DNN_Lab

May 20, 2020

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.

2 Part 1: Get the data

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

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.

[1]:
```
import os
import warnings

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

import keras.backend as K
import tensorflow as tf

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

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

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

```
Using TensorFlow backend.
```

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.

Question 1: What graphics card is available in the cloud machine? Run 'nvidia-smi' in the terminal.

Question 2: Google the name of the graphics card, how many CUDA cores does it have?

Question 3: How much memory does the graphics card have?

Question 4: What is stored in the GPU memory while training a DNN ?

Question 5: What CPU is available in the cloud machine? How many cores does it have? Run 'lscpu' in the terminal.

Question 6: How much CPU memory (RAM) is available in the cloud machine? Run 'free -g' in the terminal.

Answer 1: The graphic card used is Tesla K80 in cloud machine.

Answer 2: The graphic card uses 4992 Nvidia cuda cores

Answer 3: The graphic card memory is 24 GB of DDR5

Answer 4:

Answer 5: The CPU model name is Intel(R) Xeon(R) CPU E5-2690 v3 @ 2.60GHz. It has six cores.

Answer 6: The CPU memory(RAM) available is 55GB.

5 Part 4: Load the data

Load the dataset from the csv files, it will take some time since it is almost 1.4 GB.

We will use the function `genfromtxt` to load the data.

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.

[2]:
```
from numpy import genfromtxt
import numpy as np
import pandas as pd

# Load data from file
# X = covariates, Y = labels
X = pd.read_csv("Mirai_dataset.csv")
Y = pd.read_csv("mirai_labels.csv")

# Save data as numpy arrays, for faster loading in future calls to this cell
np.save('Mirai_data.npy', X)
np.save('Mirai_labels.npy', Y)


# Load data from numpy arrays, for faster loading
X = np.load('Mirai_data.npy')
Y = np.load('Mirai_labels.npy')

# Remove the first 24 covariates
X = np.delete(X,np.s_[0:24], axis=1)

# Print the number of examples of each class
print('The covariates have size {}.'.format(X.shape))
print('The labels have size {}.'.format(Y.shape))
label_0 = len(np.where(Y == 0)[0])
print('The Length of label 0 is {}.'.format(label_0))
label_1 = len(np.where(Y == 1)[0])
print('The Length of label 1 is {}.'.format(label_1))
```

```
The covariates have size (764136, 92).
The labels have size (764136, 1).
The Length of label 0 is 121620.
The Length of label 1 is 642516.
```

6 Part 5: How good is a naive classifier?

Question 7: Given the distribution of examples, 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 random chance.

[3]:
```
# It is common to have NaNs in the data, lets check for it. Hint: np.isnan()

# Print the number of NaNs (not a number) in the labels
nan_Y = np.sum(np.isnan(Y))
print(f"The NaNs in Label or target is {nan_Y}")
# Print the number of NaNs in the covariates
```

```
nan_X = np.sum(np.isnan(X))
print(f"The NaNs in covariates is {nan_X}")
```

```
The NaNs in Label or target is 0
The NaNs in covariates is 0
```

7 Part 6: Preprocessing

Lets do some simple preprocessing

[4]:
```
# Convert covariates to floats
X = X.astype(float)

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

# Remove mean of each covariate (column)
X = X - np.mean(X,axis=0)

# Divide each covariate (column) by its standard deviation
X = X / np.std(X,axis=0)

# Check that mean is 0 and standard deviation is 1 for all covariates, by␣
 ↪printing mean and std
print(f"The mean for all covariates is \n {np.mean(X,axis=0)}")
print(f"The standard deviation for all covariates is \n {np.std(X,axis=0)}")
```

```
The mean for all covariates is
 [ 4.62142524e-18  3.18385513e-16 -1.84633842e-16 -2.48459723e-17
 -3.04697912e-16 -1.17921663e-15 -1.52348956e-16 -1.24043889e-15
  1.71526476e-15 -7.61372834e-17  7.85102969e-16  7.82015820e-18
  2.20377823e-18 -2.35441624e-17  4.64188225e-16  4.78842886e-16
  1.73921806e-16 -1.55644395e-15  8.74072379e-19 -6.54624420e-18
  1.74814476e-18 -1.63507327e-16  2.70702076e-16  8.86718533e-17
 -1.25799472e-15  2.18146149e-17  3.45909495e-18  7.12647948e-17
 -5.62047137e-16 -6.10362883e-16 -6.27100439e-17 -9.82606133e-16
 -2.17774203e-17  1.10095925e-17  4.20298633e-17  5.51967409e-17
  4.07503702e-16 -2.18629679e-16  6.54624420e-18 -1.99362892e-17
 -4.56749311e-17  7.29013559e-17 -1.24058767e-15 -1.48778277e-18
  1.33342531e-17  4.64188225e-16  7.99683241e-18  6.22916050e-17
 -1.62912214e-16 -4.94687772e-18  5.00638903e-17 -5.37535916e-16
 -6.45325778e-18 -1.16790948e-17  2.17216285e-17 -1.72954747e-17
 -7.98195458e-17  4.90075646e-16  2.99193116e-16 -1.90064249e-17
  4.81148949e-16  3.08714925e-18  1.85217332e-18 -1.76674204e-19
  9.23913102e-17 -3.36238907e-17  6.63365144e-17  2.26886873e-17
  1.33900450e-17  1.62610008e-18 -1.90622168e-18  5.89757091e-16
 -1.57221445e-15 -8.92669664e-19 -2.79182437e-16 -2.99602256e-17
```

```
  8.28741498e-19 -1.19952486e-17 -6.20256638e-16 -3.15112391e-16
 -2.30606330e-18  1.67896286e-15  3.39958364e-17  2.64255792e-18
 -5.46760169e-18 -2.18108955e-16 -5.68333019e-17 -4.01701349e-18
  5.85293743e-16 -2.69288682e-17  2.37580312e-18  1.98990946e-18]
The standard deviation for all covariates is
 [1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.
 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.
 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.
 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]
```

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%)

[6]:
```
from sklearn.model_selection import train_test_split

Xtrain,Xtemp,Ytrain,Ytemp=train_test_split(X,Y,test_size=0.3)

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

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

# for training data

print('The covariates have size in training data {}.'.format(Xtrain.shape))
print('The labels have size in training data{}.'.format(Ytrain.shape))
label_0 = len(np.where(Ytrain == 0)[0])
print('The Length of label 0 for training data {}.'.format(label_0))
label_1 = len(np.where(Ytrain == 1)[0])
print('The Length of label 1 for validation data {}.'.format(label_1))

# for Validation data
print('The covariates have size in validation + test data{}.'.format(Xtemp.
 ↪shape))
print('The labels have size in valiadation + test data{}.'.format(Ytemp.shape))
label_0 = len(np.where(Ytemp == 0)[0])
print('The Length of label 0 for validation + test data {}.'.format(label_0))
label_1 = len(np.where(Ytemp == 1)[0])
```

```
print('The Length of label 1 for validation + test data {}.'.format(label_1))
```

```
Xtrain has size (534895, 92).
Ytrain has size (534895, 1).
Xtemp has size (229241, 92).
Ytemp has size (229241, 1).
The covariates have size in training data (534895, 92).
The labels have size in training data(534895, 1).
The Length of label 0 for training data 85263.
The Length of label 1 for validation data 449632.
The covariates have size in validation + test data(229241, 92).
The labels have size in valiadation + test data(229241, 1).
The Length of label 0 for validation + test data 36357.
The Length of label 1 for validation + test data 192884.
```

9 Part 8: Number of examples per class

Question 8: Can we use the dataset as it is? Why not?

Lets randomly shuffle the data, to get some examples of each class in training data and in the remaining 30%. Use the function `shuffle` in scikit learn

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

Answer 8: The data may be in a definite pattern. So it need to be shuffled so that randomness is bought into account.

[7]:
```
from sklearn.utils import shuffle

# Randomly shuffle data, to get both classes in training and testing
X,Y = shuffle(X, Y, random_state=0)

# Divide the data into training and validation/test again
Xtrain,Xtemp,Ytrain,Ytemp=train_test_split(X,Y,test_size=0.3)

# for training data

print('The covariates have size in training data {}.'.format(Xtrain.shape))
print('The labels have size in training data{}.'.format(Ytrain.shape))
label_0 = len(np.where(Ytrain == 0)[0])
print('The Length of label 0 for training data {}.'.format(label_0))
label_1 = len(np.where(Ytrain == 1)[0])
print('The Length of label 1 for validation data {}.'.format(label_1))

# for Validation data
print('The covariates have size in validation + test data{}.'.format(Xtemp.
 ↪shape))
print('The labels have size in valiadation + test data{}.'.format(Ytemp.shape))
```

```
label_0 = len(np.where(Ytemp == 0)[0])
print('The Length of label 0 for validation + test data {}.'.format(label_0))
label_1 = len(np.where(Ytemp == 1)[0])
print('The Length of label 1 for validation + test data {}.'.format(label_1))
```

```
The covariates have size in training data (534895, 92).
The labels have size in training data(534895, 1).
The Length of label 0 for training data 85041.
The Length of label 1 for validation data 449854.
The covariates have size in validation + test data(229241, 92).
The labels have size in valiadation + test data(229241, 1).
The Length of label 0 for validation + test data 36579.
The Length of label 1 for validation + test data 192662.
```

After shuffle, the propotion for label 0 and label 1 was changed and it will increase randomness.

10 Part 9: Split non-training data data into validation and test

Split your non-training data (Xtemp, Ytemp) into 50% validation (Xval, Yval) and 50% testing (Xtest, Ytest), we use a function from scikit learn.

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

[8]:
```
Xval,Xtest,Yval,Ytest=train_test_split(Xtemp,Ytemp,test_size=0.5)

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

```
The validation and test data have size (114620, 92), (114621, 92), (114620, 1)
and (114621, 1)
```

11 Part 10: 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.

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

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

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

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 https://keras.io/models/model/ for how to compile, train and evaluate the model

[9]:
```
from keras.models import Sequential, Model
from keras.layers import Input, Dense, BatchNormalization, Dropout
from keras.losses import binary_crossentropy as BC
from keras import optimizers

# 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 a sequential model
    model = Sequential()

    # Setup optimizer, depending on input parameter string
    if optimizer == 'sgd':
        opt = optimizers.SGD(lr=learning_rate)
    elif optimizer == 'adam':
        opt = optimizers.Adam(lr=learning_rate)
    else:
        raise ValueError('Optimizer should be sgd or adam')

    # Add layers to the model, using the input parameters of the build_DNN␣
 ↪function
    if use_bn is True:
        model.add(BatchNormalization())
    if use_dropout is True:
        model.add(Dropout(0.5))
    if use_custom_dropout is True:
        model.add(myDropout(0.5))

    # Add first layer, requires input shape
    model.add(Dense(n_nodes,activation=act_fun,input_dim =input_shape))
    # Add remaining layers, do not require input shape
    for i in range(n_layers-1):
        model.add(Dense(n_nodes,activation=act_fun))
        if use_bn is True:
            model.add(BatchNormalization())
        if use_dropout is True:
            model.add(Dropout(0.3))
        if use_custom_dropout is True:
```

```
            model.add(myDropout(0.3))

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

    # Compile model
    model.compile(loss=BC, optimizer=optimizer, metrics=['accuracy'])

    return model
```

[10]:
```
# Lets define a help function for plotting the training results

# IMPORTANT NOTE
# The history unfortunately behaves a bit randomly for every user
# If the plots for accuracy and loss look mixed, change the order of
# val_loss, val_acc, loss, acc
# until the plots look as they "should"

import matplotlib.pyplot as plt
def plot_results(history):
    val_loss, val_acc, loss, acc = history.history.values()

    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()
```

12 Part 11: Train the DNN

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

12.0.1 2 layers, 20 nodes

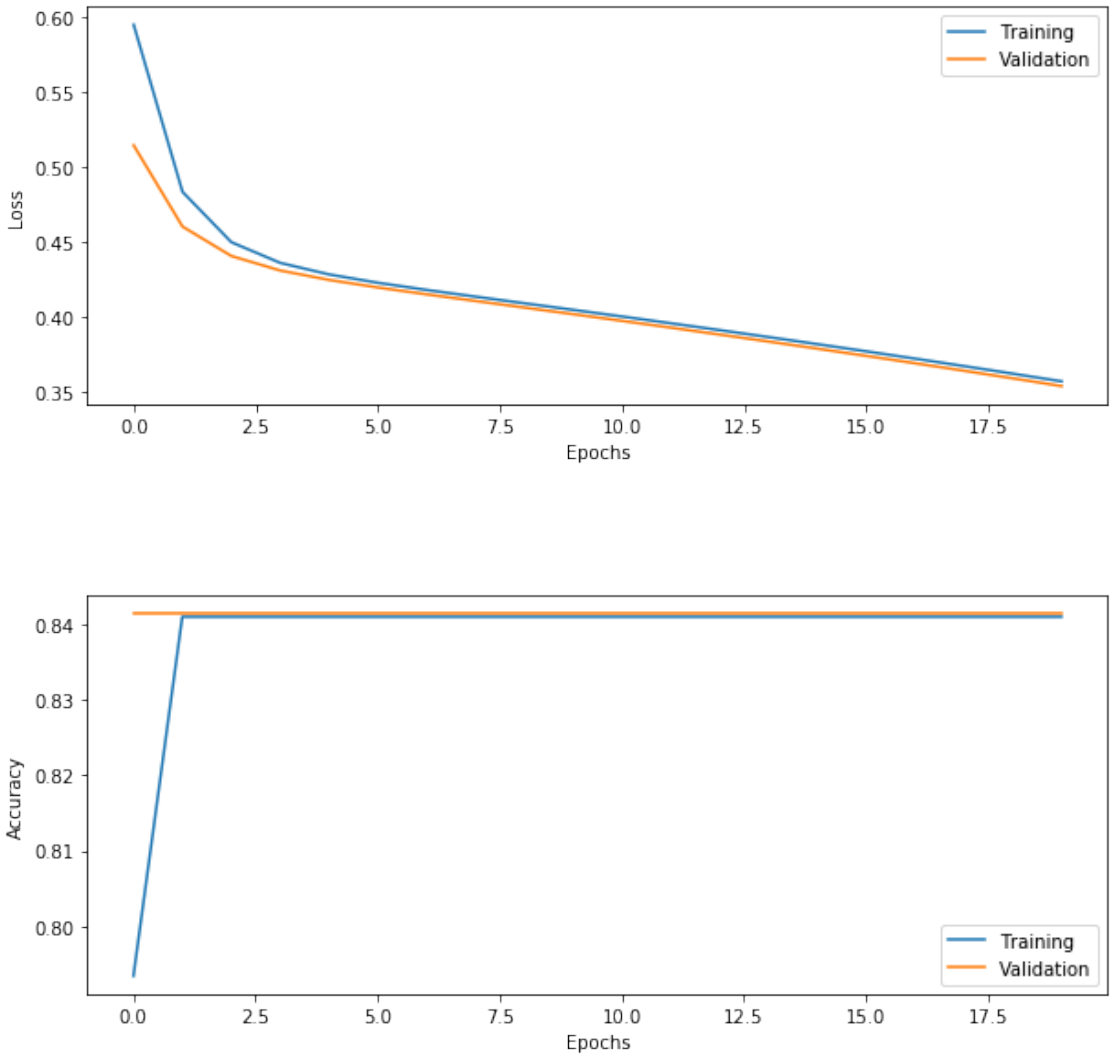
[11]:
```
# Setup some training parameters
batch_size = 10000
epochs = 20

input_shape = 92
n_nodes = 20
n_layers = 2
# Build the model
model1 = build_DNN(input_shape=input_shape, n_layers=n_layers,␣
 ↪n_nodes=n_nodes,learning_rate=0.1)
# Train the model, provide training data and validation data
history1 = model1.
 ↪fit(Xtrain,Ytrain,batch_size=batch_size,epochs=epochs,verbose=92,validation_data=(Xval,Yval
```

```
Train on 534895 samples, validate on 114620 samples
Epoch 1/20
Epoch 2/20
Epoch 3/20
Epoch 4/20
Epoch 5/20
Epoch 6/20
Epoch 7/20
Epoch 8/20
Epoch 9/20
Epoch 10/20
Epoch 11/20
Epoch 12/20
Epoch 13/20
Epoch 14/20
Epoch 15/20
Epoch 16/20
Epoch 17/20
Epoch 18/20
Epoch 19/20
Epoch 20/20
```

[12]:
```
# Evaluate the model on the test data
score = model1.evaluate(Xtest,Ytest)

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

```
114621/114621 [==============================] - 5s 47us/step
Test loss: 0.3563
Test accuracy: 0.8395
```

[13]:
```
# Plot the history from the training run
plot_results(history1)
```



13 Part 12: More questions

Question 9: What happens if you add several Dense layers without specifying the activation function?

Question 10: How are the weights in each dense layer initialized as default? How are the bias weights initialized?

Answer 9: The activation function will help to bring in the complexity to bring into the output model. If it is not specified, then it will just be regression model and the effieciency will reduce.

Answer 10: if none is given as default, then it will take from the global default initializer which

uses the glorot_uniform_initializer defined in tensorflow.python.ops.init_ops

14 Part 13: 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

[14]:
```
from sklearn.utils import class_weight

# Calculate class weights
class_wt = class_weight.compute_class_weight("balanced", np.
 ↪unique(Ytrain),Ytrain[:,0])

# Print the class weights
print(class_wt)
```

```
[3.14492421 0.59452067]
```

14.0.1 2 layers, 20 nodes, class weights

[15]:
```
# Setup some training parameters
batch_size = 10000
epochs = 20
input_shape = 92
n_nodes = 20
n_layers = 2
# Build and train model
model2 = build_DNN(input_shape = input_shape, n_layers=n_layers,␣
 ↪n_nodes=n_nodes,learning_rate=0.1)
history2 = model2.
 ↪fit(Xtrain,Ytrain,batch_size=batch_size,epochs=epochs,validation_data=(Xval,Yval),class_wei
model2.summary()
```

