DNN_Lab_2023_rv1

May 13, 2023

1 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.

2 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

3 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
```

```python
# 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)
```

4 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, and the gradients used for backpropagation.)

5 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.

[ ]:
```python
from numpy import genfromtxt # Not needed if you load data from numpy arrays
import numpy as np
```

```python
# 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
```

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

[ ]:
```python
print(f"classes of labels:{np.unique(Y)}")
print(f"highest acuuracy of naive model:{max(np.sum(Y) / Y.shape[0], 1-np.
↪sum(Y) / Y.shape[0])}")
```

```
classes of labels:[0. 1.]
highest acuuracy of naive model:0.8408387501194158
```

6 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. - there are 2 classes from the labels, so the naive classifier will have a 50% misclassification rate if the label is qually distributed. But within this dataset, it has a highest accuracy of 0.84

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.
  - as discussed above, it's 0.84
- 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.
  - this is a binary classification. the accuracy will be 0.5

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(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
```

7 Part 6: Preprocessing

Lets do some simple preprocessing

```
[ ]: # 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␣
↪printing mean and std
meanList = []
stdevList = []
for col in range(X.shape[1]):
    meanList.append(np.mean(X[:,col]))
    stdevList.append(np.std(X[:,col]))
```

```python
print(meanList)
print(stdevList)
```

```
[-2.615704665123089e-17, 2.910099296114191e-16, 1.757069155782648e-16,
1.582998799113241e-16, 5.570251413421029e-16, 7.989383036877916e-17,
1.1143478388494525e-16, 5.604098427217858e-16, 2.423594965935592e-16,
-5.4192416595582517e-17, -2.4771550756800247e-17, -7.327320569203676e-18,
9.74496441183433e-18, -1.5398531551524477e-17, -4.945383466402643e-16,
-5.787467414051228e-17, 6.27843508670853e-17, -7.659095693453913e-16,
1.2692630173810936e-17, -2.7152000078775045e-18, -1.8597260327928113e-17,
4.0021304225701295e-17, 1.0034337782536893e-15, 1.4357084973160502e-16,
-7.648681227670274e-16, 9.921638384949648e-18, -1.9564317864980374e-17,
1.5007989084637987e-16, -3.670355298319892e-16, -7.58768221379467e-16,
-1.6960701419070437e-17, -7.58247498090285e-16, -6.267276730511774e-18,
-2.0456986360720924e-17, -1.0473977016689112e-16, -8.084601009756909e-16,
5.890124291061392e-16, 1.2051024692497417e-17, -3.865254586556579e-16,
-2.287463020335158e-17, -7.438904131171245e-19, 1.0265687701016318e-16,
5.604098427217858e-16, -3.0499506937802103e-18, 6.760104129201869e-18,
-4.992992452842139e-16, 1.0098312358064965e-17, 5.767010427690507e-17,
3.987252614307787e-17, 1.368758360135509e-17, 4.7162652191625695e-17,
-3.616051298162342e-16, -1.320405483282896e-18, -2.9383671318126417e-17,
-1.785336991481099e-18, 8.123283311239e-17, -7.156225774186738e-17,
-2.5842752951688905e-16, 7.607023364535715e-16, 9.376738657341354e-17,
3.9240219291928315e-16, -1.575187949775511e-17, -9.388529029807068e-19,
9.298630163964056e-20, 7.141347965924395e-18, 7.58768221379467e-17,
-3.0276339813866965e-17, 2.917538200245362e-16, 7.438904131171245e-20,
8.823528279024017e-19, 1.2134712363973092e-18, -4.2520776013774835e-16,
-2.097473408825044e-15, -2.5143495963358808e-17, -7.38831958307928e-16,
-8.219989064944225e-18, 1.4761575385292939e-19, -2.3641767191878614e-18,
-1.398513976660194e-16, -1.2193107761402787e-15, 7.0669589246126825e-19,
8.951977231451477e-16, 5.806064674379157e-17, 1.6281320252715815e-18,
-5.497815084443748e-18, 6.843791800677545e-17, 9.091084738704377e-16,
-7.167384130383494e-17, -6.263557278446188e-16, -1.6123824704313673e-17,
8.694219203306392e-19, -4.760898643949597e-18]
[1.0, 0.9999999999999998, 0.9999999999999998, 1.0000000000000002,
0.9999999999999996, 0.9999999999999997, 0.9999999999999999, 0.9999999999999998,
0.9999999999999998, 1.0000000000000004, 1.0, 1.0, 1.0000000000000002, 1.0,
0.9999999999999998, 1.0, 1.0000000000000004, 0.9999999999999997,
1.0000000000000002, 0.9999999999999996, 1.0, 1.0000000000000004,
0.9999999999999998, 0.9999999999999996, 0.9999999999999998, 1.0,
0.9999999999999997, 1.0, 1.0, 1.0, 1.0000000000000002, 1.0, 1.0,
0.9999999999999997, 0.9999999999999997, 0.9999999999999998, 0.9999999999999998,
1.0, 1.0, 0.9999999999999996, 0.9999999999999998, 0.9999999999999999,
0.9999999999999998, 1.0, 0.9999999999999968, 0.9999999999999997, 1.0,
0.9999999999999956, 1.0000000000000004, 0.9999999999999997, 1.0000000000000016,
1.0000000000000002, 0.9999999999999999, 1.0000000000000004, 0.9999999999999998,
1.0000000000000004, 1.0, 1.0, 1.0, 0.9999999999999997, 1.0, 0.9999999999999997,
0.999999999999999, 0.9999999999999976, 1.0, 1.0, 1.0000000000000002,
```

```
0.9999999999999998, 0.9999999999999998, 1.0000000000000033, 0.9999999999999993,
1.0, 0.9999999999999998, 1.0, 0.9999999999999998, 1.0000000000000002,
1.0000000000000013, 0.9999999999999991, 1.0, 1.0000000000000002,
1.000000000000004, 1.000000000000004, 1.0000000000000002, 1.0000000000000018,
0.9999999999999991, 1.0, 0.9999999999999996, 1.0, 0.9999999999999998,
0.9999999999999993, 0.9999999999999994, 1.0000000000000007]
```

8 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.tra

```
[ ]: 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_state=42)



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%
```

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

9 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,␣
↪random_state=42)

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,)
```

10 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, Activation, Dropout
from tensorflow.keras.optimizers import SGD, Adam
from keras.losses import CategoricalCrossentropy
```

```python
# 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':
        optimizer = SGD(learning_rate=learning_rate, decay=1e-6, momentum=0.9,␣
 ↪nesterov=True)

    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␣
 ↪function


    # 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, activation=act_fun))
            # model.add(Activation(act_fun))
            model.add(BatchNormalization())

            if use_dropout:
                if use_dropout == True:
                    use_dropout = 0.5
                model.add(Dropout(rate=use_dropout))

            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=['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()
```

11 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 !

11.0.1 2 layers, 20 nodes

[ ]:
```
# 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"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense (Dense)                (None, 20)                1860
_________________________________________________________________
dense_1 (Dense)              (None, 20)                420
_________________________________________________________________
dense_2 (Dense)              (None, 1)                 21
=================================================================
Total params: 2,301
Trainable params: 2,301
Non-trainable params: 0
_________________________________________________________________
```

[ ]:
```
# Train the model, provide training data and validation data7
history1 = model1.fit(Xtrain, Ytrain, epochs=epochs,␣
 ↪batch_size=batch_size,validation_data=(Xval,Yval))
```

```
Epoch 1/20
54/54 [==============================] - 0s 5ms/step - loss: 0.4718 - accuracy:
0.7896 - val_loss: 0.4072 - val_accuracy: 0.8404
Epoch 2/20
54/54 [==============================] - 0s 3ms/step - loss: 0.3901 - accuracy:
0.8406 - val_loss: 0.3709 - val_accuracy: 0.8404
Epoch 3/20
54/54 [==============================] - 0s 3ms/step - loss: 0.3488 - accuracy:
0.8406 - val_loss: 0.3235 - val_accuracy: 0.8404
Epoch 4/20
```

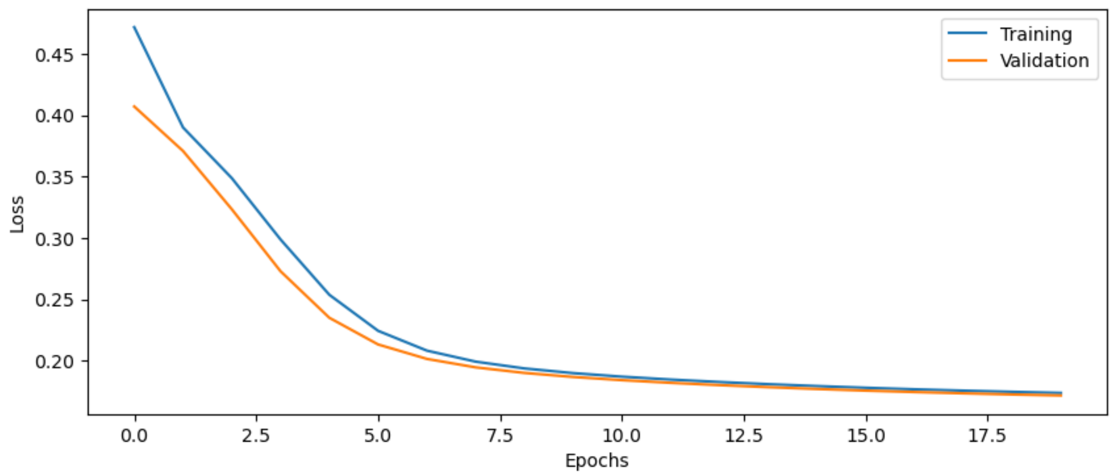
```
54/54 [==============================] - 0s 3ms/step - loss: 0.2988 - accuracy:
0.8406 - val_loss: 0.2729 - val_accuracy: 0.8404
Epoch 5/20
54/54 [==============================] - 0s 3ms/step - loss: 0.2538 - accuracy:
0.8441 - val_loss: 0.2351 - val_accuracy: 0.8526
Epoch 6/20
54/54 [==============================] - 0s 3ms/step - loss: 0.2244 - accuracy:
0.8655 - val_loss: 0.2133 - val_accuracy: 0.8784
Epoch 7/20
54/54 [==============================] - 0s 3ms/step - loss: 0.2083 - accuracy:
0.8877 - val_loss: 0.2016 - val_accuracy: 0.8962
Epoch 8/20
54/54 [==============================] - 0s 3ms/step - loss: 0.1993 - accuracy:
0.8965 - val_loss: 0.1947 - val_accuracy: 0.9007
Epoch 9/20
54/54 [==============================] - 0s 3ms/step - loss: 0.1938 - accuracy:
0.9012 - val_loss: 0.1901 - val_accuracy: 0.9043
Epoch 10/20
54/54 [==============================] - 0s 3ms/step - loss: 0.1900 - accuracy:
0.9027 - val_loss: 0.1869 - val_accuracy: 0.9051
Epoch 11/20
54/54 [==============================] - 0s 3ms/step - loss: 0.1871 - accuracy:
0.9038 - val_loss: 0.1843 - val_accuracy: 0.9061
Epoch 12/20
54/54 [==============================] - 0s 3ms/step - loss: 0.1848 - accuracy:
0.9051 - val_loss: 0.1821 - val_accuracy: 0.9077
Epoch 13/20
54/54 [==============================] - 0s 3ms/step - loss: 0.1827 - accuracy:
0.9065 - val_loss: 0.1802 - val_accuracy: 0.9088
Epoch 14/20
54/54 [==============================] - 0s 3ms/step - loss: 0.1810 - accuracy:
0.9073 - val_loss: 0.1785 - val_accuracy: 0.9093
Epoch 15/20
54/54 [==============================] - 0s 3ms/step - loss: 0.1794 - accuracy:
0.9078 - val_loss: 0.1771 - val_accuracy: 0.9097
Epoch 16/20
54/54 [==============================] - 0s 3ms/step - loss: 0.1780 - accuracy:
0.9081 - val_loss: 0.1757 - val_accuracy: 0.9100
Epoch 17/20
54/54 [==============================] - 0s 3ms/step - loss: 0.1768 - accuracy:
0.9083 - val_loss: 0.1746 - val_accuracy: 0.9102
Epoch 18/20
54/54 [==============================] - 0s 3ms/step - loss: 0.1757 - accuracy:
0.9085 - val_loss: 0.1735 - val_accuracy: 0.9106
Epoch 19/20
54/54 [==============================] - 0s 3ms/step - loss: 0.1747 - accuracy:
0.9087 - val_loss: 0.1726 - val_accuracy: 0.9107
Epoch 20/20
```

