense(units=16, activation='relu'))
# use sigmoid in last layer because it is a binary classification problem
model.add(Dense(units=1, activation='sigmoid'))
```

5.1 Visualization of Neural Network Dense Layers

[18]:
```
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense (Dense)                (None, 16)                336
_________________________________________________________________
dense_1 (Dense)              (None, 16)                272
_________________________________________________________________
dense_2 (Dense)              (None, 1)                 17
=================================================================
Total params: 625
Trainable params: 625
Non-trainable params: 0
_________________________________________________________________
```

6 Compile the Model

- Assign loss function: the function that is minimized by the optimizer
- Assign optimizer function: how the model learns and minimizes the loss function
- Choose metrics: used to evaluate the performance of the model

[19]:
```
# compilation step - 1) loss function 2) optimizer 3) Metrics to monitor during␣
↪training and testing
model.compile(optimizer=optimizers.RMSprop(learning_rate=0.001),loss=losses.
↪binary_crossentropy,metrics=["accuracy"])

# alt ways to construct optimizer
# opt = Adam(learning_rate=0.01) # defining the optimizer
# model.compile(optimizer=opt, loss='binary_crossentropy', metrics=['accuracy'])
```

```
# model.compile(optimizer='rmsprop', loss='binary_crossentropy',␣
↪metrics=['accuracy'])
# model.compile(optimizer=optimizers.RMSprop(learning_rate=0.001),␣
↪loss='binary_crossentropy', metrics=['accuracy'])
```

7 Train the model

- Assign the train samples as x with their train labels as y
- Assign batch size, the number of samples that will be propagated through the network
- Assign epochs, the number of iterations the model will run through the layers

[20]:
```
# training the model by fitting the normalized training data
history = model.fit(x=scaled_train_samples, y=train_labels, batch_size=20,␣
↪epochs=30)
history
```

```
Epoch 1/30
37/37 [==============================] - 0s 703us/step - loss: 0.6031 -
accuracy: 0.7520
Epoch 2/30
37/37 [==============================] - 0s 836us/step - loss: 0.4909 -
accuracy: 0.8270
Epoch 3/30
37/37 [==============================] - 0s 738us/step - loss: 0.4123 -
accuracy: 0.8420
Epoch 4/30
37/37 [==============================] - 0s 916us/step - loss: 0.3705 -
accuracy: 0.8515
Epoch 5/30
37/37 [==============================] - 0s 819us/step - loss: 0.3537 -
accuracy: 0.8529
Epoch 6/30
37/37 [==============================] - 0s 840us/step - loss: 0.3438 -
accuracy: 0.8569
Epoch 7/30
37/37 [==============================] - 0s 976us/step - loss: 0.3373 -
accuracy: 0.8638
Epoch 8/30
37/37 [==============================] - 0s 1ms/step - loss: 0.3324 - accuracy:
0.8624
Epoch 9/30
37/37 [==============================] - 0s 745us/step - loss: 0.3292 -
accuracy: 0.8610
Epoch 10/30
37/37 [==============================] - 0s 804us/step - loss: 0.3283 -
accuracy: 0.8624
Epoch 11/30
```

```
37/37 [==============================] - 0s 960us/step - loss: 0.3260 -
accuracy: 0.8665
Epoch 12/30
37/37 [==============================] - 0s 833us/step - loss: 0.3231 -
accuracy: 0.8610
Epoch 13/30
37/37 [==============================] - 0s 806us/step - loss: 0.3228 -
accuracy: 0.8692
Epoch 14/30
37/37 [==============================] - 0s 915us/step - loss: 0.3184 -
accuracy: 0.8665
Epoch 15/30
37/37 [==============================] - 0s 883us/step - loss: 0.3206 -
accuracy: 0.8678
Epoch 16/30
37/37 [==============================] - 0s 837us/step - loss: 0.3206 -
accuracy: 0.8665
Epoch 17/30
37/37 [==============================] - 0s 955us/step - loss: 0.3174 -
accuracy: 0.8651
Epoch 18/30
37/37 [==============================] - 0s 916us/step - loss: 0.3173 -
accuracy: 0.8651
Epoch 19/30
37/37 [==============================] - 0s 893us/step - loss: 0.3165 -
accuracy: 0.8665
Epoch 20/30
37/37 [==============================] - 0s 881us/step - loss: 0.3164 -
accuracy: 0.8665
Epoch 21/30
37/37 [==============================] - 0s 1ms/step - loss: 0.3143 - accuracy:
0.8665
Epoch 22/30
37/37 [==============================] - 0s 1ms/step - loss: 0.3133 - accuracy:
0.8651
Epoch 23/30
37/37 [==============================] - 0s 928us/step - loss: 0.3113 -
accuracy: 0.8665
Epoch 24/30
37/37 [==============================] - 0s 917us/step - loss: 0.3095 -
accuracy: 0.8719
Epoch 25/30
37/37 [==============================] - 0s 1ms/step - loss: 0.3114 - accuracy:
0.8760
Epoch 26/30
37/37 [==============================] - 0s 968us/step - loss: 0.3079 -
accuracy: 0.8665
Epoch 27/30
```

