Deep Neural Networks Laboration

Data used in this laboration are from the Kitsune Network Attack Dataset, https://archive.ics.uci.edu/ml/datasets/Kitsune+Network+Attack+Dataset . We will focus on the 'Mirai' part of the dataset. Your task is to make a DNN that can classify if each attack is benign or malicious. The dataset has 116 covariates, but to make it a bit more difficult we will remove the first 24 covariates.

You need to answer all questions in this notebook.

If the training is too slow on your own computer, use the smaller datasets (*half or* quarter).

Dense networks are not optimal for tabular datasets like the one used here, but here the main goal is to learn deep learning.

Part 1: Get the data

Skip this part if you load stored numpy arrays (Mirai*.npy) (which is recommended)

Use `wget` in the terminal of your cloud machine (in the same directory as where you have saved this notebook) to download the data, i.e.

wget https://archive.ics.uci.edu/ml/machine-learning-databases/00516/mirai/Mirai_dataset.csv.gz

wget https://archive.ics.uci.edu/ml/machine-learning-databases/00516/mirai/Mirai_labels.csv.gz

Then unpack the files using `gunzip` in the terminal, i.e.

gunzip Mirai_dataset.csv.gz

gunzip Mirai_labels.csv.gz

Part 2: Get a graphics card

Skip this part if you run on the CPU (recommended)

Lets make sure that our script can see the graphics card that will be used. The graphics cards will perform all the time consuming calculations in every training iteration.

```
import os
import warnings

# Ignore FutureWarning from numpy
warnings.simplefilter(action='ignore', category=FutureWarning)
```

```
import keras.backend as K
import tensorflow as tf

os.environ["CUDA_DEVICE_ORDER"]="PCI_BUS_ID";

# The GPU id to use, usually either "0" or "1";
os.environ["CUDA_VISIBLE_DEVICES"]="0";

# Allow growth of GPU memory, otherwise it will always look like all
the memory is being used
physical_devices = tf.config.experimental.list_physical_devices('GPU')
tf.config.experimental.set_memory_growth(physical_devices[0], True)

----------------------------------------------------------------------
-----
IndexError                                Traceback (most recent call
last)
Cell In[155], line 17
     15 # Allow growth of GPU memory, otherwise it will always look
like all the memory is being used
     16 physical_devices =
tf.config.experimental.list_physical_devices('GPU')
---> 17 tf.config.experimental.set_memory_growth(physical_devices[0],
True)

IndexError: list index out of range
```

Part 3: Hardware

In deep learning, the computer hardware is very important. You should always know what kind of hardware you are working on. Lets pretend that everyone is using an Nvidia RTX 3090 graphics card.

Question 1: Google the name of the graphics card, how many CUDA cores does it have?

> 10,496

Question 2: How much memory does the graphics card have?

> 24GB

Question 3: What is stored in the GPU memory while training a DNN?

> Current batch data, weight/parameter values, current forward and backward propagation values

Part 4: Load the data

To make this step easier, directly load the data from saved numpy arrays (.npy) (recommended)

Load the dataset from the csv files, it will take some time since it is almost 1.4 GB. (not recommended, unless you want to learn how to do it)

We will use the function `genfromtxt` to load the data. (not recommended, unless you want to learn how to do it)

https://docs.scipy.org/doc/numpy/reference/generated/numpy.genfromtxt.html

Load the data from csv files the first time, then save the data as numpy files for faster loading the next time.

Remove the first 24 covariates to make the task harder.

```
from numpy import genfromtxt # Not needed if you load data from numpy arrays
import numpy as np

# Load data from numpy arrays, choose reduced files if the training takes too long
X = np.load('Mirai_data.npy')
Y = np.load('Mirai_labels.npy')

# Remove the first 24 covariates (columns)
X = X[:, 24:]

print('The covariates have size {}.'.format(X.shape))
print('The labels have size {}.'.format(Y.shape))

# Print the number of examples of each class
class_counts = np.unique(Y, return_counts=True)

print()
for class_i in range(len(class_counts[0])):
    print(f"Number of class {class_counts[0][class_i]}: {class_counts[1][class_i]}")
```

```
The covariates have size (764137, 92).
The labels have size (764137,).

Number of class 0.0: 121621
Number of class 1.0: 642516
```

Part 5: How good is a naive classifier?

Question 4: Given the number of examples from each class, how high classification performance can a naive classifier obtain? The naive classifier will assume that all examples belong to one class. Note: you do not need to make a naive classifier, this is a theoretical question, just to understand how good performance we can obtain by guessing that all examples belong to one class.

> In this problem, there are only two classes. About 84% of the rows belong to class 1, so a naive classifier predicting everything to be class 1 would have 84% accuracy.

In all classification tasks you should always ask these questions

- How good classification accuracy can a naive classifier obtain? The naive classifier will assume that all examples belong to one class.
- What is random chance classification accuracy if you randomly guess the label of each (test) example? For a balanced dataset and binary classification this is easy (50%), but in many cases it is more complicated and a Monte Carlo simulation may be required to estimate random chance accuracy.

If your classifier cannot perform better than a naive classifier or a random classifier, you are doing something wrong.

```
# It is common to have NaNs in the data, lets check for it. Hint:
np.isnan()

# Print the number of NaNs (not a number) in the labels
print(f"Number of NaNs in Y: {np.sum(np.isnan(Y))}")

# Print the number of NaNs in the covariates
print(f"Number of NaNs in X: {np.sum(np.isnan(X))}")
```

```
Number of NaNs in Y: 0
Number of NaNs in X: 0
```

Part 6: Preprocessing

Lets do some simple preprocessing

```
# Convert covariates to floats
X = X.astype(float)

# Convert labels to integers
Y = Y.astype(int)
```

```python
# Remove mean of each covariate (column)
means = np.mean(X, axis=0, keepdims=True)
# means = means.repeat(X.shape[0], axis=0)

stds = np.std(X, axis=0).reshape(-1, X.shape[1])
stds = stds.repeat(X.shape[0], axis=0)

X = X - means

# Divide each covariate (column) by its standard deviation
X = X / stds

# Check that mean is 0 and standard deviation is 1 for all covariates,
by printing mean and std
print("Column means:\n")
print(np.mean(X, axis=0))
print("\nColumn stds:\n")
print(np.std(X, axis=0))
```

```
Column means:

[-3.19451533e-18 -6.30927527e-14  1.19963828e-13  4.56743018e-15
  4.08813918e-14  1.46461039e-13  5.65402045e-16 -1.69587525e-14
 -3.03376191e-13  1.25514109e-12 -2.72042402e-12 -1.10780892e-13
 -1.22468718e-13 -1.70290612e-13 -1.02139901e-14 -2.32208048e-12
  1.40695736e-12  1.20673259e-12 -1.05095447e-13  6.81889584e-14
 -1.00490973e-13  5.98862427e-14 -1.01547416e-12 -1.66283323e-12
 -1.58597771e-12 -1.31674067e-13  4.43360813e-13  8.41389037e-13
  5.77665264e-14 -4.50766872e-13 -2.54973195e-12  3.12056823e-13
 -1.53665212e-13  1.69273859e-12  9.50945604e-13  1.50953004e-13
 -1.01059397e-12 -5.11453792e-13 -1.86373908e-12 -2.09806690e-13
  1.03169903e-12 -1.47389966e-12 -1.69587525e-14 -1.64918984e-16
 -5.13325984e-14 -1.02166240e-14 -1.74685907e-15  1.34329189e-13
  5.98601714e-14  1.48745574e-17 -4.24927612e-13  5.77728088e-14
  1.25638129e-15  1.71850347e-13  1.50955720e-13  2.14478905e-14
  3.65405571e-14  1.21380412e-13 -9.10989074e-13 -6.30800138e-13
 -1.58038622e-12  2.62938522e-13 -7.57219424e-15 -2.89359393e-14
 -3.88851503e-13 -1.52984352e-12 -1.03687066e-12  2.75437407e-13
  2.44539294e-13 -6.73050928e-15  1.07511476e-13  2.60284113e-13
 -2.18130768e-13 -1.18954843e-12 -2.82172408e-12  5.45994503e-14
  5.46183481e-15  3.71306144e-14  2.33292940e-13 -1.73194638e-12
 -1.42020216e-13 -1.71843058e-12  5.29632937e-13 -3.21444033e-14
 -4.59764043e-14  3.53584091e-13 -1.44369337e-12 -1.26142028e-13
  1.25980728e-13  1.20260434e-13  4.34647402e-14 -4.07680570e-14]

Column stds:

[1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.
1.
 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.
```

```
1.
 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.
1.
 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]
```

Part 7: Split the dataset

Use the first 70% of the dataset for training, leave the other 30% for validation and test, call the variables

Xtrain (70%)

Xtemp (30%)

Ytrain (70%)

Ytemp (30%)

We use a function from scikit learn. https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.train_test_split.html

```
from sklearn.model_selection import train_test_split

# Your code to split the dataset
Xtrain, Xtemp, Ytrain, Ytemp = train_test_split(X, Y, test_size=0.3)

print('Xtrain has size {}.'.format(Xtrain.shape))
print('Ytrain has size {}.'.format(Ytrain.shape))

print('Xtemp has size {}.'.format(Xtemp.shape))
print('Ytemp has size {}.'.format(Ytemp.shape))

# Print the number of examples of each class, for the training data
and the remaining 30%
train_class_counts = np.unique(Ytrain, return_counts=True)
temp_class_counts = np.unique(Ytemp, return_counts=True)

print("\nTraining data:")
for class_i in range(len(train_class_counts[0])):
    print(f"Number of class {train_class_counts[0][class_i]}:
{train_class_counts[1][class_i]}")

print("\nTemp data:")
for class_i in range(len(temp_class_counts[0])):
    print(f"Number of class {temp_class_counts[0][class_i]}:
{temp_class_counts[1][class_i]}")
```

```
Xtrain has size (534895, 92).
Ytrain has size (534895,).
Xtemp has size (229242, 92).
Ytemp has size (229242,).

Training data:
Number of class 0: 85070
Number of class 1: 449825

Temp data:
Number of class 0: 36551
Number of class 1: 192691
```

Part 8: Split non-training data data into validation and test

Now split your non-training data (Xtemp, Ytemp) into 50% validation (Xval, Yval) and 50% testing (Xtest, Ytest), we use a function from scikit learn. In total this gives us 70% for training, 15% for validation, 15% for test.

https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.train_test_split.html

Do all variables (Xtrain,Ytrain), (Xval,Yval), (Xtest,Ytest) have the shape that you expect?

```
from sklearn.model_selection import train_test_split

# Your code
Xval, Xtest, Yval, Ytest = train_test_split(Xtemp, Ytemp,
test_size=0.5)

print('The validation and test data have size {}, {}, {} and
{}'.format(Xval.shape, Xtest.shape, Yval.shape, Ytest.shape))

The validation and test data have size (114621, 92), (114621, 92),
(114621,) and (114621,)
```

Part 9: DNN classification

Finish this code to create a first version of the classifier using a DNN. Start with a simple network with 2 dense layers (with 20 nodes each), using sigmoid activation functions. The final dense layer should have a single node and a sigmoid activation function. We start with the SGD optimizer.

For different parts of this notebook you need to go back here, add more things, and re-run this cell to re-define the build function.

Relevant functions are

`model.add()`, adds a layer to the network

`Dense()`, a dense network layer

`model.compile()`, compile the model, add " metrics=['accuracy'] " to print the classification accuracy during the training

See https://keras.io/layers/core/ for information on how the `Dense()` function works

Import a relevant cost / loss function for binary classification from keras.losses (https://keras.io/losses/)

See the following links for how to compile, train and evaluate the model

https://keras.io/api/models/model_training_apis/#compile-method

https://keras.io/api/models/model_training_apis/#fit-method

https://keras.io/api/models/model_training_apis/#evaluate-method

Make sure that the last layer always has a sigmoid activation function (why?).

```
from keras.models import Sequential, Model
from keras.layers import Input, Dense, BatchNormalization, Dropout
from tensorflow.keras.optimizers import SGD, Adam
from keras.losses import BinaryCrossentropy

# Set seed from random number generator, for better comparisons
from numpy.random import seed
seed(123)

def build_DNN(input_shape, n_layers, n_nodes, act_fun='sigmoid',
optimizer='sgd', learning_rate=0.01,
              use_bn=False, use_dropout=False,
use_custom_dropout=False):

    # Setup optimizer, depending on input parameter string
    if optimizer == "sgd":
        opt = SGD(learning_rate=learning_rate)
    elif optimizer == "adam":
        opt = Adam(learning_rate=learning_rate)

    # Setup a sequential model
    model = Sequential()

    # Add layers to the model, using the input parameters of the
build_DNN function
    # Add first layer, requires input shape
    model.add(Input(shape=input_shape))
```

```
    # Add remaining layers, do not require input shape
    for i in range(n_layers-1):
        model.add(Dense(n_nodes, activation=act_fun))

        if use_bn:
            model.add(BatchNormalization())

        if use_dropout:
            model.add(Dropout(rate=0.5))
        elif use_custom_dropout:
            model.add(myDropout(rate=0.5))

    # Add final layer
    model.add(Dense(1, activation="sigmoid"))

    # Compile model
    model.compile(loss=BinaryCrossentropy(), optimizer=opt,
metrics=["accuracy"])

    return model

# Lets define a help function for plotting the training results

import matplotlib.pyplot as plt
def plot_results(history):

    val_loss = history.history['val_loss']
    acc = history.history['accuracy']
    loss = history.history['loss']
    val_acc = history.history['val_accuracy']

    plt.figure(figsize=(10,4))
    plt.xlabel('Epochs')
    plt.ylabel('Loss')
    plt.plot(loss)
    plt.plot(val_loss)
    plt.legend(['Training','Validation'])

    plt.figure(figsize=(10,4))
    plt.xlabel('Epochs')
    plt.ylabel('Accuracy')
    plt.plot(acc)
    plt.plot(val_acc)
    plt.legend(['Training','Validation'])

    plt.show()
```

Part 10: Train the DNN

Time to train the DNN, we start simple with 2 layers with 20 nodes each, learning rate 0.1.

Relevant functions

`build_DNN`, the function we defined in Part 9, call it with the parameters you want to use

`model.fit()`, train the model with some training data

`model.evaluate()`, apply the trained model to some test data

See the following links for how to train and evaluate the model

https://keras.io/api/models/model_training_apis/#fit-method

https://keras.io/api/models/model_training_apis/#evaluate-method

Make sure that you are using learning rate 0.1 !

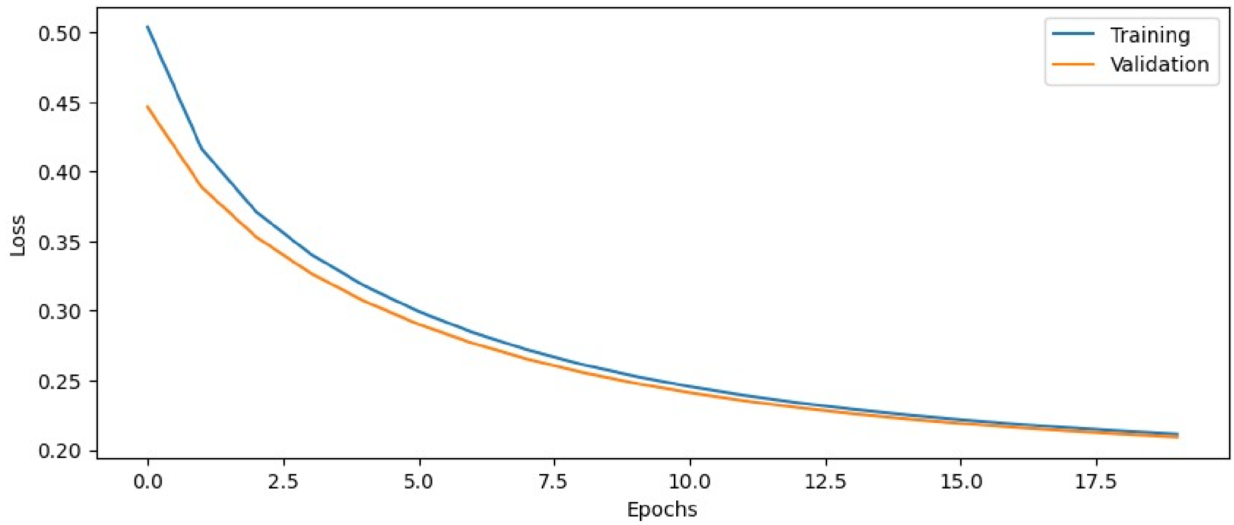
2 layers, 20 nodes

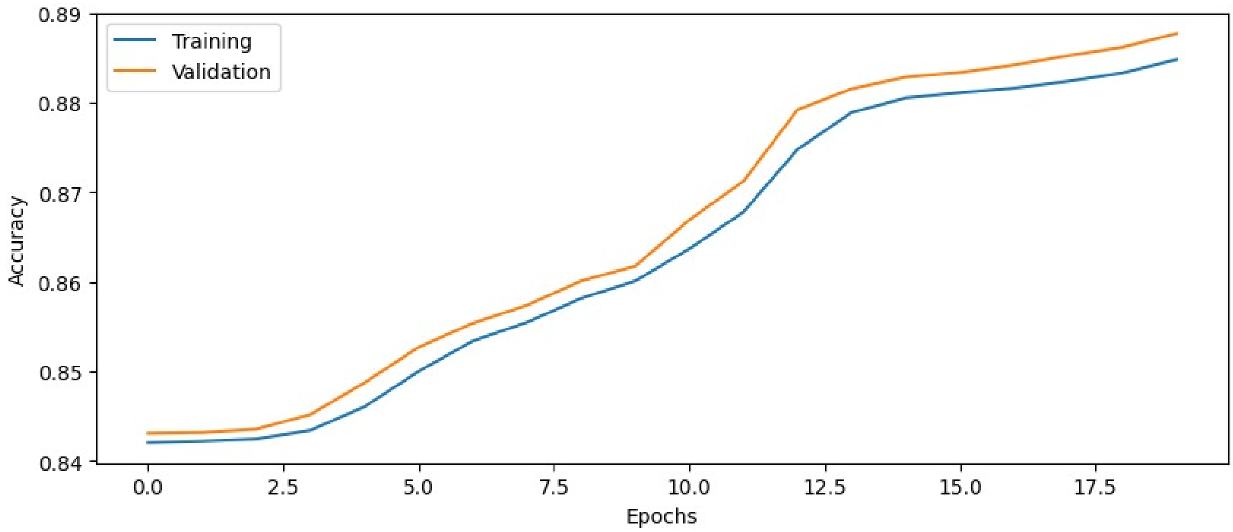
```
# Setup some training parameters
batch_size = 10000
epochs = 20

input_shape = (Xtrain.shape[1],)

# Build the model
model1 = build_DNN(input_shape=input_shape, n_layers=2, n_nodes=20)

# Train the model, provide training data and validation data
history1 = model1.fit(x=Xtrain, y=Ytrain, validation_data=(Xval,
Yval), batch_size=batch_size, epochs=epochs)
```

```
Epoch 1/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 2s 11ms/step - accuracy: 0.8424 - loss:
0.5363 - val_accuracy: 0.8431 - val_loss: 0.4463
Epoch 2/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.8419 - loss:
0.4305 - val_accuracy: 0.8432 - val_loss: 0.3887
Epoch 3/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.8419 - loss:
0.3804 - val_accuracy: 0.8436 - val_loss: 0.3532
Epoch 4/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.8427 - loss:
0.3477 - val_accuracy: 0.8452 - val_loss: 0.3272
Epoch 5/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.8462 - loss:
0.3221 - val_accuracy: 0.8487 - val_loss: 0.3068
Epoch 6/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.8495 - loss:
```

```
0.3029 - val_accuracy: 0.8527 - val_loss: 0.2903
Epoch 7/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.8529 - loss:
0.2875 - val_accuracy: 0.8554 - val_loss: 0.2767
Epoch 8/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.8554 - loss:
0.2743 - val_accuracy: 0.8574 - val_loss: 0.2653
Epoch 9/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.8577 - loss:
0.2634 - val_accuracy: 0.8601 - val_loss: 0.2559
Epoch 10/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.8596 - loss:
0.2551 - val_accuracy: 0.8618 - val_loss: 0.2479
Epoch 11/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.8614 - loss:
0.2477 - val_accuracy: 0.8669 - val_loss: 0.2411
Epoch 12/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.8662 - loss:
0.2415 - val_accuracy: 0.8712 - val_loss: 0.2354
Epoch 13/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.8736 - loss:
0.2350 - val_accuracy: 0.8792 - val_loss: 0.2304
Epoch 14/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.8781 - loss:
0.2304 - val_accuracy: 0.8815 - val_loss: 0.2261
Epoch 15/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.8805 - loss:
0.2256 - val_accuracy: 0.8829 - val_loss: 0.2224
Epoch 16/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.8815 - loss:
0.2217 - val_accuracy: 0.8834 - val_loss: 0.2191
Epoch 17/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.8812 - loss:
0.2198 - val_accuracy: 0.8842 - val_loss: 0.2162
Epoch 18/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.8824 - loss:
0.2166 - val_accuracy: 0.8852 - val_loss: 0.2136
Epoch 19/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.8834 - loss:
0.2137 - val_accuracy: 0.8862 - val_loss: 0.2113
Epoch 20/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.8853 - loss:
0.2112 - val_accuracy: 0.8877 - val_loss: 0.2093

# Evaluate the model on the test data
score = model1.evaluate(x=Xtest, y=Ytest)

print('Test loss: %.4f' % score[0])
print('Test accuracy: %.4f' % score[1])
```

```
   1/3582 ━━━━━━━━━━━━━━━━━━━━ 2:37 44ms/step - accuracy: 0.8125 -
loss: 0.2833

3582/3582 ━━━━━━━━━━━━━━━━━━━━ 4s 1ms/step - accuracy: 0.8889 - loss:
0.2080
Test loss: 0.2090
Test accuracy: 0.8875

# Plot the history from the training run
plot_results(history1)
```





Part 11: More questions

Question 5: What happens if you add several Dense layers without specifying the activation function?

The model will be linear, and adding more layers doesn't add more complexity. Mathematically it will reduce to a linear regression model.

Question 6: How are the weights in each dense layer initialized as default? How are the bias weights initialized?

By default in the Dense layers in Keras, bias weights are initialized as zero and kernel weights are initialized using the Glorot uniform initializer. The Glorot uniform initializer "draws samples from a uniform distribution within [-limit, limit], where limit = sqrt(6 / (fan_in + fan_out)) (fan_in is the number of input units in the weight tensor and fan_out is the number of output units)."

Source: https://keras.io/api/layers/initializers/

Part 12: Balancing the classes

This dataset is rather unbalanced, we need to define class weights so that the training pays more attention to the class with fewer samples. We use a function in scikit learn

https://scikit-learn.org/stable/modules/generated/sklearn.utils.class_weight.compute_class_weight.html

You need to call the function something like this

class_weights = class_weight.compute_class_weight(class_weight = , classes = , y =)

otherwise it will complain

```
from sklearn.utils import class_weight

# Calculate class weights
class_weights = 
class_weight.compute_class_weight(class_weight="balanced",
classes=np.unique(Y), y=Ytrain)

# Print the class weights
print(class_weights)

# Keras wants the weights in this form, uncomment and change value1 
and value2 to your weights,
# or get them from the array that is returned from class_weight

class_weights = {0: class_weights[0],
                1: class_weights[1]}
```

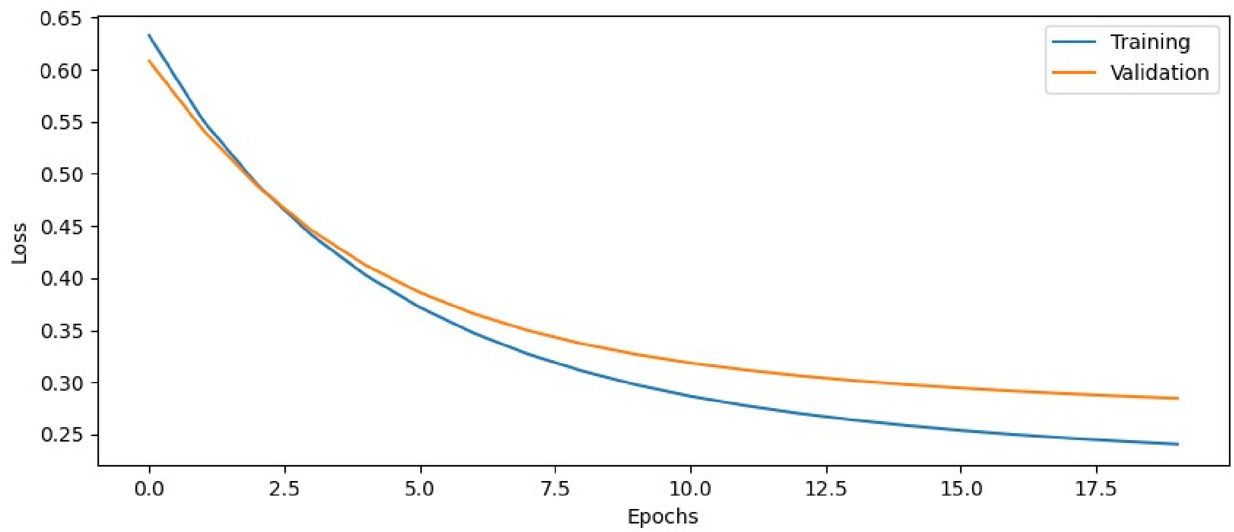
```
[3.14385212 0.594559  ]
```

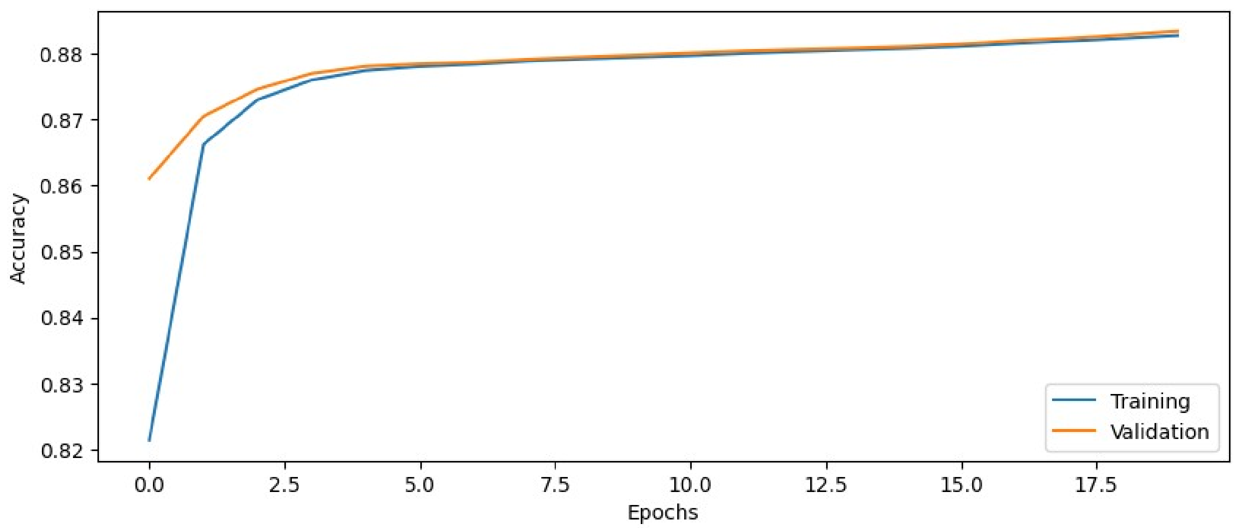
2 layers, 20 nodes, class weights

```
# Setup some training parameters
batch_size = 10000
epochs = 20
input_shape = (X.shape[1],)

# Build and train model
model2 = build_DNN(input_shape=input_shape, n_layers=2, n_nodes=20)

history2 = model2.fit(x=Xtrain, y=Ytrain, validation_data=(Xval,
Yval), batch_size=batch_size, epochs=epochs,
class_weight=class_weights)
```

```
Epoch 1/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 1s 9ms/step - accuracy: 0.7705 - loss:
0.6567 - val_accuracy: 0.8611 - val_loss: 0.6084
Epoch 2/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.8645 - loss:
0.5682 - val_accuracy: 0.8705 - val_loss: 0.5418
Epoch 3/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.8707 - loss:
0.5022 - val_accuracy: 0.8747 - val_loss: 0.4883
Epoch 4/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.8755 - loss:
0.4507 - val_accuracy: 0.8770 - val_loss: 0.4458
Epoch 5/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.8768 - loss:
0.4113 - val_accuracy: 0.8781 - val_loss: 0.4123
Epoch 6/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.8793 - loss:
0.3784 - val_accuracy: 0.8785 - val_loss: 0.3861
Epoch 7/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.8782 - loss:
0.3523 - val_accuracy: 0.8787 - val_loss: 0.3657
Epoch 8/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.8784 - loss:
0.3319 - val_accuracy: 0.8792 - val_loss: 0.3496
Epoch 9/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.8786 - loss:
0.3148 - val_accuracy: 0.8795 - val_loss: 0.3369
Epoch 10/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.8794 - loss:
0.3000 - val_accuracy: 0.8798 - val_loss: 0.3267
Epoch 11/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.8800 - loss:
0.2892 - val_accuracy: 0.8801 - val_loss: 0.3185
Epoch 12/20
```

```
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.8797 - loss:
0.2794 - val_accuracy: 0.8804 - val_loss: 0.3118
Epoch 13/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.8803 - loss:
0.2724 - val_accuracy: 0.8807 - val_loss: 0.3063
Epoch 14/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.8805 - loss:
0.2654 - val_accuracy: 0.8809 - val_loss: 0.3017
Epoch 15/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.8817 - loss:
0.2587 - val_accuracy: 0.8811 - val_loss: 0.2977
Epoch 16/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.8810 - loss:
0.2547 - val_accuracy: 0.8815 - val_loss: 0.2944
Epoch 17/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.8808 - loss:
0.2513 - val_accuracy: 0.8820 - val_loss: 0.2915
Epoch 18/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.8824 - loss:
0.2461 - val_accuracy: 0.8824 - val_loss: 0.2889
Epoch 19/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.8821 - loss:
0.2438 - val_accuracy: 0.8829 - val_loss: 0.2867
Epoch 20/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.8823 - loss:
0.2420 - val_accuracy: 0.8835 - val_loss: 0.2846

# Evaluate model on test data
score = model2.evaluate(x=Xtest, y=Ytest)

print('Test loss: %.4f' % score[0])
print('Test accuracy: %.4f' % score[1])

  84/3582 ━━━━━━━━━━━━━━━━━━━━ 4s 1ms/step - accuracy: 0.8946 - loss:
0.2640

3582/3582 ━━━━━━━━━━━━━━━━━━━━ 4s 1ms/step - accuracy: 0.8853 - loss:
0.2802
Test loss: 0.2832
Test accuracy: 0.8842

plot_results(history2)
```






```
model2.summary()

Model: "sequential_1"

┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━
━━━━━┓
┃ Layer (type)                         ┃ Output Shape            ┃
Param # ┃
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━
━━━━━┩
│ dense_2 (Dense)                      │ (None, 20)              │
1,860 │
├──────────────────────────────────────┼─────────────────────────┼───────────────
─────┤
│ dense_3 (Dense)                      │ (None, 1)               │
21 │
```

```
└──────────────────────────────────┴─────────────────────────┴──────────
──────┘

 Total params: 1,883 (7.36 KB)

 Trainable params: 1,881 (7.35 KB)

 Non-trainable params: 0 (0.00 B)

 Optimizer params: 2 (12.00 B)
```

Part 13: More questions

Skip questions 8 and 9 if you run on the CPU (recommended)

Question 7: Why do we have to use a batch size? Why can't we simply use all data at once? This is more relevant for even larger datasets.

When using really large datasets, the data can often not be fit fully into GPU (or even CPU) memory. Even if it does, it might be too slow to train on the entire dataset in each epoch. We must then use batches of data to train the model instead. The batch size may depend on how much data we can fit in memory.

Question 8: How busy is the GPU for a batch size of 100? How much GPU memory is used? Hint: run 'nvidia-smi' on the computer a few times during training.

Question 9: What is the processing time for one training epoch when the batch size is 100? What is the processing time for one epoch when the batch size is 1,000? What is the processing time for one epoch when the batch size is 10,000? Explain the results.

Question 10: How many times are the weights in the DNN updated in each training epoch if the batch size is 100? How many times are the weights in the DNN updated in each training epoch if the batch size is 1,000? How many times are the weights in the DNN updated in each training epoch if the batch size is 10,000?

If the training data has 534895 rows. The weights are updated 534895 / 100 = 5349 times if the batch size is 100. If the batch size is 1000, the weights are updated 534895 / 1000 = 535 times. If the batch size is 10,000, the weights are updated 534895 / 10,000 = 54 times.

Question 11: What limits how large the batch size can be?

The memory capacity of the GPU or CPU, and the size of the data.

Question 12: Generally speaking, how is the learning rate related to the batch size? If the batch size is decreased, how should the learning rate be changed?

If the batch size is lower, there is more variation in each pass of the data, so we will want to use a smaller learning rate.

Lets use a batch size of 10,000 from now on, and a learning rate of 0.1.

Part 14: Increasing the complexity

Lets try some different configurations of number of layers and number of nodes per layer.

Question 13: How many trainable parameters does the network with 4 dense layers with 50 nodes each have, compared to the initial network with 2 layers and 20 nodes per layer? Hint: use model.summary()

The initial network has 1,881 trainable paramaters, whereas the model with 4 layers and 50 nodes has 9,801 trainable parameters.

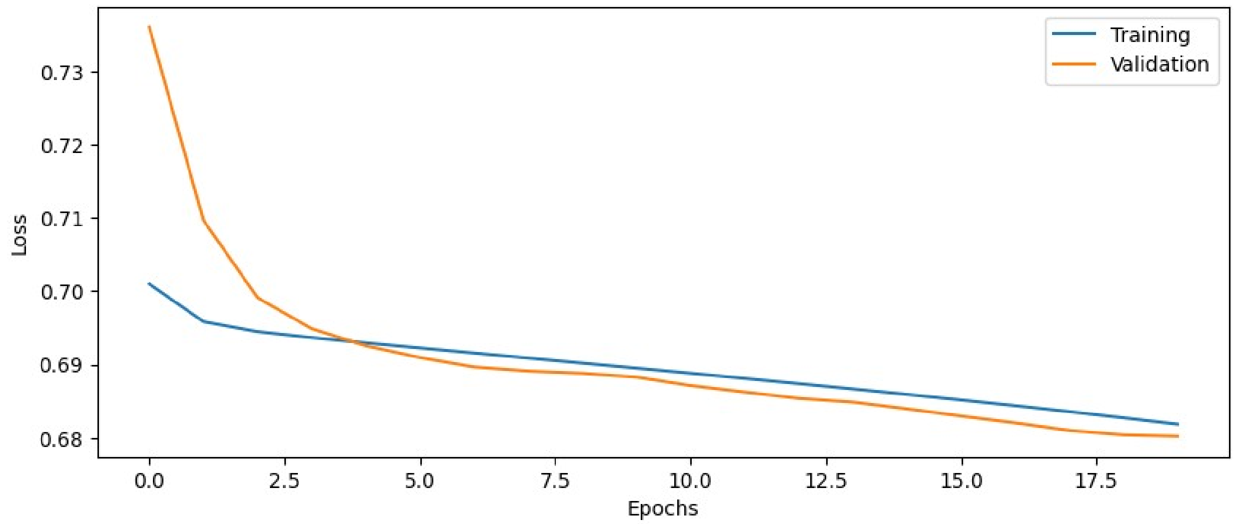
4 layers, 20 nodes, class weights

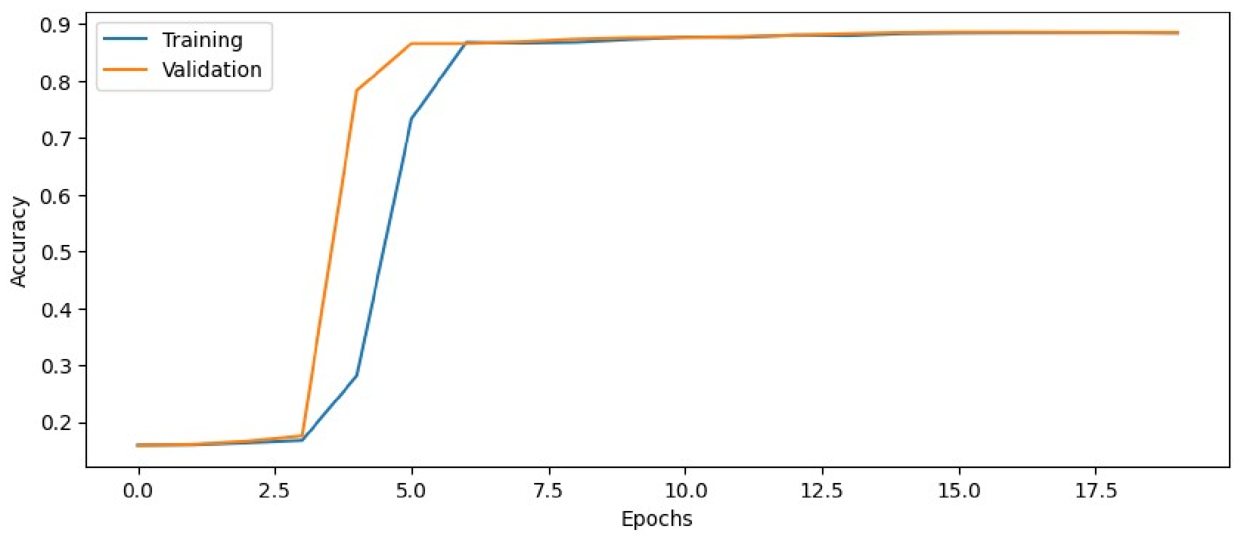
```
# Setup some training parameters
batch_size = 10000
epochs = 20
input_shape = (X.shape[1],)

# Build and train model
model3 = build_DNN(input_shape=input_shape, n_layers=4, n_nodes=20)

history3 = model3.fit(x=Xtrain, y=Ytrain, validation_data=(Xval,
Yval), batch_size=batch_size, epochs=epochs,
class_weight=class_weights)
```

```
Epoch 1/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 2s 12ms/step - accuracy: 0.1596 - loss:
0.7043 - val_accuracy: 0.1583 - val_loss: 0.7360
Epoch 2/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.1589 - loss:
0.6959 - val_accuracy: 0.1604 - val_loss: 0.7097
Epoch 3/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.1618 - loss:
0.6947 - val_accuracy: 0.1659 - val_loss: 0.6991
Epoch 4/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.1658 - loss:
0.6942 - val_accuracy: 0.1750 - val_loss: 0.6949
Epoch 5/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.1990 - loss:
0.6934 - val_accuracy: 0.7834 - val_loss: 0.6926
Epoch 6/20
```

```
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.6382 - loss:
0.6936 - val_accuracy: 0.8661 - val_loss: 0.6910
Epoch 7/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.8687 - loss:
0.6940 - val_accuracy: 0.8661 - val_loss: 0.6897
Epoch 8/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.8665 - loss:
0.6916 - val_accuracy: 0.8695 - val_loss: 0.6891
Epoch 9/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.8680 - loss:
0.6887 - val_accuracy: 0.8741 - val_loss: 0.6888
Epoch 10/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.8735 - loss:
0.6899 - val_accuracy: 0.8765 - val_loss: 0.6883
Epoch 11/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.8778 - loss:
0.6902 - val_accuracy: 0.8771 - val_loss: 0.6871
Epoch 12/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.8761 - loss:
0.6888 - val_accuracy: 0.8784 - val_loss: 0.6862
Epoch 13/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.8807 - loss:
0.6887 - val_accuracy: 0.8809 - val_loss: 0.6854
Epoch 14/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.8800 - loss:
0.6860 - val_accuracy: 0.8835 - val_loss: 0.6849
Epoch 15/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 1s 8ms/step - accuracy: 0.8835 - loss:
0.6860 - val_accuracy: 0.8856 - val_loss: 0.6840
Epoch 16/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.8846 - loss:
0.6857 - val_accuracy: 0.8863 - val_loss: 0.6830
Epoch 17/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.8851 - loss:
0.6851 - val_accuracy: 0.8863 - val_loss: 0.6820
Epoch 18/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.8850 - loss:
0.6848 - val_accuracy: 0.8861 - val_loss: 0.6810
Epoch 19/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.8854 - loss:
0.6829 - val_accuracy: 0.8860 - val_loss: 0.6804
Epoch 20/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.8845 - loss:
0.6797 - val_accuracy: 0.8857 - val_loss: 0.6802

# Evaluate model on test data
score = model3.evaluate(x=Xtest, y=Ytest)

print('Test loss: %.4f' % score[0])
print('Test accuracy: %.4f' % score[1])
```

```
  42/3582 ━━━━━━━━━━━━━━━━━━━━ 4s 1ms/step - accuracy: 0.8984 - loss: 
0.6802

3582/3582 ━━━━━━━━━━━━━━━━━━━━ 4s 1ms/step - accuracy: 0.8897 - loss: 
0.6802
Test loss: 0.6802
Test accuracy: 0.8885

plot_results(history3)
```





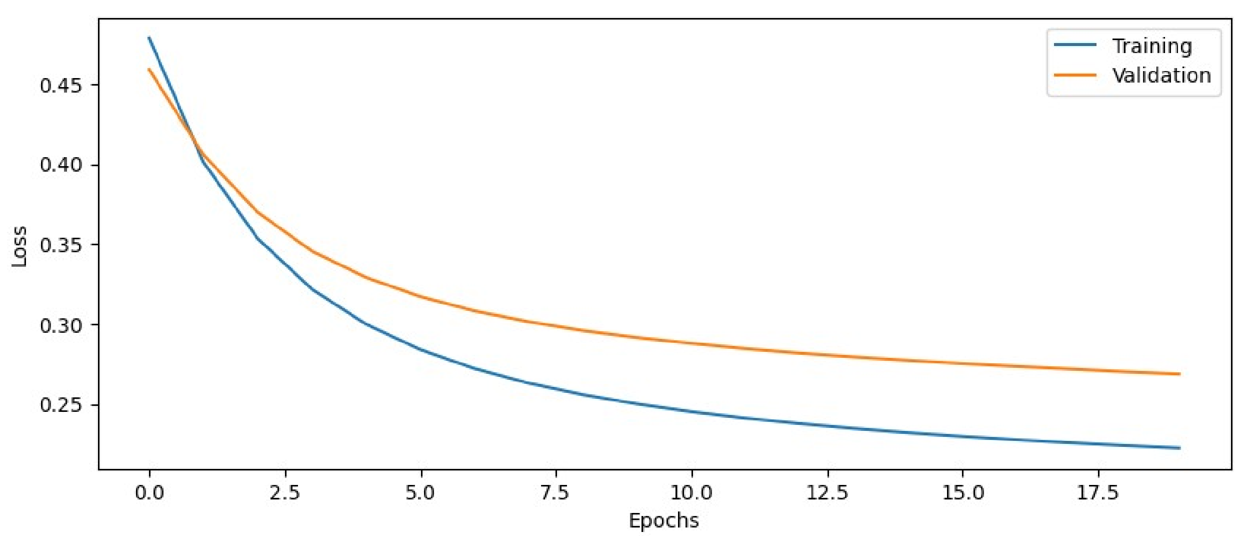
2 layers, 50 nodes, class weights

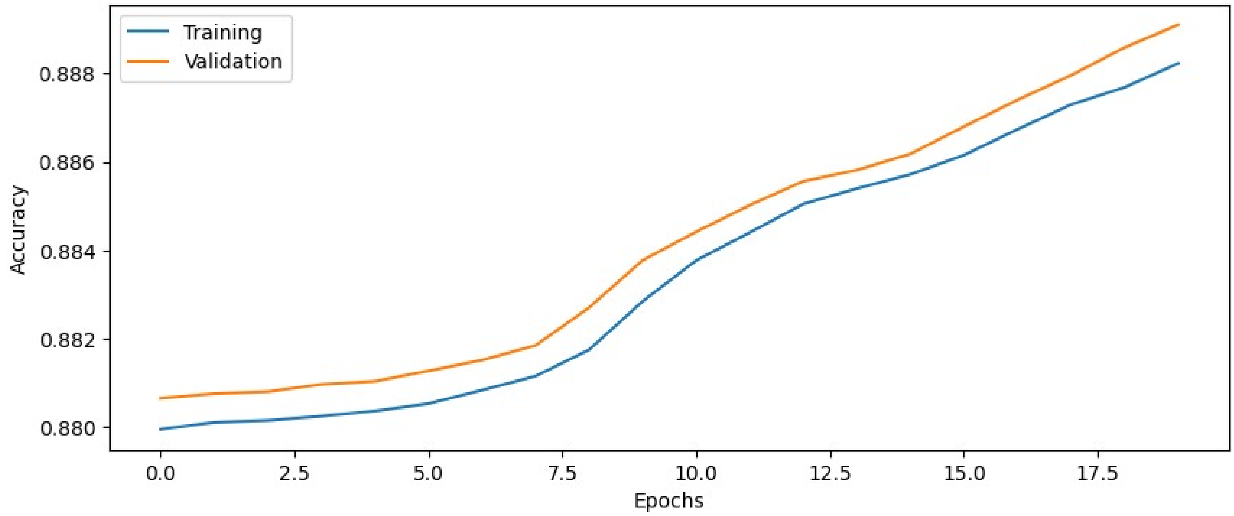
```
# Setup some training parameters
batch_size = 10000
epochs = 20
input_shape = (X.shape[1],)
```

```python
# Build and train model
model4 = build_DNN(input_shape=input_shape, n_layers=2, n_nodes=50)

history4 = model4.fit(x=Xtrain, y=Ytrain, validation_data=(Xval,
Yval), batch_size=batch_size, epochs=epochs,
class_weight=class_weights)
```

```
Epoch 1/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 1s 11ms/step - accuracy: 0.8794 - loss:
0.5035 - val_accuracy: 0.8807 - val_loss: 0.4589
Epoch 2/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.8797 - loss:
0.4160 - val_accuracy: 0.8808 - val_loss: 0.4057
Epoch 3/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.8798 - loss:
0.3622 - val_accuracy: 0.8808 - val_loss: 0.3698
Epoch 4/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.8806 - loss:
0.3283 - val_accuracy: 0.8810 - val_loss: 0.3455
Epoch 5/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.8799 - loss:
0.3042 - val_accuracy: 0.8810 - val_loss: 0.3290
Epoch 6/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.8806 - loss:
0.2867 - val_accuracy: 0.8813 - val_loss: 0.3171
Epoch 7/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.8809 - loss:
0.2745 - val_accuracy: 0.8815 - val_loss: 0.3081
Epoch 8/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.8810 - loss:
0.2657 - val_accuracy: 0.8818 - val_loss: 0.3013
Epoch 9/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.8814 - loss:
0.2576 - val_accuracy: 0.8827 - val_loss: 0.2958
Epoch 10/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.8819 - loss:
0.2524 - val_accuracy: 0.8838 - val_loss: 0.2915
Epoch 11/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.8838 - loss:
0.2458 - val_accuracy: 0.8844 - val_loss: 0.2878
Epoch 12/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.8845 - loss:
0.2420 - val_accuracy: 0.8850 - val_loss: 0.2845
Epoch 13/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.8846 - loss:
0.2388 - val_accuracy: 0.8856 - val_loss: 0.2817
Epoch 14/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.8853 - loss:
0.2358 - val_accuracy: 0.8858 - val_loss: 0.2794
```

```
Epoch 15/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.8860 - loss:
0.2325 - val_accuracy: 0.8862 - val_loss: 0.2772
Epoch 16/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.8864 - loss:
0.2296 - val_accuracy: 0.8868 - val_loss: 0.2752
Epoch 17/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.8865 - loss:
0.2278 - val_accuracy: 0.8874 - val_loss: 0.2735
Epoch 18/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.8866 - loss:
0.2264 - val_accuracy: 0.8880 - val_loss: 0.2718
Epoch 19/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.8875 - loss:
0.2252 - val_accuracy: 0.8886 - val_loss: 0.2701
Epoch 20/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.8881 - loss:
0.2227 - val_accuracy: 0.8891 - val_loss: 0.2687

# Evaluate model on test data
score = model4.evaluate(x=Xtest, y=Ytest)

print('Test loss: %.4f' % score[0])
print('Test accuracy: %.4f' % score[1])

  38/3582 ━━━━━━━━━━━━━━━━━━━━ 4s 1ms/step - accuracy: 0.8971 - loss:
0.2600

3582/3582 ━━━━━━━━━━━━━━━━━━━━ 4s 1ms/step - accuracy: 0.8909 - loss:
0.2637
Test loss: 0.2666
Test accuracy: 0.8900

plot_results(history4)
```





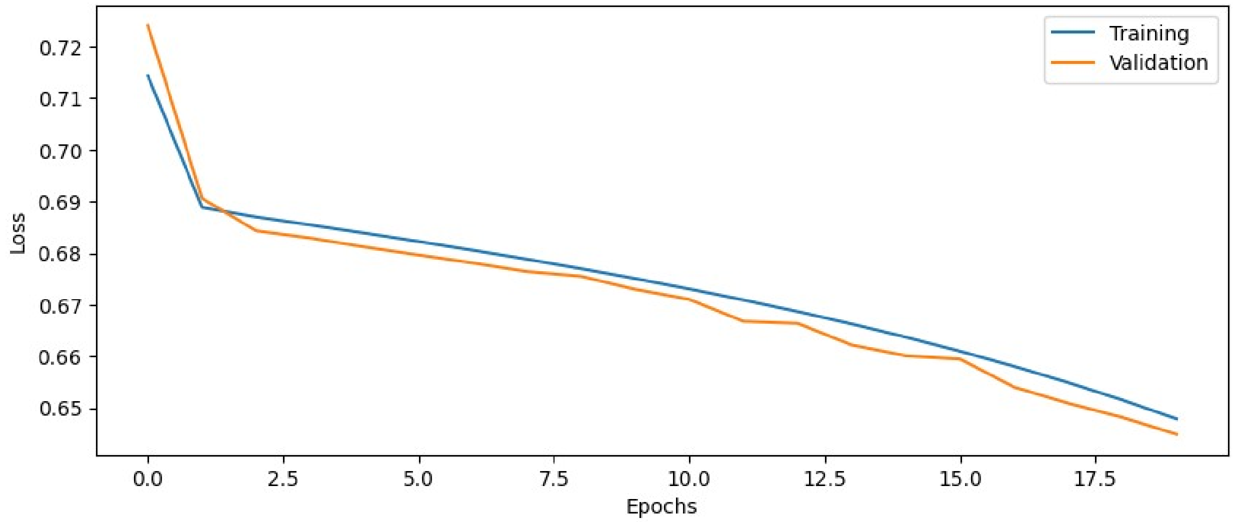
4 layers, 50 nodes, class weights

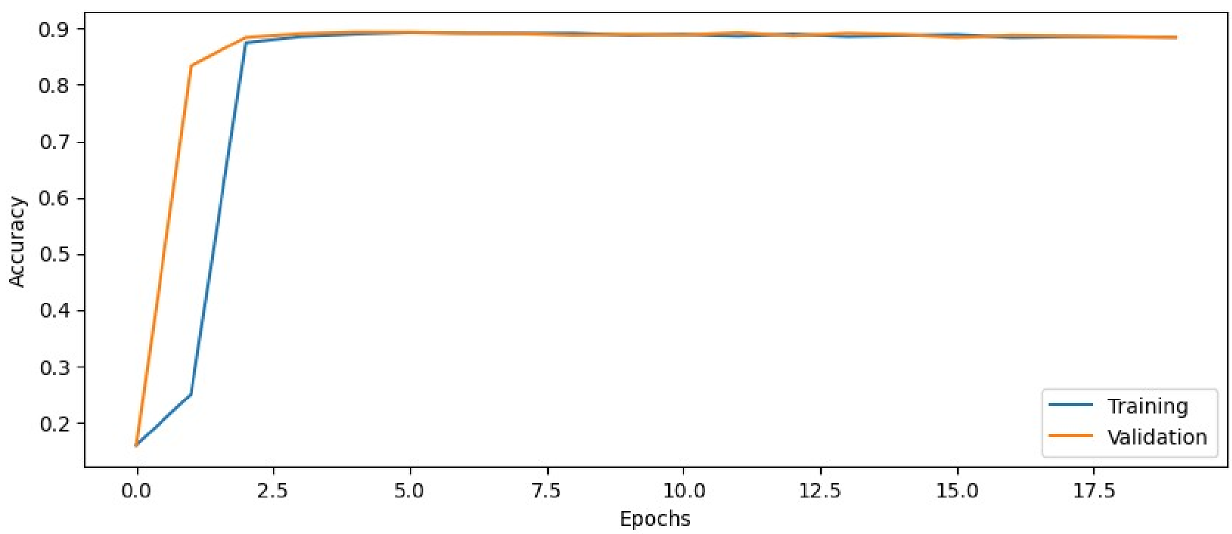
```
# Setup some training parameters
batch_size = 10000
epochs = 20
input_shape = (X.shape[1],)

# Build and train model
model5 = build_DNN(input_shape=input_shape, n_layers=4, n_nodes=50)

history5 = model5.fit(x=Xtrain, y=Ytrain, validation_data=(Xval,
Yval), batch_size=batch_size, epochs=epochs,
class_weight=class_weights)

Epoch 1/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 2s 16ms/step - accuracy: 0.1594 - loss:
0.7392 - val_accuracy: 0.1583 - val_loss: 0.7241
Epoch 2/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.1713 - loss:
0.6879 - val_accuracy: 0.8336 - val_loss: 0.6906
Epoch 3/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.8610 - loss:
0.6885 - val_accuracy: 0.8845 - val_loss: 0.6843
Epoch 4/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.8834 - loss:
0.6850 - val_accuracy: 0.8909 - val_loss: 0.6829
Epoch 5/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 1s 11ms/step - accuracy: 0.8910 - loss:
0.6842 - val_accuracy: 0.8939 - val_loss: 0.6812
Epoch 6/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.8935 - loss:
0.6832 - val_accuracy: 0.8938 - val_loss: 0.6796
Epoch 7/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.8930 - loss:
```

```
0.6800 - val_accuracy: 0.8920 - val_loss: 0.6781
Epoch 8/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.8914 - loss:
0.6792 - val_accuracy: 0.8912 - val_loss: 0.6764
Epoch 9/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.8918 - loss:
0.6767 - val_accuracy: 0.8883 - val_loss: 0.6755
Epoch 10/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.8876 - loss:
0.6758 - val_accuracy: 0.8897 - val_loss: 0.6730
Epoch 11/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 1s 11ms/step - accuracy: 0.8897 - loss:
0.6732 - val_accuracy: 0.8885 - val_loss: 0.6710
Epoch 12/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 1s 11ms/step - accuracy: 0.8856 - loss:
0.6731 - val_accuracy: 0.8929 - val_loss: 0.6668
Epoch 13/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.8906 - loss:
0.6695 - val_accuracy: 0.8872 - val_loss: 0.6664
Epoch 14/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.8852 - loss:
0.6681 - val_accuracy: 0.8918 - val_loss: 0.6622
Epoch 15/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 1s 11ms/step - accuracy: 0.8886 - loss:
0.6653 - val_accuracy: 0.8897 - val_loss: 0.6601
Epoch 16/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.8900 - loss:
0.6602 - val_accuracy: 0.8843 - val_loss: 0.6595
Epoch 17/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.8846 - loss:
0.6590 - val_accuracy: 0.8883 - val_loss: 0.6540
Epoch 18/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 1s 11ms/step - accuracy: 0.8860 - loss:
0.6565 - val_accuracy: 0.8871 - val_loss: 0.6509
Epoch 19/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 1s 9ms/step - accuracy: 0.8858 - loss:
0.6528 - val_accuracy: 0.8854 - val_loss: 0.6481
Epoch 20/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.8847 - loss:
0.6488 - val_accuracy: 0.8847 - val_loss: 0.6449

# Evaluate model on test data
score = model5.evaluate(x=Xtest, y=Ytest)

print('Test loss: %.4f' % score[0])
print('Test accuracy: %.4f' % score[1])

   1/3582 ━━━━━━━━━━━━━━━━━━━━ 3:11 54ms/step - accuracy: 0.8750 -
loss: 0.6547
```

```
3582/3582 ━━━━━━━━━━━━━━━━━━━━ 4s 1ms/step - accuracy: 0.8866 - loss:
0.6447
Test loss: 0.6448
Test accuracy: 0.8855
```

```
plot_results(history5)
```






```
model5.summary()
```

```
Model: "sequential_4"

┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━
━━━━━┓
┃ Layer (type)                         ┃ Output Shape                ┃
Param # ┃
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━
```

```
──────┤
│ dense_10 (Dense)                  │ (None, 50)                │
4,650 │
├──────────────────────────────────┼─────────────────────────┼────────
──────┤
│ dense_11 (Dense)                  │ (None, 50)                │
2,550 │
├──────────────────────────────────┼─────────────────────────┼────────
──────┤
│ dense_12 (Dense)                  │ (None, 50)                │
2,550 │
├──────────────────────────────────┼─────────────────────────┼────────
──────┤
│ dense_13 (Dense)                  │ (None, 1)                 │
51 │
└──────────────────────────────────┴─────────────────────────┴────────
──────┘

 Total params: 9,803 (38.30 KB)

 Trainable params: 9,801 (38.29 KB)

 Non-trainable params: 0 (0.00 B)

 Optimizer params: 2 (12.00 B)
```

Part 15: Batch normalization

Now add batch normalization after each dense layer in `build_DNN`. Remember to import BatchNormalization from keras.layers.

See https://keras.io/layers/normalization/ for information about how to call the function.

Question 14: Why is batch normalization important when training deep networks?

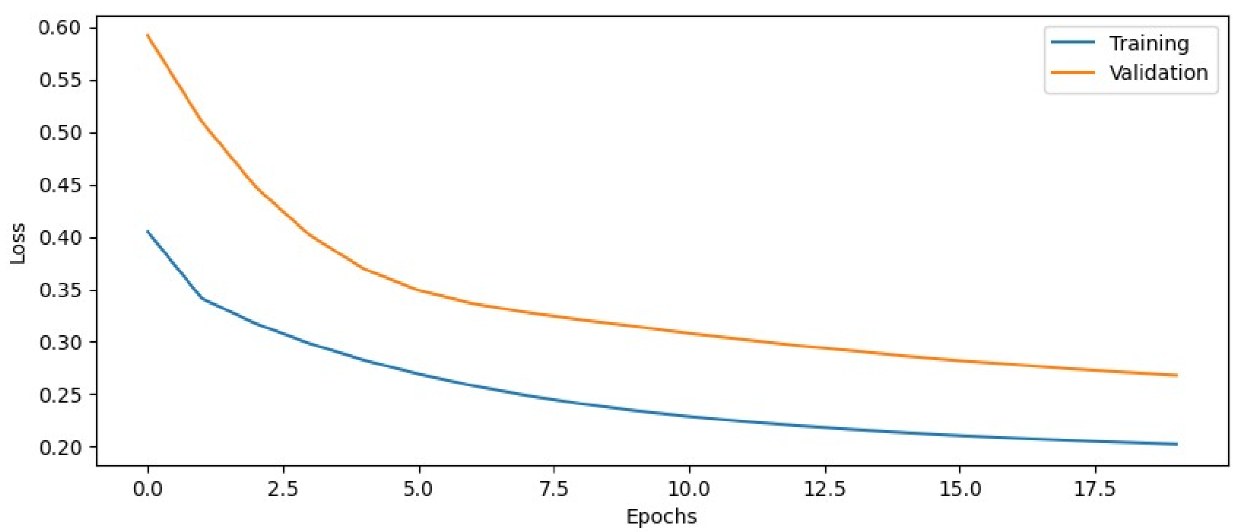
Why it works is not really known, but adding batch normalization makes the model converge faster.

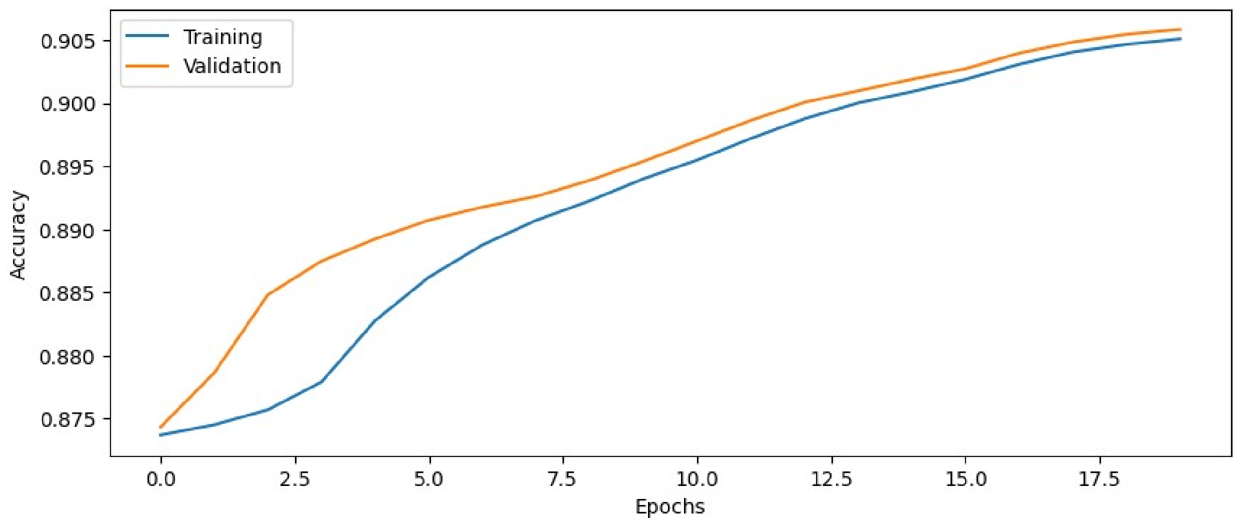
2 layers, 20 nodes, class weights, batch normalization

```
# Setup some training parameters
batch_size = 10000
epochs = 20
input_shape = (X.shape[1],)

# Build and train model
model6 = build_DNN(input_shape=input_shape, n_layers=2, n_nodes=20,
use_bn=True)
```

```
history6 = model6.fit(x=Xtrain, y=Ytrain, validation_data=(Xval,
Yval), batch_size=batch_size, epochs=epochs,
class_weight=class_weights)

Epoch 1/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 2s 12ms/step - accuracy: 0.8710 - loss:
0.4578 - val_accuracy: 0.8743 - val_loss: 0.5916
Epoch 2/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.8747 - loss:
0.3470 - val_accuracy: 0.8786 - val_loss: 0.5095
Epoch 3/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.8746 - loss:
0.3220 - val_accuracy: 0.8848 - val_loss: 0.4472
Epoch 4/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.8777 - loss:
0.3017 - val_accuracy: 0.8875 - val_loss: 0.4012
Epoch 5/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.8816 - loss:
0.2860 - val_accuracy: 0.8892 - val_loss: 0.3693
Epoch 6/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.8853 - loss:
0.2717 - val_accuracy: 0.8907 - val_loss: 0.3490
Epoch 7/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.8880 - loss:
0.2609 - val_accuracy: 0.8918 - val_loss: 0.3364
Epoch 8/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.8907 - loss:
0.2505 - val_accuracy: 0.8926 - val_loss: 0.3279
Epoch 9/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.8921 - loss:
0.2425 - val_accuracy: 0.8939 - val_loss: 0.3209
Epoch 10/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.8935 - loss:
0.2361 - val_accuracy: 0.8954 - val_loss: 0.3147
Epoch 11/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.8954 - loss:
0.2295 - val_accuracy: 0.8970 - val_loss: 0.3079
Epoch 12/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.8967 - loss:
0.2255 - val_accuracy: 0.8986 - val_loss: 0.3022
Epoch 13/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.8985 - loss:
0.2207 - val_accuracy: 0.9001 - val_loss: 0.2964
Epoch 14/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.8993 - loss:
0.2179 - val_accuracy: 0.9010 - val_loss: 0.2915
Epoch 15/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9002 - loss:
0.2149 - val_accuracy: 0.9019 - val_loss: 0.2864
Epoch 16/20
```

```
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9022 - loss: 
0.2102 - val_accuracy: 0.9027 - val_loss: 0.2820
Epoch 17/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9028 - loss: 
0.2088 - val_accuracy: 0.9040 - val_loss: 0.2783
Epoch 18/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9038 - loss: 
0.2069 - val_accuracy: 0.9048 - val_loss: 0.2744
Epoch 19/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9042 - loss: 
0.2047 - val_accuracy: 0.9054 - val_loss: 0.2711
Epoch 20/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9054 - loss: 
0.2020 - val_accuracy: 0.9058 - val_loss: 0.2681

# Evaluate model on test data
score = model6.evaluate(x=Xtest, y=Ytest)

print('Test loss: %.4f' % score[0])
print('Test accuracy: %.4f' % score[1])

  40/3582 ━━━━━━━━━━━━━━━━━━━━ 4s 1ms/step - accuracy: 0.9141 - loss: 
0.2668

3582/3582 ━━━━━━━━━━━━━━━━━━━━ 4s 1ms/step - accuracy: 0.9084 - loss: 
0.2632
Test loss: 0.2661
Test accuracy: 0.9072

plot_results(history6)
```





Part 16: Activation function

Try changing the activation function in each layer from sigmoid to ReLU, write down the test accuracy.

Note: the last layer should still have a sigmoid activation function.

https://keras.io/api/layers/activations/

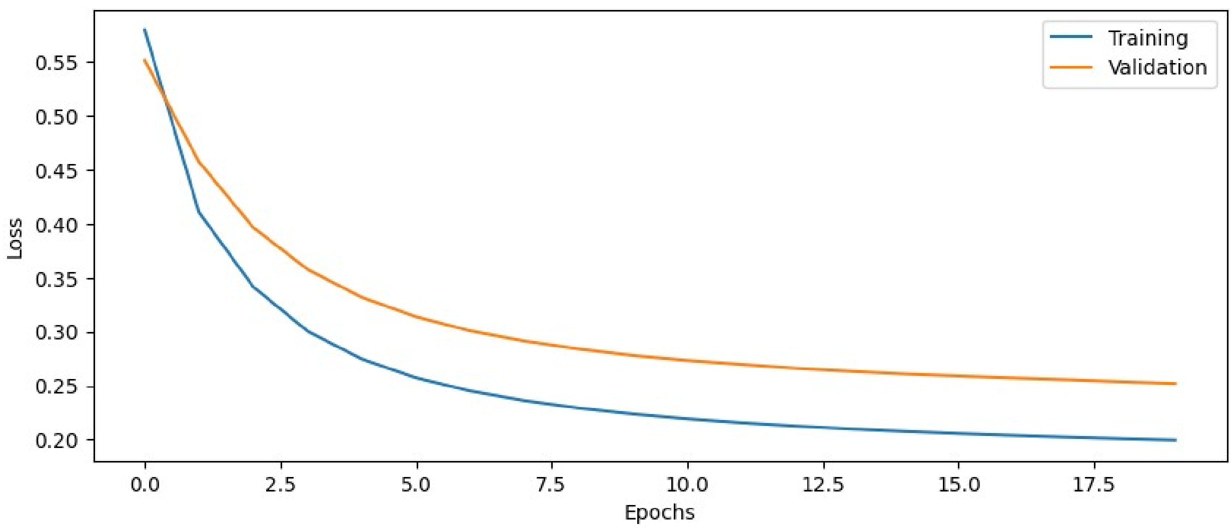
2 layers, 20 nodes, class weights, ReLU, no batch normalization

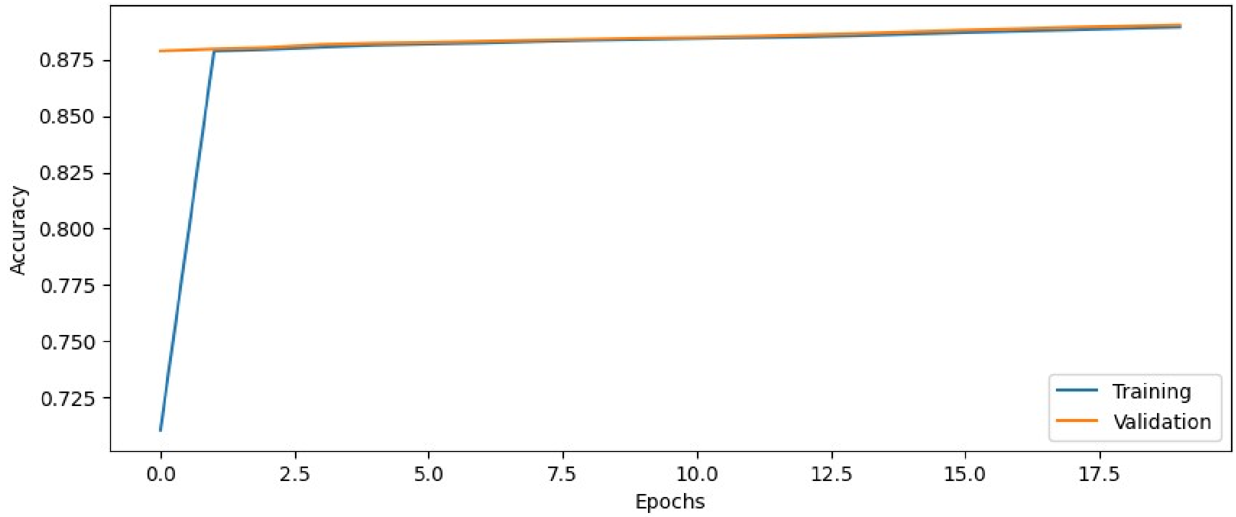
```
# Setup some training parameters
batch_size = 10000
epochs = 20
input_shape = (X.shape[1],)

# Build and train model
model7 = build_DNN(input_shape=input_shape, n_layers=2, n_nodes=20,
act_fun="relu")

history7 = model7.fit(x=Xtrain, y=Ytrain, validation_data=(Xval,
Yval), batch_size=batch_size, epochs=epochs,
class_weight=class_weights)
```

```
Epoch 1/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.5148 - loss:
0.6672 - val_accuracy: 0.8787 - val_loss: 0.5515
Epoch 2/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.8791 - loss:
0.4324 - val_accuracy: 0.8796 - val_loss: 0.4575
Epoch 3/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.8790 - loss:
0.3536 - val_accuracy: 0.8803 - val_loss: 0.3966
```

```
Epoch 4/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.8802 - loss:
0.3076 - val_accuracy: 0.8816 - val_loss: 0.3578
Epoch 5/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.8811 - loss:
0.2796 - val_accuracy: 0.8821 - val_loss: 0.3319
Epoch 6/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.8820 - loss:
0.2607 - val_accuracy: 0.8825 - val_loss: 0.3138
Epoch 7/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.8822 - loss:
0.2480 - val_accuracy: 0.8829 - val_loss: 0.3008
Epoch 8/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.8829 - loss:
0.2375 - val_accuracy: 0.8835 - val_loss: 0.2914
Epoch 9/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.8833 - loss:
0.2306 - val_accuracy: 0.8840 - val_loss: 0.2840
Epoch 10/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.8834 - loss:
0.2250 - val_accuracy: 0.8844 - val_loss: 0.2779
Epoch 11/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.8836 - loss:
0.2210 - val_accuracy: 0.8847 - val_loss: 0.2733
Epoch 12/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.8837 - loss:
0.2172 - val_accuracy: 0.8852 - val_loss: 0.2697
Epoch 13/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.8854 - loss:
0.2132 - val_accuracy: 0.8859 - val_loss: 0.2661
Epoch 14/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.8859 - loss:
0.2099 - val_accuracy: 0.8864 - val_loss: 0.2634
Epoch 15/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.8860 - loss:
0.2078 - val_accuracy: 0.8872 - val_loss: 0.2609
Epoch 16/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.8869 - loss:
0.2058 - val_accuracy: 0.8879 - val_loss: 0.2589
Epoch 17/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.8869 - loss:
0.2046 - val_accuracy: 0.8885 - val_loss: 0.2571
Epoch 18/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.8880 - loss:
0.2023 - val_accuracy: 0.8892 - val_loss: 0.2554
Epoch 19/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.8889 - loss:
0.2009 - val_accuracy: 0.8897 - val_loss: 0.2536
Epoch 20/20
```

```
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.8895 - loss:
0.1995 - val_accuracy: 0.8903 - val_loss: 0.2519

# Evaluate model on test data
score = model7.evaluate(x=Xtest, y=Ytest)

print('Test loss: %.4f' % score[0])
print('Test accuracy: %.4f' % score[1])

  43/3582 ━━━━━━━━━━━━━━━━━━━━ 4s 1ms/step - accuracy: 0.8969 - loss:
0.2440

3582/3582 ━━━━━━━━━━━━━━━━━━━━ 4s 1ms/step - accuracy: 0.8923 - loss:
0.2470
Test loss: 0.2492
Test accuracy: 0.8913

plot_results(history7)
```





Part 17: Optimizer

Try changing the optimizer from SGD to Adam (with learning rate 0.1 as before). Remember to import the Adam optimizer from keras.optimizers.

https://keras.io/optimizers/

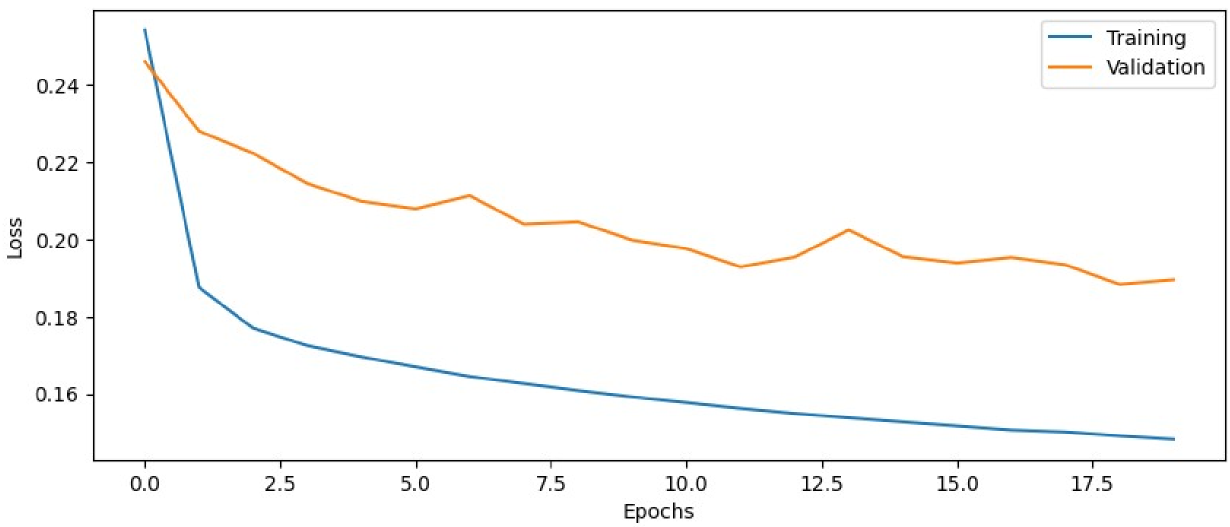
2 layers, 20 nodes, class weights, Adam optimizer, no batch normalization, sigmoid activations

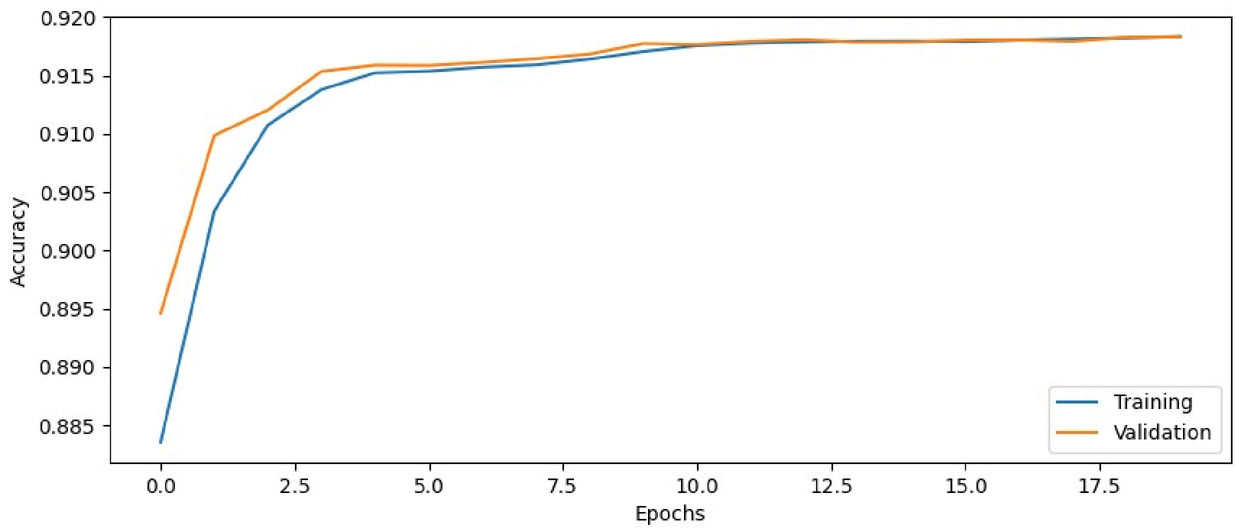
```
# Setup some training parameters
batch_size = 10000
epochs = 20
input_shape = (X.shape[1],)

# Build and train model
model8 = build_DNN(input_shape=input_shape, n_layers=2, n_nodes=20,
optimizer="adam")

history8 = model8.fit(x=Xtrain, y=Ytrain, validation_data=(Xval,
Yval), batch_size=batch_size, epochs=epochs,
class_weight=class_weights)
```

```
Epoch 1/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 2s 9ms/step - accuracy: 0.8778 - loss:
0.3310 - val_accuracy: 0.8946 - val_loss: 0.2460
Epoch 2/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.8995 - loss:
0.1923 - val_accuracy: 0.9099 - val_loss: 0.2280
Epoch 3/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.9101 - loss:
0.1785 - val_accuracy: 0.9120 - val_loss: 0.2223
```

```
Epoch 4/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.9128 - loss:
0.1734 - val_accuracy: 0.9154 - val_loss: 0.2145
Epoch 5/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.9152 - loss:
0.1705 - val_accuracy: 0.9159 - val_loss: 0.2098
Epoch 6/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.9157 - loss:
0.1671 - val_accuracy: 0.9159 - val_loss: 0.2079
Epoch 7/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.9153 - loss:
0.1657 - val_accuracy: 0.9161 - val_loss: 0.2113
Epoch 8/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.9156 - loss:
0.1636 - val_accuracy: 0.9164 - val_loss: 0.2039
Epoch 9/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.9167 - loss:
0.1613 - val_accuracy: 0.9168 - val_loss: 0.2046
Epoch 10/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.9165 - loss:
0.1601 - val_accuracy: 0.9177 - val_loss: 0.1998
Epoch 11/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.9176 - loss:
0.1582 - val_accuracy: 0.9177 - val_loss: 0.1976
Epoch 12/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.9176 - loss:
0.1566 - val_accuracy: 0.9179 - val_loss: 0.1929
Epoch 13/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.9184 - loss:
0.1543 - val_accuracy: 0.9181 - val_loss: 0.1954
Epoch 14/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.9177 - loss:
0.1540 - val_accuracy: 0.9179 - val_loss: 0.2024
Epoch 15/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.9182 - loss:
0.1526 - val_accuracy: 0.9179 - val_loss: 0.1955
Epoch 16/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.9175 - loss:
0.1523 - val_accuracy: 0.9181 - val_loss: 0.1939
Epoch 17/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.9179 - loss:
0.1508 - val_accuracy: 0.9181 - val_loss: 0.1953
Epoch 18/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.9184 - loss:
0.1500 - val_accuracy: 0.9179 - val_loss: 0.1934
Epoch 19/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.9182 - loss:
0.1494 - val_accuracy: 0.9183 - val_loss: 0.1884
Epoch 20/20
```

```
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.9180 - loss:
0.1490 - val_accuracy: 0.9183 - val_loss: 0.1896

# Evaluate model on test data
score = model8.evaluate(x=Xtest, y=Ytest)

print('Test loss: %.4f' % score[0])
print('Test accuracy: %.4f' % score[1])

  88/3582 ━━━━━━━━━━━━━━━━━━━━ 4s 1ms/step - accuracy: 0.9343 - loss:
0.1594

3582/3582 ━━━━━━━━━━━━━━━━━━━━ 4s 1ms/step - accuracy: 0.9215 - loss:
0.1834
Test loss: 0.1862
Test accuracy: 0.9201

plot_results(history8)
```





Part 18: Dropout regularization

Dropout is a type of regularization that can improve accuracy for validation and test data. It randomly removes connections to force the neural network to not rely too much on a small number of weights.

Add a Dropout layer after each Dense layer (but not after the final dense layer) in `build_DNN`, with a dropout probability of 50%. Remember to first import the Dropout layer from keras.layers

See https://keras.io/api/layers/regularization_layers/dropout/ for how the Dropout layer works.

Question 15: How does the validation accuracy change when adding dropout?

The validation accuracy is now slightly higher than the training accuracy, whereas before it was either the same or lower, though the differences are small.

Question 16: How does the test accuracy change when adding dropout?

The test accuracy stays roughly the same as before.

2 layers, 20 nodes, class weights, dropout, SGD optimizer, no batch normalization, sigmoid activations

```
# Setup some training parameters
batch_size = 10000
epochs = 20
input_shape = (X.shape[1],)

# Build and train model
```

```
model9 = build_DNN(input_shape=input_shape, n_layers=2, n_nodes=20,
use_dropout=True)

history9 = model9.fit(x=Xtrain, y=Ytrain, validation_data=(Xval,
Yval), batch_size=batch_size, epochs=epochs,
class_weight=class_weights)
```

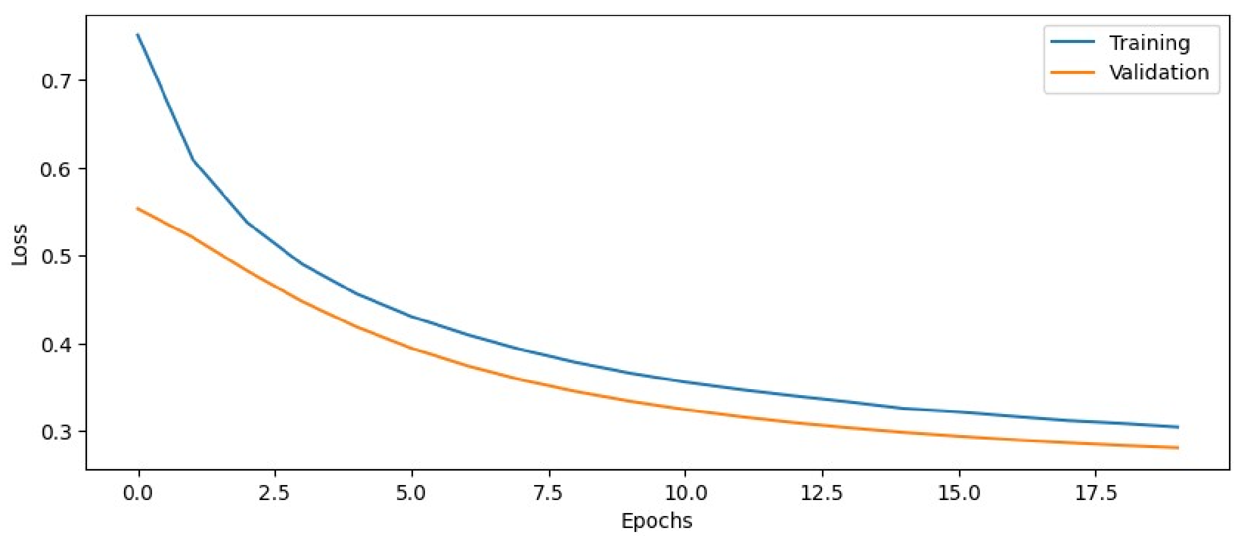
```
Epoch 1/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 1s 11ms/step - accuracy: 0.6841 - loss:
0.8053 - val_accuracy: 0.8743 - val_loss: 0.5533
Epoch 2/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.6851 - loss:
0.6316 - val_accuracy: 0.8910 - val_loss: 0.5207
Epoch 3/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.7212 - loss:
0.5513 - val_accuracy: 0.8876 - val_loss: 0.4824
Epoch 4/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.7561 - loss:
0.4991 - val_accuracy: 0.8827 - val_loss: 0.4476
Epoch 5/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.7837 - loss:
0.4636 - val_accuracy: 0.8819 - val_loss: 0.4185
Epoch 6/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.8017 - loss:
0.4363 - val_accuracy: 0.8818 - val_loss: 0.3942
Epoch 7/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.8147 - loss:
0.4135 - val_accuracy: 0.8819 - val_loss: 0.3745
Epoch 8/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.8240 - loss:
0.3958 - val_accuracy: 0.8817 - val_loss: 0.3583
Epoch 9/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.8316 - loss:
0.3815 - val_accuracy: 0.8816 - val_loss: 0.3450
Epoch 10/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.8382 - loss:
0.3686 - val_accuracy: 0.8815 - val_loss: 0.3338
Epoch 11/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.8425 - loss:
0.3574 - val_accuracy: 0.8814 - val_loss: 0.3243
Epoch 12/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.8462 - loss:
0.3491 - val_accuracy: 0.8813 - val_loss: 0.3162
Epoch 13/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.8501 - loss:
0.3412 - val_accuracy: 0.8813 - val_loss: 0.3093
Epoch 14/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.8524 - loss:
0.3337 - val_accuracy: 0.8813 - val_loss: 0.3035
Epoch 15/20
```

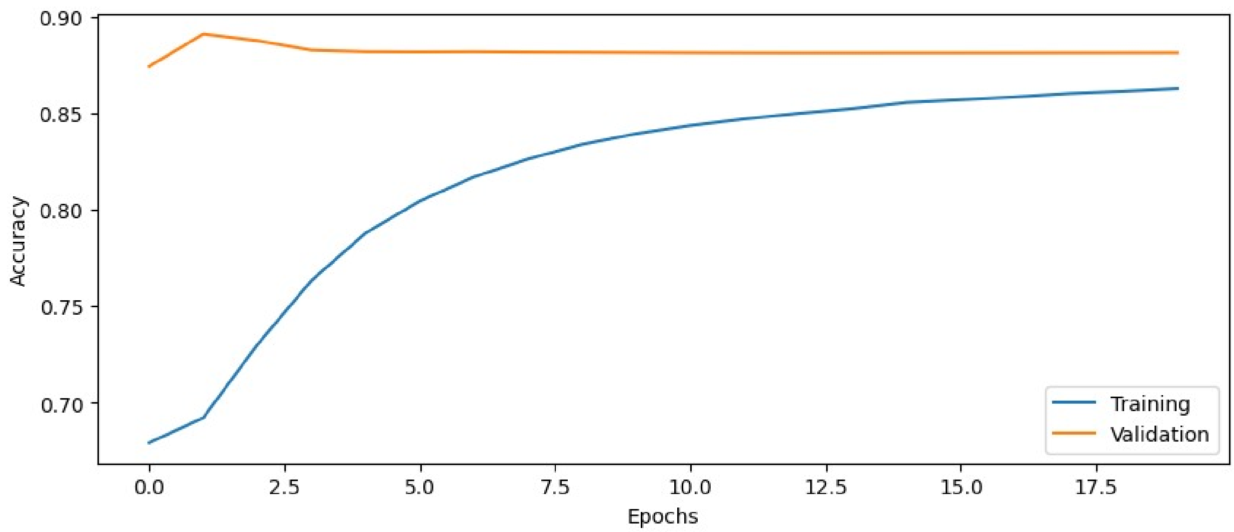
```
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.8550 - loss:
0.3268 - val_accuracy: 0.8813 - val_loss: 0.2983
Epoch 16/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.8566 - loss:
0.3229 - val_accuracy: 0.8813 - val_loss: 0.2939
Epoch 17/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.8578 - loss:
0.3175 - val_accuracy: 0.8813 - val_loss: 0.2899
Epoch 18/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.8602 - loss:
0.3117 - val_accuracy: 0.8813 - val_loss: 0.2864
Epoch 19/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.8613 - loss:
0.3091 - val_accuracy: 0.8813 - val_loss: 0.2833
Epoch 20/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.8622 - loss:
0.3050 - val_accuracy: 0.8814 - val_loss: 0.2807
```

```python
# Evaluate model on test data
score = model9.evaluate(x=Xtest, y=Ytest)

print('Test loss: %.4f' % score[0])
print('Test accuracy: %.4f' % score[1])
```

```
   1/3582 ━━━━━━━━━━━━━━━━━━━━ 2:19 39ms/step - accuracy: 0.8750 -
loss: 0.3408

3582/3582 ━━━━━━━━━━━━━━━━━━━━ 4s 1ms/step - accuracy: 0.8835 - loss:
0.2772
Test loss: 0.2795
Test accuracy: 0.8822
```

```python
plot_results(history9)
```





Part 19: Improving performance

Spend some time (30 - 90 minutes) playing with the network architecture (number of layers, number of nodes per layer, activation function) and other hyper parameters (optimizer, learning rate, batch size, number of epochs, degree of regularization). For example, try a much deeper network. How much does the training time increase for a network with 10 layers?

Question 17: How high classification accuracy can you achieve for the test data? What is your best configuration?

```
# Find your best configuration for the DNN
# Build and train DNN
model10 = build_DNN(input_shape=input_shape, n_layers=4, n_nodes=50,
optimizer="adam", act_fun="relu", use_bn=True, use_dropout=True)

history10 = model10.fit(x=Xtrain, y=Ytrain, validation_data=(Xval,
Yval), batch_size=batch_size, epochs=epochs,
class_weight=class_weights)
```

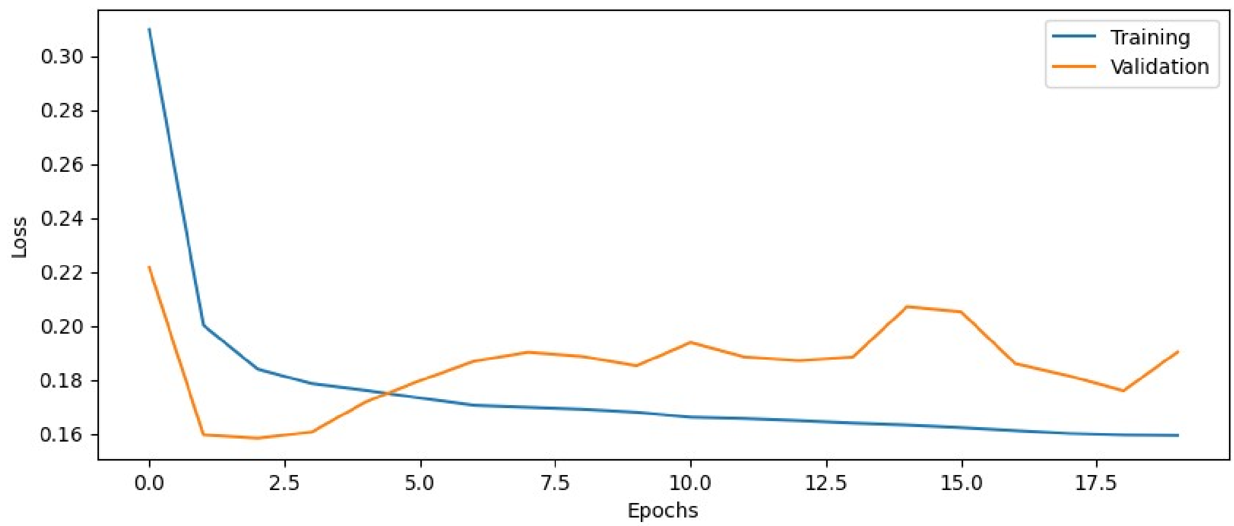
```
Epoch 1/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 5s 28ms/step - accuracy: 0.8217 - loss:
0.4298 - val_accuracy: 0.9134 - val_loss: 0.2215
Epoch 2/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 1s 21ms/step - accuracy: 0.9107 - loss:
0.2078 - val_accuracy: 0.9159 - val_loss: 0.1596
Epoch 3/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 1s 21ms/step - accuracy: 0.9139 - loss:
0.1858 - val_accuracy: 0.9157 - val_loss: 0.1583
Epoch 4/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 1s 20ms/step - accuracy: 0.9143 - loss:
0.1799 - val_accuracy: 0.9160 - val_loss: 0.1607
```

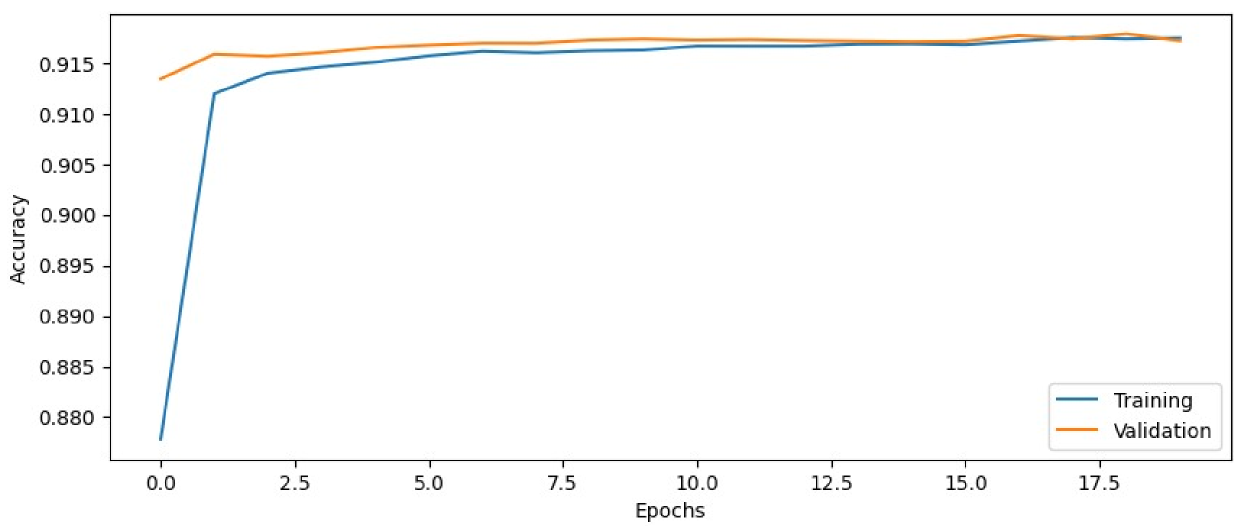
```
Epoch 5/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 1s 20ms/step - accuracy: 0.9144 - loss:
0.1776 - val_accuracy: 0.9165 - val_loss: 0.1718
Epoch 6/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 1s 21ms/step - accuracy: 0.9156 - loss:
0.1740 - val_accuracy: 0.9168 - val_loss: 0.1797
Epoch 7/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 1s 20ms/step - accuracy: 0.9157 - loss:
0.1712 - val_accuracy: 0.9170 - val_loss: 0.1868
Epoch 8/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 1s 21ms/step - accuracy: 0.9160 - loss:
0.1700 - val_accuracy: 0.9170 - val_loss: 0.1902
Epoch 9/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 1s 21ms/step - accuracy: 0.9163 - loss:
0.1691 - val_accuracy: 0.9173 - val_loss: 0.1886
Epoch 10/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 1s 21ms/step - accuracy: 0.9159 - loss:
0.1693 - val_accuracy: 0.9174 - val_loss: 0.1852
Epoch 11/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 1s 21ms/step - accuracy: 0.9171 - loss:
0.1658 - val_accuracy: 0.9173 - val_loss: 0.1938
Epoch 12/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 1s 21ms/step - accuracy: 0.9165 - loss:
0.1659 - val_accuracy: 0.9173 - val_loss: 0.1884
Epoch 13/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 1s 22ms/step - accuracy: 0.9171 - loss:
0.1644 - val_accuracy: 0.9172 - val_loss: 0.1871
Epoch 14/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 1s 21ms/step - accuracy: 0.9168 - loss:
0.1643 - val_accuracy: 0.9172 - val_loss: 0.1884
Epoch 15/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 1s 22ms/step - accuracy: 0.9166 - loss:
0.1639 - val_accuracy: 0.9171 - val_loss: 0.2070
Epoch 16/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 1s 21ms/step - accuracy: 0.9169 - loss:
0.1627 - val_accuracy: 0.9171 - val_loss: 0.2051
Epoch 17/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 1s 21ms/step - accuracy: 0.9173 - loss:
0.1612 - val_accuracy: 0.9177 - val_loss: 0.1860
Epoch 18/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 1s 21ms/step - accuracy: 0.9168 - loss:
0.1610 - val_accuracy: 0.9174 - val_loss: 0.1814
Epoch 19/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 1s 22ms/step - accuracy: 0.9168 - loss:
0.1609 - val_accuracy: 0.9179 - val_loss: 0.1759
Epoch 20/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 1s 22ms/step - accuracy: 0.9168 - loss:
0.1602 - val_accuracy: 0.9172 - val_loss: 0.1903
```

```
# Evaluate DNN on test data
score = model10.evaluate(x=Xtest, y=Ytest)

print('Test loss: %.4f' % score[0])
print('Test accuracy: %.4f' % score[1])
```

```
  37/3582 ━━━━━━━━━━━━━━━━━━━━ 5s 1ms/step - accuracy: 0.9286 - loss: 0.1627

3582/3582 ━━━━━━━━━━━━━━━━━━━━ 5s 1ms/step - accuracy: 0.9198 - loss: 0.1846
Test loss: 0.1873
Test accuracy: 0.9188
```

```
plot_results(history10)
```





Part 20: Dropout uncertainty

Dropout can also be used during testing, to obtain an estimate of the model uncertainty. Since dropout will randomly remove connections, the network will produce different results every time the same (test) data is put into the network. This technique is called Monte Carlo dropout. For more information, see this paper http://proceedings.mlr.press/v48/gal16.pdf

To achieve this, we need to redefine the Keras Dropout call by running the cell below, and use 'myDropout' in each call to Dropout, in the cell that defines the DNN. The `build_DNN` function takes two boolean arguments, use_dropout and use_custom_dropout, add a standard Dropout layer if use_dropout is true, add a myDropout layer if use_custom_dropout is true.

Run the same test data through the trained network 100 times, with dropout turned on.

Question 18: What is the mean and the standard deviation of the test accuracy?

```
import keras.backend as K
import keras

class myDropout(keras.layers.Dropout):
    """Applies Dropout to the input.
    Dropout consists in randomly setting
    a fraction `rate` of input units to 0 at each update during
training time,
    which helps prevent overfitting.
    # Arguments
        rate: float between 0 and 1. Fraction of the input units to
drop.
        noise_shape: 1D integer tensor representing the shape of the
            binary dropout mask that will be multiplied with the
input.
            For instance, if your inputs have shape
            `(batch_size, timesteps, features)` and
            you want the dropout mask to be the same for all
timesteps,
            you can use `noise_shape=(batch_size, 1, features)`.
        seed: A Python integer to use as random seed.
    # References
        - [Dropout: A Simple Way to Prevent Neural Networks from
Overfitting](

http://www.jmlr.org/papers/volume15/srivastava14a/srivastava14a.pdf)
    """
    def __init__(self, rate, training=True, noise_shape=None,
seed=None, **kwargs):
        super(myDropout, self).__init__(rate, noise_shape=None,
seed=None,**kwargs)
        self.training = training
```

```
    def call(self, inputs, training=None):
        if 0. < self.rate < 1.:
            noise_shape = self._get_noise_shape(inputs)

            def dropped_inputs():
                return K.dropout(inputs, self.rate, noise_shape,
                                 seed=self.seed)
            if not training:
                return K.in_train_phase(dropped_inputs, inputs,
training=self.training)
            return K.in_train_phase(dropped_inputs, inputs,
training=training)
        return inputs
```

Your best config, custom dropout

```
# Your best training parameters
# Build and train model

"""
We were not able to install tensorflow==2.7.0, and perhaps because of
this we were getting an error on the self._get_noise_shape(inputs)
line. We didn't manage to fix the problem, so we're skipping the
questions in this section.
"""
model11 = build_DNN(input_shape=input_shape, n_layers=4, n_nodes=50,
optimizer="adam", act_fun="relu", use_bn=True,
use_custom_dropout=True)

history11 = model11.fit(x=Xtrain, y=Ytrain, validation_data=(Xval,
Yval), batch_size=batch_size, epochs=epochs,
class_weight=class_weights)
```

```
Epoch 1/20
```

```
----------------------------------------------------------------------
-----
AttributeError                            Traceback (most recent call
last)
Cell In[49], line 9
      4 """
      5 We were not able to install tensorflow==2.7.0, and perhaps
because of this we were getting an error on the
self._get_noise_shape(inputs) line. We didn't manage to fix the
problem, so we're skipping the questions in this section.
      6 """
      7 model11 = build_DNN(input_shape=input_shape, n_layers=4,
n_nodes=50, optimizer="adam", act_fun="relu", use_bn=True,
use_custom_dropout=True)
----> 9 history11 = model11.fit(x=Xtrain, y=Ytrain,
```

```
validation_data=(Xval, Yval), batch_size=batch_size, epochs=epochs,
class_weight=class_weights)

File c:\Users\marij\AppData\Local\Programs\Python\Python312\Lib\site-
packages\keras\src\utils\traceback_utils.py:122, in
filter_traceback.<locals>.error_handler(*args, **kwargs)
    119     filtered_tb = _process_traceback_frames(e.__traceback__)
    120     # To get the full stack trace, call:
    121     # `keras.config.disable_traceback_filtering()`
--> 122     raise e.with_traceback(filtered_tb) from None
    123 finally:
    124     del filtered_tb

Cell In[47], line 29, in myDropout.call(self, inputs, training)
     27 def call(self, inputs, training=None):
     28     if 0. < self.rate < 1.:
---> 29         noise_shape = self._get_noise_shape(inputs)
     31         def dropped_inputs():
     32             return K.dropout(inputs, self.rate, noise_shape,
     33                              seed=self.seed)

AttributeError: Exception encountered when calling myDropout.call().

'myDropout' object has no attribute '_get_noise_shape'

Arguments received by myDropout.call():
  • inputs=tf.Tensor(shape=(None, 50), dtype=float32)
  • training=True

# Run this cell a few times to evalute the model on test data,
# if you get slightly different test accuracy every time, Dropout
during testing is working

# Evaluate model on test data
score = model11.evaluate(x=Xtest, y=Ytest)

print('Test accuracy: %.4f' % score[1])

# Run the testing 100 times, and save the accuracies in an array


# Calculate and print mean and std of accuracies
```

Part 21: Cross validation uncertainty

Cross validation (CV) is often used to evaluate a model, by training and testing using different subsets of the data it is possible to get the uncertainty as the standard deviation over folds. We here use a help function from scikit-learn to setup the CV, see https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.StratifiedKFold.html . Use 10 folds with shuffling, random state 1234.

Note: We here assume that you have found the best hyper parameters, so here the data are only split into training and testing, no validation.

Question 19: What is the mean and the standard deviation of the test accuracy?

The mean accuracy is 0.919 and the standard deviation 0.002.

Question 20: What is the main advantage of dropout compared to CV for estimating test uncertainty? The difference may not be so large in this notebook, but imagine that you have a network that takes 24 hours to train.

When using 10-fold CV, the model has to be trained 10 times. If your network takes 24 hours to train, this means it's going to take 10 days instead of just one. This is very expensive, compared to dropout which only needs to train the model once.

```
from sklearn.model_selection import StratifiedKFold

# Define 10-fold cross validation
skf = StratifiedKFold(n_splits=10, shuffle=True, random_state=1234)

test_acc = np.zeros(shape=(10,))

# Loop over cross validation folds
for i, (train_index, test_index) in enumerate(skf.split(X, Y)):
    # Calculate class weights for current split
    Xtrain = X[train_index, :]
    Ytrain = Y[train_index]

    Xtest = X[test_index, :]
    Ytest = Y[test_index]

    class_weights = 
class_weight.compute_class_weight(class_weight="balanced", 
classes=np.unique(Y), y=Ytrain)
    class_weights = {0: class_weights[0],
                1: class_weights[1]}

    # Rebuild the DNN model, to not continue training on the 
previously trained model
    model12 = build_DNN(input_shape=input_shape, n_layers=4, 
n_nodes=50, optimizer="adam", act_fun="relu", use_bn=True, 
use_dropout=True)
```

```
    # Fit the model with training set and class weights for this fold
    history12 = model12.fit(x=Xtrain, y=Ytrain, batch_size=batch_size,
epochs=epochs, class_weight=class_weights)

    # Evaluate the model using the test set for this fold
    score = model12.evaluate(x=Xtest, y=Ytest)

    # Save the test accuracy in an array
    test_acc[i] = score[1]

# Calculate and print mean and std of accuracies
mean_test_acc = np.mean(test_acc)
std_test_acc = np.std(test_acc)

print(f"Mean test acc.: {np.round(mean_test_acc, 3)}\nStd of test
acc.: {np.round(std_test_acc, 3)}")

Epoch 1/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 5s 19ms/step - accuracy: 0.8656 - loss:
0.3497
Epoch 2/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 1s 18ms/step - accuracy: 0.9127 - loss:
0.1923
Epoch 3/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 1s 18ms/step - accuracy: 0.9144 - loss:
0.1791
Epoch 4/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 1s 19ms/step - accuracy: 0.9151 - loss:
0.1741
Epoch 5/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 1s 19ms/step - accuracy: 0.9158 - loss:
0.1714
Epoch 6/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 1s 18ms/step - accuracy: 0.9159 - loss:
0.1688
Epoch 7/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 1s 19ms/step - accuracy: 0.9164 - loss:
0.1667
Epoch 8/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 1s 19ms/step - accuracy: 0.9164 - loss:
0.1653
Epoch 9/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 1s 19ms/step - accuracy: 0.9172 - loss:
0.1634
Epoch 10/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 1s 19ms/step - accuracy: 0.9167 - loss:
0.1625
Epoch 11/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 1s 19ms/step - accuracy: 0.9171 - loss:
```

```
0.1608
Epoch 12/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 1s 19ms/step - accuracy: 0.9174 - loss:
0.1597
Epoch 13/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 1s 18ms/step - accuracy: 0.9177 - loss:
0.1591
Epoch 14/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 1s 19ms/step - accuracy: 0.9184 - loss:
0.1576
Epoch 15/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 1s 18ms/step - accuracy: 0.9183 - loss:
0.1563
Epoch 16/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 1s 19ms/step - accuracy: 0.9193 - loss:
0.1545
Epoch 17/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 1s 19ms/step - accuracy: 0.9195 - loss:
0.1528
Epoch 18/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 1s 18ms/step - accuracy: 0.9189 - loss:
0.1536
Epoch 19/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 1s 18ms/step - accuracy: 0.9199 - loss:
0.1520
Epoch 20/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 1s 20ms/step - accuracy: 0.9200 - loss:
0.1509
2388/2388 ━━━━━━━━━━━━━━━━━━━━ 3s 1ms/step - accuracy: 0.9439 - loss:
0.1553
Epoch 1/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 6s 19ms/step - accuracy: 0.8320 - loss:
0.3922
Epoch 2/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 1s 19ms/step - accuracy: 0.9127 - loss:
0.1941
Epoch 3/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 1s 20ms/step - accuracy: 0.9140 - loss:
0.1797
Epoch 4/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 1s 19ms/step - accuracy: 0.9142 - loss:
0.1757
Epoch 5/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 1s 20ms/step - accuracy: 0.9152 - loss:
0.1726
Epoch 6/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 1s 19ms/step - accuracy: 0.9154 - loss:
0.1716
Epoch 7/20
```

```
69/69 ━━━━━━━━━━━━━━━━━━━━ 1s 19ms/step - accuracy: 0.9162 - loss:
0.1680
Epoch 8/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 1s 19ms/step - accuracy: 0.9156 - loss:
0.1676
Epoch 9/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 1s 19ms/step - accuracy: 0.9168 - loss:
0.1665
Epoch 10/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 1s 19ms/step - accuracy: 0.9164 - loss:
0.1657
Epoch 11/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 1s 20ms/step - accuracy: 0.9162 - loss:
0.1647
Epoch 12/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 1s 19ms/step - accuracy: 0.9170 - loss:
0.1627
Epoch 13/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 1s 19ms/step - accuracy: 0.9168 - loss:
0.1617
Epoch 14/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 1s 19ms/step - accuracy: 0.9171 - loss:
0.1600
Epoch 15/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 1s 19ms/step - accuracy: 0.9169 - loss:
0.1596
Epoch 16/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 1s 19ms/step - accuracy: 0.9168 - loss:
0.1586
Epoch 17/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 1s 18ms/step - accuracy: 0.9178 - loss:
0.1562
Epoch 18/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 1s 19ms/step - accuracy: 0.9181 - loss:
0.1557
Epoch 19/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 1s 18ms/step - accuracy: 0.9181 - loss:
0.1558
Epoch 20/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 1s 19ms/step - accuracy: 0.9190 - loss:
0.1542
2388/2388 ━━━━━━━━━━━━━━━━━━━━ 3s 1ms/step - accuracy: 0.9430 - loss:
0.1555
Epoch 1/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 6s 19ms/step - accuracy: 0.8506 - loss:
0.3734
Epoch 2/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 1s 19ms/step - accuracy: 0.9130 - loss:
0.1970
```

```
Epoch 3/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 1s 20ms/step - accuracy: 0.9141 - loss:
0.1804
Epoch 4/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 1s 18ms/step - accuracy: 0.9146 - loss:
0.1762
Epoch 5/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 1s 19ms/step - accuracy: 0.9154 - loss:
0.1729
Epoch 6/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 1s 19ms/step - accuracy: 0.9152 - loss:
0.1719
Epoch 7/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 1s 19ms/step - accuracy: 0.9158 - loss:
0.1697
Epoch 8/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 1s 18ms/step - accuracy: 0.9160 - loss:
0.1691
Epoch 9/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 1s 19ms/step - accuracy: 0.9163 - loss:
0.1672
Epoch 10/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 1s 19ms/step - accuracy: 0.9175 - loss:
0.1645
Epoch 11/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 1s 19ms/step - accuracy: 0.9171 - loss:
0.1643
Epoch 12/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 1s 19ms/step - accuracy: 0.9173 - loss:
0.1625
Epoch 13/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 1s 19ms/step - accuracy: 0.9173 - loss:
0.1620
Epoch 14/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 1s 19ms/step - accuracy: 0.9170 - loss:
0.1607
Epoch 15/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 1s 19ms/step - accuracy: 0.9172 - loss:
0.1610
Epoch 16/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 1s 18ms/step - accuracy: 0.9174 - loss:
0.1584
Epoch 17/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 1s 19ms/step - accuracy: 0.9184 - loss:
0.1560
Epoch 18/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 1s 19ms/step - accuracy: 0.9181 - loss:
0.1557
Epoch 19/20
```

```
69/69 ━━━━━━━━━━━━━━━━━━━━ 1s 19ms/step - accuracy: 0.9187 - loss:
0.1543
Epoch 20/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 1s 19ms/step - accuracy: 0.9197 - loss:
0.1532
2388/2388 ━━━━━━━━━━━━━━━━━━━━ 3s 1ms/step - accuracy: 0.9420 - loss:
0.1618
Epoch 1/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 5s 20ms/step - accuracy: 0.8323 - loss:
0.4040
Epoch 2/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 1s 18ms/step - accuracy: 0.9119 - loss:
0.1996
Epoch 3/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 1s 19ms/step - accuracy: 0.9147 - loss:
0.1807
Epoch 4/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 1s 19ms/step - accuracy: 0.9152 - loss:
0.1763
Epoch 5/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 1s 18ms/step - accuracy: 0.9157 - loss:
0.1734
Epoch 6/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 1s 19ms/step - accuracy: 0.9164 - loss:
0.1719
Epoch 7/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 1s 18ms/step - accuracy: 0.9169 - loss:
0.1693
Epoch 8/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 1s 19ms/step - accuracy: 0.9166 - loss:
0.1683
Epoch 9/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 1s 19ms/step - accuracy: 0.9170 - loss:
0.1662
Epoch 10/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 1s 19ms/step - accuracy: 0.9168 - loss:
0.1660
Epoch 11/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 1s 19ms/step - accuracy: 0.9169 - loss:
0.1646
Epoch 12/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 1s 19ms/step - accuracy: 0.9175 - loss:
0.1621
Epoch 13/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 1s 20ms/step - accuracy: 0.9176 - loss:
0.1616
Epoch 14/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 1s 20ms/step - accuracy: 0.9174 - loss:
0.1603
```

```
Epoch 15/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 2s 20ms/step - accuracy: 0.9174 - loss:
0.1589
Epoch 16/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 2s 21ms/step - accuracy: 0.9181 - loss:
0.1576
Epoch 17/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 1s 20ms/step - accuracy: 0.9181 - loss:
0.1554
Epoch 18/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 1s 18ms/step - accuracy: 0.9181 - loss:
0.1560
Epoch 19/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 1s 20ms/step - accuracy: 0.9189 - loss:
0.1538
Epoch 20/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 1s 19ms/step - accuracy: 0.9191 - loss:
0.1526
2388/2388 ━━━━━━━━━━━━━━━━━━━━ 3s 1ms/step - accuracy: 0.9410 - loss:
0.1615
Epoch 1/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 5s 20ms/step - accuracy: 0.8528 - loss:
0.3845
Epoch 2/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 1s 19ms/step - accuracy: 0.9137 - loss:
0.1947
Epoch 3/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 1s 18ms/step - accuracy: 0.9143 - loss:
0.1801
Epoch 4/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 1s 19ms/step - accuracy: 0.9150 - loss:
0.1756
Epoch 5/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 1s 19ms/step - accuracy: 0.9158 - loss:
0.1721
Epoch 6/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 1s 18ms/step - accuracy: 0.9165 - loss:
0.1694
Epoch 7/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 1s 19ms/step - accuracy: 0.9167 - loss:
0.1678
Epoch 8/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 1s 19ms/step - accuracy: 0.9167 - loss:
0.1666
Epoch 9/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 1s 19ms/step - accuracy: 0.9165 - loss:
0.1663
Epoch 10/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 1s 19ms/step - accuracy: 0.9175 - loss:
```

```
0.1630
Epoch 11/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 1s 19ms/step - accuracy: 0.9171 - loss:
0.1631
Epoch 12/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 1s 18ms/step - accuracy: 0.9168 - loss:
0.1620
Epoch 13/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 1s 18ms/step - accuracy: 0.9175 - loss:
0.1601
Epoch 14/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 1s 19ms/step - accuracy: 0.9183 - loss:
0.1588
Epoch 15/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 1s 19ms/step - accuracy: 0.9178 - loss:
0.1582
Epoch 16/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 1s 19ms/step - accuracy: 0.9177 - loss:
0.1591
Epoch 17/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 1s 19ms/step - accuracy: 0.9175 - loss:
0.1569
Epoch 18/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 1s 18ms/step - accuracy: 0.9181 - loss:
0.1580
Epoch 19/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 1s 19ms/step - accuracy: 0.9176 - loss:
0.1556
Epoch 20/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 1s 18ms/step - accuracy: 0.9179 - loss:
0.1550
2388/2388 ━━━━━━━━━━━━━━━━━━━━ 3s 1ms/step - accuracy: 0.9415 - loss:
0.1726
Epoch 1/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 5s 20ms/step - accuracy: 0.8554 - loss:
0.3734
Epoch 2/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 1s 19ms/step - accuracy: 0.9133 - loss:
0.1916
Epoch 3/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 1s 19ms/step - accuracy: 0.9147 - loss:
0.1779
Epoch 4/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 1s 19ms/step - accuracy: 0.9146 - loss:
0.1746
Epoch 5/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 1s 19ms/step - accuracy: 0.9153 - loss:
0.1722
Epoch 6/20
```

```
69/69 ━━━━━━━━━━━━━━━━━━━━ 1s 19ms/step - accuracy: 0.9155 - loss:
0.1704
Epoch 7/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 1s 20ms/step - accuracy: 0.9158 - loss:
0.1686
Epoch 8/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 1s 19ms/step - accuracy: 0.9167 - loss:
0.1662
Epoch 9/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 1s 19ms/step - accuracy: 0.9161 - loss:
0.1653
Epoch 10/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 1s 19ms/step - accuracy: 0.9169 - loss:
0.1625
Epoch 11/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 1s 19ms/step - accuracy: 0.9172 - loss:
0.1618
Epoch 12/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 1s 19ms/step - accuracy: 0.9170 - loss:
0.1609
Epoch 13/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 1s 19ms/step - accuracy: 0.9182 - loss:
0.1584
Epoch 14/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 1s 19ms/step - accuracy: 0.9188 - loss:
0.1579
Epoch 15/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 1s 19ms/step - accuracy: 0.9183 - loss:
0.1573
Epoch 16/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 1s 18ms/step - accuracy: 0.9189 - loss:
0.1566
Epoch 17/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 1s 19ms/step - accuracy: 0.9200 - loss:
0.1537
Epoch 18/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 1s 19ms/step - accuracy: 0.9193 - loss:
0.1541
Epoch 19/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 1s 19ms/step - accuracy: 0.9194 - loss:
0.1523
Epoch 20/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 1s 20ms/step - accuracy: 0.9195 - loss:
0.1522
2388/2388 ━━━━━━━━━━━━━━━━━━━━ 3s 1ms/step - accuracy: 0.9443 - loss:
0.1567
Epoch 1/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 6s 21ms/step - accuracy: 0.8697 - loss:
0.3575
```

```
Epoch 2/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 1s 20ms/step - accuracy: 0.9127 - loss:
0.1897
Epoch 3/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 1s 20ms/step - accuracy: 0.9140 - loss:
0.1779
Epoch 4/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 1s 20ms/step - accuracy: 0.9143 - loss:
0.1754
Epoch 5/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 2s 21ms/step - accuracy: 0.9154 - loss:
0.1721
Epoch 6/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 2s 20ms/step - accuracy: 0.9164 - loss:
0.1687
Epoch 7/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 1s 20ms/step - accuracy: 0.9165 - loss:
0.1669
Epoch 8/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 1s 20ms/step - accuracy: 0.9170 - loss:
0.1648
Epoch 9/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 1s 20ms/step - accuracy: 0.9167 - loss:
0.1632
Epoch 10/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 1s 20ms/step - accuracy: 0.9170 - loss:
0.1618
Epoch 11/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 1s 19ms/step - accuracy: 0.9171 - loss:
0.1611
Epoch 12/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 2s 20ms/step - accuracy: 0.9173 - loss:
0.1593
Epoch 13/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 1s 20ms/step - accuracy: 0.9176 - loss:
0.1577
Epoch 14/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 1s 20ms/step - accuracy: 0.9172 - loss:
0.1568
Epoch 15/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 1s 20ms/step - accuracy: 0.9173 - loss:
0.1572
Epoch 16/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 1s 19ms/step - accuracy: 0.9177 - loss:
0.1554
Epoch 17/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 1s 20ms/step - accuracy: 0.9191 - loss:
0.1524
Epoch 18/20
```

```
69/69 ━━━━━━━━━━━━━━━━━━━━ 1s 20ms/step - accuracy: 0.9190 - loss:
0.1533
Epoch 19/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 1s 19ms/step - accuracy: 0.9205 - loss:
0.1513
Epoch 20/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 1s 20ms/step - accuracy: 0.9206 - loss:
0.1509
2388/2388 ━━━━━━━━━━━━━━━━━━━━ 3s 1ms/step - accuracy: 0.9457 - loss:
0.1587
Epoch 1/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 5s 22ms/step - accuracy: 0.8170 - loss:
0.4338
Epoch 2/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 2s 21ms/step - accuracy: 0.9119 - loss:
0.2040
Epoch 3/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 2s 21ms/step - accuracy: 0.9138 - loss:
0.1845
Epoch 4/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 2s 21ms/step - accuracy: 0.9149 - loss:
0.1793
Epoch 5/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 2s 21ms/step - accuracy: 0.9159 - loss:
0.1746
Epoch 6/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 2s 21ms/step - accuracy: 0.9166 - loss:
0.1719
Epoch 7/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 2s 21ms/step - accuracy: 0.9163 - loss:
0.1716
Epoch 8/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 2s 21ms/step - accuracy: 0.9159 - loss:
0.1707
Epoch 9/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 2s 21ms/step - accuracy: 0.9175 - loss:
0.1676
Epoch 10/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 1s 20ms/step - accuracy: 0.9165 - loss:
0.1677
Epoch 11/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 2s 21ms/step - accuracy: 0.9168 - loss:
0.1677
Epoch 12/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 2s 20ms/step - accuracy: 0.9176 - loss:
0.1652
Epoch 13/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 2s 21ms/step - accuracy: 0.9170 - loss:
0.1647
```

```
Epoch 14/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 1s 20ms/step - accuracy: 0.9173 - loss:
0.1629
Epoch 15/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 1s 20ms/step - accuracy: 0.9168 - loss:
0.1632
Epoch 16/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 2s 21ms/step - accuracy: 0.9175 - loss:
0.1619
Epoch 17/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 1s 20ms/step - accuracy: 0.9176 - loss:
0.1601
Epoch 18/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 2s 21ms/step - accuracy: 0.9180 - loss:
0.1590
Epoch 19/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 1s 20ms/step - accuracy: 0.9180 - loss:
0.1588
Epoch 20/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 1s 20ms/step - accuracy: 0.9188 - loss:
0.1571
2388/2388 ━━━━━━━━━━━━━━━━━━━━ 3s 1ms/step - accuracy: 0.9434 - loss:
0.1671
Epoch 1/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 6s 22ms/step - accuracy: 0.8455 - loss:
0.3896
Epoch 2/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 2s 22ms/step - accuracy: 0.9121 - loss:
0.1958
Epoch 3/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 2s 20ms/step - accuracy: 0.9139 - loss:
0.1802
Epoch 4/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 2s 21ms/step - accuracy: 0.9150 - loss:
0.1755
Epoch 5/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 2s 21ms/step - accuracy: 0.9158 - loss:
0.1726
Epoch 6/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 2s 21ms/step - accuracy: 0.9163 - loss:
0.1704
Epoch 7/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 1s 20ms/step - accuracy: 0.9166 - loss:
0.1684
Epoch 8/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 1s 20ms/step - accuracy: 0.9161 - loss:
0.1679
Epoch 9/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 2s 21ms/step - accuracy: 0.9174 - loss:
0.1659
```

```
Epoch 10/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 1s 20ms/step - accuracy: 0.9172 - loss:
0.1642
Epoch 11/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 2s 20ms/step - accuracy: 0.9171 - loss:
0.1642
Epoch 12/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 2s 21ms/step - accuracy: 0.9171 - loss:
0.1629
Epoch 13/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 2s 23ms/step - accuracy: 0.9170 - loss:
0.1619
Epoch 14/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 1s 20ms/step - accuracy: 0.9173 - loss:
0.1611
Epoch 15/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 1s 20ms/step - accuracy: 0.9179 - loss:
0.1601
Epoch 16/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 1s 19ms/step - accuracy: 0.9174 - loss:
0.1588
Epoch 17/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 2s 21ms/step - accuracy: 0.9178 - loss:
0.1572
Epoch 18/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 1s 19ms/step - accuracy: 0.9172 - loss:
0.1577
Epoch 19/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 1s 19ms/step - accuracy: 0.9177 - loss:
0.1558
Epoch 20/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 1s 20ms/step - accuracy: 0.9183 - loss:
0.1552
2388/2388 ━━━━━━━━━━━━━━━━━━━━ 4s 1ms/step - accuracy: 0.9439 - loss:
0.1581
Epoch 1/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 5s 21ms/step - accuracy: 0.8493 - loss:
0.3811
Epoch 2/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 1s 20ms/step - accuracy: 0.9138 - loss:
0.1920
Epoch 3/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 2s 21ms/step - accuracy: 0.9143 - loss:
0.1799
Epoch 4/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 1s 20ms/step - accuracy: 0.9151 - loss:
0.1754
Epoch 5/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 1s 20ms/step - accuracy: 0.9153 - loss:
```

```
0.1728
Epoch 6/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 2s 21ms/step - accuracy: 0.9156 - loss:
0.1716
Epoch 7/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 1s 20ms/step - accuracy: 0.9171 - loss:
0.1677
Epoch 8/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 2s 21ms/step - accuracy: 0.9168 - loss:
0.1669
Epoch 9/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 1s 20ms/step - accuracy: 0.9167 - loss:
0.1654
Epoch 10/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 1s 20ms/step - accuracy: 0.9173 - loss:
0.1635
Epoch 11/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 1s 20ms/step - accuracy: 0.9169 - loss:
0.1623
Epoch 12/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 1s 20ms/step - accuracy: 0.9172 - loss:
0.1611
Epoch 13/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 1s 20ms/step - accuracy: 0.9183 - loss:
0.1589
Epoch 14/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 1s 20ms/step - accuracy: 0.9173 - loss:
0.1597
Epoch 15/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 2s 21ms/step - accuracy: 0.9183 - loss:
0.1573
Epoch 16/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 1s 20ms/step - accuracy: 0.9187 - loss:
0.1554
Epoch 17/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 1s 20ms/step - accuracy: 0.9183 - loss:
0.1544
Epoch 18/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 1s 20ms/step - accuracy: 0.9178 - loss:
0.1551
Epoch 19/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 1s 20ms/step - accuracy: 0.9192 - loss:
0.1523
Epoch 20/20
69/69 ━━━━━━━━━━━━━━━━━━━━ 2s 21ms/step - accuracy: 0.9202 - loss:
0.1504
2388/2388 ━━━━━━━━━━━━━━━━━━━━ 3s 1ms/step - accuracy: 0.9430 - loss:
0.1553
```

```
Mean test acc.: 0.919
Std of test acc.: 0.002
```

Part 22: DNN regression

A similar DNN can be used for regression, instead of classification.

Question 21: How would you change the DNN used in this lab in order to use it for regression instead?

If you want to do regression instead of classification, no activation function should be used on the output layer since we don't want to compress the values down to a 0-1 range.

Report

Send in this jupyter notebook, with answers to all questions.