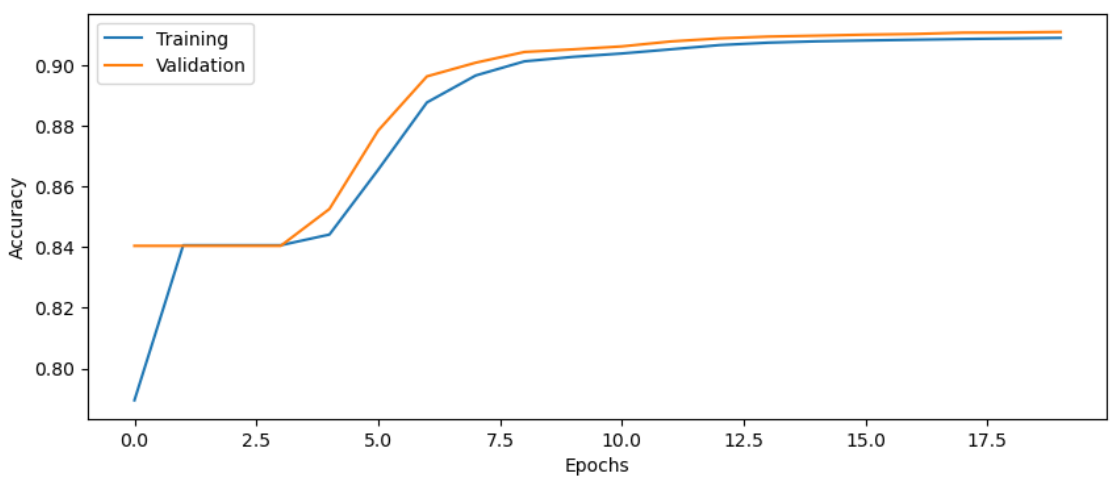
```
54/54 [==============================] - 0s 3ms/step - loss: 0.1739 - accuracy:
0.9089 - val_loss: 0.1718 - val_accuracy: 0.9109
```

[ ]:
```
# 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 2ms/step - loss: 0.1727 - accuracy:
0.9085
Test loss: 0.1727
Test accuracy: 0.9085
```

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





12 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,Draws samples from a truncated normal distribution centered on 0 with stddev = sqrt(2 / (fan_in + fan_out)) where fan_in is the number of input units in the weight tensor and fan_out is the number of output units in the weight tensor.) and the bias is initialized by zeros.

13 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

[ ]:
```python
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]}
```

```
[3.13728768 0.59479436]
```

13.0.1 2 layers, 20 nodes, class weights

[ ]:
```
# 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,validation_data=(Xval,Yval),␣
↪class_weight=class_weights)
```

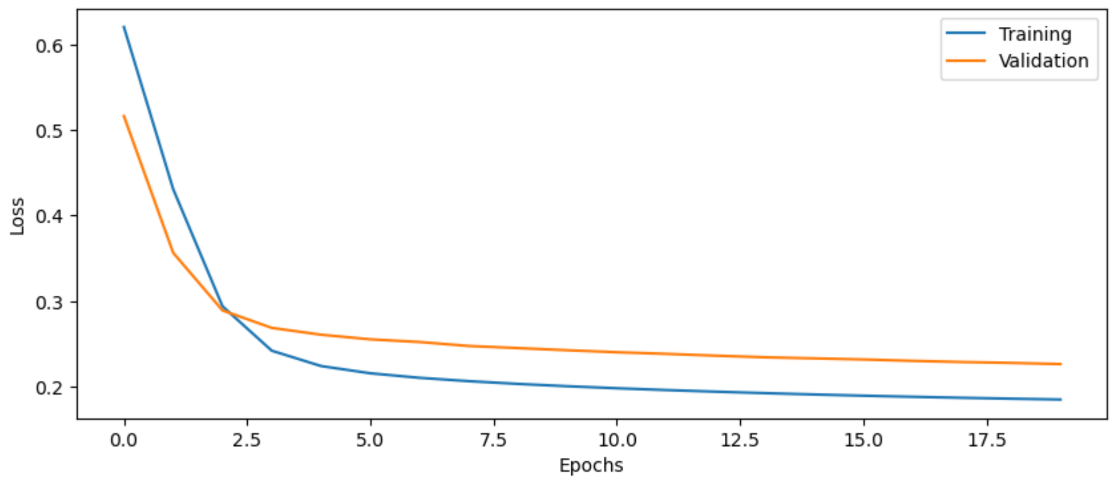
```
Epoch 1/20
54/54 [==============================] - 0s 5ms/step - loss: 0.6203 - accuracy:
0.8604 - val_loss: 0.5162 - val_accuracy: 0.8868
Epoch 2/20
54/54 [==============================] - 0s 4ms/step - loss: 0.4308 - accuracy:
0.8805 - val_loss: 0.3564 - val_accuracy: 0.8813
Epoch 3/20
54/54 [==============================] - 0s 3ms/step - loss: 0.2939 - accuracy:
0.8794 - val_loss: 0.2891 - val_accuracy: 0.8814
Epoch 4/20
54/54 [==============================] - 0s 3ms/step - loss: 0.2420 - accuracy:
0.8795 - val_loss: 0.2687 - val_accuracy: 0.8815
Epoch 5/20
54/54 [==============================] - 0s 3ms/step - loss: 0.2241 - accuracy:
0.8810 - val_loss: 0.2607 - val_accuracy: 0.8849
Epoch 6/20
54/54 [==============================] - 0s 3ms/step - loss: 0.2156 - accuracy:
0.8846 - val_loss: 0.2553 - val_accuracy: 0.8878
Epoch 7/20
54/54 [==============================] - 0s 3ms/step - loss: 0.2103 - accuracy:
0.8880 - val_loss: 0.2522 - val_accuracy: 0.8910
Epoch 8/20
54/54 [==============================] - 0s 3ms/step - loss: 0.2064 - accuracy:
0.8901 - val_loss: 0.2475 - val_accuracy: 0.8935
Epoch 9/20
54/54 [==============================] - 0s 3ms/step - loss: 0.2032 - accuracy:
0.8923 - val_loss: 0.2451 - val_accuracy: 0.8953
Epoch 10/20
54/54 [==============================] - 0s 3ms/step - loss: 0.2005 - accuracy:
0.8941 - val_loss: 0.2425 - val_accuracy: 0.8969
Epoch 11/20
54/54 [==============================] - 0s 3ms/step - loss: 0.1982 - accuracy:
0.8955 - val_loss: 0.2403 - val_accuracy: 0.8983
Epoch 12/20
```

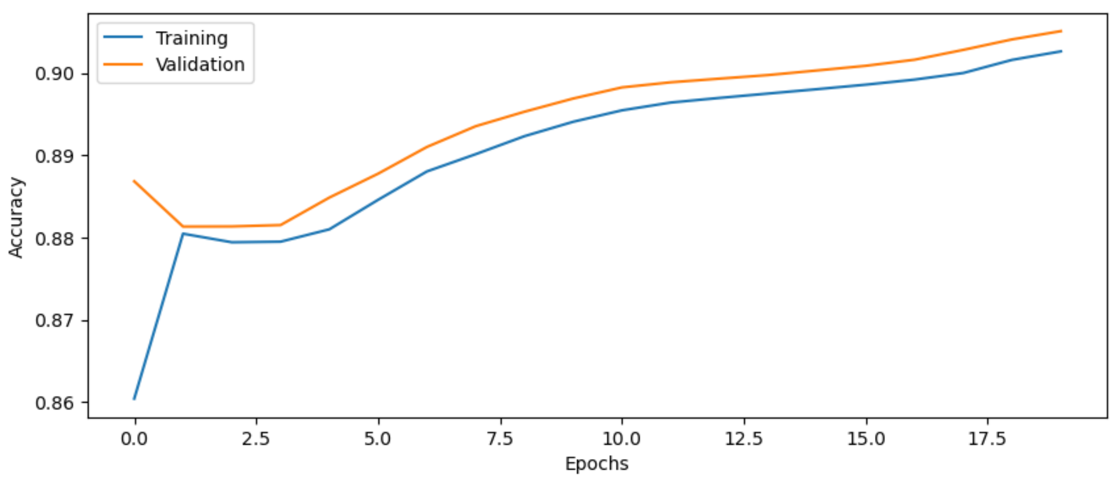
```
54/54 [==============================] - 0s 3ms/step - loss: 0.1961 - accuracy:
0.8964 - val_loss: 0.2383 - val_accuracy: 0.8989
Epoch 13/20
54/54 [==============================] - 0s 3ms/step - loss: 0.1942 - accuracy:
0.8970 - val_loss: 0.2362 - val_accuracy: 0.8993
Epoch 14/20
54/54 [==============================] - 0s 3ms/step - loss: 0.1924 - accuracy:
0.8975 - val_loss: 0.2343 - val_accuracy: 0.8998
Epoch 15/20
54/54 [==============================] - 0s 3ms/step - loss: 0.1909 - accuracy:
0.8980 - val_loss: 0.2331 - val_accuracy: 0.9003
Epoch 16/20
54/54 [==============================] - 0s 3ms/step - loss: 0.1895 - accuracy:
0.8986 - val_loss: 0.2318 - val_accuracy: 0.9009
Epoch 17/20
54/54 [==============================] - 0s 3ms/step - loss: 0.1882 - accuracy:
0.8992 - val_loss: 0.2302 - val_accuracy: 0.9016
Epoch 18/20
54/54 [==============================] - 0s 3ms/step - loss: 0.1870 - accuracy:
0.9000 - val_loss: 0.2288 - val_accuracy: 0.9028
Epoch 19/20
54/54 [==============================] - 0s 3ms/step - loss: 0.1860 - accuracy:
0.9016 - val_loss: 0.2278 - val_accuracy: 0.9041
Epoch 20/20
54/54 [==============================] - 0s 3ms/step - loss: 0.1850 - accuracy:
0.9026 - val_loss: 0.2264 - val_accuracy: 0.9051
```

[ ]:
```python
# 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: 0.2301 - accuracy:
0.9035
Test loss: 0.2301
Test accuracy: 0.9035
```

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





14 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. And the noise of each batch will also reduce when batch size increases. Thus, to deal with more noisy batches with batch size is decreased, we need a smaller learning rate to avoid overfitting of the noisy data.

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

15 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

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

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense (Dense)                (None, 20)                1860
_________________________________________________________________
dense_1 (Dense)              (None, 20)                420
_________________________________________________________________
dense_2 (Dense)              (None, 1)                 21
=================================================================
Total params: 2,301
Trainable params: 2,301
Non-trainable params: 0
_________________________________________________________________
```

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

