Report Submission Information (must be completed before submitting report!)

- Student Full Name and Number: Hesam ShokriAsri

Workshop 3 – Deep Learning [2 weeks]

Topics Covered

- Neural Networks (NNs) and Deep Learning
- Training NNs and optimisation
- Time series data and estimation
- Long Short-Term Memory (LSTM) – an example Recurrent Neural Network (RNN)
- Reinforcement learning basics
- Multi-armed bandits

Topic_mindmap

Topic Notes

The history of artificial neural networks (https://en.wikipedia.org/wiki/Artificial_neural_network#History) is full of ups and downs. People got excited about and ignored them multiple times since mid 20th century. Since the start of the 21st century, artificial neural networks have enjoyed a big comeback in the form of deep learning and DNNs (https://en.wikipedia.org/wiki/Deep_learning). This last wave rides on important and un-ignorable trends including rapid advances in computing (CPUs, GPUs, and specialised hardware), availability of sensors/data, and abundance of storage. While modern DNNs have already been applied to traditional problems in computer science such as image recognition and information retrieval with great success, their influence on engineering applications is only starting to be felt.

In this workshop, you will learn about basics of time-series analysis and how to solve various machine learning problems using DNNs. Doing this yourself will give you a chance to connect theoretical knowledge and practical usage. We will start with some of the data sets we have used in the previous workshop, which will make it easier to compare and contrast different approaches. More interesting problems will be posed as open-ended (and optional) tasks.

You will also familiarise yourself with Keras, Python Deep Learning Library (https://keras.io/), which is chosen for its popularity but most importantly, ease-of-use. Keras often uses the underlying and more flexible TensorFlow (https://www.tensorflow.org/) framework. As usual, the tools and data in this workshop are chosen completely for educational reasons (simplicity, accessibility, cost). There are and will be better Deep Learning frameworks and more complex data sets but it is not realistic to cover all in a limited time.

> In the future, you should consider learning additional Deep Learning software packages and libraries. Finding the right tool for the right job is an important skill obtained through knowledge and experience.

Table of Contents

Workflow and Assessment

This subject follows a problem- and project-oriented approach. In this learning workflow, the focus is on solving practical (engineering) problems, which motivate acquiring theoretical (background) knowledge at the same time.

Objectives

- Use these problems as a motivation to learn the fundamentals of deep learning covered in lectures.
- Gain hands-on experience with deep learning and deep neural networks.
- Familiarise yourself with Python and Keras for ***Deep Neural Networks (DNNs)*** as widely-used practical software tools.
- Basics of time series analysis relevant to engineering.
- Solve basic machine learning problems using DNNs and the Keras library.
- Solve basic reinforcement learning problems using multi-armed bandit models.
- Connect theoretical knowledge and practical usage by doing it yourself.

Common objectives of all workshops

Gain hands-on experience and learn by doing! Understand how theoretical knowledge discussed in lectures relates to practice. Develop motivation for gaining further theoretical and practical knowledge beyond the subject material.

> **Self-learning** is one of the most important skills that you should acquire as a student. Today, self-learning is much easier than it used to be thanks to a plethora of online resources.

Assessment Process

1. Follow the procedures described below, perform the given tasks, and answer the workshop questions **in this Python notebook itself! The resulting notebook will be your Workshop Report!**
2. Submit the workshop report at the announced deadline
3. Demonstrators will conduct a brief (5min) oral quiz on your submitted report in the subsequent weeks.
4. Your workshop marks will be a combination of the report you submitted and oral quiz results.

> **The goal is to learn**, NOT blindly follow the procedures in the fastest possible way! **Do not simply copy-paste answers (from Internet, friends, etc.). You can and should use all available resources but only to develop your own**

Additional packages to install

In this workshop, we will use Tensorflow and Keras. If you are using a lab computer, ***these packages should already be there in your Anaconda environment (please check!).*** If not (or if you are using your own device), the best way to go forward is by creating a new environment (e.g you can name it `tfenv`), which you can do inside of ***Anaconda Navigator*** and after that, installing Tensorflow in your new environment.

Alternatively, you can create the environment using [these instructions (https://www.pugetsystems.com/labs/hpc/How-to-Install-TensorFlow-with-GPU-Support-on-Windows-10-Without-Installing-CUDA-UPDATED-1419/#Step3%29CreateaPython%5C)](https://www.pugetsystems.com/labs/hpc/How-to-Install-TensorFlow-with-GPU-Support-on-Windows-10-Without-Installing-CUDA-UPDATED-1419/#Step3%29CreateaPython%5C) followed by the command `conda install tensorflow-gpu` or `conda install tensorflow-gpu` if you wish to make use of your computer's [NVIDIA graphics card (https://www.tensorflow.org/install/gpu)](https://www.tensorflow.org/install/gpu). Note that installing tensorflow in either case pulls all the necessary packages including `scipy` , `scikit` etc. ***In case this does not happen, be sure to install (*** *`scikit-learn`* ***,*** *`matplotlib`* ***,*** *`pandas`* ***, and any others that cause*** *`import error`* ***s) one by one.***

Another package we will need is [**statsmodels** (https://www.statsmodels.org)](https://www.statsmodels.org). You can also install it directly using ***Anaconda Navigator.*** or following instructions on their website.

Ask for help from your demonstrator in case you need it.

Don't forget to launch your notebook from the right environment!

Section 1: DNNs for Classification

We will use first the now-familiar two-moon data set as an exercise for classifying with DNNs. This will help you to learn basics of Keras and deep learning on a problem which you have already solved with classical ML methods.

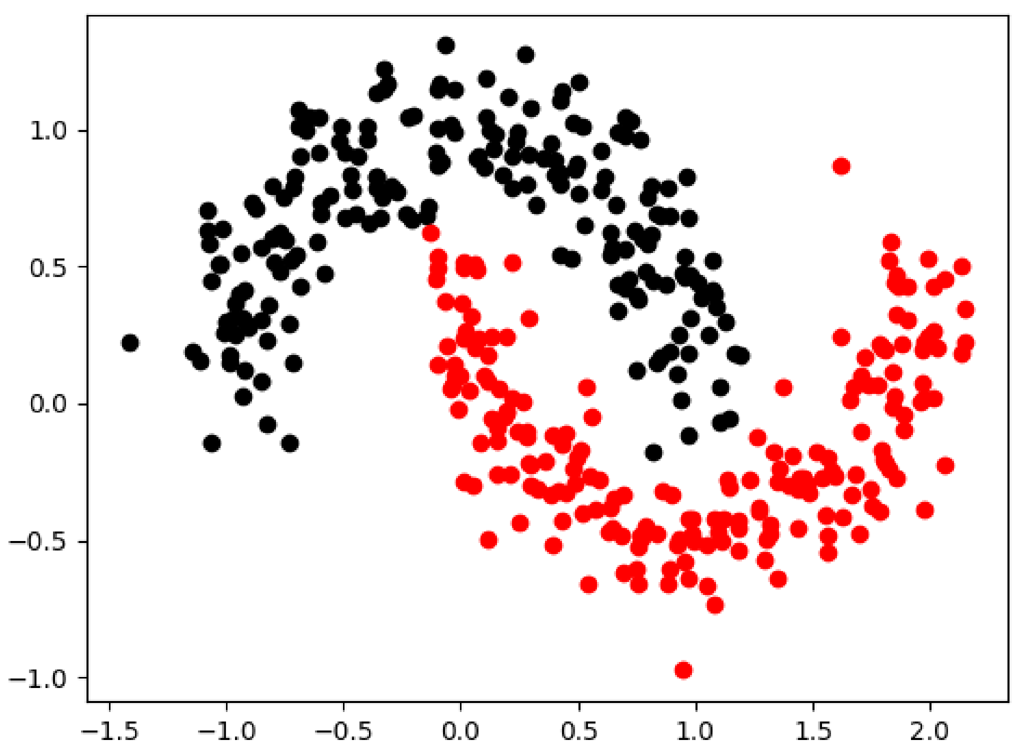
Note remember that Scikit Learn uses the [numpy random state (https://docs.scipy.org/doc/numpy-1.15.1/reference/generated/numpy.random.seed.html#numpy.random.seed)](https://docs.scipy.org/doc/numpy-1.15.1/reference/generated/numpy.random.seed.html#numpy.random.seed). See the code below and uncomment as instructed for repeatable results.

Important Note on Random Number/Vector Generation

Each group has to use a different number seed (which is an arbitrary number as illustrated above) and groups cannot share seeds. The pseudo-randomness is used here to create diversity. Otherwise, if groups use the same seed, the results will be the same (opening the door to plagiarism) and significant number of points will be taken off! As a practical hint, you can use a modified-combination of your student numbers.

In [1]:
```
%matplotlib notebook
import pandas as pd
import numpy as np
from scipy import optimize
import matplotlib.pyplot as plt
import matplotlib
from sklearn.model_selection import train_test_split
from sklearn import cluster, datasets, mixture
from sklearn.cluster import KMeans
from sklearn.utils import shuffle

# Set a random seed as you did in optimisation workshop by uncommenting the li
np.random.seed(691844)

# Create a new moons data set
new_moons = datasets.make_moons(n_samples=400, noise=0.15)
Xm = new_moons[0] # data points
ym = new_moons[1] # 0, 1 labels of class, 200 each - giving us the ground trut

# Visualise the data set
order_ind = np.argsort(ym) # order labels, 200 each class
Xm1 = Xm[order_ind[0:200]]   # class 1 - only for visualisation
Xm2 = Xm[order_ind[201:400]] # class 2 - only for visualisation
plt.figure()
plt.scatter(Xm1[:,0], Xm1[:,1], color='black')
plt.scatter(Xm2[:,0], Xm2[:,1], color='red')
plt.show()
```

```
<IPython.core.display.Javascript object>
```



In [4]:
```
# split into training and test sets
Xmtrain, Xmtest, ymtrain, ymtest = train_test_split(Xm, ym)
```

Example 1.1: DNN with Keras

We first define our neural network (model) and compile it with an optimisation method, loss function, and metrics relevant to our problem. See the following documents as a starting point:

- Keras documentation, guide to sequential model (https://www.tensorflow.org/guide/keras/sequential_model)
- Tensorflow 2 Keras API (https://www.tensorflow.org/api_docs/python/tf/keras)

We are now using Tensorflow 2 (TF2) but a lot of the online material is on TF1. Therefore, you cannot use those scripts directly anymore but it is easy to modify them to TF2!

Additional information you might find helpful is available all over the web, e.g. evaluating performance (https://machinelearningmastery.com/evaluate-performance-deep-learning-models-keras/) and Reduce Overfitting (https://machinelearningmastery.com/how-to-reduce-overfitting-in-deep-neural-networks-with-weight-constraints-in-keras/)

Reproducibility and Pseudo-randomness

It is possible to get reproducible results with Keras (https://machinelearningmastery.com/reproducible-results-neural-networks-keras/). However, this standard method (as implemented in the code below) works only with the CPU implementation of tensorflow as far as I understand.

Please don't forget to change the random seed in the code below and choose a group-specific arbitrary number as in previous workshops for full credit!

If your computer uses CUDA/GPU, don't worry about reproducibility for now. If you really wish to learn more about reproducibility with CUDA/GPU ***optionally*** you can have a look at this project (https://github.com/NVIDIA/tensorflow-determinism).

In [17]:
```
%load_ext tensorboard
import datetime
import tensorflow as tf
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.layers import Dropout
from tensorflow.keras import regularizers
#from tensorflow import keras

print(tf.__version__)
print("GPU is", "available" if tf.config.list_physical_devices('GPU') else "NO
```

```
2.6.0
GPU is NOT AVAILABLE
```

In [9]:
```
# CHANGE THE RANDOM SEED FOR YOUR GROUP!
np.random.seed(1320418)
tf.random.set_seed(1320418)

# Define the DNN sequential model

model = Sequential()
model.add(tf.keras.layers.InputLayer(input_shape=(2,)))
model.add(Dense(8, activation='relu'))
model.add(Dense(4, activation='relu'))
model.add(Dense(1, activation='sigmoid'))


model.summary()

model.compile(
    optimizer=tf.keras.optimizers.SGD(),
              loss=tf.keras.losses.BinaryCrossentropy(),
              metrics=[tf.keras.metrics.BinaryAccuracy()]
)

# log results
log_dir = "logs/" + datetime.datetime.now().strftime("%Y%m%d-%H%M%S")
tensorboard_callback = tf.keras.callbacks.TensorBoard(log_dir=log_dir, histogr
```

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                Output Shape              Param #
=================================================================
dense_3 (Dense)             (None, 8)                 24

dense_4 (Dense)             (None, 4)                 36

dense_5 (Dense)             (None, 1)                 5

=================================================================
Total params: 65
Trainable params: 65
Non-trainable params: 0
_________________________________________________________________
```

Next, we train the DNN we have created using the training data.

In [10]:
```
# The command below continues training from where you left it!
# If you wish to restart training from beginning rerun the cell above to reini

# Train the model, iterating on the data in batches, record history
train_hist = model.fit(Xmtrain, ymtrain, epochs=100, batch_size=16, verbose=1,
```

```
Epoch 62/100
19/19 [==============================] - 0s 2ms/step - loss: 0.3375 - binar
y_accuracy: 0.8967
Epoch 63/100
19/19 [==============================] - 0s 1ms/step - loss: 0.3353 - binar
y_accuracy: 0.8967
Epoch 64/100
19/19 [==============================] - 0s 2ms/step - loss: 0.3332 - binar
y_accuracy: 0.8933
Epoch 65/100
19/19 [==============================] - 0s 2ms/step - loss: 0.3312 - binar
y_accuracy: 0.8967
Epoch 66/100
19/19 [==============================] - 0s 2ms/step - loss: 0.3291 - binar
y_accuracy: 0.8933
Epoch 67/100
19/19 [==============================] - 0s 1ms/step - loss: 0.3274 - binar
y_accuracy: 0.8967
Epoch 68/100
```

Note that the accuracy and loss start from different values whenever you restart the model and you end up with a different final accuracy and loss values whenever you train it. This is due to random initialisation and local minimum solutions in training optimisation. However, since the ***fit*** command is stateful and continues training from where it left, the results improve. How many epochs are needed to get over 90% accuracy?

If for some reason you wish to restart training from the beginning, rerun the previous cell to reinitialise the model!

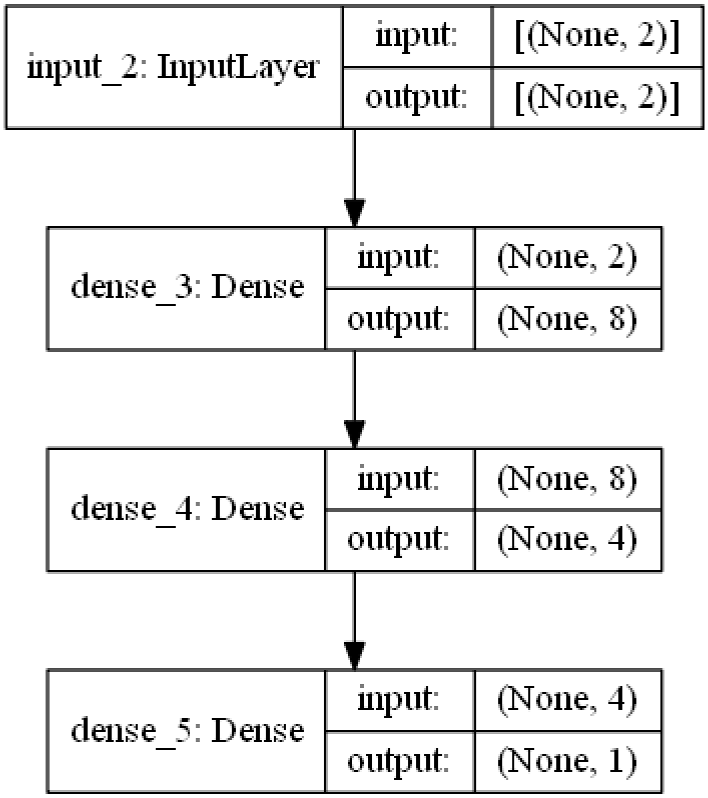
Below, we look closer at how the network is structured and which parameters are trained.

In [11]:
```
print(model.summary())
weights = model.get_weights() # Getting params
print(weights)
```

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                Output Shape              Param #
=================================================================
dense_3 (Dense)             (None, 8)                 24

_________________________________________________________________
dense_4 (Dense)             (None, 4)                 36

_________________________________________________________________
dense_5 (Dense)             (None, 1)                 5
=================================================================
Total params: 65
Trainable params: 65
Non-trainable params: 0
_________________________________________________________________
None
[array([[-0.35827398,  0.21174975,  0.7244337 ,  0.5820165 ,  0.34172708,
         0.14382075, -0.8651151 ,  0.03767662],
       [ 1.1554811 , -0.39821798, -0.25713018,  1.057208  , -0.7138594 ,
         0.19046603, -0.537145  , -0.28939357]], dtype=float32), array([ 0.43
069527,  0.06618733,  0.36842188, -0.03669309, -0.18726078,
       -0.01591526,  0.05995458,  0.16286527], dtype=float32), array([[-0.706
73907, -0.49189553,  1.2781514 ,  0.22590277],
       [-0.01375468,  0.26701468, -0.3114559 ,  0.30325425],
       [-0.31405386, -0.706759  , -0.42641553, -0.64938194],
       [ 0.01812678,  0.05346312,  0.51048344,  0.5771975 ],
       [ 0.54333216,  0.3286062 , -0.16933812,  0.5601467 ],
       [ 0.15851106, -0.62405044,  0.39168406, -0.2750819 ],
       [-0.5465781 , -0.53358483,  0.44008312,  0.1027263 ],
       [-0.3663253 ,  0.7047216 , -0.13987474, -0.41109043]],
      dtype=float32), array([-0.13161053, -0.01363175,  0.18402617,  0.142703
12], dtype=float32), array([[-0.93917173],
       [-0.8601782 ],
       [-1.7573377 ],
       [-0.7935428 ]], dtype=float32), array([1.4918351], dtype=float32)]
```

In [12]:
```
# Plot model graph
# you need to install pydot and graphviz via anaconda for this to work! restar
# A better alternative is to use the tensorboard below!

tf.keras.utils.plot_model(model, to_file='model.png', show_shapes=True, show_l
```

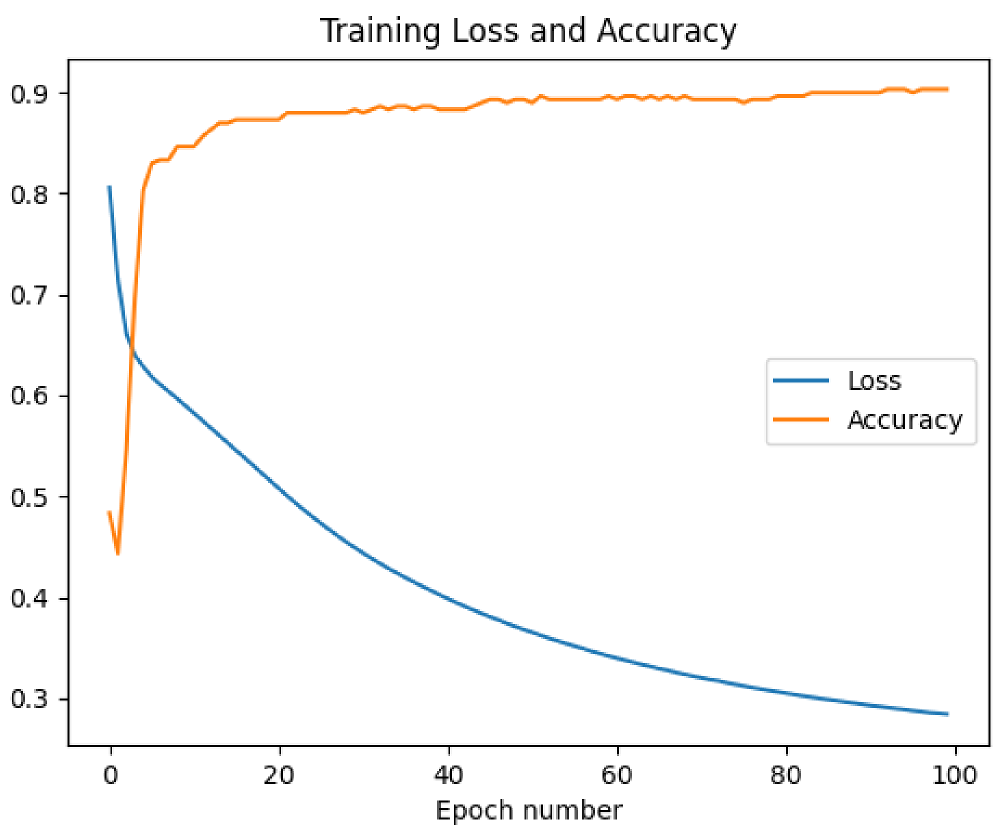
Out[12]:



We can print the the actual score we have chosen and visualise the evolution of loss and accuracy over training epochs.

In [13]:
```
score = model.evaluate(Xmtest, ymtest, batch_size=16, verbose=2)
print(score)

#train_hist.history

plt.figure()
plt.plot(train_hist.history['loss'])
plt.plot(train_hist.history['binary_accuracy'])
plt.xlabel('Epoch number')
plt.title('Training Loss and Accuracy')
plt.legend(['Loss', 'Accuracy'], loc='center right')
plt.show()
```

```
7/7 - 0s - loss: 0.3664 - binary_accuracy: 0.8400
[0.3663579523563385, 0.8399999737739563]

<IPython.core.display.Javascript object>
```



Finally, we compute and display the [confusion matrix (https://en.wikipedia.org/wiki/Confusion_matrix)](https://en.wikipedia.org/wiki/Confusion_matrix), using [sklearn's confusion_matrix (https://scikit-learn.org/stable/modules/generated/sklearn.metrics.confusion_matrix.html)](https://scikit-learn.org/stable/modules/generated/sklearn.metrics.confusion_matrix.html) function.