```
Train on 534895 samples, validate on 114620 samples
Epoch 1/20
534895/534895 [==============================] - 3s 6us/step - loss: 0.4339 -
accuracy: 0.8410 - val_loss: 0.4305 - val_accuracy: 0.8414
Epoch 2/20
534895/534895 [==============================] - 3s 5us/step - loss: 0.4291 -
accuracy: 0.8410 - val_loss: 0.4259 - val_accuracy: 0.8414
Epoch 3/20
534895/534895 [==============================] - 2s 5us/step - loss: 0.4246 -
accuracy: 0.8410 - val_loss: 0.4216 - val_accuracy: 0.8414
Epoch 4/20
534895/534895 [==============================] - 3s 5us/step - loss: 0.4203 -
accuracy: 0.8410 - val_loss: 0.4174 - val_accuracy: 0.8414
```

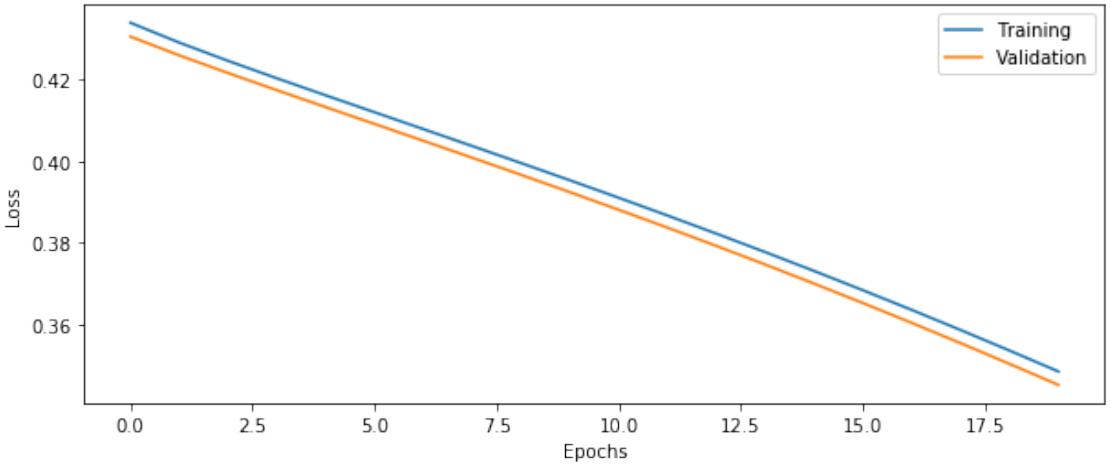
```
Epoch 5/20
534895/534895 [==============================] - 3s 6us/step - loss: 0.4161 -
accuracy: 0.8410 - val_loss: 0.4133 - val_accuracy: 0.8414
Epoch 6/20
534895/534895 [==============================] - 2s 5us/step - loss: 0.4120 -
accuracy: 0.8410 - val_loss: 0.4091 - val_accuracy: 0.8414
Epoch 7/20
534895/534895 [==============================] - 3s 5us/step - loss: 0.4079 -
accuracy: 0.8410 - val_loss: 0.4050 - val_accuracy: 0.8414
Epoch 8/20
534895/534895 [==============================] - 3s 5us/step - loss: 0.4037 -
accuracy: 0.8410 - val_loss: 0.4009 - val_accuracy: 0.8414
Epoch 9/20
534895/534895 [==============================] - 2s 4us/step - loss: 0.3996 -
accuracy: 0.8410 - val_loss: 0.3967 - val_accuracy: 0.8414
Epoch 10/20
534895/534895 [==============================] - 3s 5us/step - loss: 0.3953 -
accuracy: 0.8410 - val_loss: 0.3924 - val_accuracy: 0.8414
Epoch 11/20
534895/534895 [==============================] - 3s 5us/step - loss: 0.3911 -
accuracy: 0.8410 - val_loss: 0.3881 - val_accuracy: 0.8414
Epoch 12/20
534895/534895 [==============================] - 3s 5us/step - loss: 0.3867 -
accuracy: 0.8410 - val_loss: 0.3837 - val_accuracy: 0.8414
Epoch 13/20
534895/534895 [==============================] - 3s 5us/step - loss: 0.3823 -
accuracy: 0.8410 - val_loss: 0.3793 - val_accuracy: 0.8414
Epoch 14/20
534895/534895 [==============================] - 3s 5us/step - loss: 0.3778 -
accuracy: 0.8410 - val_loss: 0.3747 - val_accuracy: 0.8414
Epoch 15/20
534895/534895 [==============================] - 3s 5us/step - loss: 0.3731 -
accuracy: 0.8410 - val_loss: 0.3700 - val_accuracy: 0.8414
Epoch 16/20
534895/534895 [==============================] - 3s 6us/step - loss: 0.3684 -
accuracy: 0.8410 - val_loss: 0.3653 - val_accuracy: 0.8414
Epoch 17/20
534895/534895 [==============================] - 3s 5us/step - loss: 0.3636 -
accuracy: 0.8410 - val_loss: 0.3604 - val_accuracy: 0.8414
Epoch 18/20
534895/534895 [==============================] - 3s 5us/step - loss: 0.3587 -
accuracy: 0.8410 - val_loss: 0.3555 - val_accuracy: 0.8414
Epoch 19/20
534895/534895 [==============================] - 3s 5us/step - loss: 0.3537 -
accuracy: 0.8410 - val_loss: 0.3504 - val_accuracy: 0.8414
Epoch 20/20
534895/534895 [==============================] - 3s 6us/step - loss: 0.3485 -
accuracy: 0.8410 - val_loss: 0.3453 - val_accuracy: 0.8414
```

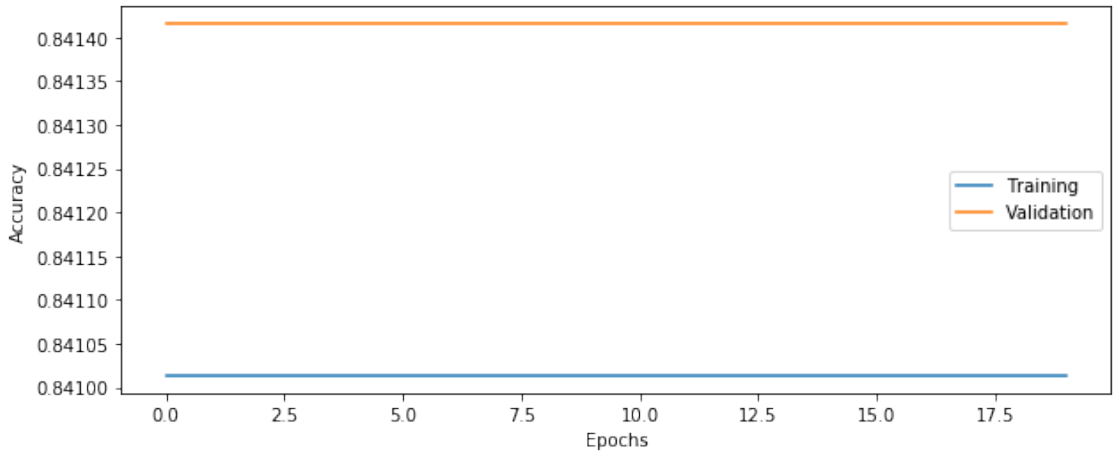
```
Model: "sequential_2"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_4 (Dense)              (None, 20)                1860
_________________________________________________________________
dense_5 (Dense)              (None, 20)                420
_________________________________________________________________
dense_6 (Dense)              (None, 1)                 21
=================================================================
Total params: 2,301
Trainable params: 2,301
Non-trainable params: 0
_________________________________________________________________
```

[16]:
```
# Evaluate model on test data
score = model2.evaluate(Xtest,Ytest)

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

```
114621/114621 [==============================] - 6s 50us/step
Test loss: 0.3479
Test accuracy: 0.8395
```

[17]:
```
plot_results(history2)
```





[18]:
```
batch_size = 100
epochs = 20
input_shape = 92
n_nodes = 20
n_layers = 2
# Build and train model
model3 = build_DNN(input_shape = input_shape, n_layers=n_layers,␣
 ↪n_nodes=n_nodes,learning_rate=0.1)
history3 = model3.
 ↪fit(Xtrain,Ytrain,batch_size=batch_size,epochs=epochs,validation_data=(Xval,Yval),class_wei
model3.summary()
```

```
Train on 534895 samples, validate on 114620 samples
Epoch 1/20
534895/534895 [==============================] - 22s 41us/step - loss: 0.2886 -
accuracy: 0.8621 - val_loss: 0.1908 - val_accuracy: 0.9015
Epoch 2/20
534895/534895 [==============================] - 21s 39us/step - loss: 0.1801 -
accuracy: 0.9054 - val_loss: 0.1741 - val_accuracy: 0.9092
Epoch 3/20
534895/534895 [==============================] - 22s 41us/step - loss: 0.1703 -
accuracy: 0.9100 - val_loss: 0.1683 - val_accuracy: 0.9114
Epoch 4/20
534895/534895 [==============================] - 22s 41us/step - loss: 0.1656 -
accuracy: 0.9116 - val_loss: 0.1643 - val_accuracy: 0.9125
Epoch 5/20
534895/534895 [==============================] - 22s 41us/step - loss: 0.1625 -
accuracy: 0.9154 - val_loss: 0.1617 - val_accuracy: 0.9159
Epoch 6/20
534895/534895 [==============================] - 23s 43us/step - loss: 0.1601 -
accuracy: 0.9171 - val_loss: 0.1596 - val_accuracy: 0.9163
```

```
Epoch 7/20
534895/534895 [==============================] - 22s 42us/step - loss: 0.1582 -
accuracy: 0.9168 - val_loss: 0.1580 - val_accuracy: 0.9159
Epoch 8/20
534895/534895 [==============================] - 24s 44us/step - loss: 0.1568 -
accuracy: 0.9170 - val_loss: 0.1568 - val_accuracy: 0.9161
Epoch 9/20
534895/534895 [==============================] - 23s 42us/step - loss: 0.1556 -
accuracy: 0.9174 - val_loss: 0.1558 - val_accuracy: 0.9165
Epoch 10/20
534895/534895 [==============================] - 23s 44us/step - loss: 0.1545 -
accuracy: 0.9177 - val_loss: 0.1548 - val_accuracy: 0.9165
Epoch 11/20
534895/534895 [==============================] - 23s 43us/step - loss: 0.1536 -
accuracy: 0.9180 - val_loss: 0.1540 - val_accuracy: 0.9166
Epoch 12/20
534895/534895 [==============================] - 23s 43us/step - loss: 0.1528 -
accuracy: 0.9182 - val_loss: 0.1532 - val_accuracy: 0.9169
Epoch 13/20
534895/534895 [==============================] - 23s 43us/step - loss: 0.1521 -
accuracy: 0.9184 - val_loss: 0.1525 - val_accuracy: 0.9172
Epoch 14/20
534895/534895 [==============================] - 23s 42us/step - loss: 0.1514 -
accuracy: 0.9187 - val_loss: 0.1519 - val_accuracy: 0.9172
Epoch 15/20
534895/534895 [==============================] - 23s 44us/step - loss: 0.1508 -
accuracy: 0.9188 - val_loss: 0.1514 - val_accuracy: 0.9175
Epoch 16/20
534895/534895 [==============================] - 24s 44us/step - loss: 0.1502 -
accuracy: 0.9191 - val_loss: 0.1508 - val_accuracy: 0.9176
Epoch 17/20
534895/534895 [==============================] - 26s 49us/step - loss: 0.1497 -
accuracy: 0.9193 - val_loss: 0.1503 - val_accuracy: 0.9178
Epoch 18/20
534895/534895 [==============================] - 22s 41us/step - loss: 0.1492 -
accuracy: 0.9196 - val_loss: 0.1497 - val_accuracy: 0.9183
Epoch 19/20
534895/534895 [==============================] - 22s 40us/step - loss: 0.1486 -
accuracy: 0.9198 - val_loss: 0.1492 - val_accuracy: 0.9183
Epoch 20/20
534895/534895 [==============================] - 22s 40us/step - loss: 0.1482 -
accuracy: 0.9201 - val_loss: 0.1488 - val_accuracy: 0.9187
Model: "sequential_3"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_7 (Dense)              (None, 20)                1860
_________________________________________________________________
```

```
dense_8 (Dense)              (None, 20)                420
_________________________________________________________________
dense_9 (Dense)              (None, 1)                 21
=================================================================
Total params: 2,301
Trainable params: 2,301
Non-trainable params: 0
_________________________________________________________________
```

[22]:
```
score = model3.evaluate(Xtest,Ytest)

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

```
114621/114621 [==============================] - 6s 51us/step
Test loss: 0.1488
Test accuracy: 0.9189
```

[21]:
```
batch_size = 1000
epochs = 20
input_shape = 92
n_nodes = 20
n_layers = 2
# Build and train model
model4 = build_DNN(input_shape = input_shape, n_layers=n_layers,␣
 ↪n_nodes=n_nodes,learning_rate=0.1)
history4 = model4.
 ↪fit(Xtrain,Ytrain,batch_size=batch_size,epochs=epochs,validation_data=(Xval,Yval),class_wei
model4.summary()
```

```
Train on 534895 samples, validate on 114620 samples
Epoch 1/20
534895/534895 [==============================] - 5s 9us/step - loss: 0.4396 -
accuracy: 0.8410 - val_loss: 0.4039 - val_accuracy: 0.8414
Epoch 2/20
534895/534895 [==============================] - 4s 8us/step - loss: 0.3810 -
accuracy: 0.8410 - val_loss: 0.3546 - val_accuracy: 0.8414
Epoch 3/20
534895/534895 [==============================] - 4s 8us/step - loss: 0.3264 -
accuracy: 0.8410 - val_loss: 0.2973 - val_accuracy: 0.8414
Epoch 4/20
534895/534895 [==============================] - 4s 8us/step - loss: 0.2728 -
accuracy: 0.8440 - val_loss: 0.2508 - val_accuracy: 0.8556
Epoch 5/20
534895/534895 [==============================] - 4s 8us/step - loss: 0.2356 -
accuracy: 0.8591 - val_loss: 0.2231 - val_accuracy: 0.8677
Epoch 6/20
534895/534895 [==============================] - 4s 8us/step - loss: 0.2148 -
```

```
accuracy: 0.8793 - val_loss: 0.2081 - val_accuracy: 0.8923
Epoch 7/20
534895/534895 [==============================] - 5s 9us/step - loss: 0.2032 -
accuracy: 0.8978 - val_loss: 0.1994 - val_accuracy: 0.9032
Epoch 8/20
534895/534895 [==============================] - 4s 8us/step - loss: 0.1962 -
accuracy: 0.9032 - val_loss: 0.1939 - val_accuracy: 0.9042
Epoch 9/20
534895/534895 [==============================] - 4s 7us/step - loss: 0.1915 -
accuracy: 0.9040 - val_loss: 0.1899 - val_accuracy: 0.9048
Epoch 10/20
534895/534895 [==============================] - 5s 9us/step - loss: 0.1881 -
accuracy: 0.9048 - val_loss: 0.1869 - val_accuracy: 0.9058
Epoch 11/20
534895/534895 [==============================] - 5s 9us/step - loss: 0.1853 -
accuracy: 0.9055 - val_loss: 0.1844 - val_accuracy: 0.9063
Epoch 12/20
534895/534895 [==============================] - 5s 9us/step - loss: 0.1831 -
accuracy: 0.9061 - val_loss: 0.1824 - val_accuracy: 0.9067
Epoch 13/20
534895/534895 [==============================] - 4s 8us/step - loss: 0.1812 -
accuracy: 0.9066 - val_loss: 0.1806 - val_accuracy: 0.9071
Epoch 14/20
534895/534895 [==============================] - 4s 8us/step - loss: 0.1795 -
accuracy: 0.9069 - val_loss: 0.1790 - val_accuracy: 0.9072
Epoch 15/20
534895/534895 [==============================] - 4s 8us/step - loss: 0.1780 -
accuracy: 0.9072 - val_loss: 0.1776 - val_accuracy: 0.9076
Epoch 16/20
534895/534895 [==============================] - 4s 7us/step - loss: 0.1767 -
accuracy: 0.9076 - val_loss: 0.1763 - val_accuracy: 0.9080
Epoch 17/20
534895/534895 [==============================] - 4s 8us/step - loss: 0.1754 -
accuracy: 0.9079 - val_loss: 0.1751 - val_accuracy: 0.9083
Epoch 18/20
534895/534895 [==============================] - 4s 8us/step - loss: 0.1743 -
accuracy: 0.9081 - val_loss: 0.1741 - val_accuracy: 0.9085
Epoch 19/20
534895/534895 [==============================] - 4s 8us/step - loss: 0.1733 -
accuracy: 0.9083 - val_loss: 0.1731 - val_accuracy: 0.9088
Epoch 20/20
534895/534895 [==============================] - 4s 8us/step - loss: 0.1724 -
accuracy: 0.9086 - val_loss: 0.1723 - val_accuracy: 0.9089
Model: "sequential_4"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_10 (Dense)             (None, 20)                1860
```

```
_________________________________________________________________
dense_11 (Dense)             (None, 20)                420
_________________________________________________________________
dense_12 (Dense)             (None, 1)                 21
=================================================================
Total params: 2,301
Trainable params: 2,301
Non-trainable params: 0
_________________________________________________________________
```

[23]:
```
score = model4.evaluate(Xtest,Ytest)

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

```
114621/114621 [==============================] - 5s 47us/step
Test loss: 0.1731
Test accuracy: 0.9084
```

15 Part 14: More questions

Question 11: 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.

Question 12: How busy is the GPU for a batch size of 100? How much GPU memory is used? Hint: run 'nvidia-smi' on the cloud computer a few times during training.

Question 13: 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 14: 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?

Question 15: What limits how large the batch size can be?