```
Model: "sequential_2"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_6 (Dense)              (None, 50)                4650
_________________________________________________________________
dense_7 (Dense)              (None, 50)                2550
_________________________________________________________________
dense_8 (Dense)              (None, 50)                2550
_________________________________________________________________
dense_9 (Dense)              (None, 50)                2550
_________________________________________________________________
dense_10 (Dense)             (None, 1)                 51
=================================================================
Total params: 12,351
Trainable params: 12,351
Non-trainable params: 0
_________________________________________________________________
```

15.0.1 4 layers, 20 nodes, class weights

[ ]:
```
# 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,validation_data=(Xval,Yval),␣
 ↪class_weight=class_weights)
```

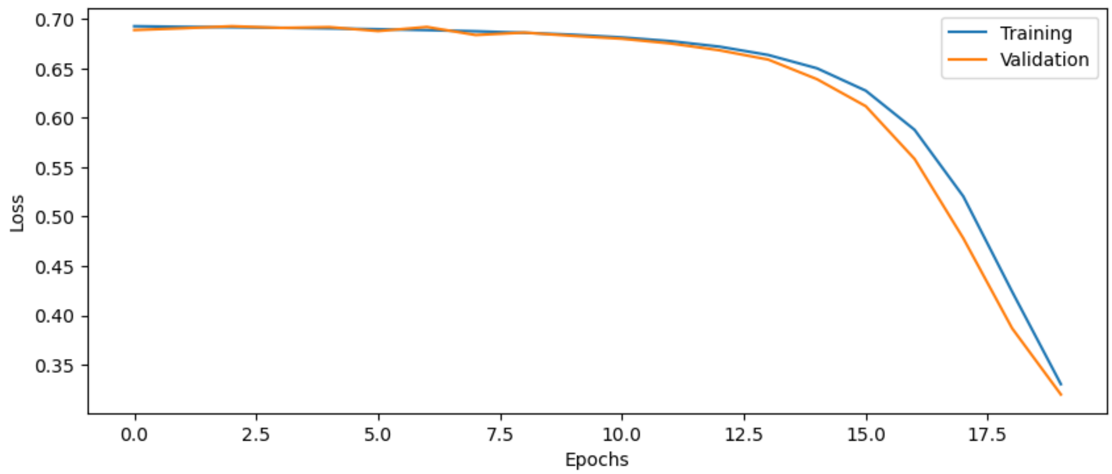
```
Epoch 1/20
54/54 [==============================] - 0s 6ms/step - loss: 0.6927 - accuracy:
0.5042 - val_loss: 0.6888 - val_accuracy: 0.8404
Epoch 2/20
54/54 [==============================] - 0s 4ms/step - loss: 0.6920 - accuracy:
0.7163 - val_loss: 0.6905 - val_accuracy: 0.8615
Epoch 3/20
54/54 [==============================] - 0s 4ms/step - loss: 0.6915 - accuracy:
0.7256 - val_loss: 0.6927 - val_accuracy: 0.7036
Epoch 4/20
54/54 [==============================] - 0s 4ms/step - loss: 0.6910 - accuracy:
0.8681 - val_loss: 0.6909 - val_accuracy: 0.8819
```

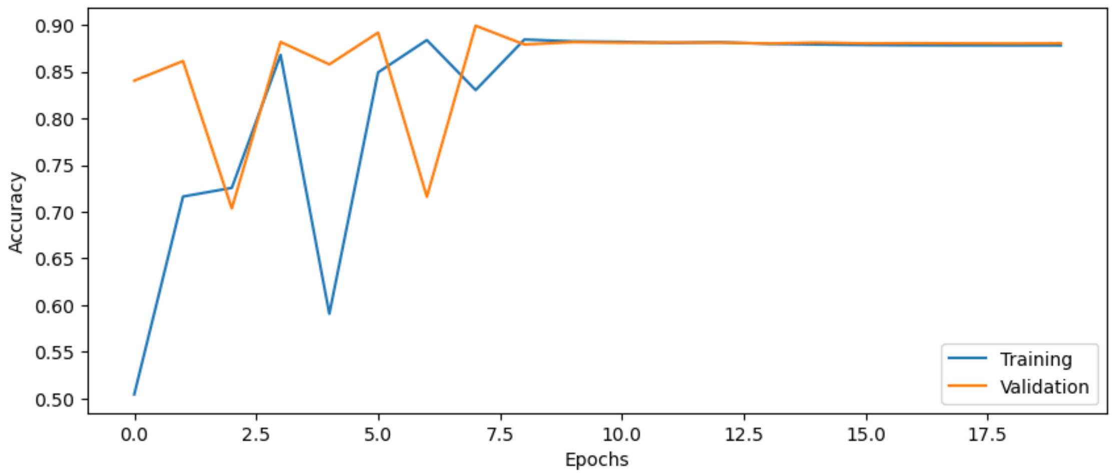
```
Epoch 5/20
54/54 [==============================] - 0s 4ms/step - loss: 0.6904 - accuracy:
0.5906 - val_loss: 0.6917 - val_accuracy: 0.8579
Epoch 6/20
54/54 [==============================] - 0s 4ms/step - loss: 0.6896 - accuracy:
0.8493 - val_loss: 0.6877 - val_accuracy: 0.8919
Epoch 7/20
54/54 [==============================] - 0s 4ms/step - loss: 0.6887 - accuracy:
0.8840 - val_loss: 0.6919 - val_accuracy: 0.7160
Epoch 8/20
54/54 [==============================] - 0s 4ms/step - loss: 0.6875 - accuracy:
0.8304 - val_loss: 0.6838 - val_accuracy: 0.8993
Epoch 9/20
54/54 [==============================] - 0s 4ms/step - loss: 0.6860 - accuracy:
0.8846 - val_loss: 0.6862 - val_accuracy: 0.8792
Epoch 10/20
54/54 [==============================] - 0s 4ms/step - loss: 0.6840 - accuracy:
0.8826 - val_loss: 0.6828 - val_accuracy: 0.8818
Epoch 11/20
54/54 [==============================] - 0s 4ms/step - loss: 0.6813 - accuracy:
0.8821 - val_loss: 0.6800 - val_accuracy: 0.8811
Epoch 12/20
54/54 [==============================] - 0s 4ms/step - loss: 0.6775 - accuracy:
0.8810 - val_loss: 0.6752 - val_accuracy: 0.8813
Epoch 13/20
54/54 [==============================] - 0s 4ms/step - loss: 0.6720 - accuracy:
0.8817 - val_loss: 0.6683 - val_accuracy: 0.8813
Epoch 14/20
54/54 [==============================] - 0s 4ms/step - loss: 0.6635 - accuracy:
0.8800 - val_loss: 0.6589 - val_accuracy: 0.8803
Epoch 15/20
54/54 [==============================] - 0s 4ms/step - loss: 0.6500 - accuracy:
0.8793 - val_loss: 0.6390 - val_accuracy: 0.8813
Epoch 16/20
54/54 [==============================] - 0s 4ms/step - loss: 0.6274 - accuracy:
0.8787 - val_loss: 0.6117 - val_accuracy: 0.8804
Epoch 17/20
54/54 [==============================] - 0s 4ms/step - loss: 0.5878 - accuracy:
0.8784 - val_loss: 0.5583 - val_accuracy: 0.8806
Epoch 18/20
54/54 [==============================] - 0s 4ms/step - loss: 0.5204 - accuracy:
0.8783 - val_loss: 0.4779 - val_accuracy: 0.8805
Epoch 19/20
54/54 [==============================] - 0s 4ms/step - loss: 0.4240 - accuracy:
0.8782 - val_loss: 0.3869 - val_accuracy: 0.8803
Epoch 20/20
54/54 [==============================] - 0s 4ms/step - loss: 0.3303 - accuracy:
0.8783 - val_loss: 0.3199 - val_accuracy: 0.8805
```

[ ]:
```
# 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: 0.3224 - accuracy:
0.8785
Test loss: 0.3224
Test accuracy: 0.8785
```

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





15.0.2 2 layers, 50 nodes, class weights

[ ]:
```
# 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,validation_data=(Xval,Yval),␣
 ↪class_weight=class_weights)
```

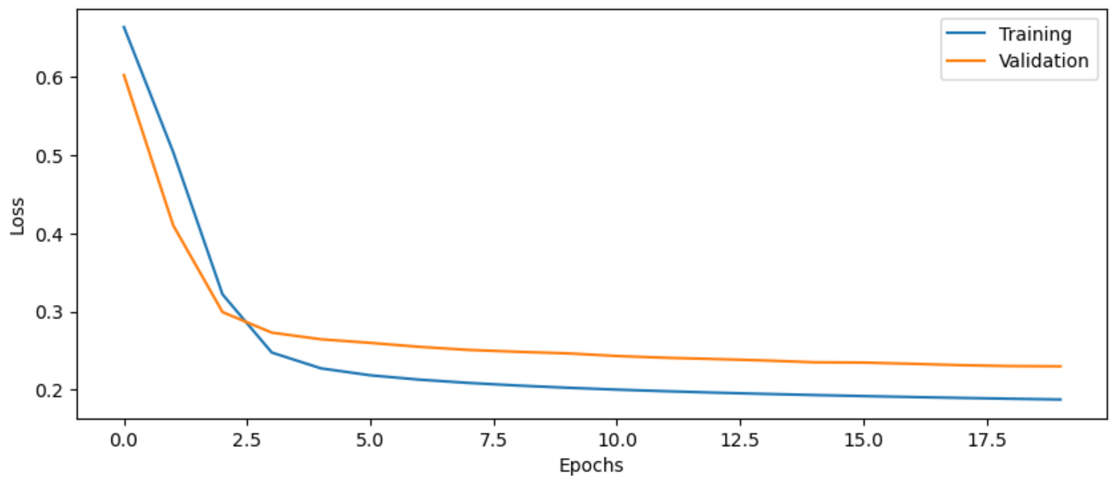
```
Epoch 1/20
54/54 [==============================] - 0s 7ms/step - loss: 0.6643 - accuracy:
0.6735 - val_loss: 0.6028 - val_accuracy: 0.8819
Epoch 2/20
54/54 [==============================] - 0s 6ms/step - loss: 0.5042 - accuracy:
0.8799 - val_loss: 0.4104 - val_accuracy: 0.8820
Epoch 3/20
54/54 [==============================] - 0s 6ms/step - loss: 0.3220 - accuracy:
0.8801 - val_loss: 0.2993 - val_accuracy: 0.8821
Epoch 4/20
54/54 [==============================] - 0s 6ms/step - loss: 0.2475 - accuracy:
0.8803 - val_loss: 0.2730 - val_accuracy: 0.8824
Epoch 5/20
54/54 [==============================] - 0s 6ms/step - loss: 0.2272 - accuracy:
0.8813 - val_loss: 0.2645 - val_accuracy: 0.8840
Epoch 6/20
54/54 [==============================] - 0s 6ms/step - loss: 0.2183 - accuracy:
0.8831 - val_loss: 0.2598 - val_accuracy: 0.8856
Epoch 7/20
54/54 [==============================] - 0s 6ms/step - loss: 0.2127 - accuracy:
0.8851 - val_loss: 0.2548 - val_accuracy: 0.8880
Epoch 8/20
54/54 [==============================] - 0s 6ms/step - loss: 0.2086 - accuracy:
0.8868 - val_loss: 0.2508 - val_accuracy: 0.8899
Epoch 9/20
54/54 [==============================] - 0s 6ms/step - loss: 0.2052 - accuracy:
0.8895 - val_loss: 0.2484 - val_accuracy: 0.8922
Epoch 10/20
54/54 [==============================] - 0s 6ms/step - loss: 0.2024 - accuracy:
0.8923 - val_loss: 0.2464 - val_accuracy: 0.8952
Epoch 11/20
54/54 [==============================] - 0s 6ms/step - loss: 0.2000 - accuracy:
0.8948 - val_loss: 0.2431 - val_accuracy: 0.8976
Epoch 12/20
```

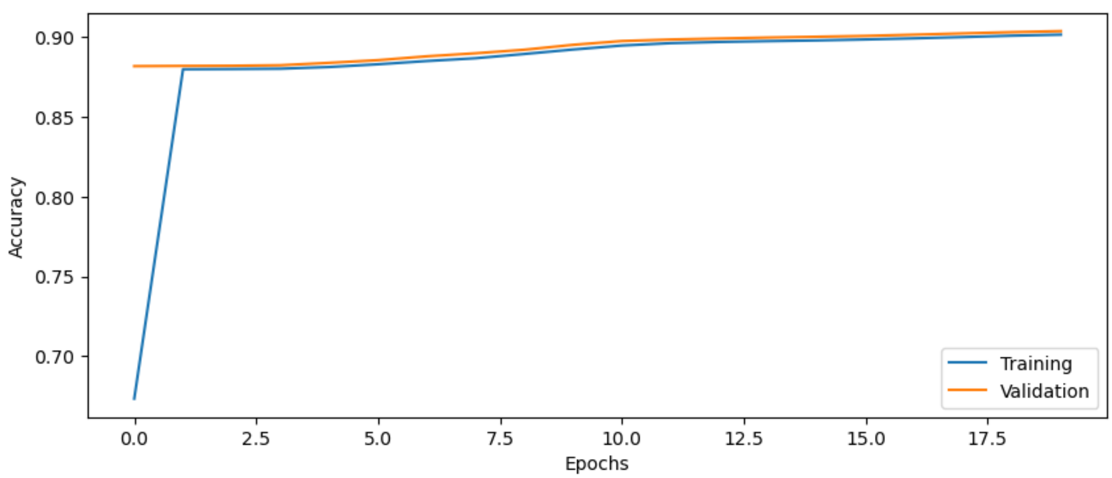
```
54/54 [==============================] - 0s 6ms/step - loss: 0.1979 - accuracy:
0.8963 - val_loss: 0.2408 - val_accuracy: 0.8986
Epoch 13/20
54/54 [==============================] - 0s 6ms/step - loss: 0.1961 - accuracy:
0.8970 - val_loss: 0.2391 - val_accuracy: 0.8992
Epoch 14/20
54/54 [==============================] - 0s 6ms/step - loss: 0.1945 - accuracy:
0.8975 - val_loss: 0.2373 - val_accuracy: 0.8998
Epoch 15/20
54/54 [==============================] - 0s 6ms/step - loss: 0.1931 - accuracy:
0.8980 - val_loss: 0.2350 - val_accuracy: 0.9003
Epoch 16/20
54/54 [==============================] - 0s 6ms/step - loss: 0.1917 - accuracy:
0.8986 - val_loss: 0.2346 - val_accuracy: 0.9008
Epoch 17/20
54/54 [==============================] - 0s 6ms/step - loss: 0.1905 - accuracy:
0.8993 - val_loss: 0.2331 - val_accuracy: 0.9016
Epoch 18/20
54/54 [==============================] - 0s 6ms/step - loss: 0.1894 - accuracy:
0.9001 - val_loss: 0.2313 - val_accuracy: 0.9024
Epoch 19/20
54/54 [==============================] - 0s 6ms/step - loss: 0.1883 - accuracy:
0.9010 - val_loss: 0.2302 - val_accuracy: 0.9032
Epoch 20/20
54/54 [==============================] - 0s 6ms/step - loss: 0.1873 - accuracy:
0.9016 - val_loss: 0.2298 - val_accuracy: 0.9038
```

[ ]:
```
# 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: 0.2335 - accuracy:
0.9018
Test loss: 0.2335
Test accuracy: 0.9018
```

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





15.0.3 4 layers, 50 nodes, class weights