```
37/37 [==============================] - 0s 761us/step - loss: 0.3082 -
accuracy: 0.8706
Epoch 28/30
37/37 [==============================] - 0s 874us/step - loss: 0.3081 -
accuracy: 0.8665
Epoch 29/30
37/37 [==============================] - 0s 1ms/step - loss: 0.3068 - accuracy:
0.8706
Epoch 30/30
37/37 [==============================] - 0s 931us/step - loss: 0.3060 -
accuracy: 0.8733
```

```
[20]: <keras.callbacks.History at 0x7fa6baecc100>
```

8 Evaluate the Model's performance

- The evaluate method takes in a test sample numpy array and their associated test labels
- Returns the loss value & metrics value (accuracy score) for the model in test mode
- Loss is the scalar value that is attempted to be minimized during training of the model. The lower the loss, the closer our predictions are to the true labels.

```
[21]: results = model.evaluate(scaled_test_samples, test_labels)
      results
```

```
6/6 [==============================] - 0s 742us/step - loss: 0.3765 - accuracy:
0.8315
```

```
[21]: [0.37645289301872253, 0.83152174949646]
```

9 Further Experiments

-

9.1 Hidden Layers

  - Try using one or three hidden layers, and see how doing so affects validation and test accuracy.

-

9.2 Hidden Units

  - Try using layers with more hidden units or fewer hidden units: 32 units, 64 units, and so on.

-

9.3 Loss Functions

- Try using the mse loss function instead of binary_crossentropy.

•

9.4 Activation Function

- Try using the tanh activation (an activation that was popular in the early days of neural networks)

9.5 Reusable function for creating Keras Sequential Models

[17]:
```python
DEFAULT_LAYERS = [
          Dense(units=16, activation='relu', input_shape=(20, )),
          Dense(units=32, activation='relu'),
          Dense(units=1, activation='sigmoid')
]
DEFAULT_OPTIMIZER = optimizers.RMSprop(learning_rate=0.001)
DEFAULT_LOSS = losses.binary_crossentropy
DEFAULT_METRICS = ["accuracy"]
```

[18]:
```python
def create_neural_network(scaled_train_samples,
                          train_labels,
                          layers = DEFAULT_LAYERS,
                          optimizer=DEFAULT_OPTIMIZER,
                          loss=DEFAULT_LOSS,
                          metrics=DEFAULT_METRICS):
    model = Sequential(layers)
    model.compile(optimizer=optimizer, loss=loss,metrics=metrics)
    model.fit(x=scaled_train_samples, y=train_labels, batch_size=20, epochs=30,␣
 ↪verbose=0)
    return model

def get_nn_results(model, scaled_test_samples, test_labels):
    return model.evaluate(scaled_test_samples, test_labels)
```

9.6 Further Experiments: Hidden Layers

[24]:
```python
layers_dict = {
    # 1 hidden layer
    '1': [
          Dense(units=1, activation='sigmoid')
         ],
    # 2 hidden layers
    '2': [
          Dense(units=16, activation='relu', input_shape=(20, )),
          Dense(units=1, activation='sigmoid')
```

```
        ],
    # 3 hidden layers
    '3': [
           Dense(units=16, activation='relu', input_shape=(20, )),
           Dense(units=16, activation='relu'),
           Dense(units=1, activation='sigmoid')
         ]
}
results_layers_dict = {}
```

[25]:

```
SEED = 2

train_samples, test_samples, train_labels, test_labels = train_test_split(X, y,␣
 ↪test_size=0.20, random_state=SEED)
# convert the dataframes to numpy arrays (tensors)
train_labels = np.array(train_labels)
train_samples = np.array(train_samples)
test_labels = np.array(test_labels)
test_samples = np.array(test_samples)

# converts the numerical values to all be within the range of [0, 1]
scaler = MinMaxScaler(feature_range=[0,1])
# fit and transform train data
scaled_train_samples = scaler.fit_transform(train_samples)
# transform but not fit test data to prevent test data bias leakage
scaled_test_samples = scaler.transform(test_samples)
# verify the data was transformed
scaled_train_samples, scaled_train_samples.shape, scaled_test_samples,␣
 ↪scaled_test_samples.shape

for key, layers in layers_dict.items():
    curr_model = create_neural_network(scaled_train_samples, train_labels,␣
 ↪layers)
    results_layers_dict[key] = get_nn_results(curr_model, scaled_test_samples,␣
 ↪test_labels)

print('\n')
for key, value in results_layers_dict.items():
    print(f'Achieved {value[0]:.4f} loss {value[1]:.4f} accuracy with {key}␣
 ↪layer(s)')
```