Question 16: 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?

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

Answer 11:The batch size is a hyperparameter which can be used to improve the models parameter. If we use the full data, the error value will be maximum. If it is divided into batch size, it will iterate each time for every batch to reduce the gradient of error.

Answer 12:nnnnn

Answer 13: The time for each batch when batch size is 100 is average of 44us. when batch size is 10,000, then the time is 9us. When the batch size is 1000, then the average time for each batch is

8us. if gradient in batch size is higher for each iteration, then the time taken for each iteration will decrease.

Answer 14: nnnn

Answer 15: The limit of batch size will be the maximum data of training set which is basically using the full data set which will reduce the effieciency. The test loss,test accuracy, the load on the GPU are some feature which can limit the batch size.

Answer 16: The learning rate will make the initial epoch unstable in terms of accuracy but in the end it will stabilize. If the batch size is decreased by k times, then the learning rate should be reduced by square root of k.

16 Part 15: Increasing the complexity

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

Question 17: 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()

16.0.1 4 layers, 20 nodes, class weights

[25]:
```
# Setup some training parameters
batch_size = 10000
epochs = 20
input_shape = 92
n_nodes = 20
n_layers = 4
# Build and train model
model5= build_DNN(input_shape = input_shape, n_layers=n_layers,␣
 ↪n_nodes=n_nodes,learning_rate=0.1)
history5= model5.
 ↪fit(Xtrain,Ytrain,batch_size=batch_size,epochs=epochs,validation_data=(Xval,Yval),class_wei
model5.ummary()
```

```
Train on 534895 samples, validate on 114620 samples
Epoch 1/20
534895/534895 [==============================] - 3s 6us/step - loss: 0.4910 -
accuracy: 0.8410 - val_loss: 0.4649 - val_accuracy: 0.8414
Epoch 2/20
534895/534895 [==============================] - 3s 6us/step - loss: 0.4547 -
accuracy: 0.8410 - val_loss: 0.4466 - val_accuracy: 0.8414
Epoch 3/20
534895/534895 [==============================] - 3s 7us/step - loss: 0.4437 -
accuracy: 0.8410 - val_loss: 0.4405 - val_accuracy: 0.8414
Epoch 4/20
534895/534895 [==============================] - 3s 6us/step - loss: 0.4399 -
accuracy: 0.8410 - val_loss: 0.4383 - val_accuracy: 0.8414
```

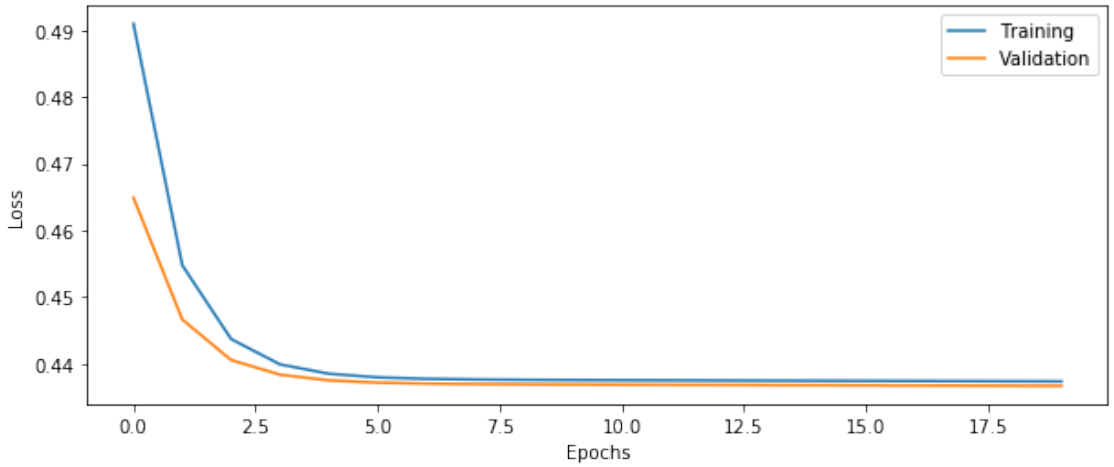
```
Epoch 5/20
534895/534895 [==============================] - 3s 6us/step - loss: 0.4385 -
accuracy: 0.8410 - val_loss: 0.4375 - val_accuracy: 0.8414
Epoch 6/20
534895/534895 [==============================] - 3s 6us/step - loss: 0.4379 -
accuracy: 0.8410 - val_loss: 0.4371 - val_accuracy: 0.8414
Epoch 7/20
534895/534895 [==============================] - 3s 6us/step - loss: 0.4377 -
accuracy: 0.8410 - val_loss: 0.4370 - val_accuracy: 0.8414
Epoch 8/20
534895/534895 [==============================] - 3s 6us/step - loss: 0.4376 -
accuracy: 0.8410 - val_loss: 0.4369 - val_accuracy: 0.8414
Epoch 9/20
534895/534895 [==============================] - 3s 5us/step - loss: 0.4376 -
accuracy: 0.8410 - val_loss: 0.4369 - val_accuracy: 0.8414
Epoch 10/20
534895/534895 [==============================] - 3s 5us/step - loss: 0.4375 -
accuracy: 0.8410 - val_loss: 0.4368 - val_accuracy: 0.8414
Epoch 11/20
534895/534895 [==============================] - 3s 5us/step - loss: 0.4375 -
accuracy: 0.8410 - val_loss: 0.4368 - val_accuracy: 0.8414
Epoch 12/20
534895/534895 [==============================] - 3s 6us/step - loss: 0.4375 -
accuracy: 0.8410 - val_loss: 0.4368 - val_accuracy: 0.8414
Epoch 13/20
534895/534895 [==============================] - 3s 5us/step - loss: 0.4375 -
accuracy: 0.8410 - val_loss: 0.4368 - val_accuracy: 0.8414
Epoch 14/20
534895/534895 [==============================] - 3s 6us/step - loss: 0.4374 -
accuracy: 0.8410 - val_loss: 0.4368 - val_accuracy: 0.8414
Epoch 15/20
534895/534895 [==============================] - 3s 5us/step - loss: 0.4374 -
accuracy: 0.8410 - val_loss: 0.4367 - val_accuracy: 0.8414
Epoch 16/20
534895/534895 [==============================] - 3s 6us/step - loss: 0.4374 -
accuracy: 0.8410 - val_loss: 0.4367 - val_accuracy: 0.8414
Epoch 17/20
534895/534895 [==============================] - 3s 6us/step - loss: 0.4374 -
accuracy: 0.8410 - val_loss: 0.4367 - val_accuracy: 0.8414
Epoch 18/20
534895/534895 [==============================] - 3s 6us/step - loss: 0.4374 -
accuracy: 0.8410 - val_loss: 0.4367 - val_accuracy: 0.8414
Epoch 19/20
534895/534895 [==============================] - 3s 6us/step - loss: 0.4373 -
accuracy: 0.8410 - val_loss: 0.4367 - val_accuracy: 0.8414
Epoch 20/20
534895/534895 [==============================] - 3s 6us/step - loss: 0.4373 -
accuracy: 0.8410 - val_loss: 0.4366 - val_accuracy: 0.8414
```

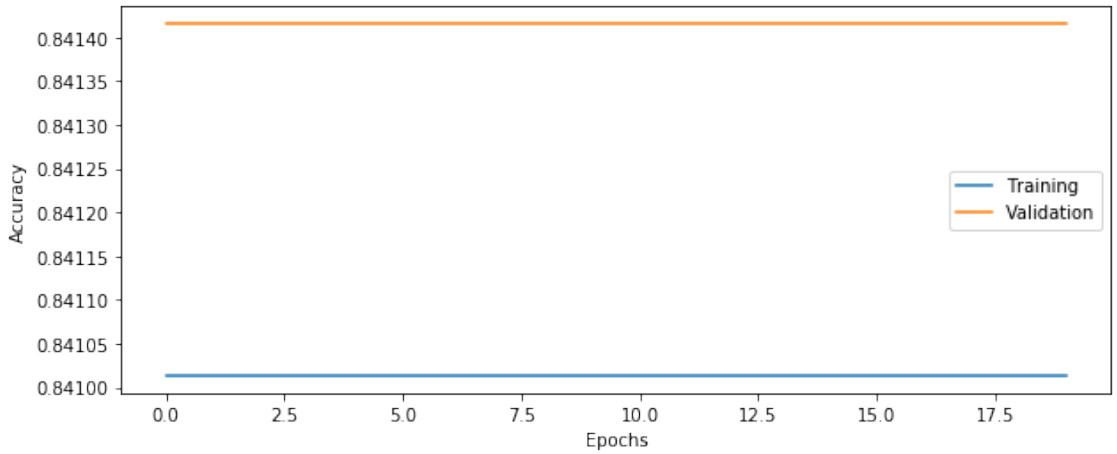
```
␣
↪---------------------------------------------------------------------------

       AttributeError                            Traceback (most recent call␣
↪last)

       <ipython-input-25-c5bc8639354f> in <module>
         8 model5= build_DNN(input_shape = input_shape, n_layers=n_layers,␣
↪n_nodes=n_nodes,learning_rate=0.1)
         9 history5= model5.
↪fit(Xtrain,Ytrain,batch_size=batch_size,epochs=epochs,validation_data=(Xval,Yval),class_weig
   ---> 10 model5.ummary()


       AttributeError: 'Sequential' object has no attribute 'ummary'
```

[26]:
```
# Evaluate model on test data
score = model5.evaluate(Xtest,Ytest)

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

```
114621/114621 [==============================] - 6s 52us/step
Test loss: 0.4399
Test accuracy: 0.8395
```

[27]:
```
plot_results(history5)
```





16.0.2 2 layers, 50 nodes, class weights

[28]:
```
# Setup some training parameters
batch_size = 10000
epochs = 20
input_shape = 92
n_nodes = 50
n_layers = 2
# Build and train model
model6 = build_DNN(input_shape = input_shape, n_layers=n_layers,␣
 ↪n_nodes=n_nodes,learning_rate=0.1)
history6 = model6.
 ↪fit(Xtrain,Ytrain,batch_size=batch_size,epochs=epochs,validation_data=(Xval,Yval),class_wei
model6.summary()
```

```
Train on 534895 samples, validate on 114620 samples
Epoch 1/20
534895/534895 [==============================] - 4s 7us/step - loss: 0.6477 -
accuracy: 0.6381 - val_loss: 0.4573 - val_accuracy: 0.8414
Epoch 2/20
534895/534895 [==============================] - 3s 6us/step - loss: 0.4411 -
accuracy: 0.8410 - val_loss: 0.4316 - val_accuracy: 0.8414
Epoch 3/20
534895/534895 [==============================] - 4s 7us/step - loss: 0.4282 -
accuracy: 0.8410 - val_loss: 0.4240 - val_accuracy: 0.8414
Epoch 4/20
534895/534895 [==============================] - 4s 7us/step - loss: 0.4214 -
accuracy: 0.8410 - val_loss: 0.4175 - val_accuracy: 0.8414
Epoch 5/20
534895/534895 [==============================] - 4s 7us/step - loss: 0.4149 -
accuracy: 0.8410 - val_loss: 0.4111 - val_accuracy: 0.8414
```

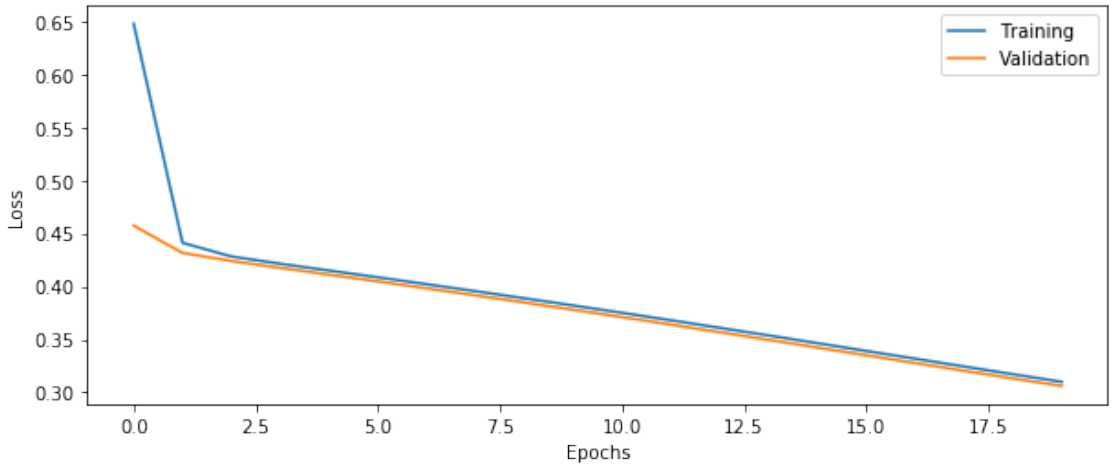
```
Epoch 6/20
534895/534895 [==============================] - 3s 6us/step - loss: 0.4085 -
accuracy: 0.8410 - val_loss: 0.4047 - val_accuracy: 0.8414
Epoch 7/20
534895/534895 [==============================] - 4s 7us/step - loss: 0.4020 -
accuracy: 0.8410 - val_loss: 0.3981 - val_accuracy: 0.8414
Epoch 8/20
534895/534895 [==============================] - 4s 7us/step - loss: 0.3954 -
accuracy: 0.8410 - val_loss: 0.3915 - val_accuracy: 0.8414
Epoch 9/20
534895/534895 [==============================] - 4s 7us/step - loss: 0.3888 -
accuracy: 0.8410 - val_loss: 0.3848 - val_accuracy: 0.8414
Epoch 10/20
534895/534895 [==============================] - 4s 7us/step - loss: 0.3820 -
accuracy: 0.8410 - val_loss: 0.3780 - val_accuracy: 0.8414
Epoch 11/20
534895/534895 [==============================] - 3s 6us/step - loss: 0.3751 -
accuracy: 0.8410 - val_loss: 0.3710 - val_accuracy: 0.8414
Epoch 12/20
534895/534895 [==============================] - 3s 6us/step - loss: 0.3680 -
accuracy: 0.8410 - val_loss: 0.3640 - val_accuracy: 0.8414
Epoch 13/20
534895/534895 [==============================] - 3s 6us/step - loss: 0.3609 -
accuracy: 0.8410 - val_loss: 0.3568 - val_accuracy: 0.8414
Epoch 14/20
534895/534895 [==============================] - 4s 7us/step - loss: 0.3537 -
accuracy: 0.8410 - val_loss: 0.3496 - val_accuracy: 0.8414
Epoch 15/20
534895/534895 [==============================] - 3s 6us/step - loss: 0.3464 -
accuracy: 0.8410 - val_loss: 0.3423 - val_accuracy: 0.8414
Epoch 16/20
534895/534895 [==============================] - 4s 7us/step - loss: 0.3391 -
accuracy: 0.8410 - val_loss: 0.3350 - val_accuracy: 0.8414
Epoch 17/20
534895/534895 [==============================] - 4s 7us/step - loss: 0.3318 -
accuracy: 0.8410 - val_loss: 0.3277 - val_accuracy: 0.8414
Epoch 18/20
534895/534895 [==============================] - 4s 7us/step - loss: 0.3244 -
accuracy: 0.8410 - val_loss: 0.3204 - val_accuracy: 0.8414
Epoch 19/20
534895/534895 [==============================] - 4s 7us/step - loss: 0.3172 -
accuracy: 0.8410 - val_loss: 0.3132 - val_accuracy: 0.8414
Epoch 20/20
534895/534895 [==============================] - 4s 7us/step - loss: 0.3100 -
accuracy: 0.8410 - val_loss: 0.3061 - val_accuracy: 0.8414
Model: "sequential_6"

_________________________________________________________________
Layer (type)                 Output Shape              Param #
```

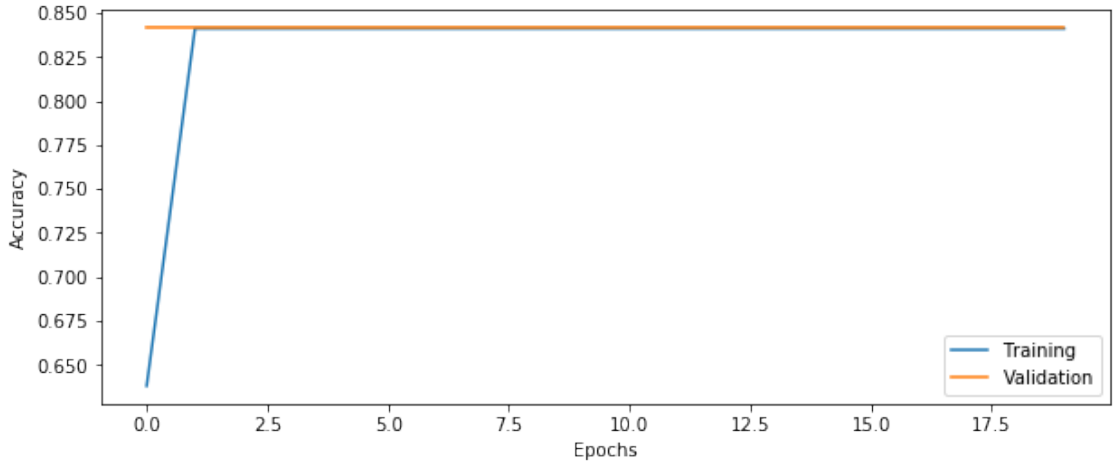
```
=================================================================
dense_18 (Dense)             (None, 50)                4650
_________________________________________________________________
dense_19 (Dense)             (None, 50)                2550
_________________________________________________________________
dense_20 (Dense)             (None, 1)                 51
=================================================================
Total params: 7,251
Trainable params: 7,251
Non-trainable params: 0
_________________________________________________________________
```

[29]:
```
# Evaluate model on test data
score = model6.evaluate(Xtest,Ytest)

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

```
114621/114621 [==============================] - 5s 47us/step
Test loss: 0.3080
Test accuracy: 0.8395
```