In [14]:
```
from sklearn.metrics import confusion_matrix

ympred = model.predict(Xmtest)
ympredbinary = (ympred > 0.5)

cm = confusion_matrix(ymtest, ympredbinary)

pd.DataFrame(cm, columns=["Pred 0", "Pred 1"], index=["True 0", "True 1"])
```

Out[14]:

	Pred 0	Pred 1
True 0	44	4
True 1	12	40

We can use the tool `tensorboard` to analyse our results! Note the ***log*** directory in the folder where you have run your script.

In [21]:
```
%tensorboard --logdir logs --host localhost --port 6006
```

Question 1.1 [10%]

Use the same two-moon data (Xm, ym) given above for deriving the training and test sets. You can use the default ratio as done before or change it a bit, e.g. 0.3. The range of data values is OK so you can skip data normalisation.

1. Try different DNN structures instead of (8, 4, 1). For example, you can use only one hidden layer or many more layers. You can also use different activation functions as long as you end up with a single node binary classifier. Try also different optimisers and loss functions. Which one works best? Try, observe, and discuss!
2. For the best combination you managed to find, investigate the impact of training epochs and batch sizes on DNN performance. Measure performance in different ways using the metrics from Keras (https://www.tensorflow.org/api_docs/python/tf/keras/metrics) or classical Machine Learning as discussed during ML lectures. You can use the same sklearn library functions as in WS2 to document performance (see e.g. above). Observe the difference between training and test set loss and accuracy. Interpret your results. What does a big difference between training and test set performance mean?
3. Try different regularizers from Keras (https://www.tensorflow.org/api_docs/python/tf/keras/regularizers) to prevent over-fitting. Document your results and observations.

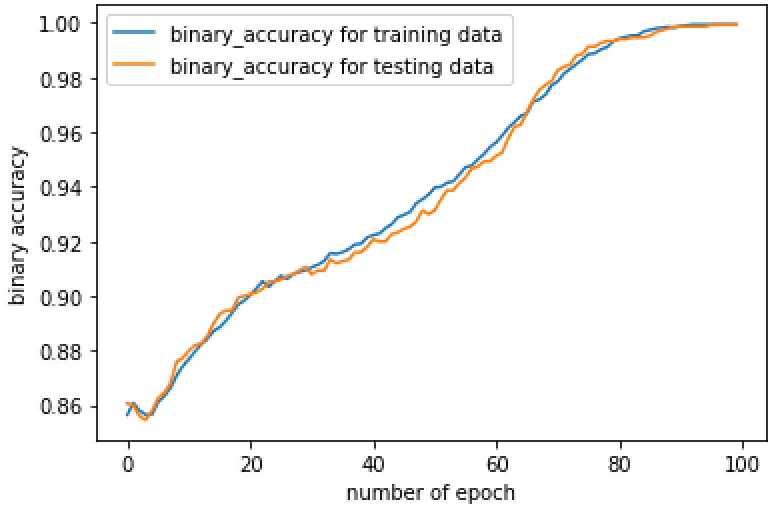
Some resources from the web, which may or may not be relevant:

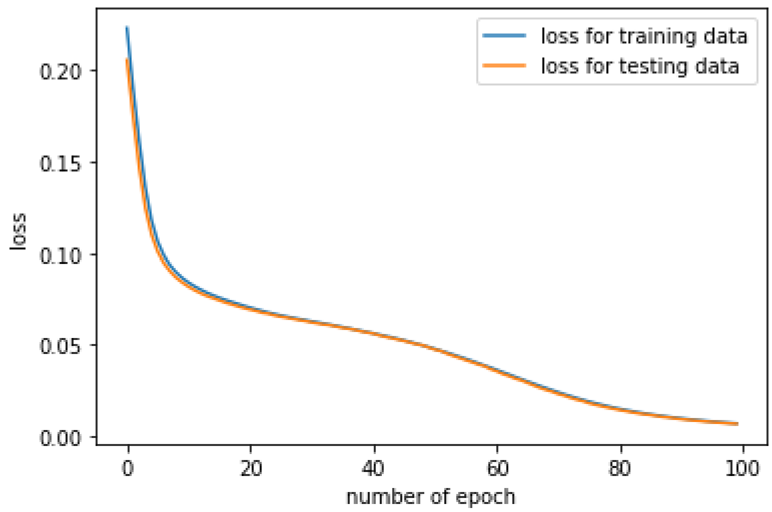
- Measuring performance and basics (https://machinelearningmastery.com/evaluate-performance-deep-learning-models-keras/)
- Weight constraints (different from regularisation) (https://machinelearningmastery.com/how-to-reduce-overfitting-in-deep-neural-networks-with-weight-constraints-in-keras/)
- A nice example (https://heartbeat.fritz.ai/introduction-to-deep-learning-with-keras-c7c3d14e1527)

Note: We are now using Tensorflow 2 which integrates Keras. Therefore, you probably cannot copy paste old scripts from web!

In [2]:
```
## Question 1.1 - part 1.a (different DNN structures) ##

import numpy as np
import datetime
import tensorflow as tf
from keras.layers import Dense
from keras.models import Input,Model
from IPython.display import clear_output
########
import keras
from sklearn import datasets
import numpy as np
from sklearn.model_selection import train_test_split
from matplotlib import pyplot as plt

import pandas as pd
from sklearn.metrics import confusion_matrix

################### plot twice
class PlotLosses(keras.callbacks.Callback):
    def on_train_begin(self, logs={}):
        self.i = 0
        self.x = []
        self.losses = []
        self.val_losses = []

        self.fig = plt.figure()

        self.logs = []

    def on_epoch_end(self, epoch, logs={}):

        self.logs.append(logs)
        self.x.append(self.i)
        self.losses.append(logs.get('binary_accuracy'))
        self.val_losses.append(logs.get('val_binary_accuracy'))
        self.i += 1

        clear_output(wait=True)
        plt.plot(self.x, self.losses, label="binary_accuracy for training data
        plt.plot(self.x, self.val_losses, label="binary_accuracy for testing da
        plt.legend()
        plt.xlabel("number of epoch")
        plt.ylabel("binary accuracy")
        plt.pause(.1)
        plt.cla()

plot_losses = PlotLosses()

################### NN Params
epoch_number=100
batch_size=16
activation_function = 'relu'
optimizer='SGD'
loss_function = 'mse'
#################### create model
def create_model():
```

```
    input_data = Input(shape=(2,))
    D1=Dense(8, activation=activation_function)(input_data)
    D2=Dense(4, activation=activation_function)(D1)
    output=Dense(1, activation='sigmoid')(D2)
    model = Model(inputs=input_data, outputs=output)
    return (model)

#################### Load model and define compiler setting
model=create_model()
model.summary()
model.compile(loss=loss_function, optimizer=optimizer,metrics=[tf.keras.metric
log_dir = "logs/" + datetime.datetime.now().strftime("%Y%m%d-%H%M%S")
tensorboard_callback = tf.keras.callbacks.TensorBoard(log_dir=log_dir, histogr

################### Load moon data and split it
random = np.random.seed(691844)

noisy_moons = datasets.make_moons(n_samples=5000, noise=0.05)
X = noisy_moons[0] # data points
Y = noisy_moons[1] # data points

X_train, X_test, y_train, y_test = train_test_split(X, Y, test_size=0.3,random_

train_hist = model.fit(X_train, y_train, epochs=epoch_number, batch_size=batch_
#train_hist = model.fit(X_train, y_train, epochs=epoch_number, batch_size=batcl

############### joint evalution of  training and testing data
plt.plot(np.array(range(epoch_number)),train_hist.history['loss'],label="loss 
plt.plot(np.array(range(epoch_number)),train_hist.history['val_loss'],label="l
plt.legend()
plt.xlabel("number of epoch")
plt.ylabel("loss")
plt.show()

############################################
print('#####################################')
print('Question 1.1 - part 1.a (different DNN structures)')
print('Two hidden layers --> The DNN structure is : (8, 4, 1)')
print('The activation_function is relu')
print('The optimizer is SGD')
print('The loss function is mse')
print('#####################################')
print()
print(tf.__version__)
print("GPU is", "available" if tf.config.experimental.list_physical_devices('G

################# evalution test Data
print('##############################################################
score = model.evaluate(X_test, y_test, batch_size=batch_size, verbose=2)
print('loss: ',score[0],'accuracy: ',score[1])

ympred = model.predict(X_test)
ympredbinary = (ympred > 0.5)
cm = confusion_matrix(y_test, ympredbinary)
print(pd.DataFrame(cm, columns=["Pred 0", "Pred 1"], index=["True 0", "True 1"
#################
```

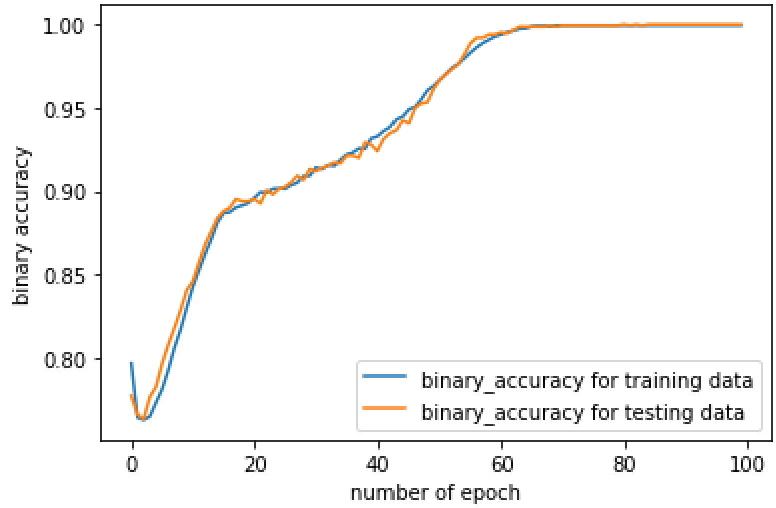


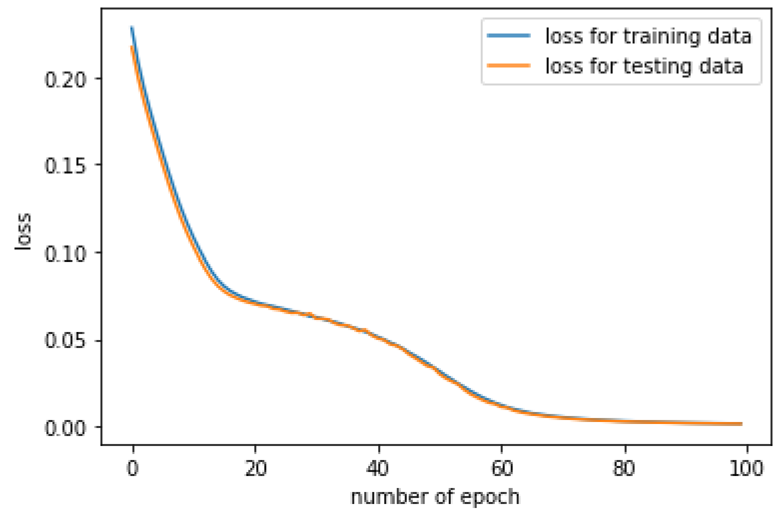



```
#####################################
Question 1.1 - part 1.a (different DNN structures)
Two hidden layers --> The DNN structure is : (8, 4, 1)
The activation_function is relu
The optimizer is SGD
The loss function is mse
#####################################

2.6.0
GPU is NOT AVAILA
##############################################################################
94/94 - 0s - loss: 0.0067 - binary_accuracy: 0.9993
loss:  0.006678552832454443 accuracy:  0.9993333220481873
        Pred 0  Pred 1
True 0     740       1
True 1       0     759
```

In [3]:
```
## Question 1.1 - part 1.a (different DNN structures) ##

import numpy as np
import datetime
import tensorflow as tf
from keras.layers import Dense
from keras.models import Input,Model
from IPython.display import clear_output
########
import keras
from sklearn import datasets
import numpy as np
from sklearn.model_selection import train_test_split
from matplotlib import pyplot as plt

import pandas as pd
from sklearn.metrics import confusion_matrix

################### plot twice
class PlotLosses(keras.callbacks.Callback):
    def on_train_begin(self, logs={}):
        self.i = 0
        self.x = []
        self.losses = []
        self.val_losses = []

        self.fig = plt.figure()

        self.logs = []

    def on_epoch_end(self, epoch, logs={}):

        self.logs.append(logs)
        self.x.append(self.i)
        self.losses.append(logs.get('binary_accuracy'))
        self.val_losses.append(logs.get('val_binary_accuracy'))
        self.i += 1

        clear_output(wait=True)
        plt.plot(self.x, self.losses, label="binary_accuracy for training data
        plt.plot(self.x, self.val_losses, label="binary_accuracy for testing da
        plt.legend()
        plt.xlabel("number of epoch")
        plt.ylabel("binary accuracy")
        plt.pause(.1)
        plt.cla()

plot_losses = PlotLosses()

################### NN Params
epoch_number=100
batch_size=16
activation_function = 'relu'
optimizer='SGD'
loss_function = 'mse'
#################### create model
def create_model():
```

```
    input_data = Input(shape=(2,))
    D1=Dense(8, activation=activation_function)(input_data)
    D2=Dense(4, activation=activation_function)(D1)
    D3=Dense(8, activation=activation_function)(D2)
    output=Dense(1, activation='sigmoid')(D3)
    model = Model(inputs=input_data, outputs=output)
    return (model)

#################### Load model and define compiler setting
model=create_model()
model.summary()
model.compile(loss=loss_function, optimizer=optimizer,metrics=[tf.keras.metric
log_dir = "logs/" + datetime.datetime.now().strftime("%Y%m%d-%H%M%S")
tensorboard_callback = tf.keras.callbacks.TensorBoard(log_dir=log_dir, histogr

################### Load moon data and split it
random = np.random.seed(691844)

noisy_moons = datasets.make_moons(n_samples=5000, noise=0.05)
X = noisy_moons[0] # data points
Y = noisy_moons[1] # data points

X_train, X_test, y_train, y_test = train_test_split(X, Y, test_size=0.3,random_

train_hist = model.fit(X_train, y_train, epochs=epoch_number, batch_size=batch_
#train_hist = model.fit(X_train, y_train, epochs=epoch_number, batch_size=batch

############### joint evalution of  training and testing data
plt.plot(np.array(range(epoch_number)),train_hist.history['loss'],label="loss 
plt.plot(np.array(range(epoch_number)),train_hist.history['val_loss'],label="l
plt.legend()
plt.xlabel("number of epoch")
plt.ylabel("loss")
plt.show()

############################################
print('#####################################')
print('Question 1.1 - part 1.a (different DNN structures)')
print('Three hidden layers --> The DNN structure is : (8, 4, 8, 1)')
print('The activation_function is relu')
print('The optimizer is SGD')
print('The loss function is mse')
print('#####################################')
print()
print(tf.__version__)
print("GPU is", "available" if tf.config.experimental.list_physical_devices('G

################# evalution test Data
print('####################################################################
score = model.evaluate(X_test, y_test, batch_size=batch_size, verbose=2)
print('loss: ',score[0],'accuracy: ',score[1])

ympred = model.predict(X_test)
ympredbinary = (ympred > 0.5)
cm = confusion_matrix(y_test, ympredbinary)
print(pd.DataFrame(cm, columns=["Pred 0", "Pred 1"], index=["True 0", "True 1"
#################
```

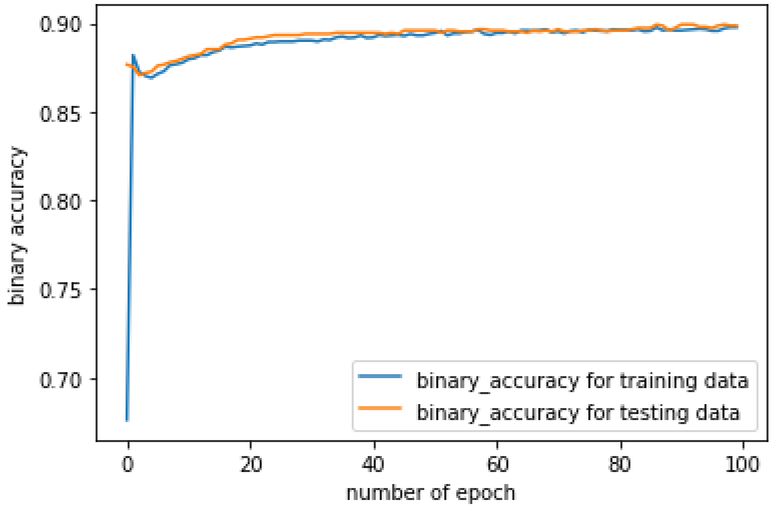


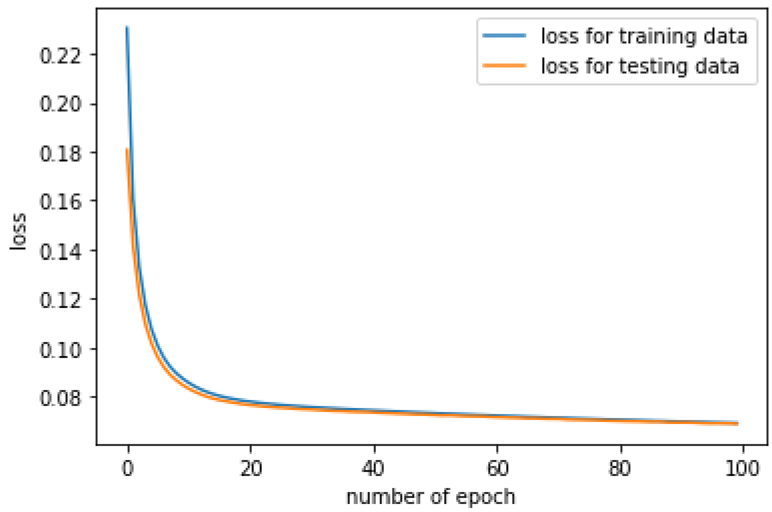



```
#####################################
Question 1.1 - part 1.a (different DNN structures)
Three hidden layers --> The DNN structure is : (8, 4, 8, 1)
The activation_function is relu
The optimizer is SGD
The loss function is mse
#####################################

2.6.0
GPU is NOT AVAILA
#############################################################################
94/94 - 0s - loss: 0.0015 - binary_accuracy: 1.0000
loss:  0.0014584178570657969 accuracy:  1.0
        Pred 0  Pred 1
True 0     741       0
True 1       0     759
```

In [4]:
```
## Question 1.1 - part 1.a (different DNN structures) ##

import numpy as np
import datetime
import tensorflow as tf
from keras.layers import Dense
from keras.models import Input,Model
from IPython.display import clear_output
########
import keras
from sklearn import datasets
import numpy as np
from sklearn.model_selection import train_test_split
from matplotlib import pyplot as plt

import pandas as pd
from sklearn.metrics import confusion_matrix

################### plot twice
class PlotLosses(keras.callbacks.Callback):
    def on_train_begin(self, logs={}):
        self.i = 0
        self.x = []
        self.losses = []
        self.val_losses = []

        self.fig = plt.figure()

        self.logs = []

    def on_epoch_end(self, epoch, logs={}):

        self.logs.append(logs)
        self.x.append(self.i)
        self.losses.append(logs.get('binary_accuracy'))
        self.val_losses.append(logs.get('val_binary_accuracy'))
        self.i += 1

        clear_output(wait=True)
        plt.plot(self.x, self.losses, label="binary_accuracy for training data
        plt.plot(self.x, self.val_losses, label="binary_accuracy for testing da
        plt.legend()
        plt.xlabel("number of epoch")
        plt.ylabel("binary accuracy")
        plt.pause(.1)
        plt.cla()

plot_losses = PlotLosses()

################### NN Params
epoch_number=100
batch_size=16
activation_function = 'relu'
optimizer='SGD'
loss_function = 'mse'
#################### create model
def create_model():
```

```
    input_data = Input(shape=(2,))
    D1=Dense(8, activation=activation_function)(input_data)
    output=Dense(1, activation='sigmoid')(D1)
    model = Model(inputs=input_data, outputs=output)
    return (model)

#################### load model and define compiler setting
model=create_model()
model.summary()
model.compile(loss=loss_function, optimizer=optimizer,metrics=[tf.keras.metric
log_dir = "logs/" + datetime.datetime.now().strftime("%Y%m%d-%H%M%S")
tensorboard_callback = tf.keras.callbacks.TensorBoard(log_dir=log_dir, histogr

################### load moon data and split it
random = np.random.seed(691844)

noisy_moons = datasets.make_moons(n_samples=5000, noise=0.05)
X = noisy_moons[0] # data points
Y = noisy_moons[1] # data points

X_train, X_test, y_train, y_test = train_test_split(X, Y, test_size=0.3,random_

train_hist = model.fit(X_train, y_train, epochs=epoch_number, batch_size=batch_
#train_hist = model.fit(X_train, y_train, epochs=epoch_number, batch_size=batch

############### joint evalution of  training and testing data
plt.plot(np.array(range(epoch_number)),train_hist.history['loss'],label="loss 
plt.plot(np.array(range(epoch_number)),train_hist.history['val_loss'],label="l
plt.legend()
plt.xlabel("number of epoch")
plt.ylabel("loss")
plt.show()

############################################
print('#####################################')
print('Question 1.1 - part 1.a (different DNN structures)')
print('One hidden layer --> The DNN structure is : (8, 1)')
print('The activation_function is relu')
print('The optimizer is SGD')
print('The loss function is mse')
print('#####################################')
print()
print(tf.__version__)
print("GPU is", "available" if tf.config.experimental.list_physical_devices('G

################# evalution test Data
print('##################################################################
score = model.evaluate(X_test, y_test, batch_size=batch_size, verbose=2)
print('loss: ',score[0],'accuracy: ',score[1])

ympred = model.predict(X_test)
ympredbinary = (ympred > 0.5)
cm = confusion_matrix(y_test, ympredbinary)
print(pd.DataFrame(cm, columns=["Pred 0", "Pred 1"], index=["True 0", "True 1"
#################
```