```
6/6 [==============================] - 0s 1ms/step - loss: 0.3875 - accuracy:
0.8641
6/6 [==============================] - 0s 940us/step - loss: 0.3684 - accuracy:
0.8696
6/6 [==============================] - 0s 968us/step - loss: 0.3511 - accuracy:
0.8587
```

```
Achieved 0.3875 loss 0.8641 accuracy with 1 layer(s)
Achieved 0.3684 loss 0.8696 accuracy with 2 layer(s)
Achieved 0.3511 loss 0.8587 accuracy with 3 layer(s)
```

9.7 Further Experiments: Hidden Units

[26]:
```
units_dict = {
    # 16 hidden layer
    '16': [
          Dense(units=16, activation='relu', input_shape=(20, )),
          Dense(units=16, activation='relu'),
          Dense(units=1, activation='sigmoid')
         ],
    # 32 hidden layers
    '32': [
          Dense(units=32, activation='relu', input_shape=(20, )),
          Dense(units=32, activation='relu'),
          Dense(units=1, activation='sigmoid')
         ],
    # 64 hidden layers
    '64': [
          Dense(units=64, activation='relu', input_shape=(20, )),
          Dense(units=64, activation='relu'),
          Dense(units=1, activation='sigmoid')
         ]
}
results_units_dict = {}
```

[27]:
```
SEED = 2

train_samples, test_samples, train_labels, test_labels = train_test_split(X, y,␣
 ↪test_size=0.20, random_state=SEED)
# convert the dataframes to numpy arrays (tensors)
train_labels = np.array(train_labels)
train_samples = np.array(train_samples)
test_labels = np.array(test_labels)
test_samples = np.array(test_samples)

# converts the numerical values to all be within the range of [0, 1]
scaler = MinMaxScaler(feature_range=[0,1])
# fit and transform train data
scaled_train_samples = scaler.fit_transform(train_samples)
# transform but not fit test data to prevent test data bias leakage
scaled_test_samples = scaler.transform(test_samples)
# verify the data was transformed
```

```
scaled_train_samples, scaled_train_samples.shape, scaled_test_samples,␣
↪scaled_test_samples.shape

for key, layers in units_dict.items():
    curr_model = create_neural_network(scaled_train_samples, train_labels,␣
↪layers)
    results_units_dict[key] = get_nn_results(curr_model, scaled_test_samples,␣
↪test_labels)

print('\n')
for key, value in results_units_dict.items():
    print(f'Achieved {value[0]:.4f} loss {value[1]:.4f} accuracy with {key}␣
↪units/nodes per deep layer')
```

```
6/6 [==============================] - 0s 1ms/step - loss: 0.3605 - accuracy:
0.8587
6/6 [==============================] - 0s 816us/step - loss: 0.3748 - accuracy:
0.8370
6/6 [==============================] - 0s 846us/step - loss: 0.3888 - accuracy:
0.8370


Achieved 0.3605 loss 0.8587 accuracy with 16 units/nodes per deep layer
Achieved 0.3748 loss 0.8370 accuracy with 32 units/nodes per deep layer
Achieved 0.3888 loss 0.8370 accuracy with 64 units/nodes per deep layer
```

9.8 Further Experiments: Loss Functions

[28]:
```
loss_functions = ['binary_crossentropy', 'hinge', 'squared_hinge']
results_loss_dict = {}
```

[29]:
```
SEED = 2

train_samples, test_samples, train_labels, test_labels = train_test_split(X, y,␣
↪test_size=0.20, random_state=SEED)
# convert the dataframes to numpy arrays (tensors)
train_labels = np.array(train_labels)
train_samples = np.array(train_samples)
test_labels = np.array(test_labels)
test_samples = np.array(test_samples)


# converts the numerical values to all be within the range of [0, 1]
scaler = MinMaxScaler(feature_range=[0,1])
# fit and transform train data
scaled_train_samples = scaler.fit_transform(train_samples)
# transform but not fit test data to prevent test data bias leakage
```

```
scaled_test_samples = scaler.transform(test_samples)
# verify the data was transformed
scaled_train_samples, scaled_train_samples.shape, scaled_test_samples,␣
↪scaled_test_samples.shape

for loss_func in loss_functions:
    curr_model = create_neural_network(scaled_train_samples, train_labels,␣
↪loss=loss_func)
    results_loss_dict[loss_func] = get_nn_results(curr_model,␣
↪scaled_test_samples, test_labels)

print('\n')
for key, value in results_loss_dict.items():
    print(f'Achieved {value[0]:.4f} loss {value[1]:.4f} accuracy with the {key}␣
↪loss function')
```