[30]:
```
plot_results(history6)
```





16.0.3 4 layers, 50 nodes, class weights

[31]:
```
# Setup some training parameters
batch_size = 10000
epochs = 20
input_shape = 92
n_nodes = 50
n_layers = 4
# Build and train model
model7 = build_DNN(input_shape = input_shape, n_layers=n_layers,␣
 ↪n_nodes=n_nodes,learning_rate=0.1)
history7 = model7.
 ↪fit(Xtrain,Ytrain,batch_size=batch_size,epochs=epochs,validation_data=(Xval,Yval),class_wei
model7.summary()
```

```
Train on 534895 samples, validate on 114620 samples
Epoch 1/20
534895/534895 [==============================] - 5s 9us/step - loss: 0.4415 -
accuracy: 0.8410 - val_loss: 0.4382 - val_accuracy: 0.8414
Epoch 2/20
534895/534895 [==============================] - 4s 8us/step - loss: 0.4384 -
accuracy: 0.8410 - val_loss: 0.4376 - val_accuracy: 0.8414
Epoch 3/20
534895/534895 [==============================] - 5s 9us/step - loss: 0.4382 -
accuracy: 0.8410 - val_loss: 0.4375 - val_accuracy: 0.8414
Epoch 4/20
534895/534895 [==============================] - 4s 8us/step - loss: 0.4381 -
accuracy: 0.8410 - val_loss: 0.4375 - val_accuracy: 0.8414
Epoch 5/20
534895/534895 [==============================] - 5s 8us/step - loss: 0.4381 -
accuracy: 0.8410 - val_loss: 0.4374 - val_accuracy: 0.8414
```

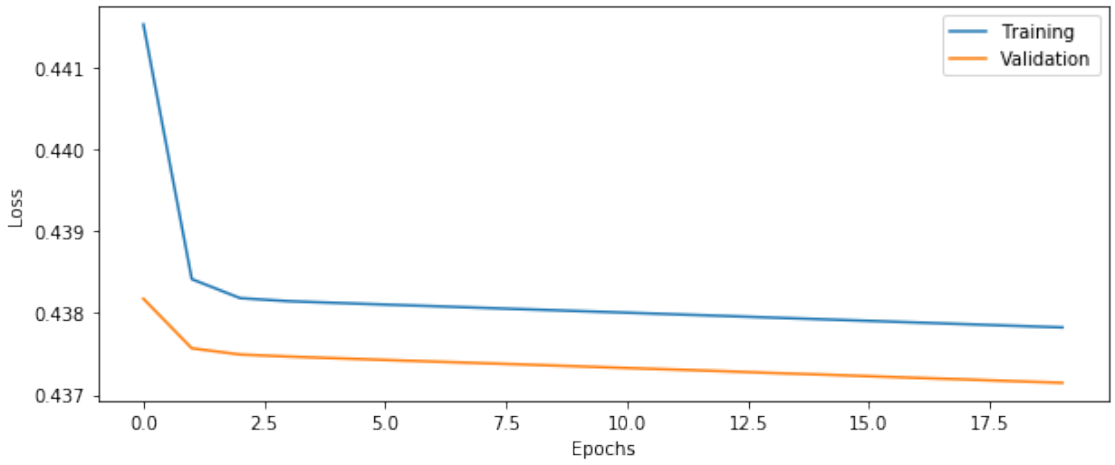
```
Epoch 6/20
534895/534895 [==============================] - 4s 8us/step - loss: 0.4381 -
accuracy: 0.8410 - val_loss: 0.4374 - val_accuracy: 0.8414
Epoch 7/20
534895/534895 [==============================] - 5s 9us/step - loss: 0.4381 -
accuracy: 0.8410 - val_loss: 0.4374 - val_accuracy: 0.8414
Epoch 8/20
534895/534895 [==============================] - 4s 8us/step - loss: 0.4381 -
accuracy: 0.8410 - val_loss: 0.4374 - val_accuracy: 0.8414
Epoch 9/20
534895/534895 [==============================] - 5s 8us/step - loss: 0.4380 -
accuracy: 0.8410 - val_loss: 0.4374 - val_accuracy: 0.8414
Epoch 10/20
534895/534895 [==============================] - 5s 9us/step - loss: 0.4380 -
accuracy: 0.8410 - val_loss: 0.4373 - val_accuracy: 0.8414
Epoch 11/20
534895/534895 [==============================] - 4s 8us/step - loss: 0.4380 -
accuracy: 0.8410 - val_loss: 0.4373 - val_accuracy: 0.8414
Epoch 12/20
534895/534895 [==============================] - 5s 9us/step - loss: 0.4380 -
accuracy: 0.8410 - val_loss: 0.4373 - val_accuracy: 0.8414
Epoch 13/20
534895/534895 [==============================] - 5s 9us/step - loss: 0.4380 -
accuracy: 0.8410 - val_loss: 0.4373 - val_accuracy: 0.8414
Epoch 14/20
534895/534895 [==============================] - 5s 9us/step - loss: 0.4379 -
accuracy: 0.8410 - val_loss: 0.4373 - val_accuracy: 0.8414
Epoch 15/20
534895/534895 [==============================] - 5s 8us/step - loss: 0.4379 -
accuracy: 0.8410 - val_loss: 0.4372 - val_accuracy: 0.8414
Epoch 16/20
534895/534895 [==============================] - 4s 8us/step - loss: 0.4379 -
accuracy: 0.8410 - val_loss: 0.4372 - val_accuracy: 0.8414
Epoch 17/20
534895/534895 [==============================] - 4s 8us/step - loss: 0.4379 -
accuracy: 0.8410 - val_loss: 0.4372 - val_accuracy: 0.8414
Epoch 18/20
534895/534895 [==============================] - 4s 8us/step - loss: 0.4379 -
accuracy: 0.8410 - val_loss: 0.4372 - val_accuracy: 0.8414
Epoch 19/20
534895/534895 [==============================] - 4s 8us/step - loss: 0.4378 -
accuracy: 0.8410 - val_loss: 0.4372 - val_accuracy: 0.8414
Epoch 20/20
534895/534895 [==============================] - 4s 8us/step - loss: 0.4378 -
accuracy: 0.8410 - val_loss: 0.4371 - val_accuracy: 0.8414
Model: "sequential_7"

_________________________________________________________________
Layer (type)                 Output Shape              Param #
```

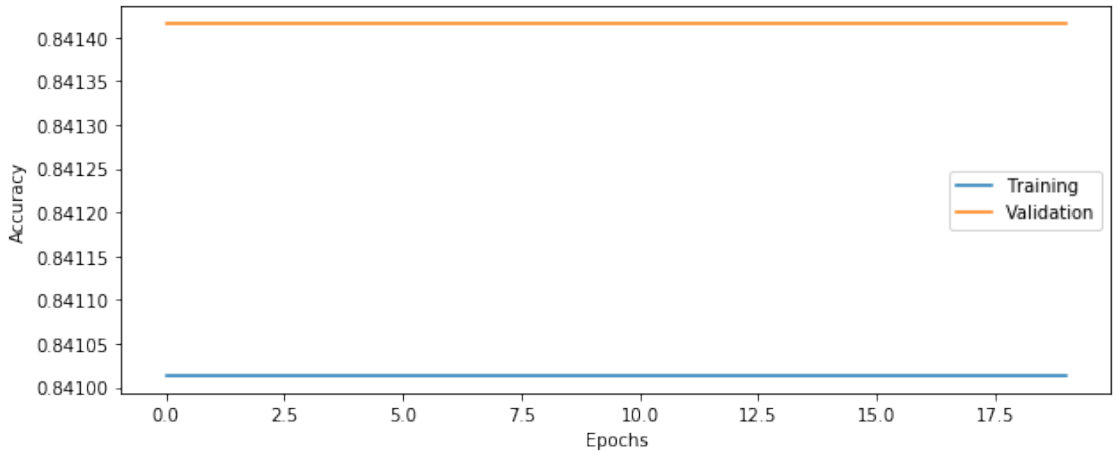
```
=================================================================
dense_21 (Dense)             (None, 50)                4650
_________________________________________________________________
dense_22 (Dense)             (None, 50)                2550
_________________________________________________________________
dense_23 (Dense)             (None, 50)                2550
_________________________________________________________________
dense_24 (Dense)             (None, 50)                2550
_________________________________________________________________
dense_25 (Dense)             (None, 1)                 51
=================================================================
Total params: 12,351
Trainable params: 12,351
Non-trainable params: 0
_________________________________________________________________
```

[32]:
```
# Evaluate model on test data
score = model7.evaluate(Xtest,Ytest,batch_size=100)

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

```
114621/114621 [==============================] - 3s 24us/step
Test loss: 0.4404
Test accuracy: 0.8395
```

[33]:
```
plot_results(history7)
```





17 Part 16: Batch normalization

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

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

Question 18: Why is batch normalization important when training deep networks?

It will increase the learning rate and normalize each input for each layer. This will also give a easy start to intialize weight.

17.0.1 2 layers, 20 nodes, class weights, batch normalization

[34]:
```
# Setup some training parameters
batch_size = 10000
epochs = 20
input_shape = 92
n_nodes = 20
n_layers = 2
# Build and train model
model8 = build_DNN(input_shape = input_shape, n_layers=n_layers,␣
 ↪n_nodes=n_nodes,learning_rate=0.1,use_bn=True)
history8 = model8.
 ↪fit(Xtrain,Ytrain,batch_size=batch_size,epochs=epochs,validation_data=(Xval,Yval),class_wei
model8.summary()
```

```
Train on 534895 samples, validate on 114620 samples
Epoch 1/20
534895/534895 [==============================] - 5s 10us/step - loss: 0.5165 -
accuracy: 0.8483 - val_loss: 0.5339 - val_accuracy: 0.8414
Epoch 2/20
```

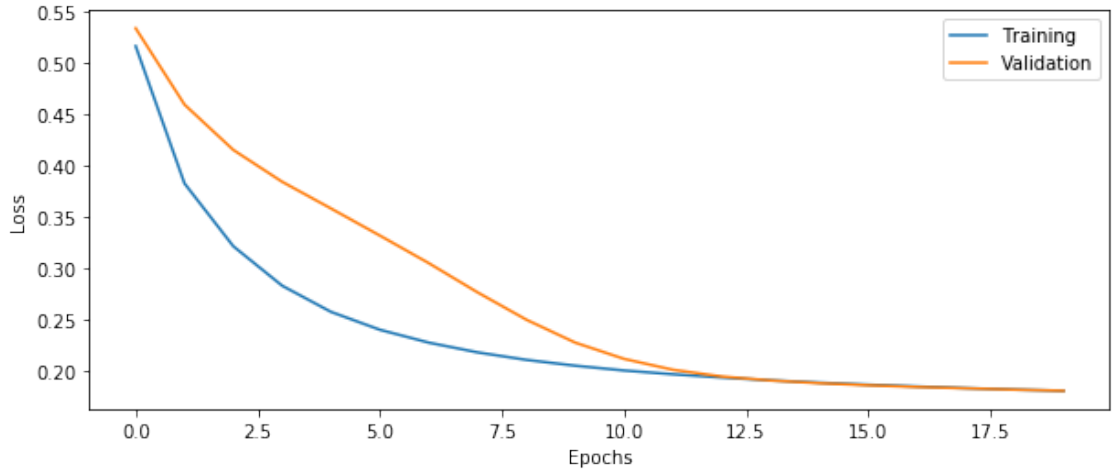
```
534895/534895 [==============================] - 5s 9us/step - loss: 0.3825 -
accuracy: 0.8901 - val_loss: 0.4595 - val_accuracy: 0.8414
Epoch 3/20
534895/534895 [==============================] - 4s 8us/step - loss: 0.3212 -
accuracy: 0.8978 - val_loss: 0.4152 - val_accuracy: 0.8414
Epoch 4/20
534895/534895 [==============================] - 5s 9us/step - loss: 0.2828 -
accuracy: 0.9013 - val_loss: 0.3842 - val_accuracy: 0.8414
Epoch 5/20
534895/534895 [==============================] - 5s 9us/step - loss: 0.2573 -
accuracy: 0.9036 - val_loss: 0.3581 - val_accuracy: 0.8414
Epoch 6/20
534895/534895 [==============================] - 5s 8us/step - loss: 0.2398 -
accuracy: 0.9050 - val_loss: 0.3317 - val_accuracy: 0.8414
Epoch 7/20
534895/534895 [==============================] - 5s 9us/step - loss: 0.2273 -
accuracy: 0.9063 - val_loss: 0.3048 - val_accuracy: 0.8414
Epoch 8/20
534895/534895 [==============================] - 4s 8us/step - loss: 0.2178 -
accuracy: 0.9070 - val_loss: 0.2763 - val_accuracy: 0.8415
Epoch 9/20
534895/534895 [==============================] - 4s 8us/step - loss: 0.2105 -
accuracy: 0.9075 - val_loss: 0.2497 - val_accuracy: 0.8488
Epoch 10/20
534895/534895 [==============================] - 4s 8us/step - loss: 0.2048 -
accuracy: 0.9079 - val_loss: 0.2273 - val_accuracy: 0.8632
Epoch 11/20
534895/534895 [==============================] - 5s 8us/step - loss: 0.2001 -
accuracy: 0.9080 - val_loss: 0.2114 - val_accuracy: 0.8830
Epoch 12/20
534895/534895 [==============================] - 5s 9us/step - loss: 0.1964 -
accuracy: 0.9082 - val_loss: 0.2008 - val_accuracy: 0.8984
Epoch 13/20
534895/534895 [==============================] - 5s 8us/step - loss: 0.1932 -
accuracy: 0.9083 - val_loss: 0.1942 - val_accuracy: 0.9025
Epoch 14/20
534895/534895 [==============================] - 5s 9us/step - loss: 0.1905 -
accuracy: 0.9087 - val_loss: 0.1904 - val_accuracy: 0.9046
Epoch 15/20
534895/534895 [==============================] - 5s 9us/step - loss: 0.1882 -
accuracy: 0.9090 - val_loss: 0.1877 - val_accuracy: 0.9054
Epoch 16/20
534895/534895 [==============================] - 4s 8us/step - loss: 0.1862 -
accuracy: 0.9092 - val_loss: 0.1857 - val_accuracy: 0.9061
Epoch 17/20
534895/534895 [==============================] - 5s 9us/step - loss: 0.1845 -
accuracy: 0.9095 - val_loss: 0.1840 - val_accuracy: 0.9063
Epoch 18/20
```

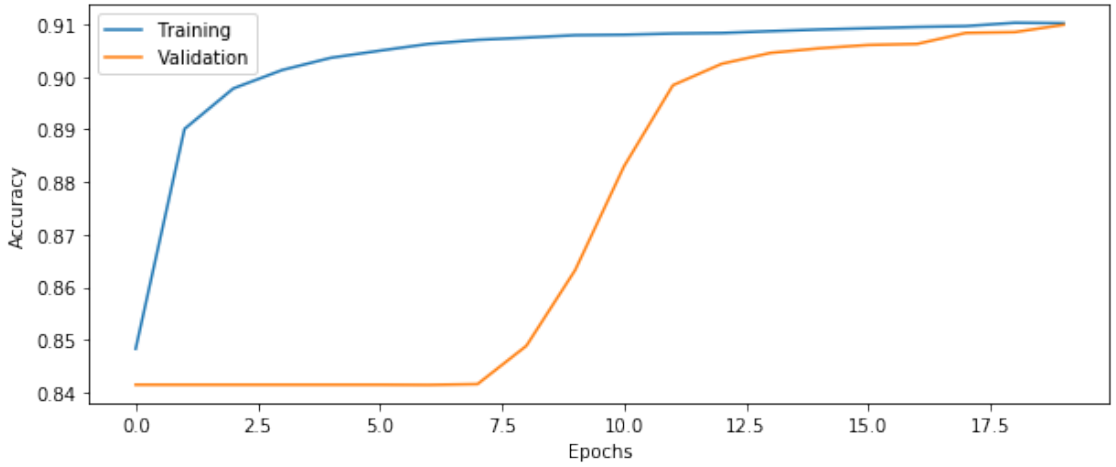
```
534895/534895 [==============================] - 5s 9us/step - loss: 0.1829 -
accuracy: 0.9097 - val_loss: 0.1827 - val_accuracy: 0.9084
Epoch 19/20
534895/534895 [==============================] - 5s 8us/step - loss: 0.1815 -
accuracy: 0.9103 - val_loss: 0.1814 - val_accuracy: 0.9085
Epoch 20/20
534895/534895 [==============================] - 5s 9us/step - loss: 0.1803 -
accuracy: 0.9102 - val_loss: 0.1803 - val_accuracy: 0.9099
Model: "sequential_8"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
batch_normalization_1 (Batch (None, 92)                368
_________________________________________________________________
dense_26 (Dense)             (None, 20)                1860
_________________________________________________________________
dense_27 (Dense)             (None, 20)                420
_________________________________________________________________
batch_normalization_2 (Batch (None, 20)                80
_________________________________________________________________
dense_28 (Dense)             (None, 1)                 21
=================================================================
Total params: 2,749
Trainable params: 2,525
Non-trainable params: 224
_________________________________________________________________
```

[35]:
```
# Evaluate model on test data
score = model8.evaluate(Xtest,Ytest)

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

```
114621/114621 [==============================] - 8s 68us/step
Test loss: 0.1808
Test accuracy: 0.9100
```

[36]:
```
plot_results(history8)
```





18 Part 17: 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.

18.0.1 2 layers, 20 nodes, class weights, ReLU, no batch normalization

[37]:
```
# Setup some training parameters
batch_size = 10000
epochs = 20
input_shape = 92
n_nodes = 20
```

```
n_layers = 2
# Build and train model
model9 = build_DNN(input_shape = input_shape, n_layers=n_layers,␣
 ↪n_nodes=n_nodes, act_fun='relu',learning_rate=0.1)
history9 = model9.
 ↪fit(Xtrain,Ytrain,batch_size=batch_size,epochs=epochs,validation_data=(Xval,Yval),class_wei
model9.summary()
```

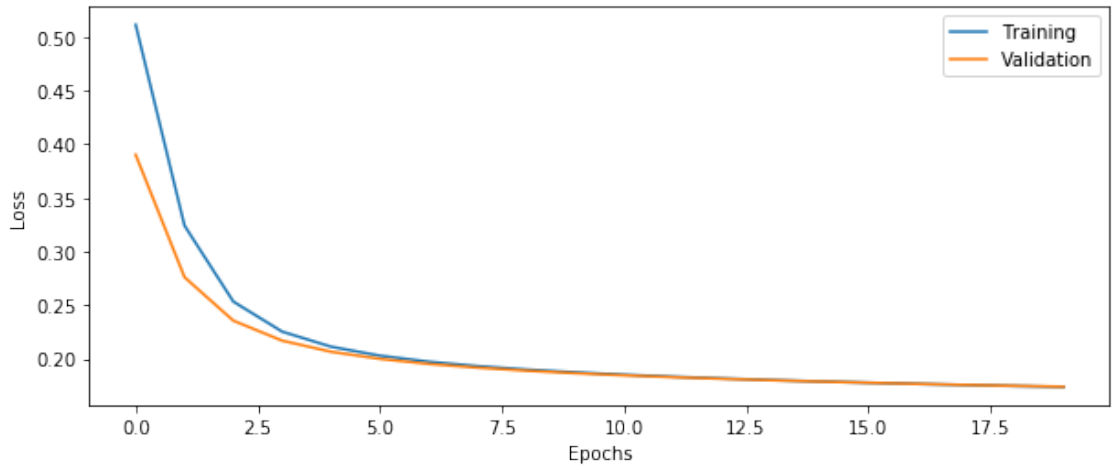
```
Train on 534895 samples, validate on 114620 samples
Epoch 1/20
534895/534895 [==============================] - 3s 5us/step - loss: 0.5112 -
accuracy: 0.8592 - val_loss: 0.3901 - val_accuracy: 0.8662
Epoch 2/20
534895/534895 [==============================] - 3s 5us/step - loss: 0.3241 -
accuracy: 0.8689 - val_loss: 0.2761 - val_accuracy: 0.8728
Epoch 3/20
534895/534895 [==============================] - 3s 5us/step - loss: 0.2534 -
accuracy: 0.8774 - val_loss: 0.2356 - val_accuracy: 0.8809
Epoch 4/20
534895/534895 [==============================] - 2s 4us/step - loss: 0.2254 -
accuracy: 0.8828 - val_loss: 0.2170 - val_accuracy: 0.8853
Epoch 5/20
534895/534895 [==============================] - 3s 5us/step - loss: 0.2114 -
accuracy: 0.8869 - val_loss: 0.2067 - val_accuracy: 0.8896
Epoch 6/20
534895/534895 [==============================] - 3s 5us/step - loss: 0.2030 -
accuracy: 0.8913 - val_loss: 0.2000 - val_accuracy: 0.8937
Epoch 7/20
534895/534895 [==============================] - 3s 5us/step - loss: 0.1974 -
accuracy: 0.8954 - val_loss: 0.1954 - val_accuracy: 0.8976
Epoch 8/20
534895/534895 [==============================] - 3s 5us/step - loss: 0.1933 -
accuracy: 0.8988 - val_loss: 0.1918 - val_accuracy: 0.9004
Epoch 9/20
534895/534895 [==============================] - 3s 5us/step - loss: 0.1901 -
accuracy: 0.9008 - val_loss: 0.1890 - val_accuracy: 0.9019
Epoch 10/20
534895/534895 [==============================] - 2s 5us/step - loss: 0.1875 -
accuracy: 0.9020 - val_loss: 0.1867 - val_accuracy: 0.9029
Epoch 11/20
534895/534895 [==============================] - 3s 6us/step - loss: 0.1853 -
accuracy: 0.9031 - val_loss: 0.1847 - val_accuracy: 0.9042
Epoch 12/20
534895/534895 [==============================] - 3s 5us/step - loss: 0.1834 -
accuracy: 0.9050 - val_loss: 0.1830 - val_accuracy: 0.9069
Epoch 13/20
534895/534895 [==============================] - 3s 5us/step - loss: 0.1818 -
```

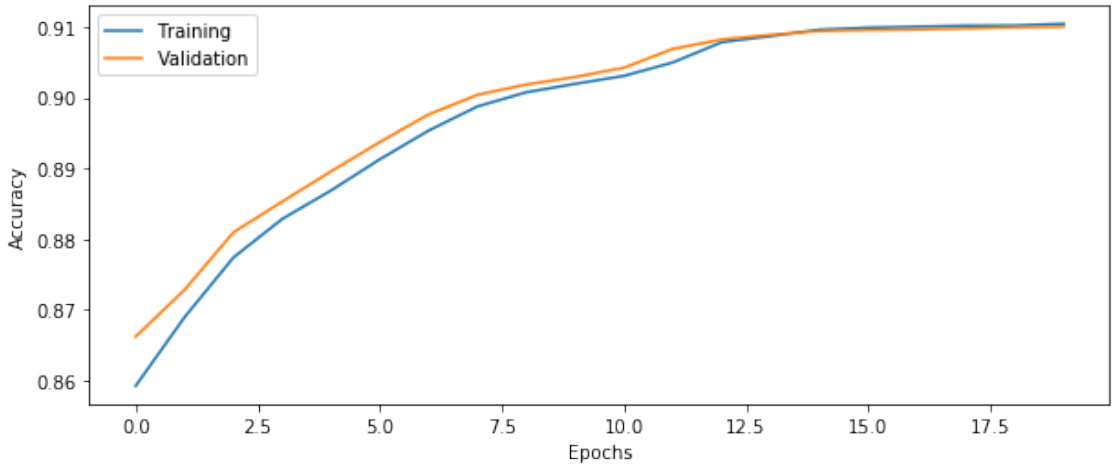
```
accuracy: 0.9078 - val_loss: 0.1815 - val_accuracy: 0.9082
Epoch 14/20
534895/534895 [==============================] - 3s 5us/step - loss: 0.1803 -
accuracy: 0.9087 - val_loss: 0.1801 - val_accuracy: 0.9089
Epoch 15/20
534895/534895 [==============================] - 3s 5us/step - loss: 0.1790 -
accuracy: 0.9096 - val_loss: 0.1789 - val_accuracy: 0.9094
Epoch 16/20
534895/534895 [==============================] - 3s 5us/step - loss: 0.1778 -
accuracy: 0.9099 - val_loss: 0.1778 - val_accuracy: 0.9096
Epoch 17/20
534895/534895 [==============================] - 3s 6us/step - loss: 0.1767 -
accuracy: 0.9100 - val_loss: 0.1768 - val_accuracy: 0.9096
Epoch 18/20
534895/534895 [==============================] - 3s 6us/step - loss: 0.1757 -
accuracy: 0.9102 - val_loss: 0.1758 - val_accuracy: 0.9098
Epoch 19/20
534895/534895 [==============================] - 3s 6us/step - loss: 0.1748 -
accuracy: 0.9102 - val_loss: 0.1749 - val_accuracy: 0.9100
Epoch 20/20
534895/534895 [==============================] - 3s 5us/step - loss: 0.1739 -
accuracy: 0.9105 - val_loss: 0.1741 - val_accuracy: 0.9101
Model: "sequential_9"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_29 (Dense)             (None, 20)                1860
_________________________________________________________________
dense_30 (Dense)             (None, 20)                420
_________________________________________________________________
dense_31 (Dense)             (None, 1)                 21
=================================================================
Total params: 2,301
Trainable params: 2,301
Non-trainable params: 0
_________________________________________________________________
```

[38]:
```
# Evaluate model on test data
score = model9.evaluate(Xtest,Ytest)
print('Test loss: %.4f' % score[0])
print('Test accuracy: %.4f' % score[1])
```

```
114621/114621 [==============================] - 6s 55us/step
Test loss: 0.1754
Test accuracy: 0.9103
```

[39]:
```
plot_results(history9)
```





19 Part 18: 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/

19.0.1 2 layers, 20 nodes, class weights, Adam optimizer, no batch normalization, sigmoid activations

[40]:
```
# Setup some training parameters
batch_size = 10000
epochs = 20
input_shape = 92
```

```
n_nodes = 20
n_layers = 2
# Build and train model
model10 = build_DNN(input_shape = input_shape, n_layers=n_layers,␣
 ↪n_nodes=n_nodes,learning_rate=0.1,optimizer = "adam")
history10 = model10.
 ↪fit(Xtrain,Ytrain,batch_size=batch_size,epochs=epochs,validation_data=(Xval,Yval))
mode10.summary()
```

```
Train on 534895 samples, validate on 114620 samples
Epoch 1/20
534895/534895 [==============================] - 3s 6us/step - loss: 0.4487 -
accuracy: 0.8410 - val_loss: 0.3604 - val_accuracy: 0.8414
Epoch 2/20
534895/534895 [==============================] - 3s 5us/step - loss: 0.3140 -
accuracy: 0.8410 - val_loss: 0.2709 - val_accuracy: 0.8457
Epoch 3/20
534895/534895 [==============================] - 3s 5us/step - loss: 0.2466 -
accuracy: 0.8769 - val_loss: 0.2275 - val_accuracy: 0.8966
Epoch 4/20
534895/534895 [==============================] - 3s 5us/step - loss: 0.2170 -
accuracy: 0.8997 - val_loss: 0.2083 - val_accuracy: 0.9018
Epoch 5/20
534895/534895 [==============================] - 3s 6us/step - loss: 0.2019 -
accuracy: 0.9034 - val_loss: 0.1966 - val_accuracy: 0.9050
Epoch 6/20
534895/534895 [==============================] - 3s 6us/step - loss: 0.1919 -
accuracy: 0.9058 - val_loss: 0.1883 - val_accuracy: 0.9067
Epoch 7/20
534895/534895 [==============================] - 3s 6us/step - loss: 0.1847 -
accuracy: 0.9075 - val_loss: 0.1823 - val_accuracy: 0.9081
Epoch 8/20
534895/534895 [==============================] - 3s 5us/step - loss: 0.1795 -
accuracy: 0.9085 - val_loss: 0.1778 - val_accuracy: 0.9087
Epoch 9/20
534895/534895 [==============================] - 3s 6us/step - loss: 0.1755 -
accuracy: 0.9094 - val_loss: 0.1745 - val_accuracy: 0.9094
Epoch 10/20
534895/534895 [==============================] - 3s 5us/step - loss: 0.1725 -
accuracy: 0.9101 - val_loss: 0.1718 - val_accuracy: 0.9099
Epoch 11/20
534895/534895 [==============================] - 3s 5us/step - loss: 0.1701 -
accuracy: 0.9105 - val_loss: 0.1697 - val_accuracy: 0.9105
Epoch 12/20
534895/534895 [==============================] - 3s 5us/step - loss: 0.1681 -
accuracy: 0.9112 - val_loss: 0.1679 - val_accuracy: 0.9117
Epoch 13/20
```

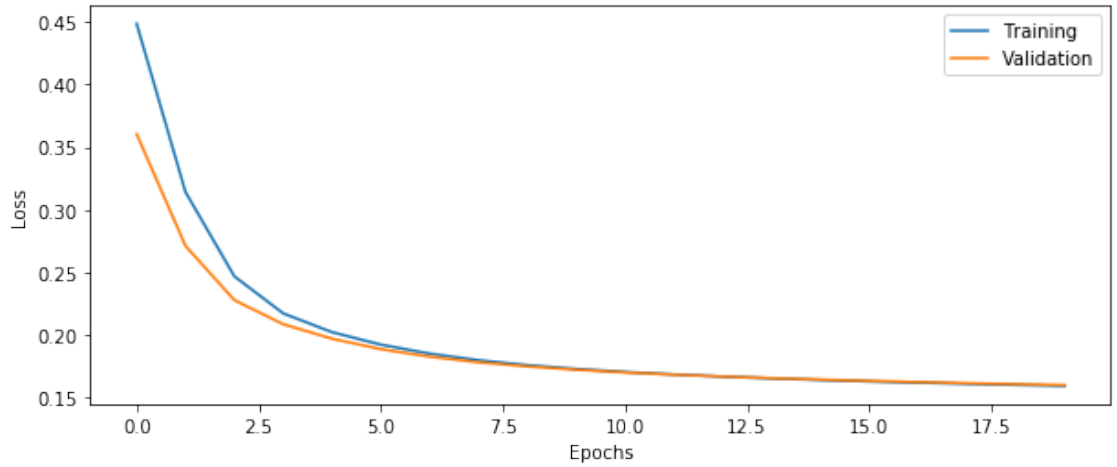
```
534895/534895 [==============================] - 3s 5us/step - loss: 0.1664 -
accuracy: 0.9134 - val_loss: 0.1664 - val_accuracy: 0.9140
Epoch 14/20
534895/534895 [==============================] - 3s 5us/step - loss: 0.1649 -
accuracy: 0.9158 - val_loss: 0.1651 - val_accuracy: 0.9155
Epoch 15/20
534895/534895 [==============================] - 3s 6us/step - loss: 0.1637 -
accuracy: 0.9163 - val_loss: 0.1639 - val_accuracy: 0.9156
Epoch 16/20
534895/534895 [==============================] - 3s 6us/step - loss: 0.1625 -
accuracy: 0.9164 - val_loss: 0.1629 - val_accuracy: 0.9157
Epoch 17/20
534895/534895 [==============================] - 3s 6us/step - loss: 0.1615 -
accuracy: 0.9174 - val_loss: 0.1620 - val_accuracy: 0.9171
Epoch 18/20
534895/534895 [==============================] - 3s 6us/step - loss: 0.1606 -
accuracy: 0.9180 - val_loss: 0.1611 - val_accuracy: 0.9168
Epoch 19/20
534895/534895 [==============================] - 3s 6us/step - loss: 0.1597 -
accuracy: 0.9178 - val_loss: 0.1603 - val_accuracy: 0.9168
Epoch 20/20
534895/534895 [==============================] - 3s 6us/step - loss: 0.1589 -
accuracy: 0.9180 - val_loss: 0.1596 - val_accuracy: 0.9165
```

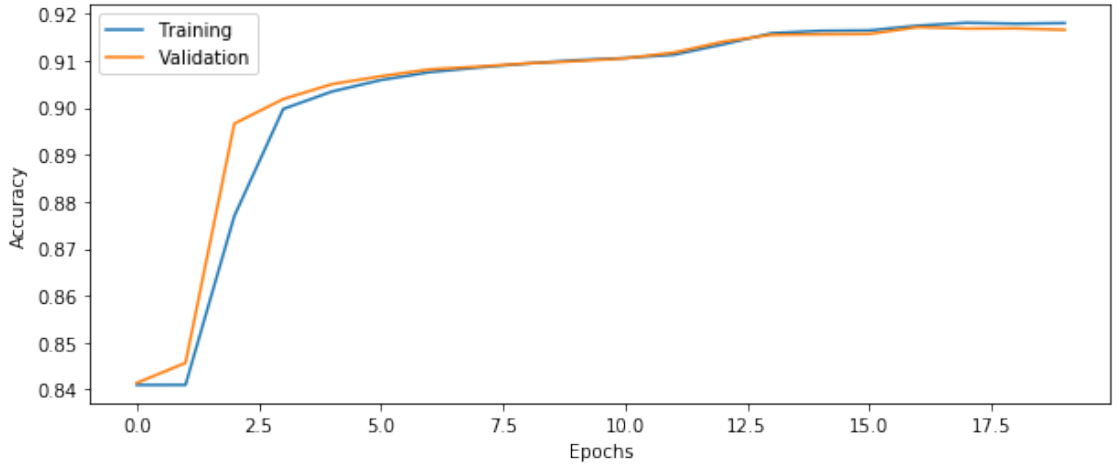
```
␣
↪---------------------------------------------------------------------------

NameError                                 Traceback (most recent call␣
↪last)

<ipython-input-40-023cd7f4f56d> in <module>
  8 model10 = build_DNN(input_shape = input_shape, n_layers=n_layers,␣
↪n_nodes=n_nodes,learning_rate=0.1,optimizer = "adam")
  9 history10 = model10.
↪fit(Xtrain,Ytrain,batch_size=batch_size,epochs=epochs,validation_data=(Xval,Yval))
---> 10 mode10.summary()

NameError: name 'mode10' is not defined
```

[41]:
```
# Evaluate model on test data
score = model10.evaluate(Xtest,Ytest)

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

```
114621/114621 [==============================] - 7s 62us/step
Test loss: 0.1596
Test accuracy: 0.9173
```

[42]:
```
plot_results(history10)
```





20 Part 19: Dropout regularization

Dropout is a type of regularization that can improve accuracy for validation and test data.

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

See https://keras.io/layers/core/ for how the Dropout layer works.

Question 19: How does the validation accuracy change when adding dropout?

The validation accuracy is constant even when epochs is changing. When we compare with and without dropout regularization, The validation accuracy has decreased by 0.08%.

Question 20: How does the test accuracy change when adding dropout?

The test accuarcy decreased.

20.0.1 2 layers, 20 nodes, class weights, dropout, SGD optimizer, no batch normalization, sigmoid activations

[43]:
```
# Setup some training parameters
batch_size = 10000
epochs = 20
input_shape = 92
n_nodes = 20
n_layers = 2
# Build and train model
model11 = build_DNN(input_shape = input_shape, n_layers=n_layers,␣
 ↪n_nodes=n_nodes,learning_rate=0.1,use_dropout = True)
history11 = model11.
 ↪fit(Xtrain,Ytrain,batch_size=batch_size,epochs=epochs,validation_data=(Xval,Yval))
model11.summary()
```