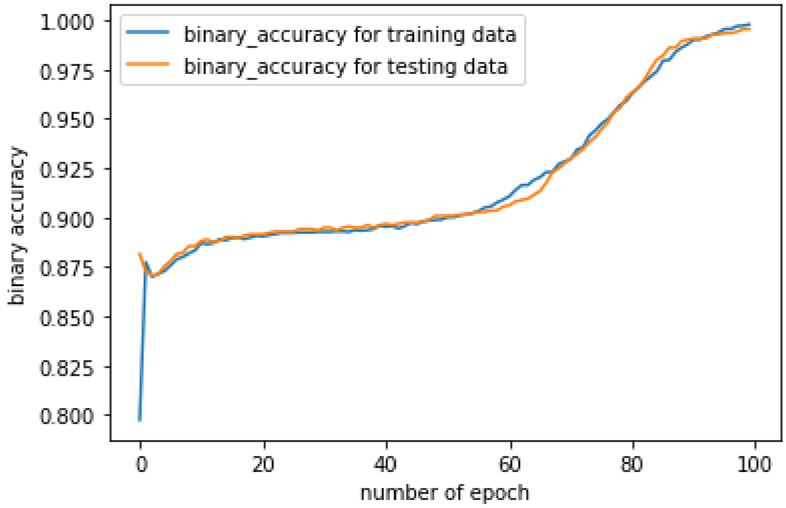


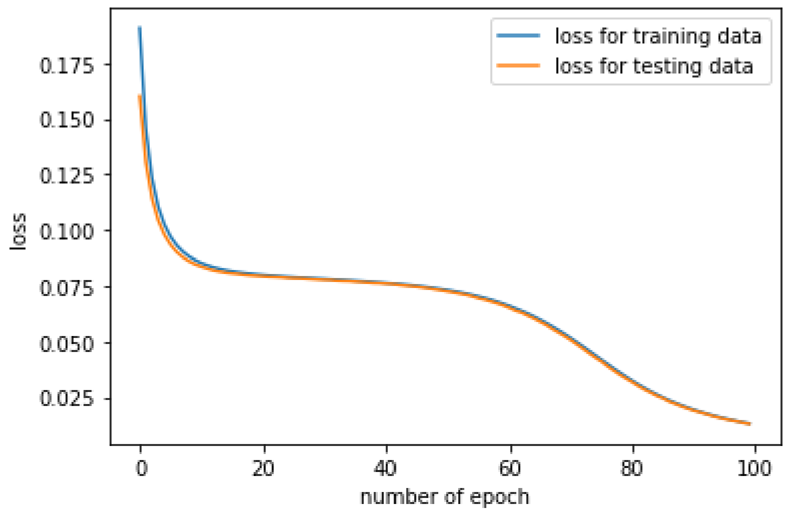

```
#####################################
Question 1.1 - part 1.a (different DNN structures)
One hidden layer --> The DNN structure is : (8, 1)
The activation_function is relu
The optimizer is SGD
The loss function is mse
#####################################

2.6.0
GPU is NOT AVAILA
#############################################################################
94/94 - 0s - loss: 0.0688 - binary_accuracy: 0.8987
loss:  0.06884559988975525 accuracy:  0.8986666798591614
        Pred 0  Pred 1
True 0     650      91
True 1      61     698
```

Having more hidden layers results higher accuracy and lower loss for testing data. Of course, if the number of hidden layers is numerous, over fitting happens and accuracy will decrease and loss increases for testing data.

In [6]:
```
## Question 1.1 - part 1.b (different activation_functions) ##

import numpy as np
import datetime
import tensorflow as tf
from keras.layers import Dense
from keras.models import Input,Model
from IPython.display import clear_output
########
import keras
from sklearn import datasets
import numpy as np
from sklearn.model_selection import train_test_split
from matplotlib import pyplot as plt

import pandas as pd
from sklearn.metrics import confusion_matrix

################### plot twice
class PlotLosses(keras.callbacks.Callback):
    def on_train_begin(self, logs={}):
        self.i = 0
        self.x = []
        self.losses = []
        self.val_losses = []

        self.fig = plt.figure()

        self.logs = []

    def on_epoch_end(self, epoch, logs={}):

        self.logs.append(logs)
        self.x.append(self.i)
        self.losses.append(logs.get('binary_accuracy'))
        self.val_losses.append(logs.get('val_binary_accuracy'))
        self.i += 1

        clear_output(wait=True)
        plt.plot(self.x, self.losses, label="binary_accuracy for training data
        plt.plot(self.x, self.val_losses, label="binary_accuracy for testing da
        plt.legend()
        plt.xlabel("number of epoch")
        plt.ylabel("binary accuracy")
        plt.pause(.1)
        plt.cla()

plot_losses = PlotLosses()

################### NN Params
epoch_number=100
batch_size=16
activation_function = 'tanh'
optimizer='SGD'
loss_function = 'mse'
#################### create model
def create_model():
```

```
    input_data = Input(shape=(2,))
    D1=Dense(8, activation=activation_function)(input_data)
    D2=Dense(4, activation=activation_function)(D1)
    output=Dense(1, activation='sigmoid')(D2)
    model = Model(inputs=input_data, outputs=output)
    return (model)

#################### Load model and define compiler setting
model=create_model()
model.summary()
model.compile(loss=loss_function, optimizer=optimizer,metrics=[tf.keras.metric
log_dir = "logs/" + datetime.datetime.now().strftime("%Y%m%d-%H%M%S")
tensorboard_callback = tf.keras.callbacks.TensorBoard(log_dir=log_dir, histogr

################### Load moon data and split it
random = np.random.seed(691844)

noisy_moons = datasets.make_moons(n_samples=5000, noise=0.05)
X = noisy_moons[0] # data points
Y = noisy_moons[1] # data points

X_train, X_test, y_train, y_test = train_test_split(X, Y, test_size=0.3,random_

train_hist = model.fit(X_train, y_train, epochs=epoch_number, batch_size=batch_
#train_hist = model.fit(X_train, y_train, epochs=epoch_number, batch_size=batcl

############### joint evalution of  training and testing data
plt.plot(np.array(range(epoch_number)),train_hist.history['loss'],label="loss 
plt.plot(np.array(range(epoch_number)),train_hist.history['val_loss'],label="l
plt.legend()
plt.xlabel("number of epoch")
plt.ylabel("loss")
plt.show()

############################################
print('#####################################')
print('Question 1.1 - part 1.b (different activation_functions)')
print('Two hidden layers --> The DNN structure is : (8, 4, 1)')
print('The activation_function is tanh')
print('The optimizer is SGD')
print('The loss function is mse')
print('#####################################')
print()
print(tf.__version__)
print("GPU is", "available" if tf.config.experimental.list_physical_devices('G

################# evalution test Data
print('##################################################################
score = model.evaluate(X_test, y_test, batch_size=batch_size, verbose=2)
print('loss: ',score[0],'accuracy: ',score[1])

ympred = model.predict(X_test)
ympredbinary = (ympred > 0.5)
cm = confusion_matrix(y_test, ympredbinary)
print(pd.DataFrame(cm, columns=["Pred 0", "Pred 1"], index=["True 0", "True 1"
#################
```

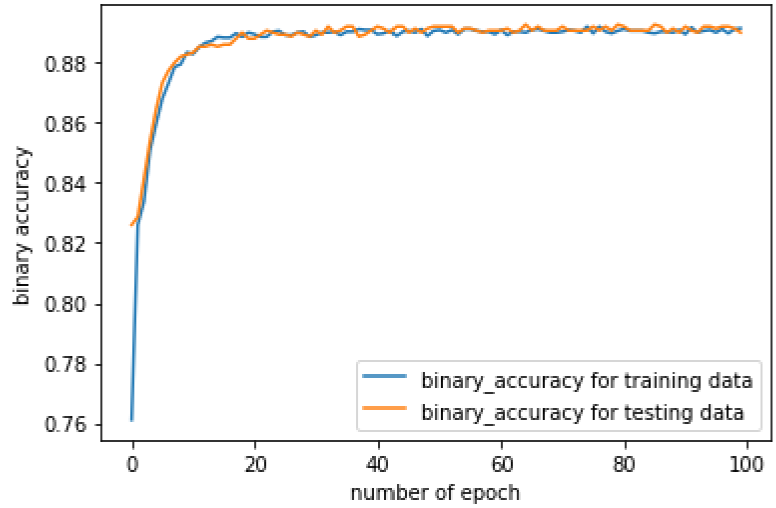


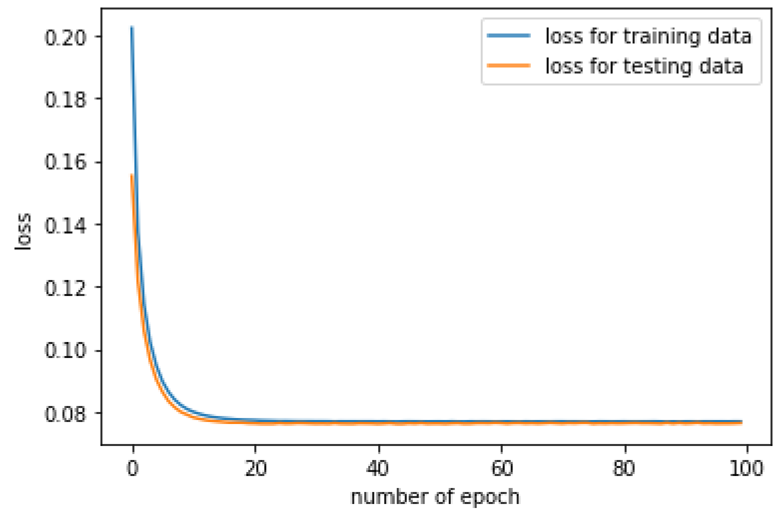



```
#####################################
Question 1.1 - part 1.b (different activation_functions)
Two hidden layers --> The DNN structure is : (8, 4, 1)
The activation_function is tanh
The optimizer is SGD
The loss function is mse
#####################################

2.6.0
GPU is NOT AVAILA
#############################################################################
94/94 - 0s - loss: 0.0131 - binary_accuracy: 0.9953
loss:  0.013080407865345478 accuracy:  0.9953333139419556
        Pred 0  Pred 1
True 0     737       4
True 1       3     756
```

In [7]:
```
## Question 1.1 - part 1.b (different activation_functions) ##

import numpy as np
import datetime
import tensorflow as tf
from keras.layers import Dense
from keras.models import Input,Model
from IPython.display import clear_output
########
import keras
from sklearn import datasets
import numpy as np
from sklearn.model_selection import train_test_split
from matplotlib import pyplot as plt

import pandas as pd
from sklearn.metrics import confusion_matrix

################### plot twice
class PlotLosses(keras.callbacks.Callback):
    def on_train_begin(self, logs={}):
        self.i = 0
        self.x = []
        self.losses = []
        self.val_losses = []

        self.fig = plt.figure()

        self.logs = []

    def on_epoch_end(self, epoch, logs={}):

        self.logs.append(logs)
        self.x.append(self.i)
        self.losses.append(logs.get('binary_accuracy'))
        self.val_losses.append(logs.get('val_binary_accuracy'))
        self.i += 1

        clear_output(wait=True)
        plt.plot(self.x, self.losses, label="binary_accuracy for training data
        plt.plot(self.x, self.val_losses, label="binary_accuracy for testing da
        plt.legend()
        plt.xlabel("number of epoch")
        plt.ylabel("binary accuracy")
        plt.pause(.1)
        plt.cla()

plot_losses = PlotLosses()

################### NN Params
epoch_number=100
batch_size=16
activation_function = 'linear'
optimizer='SGD'
loss_function = 'mse'
#################### create model
def create_model():
```

```
    input_data = Input(shape=(2,))
    D1=Dense(8, activation=activation_function)(input_data)
    D2=Dense(4, activation=activation_function)(D1)
    output=Dense(1, activation='sigmoid')(D2)
    model = Model(inputs=input_data, outputs=output)
    return (model)

#################### Load model and define compiler setting
model=create_model()
model.summary()
model.compile(loss=loss_function, optimizer=optimizer,metrics=[tf.keras.metric
log_dir = "logs/" + datetime.datetime.now().strftime("%Y%m%d-%H%M%S")
tensorboard_callback = tf.keras.callbacks.TensorBoard(log_dir=log_dir, histogr

################### Load moon data and split it
random = np.random.seed(691844)

noisy_moons = datasets.make_moons(n_samples=5000, noise=0.05)
X = noisy_moons[0] # data points
Y = noisy_moons[1] # data points

X_train, X_test, y_train, y_test = train_test_split(X, Y, test_size=0.3,random_

train_hist = model.fit(X_train, y_train, epochs=epoch_number, batch_size=batch_
#train_hist = model.fit(X_train, y_train, epochs=epoch_number, batch_size=batcl

############### joint evalution of  training and testing data
plt.plot(np.array(range(epoch_number)),train_hist.history['loss'],label="loss 
plt.plot(np.array(range(epoch_number)),train_hist.history['val_loss'],label="l
plt.legend()
plt.xlabel("number of epoch")
plt.ylabel("loss")
plt.show()

############################################
print('#####################################')
print('Question 1.1 - part 1.b (different activation_functions)')
print('Two hidden layers --> The DNN structure is : (8, 4, 1)')
print('The activation_function is linear')
print('The optimizer is SGD')
print('The loss function is mse')
print('#####################################')
print()
print(tf.__version__)
print("GPU is", "available" if tf.config.experimental.list_physical_devices('G

################# evalution test Data
print('#################################################################
score = model.evaluate(X_test, y_test, batch_size=batch_size, verbose=2)
print('loss: ',score[0],'accuracy: ',score[1])

ympred = model.predict(X_test)
ympredbinary = (ympred > 0.5)
cm = confusion_matrix(y_test, ympredbinary)
print(pd.DataFrame(cm, columns=["Pred 0", "Pred 1"], index=["True 0", "True 1"
#################
```



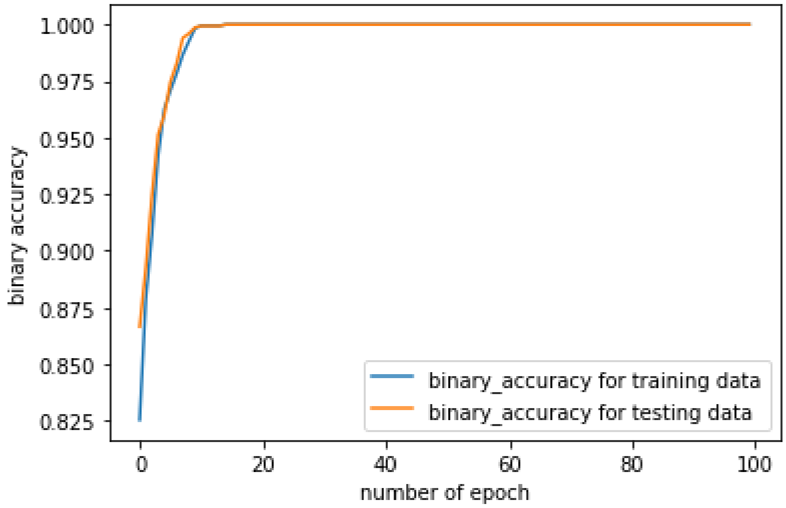


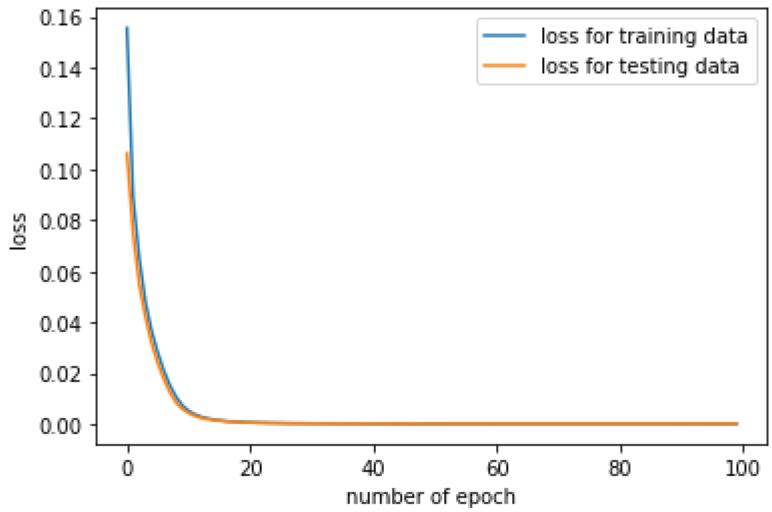

```
#####################################
Question 1.1 - part 1.b (different activation_functions)
Two hidden layers --> The DNN structure is : (8, 4, 1)
The activation_function is linear
The optimizer is SGD
The loss function is mse
#####################################

2.6.0
GPU is NOT AVAILA
###############################################################################
94/94 - 0s - loss: 0.0764 - binary_accuracy: 0.8900
loss:  0.07637093216180801 accuracy:  0.8899999856948853
        Pred 0  Pred 1
True 0     647      94
True 1      71     688
```

When the activation function is Relu, accuracy is higher and loss is lower for testing data which shows Relu as the activation function works better than Tanh and Linear functions.

In [9]:
```
## Question 1.1 - part 1.c (different optimizers) ##

import numpy as np
import datetime
import tensorflow as tf
from keras.layers import Dense
from keras.models import Input,Model
from IPython.display import clear_output
########
import keras
from sklearn import datasets
import numpy as np
from sklearn.model_selection import train_test_split
from matplotlib import pyplot as plt

import pandas as pd
from sklearn.metrics import confusion_matrix

################### plot twice
class PlotLosses(keras.callbacks.Callback):
    def on_train_begin(self, logs={}):
        self.i = 0
        self.x = []
        self.losses = []
        self.val_losses = []

        self.fig = plt.figure()

        self.logs = []

    def on_epoch_end(self, epoch, logs={}):

        self.logs.append(logs)
        self.x.append(self.i)
        self.losses.append(logs.get('binary_accuracy'))
        self.val_losses.append(logs.get('val_binary_accuracy'))
        self.i += 1

        clear_output(wait=True)
        plt.plot(self.x, self.losses, label="binary_accuracy for training data
        plt.plot(self.x, self.val_losses, label="binary_accuracy for testing da
        plt.legend()
        plt.xlabel("number of epoch")
        plt.ylabel("binary accuracy")
        plt.pause(.1)
        plt.cla()

plot_losses = PlotLosses()

################### NN Params
epoch_number=100
batch_size=16
activation_function = 'relu'
optimizer='adam'
loss_function = 'mse'
#################### create model
def create_model():
```

```
    input_data = Input(shape=(2,))
    D1=Dense(8, activation=activation_function)(input_data)
    D2=Dense(4, activation=activation_function)(D1)
    output=Dense(1, activation='sigmoid')(D2)
    model = Model(inputs=input_data, outputs=output)
    return (model)

#################### Load model and define compiler setting
model=create_model()
model.summary()
model.compile(loss=loss_function, optimizer=optimizer,metrics=[tf.keras.metric
log_dir = "logs/" + datetime.datetime.now().strftime("%Y%m%d-%H%M%S")
tensorboard_callback = tf.keras.callbacks.TensorBoard(log_dir=log_dir, histogr

################### Load moon data and split it
random = np.random.seed(691844)

noisy_moons = datasets.make_moons(n_samples=5000, noise=0.05)
X = noisy_moons[0] # data points
Y = noisy_moons[1] # data points

X_train, X_test, y_train, y_test = train_test_split(X, Y, test_size=0.3,random_

train_hist = model.fit(X_train, y_train, epochs=epoch_number, batch_size=batch_
#train_hist = model.fit(X_train, y_train, epochs=epoch_number, batch_size=batcl

############### joint evalution of  training and testing data
plt.plot(np.array(range(epoch_number)),train_hist.history['loss'],label="loss 
plt.plot(np.array(range(epoch_number)),train_hist.history['val_loss'],label="l
plt.legend()
plt.xlabel("number of epoch")
plt.ylabel("loss")
plt.show()

############################################
print('#####################################')
print('Question 1.1 - part 1.c (different optimizers)')
print('Two hidden layers --> The DNN structure is : (8, 4, 1)')
print('The activation_function is relu')
print('The optimizer is adam')
print('The loss function is mse')
print('#####################################')
print()
print(tf.__version__)
print("GPU is", "available" if tf.config.experimental.list_physical_devices('G

################# evalution test Data
print('#################################################################
score = model.evaluate(X_test, y_test, batch_size=batch_size, verbose=2)
print('loss: ',score[0],'accuracy: ',score[1])

ympred = model.predict(X_test)
ympredbinary = (ympred > 0.5)
cm = confusion_matrix(y_test, ympredbinary)
print(pd.DataFrame(cm, columns=["Pred 0", "Pred 1"], index=["True 0", "True 1"
#################
```

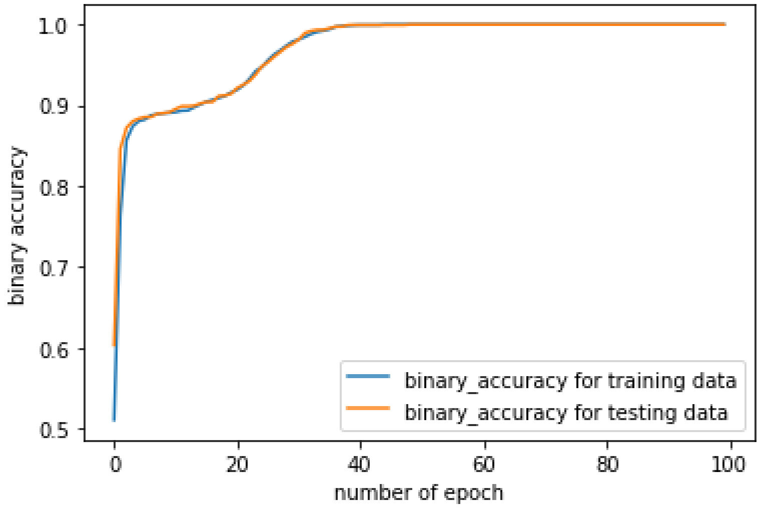


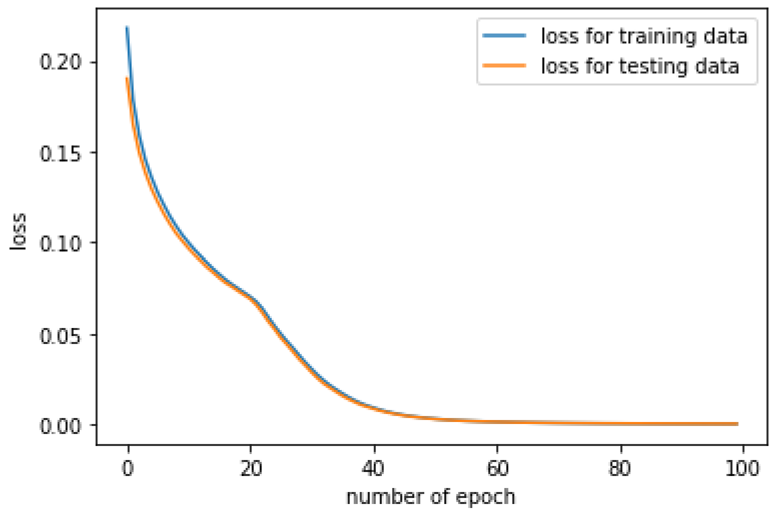



```
#####################################
Question 1.1 - part 1.c (different optimizers)
Two hidden layers --> The DNN structure is : (8, 4, 1)
The activation_function is relu
The optimizer is adam
The loss function is mse
#####################################

2.6.0
GPU is NOT AVAILA
##############################################################################
94/94 - 0s - loss: 6.4233e-07 - binary_accuracy: 1.0000
loss:  6.423347258532885e-07 accuracy:  1.0
        Pred 0  Pred 1
True 0     741       0
True 1       0     759
```

In [10]:
```
## Question 1.1 - part 1.c (different optimizers) ##

import numpy as np
import datetime
import tensorflow as tf
from keras.layers import Dense
from keras.models import Input,Model
from IPython.display import clear_output
########
import keras
from sklearn import datasets
import numpy as np
from sklearn.model_selection import train_test_split
from matplotlib import pyplot as plt

import pandas as pd
from sklearn.metrics import confusion_matrix

################### plot twice
class PlotLosses(keras.callbacks.Callback):
    def on_train_begin(self, logs={}):
        self.i = 0
        self.x = []
        self.losses = []
        self.val_losses = []

        self.fig = plt.figure()

        self.logs = []

    def on_epoch_end(self, epoch, logs={}):

        self.logs.append(logs)
        self.x.append(self.i)
        self.losses.append(logs.get('binary_accuracy'))
        self.val_losses.append(logs.get('val_binary_accuracy'))
        self.i += 1

        clear_output(wait=True)
        plt.plot(self.x, self.losses, label="binary_accuracy for training data
        plt.plot(self.x, self.val_losses, label="binary_accuracy for testing da
        plt.legend()
        plt.xlabel("number of epoch")
        plt.ylabel("binary accuracy")
        plt.pause(.1)
        plt.cla()

plot_losses = PlotLosses()

################### NN Params
epoch_number=100
batch_size=16
activation_function = 'relu'
optimizer='adamax'
loss_function = 'mse'
#################### create model
def create_model():
```

```
    input_data = Input(shape=(2,))
    D1=Dense(8, activation=activation_function)(input_data)
    D2=Dense(4, activation=activation_function)(D1)
    output=Dense(1, activation='sigmoid')(D2)
    model = Model(inputs=input_data, outputs=output)
    return (model)

#################### Load model and define compiler setting
model=create_model()
model.summary()
model.compile(loss=loss_function, optimizer=optimizer,metrics=[tf.keras.metric
log_dir = "logs/" + datetime.datetime.now().strftime("%Y%m%d-%H%M%S")
tensorboard_callback = tf.keras.callbacks.TensorBoard(log_dir=log_dir, histogr

################### Load moon data and split it
random = np.random.seed(691844)

noisy_moons = datasets.make_moons(n_samples=5000, noise=0.05)
X = noisy_moons[0] # data points
Y = noisy_moons[1] # data points

X_train, X_test, y_train, y_test = train_test_split(X, Y, test_size=0.3,random_

train_hist = model.fit(X_train, y_train, epochs=epoch_number, batch_size=batch_
#train_hist = model.fit(X_train, y_train, epochs=epoch_number, batch_size=batch

############### joint evalution of  training and testing data
plt.plot(np.array(range(epoch_number)),train_hist.history['loss'],label="loss 
plt.plot(np.array(range(epoch_number)),train_hist.history['val_loss'],label="l
plt.legend()
plt.xlabel("number of epoch")
plt.ylabel("loss")
plt.show()

############################################
print('#####################################')
print('Question 1.1 - part 1.c (different optimizers)')
print('Two hidden layers --> The DNN structure is : (8, 4, 1)')
print('The activation_function is relu')
print('The optimizer is adamax')
print('The loss function is mse')
print('#####################################')
print()
print(tf.__version__)
print("GPU is", "available" if tf.config.experimental.list_physical_devices('G

################# evalution test Data
print('#####################################################################
score = model.evaluate(X_test, y_test, batch_size=batch_size, verbose=2)
print('loss: ',score[0],'accuracy: ',score[1])

ympred = model.predict(X_test)
ympredbinary = (ympred > 0.5)
cm = confusion_matrix(y_test, ympredbinary)
print(pd.DataFrame(cm, columns=["Pred 0", "Pred 1"], index=["True 0", "True 1"
#################
```



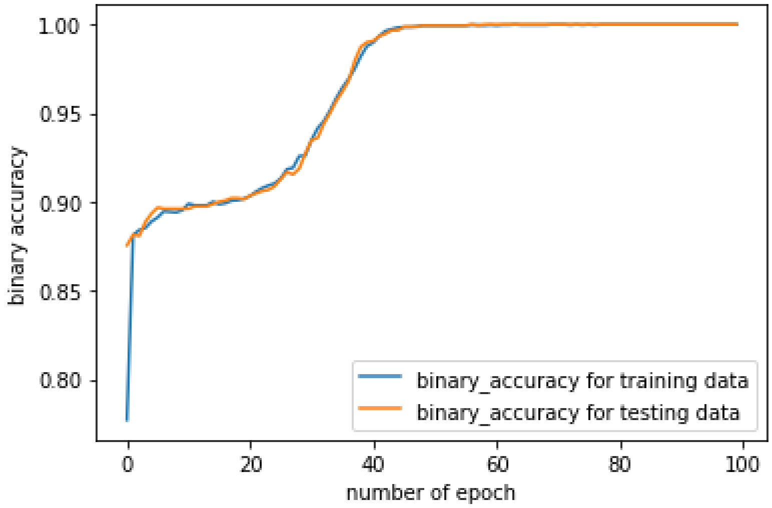


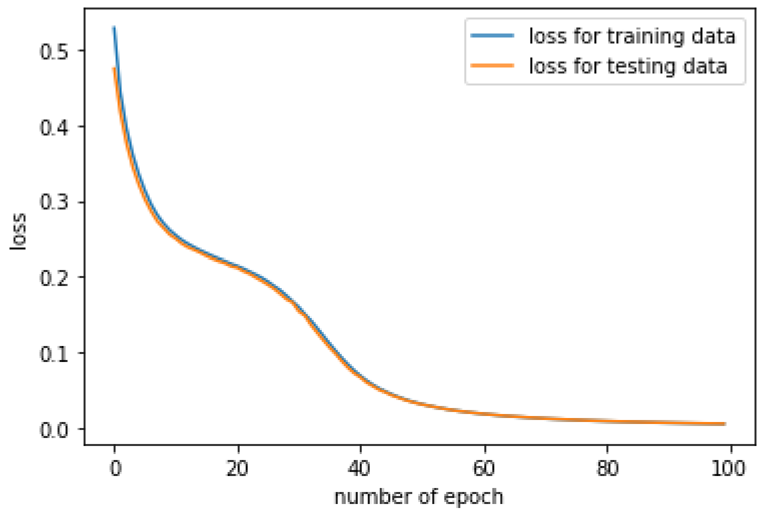

```
#####################################
Question 1.1 - part 1.c (different optimizers)
Two hidden layers --> The DNN structure is : (8, 4, 1)
The activation_function is relu
The optimizer is adamax
The loss function is mse
#####################################

2.6.0
GPU is NOT AVAILA
##############################################################################
94/94 - 0s - loss: 1.4913e-04 - binary_accuracy: 1.0000
loss:  0.00014912715414538980 accuracy:  1.0
        Pred 0  Pred 1
True 0     741       0
True 1       0     759
```

When the optimizer is Adam, accuracy is higher and loss is lower for testing data which shows Adam as the optimizer works better than SGD and Adamax. (Accuracy reaches 1 earlier when the optimizer is Adam).

In [11]:
```
## Question 1.1 - part 1.d (different loss_functions) ##

import numpy as np
import datetime
import tensorflow as tf
from keras.layers import Dense
from keras.models import Input,Model
from IPython.display import clear_output
########
import keras
from sklearn import datasets
import numpy as np
from sklearn.model_selection import train_test_split
from matplotlib import pyplot as plt

import pandas as pd
from sklearn.metrics import confusion_matrix

################### plot twice
class PlotLosses(keras.callbacks.Callback):
    def on_train_begin(self, logs={}):
        self.i = 0
        self.x = []
        self.losses = []
        self.val_losses = []

        self.fig = plt.figure()

        self.logs = []

    def on_epoch_end(self, epoch, logs={}):

        self.logs.append(logs)
        self.x.append(self.i)
        self.losses.append(logs.get('binary_accuracy'))
        self.val_losses.append(logs.get('val_binary_accuracy'))
        self.i += 1

        clear_output(wait=True)
        plt.plot(self.x, self.losses, label="binary_accuracy for training data
        plt.plot(self.x, self.val_losses, label="binary_accuracy for testing da
        plt.legend()
        plt.xlabel("number of epoch")
        plt.ylabel("binary accuracy")
        plt.pause(.1)
        plt.cla()

plot_losses = PlotLosses()

################### NN Params
epoch_number=100
batch_size=16
activation_function = 'relu'
optimizer='SGD'
loss_function = 'binary_crossentropy'
#################### create model
def create_model():
```

```
    input_data = Input(shape=(2,))
    D1=Dense(8, activation=activation_function)(input_data)
    D2=Dense(4, activation=activation_function)(D1)
    output=Dense(1, activation='sigmoid')(D2)
    model = Model(inputs=input_data, outputs=output)
    return (model)

#################### Load model and define compiler setting
model=create_model()
model.summary()
model.compile(loss=loss_function, optimizer=optimizer,metrics=[tf.keras.metric
log_dir = "logs/" + datetime.datetime.now().strftime("%Y%m%d-%H%M%S")
tensorboard_callback = tf.keras.callbacks.TensorBoard(log_dir=log_dir, histogra

################### Load moon data and split it
random = np.random.seed(691844)

noisy_moons = datasets.make_moons(n_samples=5000, noise=0.05)
X = noisy_moons[0] # data points
Y = noisy_moons[1] # data points

X_train, X_test, y_train, y_test = train_test_split(X, Y, test_size=0.3,random_

train_hist = model.fit(X_train, y_train, epochs=epoch_number, batch_size=batch_
#train_hist = model.fit(X_train, y_train, epochs=epoch_number, batch_size=batch

############### joint evalution of  training and testing data
plt.plot(np.array(range(epoch_number)),train_hist.history['loss'],label="loss 
plt.plot(np.array(range(epoch_number)),train_hist.history['val_loss'],label="l
plt.legend()
plt.xlabel("number of epoch")
plt.ylabel("loss")
plt.show()

############################################
print('#####################################')
print('Question 1.1 - part 1.d (different loss_functions)')
print('Two hidden layers --> The DNN structure is : (8, 4, 1)')
print('The activation_function is relu')
print('The optimizer is SGD')
print('The loss function is binary_crossentropy')
print('#####################################')
print()
print(tf.__version__)
print("GPU is", "available" if tf.config.experimental.list_physical_devices('G

################# evalution test Data
print('##############################################################
score = model.evaluate(X_test, y_test, batch_size=batch_size, verbose=2)
print('loss: ',score[0],'accuracy: ',score[1])

ympred = model.predict(X_test)
ympredbinary = (ympred > 0.5)
cm = confusion_matrix(y_test, ympredbinary)
print(pd.DataFrame(cm, columns=["Pred 0", "Pred 1"], index=["True 0", "True 1"
#################
```






```
#####################################
Question 1.1 - part 1.d (different loss_functions)
Two hidden layers --> The DNN structure is : (8, 4, 1)
The activation_function is relu
The optimizer is SGD
The loss function is binary_crossentropy
#####################################

2.6.0
GPU is NOT AVAILA
#############################################################################
94/94 - 0s - loss: 0.0064 - binary_accuracy: 1.0000
loss:  0.006412905175238848 accuracy:  1.0
        Pred 0  Pred 1
True 0     741       0
True 1       0     759
```

When the loss function is Cross Entropy, accuracy is higher and loss is lower for testing data which shows Cross Entropy as the loss function works better than MSE. (As we know, Cross Entropy is better for the binary classification problem, and MSE (Mean Sqaured Error) is better for the regression problem.)

In [118]:
```
# split into training and test sets
Xmtrain, Xmtest, ymtrain, ymtest = train_test_split(Xm, ym,test_size=0.3, rand
```

In [196]:
```
# Define the DNN sequential model

model = Sequential()
model.add(tf.keras.layers.InputLayer(input_shape=(2,)))
model.add(Dense(8, activation='relu'))
# model.add(Dense(6, activation='tanh'))
model.add(Dense(4, activation='relu'))
# model.add(Dense(4, activation='relu'))
model.add(Dense(2, activation='relu'))
model.add(Dense(1, activation='sigmoid'))


model.summary()
#Configures the model for training
model.compile(
    optimizer=tf.keras.optimizers.Adam(learning_rate=0.01,epsilon=0.08),
              loss=tf.keras.losses.BinaryCrossentropy(),
              metrics=[tf.keras.metrics.BinaryAccuracy(),tf.keras.metrics.AUC(
)

# log results
log_dir = "logs/" + datetime.datetime.now().strftime("%Y%m%d-%H%M%S")
tensorboard_callback = tf.keras.callbacks.TensorBoard(log_dir=log_dir, histogr
```

```
Model: "sequential_11"
_________________________________________________________________
Layer (type)                Output Shape              Param #
=================================================================
dense_44 (Dense)            (None, 8)                 24

dense_45 (Dense)            (None, 4)                 36

dense_46 (Dense)            (None, 2)                 10

dense_47 (Dense)            (None, 1)                 3

=================================================================
Total params: 73
Trainable params: 73
Non-trainable params: 0
_________________________________________________________________
```

In [197]:
```
# Train the model, iterating on the data in batches, record history
train_hist = model.fit(Xmtrain, ymtrain, epochs=100, batch_size=16, verbose=1,
```

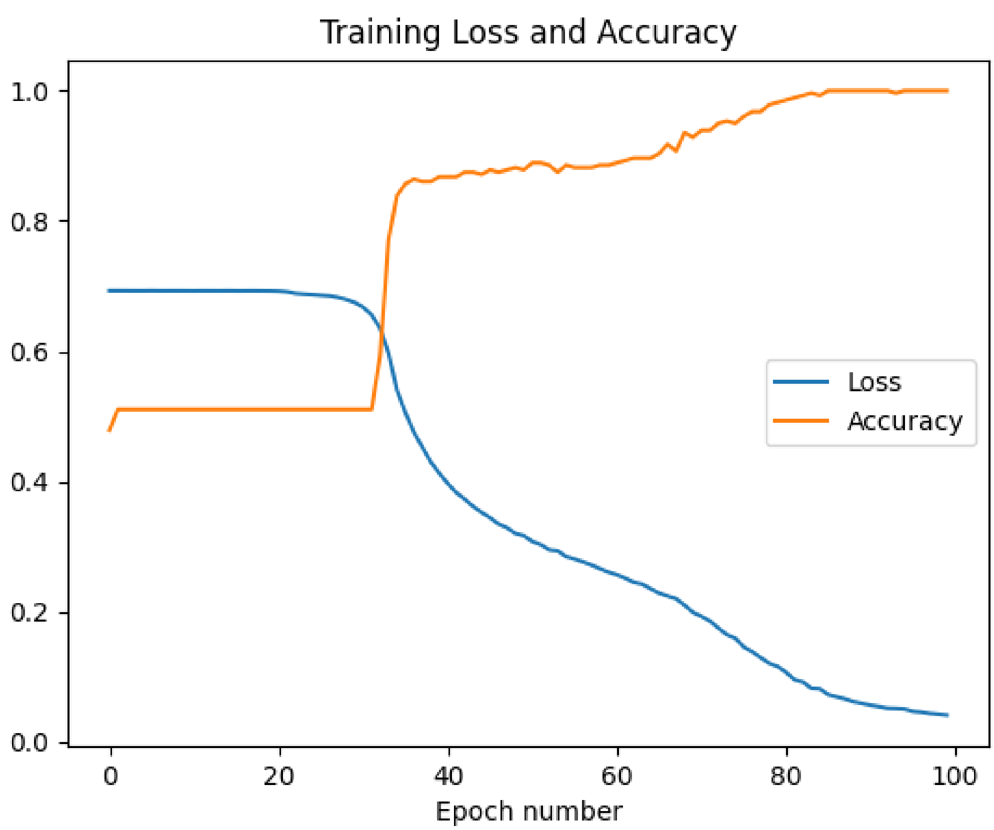
```
Epoch 1/100
18/18 [==============================] - 2s 50ms/step - loss: 0.6931 - bina
ry_accuracy: 0.4786 - auc_8: 0.5035
Epoch 2/100
18/18 [==============================] - 0s 2ms/step - loss: 0.6931 - binar
y_accuracy: 0.5107 - auc_8: 0.5035
Epoch 3/100
18/18 [==============================] - 0s 2ms/step - loss: 0.6931 - binar
y_accuracy: 0.5107 - auc_8: 0.5035
Epoch 4/100
18/18 [==============================] - 0s 1ms/step - loss: 0.6930 - binar
y_accuracy: 0.5107 - auc_8: 0.5239
Epoch 5/100
18/18 [==============================] - 0s 2ms/step - loss: 0.6930 - binar
y_accuracy: 0.5107 - auc_8: 0.5035
Epoch 6/100
18/18 [==============================] - 0s 2ms/step - loss: 0.6931 - binar
y_accuracy: 0.5107 - auc_8: 0.5035
Epoch 7/100
```

In [198]:
```
print(model.summary())
weights = model.get_weights() # Getting params
print(weights)
```

```
Model: "sequential_11"
_________________________________________________________________
Layer (type)                Output Shape              Param #
=================================================================
dense_44 (Dense)            (None, 8)                 24

_________________________________________________________________
dense_45 (Dense)            (None, 4)                 36

_________________________________________________________________
dense_46 (Dense)            (None, 2)                 10

_________________________________________________________________
dense_47 (Dense)            (None, 1)                 3
=================================================================
Total params: 73
Trainable params: 73
Non-trainable params: 0

_________________________________________________________________
None
[array([[ 0.6899282 ,  1.188518  , -0.80559087,  1.0036772 ,  0.39396805,
        -1.1700078 ,  0.94696236,  0.18993352],
       [ 0.632597  , -0.11784611,  0.6571538 ,  0.71379715, -0.18144855,
         0.5911901 , -0.05977149, -1.0692799 ]], dtype=float32), array([-0.15
615244, -1.2197145 , -0.27085567, -0.14683333, -0.12912919,
       -0.29377073, -0.9864979 , -0.14429046], dtype=float32), array([[ 0.577
6936 , -0.74158835, -0.30027798, -0.1425357 ],
       [-0.02606203, -0.20593892, -0.40616226,  1.6089604 ],
       [ 0.7365362 , -0.42395478,  0.13067326, -0.5991773 ],
       [ 0.6651437 , -0.21913566, -0.2046439 , -0.8246826 ],
       [ 0.36217448, -0.39066187,  0.5772232 ,  0.18751083],
       [ 0.8080516 , -0.31036738,  0.31568852, -0.7889255 ],
       [-0.2626635 , -0.4859837 , -0.11726785,  1.111831  ],
       [-0.53405374,  0.46077216, -0.33686656,  1.0700886 ]],
      dtype=float32), array([ 0.17209835,  0.6086832 , -0.00382426,  0.736749
5 ], dtype=float32), array([[-0.9579544 , -0.68264836],
       [-0.2673924 ,  1.0157435 ],
       [ 0.59546447, -0.37657964],
       [-0.1200521 ,  2.4284642 ]], dtype=float32), array([0.        , 0.5893
7585], dtype=float32), array([[0.05230725],
       [2.734109  ]], dtype=float32), array([-2.88011], dtype=float32)]
```

In [199]:
```
# Test data Scores
score = model.evaluate(Xmtest, ymtest, batch_size=16, verbose=2)
print("Test Data Accuracy Details")
print(score)

#train_hist.history

plt.figure()
plt.plot(train_hist.history['loss'])
plt.plot(train_hist.history['binary_accuracy'])
plt.xlabel('Epoch number')
plt.title('Training Loss and Accuracy')
plt.legend(['Loss', 'Accuracy'], loc='center right')
plt.show()
```

```
8/8 - 1s - loss: 0.0733 - binary_accuracy: 0.9917 - auc_8: 0.9981
Test Data Accuracy Details
[0.0733213797211647, 0.9916666746139526, 0.9980506896972656]

<IPython.core.display.Javascript object>
```



In [200]:
```
import sklearn.metrics as metrics

ympred = model.predict(Xmtest)
ympredbinary = (ympred > 0.5)

cm = metrics.confusion_matrix(ymtest, ympredbinary)
##Print Confusion Matrix
print("TEST DATA CONFUSION MATRIX")
print(pd.DataFrame(cm, columns=["Pred 0", "Pred 1"], index=["True 0", "True 1"
##Print Classification Report
print("\nTEST DATA CLASSIFICATION REPORT")
print(metrics.classification_report(ymtest, ympredbinary, target_names=["True (
##Print AUC
#Finding the TP and FP of the predicted test data
fpr, tpr, thresholds = metrics.roc_curve(ymtest, ympredbinary)
print("AUC = %0.2f"% metrics.auc(fpr, tpr))
```

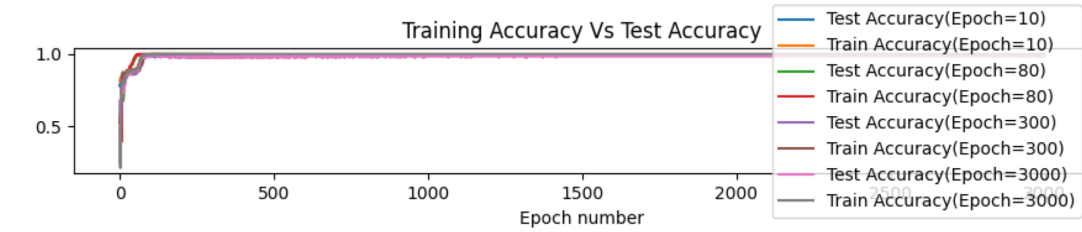
```
TEST DATA CONFUSION MATRIX
        Pred 0  Pred 1
True 0      63       0
True 1       1      56

TEST DATA CLASSIFICATION REPORT
              precision    recall  f1-score   support

      True 0       0.98      1.00      0.99        63
      True 1       1.00      0.98      0.99        57

    accuracy                           0.99       120
   macro avg       0.99      0.99      0.99       120
weighted avg       0.99      0.99      0.99       120

AUC = 0.99
```

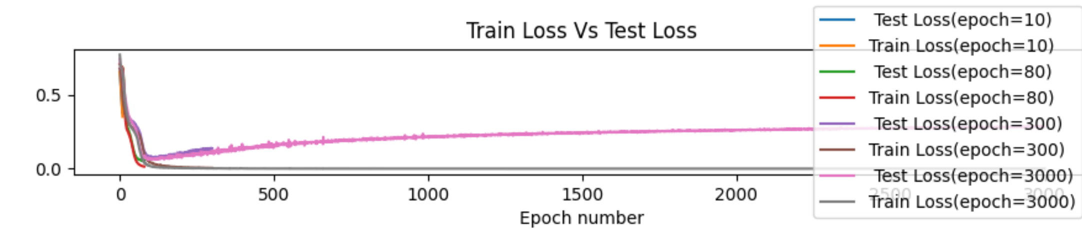
- **It is observed that when same model is trained again the resulting parameters are different and this result in different output and accuracy**
- **When the number of hidden layers are increased it does not necessarilly increase accuracy.**
- **Using different activation functions like "tanh, RELU, ELU" yeilded different results**
- **Different optimizers and loss functions were testd where each result were different. Best suited was Adam optimizer**
- **Therefore in order to improve the model we need to configure the model differently and identify the best combination suited for the given problem**

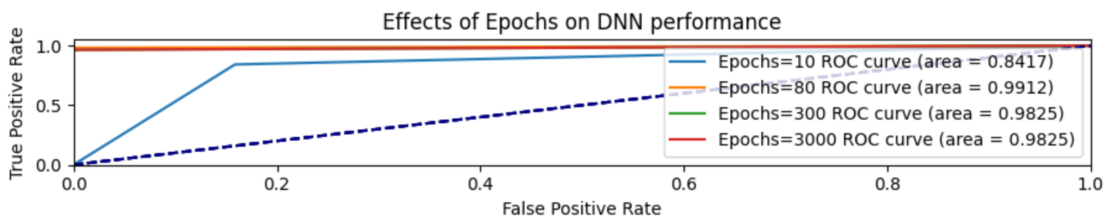
In [210]:
```
#####Checking effects of training epoch
#for make model reproducable
np.random.seed(1320418)
tf.random.set_seed(1320418)

def create_model():
    # Define the DNN sequential model
    model = Sequential()
    model.add(tf.keras.layers.InputLayer(input_shape=(2,)))
    model.add(Dense(8, activation='relu'))
    # model.add(Dense(6, activation='tanh'))
    model.add(Dense(4, activation='relu'))
    # model.add(Dense(4, activation='relu'))
    model.add(Dense(2, activation='relu'))
    model.add(Dense(1, activation='sigmoid'))

#     model.summary()
    #Configures the model for training
    model.compile(
        optimizer=tf.keras.optimizers.Adam(learning_rate=0.01,epsilon=0.08),
                  loss=tf.keras.losses.BinaryCrossentropy(),
                  metrics=[tf.keras.metrics.BinaryAccuracy(),tf.keras.metrics.
    )

    # log results
    log_dir = "logs/" + datetime.datetime.now().strftime("%Y%m%d-%H%M%S")
    tensorboard_callback = tf.keras.callbacks.TensorBoard(log_dir=log_dir, his
    return model

##Checking effects of training epoch
plt.figure()
epochs=[10,80,300,3000]
for epoc in epochs:
    tf.keras.backend.clear_session()
    tf.compat.v1.reset_default_graph()
    model=create_model()
    # Train the model, iterating on the data in batches, record history
    hist_new = model.fit(Xmtrain, ymtrain,validation_data=(Xmtest, ymtest), ep
#     print(train_hist_new.history.keys())
    #Plot train_hist.history
    plt.subplot(311)

    plt.plot(hist_new.history['val_binary_accuracy'],label=f"Test Accuracy(Epo
    plt.plot(hist_new.history['binary_accuracy'],label=f"Train Accuracy(Epoch=
    plt.legend(loc='center right')
    plt.xlabel('Epoch number')
    plt.title('Training Accuracy Vs Test Accuracy')
    plt.grid()

    #Plot test_hist.history
    plt.subplot(312)
    plt.plot(hist_new.history['val_loss'],label=f" Test Loss(epoch={epoc})")
    plt.plot(hist_new.history['loss'],label=f"Train Loss(epoch={epoc})")
    plt.legend(loc='center right')
    plt.xlabel('Epoch number')
    plt.title('Train Loss Vs Test Loss')
    plt.grid()
```

```
    # Train data Scores
    score_train = model.evaluate(Xmtrain, ymtrain, batch_size=16, verbose=2)
    print("Train Data Accuracy Details Epochs = %i" %epoc)
    print(score_train)
    # Test data Scores
    score_test = model.evaluate(Xmtest, ymtest, batch_size=16, verbose=2)
    print("Test Data Accuracy Details Epochs = %i" %epoc)
    print(score_test)

    #Calculate the performance Metric
    ym_pred = model.predict(Xmtest)
    ym_predbinary = (ym_pred > 0.5)

    ##Print Classification Report
    print("\nTEST DATA CLASSIFICATION REPORT WHEN EPOCHS = %i" % epoc)
    print(metrics.classification_report(ymtest, ym_predbinary, target_names=["
    ##Print AUC
    #Finding the TP and FP of the predicted test data
    fpr, tpr, thresholds = metrics.roc_curve(ymtest, ym_predbinary)
    AUC=metrics.auc(fpr, tpr)
    print("AUC = %0.4f \n" % AUC )
    #Plot the ROC Curve
    plt.subplot(313)
    plt.plot(fpr,tpr,label="Epochs=%i ROC curve (area = %0.4f)" % (epoc,AUC))
    plt.plot([0, 1], [0, 1], color="navy", linestyle="--")
    plt.xlim([0.0, 1.0])
    plt.ylim([0.0, 1.05])
    plt.xlabel("False Positive Rate")
    plt.ylabel("True Positive Rate")
    plt.title("Effects of Epochs on DNN performance")
    plt.legend(loc="lower right")
    plt.grid()
plt.tight_layout()
plt.show()
```

```
<IPython.core.display.Javascript object>
```


Training Accuracy Vs Test Accuracy
Test Accuracy(Epoch=10)
Train Accuracy(Epoch=10)
Test Accuracy(Epoch=80)
Train Accuracy(Epoch=80)
Test Accuracy(Epoch=300)
Train Accuracy(Epoch=300)
Test Accuracy(Epoch=3000)
Train Accuracy(Epoch=3000)
1.0
0.5
0
500
1000
1500
2000
Epoch number


Train Loss Vs Test Loss
Test Loss(epoch=10)
Train Loss(epoch=10)
Test Loss(epoch=80)
Train Loss(epoch=80)
Test Loss(epoch=300)
Train Loss(epoch=300)
Test Loss(epoch=3000)
Train Loss(epoch=3000)
0.5
0.0
0
500
1000
1500
2000
Epoch number


Effects of Epochs on DNN performance
Epochs=10 ROC curve (area = 0.8417)
Epochs=80 ROC curve (area = 0.9912)
Epochs=300 ROC curve (area = 0.9825)
Epochs=3000 ROC curve (area = 0.9825)
True Positive Rate
1.0
0.5
0.0
0.0
0.2
0.4
0.6
0.8
1.0
False Positive Rate

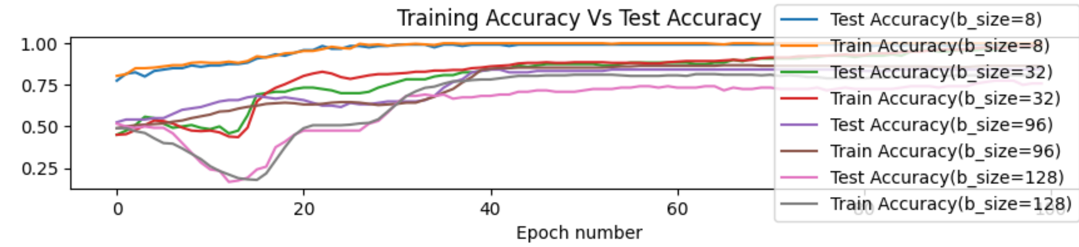
```
18/18 - 0s - loss: 0.3369 - binary_accuracy: 0.8679 - auc: 0.9573
Train Data Accuracy Details Epochs = 10
[0.3368666470050812, 0.8678571581840515, 0.9573017954826355]
8/8 - 0s - loss: 0.3790 - binary_accuracy: 0.8417 - auc: 0.9272
Test Data Accuracy Details Epochs = 10
[0.3789879083633423, 0.8416666388511658, 0.9271790981292725]

TEST DATA CLASSIFICATION REPORT WHEN EPOCHS = 10
              precision    recall  f1-score   support

      True 0       0.85      0.84      0.85        63
      True 1       0.83      0.84      0.83        57

    accuracy                           0.84       120
   macro avg       0.84      0.84      0.84       120
weighted avg       0.84      0.84      0.84       120

AUC = 0.8417

18/18 - 0s - loss: 0.0134 - binary_accuracy: 1.0000 - auc: 1.0000
Train Data Accuracy Details Epochs = 80
[0.013373599387705326, 1.0, 1.0]
8/8 - 0s - loss: 0.0574 - binary_accuracy: 0.9917 - auc: 0.9975
Test Data Accuracy Details Epochs = 80
[0.05740587040781975, 0.9916666746139526, 0.9974937438964844]

TEST DATA CLASSIFICATION REPORT WHEN EPOCHS = 80
              precision    recall  f1-score   support

      True 0       0.98      1.00      0.99        63
      True 1       1.00      0.98      0.99        57

    accuracy                           0.99       120
   macro avg       0.99      0.99      0.99       120
weighted avg       0.99      0.99      0.99       120

AUC = 0.9912

18/18 - 0s - loss: 0.0044 - binary_accuracy: 1.0000 - auc: 1.0000
Train Data Accuracy Details Epochs = 300
[0.004449901171028614, 1.0, 0.9999999403953552]
8/8 - 0s - loss: 0.1377 - binary_accuracy: 0.9833 - auc: 0.9911
Test Data Accuracy Details Epochs = 300
[0.1377221643924713, 0.9833333492279053, 0.9910888671875]

TEST DATA CLASSIFICATION REPORT WHEN EPOCHS = 300
              precision    recall  f1-score   support

      True 0       0.97      1.00      0.98        63
      True 1       1.00      0.96      0.98        57

    accuracy                           0.98       120
   macro avg       0.98      0.98      0.98       120
weighted avg       0.98      0.98      0.98       120

AUC = 0.9825
```

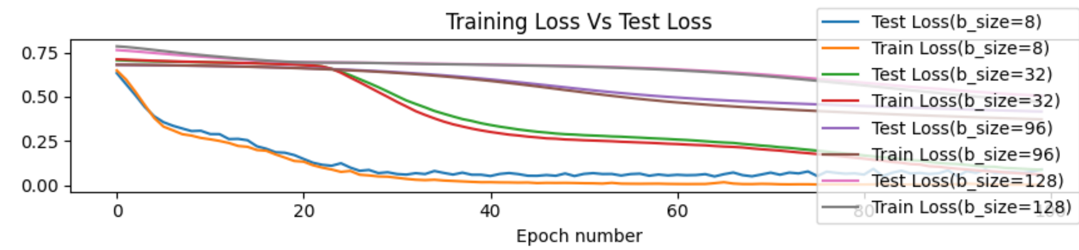
```
18/18 - 0s - loss: 2.9952e-05 - binary_accuracy: 1.0000 - auc: 1.0000
Train Data Accuracy Details Epochs = 3000
[2.9952248951303773e-05, 1.0, 1.0]
8/8 - 0s - loss: 0.2847 - binary_accuracy: 0.9833 - auc: 0.9912
Test Data Accuracy Details Epochs = 3000
[0.2847294509410858, 0.9833333492279053, 0.9912281036376953]

TEST DATA CLASSIFICATION REPORT WHEN EPOCHS = 3000
              precision    recall  f1-score   support

      True 0       0.97      1.00      0.98        63
      True 1       1.00      0.96      0.98        57

    accuracy                           0.98       120
   macro avg       0.98      0.98      0.98       120
weighted avg       0.98      0.98      0.98       120

AUC = 0.9825
```

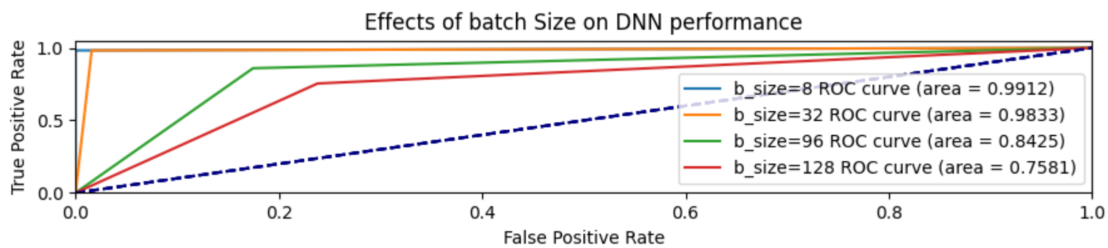
In [206]:
```
#####Checking effects of batch size
#for make model reproducable
np.random.seed(1320418)
tf.random.set_seed(1320418)

def create_model():
    # Define the DNN sequential model
    model = Sequential()
    model.add(tf.keras.layers.InputLayer(input_shape=(2,)))
    model.add(Dense(8, activation='relu'))
    # model.add(Dense(6, activation='tanh'))
    model.add(Dense(4, activation='relu'))
    # model.add(Dense(4, activation='relu'))
    model.add(Dense(2, activation='relu'))
    model.add(Dense(1, activation='sigmoid'))

#     model.summary()
    #Configures the model for training
    model.compile(
        optimizer=tf.keras.optimizers.Adam(learning_rate=0.01,epsilon=0.08),
                    loss=tf.keras.losses.BinaryCrossentropy(),
                    metrics=[tf.keras.metrics.BinaryAccuracy(),tf.keras.metrics.
    )

    # log results
    log_dir = "logs/" + datetime.datetime.now().strftime("%Y%m%d-%H%M%S")
    tensorboard_callback = tf.keras.callbacks.TensorBoard(log_dir=log_dir, his
    return model

##Checking effects of batch size
plt.figure()
bsizes=[8,32,96,128]
for bsize in bsizes:
    tf.keras.backend.clear_session()
    tf.compat.v1.reset_default_graph()
    new_model=create_model()
    # Train the model, iterating on the data in batches, record history
    hist_new = new_model.fit(Xmtrain, ymtrain,validation_data=(Xmtest, ymtest)

    #Plot train_hist.history
    plt.subplot(311)

    plt.plot(hist_new.history['val_binary_accuracy'],label=f"Test Accuracy(b_s
    plt.plot(hist_new.history['binary_accuracy'],label=f"Train Accuracy(b_size
    plt.legend(loc='center right')
    plt.xlabel('Epoch number')
    plt.title('Training Accuracy Vs Test Accuracy')
    plt.grid()
    #Plot test_hist.history
    plt.subplot(312)
    plt.plot(hist_new.history['val_loss'],label=f"Test Loss(b_size={bsize})")
    plt.plot(hist_new.history['loss'],label=f"Train Loss(b_size={bsize})")
    plt.legend(loc='center right')
    plt.xlabel('Epoch number')
    plt.title('Training Loss Vs Test Loss')
    plt.grid()
    # Train data Scores
```

```
    score_train = new_model.evaluate(Xmtrain, ymtrain, batch_size=16, verbose=
    print("Train Data Accuracy Details B_size = %i" %bsize)
    print(score_train)
    # Test data Scores
    score_test = new_model.evaluate(Xmtest, ymtest, batch_size=16, verbose=2)
    print("Test Data Accuracy Details B_size = %i" %bsize)
    print(score_test)

    #Calculate the performance Metric
    ym_pred = new_model.predict(Xmtest)
    ym_predbinary = (ym_pred > 0.5)

    ##Print Classification Report
    print("\nTEST DATA CLASSIFICATION REPORT WHEN B_size = %i" % bsize)
    print(metrics.classification_report(ymtest, ym_predbinary, target_names=["
    ##Print AUC
    #Finding the TP and FP of the predicted test data
    fpr, tpr, thresholds = metrics.roc_curve(ymtest, ym_predbinary)
    AUC=metrics.auc(fpr, tpr)
    print("AUC = %0.4f \n" % AUC )
    #Plot the ROC Curve
    plt.subplot(313)
    plt.plot(fpr,tpr,label="b_size=%i ROC curve (area = %0.4f)" % (bsize,AUC))
    plt.plot([0, 1], [0, 1], color="navy", linestyle="--")
    plt.xlim([0.0, 1.0])
    plt.ylim([0.0, 1.05])
    plt.xlabel("False Positive Rate")
    plt.ylabel("True Positive Rate")
    plt.title("Effects of batch Size on DNN performance")
    plt.legend(loc="lower right")
    plt.grid()
plt.tight_layout()
plt.show()
```

```
<IPython.core.display.Javascript object>
```


Training Accuracy Vs Test Accuracy
1.00
0.75
0.50
0.25
0
20
40
60
Epoch number
Test Accuracy(b_size=8)
Train Accuracy(b_size=8)
Test Accuracy(b_size=32)
Train Accuracy(b_size=32)
Test Accuracy(b_size=96)
Train Accuracy(b_size=96)
Test Accuracy(b_size=128)
Train Accuracy(b_size=128)


Training Loss Vs Test Loss
0.75
0.50
0.25
0.00
0
20
40
60
Epoch number
Test Loss(b_size=8)
Train Loss(b_size=8)
Test Loss(b_size=32)
Train Loss(b_size=32)
Test Loss(b_size=96)
Train Loss(b_size=96)
Test Loss(b_size=128)
Train Loss(b_size=128)


Effects of batch Size on DNN performance
True Positive Rate
1.0
0.5
0.0
0.0
0.2
0.4
0.6
0.8
1.0
False Positive Rate
b_size=8 ROC curve (area = 0.9912)
b_size=32 ROC curve (area = 0.9833)
b_size=96 ROC curve (area = 0.8425)
b_size=128 ROC curve (area = 0.7581)

```
18/18 - 1s - loss: 0.0029 - binary_accuracy: 1.0000 - auc: 1.0000
Train Data Accuracy Details B_size = 8
[0.00286491634324193, 1.0, 1.0]
8/8 - 0s - loss: 0.0890 - binary_accuracy: 0.9917 - auc: 0.9911
Test Data Accuracy Details B_size = 8
[0.08902085572481155, 0.9916666746139526, 0.9910888671875]

TEST DATA CLASSIFICATION REPORT WHEN B_size = 8
              precision    recall  f1-score   support

      True 0       0.98      1.00      0.99        63
      True 1       1.00      0.98      0.99        57

    accuracy                           0.99       120
   macro avg       0.99      0.99      0.99       120
weighted avg       0.99      0.99      0.99       120

AUC = 0.9912

18/18 - 0s - loss: 0.0572 - binary_accuracy: 0.9893 - auc: 0.9995
Train Data Accuracy Details B_size = 32
[0.057199109345674515, 0.9892857074737549, 0.9995406866073608]
8/8 - 0s - loss: 0.0872 - binary_accuracy: 0.9833 - auc: 0.9953
Test Data Accuracy Details B_size = 32
[0.08724681288003922, 0.9833333492279053, 0.9952659606933594]

TEST DATA CLASSIFICATION REPORT WHEN B_size = 32
              precision    recall  f1-score   support

      True 0       0.98      0.98      0.98        63
      True 1       0.98      0.98      0.98        57

    accuracy                           0.98       120
   macro avg       0.98      0.98      0.98       120
weighted avg       0.98      0.98      0.98       120

AUC = 0.9833

18/18 - 0s - loss: 0.3708 - binary_accuracy: 0.8643 - auc: 0.9387
Train Data Accuracy Details B_size = 96
[0.3708094656467438, 0.8642857074737549, 0.9386963248252869]
8/8 - 0s - loss: 0.4161 - binary_accuracy: 0.8417 - auc: 0.9284
Test Data Accuracy Details B_size = 96
[0.4160568118095398, 0.8416666388511658, 0.9284322261810303]

TEST DATA CLASSIFICATION REPORT WHEN B_size = 96
              precision    recall  f1-score   support

      True 0       0.87      0.83      0.85        63
      True 1       0.82      0.86      0.84        57

    accuracy                           0.84       120
   macro avg       0.84      0.84      0.84       120
weighted avg       0.84      0.84      0.84       120

AUC = 0.8425
```