```
[ ]: # 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,validation_data=(Xval,Yval),
    class_weight=class_weights)
```

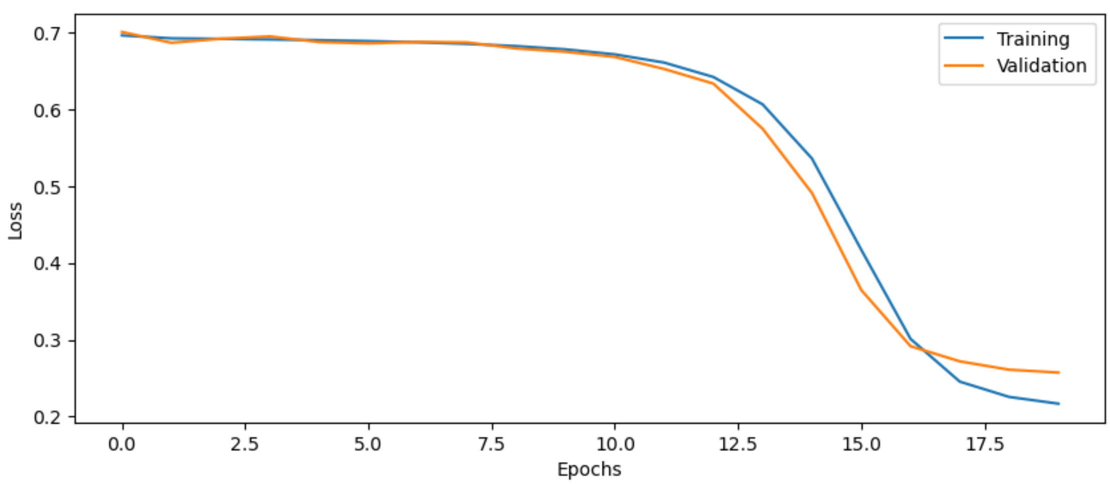
```
Epoch 1/20
54/54 [==============================] - 1s 11ms/step - loss: 0.6959 - accuracy:
0.5043 - val_loss: 0.7004 - val_accuracy: 0.1596
Epoch 2/20
54/54 [==============================] - 1s 11ms/step - loss: 0.6921 - accuracy:
0.5105 - val_loss: 0.6864 - val_accuracy: 0.8407
Epoch 3/20
54/54 [==============================] - 1s 10ms/step - loss: 0.6915 - accuracy:
0.6771 - val_loss: 0.6916 - val_accuracy: 0.8727
Epoch 4/20
54/54 [==============================] - 1s 9ms/step - loss: 0.6907 - accuracy:
0.7202 - val_loss: 0.6946 - val_accuracy: 0.1593
Epoch 5/20
54/54 [==============================] - 0s 9ms/step - loss: 0.6897 - accuracy:
0.5927 - val_loss: 0.6873 - val_accuracy: 0.8791
Epoch 6/20
54/54 [==============================] - 0s 9ms/step - loss: 0.6885 - accuracy:
0.8235 - val_loss: 0.6856 - val_accuracy: 0.8826
Epoch 7/20
54/54 [==============================] - 0s 9ms/step - loss: 0.6870 - accuracy:
0.8577 - val_loss: 0.6875 - val_accuracy: 0.8742
Epoch 8/20
54/54 [==============================] - 0s 9ms/step - loss: 0.6849 - accuracy:
0.8795 - val_loss: 0.6868 - val_accuracy: 0.8703
Epoch 9/20
54/54 [==============================] - 0s 9ms/step - loss: 0.6820 - accuracy:
0.8820 - val_loss: 0.6790 - val_accuracy: 0.8851
Epoch 10/20
54/54 [==============================] - 0s 9ms/step - loss: 0.6778 - accuracy:
0.8815 - val_loss: 0.6746 - val_accuracy: 0.8854
Epoch 11/20
54/54 [==============================] - 0s 9ms/step - loss: 0.6712 - accuracy:
0.8818 - val_loss: 0.6679 - val_accuracy: 0.8819
Epoch 12/20
54/54 [==============================] - 0s 9ms/step - loss: 0.6606 - accuracy:
0.8810 - val_loss: 0.6522 - val_accuracy: 0.8855
Epoch 13/20
54/54 [==============================] - 0s 9ms/step - loss: 0.6418 - accuracy:
0.8822 - val_loss: 0.6332 - val_accuracy: 0.8776
Epoch 14/20
54/54 [==============================] - 1s 9ms/step - loss: 0.6062 - accuracy:
0.8791 - val_loss: 0.5744 - val_accuracy: 0.8834
Epoch 15/20
54/54 [==============================] - 0s 9ms/step - loss: 0.5357 - accuracy:
0.8790 - val_loss: 0.4909 - val_accuracy: 0.8791
Epoch 16/20
54/54 [==============================] - 1s 10ms/step - loss: 0.4170 - accuracy:
0.8777 - val_loss: 0.3647 - val_accuracy: 0.8798
```

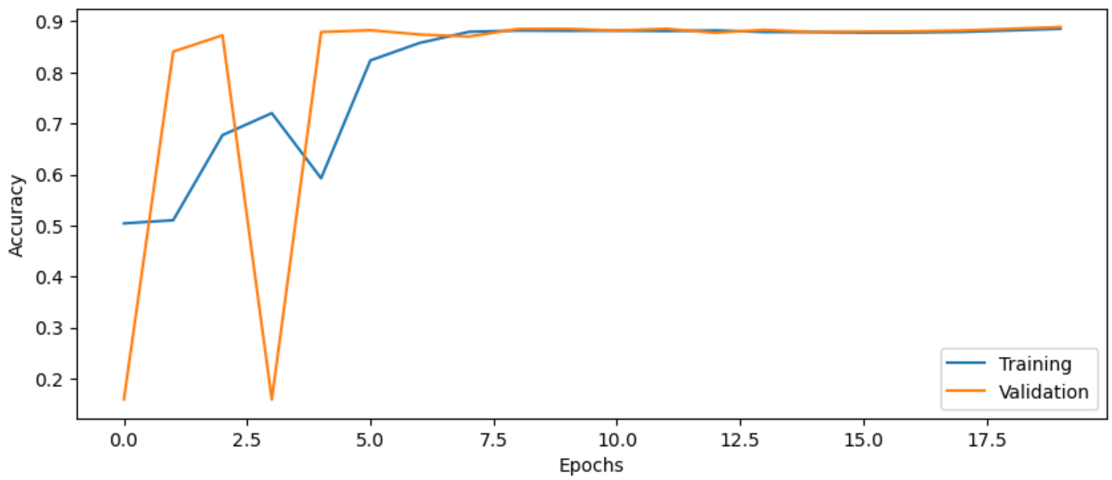
```
Epoch 17/20
54/54 [==============================] - 1s 10ms/step - loss: 0.3011 - accuracy:
0.8781 - val_loss: 0.2915 - val_accuracy: 0.8805
Epoch 18/20
54/54 [==============================] - 1s 9ms/step - loss: 0.2454 - accuracy:
0.8793 - val_loss: 0.2718 - val_accuracy: 0.8821
Epoch 19/20
54/54 [==============================] - 1s 9ms/step - loss: 0.2256 - accuracy:
0.8823 - val_loss: 0.2610 - val_accuracy: 0.8855
Epoch 20/20
54/54 [==============================] - 0s 9ms/step - loss: 0.2168 - accuracy:
0.8854 - val_loss: 0.2573 - val_accuracy: 0.8888
```

[ ]:
```
# 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: 0.2610 - accuracy:
0.8867
Test loss: 0.2610
Test accuracy: 0.8867
```

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





16 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(interrmidiate input of layers after batch normalization) makes optimization easier, since the loss function behaves nicer (more isotropic)

16.0.1 2 layers, 20 nodes, class weights, batch normalization

[ ]:
```
# 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,validation_data=(Xval,Yval),
  ↪class_weight=class_weights)
```