```
Train on 534895 samples, validate on 114620 samples
Epoch 1/20
534895/534895 [==============================] - 4s 8us/step - loss: 0.7118 -
accuracy: 0.5355 - val_loss: 0.5337 - val_accuracy: 0.8414
Epoch 2/20
534895/534895 [==============================] - 4s 8us/step - loss: 0.5114 -
accuracy: 0.8094 - val_loss: 0.4550 - val_accuracy: 0.8414
Epoch 3/20
534895/534895 [==============================] - 4s 8us/step - loss: 0.4657 -
accuracy: 0.8355 - val_loss: 0.4331 - val_accuracy: 0.8414
Epoch 4/20
534895/534895 [==============================] - 5s 9us/step - loss: 0.4507 -
accuracy: 0.8392 - val_loss: 0.4241 - val_accuracy: 0.8414
Epoch 5/20
534895/534895 [==============================] - 4s 8us/step - loss: 0.4429 -
accuracy: 0.8403 - val_loss: 0.4187 - val_accuracy: 0.8414
Epoch 6/20
534895/534895 [==============================] - 4s 8us/step - loss: 0.4386 -
accuracy: 0.8406 - val_loss: 0.4144 - val_accuracy: 0.8414
Epoch 7/20
534895/534895 [==============================] - 4s 8us/step - loss: 0.4344 -
accuracy: 0.8407 - val_loss: 0.4105 - val_accuracy: 0.8414
Epoch 8/20
534895/534895 [==============================] - 4s 8us/step - loss: 0.4304 -
```

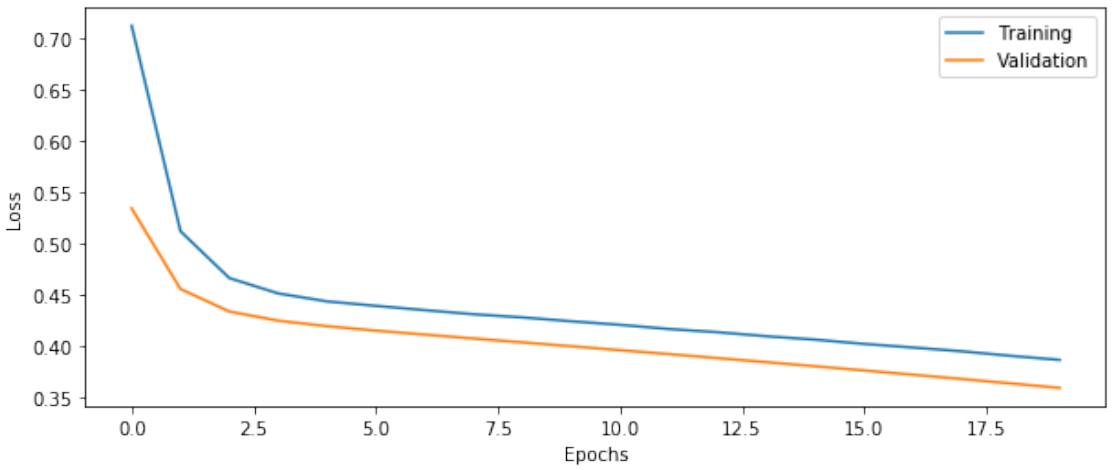
```
accuracy: 0.8408 - val_loss: 0.4067 - val_accuracy: 0.8414
Epoch 9/20
534895/534895 [==============================] - 4s 8us/step - loss: 0.4273 -
accuracy: 0.8408 - val_loss: 0.4030 - val_accuracy: 0.8414
Epoch 10/20
534895/534895 [==============================] - 4s 8us/step - loss: 0.4236 -
accuracy: 0.8409 - val_loss: 0.3992 - val_accuracy: 0.8414
Epoch 11/20
534895/534895 [==============================] - 5s 8us/step - loss: 0.4201 -
accuracy: 0.8409 - val_loss: 0.3954 - val_accuracy: 0.8414
Epoch 12/20
534895/534895 [==============================] - 4s 8us/step - loss: 0.4160 -
accuracy: 0.8409 - val_loss: 0.3915 - val_accuracy: 0.8414
Epoch 13/20
534895/534895 [==============================] - 4s 8us/step - loss: 0.4129 -
accuracy: 0.8410 - val_loss: 0.3876 - val_accuracy: 0.8414
Epoch 14/20
534895/534895 [==============================] - 4s 8us/step - loss: 0.4089 -
accuracy: 0.8410 - val_loss: 0.3836 - val_accuracy: 0.8414
Epoch 15/20
534895/534895 [==============================] - 4s 8us/step - loss: 0.4056 -
accuracy: 0.8409 - val_loss: 0.3796 - val_accuracy: 0.8414
Epoch 16/20
534895/534895 [==============================] - 4s 7us/step - loss: 0.4015 -
accuracy: 0.8411 - val_loss: 0.3756 - val_accuracy: 0.8414
Epoch 17/20
534895/534895 [==============================] - 4s 8us/step - loss: 0.3979 -
accuracy: 0.8410 - val_loss: 0.3714 - val_accuracy: 0.8414
Epoch 18/20
534895/534895 [==============================] - 4s 8us/step - loss: 0.3942 -
accuracy: 0.8411 - val_loss: 0.3672 - val_accuracy: 0.8414
Epoch 19/20
534895/534895 [==============================] - 4s 8us/step - loss: 0.3898 -
accuracy: 0.8411 - val_loss: 0.3629 - val_accuracy: 0.8414
Epoch 20/20
534895/534895 [==============================] - 4s 8us/step - loss: 0.3859 -
accuracy: 0.8412 - val_loss: 0.3585 - val_accuracy: 0.8414
Model: "sequential_11"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dropout_1 (Dropout)          (None, 92)                0
_________________________________________________________________
dense_35 (Dense)             (None, 20)                1860
_________________________________________________________________
dense_36 (Dense)             (None, 20)                420
_________________________________________________________________
dropout_2 (Dropout)          (None, 20)                0
```

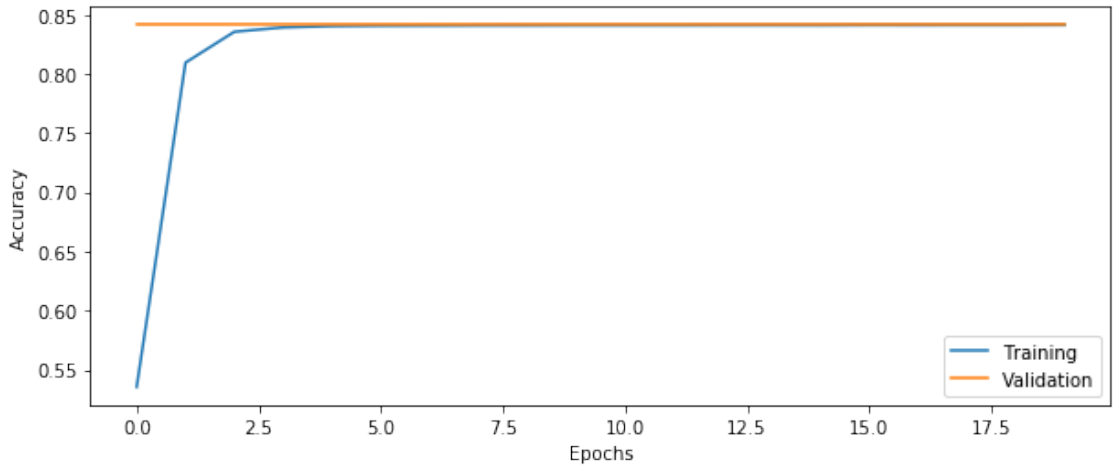
```
_________________________________________________________________
dense_37 (Dense)             (None, 1)                 21
=================================================================
Total params: 2,301
Trainable params: 2,301
Non-trainable params: 0
_________________________________________________________________
```

[44]:
```
# Evaluate model on test data
score = model11.evaluate(Xtest,Ytest)

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

```
114621/114621 [==============================] - 6s 55us/step
Test loss: 0.3612
Test accuracy: 0.8395
```

[45]:
```
plot_results(history11)
```





21 Part 20: 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 21: How high classification accuracy can you achieve for the test data? What is your best configuration?

[48]:
```
# Find your best configuration for the DNN
batch_size = 1000
epochs = 20
input_shape = 92
n_nodes = 30
n_layers = 2
# Build and train DNN
model12 = build_DNN(input_shape = input_shape, n_layers=n_layers,␣
 ↪n_nodes=n_nodes,learning_rate=0.1,optimizer = "adam",use_bn=True,act_fun =␣
 ↪'relu')
history12 = model12.
 ↪fit(Xtrain,Ytrain,batch_size=batch_size,epochs=epochs,validation_data=(Xval,Yval))
model12.summary()
```

```
Train on 534895 samples, validate on 114620 samples
Epoch 1/20
534895/534895 [==============================] - 9s 17us/step - loss: 0.2387 -
accuracy: 0.9109 - val_loss: 0.1584 - val_accuracy: 0.9192
Epoch 2/20
534895/534895 [==============================] - 8s 15us/step - loss: 0.1503 -
accuracy: 0.9189 - val_loss: 0.1450 - val_accuracy: 0.9214
```

```
Epoch 3/20
534895/534895 [==============================] - 8s 15us/step - loss: 0.1438 -
accuracy: 0.9211 - val_loss: 0.1425 - val_accuracy: 0.9179
Epoch 4/20
534895/534895 [==============================] - 9s 16us/step - loss: 0.1396 -
accuracy: 0.9230 - val_loss: 0.1348 - val_accuracy: 0.9288
Epoch 5/20
534895/534895 [==============================] - 8s 15us/step - loss: 0.1357 -
accuracy: 0.9261 - val_loss: 0.1329 - val_accuracy: 0.9285
Epoch 6/20
534895/534895 [==============================] - 9s 16us/step - loss: 0.1331 -
accuracy: 0.9276 - val_loss: 0.1300 - val_accuracy: 0.9296
Epoch 7/20
534895/534895 [==============================] - 8s 15us/step - loss: 0.1305 -
accuracy: 0.9293 - val_loss: 0.1461 - val_accuracy: 0.9080
Epoch 8/20
534895/534895 [==============================] - 9s 17us/step - loss: 0.1282 -
accuracy: 0.9310 - val_loss: 0.1248 - val_accuracy: 0.9346
Epoch 9/20
534895/534895 [==============================] - 9s 18us/step - loss: 0.1266 -
accuracy: 0.9316 - val_loss: 0.1237 - val_accuracy: 0.9357
Epoch 10/20
534895/534895 [==============================] - 8s 15us/step - loss: 0.1240 -
accuracy: 0.9334 - val_loss: 0.1233 - val_accuracy: 0.9369
Epoch 11/20
534895/534895 [==============================] - 7s 13us/step - loss: 0.1227 -
accuracy: 0.9340 - val_loss: 0.1312 - val_accuracy: 0.9299
Epoch 12/20
534895/534895 [==============================] - 7s 14us/step - loss: 0.1215 -
accuracy: 0.9344 - val_loss: 0.1251 - val_accuracy: 0.9327
Epoch 13/20
534895/534895 [==============================] - 7s 13us/step - loss: 0.1206 -
accuracy: 0.9347 - val_loss: 0.1242 - val_accuracy: 0.9324
Epoch 14/20
534895/534895 [==============================] - 7s 14us/step - loss: 0.1201 -
accuracy: 0.9353 - val_loss: 0.1324 - val_accuracy: 0.9247
Epoch 15/20
534895/534895 [==============================] - 7s 13us/step - loss: 0.1182 -
accuracy: 0.9356 - val_loss: 0.1186 - val_accuracy: 0.9363
Epoch 16/20
534895/534895 [==============================] - 7s 14us/step - loss: 0.1187 -
accuracy: 0.9352 - val_loss: 0.1181 - val_accuracy: 0.9356
Epoch 17/20
534895/534895 [==============================] - 8s 15us/step - loss: 0.1176 -
accuracy: 0.9363 - val_loss: 0.1168 - val_accuracy: 0.9366
Epoch 18/20
534895/534895 [==============================] - 8s 15us/step - loss: 0.1161 -
accuracy: 0.9369 - val_loss: 0.1406 - val_accuracy: 0.9254
```

```
Epoch 19/20
534895/534895 [==============================] - 7s 12us/step - loss: 0.1158 -
accuracy: 0.9374 - val_loss: 0.1205 - val_accuracy: 0.9342
Epoch 20/20
534895/534895 [==============================] - 8s 15us/step - loss: 0.1153 -
accuracy: 0.9378 - val_loss: 0.1501 - val_accuracy: 0.9207
Model: "sequential_12"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
batch_normalization_3 (Batch (None, 92)                368
_________________________________________________________________
dense_38 (Dense)             (None, 30)                2790
_________________________________________________________________
dense_39 (Dense)             (None, 30)                930
_________________________________________________________________
batch_normalization_4 (Batch (None, 30)                120
_________________________________________________________________
dense_40 (Dense)             (None, 1)                 31
=================================================================
Total params: 4,239
Trainable params: 3,995
Non-trainable params: 244
_________________________________________________________________
```

[49]:
```
# Evaluate DNN on test data
score = model12.evaluate(Xtest,Ytest)
print('Test loss: %.4f' % score[0])
print('Test accuracy: %.4f' % score[1])
```

```
114621/114621 [==============================] - 8s 70us/step
Test loss: 0.1626
Test accuracy: 0.9196
```

22 Part 21: 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.

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

Question 22: What is the mean and the standard deviation of the test accuracy?

[50]:
```
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
```

22.0.1 Your best config, custom dropout

[53]:
```
# Your best training parameters
batch_size = 1000
epochs = 20
input_shape = 92
```

```
n_nodes = 30
n_layers = 2
# Build and train DNN
model13 = build_DNN(input_shape = input_shape, n_layers=n_layers,␣
 ↪n_nodes=n_nodes,learning_rate=0.1,optimizer =␣
 ↪"adam",use_bn=True,act_fun="relu")
history13 = model13.
 ↪fit(Xtrain,Ytrain,batch_size=batch_size,epochs=epochs,validation_data=(Xval,Yval),class_wei
 ↪= class_wt)
model13.summary()
```

```
Train on 534895 samples, validate on 114620 samples
Epoch 1/20
534895/534895 [==============================] - 9s 17us/step - loss: 0.2433 -
accuracy: 0.9084 - val_loss: 0.1568 - val_accuracy: 0.9195
Epoch 2/20
534895/534895 [==============================] - 8s 15us/step - loss: 0.1500 -
accuracy: 0.9196 - val_loss: 0.1455 - val_accuracy: 0.9210
Epoch 3/20
534895/534895 [==============================] - 8s 15us/step - loss: 0.1437 -
accuracy: 0.9215 - val_loss: 0.1391 - val_accuracy: 0.9238
Epoch 4/20
534895/534895 [==============================] - 8s 15us/step - loss: 0.1399 -
accuracy: 0.9238 - val_loss: 0.1397 - val_accuracy: 0.9288
Epoch 5/20
534895/534895 [==============================] - 8s 15us/step - loss: 0.1371 -
accuracy: 0.9258 - val_loss: 0.1347 - val_accuracy: 0.9258
Epoch 6/20
534895/534895 [==============================] - 8s 16us/step - loss: 0.1354 -
accuracy: 0.9264 - val_loss: 0.1324 - val_accuracy: 0.9267
Epoch 7/20
534895/534895 [==============================] - 8s 15us/step - loss: 0.1328 -
accuracy: 0.9282 - val_loss: 0.1283 - val_accuracy: 0.9327
Epoch 8/20
534895/534895 [==============================] - 8s 15us/step - loss: 0.1315 -
accuracy: 0.9293 - val_loss: 0.1275 - val_accuracy: 0.9291
Epoch 9/20
534895/534895 [==============================] - 8s 15us/step - loss: 0.1288 -
accuracy: 0.9306 - val_loss: 0.1281 - val_accuracy: 0.9319- loss: 0
Epoch 10/20
534895/534895 [==============================] - 8s 15us/step - loss: 0.1274 -
accuracy: 0.9319 - val_loss: 0.1251 - val_accuracy: 0.9329
Epoch 11/20
534895/534895 [==============================] - 8s 15us/step - loss: 0.1257 -
accuracy: 0.9327 - val_loss: 0.1286 - val_accuracy: 0.9341
Epoch 12/20
534895/534895 [==============================] - 8s 15us/step - loss: 0.1240 -
```

```
accuracy: 0.9336 - val_loss: 0.1404 - val_accuracy: 0.9289
Epoch 13/20
534895/534895 [==============================] - 8s 16us/step - loss: 0.1231 -
accuracy: 0.9337 - val_loss: 0.1385 - val_accuracy: 0.9258
Epoch 14/20
534895/534895 [==============================] - 8s 16us/step - loss: 0.1219 -
accuracy: 0.9348 - val_loss: 0.1249 - val_accuracy: 0.9319
Epoch 15/20
534895/534895 [==============================] - 8s 15us/step - loss: 0.1213 -
accuracy: 0.9351 - val_loss: 0.1316 - val_accuracy: 0.9310
Epoch 16/20
534895/534895 [==============================] - 8s 15us/step - loss: 0.1198 -
accuracy: 0.9354 - val_loss: 0.1394 - val_accuracy: 0.9207
Epoch 17/20
534895/534895 [==============================] - 7s 14us/step - loss: 0.1185 -
accuracy: 0.9363 - val_loss: 0.1306 - val_accuracy: 0.9300
Epoch 18/20
534895/534895 [==============================] - 8s 15us/step - loss: 0.1189 -
accuracy: 0.9358 - val_loss: 0.1175 - val_accuracy: 0.9370
Epoch 19/20
534895/534895 [==============================] - 8s 15us/step - loss: 0.1179 -
accuracy: 0.9365 - val_loss: 0.1352 - val_accuracy: 0.9300
Epoch 20/20
534895/534895 [==============================] - 9s 17us/step - loss: 0.1172 -
accuracy: 0.9369 - val_loss: 0.1252 - val_accuracy: 0.9336
Model: "sequential_13"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
batch_normalization_5 (Batch (None, 92)                368
_________________________________________________________________
dense_41 (Dense)             (None, 30)                2790
_________________________________________________________________
dense_42 (Dense)             (None, 30)                930
_________________________________________________________________
batch_normalization_6 (Batch (None, 30)                120
_________________________________________________________________
dense_43 (Dense)             (None, 1)                 31
=================================================================
Total params: 4,239
Trainable params: 3,995
Non-trainable params: 244
_________________________________________________________________
```

[54]:
```
# 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 = model13.evaluate(Xtest,Ytest)

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

```
114621/114621 [==============================] - 8s 70us/step
Test accuracy: 0.9331
```

[55]:
```
# Run the testing 100 times, and save the accuracies in an array
accuracy_score = np.empty(100)
for i in range(100):
    score = model13.evaluate(Xtest,Ytest)
    accuracy_score[i] = score[1]
```

```
114621/114621 [==============================] - 8s 68us/step
114621/114621 [==============================] - 8s 69us/step
114621/114621 [==============================] - 8s 71us/step
114621/114621 [==============================] - 8s 70us/step
114621/114621 [==============================] - 8s 68us/step
114621/114621 [==============================] - 8s 68us/step
114621/114621 [==============================] - 8s 68us/step
114621/114621 [==============================] - 8s 68us/step
114621/114621 [==============================] - 8s 70us/step
114621/114621 [==============================] - 8s 69us/step
114621/114621 [==============================] - 8s 70us/step
114621/114621 [==============================] - 8s 68us/step
114621/114621 [==============================] - 8s 70us/step
114621/114621 [==============================] - 8s 67us/step
114621/114621 [==============================] - 8s 71us/step
114621/114621 [==============================] - 8s 71us/step
114621/114621 [==============================] - 8s 69us/step
114621/114621 [==============================] - 8s 70us/step
114621/114621 [==============================] - 8s 74us/step
114621/114621 [==============================] - 8s 72us/step
114621/114621 [==============================] - 8s 71us/step
114621/114621 [==============================] - 8s 74us/step
114621/114621 [==============================] - 8s 69us/step
114621/114621 [==============================] - 8s 74us/step
114621/114621 [==============================] - 7s 57us/step
114621/114621 [==============================] - 2s 18us/step
114621/114621 [==============================] - 2s 21us/step
114621/114621 [==============================] - 2s 17us/step
114621/114621 [==============================] - 2s 17us/step
114621/114621 [==============================] - 2s 16us/step
114621/114621 [==============================] - 2s 18us/step
114621/114621 [==============================] - 2s 14us/step
114621/114621 [==============================] - 3s 29us/step
```

```
114621/114621 [==============================] - 2s 15us/step
114621/114621 [==============================] - 2s 14us/step
114621/114621 [==============================] - 3s 25us/step
114621/114621 [==============================] - 2s 21us/step
114621/114621 [==============================] - 2s 14us/step
114621/114621 [==============================] - 2s 15us/step
114621/114621 [==============================] - 2s 14us/step
114621/114621 [==============================] - 3s 27us/step
114621/114621 [==============================] - 2s 15us/step
114621/114621 [==============================] - 2s 14us/step
114621/114621 [==============================] - 3s 26us/step
114621/114621 [==============================] - 2s 15us/step
114621/114621 [==============================] - 2s 15us/step
114621/114621 [==============================] - 3s 26us/step
114621/114621 [==============================] - 2s 14us/step
114621/114621 [==============================] - 1s 13us/step
114621/114621 [==============================] - 1s 12us/step
114621/114621 [==============================] - 1s 12us/step
114621/114621 [==============================] - 1s 13us/step
114621/114621 [==============================] - 1s 13us/step
114621/114621 [==============================] - 1s 13us/step
114621/114621 [==============================] - 1s 12us/step
114621/114621 [==============================] - 1s 12us/step
114621/114621 [==============================] - 2s 16us/step
114621/114621 [==============================] - 3s 23us/step
114621/114621 [==============================] - 2s 14us/step
114621/114621 [==============================] - 3s 24us/step
114621/114621 [==============================] - 2s 15us/step
114621/114621 [==============================] - 2s 15us/step
114621/114621 [==============================] - 2s 14us/step
114621/114621 [==============================] - 2s 14us/step
114621/114621 [==============================] - 2s 15us/step
114621/114621 [==============================] - 3s 26us/step
114621/114621 [==============================] - 2s 14us/step
114621/114621 [==============================] - 2s 14us/step
114621/114621 [==============================] - 3s 26us/step
114621/114621 [==============================] - 2s 14us/step
114621/114621 [==============================] - 2s 14us/step
114621/114621 [==============================] - 2s 14us/step
114621/114621 [==============================] - 3s 23us/step
114621/114621 [==============================] - 2s 17us/step
114621/114621 [==============================] - 2s 14us/step
114621/114621 [==============================] - 3s 23us/step
114621/114621 [==============================] - 2s 13us/step
114621/114621 [==============================] - 3s 22us/step
114621/114621 [==============================] - 2s 16us/step
114621/114621 [==============================] - 2s 14us/step
114621/114621 [==============================] - 2s 15us/step
```

```
114621/114621 [==============================] - 2s 14us/step
114621/114621 [==============================] - 3s 25us/step
114621/114621 [==============================] - 2s 16us/step
114621/114621 [==============================] - 1s 12us/step
114621/114621 [==============================] - 3s 24us/step
114621/114621 [==============================] - 2s 14us/step
114621/114621 [==============================] - 2s 19us/step
114621/114621 [==============================] - 2s 14us/step
114621/114621 [==============================] - 2s 14us/step
114621/114621 [==============================] - 2s 15us/step
114621/114621 [==============================] - 3s 25us/step
114621/114621 [==============================] - 2s 16us/step
114621/114621 [==============================] - 2s 14us/step
114621/114621 [==============================] - 2s 14us/step
114621/114621 [==============================] - 2s 13us/step
114621/114621 [==============================] - 2s 14us/step
114621/114621 [==============================] - 2s 14us/step
114621/114621 [==============================] - 3s 26us/step
114621/114621 [==============================] - 2s 16us/step
```

23 Part 22: 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 23: What is the mean and the standard deviation of the test accuracy?

Question 24: What is the main advantage of dropout compared to CV for estimating test uncertainty?

[56]:
```
mean = np.mean(accuracy_score)
stan_dev = np.std(accuracy_score)
# Calculate and print mean and std of accuracies
print("Mean of the accuracies: {}".format(mean))
print("Standard Deviation of the accuracies : {}".format(stan_dev))
```

```
Mean of the accuracies: 0.9330750703811646
Standard Deviation of the accuracies : 0.0
```

[68]:
```
from sklearn.model_selection import StratifiedKFold
# initial suitable parameters
```

```
batch_size = 1000
epochs = 20
input_shape = 92
n_nodes = 30
n_layers = 2
i = 0
# Define 10-fold cross validation
KFold = StratifiedKFold(n_splits=10,shuffle=True,random_state=1234)
accuracy_score_k = np.empty(100)
# Loop over cross validation folds
for train_index, test_index in KFold.split(X,Y):
    print("TRAIN:", train_index, "TEST:", test_index)
    X_train, X_test = X[train_index,:], X[test_index]
    Y_train, Y_test = Y[train_index,:], Y[test_index]
    # Calculate class weights for current split
    class_wt = class_weight.compute_class_weight("balanced", np.
 ↪unique(Ytrain),Ytrain[:,0])

    # Rebuild the DNN model, to not continue training on the previously trained␣
 ↪model
    model = build_DNN(input_shape = input_shape, n_layers=n_layers,␣
 ↪n_nodes=n_nodes,learning_rate=0.1,optimizer = "adam",use_bn=True)

    # Fit the model with training set and class weights for this fold
    history = model.fit(X_train, Y_train, epochs = epochs, batch_size =␣
 ↪batch_size, class_weight = class_wt)

    # Evaluate the model using the test set for this fold
    score = model.evaluate(X_test,Y_test)

    # Save the test accuracy in an array
    accuracy_score_k[i] = score[1]
    i+1
# Calculate and print mean and std of accuracies
```

```
TRAIN: [     0      1      2 … 764133 764134 764135] TEST: [    16     21
24 … 764119 764122 764123]
Epoch 1/20
687722/687722 [==============================] - 4s 6us/step - loss: 0.2302 -
accuracy: 0.9058
Epoch 2/20
687722/687722 [==============================] - 3s 4us/step - loss: 0.1527 -
accuracy: 0.9170
Epoch 3/20
687722/687722 [==============================] - 3s 4us/step - loss: 0.1463 -
accuracy: 0.9191
Epoch 4/20
```

```
687722/687722 [==============================] - 3s 4us/step - loss: 0.1429 -
accuracy: 0.9208
Epoch 5/20
687722/687722 [==============================] - 3s 4us/step - loss: 0.1399 -
accuracy: 0.9227
Epoch 6/20
687722/687722 [==============================] - 3s 4us/step - loss: 0.1380 -
accuracy: 0.9238
Epoch 7/20
687722/687722 [==============================] - 3s 5us/step - loss: 0.1357 -
accuracy: 0.9256
Epoch 8/20
687722/687722 [==============================] - 3s 4us/step - loss: 0.1338 -
accuracy: 0.9269
Epoch 9/20
687722/687722 [==============================] - 3s 4us/step - loss: 0.1317 -
accuracy: 0.9274
Epoch 10/20
687722/687722 [==============================] - 3s 4us/step - loss: 0.1303 -
accuracy: 0.9288
Epoch 11/20
687722/687722 [==============================] - 3s 4us/step - loss: 0.1285 -
accuracy: 0.9301
Epoch 12/20
687722/687722 [==============================] - 3s 5us/step - loss: 0.1271 -
accuracy: 0.9306
Epoch 13/20
687722/687722 [==============================] - 3s 4us/step - loss: 0.1259 -
accuracy: 0.9316
Epoch 14/20
687722/687722 [==============================] - 3s 4us/step - loss: 0.1250 -
accuracy: 0.9314
Epoch 15/20
687722/687722 [==============================] - 3s 4us/step - loss: 0.1242 -
accuracy: 0.9318
Epoch 16/20
687722/687722 [==============================] - 3s 4us/step - loss: 0.1231 -
accuracy: 0.9322
Epoch 17/20
687722/687722 [==============================] - 3s 4us/step - loss: 0.1226 -
accuracy: 0.9327
Epoch 18/20
687722/687722 [==============================] - 3s 4us/step - loss: 0.1216 -
accuracy: 0.9330
Epoch 19/20
687722/687722 [==============================] - 3s 4us/step - loss: 0.1209 -
accuracy: 0.9336
Epoch 20/20
```

```
687722/687722 [==============================] - 3s 5us/step - loss: 0.1202 -
accuracy: 0.9337
76414/76414 [==============================] - 1s 16us/step
TRAIN: [     0      2      3 … 764133 764134 764135] TEST: [     1      8
53 … 764096 764114 764117]
Epoch 1/20
687722/687722 [==============================] - 4s 5us/step - loss: 0.2232 -
accuracy: 0.9119
Epoch 2/20
687722/687722 [==============================] - 3s 5us/step - loss: 0.1492 -
accuracy: 0.9186
Epoch 3/20
687722/687722 [==============================] - 3s 4us/step - loss: 0.1439 -
accuracy: 0.9208
Epoch 4/20
687722/687722 [==============================] - 3s 4us/step - loss: 0.1411 -
accuracy: 0.9219
Epoch 5/20
687722/687722 [==============================] - 3s 4us/step - loss: 0.1388 -
accuracy: 0.9231
Epoch 6/20
687722/687722 [==============================] - 3s 4us/step - loss: 0.1370 -
accuracy: 0.9243
Epoch 7/20
687722/687722 [==============================] - 3s 5us/step - loss: 0.1347 -
accuracy: 0.9257
Epoch 8/20
687722/687722 [==============================] - 3s 5us/step - loss: 0.1331 -
accuracy: 0.9266
Epoch 9/20
687722/687722 [==============================] - 3s 5us/step - loss: 0.1317 -
accuracy: 0.9277
Epoch 10/20
687722/687722 [==============================] - 3s 5us/step - loss: 0.1291 -
accuracy: 0.9294
Epoch 11/20
687722/687722 [==============================] - 3s 4us/step - loss: 0.1287 -
accuracy: 0.9295
Epoch 12/20
687722/687722 [==============================] - 3s 4us/step - loss: 0.1269 -
accuracy: 0.9308
Epoch 13/20
687722/687722 [==============================] - 3s 4us/step - loss: 0.1261 -
accuracy: 0.9307
Epoch 14/20
687722/687722 [==============================] - 3s 4us/step - loss: 0.1248 -
accuracy: 0.9313
Epoch 15/20
```

```
687722/687722 [==============================] - 3s 4us/step - loss: 0.1240 -
accuracy: 0.9321
Epoch 16/20
687722/687722 [==============================] - 3s 4us/step - loss: 0.1230 -
accuracy: 0.9326
Epoch 17/20
687722/687722 [==============================] - 3s 5us/step - loss: 0.1219 -
accuracy: 0.9331
Epoch 18/20
687722/687722 [==============================] - 3s 5us/step - loss: 0.1213 -
accuracy: 0.9332
Epoch 19/20
687722/687722 [==============================] - 3s 4us/step - loss: 0.1207 -
accuracy: 0.9335
Epoch 20/20
687722/687722 [==============================] - 3s 5us/step - loss: 0.1208 -
accuracy: 0.9332
76414/76414 [==============================] - 2s 31us/step
TRAIN: [     0      1      2 … 764133 764134 764135] TEST: [     3      5
9 … 764125 764127 764128]
Epoch 1/20
687722/687722 [==============================] - 4s 6us/step - loss: 0.2297 -
accuracy: 0.9094
Epoch 2/20
687722/687722 [==============================] - 3s 4us/step - loss: 0.1505 -
accuracy: 0.9184
Epoch 3/20
687722/687722 [==============================] - 3s 5us/step - loss: 0.1447 -
accuracy: 0.9199
Epoch 4/20
687722/687722 [==============================] - 3s 5us/step - loss: 0.1410 -
accuracy: 0.9221
Epoch 5/20
687722/687722 [==============================] - 3s 5us/step - loss: 0.1388 -
accuracy: 0.9232
Epoch 6/20
687722/687722 [==============================] - 3s 4us/step - loss: 0.1365 -
accuracy: 0.9247
Epoch 7/20
687722/687722 [==============================] - 3s 4us/step - loss: 0.1349 -
accuracy: 0.9258
Epoch 8/20
687722/687722 [==============================] - 3s 4us/step - loss: 0.1336 -
accuracy: 0.9265
Epoch 9/20
687722/687722 [==============================] - 3s 4us/step - loss: 0.1320 -
accuracy: 0.9274
Epoch 10/20
```

```
687722/687722 [==============================] - 3s 4us/step - loss: 0.1310 -
accuracy: 0.9283
Epoch 11/20
687722/687722 [==============================] - 3s 4us/step - loss: 0.1292 -
accuracy: 0.9293
Epoch 12/20
687722/687722 [==============================] - 3s 4us/step - loss: 0.1281 -
accuracy: 0.9299
Epoch 13/20
687722/687722 [==============================] - 3s 4us/step - loss: 0.1275 -
accuracy: 0.9302
Epoch 14/20
687722/687722 [==============================] - 4s 5us/step - loss: 0.1267 -
accuracy: 0.9303
Epoch 15/20
687722/687722 [==============================] - 3s 5us/step - loss: 0.1254 -
accuracy: 0.9312
Epoch 16/20
687722/687722 [==============================] - 3s 5us/step - loss: 0.1247 -
accuracy: 0.9315
Epoch 17/20
687722/687722 [==============================] - 3s 4us/step - loss: 0.1244 -
accuracy: 0.9313
Epoch 18/20
687722/687722 [==============================] - 3s 5us/step - loss: 0.1228 -
accuracy: 0.9324
Epoch 19/20
687722/687722 [==============================] - 3s 4us/step - loss: 0.1234 -
accuracy: 0.9315
Epoch 20/20
687722/687722 [==============================] - 3s 4us/step - loss: 0.1218 -
accuracy: 0.9325
76414/76414 [==============================] - 1s 13us/step
TRAIN: [     0      1      2 … 764133 764134 764135] TEST: [    28     48
55 … 764099 764105 764132]
Epoch 1/20
687722/687722 [==============================] - 4s 5us/step - loss: 0.2258 -
accuracy: 0.9098
Epoch 2/20
687722/687722 [==============================] - 3s 5us/step - loss: 0.1513 -
accuracy: 0.9177
Epoch 3/20
687722/687722 [==============================] - 3s 5us/step - loss: 0.1456 -
accuracy: 0.9195
Epoch 4/20
687722/687722 [==============================] - 3s 4us/step - loss: 0.1428 -
accuracy: 0.9202
Epoch 5/20
```

```
687722/687722 [==============================] - 3s 4us/step - loss: 0.1395 -
accuracy: 0.9223
Epoch 6/20
687722/687722 [==============================] - 3s 4us/step - loss: 0.1370 -
accuracy: 0.9239
Epoch 7/20
687722/687722 [==============================] - 3s 4us/step - loss: 0.1343 -
accuracy: 0.9260
Epoch 8/20
687722/687722 [==============================] - 3s 4us/step - loss: 0.1321 -
accuracy: 0.9275
Epoch 9/20
687722/687722 [==============================] - 3s 4us/step - loss: 0.1290 -
accuracy: 0.9299
Epoch 10/20
687722/687722 [==============================] - 3s 4us/step - loss: 0.1274 -
accuracy: 0.9307
Epoch 11/20
687722/687722 [==============================] - 3s 4us/step - loss: 0.1261 -
accuracy: 0.9307
Epoch 12/20
687722/687722 [==============================] - 3s 4us/step - loss: 0.1251 -
accuracy: 0.9312
Epoch 13/20
687722/687722 [==============================] - 4s 6us/step - loss: 0.1237 -
accuracy: 0.9320
Epoch 14/20
687722/687722 [==============================] - 4s 6us/step - loss: 0.1227 -
accuracy: 0.9322
Epoch 15/20
687722/687722 [==============================] - 3s 4us/step - loss: 0.1217 -
accuracy: 0.9329
Epoch 16/20
687722/687722 [==============================] - 3s 5us/step - loss: 0.1206 -
accuracy: 0.9331
Epoch 17/20
687722/687722 [==============================] - 3s 4us/step - loss: 0.1202 -
accuracy: 0.9334
Epoch 18/20
687722/687722 [==============================] - 3s 4us/step - loss: 0.1192 -
accuracy: 0.9339: 0s - loss: 0.1191 - accuracy:
Epoch 19/20
687722/687722 [==============================] - 3s 4us/step - loss: 0.1187 -
accuracy: 0.9344
Epoch 20/20
687722/687722 [==============================] - 3s 4us/step - loss: 0.1178 -
accuracy: 0.9348
76414/76414 [==============================] - 1s 13us/step
```

```
TRAIN: [     0      1      2 … 764132 764133 764134] TEST: [    27     30
44 … 764115 764130 764135]
Epoch 1/20
687722/687722 [==============================] - 3s 5us/step - loss: 0.2268 -
accuracy: 0.9097
Epoch 2/20
687722/687722 [==============================] - 3s 5us/step - loss: 0.1518 -
accuracy: 0.9175
Epoch 3/20
687722/687722 [==============================] - 3s 5us/step - loss: 0.1455 -
accuracy: 0.9197
Epoch 4/20
687722/687722 [==============================] - 3s 5us/step - loss: 0.1418 -
accuracy: 0.9213
Epoch 5/20
687722/687722 [==============================] - 3s 5us/step - loss: 0.1390 -
accuracy: 0.9230
Epoch 6/20
687722/687722 [==============================] - 3s 5us/step - loss: 0.1367 -
accuracy: 0.9245
Epoch 7/20
687722/687722 [==============================] - 4s 5us/step - loss: 0.1341 -
accuracy: 0.9263
Epoch 8/20
687722/687722 [==============================] - 3s 5us/step - loss: 0.1325 -
accuracy: 0.9273
Epoch 9/20
687722/687722 [==============================] - 3s 5us/step - loss: 0.1313 -
accuracy: 0.9277
Epoch 10/20
687722/687722 [==============================] - 3s 4us/step - loss: 0.1300 -
accuracy: 0.9290
Epoch 11/20
687722/687722 [==============================] - 3s 5us/step - loss: 0.1285 -
accuracy: 0.9297
Epoch 12/20
687722/687722 [==============================] - 3s 5us/step - loss: 0.1279 -
accuracy: 0.9294
Epoch 13/20
687722/687722 [==============================] - 3s 5us/step - loss: 0.1263 -
accuracy: 0.9309
Epoch 14/20
687722/687722 [==============================] - 3s 5us/step - loss: 0.1258 -
accuracy: 0.9314
Epoch 15/20
687722/687722 [==============================] - 3s 5us/step - loss: 0.1239 -
accuracy: 0.9320
Epoch 16/20
```

```
687722/687722 [==============================] - 3s 5us/step - loss: 0.1236 -
accuracy: 0.9319
Epoch 17/20
687722/687722 [==============================] - 3s 5us/step - loss: 0.1225 -
accuracy: 0.9326
Epoch 18/20
687722/687722 [==============================] - 4s 5us/step - loss: 0.1222 -
accuracy: 0.9325
Epoch 19/20
687722/687722 [==============================] - 3s 5us/step - loss: 0.1205 -
accuracy: 0.9335
Epoch 20/20
687722/687722 [==============================] - 4s 5us/step - loss: 0.1201 -
accuracy: 0.9337
76414/76414 [==============================] - 2s 27us/step
TRAIN: [     0      1      2 … 764133 764134 764135] TEST: [    15     35
51 … 764091 764101 764104]
Epoch 1/20
687722/687722 [==============================] - 5s 7us/step - loss: 0.2219 -
accuracy: 0.9102
Epoch 2/20
687722/687722 [==============================] - 3s 4us/step - loss: 0.1519 -
accuracy: 0.9170
Epoch 3/20
687722/687722 [==============================] - 3s 4us/step - loss: 0.1456 -
accuracy: 0.9194
Epoch 4/20
687722/687722 [==============================] - 3s 4us/step - loss: 0.1424 -
accuracy: 0.9213
Epoch 5/20
687722/687722 [==============================] - 3s 5us/step - loss: 0.1400 -
accuracy: 0.9224
Epoch 6/20
687722/687722 [==============================] - 3s 4us/step - loss: 0.1381 -
accuracy: 0.9240
Epoch 7/20
687722/687722 [==============================] - 3s 4us/step - loss: 0.1361 -
accuracy: 0.9250
Epoch 8/20
687722/687722 [==============================] - 3s 4us/step - loss: 0.1339 -
accuracy: 0.9263
Epoch 9/20
687722/687722 [==============================] - 3s 5us/step - loss: 0.1331 -
accuracy: 0.9269
Epoch 10/20
687722/687722 [==============================] - 3s 4us/step - loss: 0.1311 -
accuracy: 0.9280
Epoch 11/20
```

```
687722/687722 [==============================] - 3s 4us/step - loss: 0.1304 -
accuracy: 0.9284
Epoch 12/20
687722/687722 [==============================] - 3s 5us/step - loss: 0.1290 -
accuracy: 0.9292
Epoch 13/20
687722/687722 [==============================] - 3s 5us/step - loss: 0.1280 -
accuracy: 0.9296
Epoch 14/20
687722/687722 [==============================] - 3s 4us/step - loss: 0.1273 -
accuracy: 0.9299
Epoch 15/20
687722/687722 [==============================] - 3s 4us/step - loss: 0.1260 -
accuracy: 0.9305
Epoch 16/20
687722/687722 [==============================] - 3s 4us/step - loss: 0.1251 -
accuracy: 0.9308
Epoch 17/20
687722/687722 [==============================] - 3s 4us/step - loss: 0.1238 -
accuracy: 0.9320
Epoch 18/20
687722/687722 [==============================] - 3s 4us/step - loss: 0.1235 -
accuracy: 0.9318
Epoch 19/20
687722/687722 [==============================] - 3s 4us/step - loss: 0.1234 -
accuracy: 0.9316
Epoch 20/20
687722/687722 [==============================] - ETA: 0s - loss: 0.1224 -
accuracy: 0.93 - 3s 4us/step - loss: 0.1224 - accuracy: 0.9319
76414/76414 [==============================] - 1s 13us/step
TRAIN: [     0      1      2 … 764133 764134 764135] TEST: [     6     11
18 … 764120 764126 764131]
Epoch 1/20
687723/687723 [==============================] - 4s 6us/step - loss: 0.2209 -
accuracy: 0.9120
Epoch 2/20
687723/687723 [==============================] - 3s 4us/step - loss: 0.1505 -
accuracy: 0.9173
Epoch 3/20
687723/687723 [==============================] - 3s 5us/step - loss: 0.1458 -
accuracy: 0.9186
Epoch 4/20
687723/687723 [==============================] - 3s 5us/step - loss: 0.1419 -
accuracy: 0.9210
Epoch 5/20
687723/687723 [==============================] - 4s 5us/step - loss: 0.1396 -
accuracy: 0.9226
Epoch 6/20
```

```
687723/687723 [==============================] - 3s 5us/step - loss: 0.1365 -
accuracy: 0.9246
Epoch 7/20
687723/687723 [==============================] - 4s 5us/step - loss: 0.1354 -
accuracy: 0.9253
Epoch 8/20
687723/687723 [==============================] - 3s 5us/step - loss: 0.1336 -
accuracy: 0.9264
Epoch 9/20
687723/687723 [==============================] - 4s 5us/step - loss: 0.1321 -
accuracy: 0.9275
Epoch 10/20
687723/687723 [==============================] - 3s 4us/step - loss: 0.1303 -
accuracy: 0.9284
Epoch 11/20
687723/687723 [==============================] - 4s 5us/step - loss: 0.1297 -
accuracy: 0.9291
Epoch 12/20
687723/687723 [==============================] - 3s 5us/step - loss: 0.1289 -
accuracy: 0.9292
Epoch 13/20
687723/687723 [==============================] - 3s 5us/step - loss: 0.1273 -
accuracy: 0.9304
Epoch 14/20
687723/687723 [==============================] - 3s 4us/step - loss: 0.1270 -
accuracy: 0.9305
Epoch 15/20
687723/687723 [==============================] - 3s 4us/step - loss: 0.1256 -
accuracy: 0.9310
Epoch 16/20
687723/687723 [==============================] - 3s 4us/step - loss: 0.1253 -
accuracy: 0.9309
Epoch 17/20
687723/687723 [==============================] - 3s 4us/step - loss: 0.1242 -
accuracy: 0.9318
Epoch 18/20
687723/687723 [==============================] - 3s 4us/step - loss: 0.1240 -
accuracy: 0.9317
Epoch 19/20
687723/687723 [==============================] - 3s 4us/step - loss: 0.1226 -
accuracy: 0.9328
Epoch 20/20
687723/687723 [==============================] - 3s 4us/step - loss: 0.1225 -
accuracy: 0.9328
76413/76413 [==============================] - 1s 13us/step
TRAIN: [     1      2      3 … 764132 764134 764135] TEST: [     0     17
20 … 764093 764103 764133]
Epoch 1/20
```

```
687723/687723 [==============================] - 3s 5us/step - loss: 0.2230 -
accuracy: 0.9111
Epoch 2/20
687723/687723 [==============================] - 3s 4us/step - loss: 0.1522 -
accuracy: 0.9175
Epoch 3/20
687723/687723 [==============================] - 3s 4us/step - loss: 0.1467 -
accuracy: 0.9189
Epoch 4/20
687723/687723 [==============================] - 3s 4us/step - loss: 0.1433 -
accuracy: 0.9209
Epoch 5/20
687723/687723 [==============================] - 3s 4us/step - loss: 0.1407 -
accuracy: 0.9222
Epoch 6/20
687723/687723 [==============================] - 3s 4us/step - loss: 0.1386 -
accuracy: 0.9229
Epoch 7/20
687723/687723 [==============================] - 3s 4us/step - loss: 0.1365 -
accuracy: 0.9244
Epoch 8/20
687723/687723 [==============================] - 3s 4us/step - loss: 0.1344 -
accuracy: 0.9259
Epoch 9/20
687723/687723 [==============================] - 3s 4us/step - loss: 0.1325 -
accuracy: 0.9274
Epoch 10/20
687723/687723 [==============================] - 3s 4us/step - loss: 0.1312 -
accuracy: 0.9284
Epoch 11/20
687723/687723 [==============================] - 3s 4us/step - loss: 0.1301 -
accuracy: 0.9289
Epoch 12/20
687723/687723 [==============================] - 3s 4us/step - loss: 0.1288 -
accuracy: 0.9297
Epoch 13/20
687723/687723 [==============================] - 3s 4us/step - loss: 0.1273 -
accuracy: 0.9306
Epoch 14/20
687723/687723 [==============================] - 3s 4us/step - loss: 0.1259 -
accuracy: 0.9315
Epoch 15/20
687723/687723 [==============================] - 3s 4us/step - loss: 0.1246 -
accuracy: 0.9323
Epoch 16/20
687723/687723 [==============================] - 3s 4us/step - loss: 0.1243 -
accuracy: 0.9323
Epoch 17/20
```

```
687723/687723 [==============================] - 3s 4us/step - loss: 0.1234 -
accuracy: 0.9325
Epoch 18/20
687723/687723 [==============================] - 3s 4us/step - loss: 0.1220 -
accuracy: 0.9333
Epoch 19/20
687723/687723 [==============================] - 3s 4us/step - loss: 0.1216 -
accuracy: 0.9332
Epoch 20/20
687723/687723 [==============================] - 3s 4us/step - loss: 0.1210 -
accuracy: 0.9334
76413/76413 [==============================] - 1s 13us/step
TRAIN: [     0      1      3 … 764132 764133 764135] TEST: [     2      7
19 … 764121 764124 764134]
Epoch 1/20
687723/687723 [==============================] - 3s 5us/step - loss: 0.2266 -
accuracy: 0.9110
Epoch 2/20
687723/687723 [==============================] - 3s 4us/step - loss: 0.1510 -
accuracy: 0.9176
Epoch 3/20
687723/687723 [==============================] - 3s 4us/step - loss: 0.1459 -
accuracy: 0.9192
Epoch 4/20
687723/687723 [==============================] - 3s 4us/step - loss: 0.1427 -
accuracy: 0.9206
Epoch 5/20
687723/687723 [==============================] - 3s 4us/step - loss: 0.1401 -
accuracy: 0.9223
Epoch 6/20
687723/687723 [==============================] - 3s 4us/step - loss: 0.1378 -
accuracy: 0.9238
Epoch 7/20
687723/687723 [==============================] - 3s 4us/step - loss: 0.1358 -
accuracy: 0.9250
Epoch 8/20
687723/687723 [==============================] - 3s 4us/step - loss: 0.1341 -
accuracy: 0.9259
Epoch 9/20
687723/687723 [==============================] - 3s 4us/step - loss: 0.1328 -
accuracy: 0.9264
Epoch 10/20
687723/687723 [==============================] - 3s 4us/step - loss: 0.1308 -
accuracy: 0.9286
Epoch 11/20
687723/687723 [==============================] - 3s 5us/step - loss: 0.1290 -
accuracy: 0.9296
Epoch 12/20
```

```
687723/687723 [==============================] - 3s 4us/step - loss: 0.1280 -
accuracy: 0.9300
Epoch 13/20
687723/687723 [==============================] - 3s 4us/step - loss: 0.1265 -
accuracy: 0.9312
Epoch 14/20
687723/687723 [==============================] - 3s 4us/step - loss: 0.1253 -
accuracy: 0.9315
Epoch 15/20
687723/687723 [==============================] - 3s 4us/step - loss: 0.1240 -
accuracy: 0.9325
Epoch 16/20
687723/687723 [==============================] - 3s 4us/step - loss: 0.1231 -
accuracy: 0.9324
Epoch 17/20
687723/687723 [==============================] - 3s 4us/step - loss: 0.1223 -
accuracy: 0.9327
Epoch 18/20
687723/687723 [==============================] - 3s 4us/step - loss: 0.1210 -
accuracy: 0.9336
Epoch 19/20
687723/687723 [==============================] - 3s 5us/step - loss: 0.1205 -
accuracy: 0.9335
Epoch 20/20
687723/687723 [==============================] - 3s 5us/step - loss: 0.1199 -
accuracy: 0.9339
76413/76413 [==============================] - 1s 12us/step
TRAIN: [     0      1      2 … 764133 764134 764135] TEST: [     4     25
31 … 764110 764113 764129]
Epoch 1/20
687723/687723 [==============================] - 3s 5us/step - loss: 0.2269 -
accuracy: 0.9085
Epoch 2/20
687723/687723 [==============================] - 3s 4us/step - loss: 0.1516 -
accuracy: 0.9170
Epoch 3/20
687723/687723 [==============================] - 3s 4us/step - loss: 0.1456 -
accuracy: 0.9193
Epoch 4/20
687723/687723 [==============================] - 3s 4us/step - loss: 0.1422 -
accuracy: 0.9208
Epoch 5/20
687723/687723 [==============================] - 3s 4us/step - loss: 0.1400 -
accuracy: 0.9220
Epoch 6/20
687723/687723 [==============================] - 3s 4us/step - loss: 0.1378 -
accuracy: 0.9235
Epoch 7/20
```

```
687723/687723 [==============================] - 3s 4us/step - loss: 0.1363 -
accuracy: 0.9244
Epoch 8/20
687723/687723 [==============================] - 3s 4us/step - loss: 0.1350 -
accuracy: 0.9253
Epoch 9/20
687723/687723 [==============================] - 3s 4us/step - loss: 0.1337 -
accuracy: 0.9262
Epoch 10/20
687723/687723 [==============================] - 3s 4us/step - loss: 0.1327 -
accuracy: 0.9266
Epoch 11/20
687723/687723 [==============================] - 3s 4us/step - loss: 0.1312 -
accuracy: 0.9281
Epoch 12/20
687723/687723 [==============================] - 3s 4us/step - loss: 0.1304 -
accuracy: 0.9282
Epoch 13/20
687723/687723 [==============================] - 3s 4us/step - loss: 0.1290 -
accuracy: 0.9291
Epoch 14/20
687723/687723 [==============================] - 3s 4us/step - loss: 0.1287 -
accuracy: 0.9293
Epoch 15/20
687723/687723 [==============================] - 3s 4us/step - loss: 0.1278 -
accuracy: 0.9296
Epoch 16/20
687723/687723 [==============================] - 3s 4us/step - loss: 0.1264 -
accuracy: 0.9303
Epoch 17/20
687723/687723 [==============================] - 3s 4us/step - loss: 0.1256 -
accuracy: 0.9313
Epoch 18/20
687723/687723 [==============================] - 3s 4us/step - loss: 0.1252 -
accuracy: 0.9314
Epoch 19/20
687723/687723 [==============================] - 3s 4us/step - loss: 0.1246 -
accuracy: 0.9315
Epoch 20/20
687723/687723 [==============================] - 3s 4us/step - loss: 0.1234 -
accuracy: 0.9324
76413/76413 [==============================] - 1s 13us/step
```

[71]:
```
mean_k = np.mean(accuracy_score_k)*100
stan_dev_k = np.std(accuracy_score_k)
# Calculate and print mean and std of accuracies
print("Mean of the accuracies: {}".format(mean_k))
```

```
print("Standard Deviation of the accuracies : {}".format(stan_dev_k))
```

```
Mean of the accuracies: 0.9352073669433594
Standard Deviation of the accuracies : 0.09305195811982034
```

24 Part 23: DNN regression

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

Question 25: How would you change the DNN in order to use it for regression instead?

24.1 Report

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