```
6/6 [==============================] - 0s 854us/step - loss: 0.3638 - accuracy:
0.8696
6/6 [==============================] - 0s 916us/step - loss: 0.6380 - accuracy:
0.8478
6/6 [==============================] - 0s 739us/step - loss: 0.7288 - accuracy:
0.8370


Achieved 0.3638 loss 0.8696 accuracy with the binary_crossentropy loss function
Achieved 0.6380 loss 0.8478 accuracy with the hinge loss function
Achieved 0.7288 loss 0.8370 accuracy with the squared_hinge loss function
```

9.9 Changing feature range to be [-1,1] for tanh activation

[21]:
```
SEED = 2
tanh_loss_dict = {}
train_samples, test_samples, train_labels, test_labels = train_test_split(X, y,␣
↪test_size=0.20, random_state=SEED)
# convert the dataframes to numpy arrays (tensors)
train_labels = np.array(train_labels)
train_samples = np.array(train_samples)
test_labels = np.array(test_labels)
test_samples = np.array(test_samples)
layers = [
          Dense(units=16, activation='relu', input_shape=(20, )),
          Dense(units=32, activation='relu'),
          Dense(units=1, activation='tanh')
]

# converts the numerical values to all be within the range of [-1, 1]
scaler = MinMaxScaler(feature_range=[-1,1])
```

```
# fit and transform train data
scaled_train_samples = scaler.fit_transform(train_samples)
# transform but not fit test data to prevent test data bias leakage
scaled_test_samples = scaler.transform(test_samples)
# verify the data was transformed
scaled_train_samples, scaled_train_samples.shape, scaled_test_samples,␣
↪scaled_test_samples.shape

curr_model = create_neural_network(scaled_train_samples, train_labels,␣
↪loss=losses.binary_crossentropy)
tanh_loss_dict['tanh'] = get_nn_results(curr_model, scaled_test_samples,␣
↪test_labels)

tanh_loss_dict
```

```
6/6 [==============================] - 0s 843us/step - loss: 0.4098 - accuracy:
0.8533
```

[21]:

```
{'tanh': [0.40980401635169983, 0.85326087474823]}
```

9.10 Further Experiments: Activation Functions

[30]:

```
activation_functions_dict = {
    # sigmoid
    'sigmoid': [
          Dense(units=16, activation='relu', input_shape=(20, )),
          Dense(units=16, activation='relu'),
          Dense(units=1, activation='sigmoid')
         ],
    # tanh
    'tanh': [
          Dense(units=16, activation='relu', input_shape=(20, )),
          Dense(units=16, activation='relu'),
          Dense(units=1, activation='tanh')
         ],
    # relu
    'relu': [
          Dense(units=16, activation='relu', input_shape=(20, )),
          Dense(units=16, activation='relu'),
          Dense(units=1, activation='relu')
         ]
}
results_activation_functions_dict = {}
```

[31]:

```
SEED = 2
```

```
train_samples, test_samples, train_labels, test_labels = train_test_split(X, y,␣
↪test_size=0.20, random_state=SEED)
# convert the dataframes to numpy arrays (tensors)
train_labels = np.array(train_labels)
train_samples = np.array(train_samples)
test_labels = np.array(test_labels)
test_samples = np.array(test_samples)

# converts the numerical values to all be within the range of [0, 1]
scaler = MinMaxScaler(feature_range=[0,1])
# fit and transform train data
scaled_train_samples = scaler.fit_transform(train_samples)
# transform but not fit test data to prevent test data bias leakage
scaled_test_samples = scaler.transform(test_samples)
# verify the data was transformed
scaled_train_samples, scaled_train_samples.shape, scaled_test_samples,␣
↪scaled_test_samples.shape

for key, layers in activation_functions_dict.items():
    curr_model = create_neural_network(scaled_train_samples, train_labels,␣
↪layers)
    results_activation_functions_dict[key] = get_nn_results(curr_model,␣
↪scaled_test_samples, test_labels)

print('\n')
for key, value in results_activation_functions_dict.items():
    print(f'Achieved {value[0]:.4f} loss {value[1]:.4f} accuracy with the {key}␣
↪activation function.')
```

```
6/6 [==============================] - 0s 792us/step - loss: 0.3668 - accuracy:
0.8533
6/6 [==============================] - 0s 721us/step - loss: 0.4986 - accuracy:
0.8207
6/6 [==============================] - 0s 1ms/step - loss: 0.7425 - accuracy:
0.8370


Achieved 0.3668 loss 0.8533 accuracy with the sigmoid activation function.
Achieved 0.4986 loss 0.8207 accuracy with the tanh activation function.
Achieved 0.7425 loss 0.8370 accuracy with the relu activation function.
```

[ ]: