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.

In [ ]:
```
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
physical_devices = tf.config.experimental.list_physical_devices('GPU')
tf.config.experimental.set_memory_growth(physical_devices[0], True)
```

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?

- 24 GB

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

- training data and parameters(weights, outputs in each layer, etc.)

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.

In [ ]:
```
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 lo
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
```

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

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 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.

In [ ]:
```
# It is common to have NaNs in the data, lets check for it. Hint: np.isnan()
print(np.isnan(X).any())
print(np.isnan(Y).any())

# Print the number of NaNs (not a number) in the labels
print(np.count_nonzero(np.isnan(X)))

# Print the number of NaNs in the covariates
print(np.count_nonzero(np.isnan(Y)))
```

```
False
False
0
0
```

Part 6: Preprocessing

Lets do some simple preprocessing

In [ ]:
```
# Convert covariates to floats
X = X.astype(float)

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

# Remove mean of each covariate (column)
for col in range(X.shape[1]):
    mean = np.mean(X[:,col])
    for row in range(X.shape[0]):
        X[row, col] -= mean


# Divide each covariate (column) by its standard deviation
for col in range(X.shape[1]):
    stdev = np.std(X[:,col])
    for row in range(X.shape[0]):
        X[row, col] /= stdev

# Check that mean is 0 and standard deviation is 1 for all covariates, by printi
meanList = []
stdevList = []
for col in range(X.shape[1]):
    meanList.append(np.mean(X[:,col]))
    stdevList.append(np.std(X[:,col]))

print(meanList)
print(stdevList)
```

```
[-2.615704665123089e-17, 2.910099296114191e-16, 1.757069155782648e-16, 1.58299879
9113241e-16, 5.570251413421029e-16, 7.989383036877916e-17, 1.1143478388494525e-1
6, 5.604098427217858e-16, 2.423594965935592e-16, -5.4192416595582517e-17, -2.4771
550756800247e-17, -7.327320569203676e-18, 9.74496441183433e-18, -1.53985315515244
77e-17, -4.945383466402643e-16, -5.787467414051228e-17, 6.27843508670853e-17, -7.
659095693453913e-16, 1.2692630173810936e-17, -2.7152000078775045e-18, -1.85972603
27928113e-17, 4.0021304225701295e-17, 1.0034337782536893e-15, 1.4357084973160502e
-16, -7.648681227670274e-16, 9.921638384949648e-18, -1.9564317864980374e-17, 1.50
07989084637987e-16, -3.670355298319892e-16, -7.58768221379467e-16, -1.69607014190
70437e-17, -7.58247498090285e-16, -6.267276730511774e-18, -2.0456986360720924e-1
7, -1.0473977016689112e-16, -8.084601009756909e-16, 5.890124291061392e-16, 1.2051
024692497417e-17, -3.865254586556579e-16, -2.287463020335158e-17, -7.438904131171
245e-19, 1.0265687701016318e-16, 5.604098427217858e-16, -3.0499506937802103e-18,
6.760104129201869e-18, -4.992992452842139e-16, 1.0098312358064965e-17, 5.76701042
7690507e-17, 3.987252614307787e-17, 1.368758360135509e-17, 4.7162652191625695e-1
7, -3.616051298162342e-16, -1.320405483282896e-18, -2.9383671318126417e-17, -1.78
5336991481099e-18, 8.123283311239e-17, -7.156225774186738e-17, -2.584275295168890
5e-16, 7.607023364535715e-16, 9.376738657341354e-17, 3.9240219291928315e-16, -1.5
751879497755l1e-17, -9.388529029807068e-19, 9.298630163964056e-20, 7.141347965924
395e-18, 7.58768221379467e-17, -3.0276339813866965e-17, 2.917538200245362e-16, 7.
438904131171245e-20, 8.823528279024017e-19, 1.2134712363973092e-18, -4.2520776013
774835e-16, -2.097473408825044e-15, -2.5143495963358808e-17, -7.38831958307928e-1
6, -8.219989064944225e-18, 1.4761575385292939e-19, -2.3641767191878614e-18, -1.39
8513976660194e-16, -1.2193107761402787e-15, 7.0669589246126825e-19, 8.95197723145
1477e-16, 5.806064674379157e-17, 1.6281320252715815e-18, -5.497815084443748e-18,
6.843791800677545e-17, 9.091084738704377e-16, -7.167384130383494e-17, -6.26355727
8446188e-16, -1.6123824704313673e-17, 8.694219203306392e-19, -4.760898643949597e-
18]
[1.0, 0.9999999999999998, 0.9999999999999998, 1.0000000000000002, 0.9999999999999
996, 0.9999999999999997, 0.9999999999999999, 0.9999999999999998, 0.99999999999999
98, 1.0000000000000004, 1.0, 1.0, 1.0000000000000002, 1.0, 0.9999999999999998, 1.
0, 1.0000000000000004, 0.9999999999999997, 1.0000000000000002, 0.999999999999999
6, 1.0, 1.0000000000000004, 0.9999999999999998, 0.9999999999999996, 0.99999999999
99998, 1.0, 0.9999999999999997, 1.0, 1.0, 1.0, 1.0000000000000002, 1.0, 1.0, 0.99
99999999999997, 0.9999999999999997, 0.9999999999999998, 0.9999999999999998, 1.0,
1.0, 0.9999999999999996, 0.9999999999999998, 0.9999999999999999, 0.99999999999999
98, 1.0, 0.9999999999999968, 0.9999999999999997, 1.0, 0.9999999999999956, 1.00000
00000000004, 0.9999999999999997, 1.0000000000000016, 1.0000000000000002, 0.999999
9999999999, 1.0000000000000004, 0.9999999999999998, 1.0000000000000004, 1.0, 1.0,
1.0, 0.9999999999999997, 1.0, 0.9999999999999997, 0.999999999999999, 0.9999999999
999976, 1.0, 1.0, 1.0000000000000002, 0.9999999999999998, 0.9999999999999998, 1.0
000000000000033, 0.9999999999999993, 1.0, 0.9999999999999998, 1.0, 0.999999999999
9998, 1.0000000000000002, 1.0000000000000013, 0.9999999999999991, 1.0, 1.00000000
00000002, 1.0000000000000004, 1.0000000000000004, 1.0000000000000002, 1.000000000
0000018, 0.9999999999999991, 1.0, 0.9999999999999996, 1.0, 0.9999999999999998, 0.
9999999999999993, 0.9999999999999994, 1.0000000000000007]
```

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

In [ ]:
```
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, random_stat



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 rema
```

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

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?

In [ ]:
```
from sklearn.model_selection import train_test_split

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

print('The validation and test data have size {}, {}, {} and {}'.format(Xval.sha
```

```
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?).

In [ ]:
```
from keras.models import Sequential, Model
from keras.layers import Input, Dense, BatchNormalization, Activation, Dropout
from tensorflow.keras.optimizers import SGD, Adam
from keras.losses import CategoricalCrossentropy

# 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'
              use_bn=False, use_dropout=False, use_custom_dropout=False):

    # Setup optimizer, depending on input parameter string
    if optimizer == 'sgd':
        optimizer = SGD(learning_rate=learning_rate, decay=1e-6, momentum=0.9, r

    if optimizer == 'adam':
        optimizer = Adam(learning_rate=0.1)

    # Setup a sequential model
    model = Sequential()

    # Add layers to the model, using the input parameters of the build_DNN funct


    # Add first layer, requires input shape
```

```
    model.add(Input(shape=(input_shape[1],)))



    # Add remaining layers, do not require input shape
    for i in range(n_layers,):
        if use_bn == False:
            model.add(Dense(n_nodes, activation=act_fun))
        if use_bn == True:
            model.add(Dense(n_nodes))
            model.add(Activation(act_fun))
            model.add(BatchNormalization())

            if use_dropout == True:
                if use_dropout < 1 and use_dropout>0 :
                    model.add(Dropout(rate=use_dropout))
            else:
                model.add(Dropout( rate=0.5))

            if use_custom_dropout:
                if use_custom_dropout == True:
                    use_custom_dropout = 0.5
                model.add(myDropout(use_custom_dropout))



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

    # Compile model
    model.compile(loss='binary_crossentropy', optimizer=optimizer, metrics=['acc


    return model
```

In [ ]:

```
# 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

In [ ]:
```
# Setup some training parameters
batch_size = 10000
epochs = 20

input_shape = X.shape

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

model1.summary()
```

```
Model: "sequential_35"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_200 (Dense)            (None, 20)                1860
_________________________________________________________________
dense_201 (Dense)            (None, 20)                420
_________________________________________________________________
dense_202 (Dense)            (None, 1)                 21
=================================================================
Total params: 2,301
Trainable params: 2,301
Non-trainable params: 0
_________________________________________________________________
```

In [ ]:
```
# Train the model, provide training data and validation data7
```

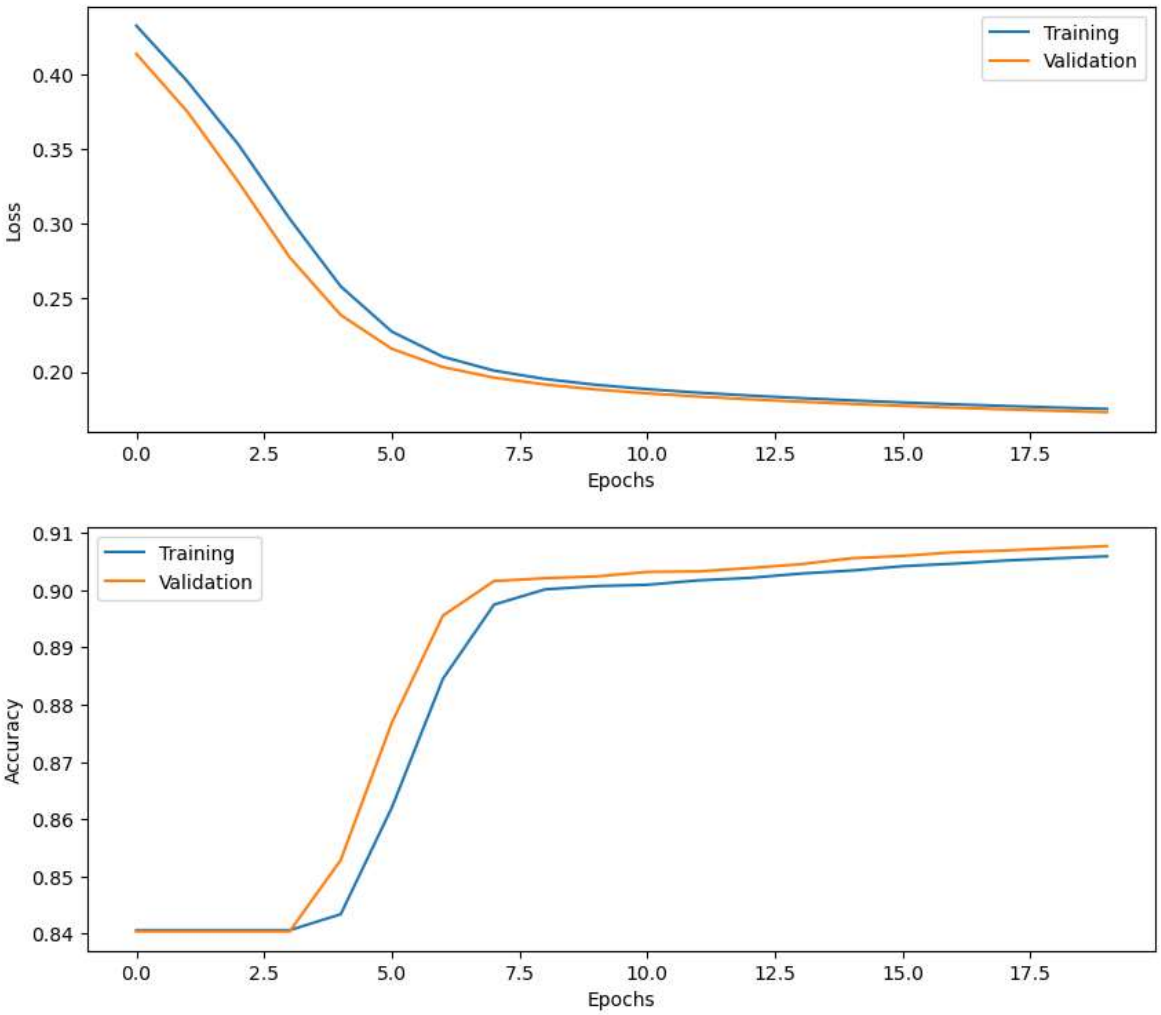
```
history1 = model1.fit(Xtrain, Ytrain, epochs=epochs, batch_size=batch_size,valic
```

```
Epoch 1/20
54/54 [==============================] - 0s 5ms/step - loss: 0.4974 - accuracy:
0.7626 - val_loss: 0.4118 - val_accuracy: 0.8404
Epoch 2/20
54/54 [==============================] - 0s 3ms/step - loss: 0.3942 - accuracy:
0.8406 - val_loss: 0.3740 - val_accuracy: 0.8404
Epoch 3/20
54/54 [==============================] - 0s 3ms/step - loss: 0.3513 - accuracy:
0.8406 - val_loss: 0.3256 - val_accuracy: 0.8404
Epoch 4/20
54/54 [==============================] - 0s 3ms/step - loss: 0.3011 - accuracy:
0.8406 - val_loss: 0.2756 - val_accuracy: 0.8404
Epoch 5/20
54/54 [==============================] - 0s 3ms/step - loss: 0.2568 - accuracy:
0.8431 - val_loss: 0.2382 - val_accuracy: 0.8533
Epoch 6/20
54/54 [==============================] - 0s 3ms/step - loss: 0.2274 - accuracy:
0.8634 - val_loss: 0.2161 - val_accuracy: 0.8811
Epoch 7/20
54/54 [==============================] - 0s 3ms/step - loss: 0.2108 - accuracy:
0.8880 - val_loss: 0.2038 - val_accuracy: 0.9030
Epoch 8/20
54/54 [==============================] - 0s 3ms/step - loss: 0.2014 - accuracy:
0.9020 - val_loss: 0.1965 - val_accuracy: 0.9043
Epoch 9/20
54/54 [==============================] - 0s 3ms/step - loss: 0.1955 - accuracy:
0.9024 - val_loss: 0.1917 - val_accuracy: 0.9043
Epoch 10/20
54/54 [==============================] - 0s 3ms/step - loss: 0.1915 - accuracy:
0.9027 - val_loss: 0.1883 - val_accuracy: 0.9046
Epoch 11/20
54/54 [==============================] - 0s 3ms/step - loss: 0.1885 - accuracy:
0.9033 - val_loss: 0.1856 - val_accuracy: 0.9055
Epoch 12/20
54/54 [==============================] - 0s 3ms/step - loss: 0.1860 - accuracy:
0.9039 - val_loss: 0.1834 - val_accuracy: 0.9059
Epoch 13/20
54/54 [==============================] - 0s 3ms/step - loss: 0.1840 - accuracy:
0.9044 - val_loss: 0.1815 - val_accuracy: 0.9069
Epoch 14/20
54/54 [==============================] - 0s 3ms/step - loss: 0.1822 - accuracy:
0.9054 - val_loss: 0.1798 - val_accuracy: 0.9072
Epoch 15/20
54/54 [==============================] - 0s 3ms/step - loss: 0.1807 - accuracy:
0.9057 - val_loss: 0.1783 - val_accuracy: 0.9076
Epoch 16/20
54/54 [==============================] - 0s 3ms/step - loss: 0.1793 - accuracy:
0.9061 - val_loss: 0.1770 - val_accuracy: 0.9078
Epoch 17/20
54/54 [==============================] - 0s 3ms/step - loss: 0.1781 - accuracy:
0.9063 - val_loss: 0.1759 - val_accuracy: 0.9081
Epoch 18/20
54/54 [==============================] - 0s 3ms/step - loss: 0.1770 - accuracy:
0.9065 - val_loss: 0.1748 - val_accuracy: 0.9083
Epoch 19/20
54/54 [==============================] - 0s 3ms/step - loss: 0.1760 - accuracy:
0.9067 - val_loss: 0.1738 - val_accuracy: 0.9086
Epoch 20/20
54/54 [==============================] - 0s 3ms/step - loss: 0.1751 - accuracy:
0.9069 - val_loss: 0.1729 - val_accuracy: 0.9090
```

In [ ]:
```
# Evaluate the model on the test data
score = model1.evaluate(Xtest,Ytest, batch_size=batch_size)

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

```
12/12 [==============================] - 0s 1ms/step - loss: 0.1741 - accuracy:
0.9058
Test loss: 0.1741
Test accuracy: 0.9058
```

In [ ]:
```
# 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?

- with no activation function, the NN will be linear and make extra dense layers cannot improve performance at all.

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

- with default setting of dense() (kernel_initializer="glorot_uniform",bias_initializer="zeros",), the weight is intialized by glorot_uniform(Xavier uniform initializer) and the bias is initialized by zeros.

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

In [ ]:
```
from sklearn.utils import class_weight

# Calculate class weights
class_weights = class_weight.compute_class_weight(class_weight='balanced', class

# Print the class weights
print(class_weights)

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

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

```
[3.13728768 0.59479436]
```

2 layers, 20 nodes, class weights

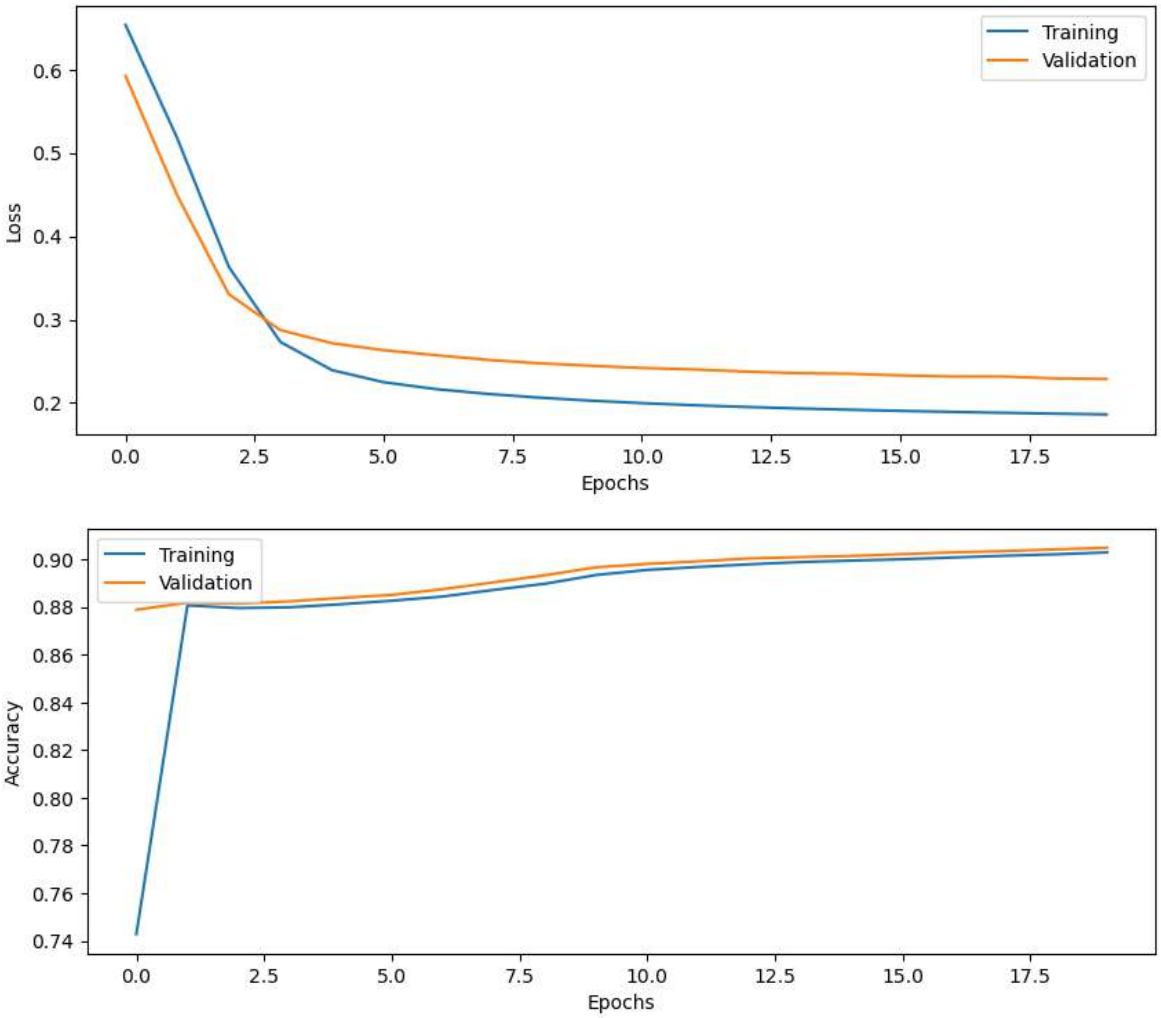
In [ ]:
```
# Setup some training parameters
batch_size = 10000
epochs = 20
input_shape = X.shape

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

history2 = model2.fit(Xtrain, Ytrain, epochs=epochs, batch_size=batch_size,valic
```

```
Epoch 1/20
54/54 [==============================] - 0s 5ms/step - loss: 0.6534 - accuracy:
0.7428 - val_loss: 0.5922 - val_accuracy: 0.8789
Epoch 2/20
54/54 [==============================] - 0s 3ms/step - loss: 0.5177 - accuracy:
0.8807 - val_loss: 0.4491 - val_accuracy: 0.8818
Epoch 3/20
54/54 [==============================] - 0s 3ms/step - loss: 0.3633 - accuracy:
0.8796 - val_loss: 0.3305 - val_accuracy: 0.8815
Epoch 4/20
54/54 [==============================] - 0s 3ms/step - loss: 0.2731 - accuracy:
0.8799 - val_loss: 0.2873 - val_accuracy: 0.8824
Epoch 5/20
54/54 [==============================] - 0s 3ms/step - loss: 0.2393 - accuracy:
0.8812 - val_loss: 0.2716 - val_accuracy: 0.8838
Epoch 6/20
54/54 [==============================] - 0s 3ms/step - loss: 0.2248 - accuracy:
0.8826 - val_loss: 0.2632 - val_accuracy: 0.8851
Epoch 7/20
54/54 [==============================] - 0s 3ms/step - loss: 0.2166 - accuracy:
0.8844 - val_loss: 0.2573 - val_accuracy: 0.8875
Epoch 8/20
54/54 [==============================] - 0s 3ms/step - loss: 0.2109 - accuracy:
0.8872 - val_loss: 0.2517 - val_accuracy: 0.8904
Epoch 9/20
54/54 [==============================] - 0s 3ms/step - loss: 0.2064 - accuracy:
0.8898 - val_loss: 0.2476 - val_accuracy: 0.8933
Epoch 10/20
54/54 [==============================] - 0s 3ms/step - loss: 0.2027 - accuracy:
0.8934 - val_loss: 0.2446 - val_accuracy: 0.8967
Epoch 11/20
54/54 [==============================] - 0s 3ms/step - loss: 0.1997 - accuracy:
0.8956 - val_loss: 0.2419 - val_accuracy: 0.8981
Epoch 12/20
54/54 [==============================] - 0s 3ms/step - loss: 0.1972 - accuracy:
0.8968 - val_loss: 0.2402 - val_accuracy: 0.8992
Epoch 13/20
54/54 [==============================] - 0s 3ms/step - loss: 0.1952 - accuracy:
0.8979 - val_loss: 0.2376 - val_accuracy: 0.9004
Epoch 14/20
54/54 [==============================] - 0s 3ms/step - loss: 0.1934 - accuracy:
0.8989 - val_loss: 0.2357 - val_accuracy: 0.9009
Epoch 15/20
54/54 [==============================] - 0s 3ms/step - loss: 0.1918 - accuracy:
0.8995 - val_loss: 0.2350 - val_accuracy: 0.9014
Epoch 16/20
54/54 [==============================] - 0s 3ms/step - loss: 0.1905 - accuracy:
0.9000 - val_loss: 0.2330 - val_accuracy: 0.9022
Epoch 17/20
54/54 [==============================] - 0s 3ms/step - loss: 0.1892 - accuracy:
0.9008 - val_loss: 0.2317 - val_accuracy: 0.9029
Epoch 18/20
54/54 [==============================] - 0s 3ms/step - loss: 0.1881 - accuracy:
0.9015 - val_loss: 0.2317 - val_accuracy: 0.9035
Epoch 19/20
54/54 [==============================] - 0s 3ms/step - loss: 0.1871 - accuracy:
0.9021 - val_loss: 0.2294 - val_accuracy: 0.9042
Epoch 20/20
54/54 [==============================] - 0s 3ms/step - loss: 0.1862 - accuracy:
0.9029 - val_loss: 0.2287 - val_accuracy: 0.9049
```

In [ ]:
```
# Evaluate model on test data
score = model2.evaluate(Xtest,Ytest, batch_size=batch_size)

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

```
12/12 [==============================] - 0s 1ms/step - loss: 0.2322 - accuracy:
0.9032
Test loss: 0.2322
Test accuracy: 0.9032
```

In [ ]:
```
plot_results(history2)
```



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.

- if the dataset is too large(compared with the memories we are using), it cannot be stored in memories if we don't divide it into smaller batches to fit the memory size.

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?

- Number of weight updates per epoch = number of training examples / batch size, so less times of weights updating will be performed when batch size grows

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

- the memoey size.

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?

- when batch size increases, the times of weight updating will reduce for each epoch. Thus we will need a greater learning rate due to fewer weight updates in same epochs.

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()

- there will be 1860+420+21 = 2301 parameters in our 2 dense layer model and 4650 + 3*2550 +51 = 12351

In [ ]:
```
model1.summary()
```

```
Model: "sequential_2"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_5 (Dense)              (None, 20)                1860
_________________________________________________________________
dense_6 (Dense)              (None, 20)                420
_________________________________________________________________
dense_7 (Dense)              (None, 1)                 21
=================================================================
Total params: 2,301
Trainable params: 2,301
Non-trainable params: 0
_________________________________________________________________
```

In [ ]:
```
model14 = build_DNN(input_shape, n_layers=4, n_nodes=50)
model14.summary()
```

```
Model: "sequential_5"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_16 (Dense)             (None, 50)                4650
_________________________________________________________________
dense_17 (Dense)             (None, 50)                2550
_________________________________________________________________
dense_18 (Dense)             (None, 50)                2550
_________________________________________________________________
dense_19 (Dense)             (None, 50)                2550
_________________________________________________________________
dense_20 (Dense)             (None, 1)                 51
=================================================================
Total params: 12,351
Trainable params: 12,351
Non-trainable params: 0
_________________________________________________________________
```

4 layers, 20 nodes, class weights

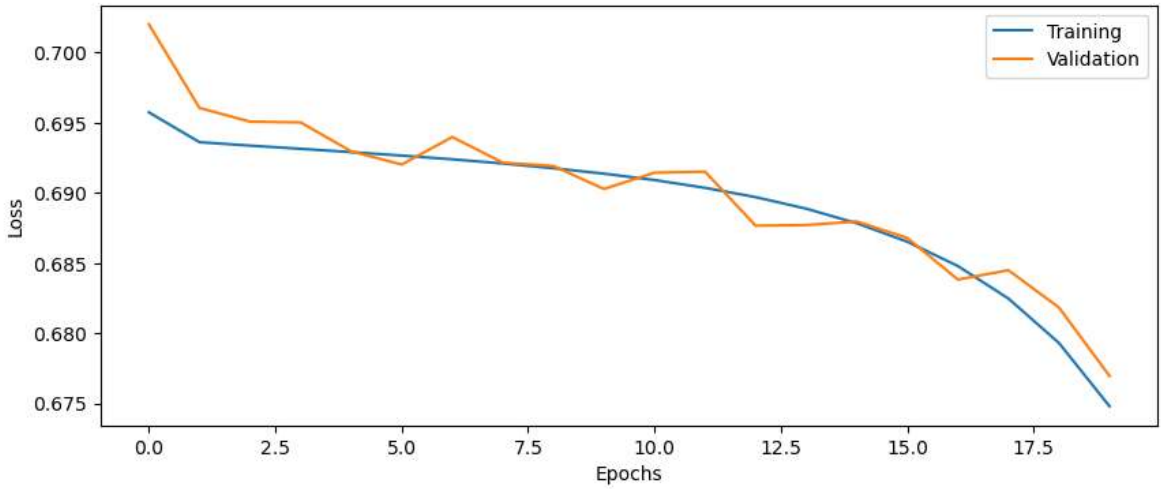
In [ ]:
```
# Setup some training parameters
batch_size = 10000
epochs = 20
input_shape = X.shape

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

history3 = model3.fit(Xtrain, Ytrain, epochs=epochs, batch_size=batch_size,valic
```

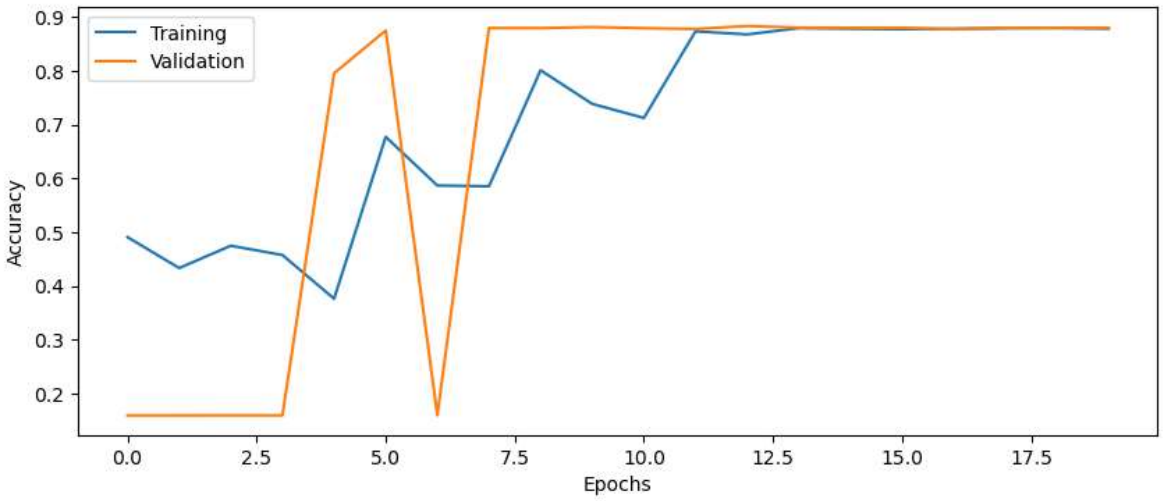
```
Epoch 1/20
54/54 [==============================] - 0s 6ms/step - loss: 0.6957 - accuracy:
0.4908 - val_loss: 0.7020 - val_accuracy: 0.1596
Epoch 2/20
54/54 [==============================] - 0s 4ms/step - loss: 0.6936 - accuracy:
0.4334 - val_loss: 0.6960 - val_accuracy: 0.1596
Epoch 3/20
54/54 [==============================] - 0s 4ms/step - loss: 0.6934 - accuracy:
0.4749 - val_loss: 0.6951 - val_accuracy: 0.1597
Epoch 4/20
54/54 [==============================] - 0s 4ms/step - loss: 0.6931 - accuracy:
0.4577 - val_loss: 0.6950 - val_accuracy: 0.1597
Epoch 5/20
54/54 [==============================] - 0s 4ms/step - loss: 0.6929 - accuracy:
0.3765 - val_loss: 0.6930 - val_accuracy: 0.7953
Epoch 6/20
54/54 [==============================] - 0s 4ms/step - loss: 0.6927 - accuracy:
0.6772 - val_loss: 0.6920 - val_accuracy: 0.8748
Epoch 7/20
54/54 [==============================] - 0s 4ms/step - loss: 0.6924 - accuracy:
0.5868 - val_loss: 0.6940 - val_accuracy: 0.1598
Epoch 8/20
54/54 [==============================] - 0s 4ms/step - loss: 0.6921 - accuracy:
0.5854 - val_loss: 0.6922 - val_accuracy: 0.8794
Epoch 9/20
54/54 [==============================] - 0s 4ms/step - loss: 0.6918 - accuracy:
0.8009 - val_loss: 0.6919 - val_accuracy: 0.8793
Epoch 10/20
54/54 [==============================] - 0s 4ms/step - loss: 0.6914 - accuracy:
0.7384 - val_loss: 0.6903 - val_accuracy: 0.8812
Epoch 11/20
54/54 [==============================] - 0s 4ms/step - loss: 0.6909 - accuracy:
0.7123 - val_loss: 0.6914 - val_accuracy: 0.8791
Epoch 12/20
54/54 [==============================] - 0s 4ms/step - loss: 0.6904 - accuracy:
0.8733 - val_loss: 0.6915 - val_accuracy: 0.8777
Epoch 13/20
54/54 [==============================] - 0s 4ms/step - loss: 0.6897 - accuracy:
0.8675 - val_loss: 0.6877 - val_accuracy: 0.8832
Epoch 14/20
54/54 [==============================] - 0s 4ms/step - loss: 0.6889 - accuracy:
0.8792 - val_loss: 0.6877 - val_accuracy: 0.8801
Epoch 15/20
54/54 [==============================] - 0s 4ms/step - loss: 0.6878 - accuracy:
0.8782 - val_loss: 0.6880 - val_accuracy: 0.8798
Epoch 16/20
54/54 [==============================] - 0s 4ms/step - loss: 0.6865 - accuracy:
0.8773 - val_loss: 0.6868 - val_accuracy: 0.8800
Epoch 17/20
54/54 [==============================] - 0s 4ms/step - loss: 0.6848 - accuracy:
0.8781 - val_loss: 0.6838 - val_accuracy: 0.8773
Epoch 18/20
54/54 [==============================] - 0s 4ms/step - loss: 0.6825 - accuracy:
0.8791 - val_loss: 0.6845 - val_accuracy: 0.8797
Epoch 19/20
54/54 [==============================] - 0s 4ms/step - loss: 0.6793 - accuracy:
0.8796 - val_loss: 0.6818 - val_accuracy: 0.8795
Epoch 20/20
54/54 [==============================] - 0s 4ms/step - loss: 0.6748 - accuracy:
0.8782 - val_loss: 0.6770 - val_accuracy: 0.8799
```

In [ ]:
```
# Evaluate model on test data
score = model3.evaluate(Xtest,Ytest, batch_size=batch_size)

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

```
12/12 [==============================] - 0s 1ms/step - loss: 0.6771 - accuracy:
0.8778
Test loss: 0.6771
Test accuracy: 0.8778
```

In [ ]:
```
plot_results(history3)
```





2 layers, 50 nodes, class weights

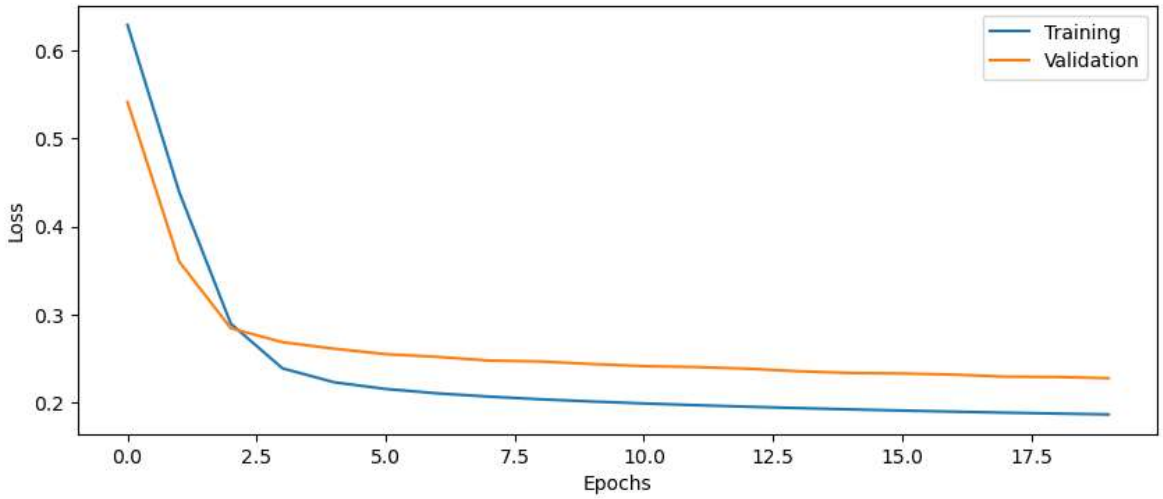
In [ ]:
```
# Setup some training parameters
batch_size = 10000
epochs = 20
input_shape = X.shape

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

history4 = model4.fit(Xtrain, Ytrain, epochs=epochs, batch_size=batch_size,valic
```

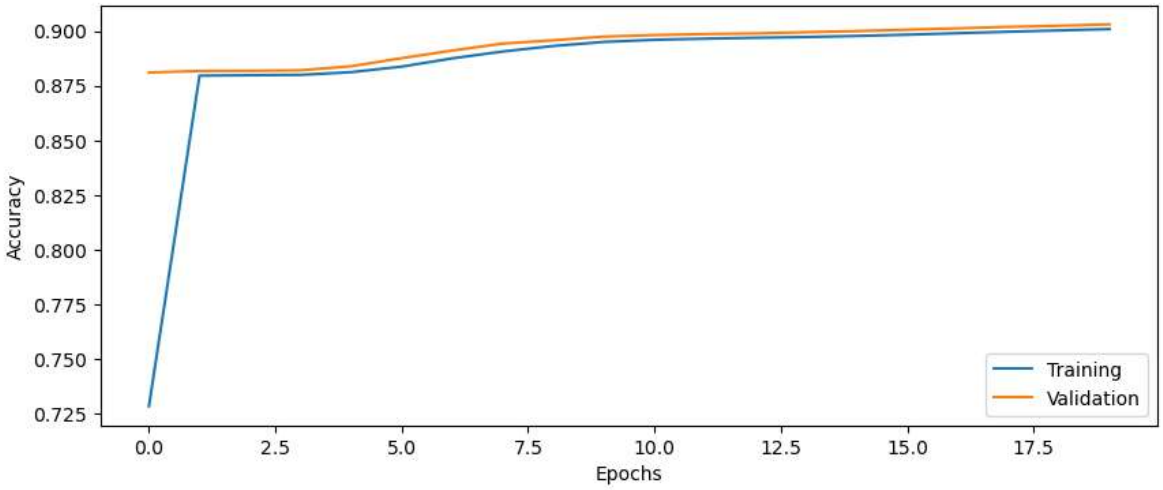
```
Epoch 1/20
54/54 [==============================] - 0s 7ms/step - loss: 0.6293 - accuracy:
0.7286 - val_loss: 0.5416 - val_accuracy: 0.8810
Epoch 2/20
54/54 [==============================] - 0s 6ms/step - loss: 0.4393 - accuracy:
0.8796 - val_loss: 0.3600 - val_accuracy: 0.8817
Epoch 3/20
54/54 [==============================] - 0s 6ms/step - loss: 0.2894 - accuracy:
0.8798 - val_loss: 0.2842 - val_accuracy: 0.8818
Epoch 4/20
54/54 [==============================] - 0s 6ms/step - loss: 0.2386 - accuracy:
0.8800 - val_loss: 0.2684 - val_accuracy: 0.8820
Epoch 5/20
54/54 [==============================] - 0s 6ms/step - loss: 0.2227 - accuracy:
0.8812 - val_loss: 0.2609 - val_accuracy: 0.8839
Epoch 6/20
54/54 [==============================] - 0s 6ms/step - loss: 0.2151 - accuracy:
0.8837 - val_loss: 0.2547 - val_accuracy: 0.8876
Epoch 7/20
54/54 [==============================] - 0s 6ms/step - loss: 0.2101 - accuracy:
0.8875 - val_loss: 0.2516 - val_accuracy: 0.8911
Epoch 8/20
54/54 [==============================] - 0s 6ms/step - loss: 0.2064 - accuracy:
0.8906 - val_loss: 0.2474 - val_accuracy: 0.8942
Epoch 9/20
54/54 [==============================] - 0s 6ms/step - loss: 0.2034 - accuracy:
0.8931 - val_loss: 0.2464 - val_accuracy: 0.8957
Epoch 10/20
54/54 [==============================] - 0s 6ms/step - loss: 0.2008 - accuracy:
0.8950 - val_loss: 0.2434 - val_accuracy: 0.8974
Epoch 11/20
54/54 [==============================] - 0s 6ms/step - loss: 0.1986 - accuracy:
0.8960 - val_loss: 0.2411 - val_accuracy: 0.8981
Epoch 12/20
54/54 [==============================] - 0s 6ms/step - loss: 0.1967 - accuracy:
0.8965 - val_loss: 0.2401 - val_accuracy: 0.8986
Epoch 13/20
54/54 [==============================] - 0s 6ms/step - loss: 0.1949 - accuracy:
0.8969 - val_loss: 0.2382 - val_accuracy: 0.8989
Epoch 14/20
54/54 [==============================] - 0s 6ms/step - loss: 0.1933 - accuracy:
0.8972 - val_loss: 0.2352 - val_accuracy: 0.8994
Epoch 15/20
54/54 [==============================] - 0s 6ms/step - loss: 0.1919 - accuracy:
0.8977 - val_loss: 0.2333 - val_accuracy: 0.8999
Epoch 16/20
54/54 [==============================] - 0s 6ms/step - loss: 0.1905 - accuracy:
0.8983 - val_loss: 0.2327 - val_accuracy: 0.9006
Epoch 17/20
54/54 [==============================] - 0s 6ms/step - loss: 0.1893 - accuracy:
0.8990 - val_loss: 0.2315 - val_accuracy: 0.9012
Epoch 18/20
54/54 [==============================] - 0s 6ms/step - loss: 0.1881 - accuracy:
0.8996 - val_loss: 0.2291 - val_accuracy: 0.9019
Epoch 19/20
54/54 [==============================] - 0s 6ms/step - loss: 0.1871 - accuracy:
0.9002 - val_loss: 0.2287 - val_accuracy: 0.9024
Epoch 20/20
54/54 [==============================] - 0s 6ms/step - loss: 0.1861 - accuracy:
0.9009 - val_loss: 0.2274 - val_accuracy: 0.9030
```

In [ ]:
```
# Evaluate model on test data
score = model4.evaluate(Xtest,Ytest, batch_size=batch_size)

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

```
12/12 [==============================] - 0s 2ms/step - loss: 0.2310 - accuracy:
0.9011
Test loss: 0.2310
Test accuracy: 0.9011
```

In [ ]:
```
plot_results(history4)
```





4 layers, 50 nodes, class weights

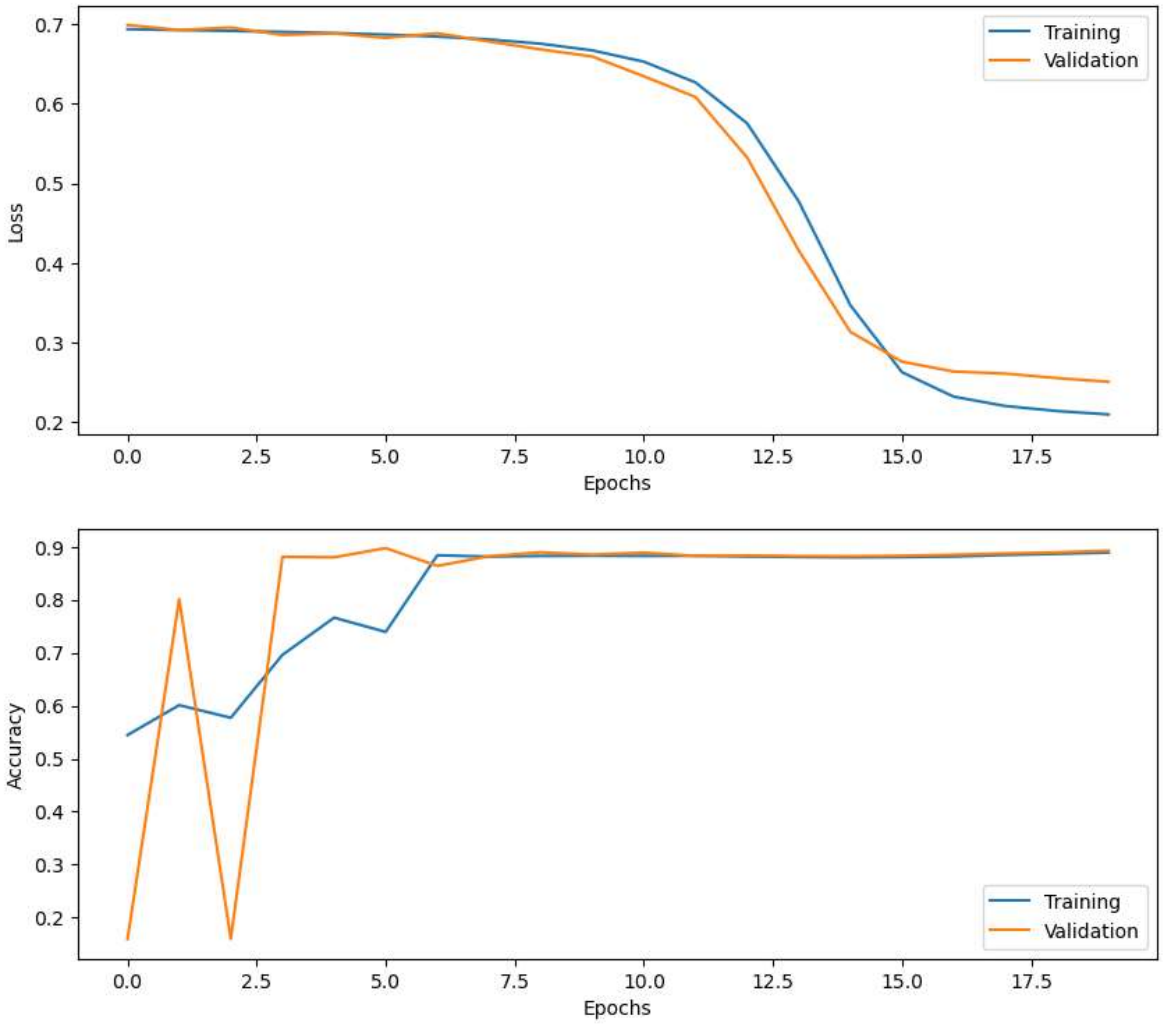
In [ ]:
```
# Setup some training parameters
batch_size = 10000
epochs = 20
input_shape = X.shape

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

history5 = model5.fit(Xtrain, Ytrain, epochs=epochs, batch_size=batch_size,valic
```

```
Epoch 1/20
54/54 [==============================] - 1s 12ms/step - loss: 0.6935 - accuracy:
0.5446 - val_loss: 0.6988 - val_accuracy: 0.1596
Epoch 2/20
54/54 [==============================] - 1s 9ms/step - loss: 0.6925 - accuracy:
0.6011 - val_loss: 0.6926 - val_accuracy: 0.8012
Epoch 3/20
54/54 [==============================] - 1s 10ms/step - loss: 0.6914 - accuracy:
0.5773 - val_loss: 0.6957 - val_accuracy: 0.1600
Epoch 4/20
54/54 [==============================] - 1s 10ms/step - loss: 0.6902 - accuracy:
0.6960 - val_loss: 0.6863 - val_accuracy: 0.8812
Epoch 5/20
54/54 [==============================] - 1s 9ms/step - loss: 0.6887 - accuracy:
0.7662 - val_loss: 0.6882 - val_accuracy: 0.8801
Epoch 6/20
54/54 [==============================] - 1s 10ms/step - loss: 0.6868 - accuracy:
0.7394 - val_loss: 0.6826 - val_accuracy: 0.8975
Epoch 7/20
54/54 [==============================] - 1s 10ms/step - loss: 0.6842 - accuracy:
0.8840 - val_loss: 0.6885 - val_accuracy: 0.8639
Epoch 8/20
54/54 [==============================] - 1s 10ms/step - loss: 0.6807 - accuracy:
0.8814 - val_loss: 0.6780 - val_accuracy: 0.8821
Epoch 9/20
54/54 [==============================] - 1s 9ms/step - loss: 0.6753 - accuracy:
0.8827 - val_loss: 0.6682 - val_accuracy: 0.8896
Epoch 10/20
54/54 [==============================] - 1s 10ms/step - loss: 0.6669 - accuracy:
0.8831 - val_loss: 0.6593 - val_accuracy: 0.8853
Epoch 11/20
54/54 [==============================] - 1s 10ms/step - loss: 0.6527 - accuracy:
0.8828 - val_loss: 0.6344 - val_accuracy: 0.8890
Epoch 12/20
54/54 [==============================] - 1s 10ms/step - loss: 0.6267 - accuracy:
0.8830 - val_loss: 0.6084 - val_accuracy: 0.8828
Epoch 13/20
54/54 [==============================] - 1s 10ms/step - loss: 0.5753 - accuracy:
0.8817 - val_loss: 0.5324 - val_accuracy: 0.8835
Epoch 14/20
54/54 [==============================] - 1s 10ms/step - loss: 0.4772 - accuracy:
0.8807 - val_loss: 0.4156 - val_accuracy: 0.8824
Epoch 15/20
54/54 [==============================] - 1s 9ms/step - loss: 0.3468 - accuracy:
0.8800 - val_loss: 0.3134 - val_accuracy: 0.8821
Epoch 16/20
54/54 [==============================] - 0s 9ms/step - loss: 0.2627 - accuracy:
0.8805 - val_loss: 0.2760 - val_accuracy: 0.8831
Epoch 17/20
54/54 [==============================] - 0s 9ms/step - loss: 0.2321 - accuracy:
0.8816 - val_loss: 0.2636 - val_accuracy: 0.8850
Epoch 18/20
54/54 [==============================] - 1s 9ms/step - loss: 0.2203 - accuracy:
0.8845 - val_loss: 0.2610 - val_accuracy: 0.8875
Epoch 19/20
54/54 [==============================] - 1s 9ms/step - loss: 0.2141 - accuracy:
0.8867 - val_loss: 0.2553 - val_accuracy: 0.8895
Epoch 20/20
54/54 [==============================] - 0s 9ms/step - loss: 0.2098 - accuracy:
0.8892 - val_loss: 0.2507 - val_accuracy: 0.8926
```

In [ ]:
```
# Evaluate model on test data
score = model5.evaluate(Xtest,Ytest, batch_size=batch_size)

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

```
12/12 [==============================] - 0s 3ms/step - loss: 0.2543 - accuracy:
0.8906
Test loss: 0.2543
Test accuracy: 0.8906
```

In [ ]:
```
plot_results(history5)
```



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?

- Normalizing the input makes optimization easier, since the loss function behaves nicer (more isotropic)

2 layers, 20 nodes, class weights, batch normalization

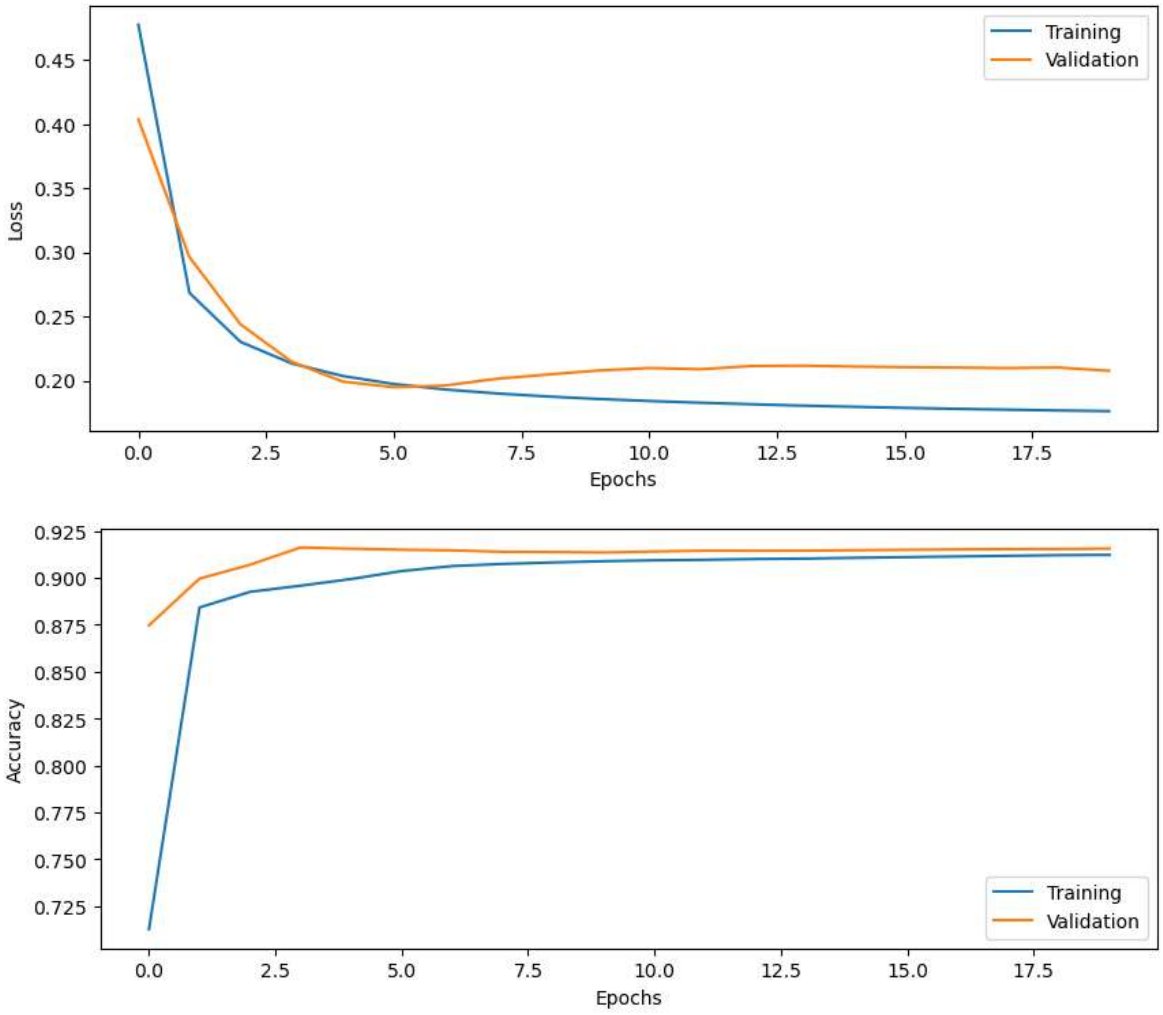
In [ ]:
```
# Setup some training parameters
batch_size = 10000
epochs = 20
input_shape = X.shape

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

history6 = model6.fit(Xtrain, Ytrain, epochs=epochs, batch_size=batch_size,valic
```

```
Epoch 1/20
54/54 [==============================] - 0s 7ms/step - loss: 0.4777 - accuracy:
0.7126 - val_loss: 0.4039 - val_accuracy: 0.8746
Epoch 2/20
54/54 [==============================] - 0s 5ms/step - loss: 0.2684 - accuracy:
0.8842 - val_loss: 0.2963 - val_accuracy: 0.8996
Epoch 3/20
54/54 [==============================] - 0s 5ms/step - loss: 0.2302 - accuracy:
0.8925 - val_loss: 0.2438 - val_accuracy: 0.9070
Epoch 4/20
54/54 [==============================] - 0s 5ms/step - loss: 0.2133 - accuracy:
0.8958 - val_loss: 0.2146 - val_accuracy: 0.9162
Epoch 5/20
54/54 [==============================] - 0s 5ms/step - loss: 0.2035 - accuracy:
0.8994 - val_loss: 0.1991 - val_accuracy: 0.9155
Epoch 6/20
54/54 [==============================] - 0s 5ms/step - loss: 0.1973 - accuracy:
0.9036 - val_loss: 0.1948 - val_accuracy: 0.9150
Epoch 7/20
54/54 [==============================] - 0s 5ms/step - loss: 0.1930 - accuracy:
0.9063 - val_loss: 0.1961 - val_accuracy: 0.9146
Epoch 8/20
54/54 [==============================] - 0s 5ms/step - loss: 0.1899 - accuracy:
0.9074 - val_loss: 0.2013 - val_accuracy: 0.9139
Epoch 9/20
54/54 [==============================] - 0s 5ms/step - loss: 0.1875 - accuracy:
0.9082 - val_loss: 0.2046 - val_accuracy: 0.9138
Epoch 10/20
54/54 [==============================] - 0s 5ms/step - loss: 0.1856 - accuracy:
0.9088 - val_loss: 0.2078 - val_accuracy: 0.9135
Epoch 11/20
54/54 [==============================] - 0s 5ms/step - loss: 0.1840 - accuracy:
0.9093 - val_loss: 0.2097 - val_accuracy: 0.9140
Epoch 12/20
54/54 [==============================] - 0s 5ms/step - loss: 0.1826 - accuracy:
0.9096 - val_loss: 0.2088 - val_accuracy: 0.9145
Epoch 13/20
54/54 [==============================] - 0s 5ms/step - loss: 0.1815 - accuracy:
0.9100 - val_loss: 0.2113 - val_accuracy: 0.9144
Epoch 14/20
54/54 [==============================] - 0s 5ms/step - loss: 0.1804 - accuracy:
0.9102 - val_loss: 0.2116 - val_accuracy: 0.9145
Epoch 15/20
54/54 [==============================] - 0s 5ms/step - loss: 0.1795 - accuracy:
0.9107 - val_loss: 0.2109 - val_accuracy: 0.9147
Epoch 16/20
54/54 [==============================] - 0s 5ms/step - loss: 0.1786 - accuracy:
0.9110 - val_loss: 0.2104 - val_accuracy: 0.9149
Epoch 17/20
54/54 [==============================] - 0s 5ms/step - loss: 0.1779 - accuracy:
0.9114 - val_loss: 0.2101 - val_accuracy: 0.9152
Epoch 18/20
54/54 [==============================] - 0s 5ms/step - loss: 0.1772 - accuracy:
0.9117 - val_loss: 0.2097 - val_accuracy: 0.9153
Epoch 19/20
54/54 [==============================] - 0s 5ms/step - loss: 0.1766 - accuracy:
0.9121 - val_loss: 0.2101 - val_accuracy: 0.9154
Epoch 20/20
54/54 [==============================] - 0s 5ms/step - loss: 0.1761 - accuracy:
0.9122 - val_loss: 0.2075 - val_accuracy: 0.9156
```

In [ ]:
```
# Evaluate model on test data
score = model6.evaluate(Xtest,Ytest, batch_size=batch_size)

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

```
12/12 [==============================] - 0s 2ms/step - loss: 0.2108 - accuracy:
0.9138
Test loss: 0.2108
Test accuracy: 0.9138
```

In [ ]:
```
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

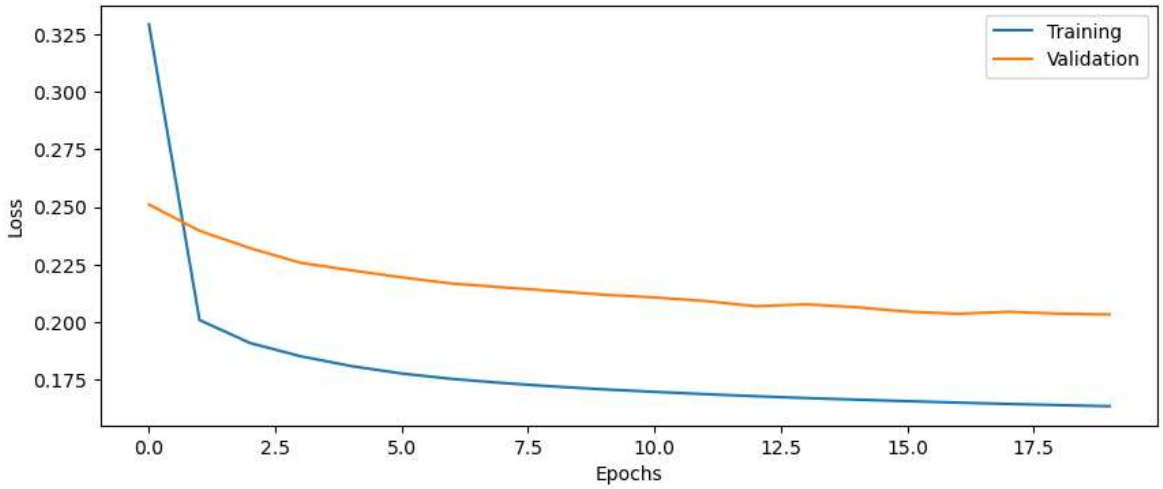
In [ ]:
```
# Setup some training parameters
batch_size = 10000
epochs = 20
input_shape = X.shape

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

history7 = model7.fit(Xtrain, Ytrain, epochs=epochs, batch_size=batch_size,valic
```

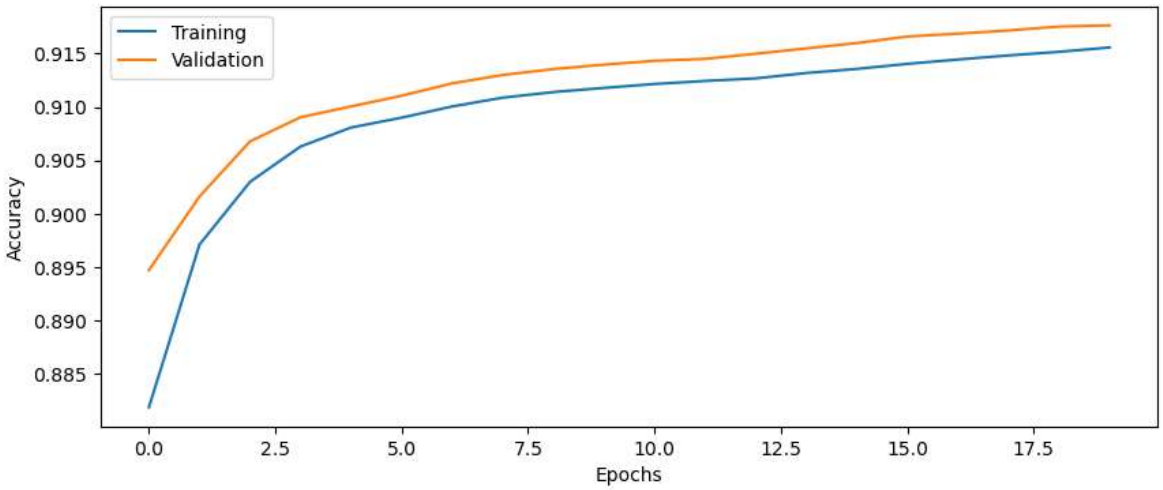
```
Epoch 1/20
54/54 [==============================] - 0s 5ms/step - loss: 0.3294 - accuracy:
0.8818 - val_loss: 0.2511 - val_accuracy: 0.8947
Epoch 2/20
54/54 [==============================] - 0s 3ms/step - loss: 0.2008 - accuracy:
0.8971 - val_loss: 0.2396 - val_accuracy: 0.9016
Epoch 3/20
54/54 [==============================] - 0s 3ms/step - loss: 0.1908 - accuracy:
0.9030 - val_loss: 0.2321 - val_accuracy: 0.9068
Epoch 4/20
54/54 [==============================] - 0s 3ms/step - loss: 0.1851 - accuracy:
0.9063 - val_loss: 0.2257 - val_accuracy: 0.9090
Epoch 5/20
54/54 [==============================] - 0s 3ms/step - loss: 0.1808 - accuracy:
0.9081 - val_loss: 0.2224 - val_accuracy: 0.9100
Epoch 6/20
54/54 [==============================] - 0s 3ms/step - loss: 0.1776 - accuracy:
0.9090 - val_loss: 0.2194 - val_accuracy: 0.9110
Epoch 7/20
54/54 [==============================] - 0s 3ms/step - loss: 0.1752 - accuracy:
0.9100 - val_loss: 0.2167 - val_accuracy: 0.9122
Epoch 8/20
54/54 [==============================] - 0s 3ms/step - loss: 0.1734 - accuracy:
0.9109 - val_loss: 0.2150 - val_accuracy: 0.9130
Epoch 9/20
54/54 [==============================] - 0s 3ms/step - loss: 0.1719 - accuracy:
0.9114 - val_loss: 0.2135 - val_accuracy: 0.9135
Epoch 10/20
54/54 [==============================] - 0s 3ms/step - loss: 0.1706 - accuracy:
0.9118 - val_loss: 0.2118 - val_accuracy: 0.9140
Epoch 11/20
54/54 [==============================] - 0s 3ms/step - loss: 0.1696 - accuracy:
0.9121 - val_loss: 0.2106 - val_accuracy: 0.9143
Epoch 12/20
54/54 [==============================] - 0s 3ms/step - loss: 0.1686 - accuracy:
0.9124 - val_loss: 0.2091 - val_accuracy: 0.9145
Epoch 13/20
54/54 [==============================] - 0s 3ms/step - loss: 0.1677 - accuracy:
0.9127 - val_loss: 0.2068 - val_accuracy: 0.9150
Epoch 14/20
54/54 [==============================] - 0s 3ms/step - loss: 0.1669 - accuracy:
0.9132 - val_loss: 0.2076 - val_accuracy: 0.9155
Epoch 15/20
54/54 [==============================] - 0s 3ms/step - loss: 0.1662 - accuracy:
0.9136 - val_loss: 0.2064 - val_accuracy: 0.9160
Epoch 16/20
54/54 [==============================] - 0s 3ms/step - loss: 0.1655 - accuracy:
0.9140 - val_loss: 0.2045 - val_accuracy: 0.9166
Epoch 17/20
54/54 [==============================] - 0s 3ms/step - loss: 0.1649 - accuracy:
0.9144 - val_loss: 0.2035 - val_accuracy: 0.9169
Epoch 18/20
54/54 [==============================] - 0s 3ms/step - loss: 0.1643 - accuracy:
0.9148 - val_loss: 0.2044 - val_accuracy: 0.9172
Epoch 19/20
54/54 [==============================] - 0s 3ms/step - loss: 0.1638 - accuracy:
0.9152 - val_loss: 0.2036 - val_accuracy: 0.9175
Epoch 20/20
54/54 [==============================] - 0s 3ms/step - loss: 0.1633 - accuracy:
0.9156 - val_loss: 0.2032 - val_accuracy: 0.9176
```

In [ ]:
```
# Evaluate model on test data
score = model7.evaluate(Xtest,Ytest, batch_size=batch_size)

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

```
12/12 [==============================] - 0s 2ms/step - loss: 0.2071 - accuracy:
0.9161
Test loss: 0.2071
Test accuracy: 0.9161
```

In [ ]:
```
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

In [ ]:
```
# Setup some training parameters
batch_size = 10000
```

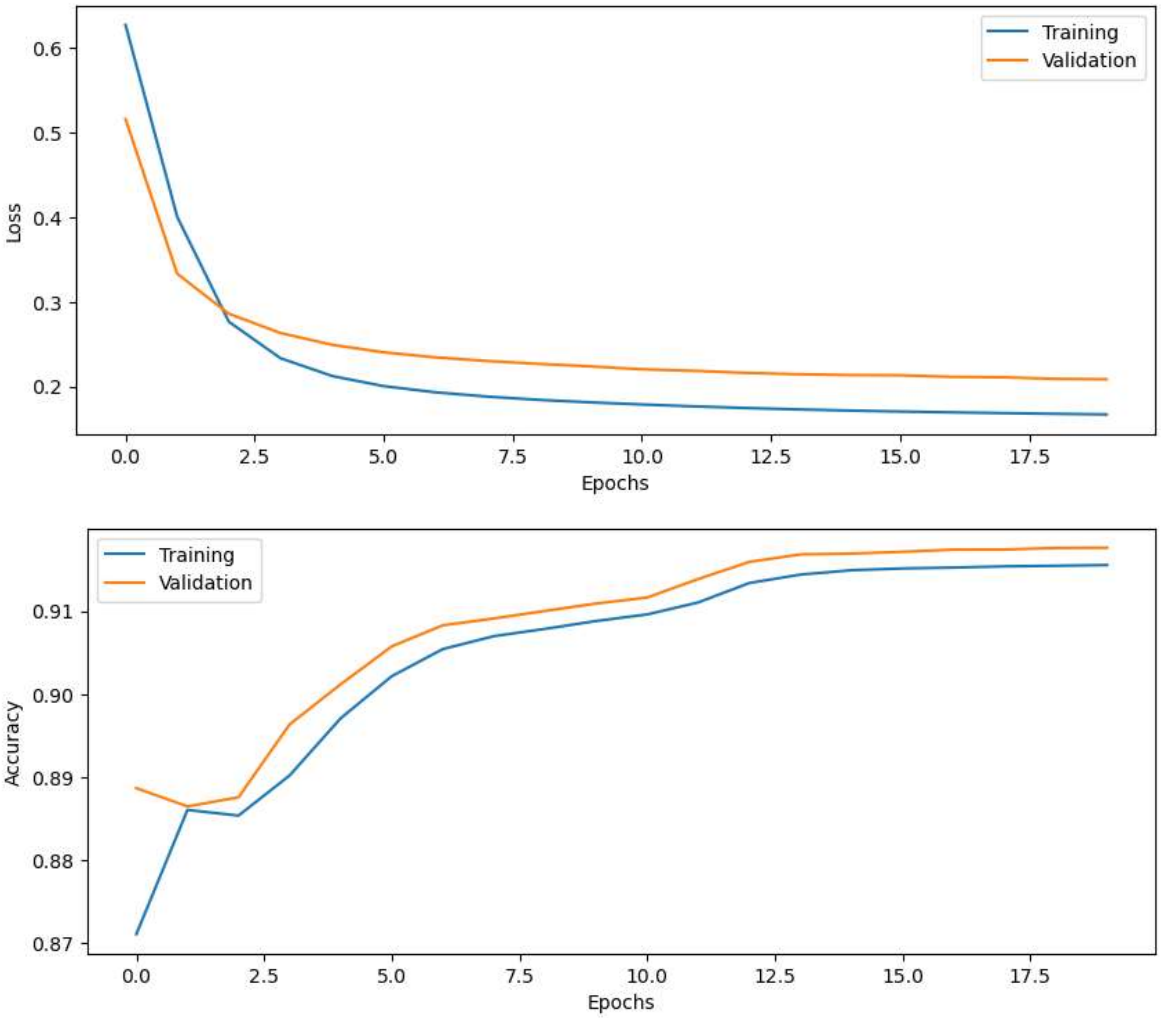
```
epochs = 20
input_shape = X.shape

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

history8 = model8.fit(Xtrain, Ytrain, epochs=epochs, batch_size=batch_size,valic
```

```
Epoch 1/20
54/54 [==============================] - 0s 5ms/step - loss: 0.6267 - accuracy:
0.8711 - val_loss: 0.5156 - val_accuracy: 0.8887
Epoch 2/20
54/54 [==============================] - 0s 3ms/step - loss: 0.4004 - accuracy:
0.8861 - val_loss: 0.3333 - val_accuracy: 0.8865
Epoch 3/20
54/54 [==============================] - 0s 3ms/step - loss: 0.2765 - accuracy:
0.8854 - val_loss: 0.2859 - val_accuracy: 0.8876
Epoch 4/20
54/54 [==============================] - 0s 3ms/step - loss: 0.2335 - accuracy:
0.8902 - val_loss: 0.2632 - val_accuracy: 0.8964
Epoch 5/20
54/54 [==============================] - 0s 3ms/step - loss: 0.2125 - accuracy:
0.8971 - val_loss: 0.2493 - val_accuracy: 0.9012
Epoch 6/20
54/54 [==============================] - 0s 3ms/step - loss: 0.2004 - accuracy:
0.9022 - val_loss: 0.2404 - val_accuracy: 0.9058
Epoch 7/20
54/54 [==============================] - 0s 3ms/step - loss: 0.1932 - accuracy:
0.9054 - val_loss: 0.2344 - val_accuracy: 0.9083
Epoch 8/20
54/54 [==============================] - 0s 3ms/step - loss: 0.1881 - accuracy:
0.9070 - val_loss: 0.2302 - val_accuracy: 0.9091
Epoch 9/20
54/54 [==============================] - 0s 3ms/step - loss: 0.1843 - accuracy:
0.9079 - val_loss: 0.2267 - val_accuracy: 0.9100
Epoch 10/20
54/54 [==============================] - 0s 3ms/step - loss: 0.1813 - accuracy:
0.9088 - val_loss: 0.2238 - val_accuracy: 0.9109
Epoch 11/20
54/54 [==============================] - 0s 3ms/step - loss: 0.1788 - accuracy:
0.9096 - val_loss: 0.2204 - val_accuracy: 0.9116
Epoch 12/20
54/54 [==============================] - 0s 3ms/step - loss: 0.1766 - accuracy:
0.9110 - val_loss: 0.2186 - val_accuracy: 0.9139
Epoch 13/20
54/54 [==============================] - 0s 4ms/step - loss: 0.1747 - accuracy:
0.9134 - val_loss: 0.2161 - val_accuracy: 0.9159
Epoch 14/20
54/54 [==============================] - 0s 3ms/step - loss: 0.1730 - accuracy:
0.9144 - val_loss: 0.2145 - val_accuracy: 0.9168
Epoch 15/20
54/54 [==============================] - 0s 3ms/step - loss: 0.1717 - accuracy:
0.9149 - val_loss: 0.2136 - val_accuracy: 0.9169
Epoch 16/20
54/54 [==============================] - 0s 3ms/step - loss: 0.1705 - accuracy:
0.9151 - val_loss: 0.2133 - val_accuracy: 0.9171
Epoch 17/20
54/54 [==============================] - 0s 3ms/step - loss: 0.1695 - accuracy:
0.9152 - val_loss: 0.2115 - val_accuracy: 0.9174
Epoch 18/20
54/54 [==============================] - 0s 3ms/step - loss: 0.1686 - accuracy:
0.9154 - val_loss: 0.2110 - val_accuracy: 0.9174
Epoch 19/20
54/54 [==============================] - 0s 3ms/step - loss: 0.1678 - accuracy:
0.9155 - val_loss: 0.2091 - val_accuracy: 0.9176
Epoch 20/20
54/54 [==============================] - 0s 3ms/step - loss: 0.1671 - accuracy:
0.9155 - val_loss: 0.2086 - val_accuracy: 0.9176
```

In [ ]:
```
# Evaluate model on test data
score = model8.evaluate(Xtest,Ytest, batch_size=batch_size)

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

```
12/12 [==============================] - 0s 1ms/step - loss: 0.2118 - accuracy:
0.9161
Test loss: 0.2118
Test accuracy: 0.9161
```

In [ ]:
```
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?

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

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

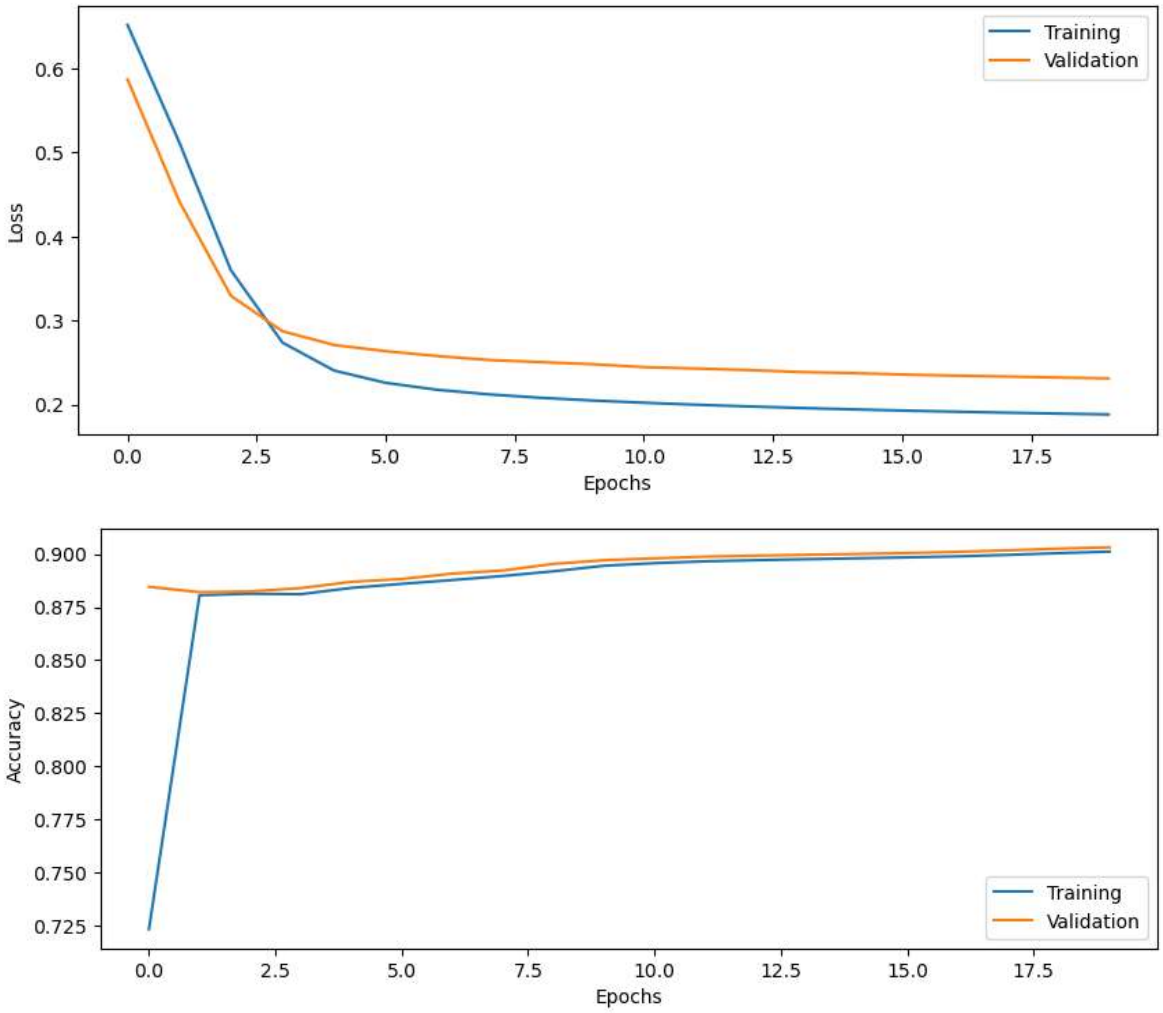
In [ ]:
```
# Setup some training parameters
batch_size = 10000
epochs = 20
input_shape = X.shape

# Build and train model
model9 = build_DNN(input_shape, n_layers=2, n_nodes=20 ,use_dropout=True)

history9 = model9.fit(Xtrain, Ytrain, epochs=epochs, batch_size=batch_size,valic
```

```
Epoch 1/20
54/54 [==============================] - 0s 5ms/step - loss: 0.6516 - accuracy:
0.7233 - val_loss: 0.5869 - val_accuracy: 0.8845
Epoch 2/20
54/54 [==============================] - 0s 3ms/step - loss: 0.5123 - accuracy:
0.8806 - val_loss: 0.4420 - val_accuracy: 0.8820
Epoch 3/20
54/54 [==============================] - 0s 3ms/step - loss: 0.3599 - accuracy:
0.8813 - val_loss: 0.3294 - val_accuracy: 0.8824
Epoch 4/20
54/54 [==============================] - 0s 3ms/step - loss: 0.2736 - accuracy:
0.8811 - val_loss: 0.2872 - val_accuracy: 0.8839
Epoch 5/20
54/54 [==============================] - 0s 3ms/step - loss: 0.2404 - accuracy:
0.8840 - val_loss: 0.2708 - val_accuracy: 0.8869
Epoch 6/20
54/54 [==============================] - 0s 3ms/step - loss: 0.2257 - accuracy:
0.8859 - val_loss: 0.2634 - val_accuracy: 0.8882
Epoch 7/20
54/54 [==============================] - 0s 3ms/step - loss: 0.2175 - accuracy:
0.8877 - val_loss: 0.2576 - val_accuracy: 0.8908
Epoch 8/20
54/54 [==============================] - 0s 3ms/step - loss: 0.2120 - accuracy:
0.8896 - val_loss: 0.2529 - val_accuracy: 0.8922
Epoch 9/20
54/54 [==============================] - 0s 3ms/step - loss: 0.2079 - accuracy:
0.8918 - val_loss: 0.2504 - val_accuracy: 0.8953
Epoch 10/20
54/54 [==============================] - 0s 3ms/step - loss: 0.2047 - accuracy:
0.8945 - val_loss: 0.2479 - val_accuracy: 0.8971
Epoch 11/20
54/54 [==============================] - 0s 3ms/step - loss: 0.2020 - accuracy:
0.8957 - val_loss: 0.2445 - val_accuracy: 0.8979
Epoch 12/20
54/54 [==============================] - 0s 3ms/step - loss: 0.1997 - accuracy:
0.8965 - val_loss: 0.2428 - val_accuracy: 0.8987
Epoch 13/20
54/54 [==============================] - 0s 3ms/step - loss: 0.1977 - accuracy:
0.8971 - val_loss: 0.2410 - val_accuracy: 0.8992
Epoch 14/20
54/54 [==============================] - 0s 3ms/step - loss: 0.1959 - accuracy:
0.8975 - val_loss: 0.2386 - val_accuracy: 0.8996
Epoch 15/20
54/54 [==============================] - 0s 3ms/step - loss: 0.1942 - accuracy:
0.8980 - val_loss: 0.2374 - val_accuracy: 0.9000
Epoch 16/20
54/54 [==============================] - 0s 3ms/step - loss: 0.1928 - accuracy:
0.8984 - val_loss: 0.2356 - val_accuracy: 0.9004
Epoch 17/20
54/54 [==============================] - 0s 3ms/step - loss: 0.1914 - accuracy:
0.8989 - val_loss: 0.2343 - val_accuracy: 0.9010
Epoch 18/20
54/54 [==============================] - 0s 3ms/step - loss: 0.1902 - accuracy:
0.8996 - val_loss: 0.2333 - val_accuracy: 0.9017
Epoch 19/20
54/54 [==============================] - 0s 3ms/step - loss: 0.1891 - accuracy:
0.9004 - val_loss: 0.2321 - val_accuracy: 0.9025
Epoch 20/20
54/54 [==============================] - 0s 3ms/step - loss: 0.1881 - accuracy:
0.9011 - val_loss: 0.2311 - val_accuracy: 0.9031
```

In [ ]:
```
# Evaluate model on test data
score = model9.evaluate(Xtest,Ytest, batch_size=batch_size)

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

```
12/12 [==============================] - 0s 1ms/step - loss: 0.2347 - accuracy:
0.9016
Test loss: 0.2347
Test accuracy: 0.9016
```

In [ ]:
```
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?

- Test accuracy: 0.9426. We added more epochs(40) and layers(10) with nodes(80) for a better performance, and use a 0.7 dropout rate to mitigate overfitting.

In [ ]:
```
# Find your best configuration for the DNN

batch_size = 10000
epochs = 50
input_shape = X.shape

# Build and train DNN
model10 = build_DNN(input_shape, n_layers=3, n_nodes=80 ,use_dropout=0.7, use_br

history10 = model10.fit(Xtrain, Ytrain, epochs=epochs, batch_size=batch_size,val
```

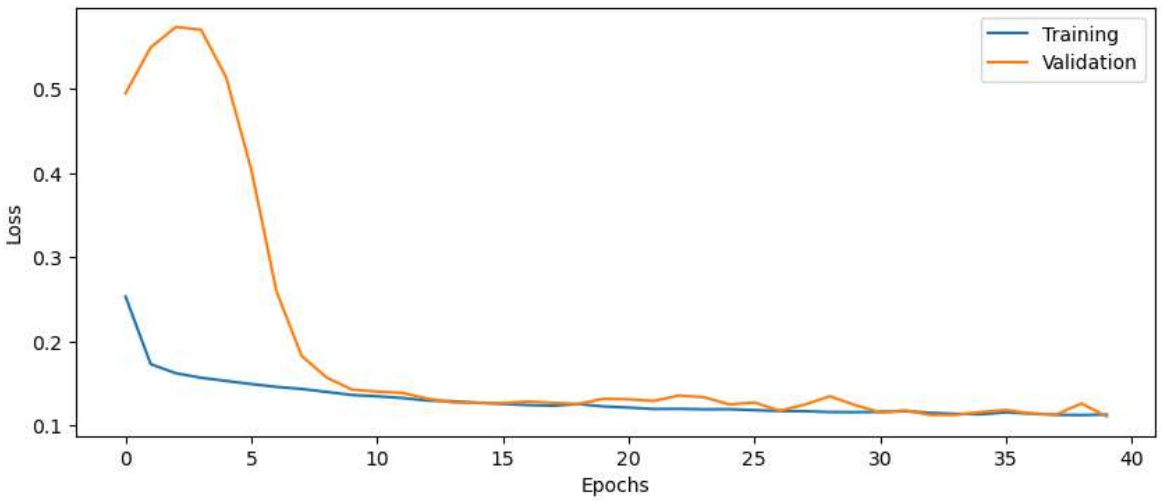
```
Epoch 1/50
54/54 [==============================] - 2s 38ms/step - loss: 0.4682 - accuracy:
0.8220 - val_loss: 0.4441 - val_accuracy: 0.8404
Epoch 2/50
54/54 [==============================] - 2s 37ms/step - loss: 0.2752 - accuracy:
0.8783 - val_loss: 0.4319 - val_accuracy: 0.8404
Epoch 3/50
54/54 [==============================] - 2s 37ms/step - loss: 0.2453 - accuracy:
0.8823 - val_loss: 0.3871 - val_accuracy: 0.8404
Epoch 4/50
54/54 [==============================] - 2s 37ms/step - loss: 0.2298 - accuracy:
0.8853 - val_loss: 0.3207 - val_accuracy: 0.8404
Epoch 5/50
54/54 [==============================] - 2s 37ms/step - loss: 0.2203 - accuracy:
0.8880 - val_loss: 0.2543 - val_accuracy: 0.8410
Epoch 6/50
54/54 [==============================] - 2s 37ms/step - loss: 0.2137 - accuracy:
0.8897 - val_loss: 0.2095 - val_accuracy: 0.8798
Epoch 7/50
54/54 [==============================] - 2s 37ms/step - loss: 0.2086 - accuracy:
0.8913 - val_loss: 0.1901 - val_accuracy: 0.9094
Epoch 8/50
54/54 [==============================] - 2s 37ms/step - loss: 0.2050 - accuracy:
0.8926 - val_loss: 0.1864 - val_accuracy: 0.9127
Epoch 9/50
54/54 [==============================] - 2s 37ms/step - loss: 0.2012 - accuracy:
0.8931 - val_loss: 0.1881 - val_accuracy: 0.9112
Epoch 10/50
54/54 [==============================] - 2s 37ms/step - loss: 0.1989 - accuracy:
0.8939 - val_loss: 0.1883 - val_accuracy: 0.9114
Epoch 11/50
54/54 [==============================] - 2s 37ms/step - loss: 0.1962 - accuracy:
0.8954 - val_loss: 0.1880 - val_accuracy: 0.9118
Epoch 12/50
54/54 [==============================] - 2s 37ms/step - loss: 0.1940 - accuracy:
0.8959 - val_loss: 0.1881 - val_accuracy: 0.9122
Epoch 13/50
54/54 [==============================] - 2s 37ms/step - loss: 0.1918 - accuracy:
0.8970 - val_loss: 0.1872 - val_accuracy: 0.9125
Epoch 14/50
54/54 [==============================] - 2s 37ms/step - loss: 0.1909 - accuracy:
0.8968 - val_loss: 0.1862 - val_accuracy: 0.9130
Epoch 15/50
54/54 [==============================] - 2s 37ms/step - loss: 0.1893 - accuracy:
0.8977 - val_loss: 0.1868 - val_accuracy: 0.9131
Epoch 16/50
54/54 [==============================] - 2s 37ms/step - loss: 0.1886 - accuracy:
0.8979 - val_loss: 0.1848 - val_accuracy: 0.9135
Epoch 17/50
54/54 [==============================] - 2s 37ms/step - loss: 0.1869 - accuracy:
0.8985 - val_loss: 0.1839 - val_accuracy: 0.9138
Epoch 18/50
54/54 [==============================] - 2s 37ms/step - loss: 0.1860 - accuracy:
0.8986 - val_loss: 0.1805 - val_accuracy: 0.9142
Epoch 19/50
54/54 [==============================] - 2s 37ms/step - loss: 0.1848 - accuracy:
0.8993 - val_loss: 0.1795 - val_accuracy: 0.9145
Epoch 20/50
54/54 [==============================] - 2s 37ms/step - loss: 0.1845 - accuracy:
0.8996 - val_loss: 0.1794 - val_accuracy: 0.9146
```

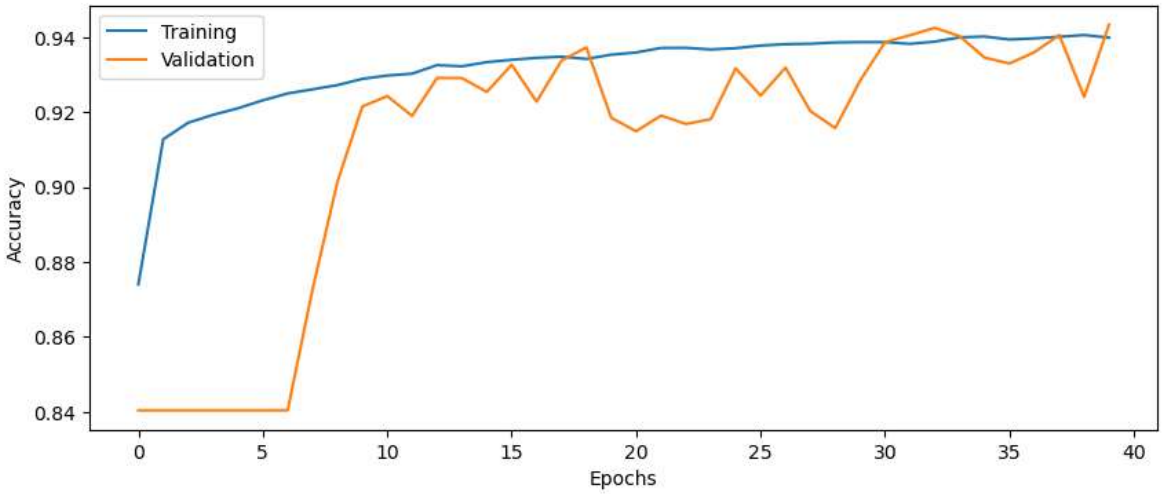
```
Epoch 21/50
54/54 [==============================] - 2s 37ms/step - loss: 0.1835 - accuracy:
0.8996 - val_loss: 0.1793 - val_accuracy: 0.9147
Epoch 22/50
54/54 [==============================] - 2s 37ms/step - loss: 0.1821 - accuracy:
0.9010 - val_loss: 0.1757 - val_accuracy: 0.9152
Epoch 23/50
54/54 [==============================] - 2s 37ms/step - loss: 0.1820 - accuracy:
0.9002 - val_loss: 0.1756 - val_accuracy: 0.9153
Epoch 24/50
54/54 [==============================] - 2s 37ms/step - loss: 0.1811 - accuracy:
0.9009 - val_loss: 0.1755 - val_accuracy: 0.9155
Epoch 25/50
54/54 [==============================] - 2s 37ms/step - loss: 0.1807 - accuracy:
0.9008 - val_loss: 0.1763 - val_accuracy: 0.9155
Epoch 26/50
54/54 [==============================] - 2s 38ms/step - loss: 0.1800 - accuracy:
0.9012 - val_loss: 0.1730 - val_accuracy: 0.9158
Epoch 27/50
54/54 [==============================] - 2s 37ms/step - loss: 0.1795 - accuracy:
0.9012 - val_loss: 0.1727 - val_accuracy: 0.9158
Epoch 28/50
54/54 [==============================] - 2s 37ms/step - loss: 0.1791 - accuracy:
0.9016 - val_loss: 0.1720 - val_accuracy: 0.9159
Epoch 29/50
54/54 [==============================] - 2s 38ms/step - loss: 0.1790 - accuracy:
0.9016 - val_loss: 0.1723 - val_accuracy: 0.9157
Epoch 30/50
54/54 [==============================] - 2s 37ms/step - loss: 0.1778 - accuracy:
0.9025 - val_loss: 0.1720 - val_accuracy: 0.9157
Epoch 31/50
54/54 [==============================] - 2s 38ms/step - loss: 0.1775 - accuracy:
0.9025 - val_loss: 0.1699 - val_accuracy: 0.9159
Epoch 32/50
54/54 [==============================] - 2s 37ms/step - loss: 0.1768 - accuracy:
0.9029 - val_loss: 0.1719 - val_accuracy: 0.9159
Epoch 33/50
54/54 [==============================] - 2s 37ms/step - loss: 0.1765 - accuracy:
0.9028 - val_loss: 0.1710 - val_accuracy: 0.9159
Epoch 34/50
54/54 [==============================] - 2s 37ms/step - loss: 0.1760 - accuracy:
0.9032 - val_loss: 0.1711 - val_accuracy: 0.9160
Epoch 35/50
54/54 [==============================] - 2s 37ms/step - loss: 0.1754 - accuracy:
0.9036 - val_loss: 0.1692 - val_accuracy: 0.9162
Epoch 36/50
54/54 [==============================] - 2s 37ms/step - loss: 0.1751 - accuracy:
0.9033 - val_loss: 0.1682 - val_accuracy: 0.9163
Epoch 37/50
54/54 [==============================] - 2s 37ms/step - loss: 0.1753 - accuracy:
0.9031 - val_loss: 0.1685 - val_accuracy: 0.9163
Epoch 38/50
54/54 [==============================] - 2s 38ms/step - loss: 0.1746 - accuracy:
0.9041 - val_loss: 0.1685 - val_accuracy: 0.9164
Epoch 39/50
54/54 [==============================] - 2s 37ms/step - loss: 0.1741 - accuracy:
0.9043 - val_loss: 0.1693 - val_accuracy: 0.9165
Epoch 40/50
54/54 [==============================] - 2s 37ms/step - loss: 0.1737 - accuracy:
0.9045 - val_loss: 0.1688 - val_accuracy: 0.9165
```

```
Epoch 41/50
54/54 [==============================] - 2s 37ms/step - loss: 0.1740 - accuracy:
0.9041 - val_loss: 0.1660 - val_accuracy: 0.9166
Epoch 42/50
54/54 [==============================] - 2s 37ms/step - loss: 0.1732 - accuracy:
0.9045 - val_loss: 0.1668 - val_accuracy: 0.9166
Epoch 43/50
54/54 [==============================] - 2s 37ms/step - loss: 0.1730 - accuracy:
0.9046 - val_loss: 0.1650 - val_accuracy: 0.9167
Epoch 44/50
54/54 [==============================] - 2s 37ms/step - loss: 0.1731 - accuracy:
0.9047 - val_loss: 0.1667 - val_accuracy: 0.9167
Epoch 45/50
54/54 [==============================] - 2s 37ms/step - loss: 0.1727 - accuracy:
0.9048 - val_loss: 0.1656 - val_accuracy: 0.9168
Epoch 46/50
54/54 [==============================] - 2s 37ms/step - loss: 0.1721 - accuracy:
0.9052 - val_loss: 0.1647 - val_accuracy: 0.9168
Epoch 47/50
54/54 [==============================] - 2s 37ms/step - loss: 0.1718 - accuracy:
0.9053 - val_loss: 0.1647 - val_accuracy: 0.9168
Epoch 48/50
54/54 [==============================] - 2s 37ms/step - loss: 0.1716 - accuracy:
0.9053 - val_loss: 0.1640 - val_accuracy: 0.9169
Epoch 49/50
54/54 [==============================] - 2s 37ms/step - loss: 0.1717 - accuracy:
0.9056 - val_loss: 0.1646 - val_accuracy: 0.9169
Epoch 50/50
54/54 [==============================] - 2s 37ms/step - loss: 0.1713 - accuracy:
0.9056 - val_loss: 0.1649 - val_accuracy: 0.9169
```

In [ ]:
```
# Evaluate DNN on test data
score = model10.evaluate(Xtest,Ytest, batch_size=batch_size)

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

plot_results(history10)
```

```
12/12 [==============================] - 0s 13ms/step - loss: 0.1111 - accuracy:
0.9426
Test loss: 0.1111
Test accuracy: 0.9426
```





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?

In [ ]:
```
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, **kwarg
        super(myDropout, self).__init__(rate, noise_shape=None, seed=None,**kwar
        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.tr
            return K.in_train_phase(dropped_inputs, inputs, training=training)
        return inputs
```

Your best config, custom dropout

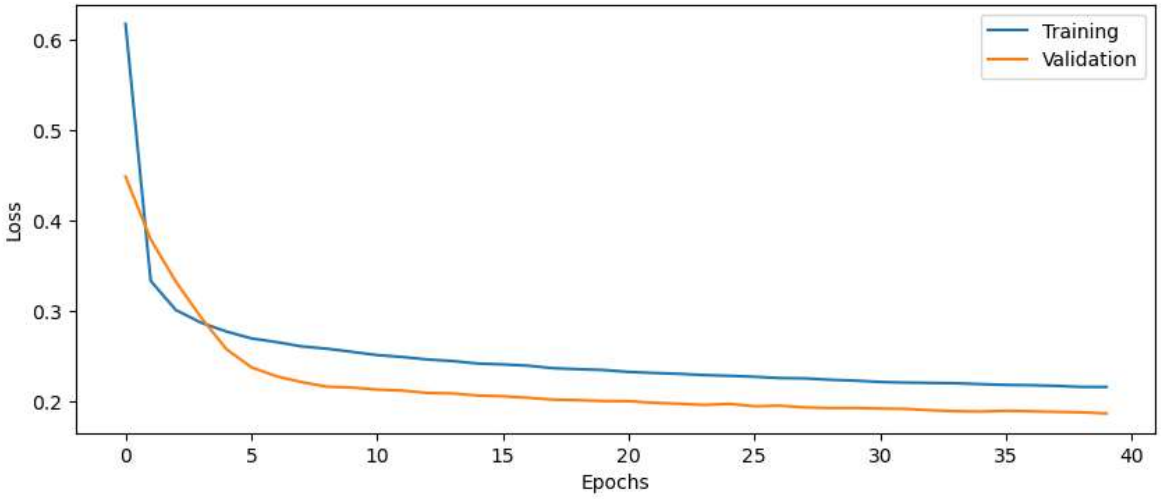
In [ ]:
```
# Your best training parameters
# the previous best training parameters took too much time so we change it a lit
batch_size = 10000
epochs = 40
input_shape = X.shape

# Build and train model
model11 = build_DNN(input_shape, n_layers=2, n_nodes=50, use_custom_dropout = 0.

history11 = model11.fit(Xtrain, Ytrain, epochs=epochs, batch_size=batch_size, va
```

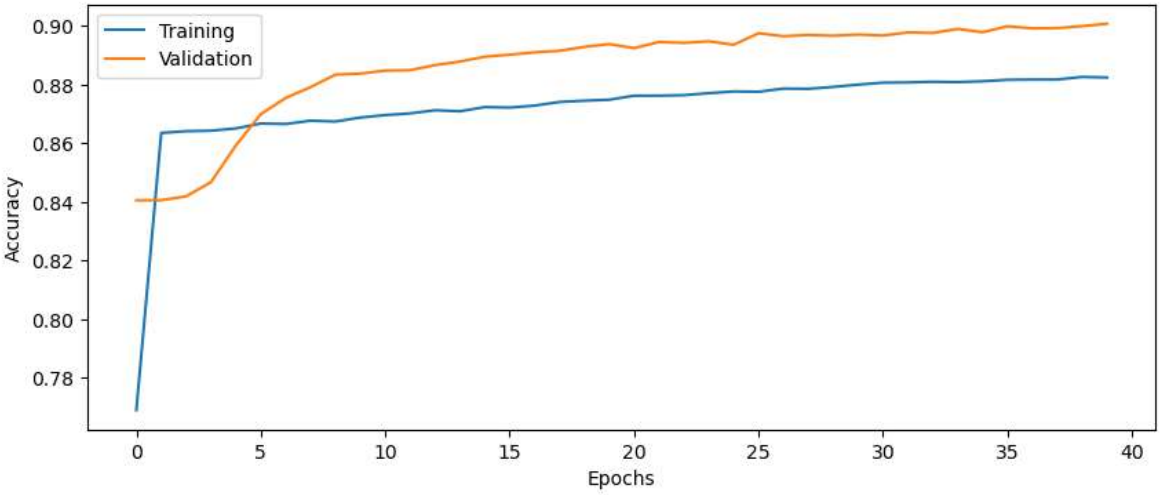
```
Epoch 1/40
54/54 [==============================] - 1s 25ms/step - loss: 0.6181 - accuracy:
0.7690 - val_loss: 0.4490 - val_accuracy: 0.8405
Epoch 2/40
54/54 [==============================] - 1s 23ms/step - loss: 0.3336 - accuracy:
0.8635 - val_loss: 0.3796 - val_accuracy: 0.8406
Epoch 3/40
54/54 [==============================] - 1s 23ms/step - loss: 0.3014 - accuracy:
0.8641 - val_loss: 0.3329 - val_accuracy: 0.8419
Epoch 4/40
54/54 [==============================] - 1s 23ms/step - loss: 0.2873 - accuracy:
0.8643 - val_loss: 0.2932 - val_accuracy: 0.8468
Epoch 5/40
54/54 [==============================] - 1s 23ms/step - loss: 0.2776 - accuracy:
0.8650 - val_loss: 0.2583 - val_accuracy: 0.8593
Epoch 6/40
54/54 [==============================] - 1s 23ms/step - loss: 0.2700 - accuracy:
0.8667 - val_loss: 0.2379 - val_accuracy: 0.8699
Epoch 7/40
54/54 [==============================] - 1s 23ms/step - loss: 0.2659 - accuracy:
0.8666 - val_loss: 0.2280 - val_accuracy: 0.8754
Epoch 8/40
54/54 [==============================] - 1s 23ms/step - loss: 0.2611 - accuracy:
0.8677 - val_loss: 0.2215 - val_accuracy: 0.8791
Epoch 9/40
54/54 [==============================] - 1s 23ms/step - loss: 0.2585 - accuracy:
0.8674 - val_loss: 0.2165 - val_accuracy: 0.8834
Epoch 10/40
54/54 [==============================] - 1s 23ms/step - loss: 0.2550 - accuracy:
0.8688 - val_loss: 0.2155 - val_accuracy: 0.8837
Epoch 11/40
54/54 [==============================] - 1s 23ms/step - loss: 0.2515 - accuracy:
0.8696 - val_loss: 0.2133 - val_accuracy: 0.8848
Epoch 12/40
54/54 [==============================] - 1s 23ms/step - loss: 0.2493 - accuracy:
0.8702 - val_loss: 0.2121 - val_accuracy: 0.8849
Epoch 13/40
54/54 [==============================] - 1s 23ms/step - loss: 0.2466 - accuracy:
0.8712 - val_loss: 0.2095 - val_accuracy: 0.8866
Epoch 14/40
54/54 [==============================] - 1s 23ms/step - loss: 0.2448 - accuracy:
0.8709 - val_loss: 0.2090 - val_accuracy: 0.8878
Epoch 15/40
54/54 [==============================] - 1s 23ms/step - loss: 0.2421 - accuracy:
0.8723 - val_loss: 0.2066 - val_accuracy: 0.8895
Epoch 16/40
54/54 [==============================] - 1s 23ms/step - loss: 0.2411 - accuracy:
0.8722 - val_loss: 0.2058 - val_accuracy: 0.8902
Epoch 17/40
54/54 [==============================] - 1s 23ms/step - loss: 0.2397 - accuracy:
0.8728 - val_loss: 0.2041 - val_accuracy: 0.8910
Epoch 18/40
54/54 [==============================] - 1s 23ms/step - loss: 0.2369 - accuracy:
0.8741 - val_loss: 0.2021 - val_accuracy: 0.8915
Epoch 19/40
54/54 [==============================] - 1s 23ms/step - loss: 0.2358 - accuracy:
0.8745 - val_loss: 0.2015 - val_accuracy: 0.8928
Epoch 20/40
54/54 [==============================] - 1s 23ms/step - loss: 0.2349 - accuracy:
0.8748 - val_loss: 0.2006 - val_accuracy: 0.8938
```

```
Epoch 21/40
54/54 [==============================] - 1s 23ms/step - loss: 0.2328 - accuracy:
0.8762 - val_loss: 0.2004 - val_accuracy: 0.8924
Epoch 22/40
54/54 [==============================] - 1s 23ms/step - loss: 0.2316 - accuracy:
0.8762 - val_loss: 0.1986 - val_accuracy: 0.8945
Epoch 23/40
54/54 [==============================] - 1s 23ms/step - loss: 0.2307 - accuracy:
0.8764 - val_loss: 0.1976 - val_accuracy: 0.8942
Epoch 24/40
54/54 [==============================] - 1s 23ms/step - loss: 0.2293 - accuracy:
0.8771 - val_loss: 0.1963 - val_accuracy: 0.8947
Epoch 25/40
54/54 [==============================] - 1s 23ms/step - loss: 0.2285 - accuracy:
0.8777 - val_loss: 0.1974 - val_accuracy: 0.8936
Epoch 26/40
54/54 [==============================] - 1s 23ms/step - loss: 0.2275 - accuracy:
0.8776 - val_loss: 0.1948 - val_accuracy: 0.8975
Epoch 27/40
54/54 [==============================] - 1s 23ms/step - loss: 0.2260 - accuracy:
0.8786 - val_loss: 0.1954 - val_accuracy: 0.8965
Epoch 28/40
54/54 [==============================] - 1s 23ms/step - loss: 0.2257 - accuracy:
0.8785 - val_loss: 0.1935 - val_accuracy: 0.8969
Epoch 29/40
54/54 [==============================] - 1s 23ms/step - loss: 0.2241 - accuracy:
0.8792 - val_loss: 0.1928 - val_accuracy: 0.8967
Epoch 30/40
54/54 [==============================] - 1s 23ms/step - loss: 0.2232 - accuracy:
0.8800 - val_loss: 0.1928 - val_accuracy: 0.8971
Epoch 31/40
54/54 [==============================] - 1s 23ms/step - loss: 0.2217 - accuracy:
0.8806 - val_loss: 0.1923 - val_accuracy: 0.8967
Epoch 32/40
54/54 [==============================] - 1s 23ms/step - loss: 0.2210 - accuracy:
0.8807 - val_loss: 0.1918 - val_accuracy: 0.8978
Epoch 33/40
54/54 [==============================] - 1s 23ms/step - loss: 0.2206 - accuracy:
0.8809 - val_loss: 0.1904 - val_accuracy: 0.8976
Epoch 34/40
54/54 [==============================] - 1s 23ms/step - loss: 0.2203 - accuracy:
0.8808 - val_loss: 0.1893 - val_accuracy: 0.8990
Epoch 35/40
54/54 [==============================] - 1s 23ms/step - loss: 0.2193 - accuracy:
0.8811 - val_loss: 0.1889 - val_accuracy: 0.8979
Epoch 36/40
54/54 [==============================] - 1s 23ms/step - loss: 0.2184 - accuracy:
0.8816 - val_loss: 0.1898 - val_accuracy: 0.8999
Epoch 37/40
54/54 [==============================] - 1s 23ms/step - loss: 0.2181 - accuracy:
0.8817 - val_loss: 0.1892 - val_accuracy: 0.8992
Epoch 38/40
54/54 [==============================] - 1s 23ms/step - loss: 0.2173 - accuracy:
0.8817 - val_loss: 0.1886 - val_accuracy: 0.8993
Epoch 39/40
54/54 [==============================] - 1s 22ms/step - loss: 0.2162 - accuracy:
0.8826 - val_loss: 0.1881 - val_accuracy: 0.8999
Epoch 40/40
54/54 [==============================] - 1s 22ms/step - loss: 0.2161 - accuracy:
0.8824 - val_loss: 0.1868 - val_accuracy: 0.9007
```

In [ ]:
```
# 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

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

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

plot_results(history11)
```

```
12/12 [==============================] - 0s 6ms/step - loss: 0.1874 - accuracy:
0.9011
Test accuracy: 0.9011
crossentropy: 0.1874
```





In [ ]:
```
# Run the testing 100 times, and save the accuracies in an array
accuracy = []
for i in range(100):
    score = model11.evaluate(Xtest,Ytest, batch_size=batch_size, verbose=0)
    accuracy.append(score[1])
```

In [ ]:
```
import numpy as np
# Calculate and print mean and std of accuracies
print(f"mean = {np.mean(accuracy)}" )
print(f"std = {np.std(accuracy)}" )
```

```
mean = 0.8994750535488129
std = 0.0006349640825825808
```

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?

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.

In [ ]:
```
from sklearn.model_selection import StratifiedKFold

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

accuracy=[]
for i, (train_index, test_index) in enumerate(skf.split(X, Y)):
# Loop over cross validation folds
    Xtrain = X[train_index,:]
    Ytrain = Y[train_index]
    Xtest = X[test_index,:]
    Ytest = Y[test_index]
    # Calculate class weights for current split
    lass_weights = class_weight.compute_class_weight(class_weight='balanced', cl

    # Rebuild the DNN model, to not continue training on the previously trained
    batch_size = 10000
    epochs = 20
    input_shape = X.shape
    modelKF = build_DNN(input_shape, n_layers=2, n_nodes=20)

    # Fit the model with training set and class weights for this fold
    historyKF = modelKF.fit(Xtrain, Ytrain, verbose = 0, epochs=epochs, batch_si

    # Evaluate the model using the test set for this fold
    score = modelKF.evaluate(Xtest,Ytest, batch_size=batch_size, verbose = 0)
    # Save the test accuracy in an array
    accuracy.append(score[1])
```

In [ ]:
```
# Calculate and print mean and std of accuracies
print(f"mean = {np.mean(accuracy)}" )
print(f"std = {np.std(accuracy)}" )
```

```
mean = 0.905460661649704
std = 0.002352768721738971
```

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?

- make the output linear(with no activation) to make possible output have a range of (-inf, inf)

Report

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