```
Epoch 1/20
54/54 [==============================] - 0s 7ms/step - loss: 0.2886 - accuracy:
0.8842 - val_loss: 0.4775 - val_accuracy: 0.8404
Epoch 2/20
54/54 [==============================] - 0s 5ms/step - loss: 0.2081 - accuracy:
0.9055 - val_loss: 0.3712 - val_accuracy: 0.8404
```

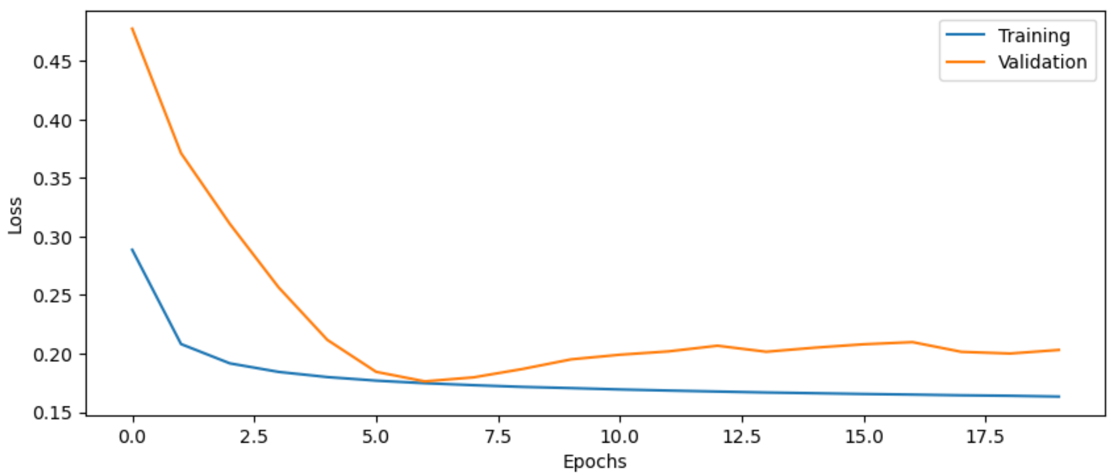
```
Epoch 3/20
54/54 [==============================] - 0s 5ms/step - loss: 0.1917 - accuracy:
0.9108 - val_loss: 0.3108 - val_accuracy: 0.8405
Epoch 4/20
54/54 [==============================] - 0s 5ms/step - loss: 0.1844 - accuracy:
0.9124 - val_loss: 0.2566 - val_accuracy: 0.8511
Epoch 5/20
54/54 [==============================] - 0s 5ms/step - loss: 0.1799 - accuracy:
0.9135 - val_loss: 0.2117 - val_accuracy: 0.8732
Epoch 6/20
54/54 [==============================] - 0s 5ms/step - loss: 0.1769 - accuracy:
0.9141 - val_loss: 0.1844 - val_accuracy: 0.9108
Epoch 7/20
54/54 [==============================] - 0s 5ms/step - loss: 0.1747 - accuracy:
0.9144 - val_loss: 0.1763 - val_accuracy: 0.9167
Epoch 8/20
54/54 [==============================] - 0s 5ms/step - loss: 0.1730 - accuracy:
0.9147 - val_loss: 0.1797 - val_accuracy: 0.9173
Epoch 9/20
54/54 [==============================] - 0s 5ms/step - loss: 0.1716 - accuracy:
0.9149 - val_loss: 0.1868 - val_accuracy: 0.9173
Epoch 10/20
54/54 [==============================] - 0s 5ms/step - loss: 0.1705 - accuracy:
0.9151 - val_loss: 0.1951 - val_accuracy: 0.9172
Epoch 11/20
54/54 [==============================] - 0s 5ms/step - loss: 0.1694 - accuracy:
0.9152 - val_loss: 0.1990 - val_accuracy: 0.9174
Epoch 12/20
54/54 [==============================] - 0s 5ms/step - loss: 0.1684 - accuracy:
0.9153 - val_loss: 0.2019 - val_accuracy: 0.9174
Epoch 13/20
54/54 [==============================] - 0s 5ms/step - loss: 0.1676 - accuracy:
0.9154 - val_loss: 0.2067 - val_accuracy: 0.9175
Epoch 14/20
54/54 [==============================] - 0s 5ms/step - loss: 0.1668 - accuracy:
0.9155 - val_loss: 0.2016 - val_accuracy: 0.9175
Epoch 15/20
54/54 [==============================] - 0s 5ms/step - loss: 0.1662 - accuracy:
0.9156 - val_loss: 0.2051 - val_accuracy: 0.9176
Epoch 16/20
54/54 [==============================] - 0s 5ms/step - loss: 0.1655 - accuracy:
0.9157 - val_loss: 0.2080 - val_accuracy: 0.9176
Epoch 17/20
54/54 [==============================] - 0s 5ms/step - loss: 0.1650 - accuracy:
0.9158 - val_loss: 0.2098 - val_accuracy: 0.9177
Epoch 18/20
54/54 [==============================] - 0s 5ms/step - loss: 0.1644 - accuracy:
0.9158 - val_loss: 0.2015 - val_accuracy: 0.9179
```

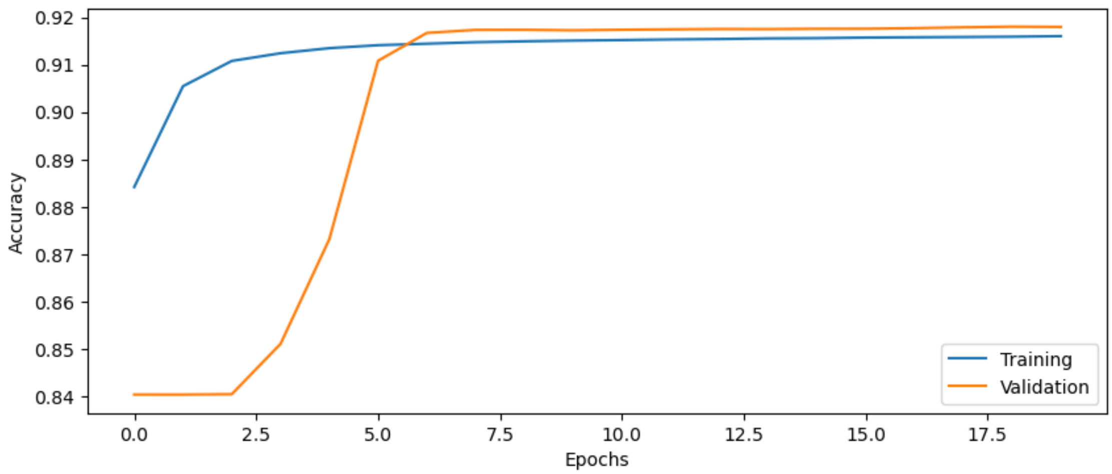
```
Epoch 19/20
54/54 [==============================] - 0s 5ms/step - loss: 0.1640 - accuracy:
0.9159 - val_loss: 0.2001 - val_accuracy: 0.9180
Epoch 20/20
54/54 [==============================] - 0s 5ms/step - loss: 0.1633 - accuracy:
0.9160 - val_loss: 0.2031 - val_accuracy: 0.9179
```

[ ]:
```
# 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 1ms/step - loss: 0.2061 - accuracy:
0.9164
Test loss: 0.2061
Test accuracy: 0.9164
```

[ ]:
```
plot_results(history6)
```





17 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/

17.0.1 2 layers, 20 nodes, class weights, ReLU, no batch normalization

[ ]:
```
# 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,validation_data=(Xval,Yval),␣
 ↪class_weight=class_weights)
```

```
Epoch 1/20
54/54 [==============================] - 1s 11ms/step - loss: 0.3301 - accuracy:
0.8771 - val_loss: 0.2632 - val_accuracy: 0.8828
Epoch 2/20
54/54 [==============================] - 0s 7ms/step - loss: 0.2081 - accuracy:
0.8843 - val_loss: 0.2489 - val_accuracy: 0.8904
Epoch 3/20
54/54 [==============================] - 0s 8ms/step - loss: 0.1971 - accuracy:
```

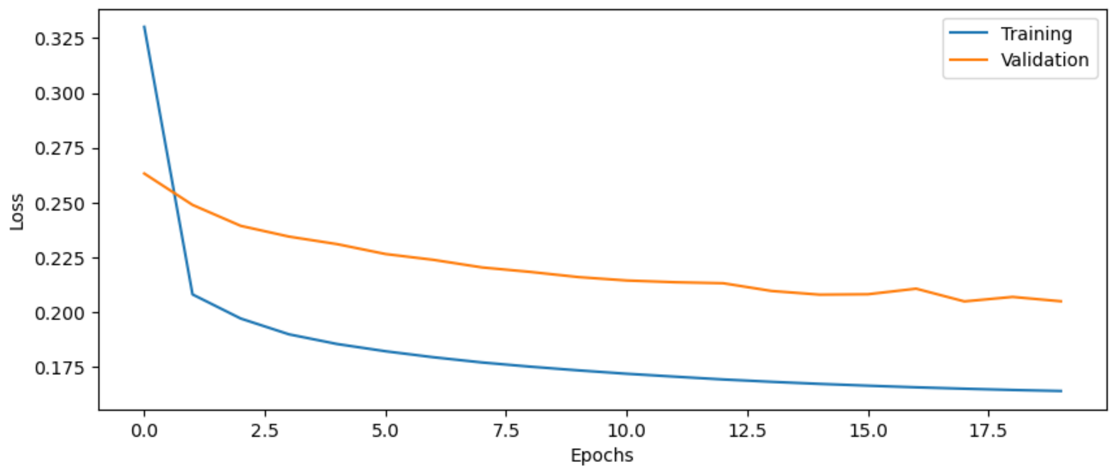
```
0.8929 - val_loss: 0.2393 - val_accuracy: 0.8979
Epoch 4/20
54/54 [==============================] - 0s 8ms/step - loss: 0.1899 - accuracy:
0.8983 - val_loss: 0.2344 - val_accuracy: 0.9010
Epoch 5/20
54/54 [==============================] - 0s 7ms/step - loss: 0.1854 - accuracy:
0.9015 - val_loss: 0.2310 - val_accuracy: 0.9063
Epoch 6/20
54/54 [==============================] - 0s 7ms/step - loss: 0.1821 - accuracy:
0.9059 - val_loss: 0.2265 - val_accuracy: 0.9083
Epoch 7/20
54/54 [==============================] - 0s 3ms/step - loss: 0.1794 - accuracy:
0.9074 - val_loss: 0.2238 - val_accuracy: 0.9094
Epoch 8/20
54/54 [==============================] - 0s 3ms/step - loss: 0.1770 - accuracy:
0.9082 - val_loss: 0.2204 - val_accuracy: 0.9100
Epoch 9/20
54/54 [==============================] - 0s 3ms/step - loss: 0.1751 - accuracy:
0.9095 - val_loss: 0.2183 - val_accuracy: 0.9127
Epoch 10/20
54/54 [==============================] - 0s 3ms/step - loss: 0.1734 - accuracy:
0.9117 - val_loss: 0.2160 - val_accuracy: 0.9136
Epoch 11/20
54/54 [==============================] - 0s 3ms/step - loss: 0.1719 - accuracy:
0.9123 - val_loss: 0.2144 - val_accuracy: 0.9141
Epoch 12/20
54/54 [==============================] - 0s 3ms/step - loss: 0.1705 - accuracy:
0.9126 - val_loss: 0.2136 - val_accuracy: 0.9143
Epoch 13/20
54/54 [==============================] - 0s 3ms/step - loss: 0.1693 - accuracy:
0.9130 - val_loss: 0.2132 - val_accuracy: 0.9145
Epoch 14/20
54/54 [==============================] - 0s 3ms/step - loss: 0.1682 - accuracy:
0.9133 - val_loss: 0.2096 - val_accuracy: 0.9152
Epoch 15/20
54/54 [==============================] - 0s 3ms/step - loss: 0.1672 - accuracy:
0.9144 - val_loss: 0.2080 - val_accuracy: 0.9169
Epoch 16/20
54/54 [==============================] - 0s 3ms/step - loss: 0.1664 - accuracy:
0.9152 - val_loss: 0.2081 - val_accuracy: 0.9172
Epoch 17/20
54/54 [==============================] - 0s 3ms/step - loss: 0.1657 - accuracy:
0.9155 - val_loss: 0.2107 - val_accuracy: 0.9172
Epoch 18/20
54/54 [==============================] - 0s 3ms/step - loss: 0.1650 - accuracy:
0.9156 - val_loss: 0.2049 - val_accuracy: 0.9176
Epoch 19/20
54/54 [==============================] - 0s 3ms/step - loss: 0.1645 - accuracy:
```

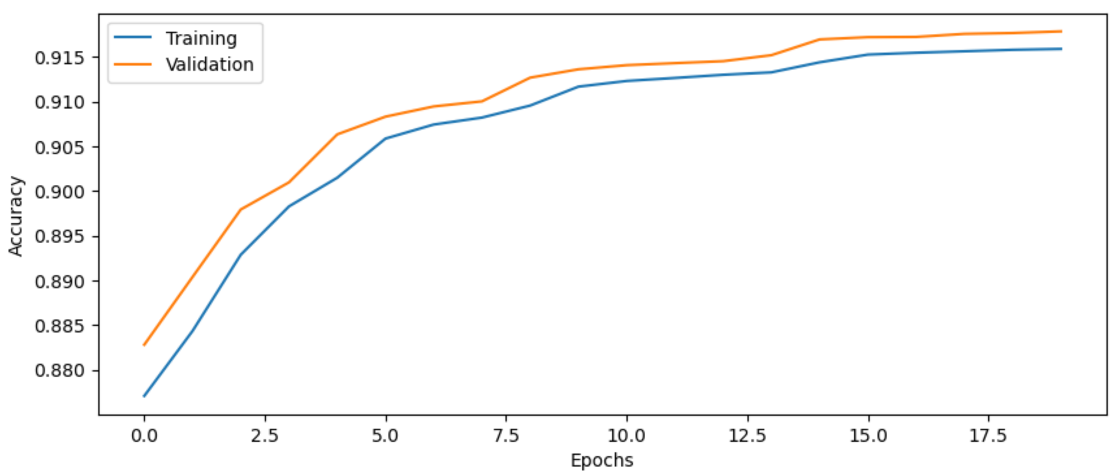
```
0.9158 - val_loss: 0.2069 - val_accuracy: 0.9177
Epoch 20/20
54/54 [==============================] - 0s 3ms/step - loss: 0.1640 - accuracy:
0.9159 - val_loss: 0.2049 - val_accuracy: 0.9178
```

[ ]:
```
# 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 1ms/step - loss: 0.2078 - accuracy:
0.9164
Test loss: 0.2078
Test accuracy: 0.9164
```

[ ]:
```
plot_results(history7)
```





18 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/

18.0.1 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

# 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,validation_data=(Xval,Yval),␣
 ↪class_weight=class_weights)
```

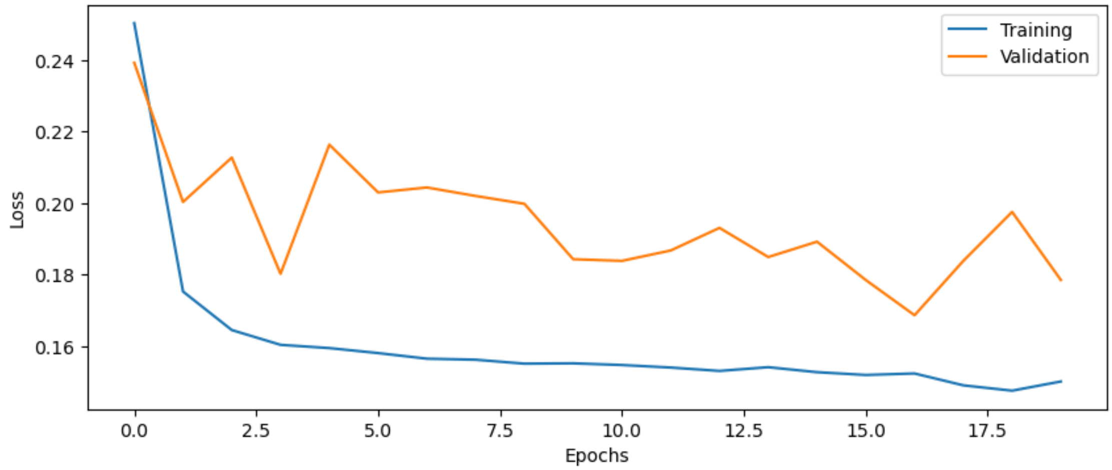
```
Epoch 1/20
54/54 [==============================] - 0s 7ms/step - loss: 0.2504 - accuracy:
0.8569 - val_loss: 0.2393 - val_accuracy: 0.9037
Epoch 2/20
54/54 [==============================] - 0s 3ms/step - loss: 0.1753 - accuracy:
0.9109 - val_loss: 0.2003 - val_accuracy: 0.9186
Epoch 3/20
54/54 [==============================] - 0s 3ms/step - loss: 0.1645 - accuracy:
0.9173 - val_loss: 0.2128 - val_accuracy: 0.9199
Epoch 4/20
54/54 [==============================] - 0s 3ms/step - loss: 0.1604 - accuracy:
0.9193 - val_loss: 0.1802 - val_accuracy: 0.9207
Epoch 5/20
54/54 [==============================] - 0s 3ms/step - loss: 0.1595 - accuracy:
0.9194 - val_loss: 0.2164 - val_accuracy: 0.9207
Epoch 6/20
54/54 [==============================] - 0s 3ms/step - loss: 0.1581 - accuracy:
0.9197 - val_loss: 0.2030 - val_accuracy: 0.9220
Epoch 7/20
54/54 [==============================] - 0s 3ms/step - loss: 0.1565 - accuracy:
0.9203 - val_loss: 0.2044 - val_accuracy: 0.9216
Epoch 8/20
54/54 [==============================] - 0s 3ms/step - loss: 0.1562 - accuracy:
```

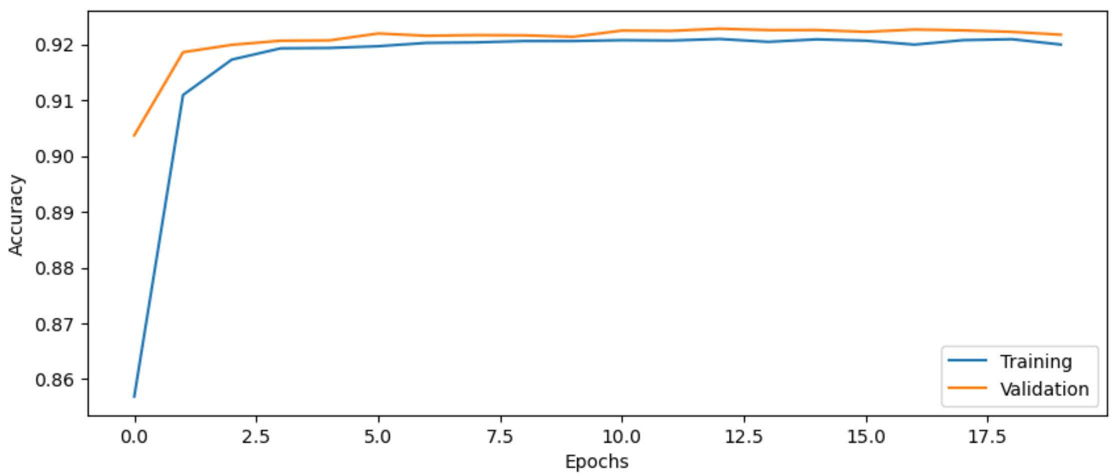
```
0.9204 - val_loss: 0.2020 - val_accuracy: 0.9217
Epoch 9/20
54/54 [==============================] - 0s 3ms/step - loss: 0.1551 - accuracy:
0.9206 - val_loss: 0.1998 - val_accuracy: 0.9216
Epoch 10/20
54/54 [==============================] - 0s 3ms/step - loss: 0.1552 - accuracy:
0.9206 - val_loss: 0.1843 - val_accuracy: 0.9214
Epoch 11/20
54/54 [==============================] - 0s 3ms/step - loss: 0.1547 - accuracy:
0.9208 - val_loss: 0.1839 - val_accuracy: 0.9225
Epoch 12/20
54/54 [==============================] - 0s 3ms/step - loss: 0.1540 - accuracy:
0.9207 - val_loss: 0.1868 - val_accuracy: 0.9224
Epoch 13/20
54/54 [==============================] - 0s 3ms/step - loss: 0.1531 - accuracy:
0.9210 - val_loss: 0.1931 - val_accuracy: 0.9228
Epoch 14/20
54/54 [==============================] - 0s 3ms/step - loss: 0.1541 - accuracy:
0.9205 - val_loss: 0.1849 - val_accuracy: 0.9226
Epoch 15/20
54/54 [==============================] - 0s 3ms/step - loss: 0.1527 - accuracy:
0.9209 - val_loss: 0.1892 - val_accuracy: 0.9226
Epoch 16/20
54/54 [==============================] - 0s 3ms/step - loss: 0.1519 - accuracy:
0.9207 - val_loss: 0.1785 - val_accuracy: 0.9222
Epoch 17/20
54/54 [==============================] - 0s 3ms/step - loss: 0.1524 - accuracy:
0.9200 - val_loss: 0.1686 - val_accuracy: 0.9227
Epoch 18/20
54/54 [==============================] - 0s 3ms/step - loss: 0.1490 - accuracy:
0.9208 - val_loss: 0.1839 - val_accuracy: 0.9225
Epoch 19/20
54/54 [==============================] - 0s 3ms/step - loss: 0.1476 - accuracy:
0.9209 - val_loss: 0.1975 - val_accuracy: 0.9222
Epoch 20/20
54/54 [==============================] - 0s 3ms/step - loss: 0.1501 - accuracy:
0.9200 - val_loss: 0.1785 - val_accuracy: 0.9217
```

[ ]:
```python
# 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: 0.1805 - accuracy:
0.9203
Test loss: 0.1805
Test accuracy: 0.9203
```

```
[ ]: plot_results(history8)
```





19 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? - After adding dropout, the validation accuracy decrease first and then increase in most cases.

Question 16: How does the test accuracy change when adding dropout? - The test accuracy will increase, expecially the original mode is overfitting

19.0.1 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

# 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,validation_data=(Xval,Yval),␣
 ↪class_weight=class_weights)
```

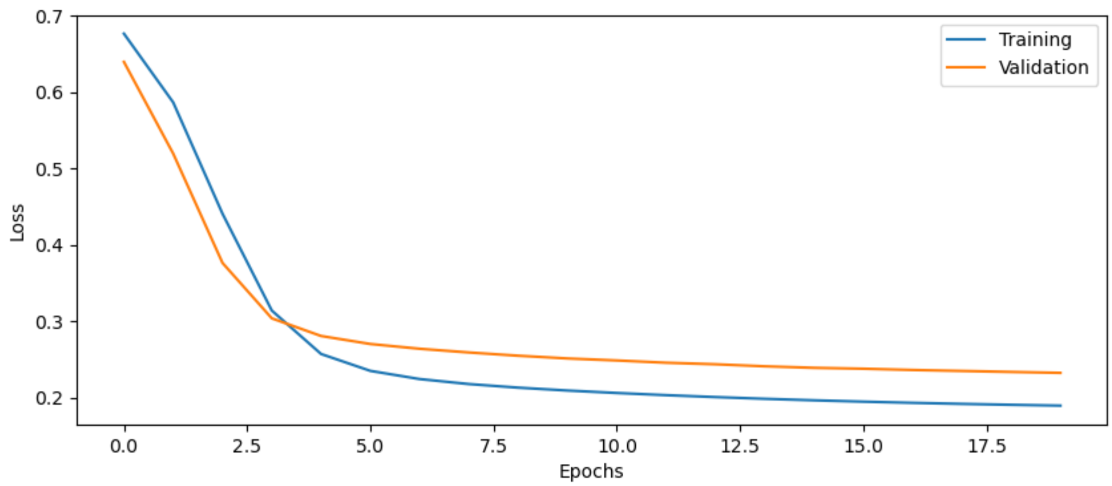
```
Epoch 1/20
54/54 [==============================] - 0s 5ms/step - loss: 0.6769 - accuracy:
0.6609 - val_loss: 0.6398 - val_accuracy: 0.8777
Epoch 2/20
54/54 [==============================] - 0s 3ms/step - loss: 0.5867 - accuracy:
0.8791 - val_loss: 0.5196 - val_accuracy: 0.8768
Epoch 3/20
54/54 [==============================] - 0s 3ms/step - loss: 0.4409 - accuracy:
0.8767 - val_loss: 0.3764 - val_accuracy: 0.8806
Epoch 4/20
54/54 [==============================] - 0s 3ms/step - loss: 0.3140 - accuracy:
0.8780 - val_loss: 0.3038 - val_accuracy: 0.8808
Epoch 5/20
54/54 [==============================] - 0s 3ms/step - loss: 0.2570 - accuracy:
0.8796 - val_loss: 0.2806 - val_accuracy: 0.8823
Epoch 6/20
54/54 [==============================] - 0s 3ms/step - loss: 0.2349 - accuracy:
0.8818 - val_loss: 0.2701 - val_accuracy: 0.8846
Epoch 7/20
54/54 [==============================] - 0s 3ms/step - loss: 0.2242 - accuracy:
0.8839 - val_loss: 0.2639 - val_accuracy: 0.8866
Epoch 8/20
54/54 [==============================] - 0s 3ms/step - loss: 0.2176 - accuracy:
0.8857 - val_loss: 0.2590 - val_accuracy: 0.8888
Epoch 9/20
54/54 [==============================] - 0s 3ms/step - loss: 0.2129 - accuracy:
0.8879 - val_loss: 0.2547 - val_accuracy: 0.8909
```

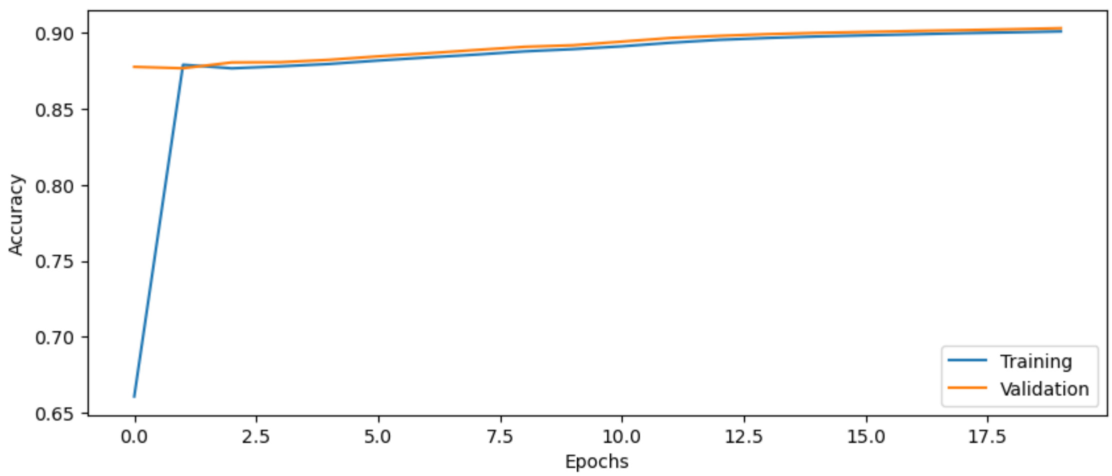
```
Epoch 10/20
54/54 [==============================] - 0s 3ms/step - loss: 0.2091 - accuracy:
0.8894 - val_loss: 0.2512 - val_accuracy: 0.8919
Epoch 11/20
54/54 [==============================] - 0s 3ms/step - loss: 0.2059 - accuracy:
0.8912 - val_loss: 0.2485 - val_accuracy: 0.8943
Epoch 12/20
54/54 [==============================] - 0s 3ms/step - loss: 0.2031 - accuracy:
0.8936 - val_loss: 0.2456 - val_accuracy: 0.8967
Epoch 13/20
54/54 [==============================] - 0s 3ms/step - loss: 0.2005 - accuracy:
0.8955 - val_loss: 0.2435 - val_accuracy: 0.8981
Epoch 14/20
54/54 [==============================] - 0s 3ms/step - loss: 0.1983 - accuracy:
0.8967 - val_loss: 0.2409 - val_accuracy: 0.8992
Epoch 15/20
54/54 [==============================] - 0s 3ms/step - loss: 0.1963 - accuracy:
0.8977 - val_loss: 0.2389 - val_accuracy: 0.9000
Epoch 16/20
54/54 [==============================] - 0s 3ms/step - loss: 0.1946 - accuracy:
0.8984 - val_loss: 0.2377 - val_accuracy: 0.9007
Epoch 17/20
54/54 [==============================] - 0s 3ms/step - loss: 0.1930 - accuracy:
0.8991 - val_loss: 0.2361 - val_accuracy: 0.9013
Epoch 18/20
54/54 [==============================] - 0s 3ms/step - loss: 0.1916 - accuracy:
0.8998 - val_loss: 0.2348 - val_accuracy: 0.9018
Epoch 19/20
54/54 [==============================] - 0s 3ms/step - loss: 0.1904 - accuracy:
0.9004 - val_loss: 0.2335 - val_accuracy: 0.9024
Epoch 20/20
54/54 [==============================] - 0s 3ms/step - loss: 0.1894 - accuracy:
0.9010 - val_loss: 0.2323 - val_accuracy: 0.9031
```

[ ]:
```
# 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: 0.2359 - accuracy:
0.9011
Test loss: 0.2359
Test accuracy: 0.9011
```

[ ]:
```
plot_results(history9)
```





20 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: 0.9036. We added more epochs(40) and layers(5) with nodes(80) for a better performance, and use a 0.7 dropout rate to mitigate overfitting.

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

batch_size = 10000
```

```
epochs = 40
input_shape = X.shape

# Build and train DNN
model10 = build_DNN(input_shape, n_layers=5, n_nodes=80 ,use_dropout=0.7,␣
↪use_bn=True)

history10 = model10.fit(Xtrain, Ytrain, epochs=epochs,␣
↪batch_size=batch_size,validation_data=(Xval,Yval))
```

```
Epoch 1/40
54/54 [==============================] - 3s 64ms/step - loss: 0.6623 - accuracy:
0.7046 - val_loss: 0.4401 - val_accuracy: 0.8404
Epoch 2/40
54/54 [==============================] - 3s 63ms/step - loss: 0.3991 - accuracy:
0.8404 - val_loss: 0.4309 - val_accuracy: 0.8404
Epoch 3/40
54/54 [==============================] - 3s 58ms/step - loss: 0.3400 - accuracy:
0.8473 - val_loss: 0.3763 - val_accuracy: 0.8404
Epoch 4/40
54/54 [==============================] - 3s 58ms/step - loss: 0.3030 - accuracy:
0.8539 - val_loss: 0.2722 - val_accuracy: 0.8404
Epoch 5/40
54/54 [==============================] - 3s 58ms/step - loss: 0.2811 - accuracy:
0.8578 - val_loss: 0.2147 - val_accuracy: 0.8973
Epoch 6/40
54/54 [==============================] - 3s 58ms/step - loss: 0.2675 - accuracy:
0.8602 - val_loss: 0.2058 - val_accuracy: 0.8817
Epoch 7/40
54/54 [==============================] - 3s 57ms/step - loss: 0.2592 - accuracy:
0.8606 - val_loss: 0.2103 - val_accuracy: 0.8814
Epoch 8/40
54/54 [==============================] - 3s 57ms/step - loss: 0.2517 - accuracy:
0.8622 - val_loss: 0.2177 - val_accuracy: 0.8812
Epoch 9/40
54/54 [==============================] - 5s 84ms/step - loss: 0.2461 - accuracy:
0.8631 - val_loss: 0.2217 - val_accuracy: 0.8811
Epoch 10/40
54/54 [==============================] - 5s 98ms/step - loss: 0.2427 - accuracy:
0.8635 - val_loss: 0.2251 - val_accuracy: 0.8810
Epoch 11/40
54/54 [==============================] - 5s 101ms/step - loss: 0.2389 -
accuracy: 0.8645 - val_loss: 0.2225 - val_accuracy: 0.8810
Epoch 12/40
54/54 [==============================] - 5s 99ms/step - loss: 0.2364 - accuracy:
0.8644 - val_loss: 0.2212 - val_accuracy: 0.8810
Epoch 13/40
54/54 [==============================] - 3s 60ms/step - loss: 0.2335 - accuracy:
```

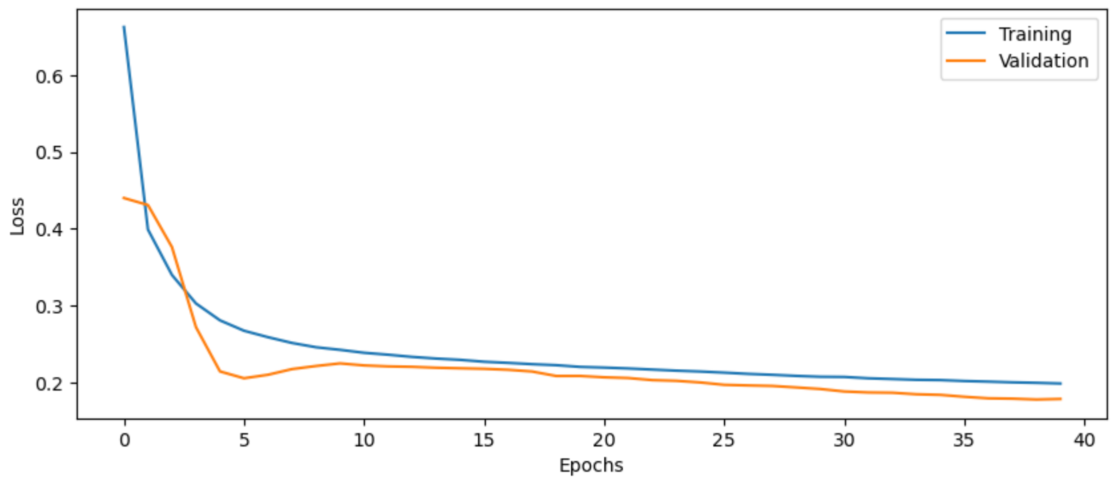
```
0.8654 - val_loss: 0.2206 - val_accuracy: 0.8809
Epoch 14/40
54/54 [==============================] - 3s 60ms/step - loss: 0.2314 - accuracy:
0.8657 - val_loss: 0.2194 - val_accuracy: 0.8810
Epoch 15/40
54/54 [==============================] - 3s 59ms/step - loss: 0.2297 - accuracy:
0.8664 - val_loss: 0.2187 - val_accuracy: 0.8810
Epoch 16/40
54/54 [==============================] - 3s 60ms/step - loss: 0.2272 - accuracy:
0.8675 - val_loss: 0.2181 - val_accuracy: 0.8811
Epoch 17/40
54/54 [==============================] - 3s 60ms/step - loss: 0.2258 - accuracy:
0.8679 - val_loss: 0.2168 - val_accuracy: 0.8811
Epoch 18/40
54/54 [==============================] - 3s 61ms/step - loss: 0.2241 - accuracy:
0.8686 - val_loss: 0.2146 - val_accuracy: 0.8824
Epoch 19/40
54/54 [==============================] - 3s 57ms/step - loss: 0.2228 - accuracy:
0.8696 - val_loss: 0.2087 - val_accuracy: 0.8844
Epoch 20/40
54/54 [==============================] - 3s 59ms/step - loss: 0.2205 - accuracy:
0.8706 - val_loss: 0.2088 - val_accuracy: 0.8858
Epoch 21/40
54/54 [==============================] - 3s 58ms/step - loss: 0.2196 - accuracy:
0.8709 - val_loss: 0.2070 - val_accuracy: 0.8864
Epoch 22/40
54/54 [==============================] - 3s 61ms/step - loss: 0.2185 - accuracy:
0.8710 - val_loss: 0.2061 - val_accuracy: 0.8871
Epoch 23/40
54/54 [==============================] - 3s 61ms/step - loss: 0.2171 - accuracy:
0.8730 - val_loss: 0.2034 - val_accuracy: 0.8883
Epoch 24/40
54/54 [==============================] - 3s 59ms/step - loss: 0.2156 - accuracy:
0.8726 - val_loss: 0.2024 - val_accuracy: 0.8900
Epoch 25/40
54/54 [==============================] - 3s 59ms/step - loss: 0.2145 - accuracy:
0.8742 - val_loss: 0.2003 - val_accuracy: 0.8912
Epoch 26/40
54/54 [==============================] - 3s 59ms/step - loss: 0.2131 - accuracy:
0.8758 - val_loss: 0.1972 - val_accuracy: 0.8928
Epoch 27/40
54/54 [==============================] - 3s 59ms/step - loss: 0.2115 - accuracy:
0.8762 - val_loss: 0.1964 - val_accuracy: 0.8941
Epoch 28/40
54/54 [==============================] - 3s 59ms/step - loss: 0.2103 - accuracy:
0.8769 - val_loss: 0.1958 - val_accuracy: 0.8952
Epoch 29/40
54/54 [==============================] - 3s 59ms/step - loss: 0.2088 - accuracy:
```

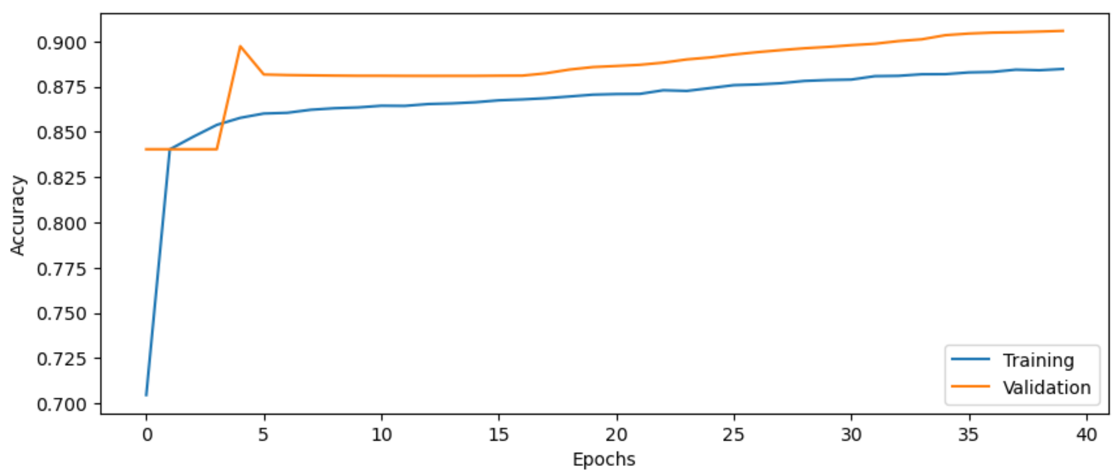
```
0.8782 - val_loss: 0.1938 - val_accuracy: 0.8962
Epoch 30/40
54/54 [==============================] - 3s 59ms/step - loss: 0.2077 - accuracy:
0.8787 - val_loss: 0.1918 - val_accuracy: 0.8970
Epoch 31/40
54/54 [==============================] - 3s 59ms/step - loss: 0.2075 - accuracy:
0.8789 - val_loss: 0.1886 - val_accuracy: 0.8979
Epoch 32/40
54/54 [==============================] - 3s 58ms/step - loss: 0.2057 - accuracy:
0.8808 - val_loss: 0.1873 - val_accuracy: 0.8987
Epoch 33/40
54/54 [==============================] - 3s 57ms/step - loss: 0.2048 - accuracy:
0.8810 - val_loss: 0.1870 - val_accuracy: 0.9002
Epoch 34/40
54/54 [==============================] - 3s 59ms/step - loss: 0.2038 - accuracy:
0.8819 - val_loss: 0.1850 - val_accuracy: 0.9012
Epoch 35/40
54/54 [==============================] - 3s 58ms/step - loss: 0.2034 - accuracy:
0.8819 - val_loss: 0.1842 - val_accuracy: 0.9034
Epoch 36/40
54/54 [==============================] - 3s 59ms/step - loss: 0.2022 - accuracy:
0.8829 - val_loss: 0.1817 - val_accuracy: 0.9043
Epoch 37/40
54/54 [==============================] - 3s 61ms/step - loss: 0.2014 - accuracy:
0.8831 - val_loss: 0.1797 - val_accuracy: 0.9049
Epoch 38/40
54/54 [==============================] - 3s 61ms/step - loss: 0.2005 - accuracy:
0.8844 - val_loss: 0.1793 - val_accuracy: 0.9051
Epoch 39/40
54/54 [==============================] - 3s 57ms/step - loss: 0.1998 - accuracy:
0.8841 - val_loss: 0.1782 - val_accuracy: 0.9054
Epoch 40/40
54/54 [==============================] - 3s 59ms/step - loss: 0.1990 - accuracy:
0.8848 - val_loss: 0.1789 - val_accuracy: 0.9058
```

[ ]:

```
# 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 6ms/step - loss: 0.1811 - accuracy:
0.9036
Test loss: 0.1811
Test accuracy: 0.9036
```





21 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
```

21.0.1 Your best config, custom dropout

```
[ ]: # Your best training parameters
     # the previous best training parameters took too much time so we change it a␣
      ↪little bit
     batch_size = 10000
```

```python
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.
 ↪7, use_bn = True)

history11 = model11.fit(Xtrain, Ytrain, epochs=epochs, batch_size=batch_size,␣
 ↪validation_data=(Xval,Yval))
```

```
Epoch 1/40
54/54 [==============================] - 2s 34ms/step - loss: 0.5264 - accuracy:
0.8144 - val_loss: 0.4249 - val_accuracy: 0.8405
Epoch 2/40
54/54 [==============================] - 1s 20ms/step - loss: 0.2933 - accuracy:
0.8786 - val_loss: 0.3825 - val_accuracy: 0.8404
Epoch 3/40
54/54 [==============================] - 1s 20ms/step - loss: 0.2584 - accuracy:
0.8798 - val_loss: 0.3452 - val_accuracy: 0.8407
Epoch 4/40
54/54 [==============================] - 1s 20ms/step - loss: 0.2428 - accuracy:
0.8807 - val_loss: 0.3032 - val_accuracy: 0.8431
Epoch 5/40
54/54 [==============================] - 1s 20ms/step - loss: 0.2337 - accuracy:
0.8821 - val_loss: 0.2669 - val_accuracy: 0.8525
Epoch 6/40
54/54 [==============================] - 1s 20ms/step - loss: 0.2275 - accuracy:
0.8827 - val_loss: 0.2421 - val_accuracy: 0.8638
Epoch 7/40
54/54 [==============================] - 1s 20ms/step - loss: 0.2226 - accuracy:
0.8838 - val_loss: 0.2275 - val_accuracy: 0.8721
Epoch 8/40
54/54 [==============================] - 1s 19ms/step - loss: 0.2188 - accuracy:
0.8842 - val_loss: 0.2179 - val_accuracy: 0.8800
Epoch 9/40
54/54 [==============================] - 1s 20ms/step - loss: 0.2156 - accuracy:
0.8850 - val_loss: 0.2133 - val_accuracy: 0.8835
Epoch 10/40
54/54 [==============================] - 1s 20ms/step - loss: 0.2130 - accuracy:
0.8866 - val_loss: 0.2103 - val_accuracy: 0.8856
Epoch 11/40
54/54 [==============================] - 1s 19ms/step - loss: 0.2107 - accuracy:
0.8865 - val_loss: 0.2084 - val_accuracy: 0.8854
Epoch 12/40
54/54 [==============================] - 1s 19ms/step - loss: 0.2087 - accuracy:
0.8869 - val_loss: 0.2062 - val_accuracy: 0.8878
Epoch 13/40
54/54 [==============================] - 1s 19ms/step - loss: 0.2067 - accuracy:
```

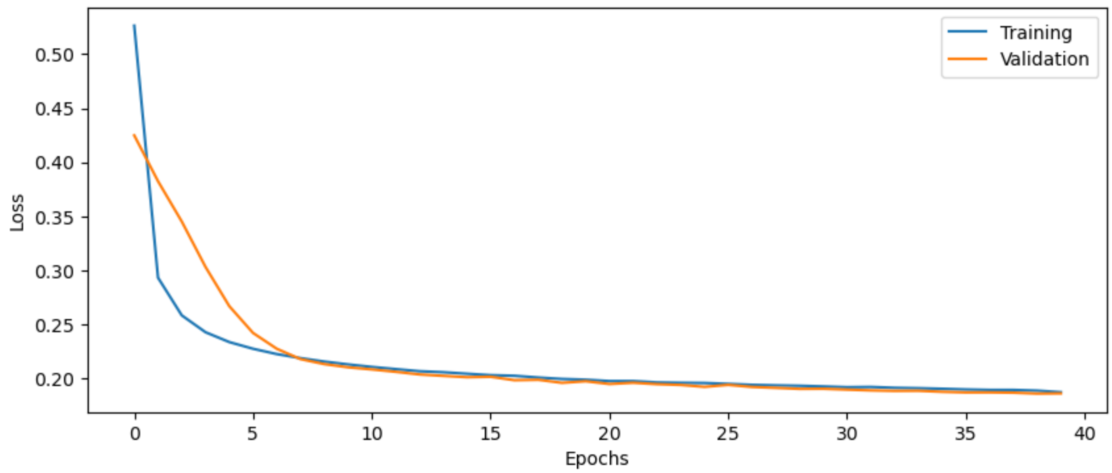
```
0.8881 - val_loss: 0.2037 - val_accuracy: 0.8899
Epoch 14/40
54/54 [==============================] - 1s 20ms/step - loss: 0.2058 - accuracy:
0.8882 - val_loss: 0.2024 - val_accuracy: 0.8904
Epoch 15/40
54/54 [==============================] - 1s 20ms/step - loss: 0.2044 - accuracy:
0.8889 - val_loss: 0.2012 - val_accuracy: 0.8905
Epoch 16/40
54/54 [==============================] - 1s 20ms/step - loss: 0.2031 - accuracy:
0.8893 - val_loss: 0.2016 - val_accuracy: 0.8901
Epoch 17/40
54/54 [==============================] - 1s 20ms/step - loss: 0.2025 - accuracy:
0.8891 - val_loss: 0.1984 - val_accuracy: 0.8918
Epoch 18/40
54/54 [==============================] - 1s 19ms/step - loss: 0.2009 - accuracy:
0.8900 - val_loss: 0.1988 - val_accuracy: 0.8912
Epoch 19/40
54/54 [==============================] - 1s 19ms/step - loss: 0.1996 - accuracy:
0.8906 - val_loss: 0.1960 - val_accuracy: 0.8923
Epoch 20/40
54/54 [==============================] - 1s 20ms/step - loss: 0.1989 - accuracy:
0.8909 - val_loss: 0.1974 - val_accuracy: 0.8916
Epoch 21/40
54/54 [==============================] - 1s 19ms/step - loss: 0.1976 - accuracy:
0.8913 - val_loss: 0.1949 - val_accuracy: 0.8930
Epoch 22/40
54/54 [==============================] - 1s 19ms/step - loss: 0.1976 - accuracy:
0.8915 - val_loss: 0.1961 - val_accuracy: 0.8911
Epoch 23/40
54/54 [==============================] - 1s 19ms/step - loss: 0.1964 - accuracy:
0.8918 - val_loss: 0.1947 - val_accuracy: 0.8931
Epoch 24/40
54/54 [==============================] - 1s 20ms/step - loss: 0.1960 - accuracy:
0.8918 - val_loss: 0.1940 - val_accuracy: 0.8938
Epoch 25/40
54/54 [==============================] - 1s 20ms/step - loss: 0.1958 - accuracy:
0.8917 - val_loss: 0.1923 - val_accuracy: 0.8934
Epoch 26/40
54/54 [==============================] - 1s 20ms/step - loss: 0.1950 - accuracy:
0.8923 - val_loss: 0.1941 - val_accuracy: 0.8920
Epoch 27/40
54/54 [==============================] - 1s 19ms/step - loss: 0.1941 - accuracy:
0.8928 - val_loss: 0.1921 - val_accuracy: 0.8946
Epoch 28/40
54/54 [==============================] - 1s 20ms/step - loss: 0.1936 - accuracy:
0.8927 - val_loss: 0.1913 - val_accuracy: 0.8944
Epoch 29/40
54/54 [==============================] - 1s 20ms/step - loss: 0.1933 - accuracy:
```

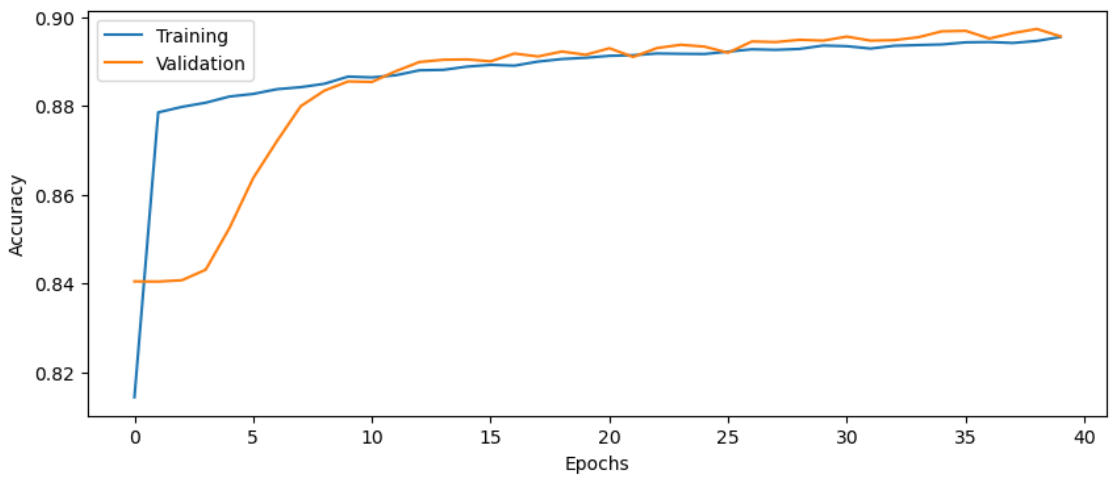
```
0.8929 - val_loss: 0.1904 - val_accuracy: 0.8949
Epoch 30/40
54/54 [==============================] - 1s 20ms/step - loss: 0.1927 - accuracy:
0.8936 - val_loss: 0.1905 - val_accuracy: 0.8947
Epoch 31/40
54/54 [==============================] - 1s 20ms/step - loss: 0.1919 - accuracy:
0.8935 - val_loss: 0.1897 - val_accuracy: 0.8956
Epoch 32/40
54/54 [==============================] - 1s 19ms/step - loss: 0.1921 - accuracy:
0.8930 - val_loss: 0.1890 - val_accuracy: 0.8947
Epoch 33/40
54/54 [==============================] - 1s 20ms/step - loss: 0.1914 - accuracy:
0.8936 - val_loss: 0.1885 - val_accuracy: 0.8949
Epoch 34/40
54/54 [==============================] - 1s 20ms/step - loss: 0.1910 - accuracy:
0.8937 - val_loss: 0.1886 - val_accuracy: 0.8955
Epoch 35/40
54/54 [==============================] - 1s 20ms/step - loss: 0.1905 - accuracy:
0.8939 - val_loss: 0.1877 - val_accuracy: 0.8968
Epoch 36/40
54/54 [==============================] - 1s 19ms/step - loss: 0.1898 - accuracy:
0.8944 - val_loss: 0.1871 - val_accuracy: 0.8969
Epoch 37/40
54/54 [==============================] - 1s 20ms/step - loss: 0.1894 - accuracy:
0.8944 - val_loss: 0.1871 - val_accuracy: 0.8952
Epoch 38/40
54/54 [==============================] - 1s 20ms/step - loss: 0.1894 - accuracy:
0.8942 - val_loss: 0.1869 - val_accuracy: 0.8965
Epoch 39/40
54/54 [==============================] - 1s 20ms/step - loss: 0.1888 - accuracy:
0.8947 - val_loss: 0.1860 - val_accuracy: 0.8974
Epoch 40/40
54/54 [==============================] - 1s 19ms/step - loss: 0.1873 - accuracy:
0.8956 - val_loss: 0.1862 - val_accuracy: 0.8957
```

[ ]:
```python
# 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(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: 0.1867 - accuracy:
```

```
0.8955
Test accuracy: 0.8955
crossentropy: 0.1867
```





[ ]:
```python
# 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])
```

[ ]:
```python
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 = 0.8955491578578949
std = 0.0006123434980572862
```

22 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.

[ ]:
```
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',␣
 ↪classes=np.unique(Y), y=Ytrain)

    # Rebuild the DNN model, to not continue training on the previously trained␣
 ↪model
    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_size=batch_size, validation_data=(Xval,Yval),␣
 ↪class_weight=class_weights)
```

```python

    # 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])
```

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

```
mean = 0.9041755616664886
std = 0.002647071085607778
```

23 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) - and change the loss funcion into regression losses such as MeanSquaredError

23.1 Report

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