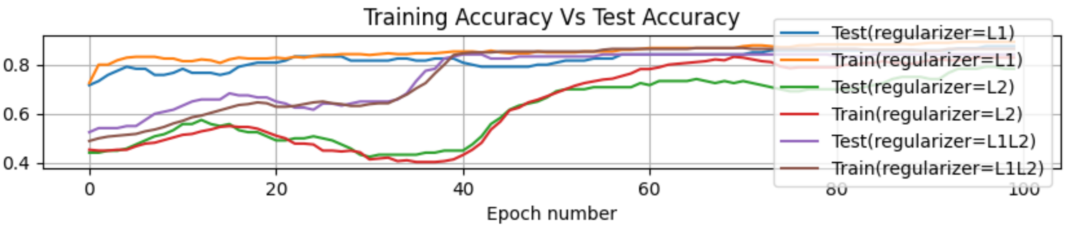
```
18/18 - 0s - loss: 0.4670 - binary_accuracy: 0.8143 - auc: 0.9383
Train Data Accuracy Details B_size = 128
[0.4669854938983917, 0.8142856955528259, 0.9382879734039307]
8/8 - 0s - loss: 0.5045 - binary_accuracy: 0.7583 - auc: 0.8985
Test Data Accuracy Details B_size = 128
[0.504530131816864, 0.7583333253860474, 0.8984962701797485]

TEST DATA CLASSIFICATION REPORT WHEN B_size = 128
              precision    recall  f1-score   support

      True 0       0.77      0.76      0.77        63
      True 1       0.74      0.75      0.75        57

    accuracy                           0.76       120
   macro avg       0.76      0.76      0.76       120
weighted avg       0.76      0.76      0.76       120

AUC = 0.7581
```

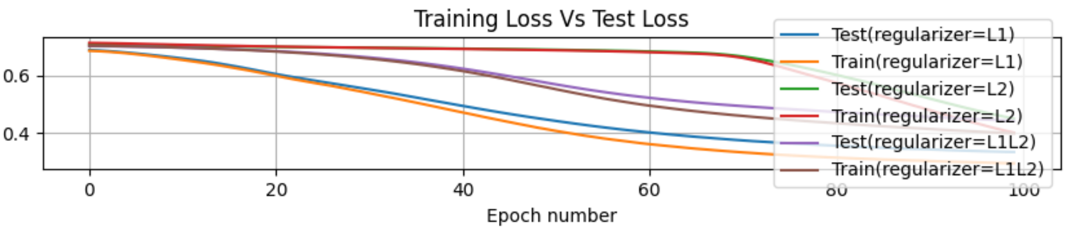
- **It is seen that when training epochs increases the DNN performance (AUC) increases initially and then decreases. In the case of batch sizes, the DNN performance decreases with the increase in the batch size of the training data.**
- **Further when the training epochs increases the test accuracy reduces compared to the training accuracy. Similarly, for higher training batch sizes the difference between training and test accuracy increases**
- **Smaller batches add noise to the learning process and this issue increases the accuracy for the testing data. This is because noise acts as a regulator which prevents over fitting. Also, smaller batches decrease the accuracy for the training data.**
- **The reason for the difference is overfitting of the model**

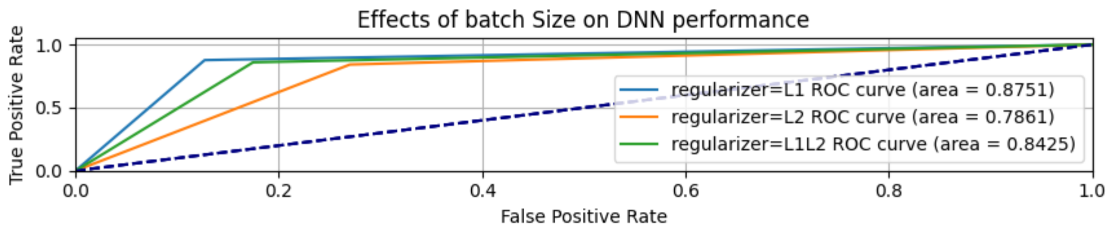
In [217]:
```
#####Different regularizers to prevent Overfitting Methods
import tensorflow.keras.regularizers as regularizer
#for make model reproducable
np.random.seed(1320418)
tf.random.set_seed(1320418)

def create_model(regularizer):
    # Define the DNN sequential model
    model = Sequential()
    model.add(tf.keras.layers.InputLayer(input_shape=(2,)))
    model.add(Dense(8, activation='relu',kernel_regularizer=regularizer,bias_r
    # model.add(Dense(6, activation='tanh'))
    model.add(Dense(4, activation='relu',kernel_regularizer=regularizer,bias_r
    # model.add(Dense(4, activation='relu'))
    model.add(Dense(2, activation='relu',kernel_regularizer=regularizer,bias_r
    model.add(Dense(1, activation='sigmoid'))

#     model.summary()
    #Configures the model for training
    model.compile(
        optimizer=tf.keras.optimizers.Adam(learning_rate=0.01,epsilon=0.08),
                  loss=tf.keras.losses.BinaryCrossentropy(),
                  metrics=[tf.keras.metrics.BinaryAccuracy(),tf.keras.metrics.
    )

    # log results
    log_dir = "logs/" + datetime.datetime.now().strftime("%Y%m%d-%H%M%S")
    tensorboard_callback = tf.keras.callbacks.TensorBoard(log_dir=log_dir, his
    return model

##Checking effects of regularizers
plt.figure()
regularizer_fn=[regularizer.L1(0.001),regularizer.L2(0.0001),regularizers.L1L2
regularizer_str=["L1","L2","L1L2"]
for regul in regularizer_fn:
    tf.keras.backend.clear_session()
    tf.compat.v1.reset_default_graph()
    new_model=create_model(regul)
    # Train the model, iterating on the data in batches, record history
    hist_new = new_model.fit(Xmtrain, ymtrain,validation_data=(Xmtest, ymtest)

    #Plot train_hist.history
    plt.subplot(311)
    regul_name=regularizer_str[regularizer_fn.index(regul)]
    plt.plot(hist_new.history['val_binary_accuracy'],label=f"Test(regularizer=
    plt.plot(hist_new.history['binary_accuracy'],label=f"Train(regularizer={re
    plt.legend(loc='center right')
    plt.xlabel('Epoch number')
    plt.title('Training Accuracy Vs Test Accuracy')
    plt.grid()
    #Plot test_hist.history
    plt.subplot(312)
    plt.plot(hist_new.history['val_loss'],label=f"Test(regularizer={regul_name
    plt.plot(hist_new.history['loss'],label=f"Train(regularizer={regul_name})"
    plt.legend(loc='center right')
    plt.xlabel('Epoch number')
    plt.title('Training Loss Vs Test Loss')
```

```
    plt.grid()
    # Train data Scores
    score_train = new_model.evaluate(Xmtrain, ymtrain, batch_size=96, verbose=
    print("Train Data Accuracy Details regularizer = %s" %regul_name)
    print(score_train)
    # Test data Scores
    score_test = new_model.evaluate(Xmtest, ymtest, batch_size=96, verbose=2)
    print("Test Data Accuracy Details regularizer = %s" %regul_name)
    print(score_test)

    #Calculate the performance Metric
    ym_pred = new_model.predict(Xmtest)
    ym_predbinary = (ym_pred > 0.5)

    ##Print Classification Report
    print("\nTEST DATA CLASSIFICATION REPORT WHEN regularizer = %s" % regul_na
    print(metrics.classification_report(ymtest, ym_predbinary, target_names=["
    ##Print AUC
    #Finding the TP and FP of the predicted test data
    fpr, tpr, thresholds = metrics.roc_curve(ymtest, ym_predbinary)
    AUC=metrics.auc(fpr, tpr)
    print("AUC = %0.4f \n" % AUC )
    #Plot the ROC Curve
    plt.subplot(313)
    plt.plot(fpr,tpr,label="regularizer=%s ROC curve (area = %0.4f)" % (regul_
    plt.plot([0, 1], [0, 1], color="navy", linestyle="--")
    plt.xlim([0.0, 1.0])
    plt.ylim([0.0, 1.05])
    plt.xlabel("False Positive Rate")
    plt.ylabel("True Positive Rate")
    plt.title("Effects of batch Size on DNN performance")
    plt.legend(loc="lower right")
    plt.grid()
plt.tight_layout()
plt.show()
```

```
<IPython.core.display.Javascript object>
```


Training Accuracy Vs Test Accuracy
Test(regularizer=L1)
Train(regularizer=L1)
Test(regularizer=L2)
Train(regularizer=L2)
Test(regularizer=L1L2)
Train(regularizer=L1L2)
Epoch number


Training Loss Vs Test Loss
Test(regularizer=L1)
Train(regularizer=L1)
Test(regularizer=L2)
Train(regularizer=L2)
Test(regularizer=L1L2)
Train(regularizer=L1L2)
Epoch number


Effects of batch Size on DNN performance
regularizer=L1 ROC curve (area = 0.8751)
regularizer=L2 ROC curve (area = 0.7861)
regularizer=L1L2 ROC curve (area = 0.8425)
True Positive Rate
False Positive Rate

```
3/3 - 0s - loss: 0.2934 - binary_accuracy: 0.8893 - auc: 0.9644
Train Data Accuracy Details regularizer = L1
[0.2933901250362396, 0.8892857432365417, 0.9643714427947998]
2/2 - 0s - loss: 0.3336 - binary_accuracy: 0.8750 - auc: 0.9460
Test Data Accuracy Details regularizer = L1
[0.33357861638069153, 0.875, 0.9459760189056396]

TEST DATA CLASSIFICATION REPORT WHEN regularizer = L1
              precision    recall  f1-score   support

      True 0       0.89      0.87      0.88        63
      True 1       0.86      0.88      0.87        57

    accuracy                           0.88       120
   macro avg       0.87      0.88      0.87       120
weighted avg       0.88      0.88      0.88       120

AUC = 0.8751

3/3 - 0s - loss: 0.3958 - binary_accuracy: 0.8286 - auc: 0.9397
Train Data Accuracy Details regularizer = L2
[0.39581823348999023, 0.8285714387893677, 0.9397172331809998]
2/2 - 0s - loss: 0.4493 - binary_accuracy: 0.7833 - auc: 0.9018
Test Data Accuracy Details regularizer = L2
[0.44932466745376587, 0.7833333611488342, 0.901837944984436]

TEST DATA CLASSIFICATION REPORT WHEN regularizer = L2
              precision    recall  f1-score   support

      True 0       0.84      0.73      0.78        63
      True 1       0.74      0.84      0.79        57

    accuracy                           0.78       120
   macro avg       0.79      0.79      0.78       120
weighted avg       0.79      0.78      0.78       120

AUC = 0.7861

3/3 - 0s - loss: 0.3970 - binary_accuracy: 0.8643 - auc: 0.9405
Train Data Accuracy Details regularizer = L1L2
[0.39697468280792236, 0.8642857074737549, 0.9405339360237122]
2/2 - 0s - loss: 0.4419 - binary_accuracy: 0.8417 - auc: 0.9283
Test Data Accuracy Details regularizer = L1L2
[0.4418533444404602, 0.8416666388511658, 0.928292989730835]

TEST DATA CLASSIFICATION REPORT WHEN regularizer = L1L2
              precision    recall  f1-score   support

      True 0       0.87      0.83      0.85        63
      True 1       0.82      0.86      0.84        57

    accuracy                           0.84       120
   macro avg       0.84      0.84      0.84       120
weighted avg       0.84      0.84      0.84       120
```

```
AUC = 0.8425
```

- **Above graph shows that using L1 regularizer suites bests for this problem and it increases the accuracy compared to without regularization. This shows that through regularization we can minimize the overfitting**

Question 1.2 [15%] Wireless Indoor Localization *revisited*

We now revisit the wireless indoor localisation [dataset (http://archive.ics.uci.edu/ml/datasets/Wireless+Indoor+Localization)](http://archive.ics.uci.edu/ml/datasets/Wireless+Indoor+Localization) from WS2. Remember that the data shows the recorded signal strength from 7 different base stations at a smart phone. The phone is in one of the four rooms $\{1, 2, 3, 4\}$. The goal is to classify the location of the phone to one of the four rooms.

1. Solve this classification problem with a DNN. Determine appropriate input and output layers and experiment with different numbers and sizes of hidden layers. ***Hint: you can use, e.g., two sigmoid outputs to binary encode four classes.***
2. Measure performance in different ways using the metrics from Keras or classical Machine Learning as discussed during ML lectures. You can use the same sklearn library functions as in WS2 to document performance. Discuss your findings.

In [4]:
```
dataw = pd.read_csv('files/wifi_localization.csv', names=[f"s{i}" for i in ran
dataw.head() # comment one to see the other
dataw.tail()
```

Out[4]:

	s1	s2	s3	s4	s5	s6	s7	Room Number
1995	-59	-59	-48	-66	-50	-86	-94	4
1996	-59	-56	-50	-62	-47	-87	-90	4
1997	-62	-59	-46	-65	-45	-87	-88	4
1998	-62	-58	-52	-61	-41	-90	-85	4
1999	-59	-50	-45	-60	-45	-88	-87	4

In [5]:
```
print(dataw.size, dataw.shape)
```

```
16000 (2000, 8)
```

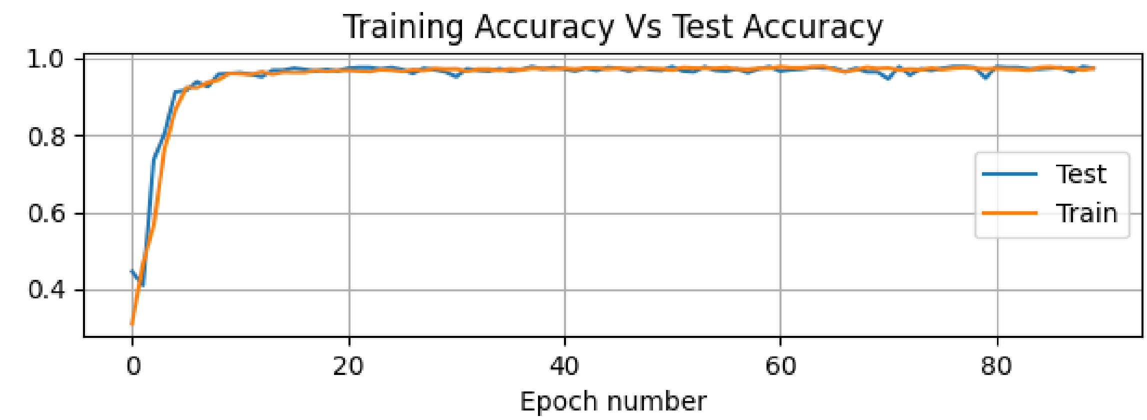
In [43]:
```
SRI = dataw.iloc[:,:7]
# a.shape
loc = dataw.iloc[:,7]-1
# loc.shape


# split into training and test sets
SRItrain, SRItest, loctrain, loctest = train_test_split(SRI, loc,random_state=
# loc.tail()
```

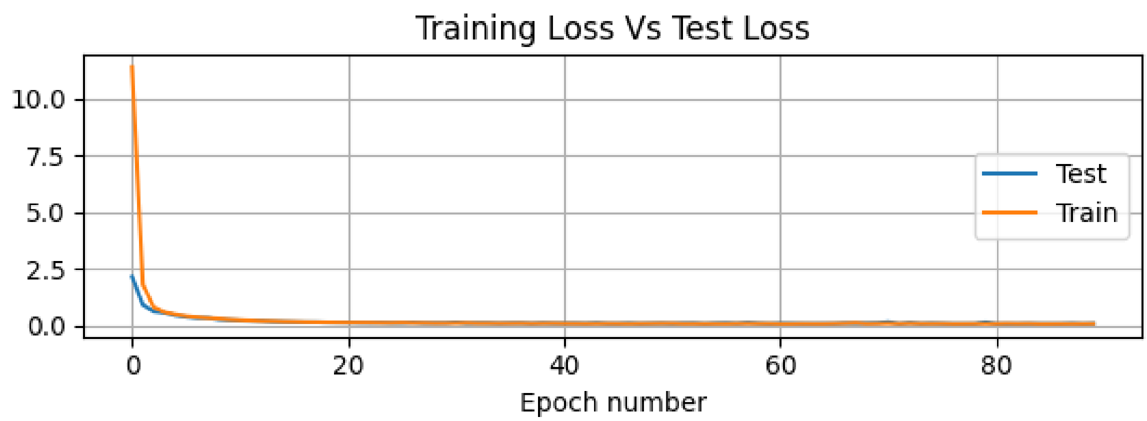
Answer

In [42]:
```
#for make model reproducable
np.random.seed(1320418)
tf.random.set_seed(1320418)

from sklearn.preprocessing import LabelBinarizer
import sklearn.metrics as metrics

def label_encoder(train,test):
    return LabelBinarizer().fit_transform(train),LabelBinarizer().fit_transfor

def create_model():
    # Define the DNN sequential model
    model = Sequential()
    model.add(Dense(14,input_shape=(7,), activation='relu'))
    model.add(Dense(14, activation='relu'))
    model.add(Dense(4, activation='softmax'))

    model.summary()
    #Configures the model for training
    model.compile(
        optimizer=tf.keras.optimizers.Adam(learning_rate=0.01,epsilon=0.2),
                  loss=tf.keras.losses.CategoricalCrossentropy(),
                  metrics=[tf.keras.metrics.CategoricalAccuracy(),tf.keras.met
    )
    return model
# log results
log_dir = "logs/" + datetime.datetime.now().strftime("%Y%m%d-%H%M%S")
tensorboard_callback = tf.keras.callbacks.TensorBoard(log_dir=log_dir, histogr

#Reset Model
tf.keras.backend.clear_session()
tf.compat.v1.reset_default_graph()

#Create Model
WI_model=create_model()

#binarize the labels
binary_loctrain,binary_loctest=label_encoder(loctrain,loctest)

# Train the model, iterating on the data in batches, record history
hist_WI = WI_model.fit(SRItrain, binary_loctrain,validation_data=(SRItest, bin

#Plot train_hist.history
plt.figure()
plt.subplot(211)
plt.plot(hist_WI.history['val_categorical_accuracy'],label="Test")
plt.plot(hist_WI.history['categorical_accuracy'],label="Train")
plt.legend(loc='center right')
plt.xlabel('Epoch number')
plt.title('Training Accuracy Vs Test Accuracy')
plt.grid()
#Plot test_hist.history
plt.subplot(212)
plt.plot(hist_WI.history['val_loss'],label="Test")
plt.plot(hist_WI.history['loss'],label="Train")
plt.legend(loc='center right')
plt.xlabel('Epoch number')
```

```
plt.title('Training Loss Vs Test Loss')
plt.grid()

# Train data Scores
score_train = WI_model.evaluate(SRItrain, binary_loctrain, batch_size=100, ver
print("Train Data Accuracy Details")
print(score_train)
# Test data Scores
score_test = WI_model.evaluate(SRItest, binary_loctest, batch_size=100, verbos
print("Test Data Accuracy Details")
print(score_test)

#Evaluate the performance Metric
binary_locpred = WI_model.predict(SRItest)
# print(SRIpred)
locpred = LabelBinarizer().fit(loctest).inverse_transform(binary_locpred)
# print(locpred)
# Calculate AMI scores
AMI_score=metrics.adjusted_mutual_info_score(loctest, locpred)
print("\nAMI Score for the Model : ",AMI_score) ##Higher the better

##Print Classification Report
print("\nTEST DATA CLASSIFICATION REPORT")
print(metrics.classification_report(loctest, locpred, target_names=["0", "1","

cm = metrics.confusion_matrix(loctest, locpred)
print(pd.DataFrame(cm, columns=["Pred 0", "Pred 1","Pred 2", "Pred 3"], index=

plt.tight_layout()
plt.show()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                Output Shape              Param #
=================================================================
dense (Dense)               (None, 14)                112
_________________________________________________________________
dense_1 (Dense)             (None, 14)                210
_________________________________________________________________
dense_2 (Dense)             (None, 4)                 60
=================================================================
Total params: 382
Trainable params: 382
Non-trainable params: 0
_________________________________________________________________

<IPython.core.display.Javascript object>
```






```
15/15 - 0s - loss: 0.0749 - categorical_accuracy: 0.9740 - auc: 0.9984
Train Data Accuracy Details
[0.07491137832403183, 0.9739999771118164, 0.9984309077262878]
5/5 - 0s - loss: 0.0784 - categorical_accuracy: 0.9740 - auc: 0.9977
Test Data Accuracy Details
[0.07840710133314133, 0.9739999771118164, 0.9977332949638367]

AMI Score for the Model :  0.9129192138328981

TEST DATA CLASSIFICATION REPORT
              precision    recall  f1-score   support

           0       1.00      0.96      0.98       111
           1       0.98      0.97      0.97       132
           2       0.94      0.96      0.95       125
           3       0.99      1.00      0.99       132

    accuracy                           0.97       500
   macro avg       0.97      0.97      0.97       500
weighted avg       0.97      0.97      0.97       500

        Pred 0  Pred 1  Pred 2  Pred 3
True 0     107       0       4       0
True 1       0     128       4       0
True 2       0       3     120       2
True 3       0       0       0     132
```

- **It is observed that by using DNN higher accuracy levels interms of the confusion matrix scores and AMI scores are achieved compared to classical ML techniques**

Question 1.3 [15%] Communications Detective

Your job as a detective is to distinguish malicious people's communications from background civilian communication traffic. As a 21st century detective, you have access to a cognitive radio network and you have ML knowledge!

The [dataset (files/crn_data.csv)](files/crn_data.csv) is collected from a simulation where there are multiple malicious people and civilians communicating in a region with multiple passive cognitive radio nodes. Data about each transmission source is collected from the listener nearest to it. **The objective is to classify if a transmission source is a rogue agent or a civilian based on the data.**

The data file contains data from 2 classes:

- civilians - 129 instances (labeled as +1)
- rogue agents - 129 instances (labeled as -1)

Features/attributes are **not** normalised.

1. label 2. carrier_frequency 2. bandwidth 3. bitrate 4. session duration 5. message_length 6. inter-arrival time (iat)

This is an open-ended mini-project. However, for full points, you should consider:

1. Run a PCA to visualize the data.
2. Try multiple classifiers, e.g. SVM and DNN.
3. Do cross validation, give performance results using metrics, compare/contrast methods.
4. Normalise the data and repeat parts 1, 2 and 3. Discuss your findings.

In [12]:
```
commdata = pd.read_csv('files/crn_data.csv')
commdata.head()
```

Out[12]:

	label	carrier_frequency	bandwidth	bitrate	duration	message_length	iat
0	-1	3.000000e+09	10000000.0	4000000.0	0.000288	138.924685	12.294593
1	-1	3.000000e+09	10000000.0	3000000.0	0.000366	133.339841	12.343191
2	-1	2.000000e+09	15000000.0	3000000.0	0.000369	134.657356	12.494220
3	-1	3.000000e+09	10000000.0	4000000.0	0.000295	142.475129	12.323291
4	-1	2.000000e+09	10000000.0	4000000.0	0.000287	138.554019	12.472884

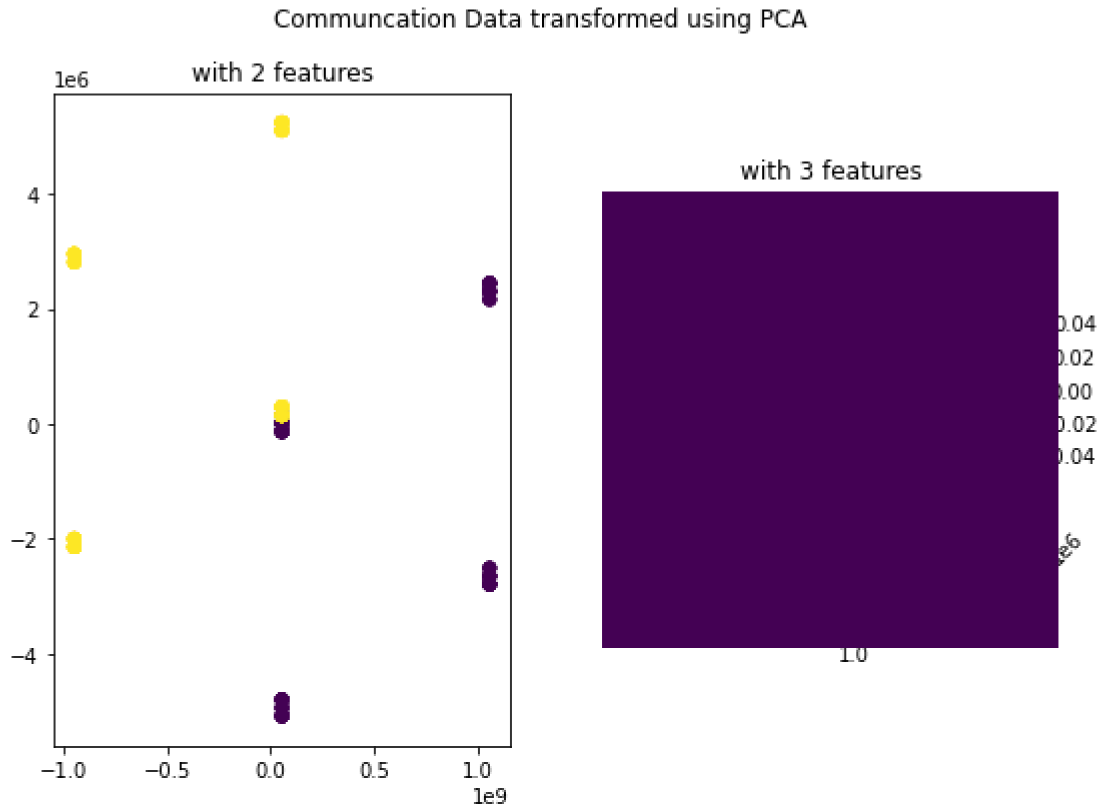
Answer as text here

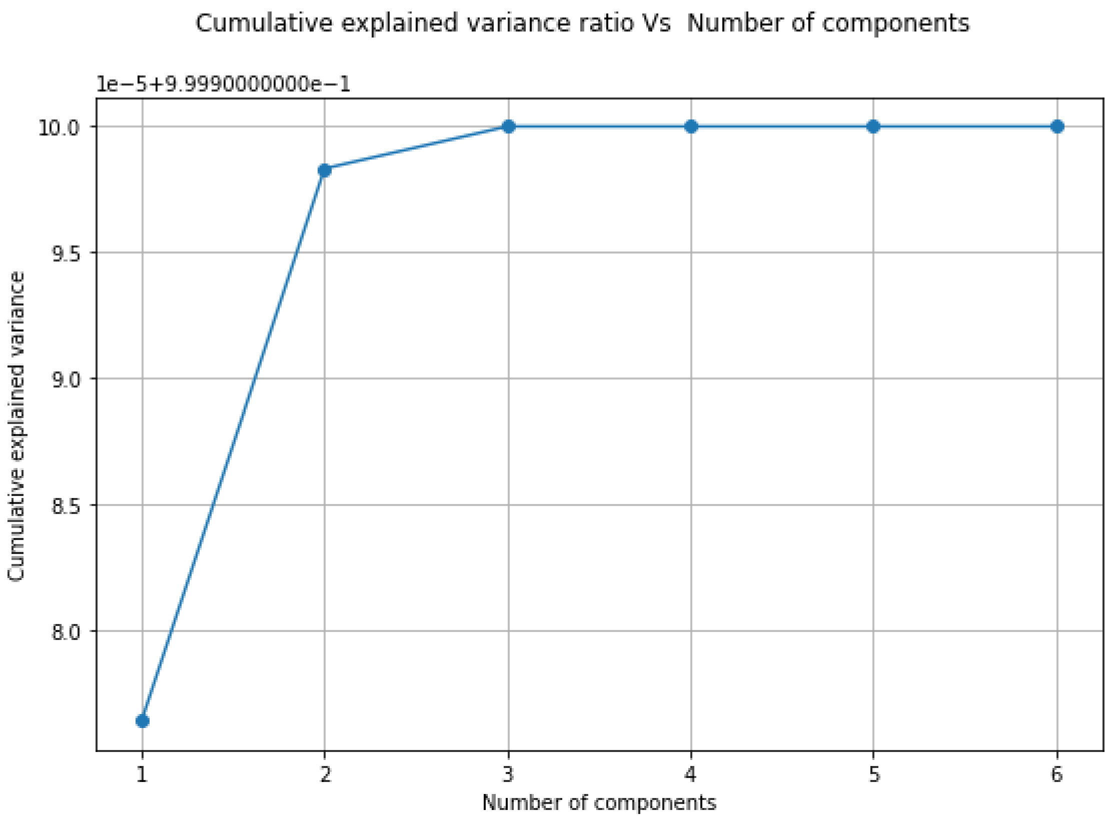
In [13]:
```
cominfo = commdata.iloc[:,1:7]
print(cominfo.shape)
# cominfo.head
person = commdata.iloc[:,0]
# # loc.shape
# person.head
```

```
(258, 6)
```

In [14]:
```
from sklearn.decomposition import PCA
from sklearn import metrics
import warnings
warnings.filterwarnings('ignore')

def run_PCA(X_data):
    #for making model reproducable
    np.random.seed(1320418)
    tf.random.set_seed(1320418)

    #Create a PCA model
    pca=PCA(n_components=6).fit(X_data)
    #Get the variance ratios
    pca_variance_ratios=pca.explained_variance_ratio_
    print(f"Variance Ratios of PCA\n{pca_variance_ratios}\n")
    #Get the singular values
    pca_singular_values=pca.singular_values_
    print(f"Singular Values of PCA\n{pca_singular_values}\n")

    #Transform the data using the PCA model with 2 and 3 features
    person_2=PCA(n_components=2).fit_transform(X_data)
    person_3=PCA(n_components=3).fit_transform(X_data)

    #Plotting Transformed dataset using PCA with 2 and 3 features
    plt.figure(figsize=[9,6]).suptitle("Communcation Data transformed using PC
    ##2D
    plt.subplot(1,2,1)
    plt.title("with 2 features")
    plt.scatter(person_2[:, 0], person_2[:, 1],c=person)
    ##3D
    plt.subplot(1,2,2,projection='3d')
    plt.title("with 3 features")
    plt.scatter(person_3[:, 0], person_3[:, 1], person_3[:, 2], c=person)
    plt.show()

    #Plotting cumulative explained variance ratio Vs the number of components
    cum_sum_pcaVariance=np.cumsum(pca_variance_ratios)
    plt.figure(figsize=[9,6]).suptitle("Cumulative explained variance ratio Vs
    plt.plot(np.linspace(1,6,6),cum_sum_pcaVariance,marker="o")
    plt.xlabel('Number of components')
    plt.ylabel('Cumulative explained variance')
    plt.grid()
    plt.show()
    return person_2

decomposed_cominfo=run_PCA(cominfo)
```

```
Variance Ratios of PCA
[9.99976469e-01 2.18584235e-05 1.67221405e-06 3.70385439e-15
 7.01708500e-20 2.41393273e-25]

Singular Values of PCA
[1.05470811e+10 4.93113411e+07 1.36390251e+07 6.41895525e+02
 2.79393157e+00 5.18203065e-03]
```





- **It is seen that more than 95% of the original data set variance is represented by first two principal components leavings others redundant.**

In [15]:
```
##USING SVM
from sklearn import svm
from sklearn.model_selection import GridSearchCV
from sklearn import preprocessing
from sklearn import metrics
from sklearn.model_selection import KFold
from sklearn.model_selection import cross_validate
from numpy import mean
from numpy import std
from sklearn.metrics import make_scorer

def run_SVM_Model(X_data, y_label):
    #for making model reproducable
    np.random.seed(1320418)
    tf.random.set_seed(1320418)

    #Create SVM Model
    svm_model = svm.SVC(kernel='rbf',random_state=0)

    # prepare the cross-validation procedure
    cv = KFold(n_splits=5, random_state=1, shuffle=True)

    # evaluate model
    scoring = {'precision_macro':'precision_macro', 'recall_macro':'recall_mac
               "roc_auc_macro": make_scorer(metrics.roc_auc_score, average='ma
    scores = cross_validate(svm_model, X_data, y_label, scoring=scoring, cv=cv
    # report performance
    print('SVM Accuracy: mean=%.3f (sd=%.3f)' % (mean(scores["test_accuracy"])
    print('SVM Precision: mean=%.3f (sd=%.3f)' % (mean(scores["test_precision_
    print('SVM recall: mean=%.3f (sd=%.3f)' % (mean(scores["test_recall_macro"
    print('SVM F1 score: mean=%.3f (sd=%.3f)' % (mean(scores["test_f1_macro"])
    print('SVM AUC: mean=%.3f (sd=%.3f)' % (mean(scores["test_roc_auc_macro"])

run_SVM_Model(decomposed_cominfo,person)
```

```
SVM Accuracy: mean=0.744 (sd=0.034)
SVM Precision: mean=0.830 (sd=0.023)
SVM recall: mean=0.740 (sd=0.045)
SVM F1 score: mean=0.719 (sd=0.045)
SVM AUC: mean=0.740 (sd=0.045)
```

In [19]:
```
### Using DNN
from tensorflow.keras.wrappers.scikit_learn import KerasClassifier
import warnings
warnings.filterwarnings('ignore')

def create_model():
    # Define the DNN sequential model
    model = Sequential()
    model.add(Dense(8,input_shape=(2,), activation='relu'))
    model.add(Dense(4, activation='relu'))
    model.add(Dense(1, activation='sigmoid'))

#     model.summary()
    #Configures the model for training
    model.compile(
        optimizer=tf.keras.optimizers.Adam(learning_rate=0.01,epsilon=0.02),
                  loss=tf.keras.losses.BinaryCrossentropy(),
                  metrics=[tf.keras.metrics.BinaryAccuracy(),tf.keras.metrics.
    )
    return model

# log results
log_dir = "logs/" + datetime.datetime.now().strftime("%Y%m%d-%H%M%S")
tensorboard_callback = tf.keras.callbacks.TensorBoard(log_dir=log_dir, histogr

def run_DNN_Model(X_data,y_label):
    #for make model reproducable
    np.random.seed(1320418)
    tf.random.set_seed(1320418)

    #Reset Model
    tf.keras.backend.clear_session()
    tf.compat.v1.reset_default_graph()

    #Create Model
    comms_model=create_model()

    # prepare the cross-validation procedure
    kfold = KFold(n_splits=5, shuffle=True,random_state=1)

    #evaluate model
    estimator = KerasClassifier(build_fn=create_model, epochs=100, batch_size=
    scoring = {'precision_macro':'precision_macro', 'recall_macro':'recall_mac
               "roc_auc_macro": make_scorer(metrics.roc_auc_score, average='ma

    scores = cross_validate(estimator,X_data,y_label,scoring=scoring, cv=kfold
    # report performance
    print('DNN Accuracy: mean=%.3f (sd=%.3f)' % (mean(scores["test_accuracy"])
    print('DNN Precision: mean=%.3f (sd=%.3f)' % (mean(scores["test_precision_
    print('DNN recall: mean=%.3f (sd=%.3f)' % (mean(scores["test_recall_macro"
    print('DNN F1 score: mean=%.3f (sd=%.3f)' % (mean(scores["test_f1_macro"])
    print('DNN AUC: mean=%.3f (sd=%.3f)' % (mean(scores["test_roc_auc_macro"])
```

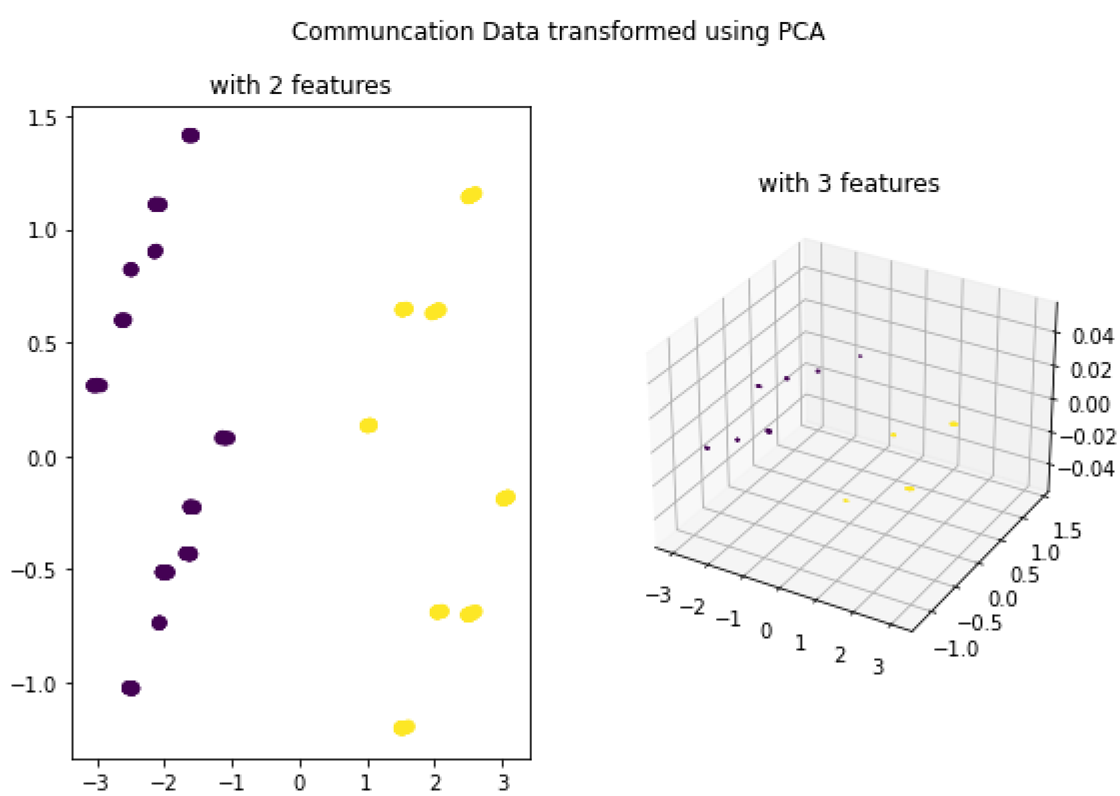
```
run_DNN_Model(decomposed_cominfo,person)
```

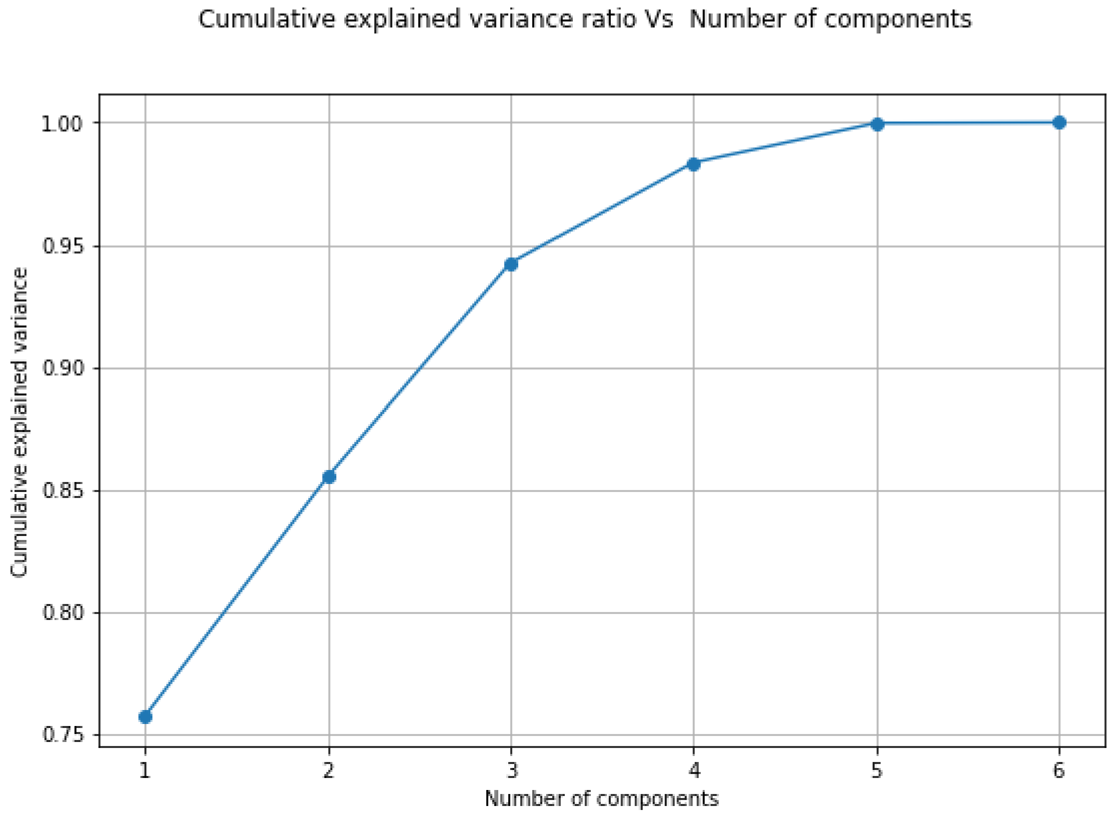
```
DNN Accuracy: mean=0.778 (sd=0.169)
DNN Precision: mean=0.761 (sd=0.269)
DNN recall: mean=0.777 (sd=0.152)
DNN F1 score: mean=0.739 (sd=0.222)
DNN AUC: mean=0.777 (sd=0.152)
```

- **It is seen that the performance of DNN shows a slightly better performance thanSVM, however both shows less performance to this dataset. Specially in DNN the training accuracy is very small even when the hyperparameters are changed.**
- **Main reason for poor performance in both models is due to the data being not normalized. This means there can be few features which has significantly large variances compared to others and they dominate the objective function preventing the model to learn from others.**

In [20]:
```
import warnings
warnings.filterwarnings('ignore')
from sklearn import preprocessing

#finding the mean and std
scaler = preprocessing.StandardScaler().fit(cominfo)
print(f"Mean before normalization\n {scaler.mean_}")
normalized_cominfo=scaler.transform(cominfo)

#run PCA
decomposed_normalized_cominfo=run_PCA(normalized_cominfo)
#run SVM
run_SVM_Model(decomposed_normalized_cominfo,person)
#RUN DNN
run_DNN_Model(decomposed_normalized_cominfo,person)
```

```
Mean before normalization
 [1.94573643e+09 1.50193798e+07 2.30620155e+06 1.13266699e-03
 2.20187468e+02 9.20356984e+00]
Variance Ratios of PCA
[7.57118413e-01 9.79370144e-02 8.73465283e-02 4.11352384e-02
 1.62077192e-02 2.55086769e-04]

Singular Values of PCA
[34.23476746 12.31285906 11.6280878   7.97980884  5.00894693  0.62839026]
```






```
SVM Accuracy: mean=1.000 (sd=0.000)
SVM Precision: mean=1.000 (sd=0.000)
SVM recall: mean=1.000 (sd=0.000)
SVM F1 score: mean=1.000 (sd=0.000)
SVM AUC: mean=1.000 (sd=0.000)
DNN Accuracy: mean=1.000 (sd=0.000)
DNN Precision: mean=1.000 (sd=0.000)
DNN recall: mean=1.000 (sd=0.000)
DNN F1 score: mean=1.000 (sd=0.000)
DNN AUC: mean=1.000 (sd=0.000)
```

- **After PCA and normalization both SVM and DNN achieved perfect performance. This is because models were able to learn from all features since all of them were normalized to zero mean and unit variance**

Section 2: Time Series Estimation

We will next use household electrical power demand as an interesting time-series, which is relevant to power systems and electrical engineering.

Electrical Power Household Demand Estimation

Estimating household power consumption is an important problem in power systems. The demand estimation is easy at the state or regional level due to low-pass filtering (or the law of large numbers) effect from aggregating thousands or even millions of customers' demands. The problem is much more challenging when the demand of individual houses is studied. It is almost impossible to predict when an individual household is going to boil water in the kettle or put on a load of washing. However, it is still possible to make good estimates.

We are given the yearly power consumption of two houses.

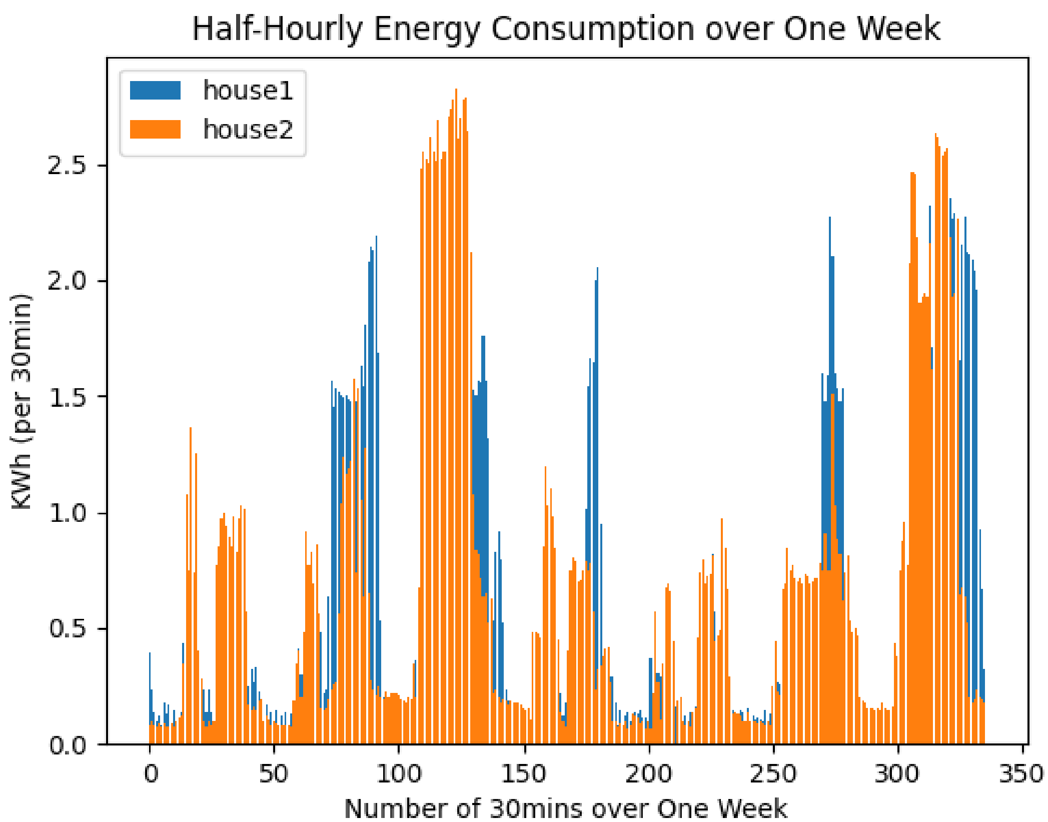
In [4]:
```
raw_data = pd.read_csv('files/two_houses.csv')
raw_data.head()
```

Out[4]:

	day	time	house1	house2
0	0	SMAPV3001	0.288	0.150
1	0	SMAPV3002	0.394	0.081
2	0	SMAPV3003	0.238	0.094
3	0	SMAPV3004	0.138	0.081
4	0	SMAPV3005	0.094	0.075

In [58]:
```
house1 = raw_data.iloc[1:,2]
house2 = raw_data.iloc[1:,3]

plt.figure()
plt.bar(np.arange(48*7),house1[0:48*7])
plt.bar(np.arange(48*7),house2[0:48*7])
plt.title('Half-Hourly Energy Consumption over One Week')
plt.xlabel('Number of 30mins over One Week')
plt.ylabel('KWh (per 30min)')
plt.legend(['house1','house2'])
plt.show()

house1.shape, house2.shape
house1.head
```

```
<IPython.core.display.Javascript object>
```



Out[58]:
```
<bound method NDFrame.head of 1        0.394
2        0.238
3        0.138
4        0.094
5        0.119
         ...
17563    0.263
17564    0.256
17565    0.306
17566    0.294
17567    0.263
Name: house1, Length: 17567, dtype: float64>
```

Question 2.1 [10%] Time Series Estimation using ARMA Models

Use the ARMA linear estimation method to estimate the power consumption of house 1 and house 2. You can use statsmodel time series analysis tools (https://www.statsmodels.org/stable/tsa.html) for this.

1. Define and fit an ARMA model for the first 960 data points. Next, forecast the next 48 points. Measure your performance, e.g. in terms of Mean-squared Error (MSE) using statsmodels tools (https://www.statsmodels.org/stable/tools.html#measure-for-fit-performance-eval-measures), and plot results.
2. Try different AR and MA degrees and different data/time windows. Document and discuss your observations, including the performance vs fitting time trade-off (being careful not to let your model fit for hours!)

Hints: see ARIMA model (https://www.statsmodels.org/stable/generated/statsmodels.tsa.arima.model.ARIMA.html), ARIMA results (https://www.statsmodels.org/stable/generated/statsmodels.tsa.arima.model.ARIMAResults.html), and ARIMA example (https://www.statsmodels.org/stable/examples/notebooks/generated/tsa_arma_1.html)

This is an alternative example implementation. (https://machinelearningmastery.com/arima-for-time-series-forecasting-with-python/)

Pointers for solution

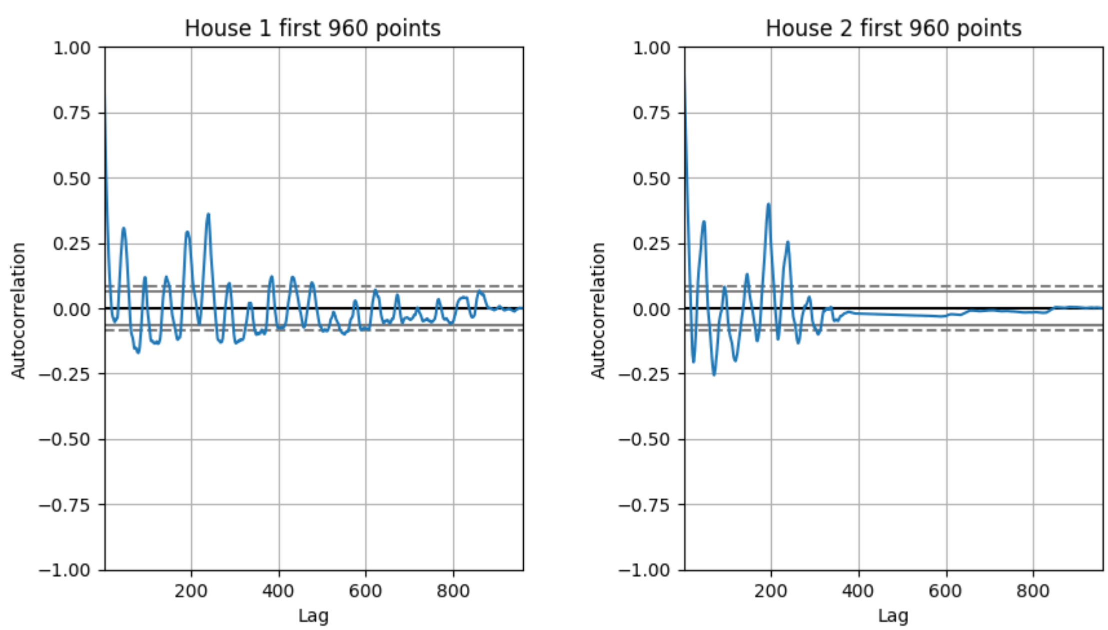
- Use ARIMA model with order (p, 0, q) for implementing a pure ARMA model. ARIMA (https://otexts.com/fpp2/arima.html) differs from ARMA.
- Specific commands to use are ARIMA (https://www.statsmodels.org/stable/generated/statsmodels.tsa.arima_model.ARIMA.html) for creating the model and ARIMA.fit (https://www.statsmodels.org/stable/generated/statsmodels.tsa.arima.model.ARIMA.fit.html) with appropriate arguments as documented.
- After training, ARIMAResults (https://www.statsmodels.org/stable/generated/statsmodels.tsa.arima.model.ARIMAResults.html) functions such as `summary()` , `fittedvalues` , `params` , and `forecast(steps=nbrsteps)` will be very useful.

In [17]:
```
from statsmodels.tsa.arima.model import ARIMA
```

Answer as text here

In [49]:
```
%matplotlib notebook
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from statsmodels.graphics.tsaplots import plot_predict

#To make the randomness reproducable
np.random.seed(1320418)
train_size=960

#Split the data to train and test
house1_train,house1_test=house1.iloc[0:train_size],house1.iloc[train_size:trai
house2_train,house2_test=house2.iloc[0:train_size],house2.iloc[train_size:trai

#Plot the covaraince of the two houses
plt.subplot(121)
pd.plotting.autocorrelation_plot(house1_train)
plt.title("House 1 first 960 points")
plt.subplot(122)
pd.plotting.autocorrelation_plot(house2_train)
plt.title("House 2 first 960 points")
plt.tight_layout()
plt.show()
```

```
<IPython.core.display.Javascript object>
```



In [89]:
```
#Define and Fit the ARMA model for train data
arma_mod_h1 = ARIMA(house1_train, order=(80, 0, 3))
arma_res_h1 = arma_mod_h1.fit()

#Display the fit summary
print(arma_res_h1.summary())

#Plot the fitted data
fig1, ax1 = plt.subplots(figsize=(9, 6))
fig1=plot_predict(arma_res_h1,ax=ax1)
fig1.suptitle("Fitted ARMA Model House 1")

#Define and Fit the ARMA model for train data
arma_mod_h2 = ARIMA(house2_train, order=(60, 0, 2))
arma_res_h2 = arma_mod_h2.fit()

#Display the fit summary
print(arma_res_h2.summary())

#Plot the fitted data
fig2, ax2 = plt.subplots(figsize=(9, 6))
fig2=plot_predict(arma_res_h2,ax=ax2)
fig2.suptitle("Fitted ARMA Model House 2")
```

```
C:\Users\pmendis\Anaconda3\envs\tf-ELEN90088\lib\site-packages\statsmodels\ba
se\model.py:604: ConvergenceWarning: Maximum Likelihood optimization failed t
o converge. Check mle_retvals
  warnings.warn("Maximum Likelihood optimization failed to "
```

```
                               SARIMAX Results
==============================================================================
=
Dep. Variable:                 house1   No. Observations:                  96
0
Model:                  ARIMA(80, 0, 3)   Log Likelihood                 238.74
6
Date:                 Mon, 04 Apr 2022   AIC                           -307.49
2
Time:                         17:36:44   BIC                            106.19
7
Sample:                              0   HQIC                          -149.95
0
                                 - 960
Covariance Type:                   opg
==============================================================================
=
                 coef    std err          z      P>|z|      [0.025      0.97
5]
------------------------------------------------------------------------------
-
const          0.3841      0.104      3.707      0.000       0.181       0.58
7
ar.L1          0.4884      1.351      0.362      0.718      -2.160       3.13
6
ar.L2          0.5416      1.738      0.312      0.755      -2.866       3.94
9
ar.L3          0.0266      1.341      0.020      0.984      -2.602       2.65
5
ar.L4         -0.0983      1.091     -0.090      0.928      -2.237       2.04
0
ar.L5         -0.1133      0.086     -1.319      0.187      -0.282       0.05
5
ar.L6         -0.0276      0.191     -0.145      0.885      -0.401       0.34
6
ar.L7         -0.0036      0.179     -0.020      0.984      -0.354       0.34
7
ar.L8          0.1420      0.149      0.955      0.340      -0.149       0.43
3
ar.L9         -0.0409      0.204     -0.201      0.841      -0.440       0.35
9
ar.L10        -0.0153      0.226     -0.068      0.946      -0.458       0.42
8
ar.L11        -0.0585      0.183     -0.319      0.750      -0.418       0.30
1
ar.L12         0.0147      0.157      0.094      0.925      -0.292       0.32
2
ar.L13        -0.0139      0.122     -0.114      0.909      -0.252       0.22
5
ar.L14         0.0423      0.109      0.387      0.699      -0.172       0.25
6
ar.L15         0.0185      0.102      0.182      0.856      -0.181       0.21
8
ar.L16         0.0009      0.103      0.009      0.993      -0.200       0.20
2
ar.L17        -0.0189      0.093     -0.203      0.839      -0.201       0.16
3
```

```
ar.L18        -0.0410      0.069     -0.595      0.552      -0.176       0.09
4
ar.L19        -0.0061      0.076     -0.080      0.936      -0.156       0.14
4
ar.L20        -0.0008      0.079     -0.010      0.992      -0.155       0.15
3
ar.L21         0.0243      0.079      0.307      0.759      -0.131       0.18
0
ar.L22         0.0061      0.074      0.083      0.933      -0.138       0.15
0
ar.L23         0.0599      0.070      0.854      0.393      -0.077       0.19
7
ar.L24        -0.0391      0.109     -0.357      0.721      -0.254       0.17
5
ar.L25        -0.0218      0.132     -0.166      0.868      -0.280       0.23
6
ar.L26        -0.0525      0.121     -0.435      0.664      -0.289       0.18
4
ar.L27         0.0174      0.120      0.145      0.885      -0.218       0.25
3
ar.L28         0.0740      0.108      0.687      0.492      -0.137       0.28
5
ar.L29         0.0352      0.123      0.285      0.776      -0.207       0.27
7
ar.L30        -0.0522      0.125     -0.416      0.677      -0.298       0.19
4
ar.L31        -0.0250      0.108     -0.231      0.817      -0.237       0.18
7
ar.L32        -0.0953      0.093     -1.026      0.305      -0.277       0.08
7
ar.L33        -0.0241      0.162     -0.149      0.881      -0.341       0.29
3
ar.L34         0.0960      0.146      0.659      0.510      -0.190       0.38
2
ar.L35         0.0878      0.165      0.533      0.594      -0.235       0.41
0
ar.L36         0.0076      0.143      0.053      0.958      -0.272       0.28
7
ar.L37        -0.0666      0.137     -0.486      0.627      -0.335       0.20
2
ar.L38        -0.0122      0.097     -0.126      0.900      -0.202       0.17
8
ar.L39         0.0009      0.080      0.011      0.991      -0.155       0.15
7
ar.L40         0.0540      0.072      0.753      0.451      -0.087       0.19
5
ar.L41         0.0076      0.093      0.082      0.935      -0.175       0.19
1
ar.L42        -0.0201      0.081     -0.248      0.804      -0.179       0.13
9
ar.L43         0.0199      0.082      0.243      0.808      -0.140       0.18
0
ar.L44         0.0323      0.067      0.484      0.628      -0.098       0.16
3
ar.L45        -0.0407      0.071     -0.572      0.567      -0.180       0.09
9
ar.L46         0.0175      0.088      0.199      0.842      -0.155       0.19
```

```
0
ar.L47         0.0086      0.086      0.100      0.921      -0.160       0.17
7
ar.L48         0.0177      0.080      0.221      0.825      -0.139       0.17
5
ar.L49         0.0085      0.078      0.109      0.913      -0.144       0.16
1
ar.L50        -0.0512      0.067     -0.760      0.447      -0.183       0.08
1
ar.L51         0.0379      0.087      0.436      0.663      -0.132       0.20
8
ar.L52        -0.0411      0.116     -0.355      0.723      -0.268       0.18
6
ar.L53         0.0228      0.126      0.180      0.857      -0.225       0.27
0
ar.L54         0.0167      0.130      0.129      0.898      -0.237       0.27
1
ar.L55        -0.0150      0.114     -0.131      0.896      -0.239       0.20
9
ar.L56         0.0101      0.080      0.127      0.899      -0.146       0.16
6
ar.L57         0.0586      0.070      0.834      0.404      -0.079       0.19
6
ar.L58        -0.0669      0.091     -0.732      0.464      -0.246       0.11
2
ar.L59        -0.1063      0.141     -0.752      0.452      -0.383       0.17
1
ar.L60         0.0606      0.156      0.389      0.697      -0.245       0.36
6
ar.L61         0.0718      0.190      0.378      0.705      -0.301       0.44
4
ar.L62        -0.0084      0.125     -0.067      0.946      -0.252       0.23
6
ar.L63        -0.0773      0.125     -0.619      0.536      -0.322       0.16
7
ar.L64        -0.0341      0.106     -0.323      0.747      -0.241       0.17
3
ar.L65         0.0272      0.115      0.236      0.813      -0.199       0.25
3
ar.L66         0.0332      0.104      0.318      0.750      -0.171       0.23
8
ar.L67         0.0690      0.084      0.822      0.411      -0.096       0.23
3
ar.L68        -0.0151      0.120     -0.126      0.900      -0.250       0.21
9
ar.L69        -0.0148      0.127     -0.116      0.907      -0.264       0.23
4
ar.L70        -0.0635      0.104     -0.609      0.542      -0.268       0.14
1
ar.L71        -0.0381      0.120     -0.319      0.750      -0.273       0.19
6
ar.L72         0.0383      0.107      0.356      0.722      -0.172       0.24
9
ar.L73         0.0823      0.108      0.760      0.447      -0.130       0.29
4
ar.L74        -0.0265      0.109     -0.243      0.808      -0.240       0.18
7
```

```
ar.L75        -0.0053      0.131     -0.041      0.968      -0.262       0.25
2
ar.L76         0.0187      0.106      0.176      0.860      -0.189       0.22
6
ar.L77        -0.0610      0.100     -0.608      0.543      -0.258       0.13
6
ar.L78         0.0043      0.127      0.034      0.973      -0.246       0.25
4
ar.L79         0.0202      0.117      0.172      0.863      -0.210       0.25
0
ar.L80        -0.0353      0.106     -0.331      0.740      -0.244       0.17
3
ma.L1          0.4723      1.355      0.349      0.727      -2.182       3.12
7
ma.L2         -0.1377      1.373     -0.100      0.920      -2.829       2.55
3
ma.L3         -0.1517      1.171     -0.130      0.897      -2.446       2.14
3
sigma2         0.0354      0.001     28.431      0.000       0.033       0.03
8
=============================================================================
======
Ljung-Box (L1) (Q):                  0.00   Jarque-Bera (JB):              6
329.12
Prob(Q):                             0.98   Prob(JB):
0.00
Heteroskedasticity (H):              0.47   Skew:
0.86
Prob(H) (two-sided):                 0.00   Kurtosis:
15.46
=============================================================================
======

Warnings:
[1] Covariance matrix calculated using the outer product of gradients (comple
x-step).

<IPython.core.display.Javascript object>
```

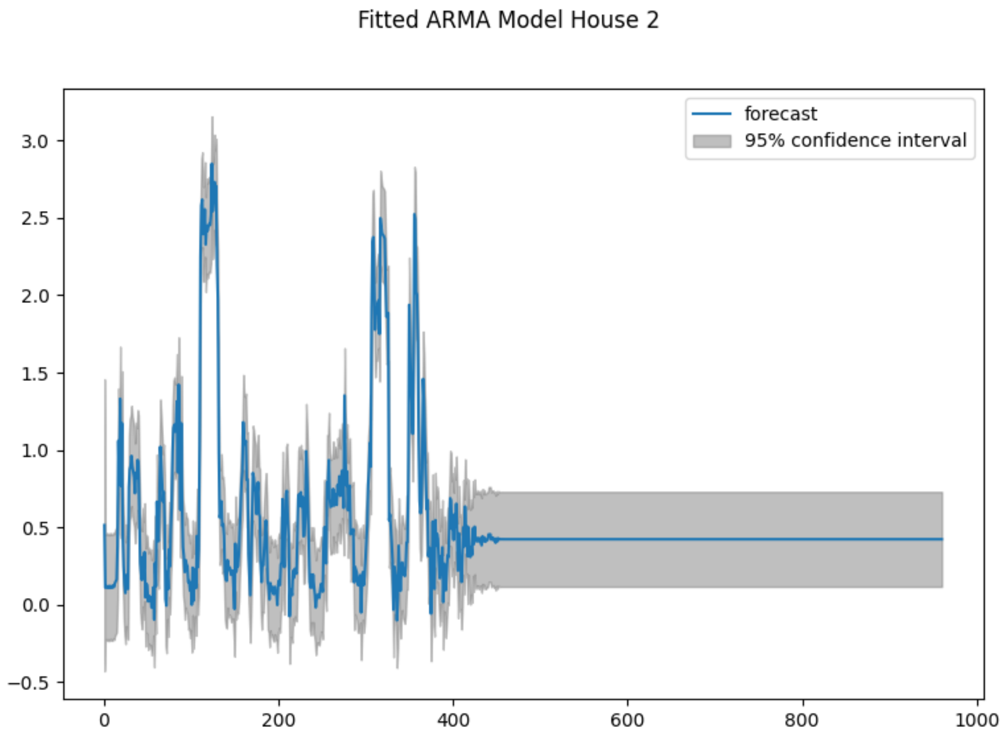

```
C:\Users\pmendis\Anaconda3\envs\tf-ELEN90088\lib\site-packages\statsmodels\ba
se\model.py:604: ConvergenceWarning: Maximum Likelihood optimization failed t
o converge. Check mle_retvals
  warnings.warn("Maximum Likelihood optimization failed to "
```

```
                                     SARIMAX Results
==============================================================================
=
Dep. Variable:                 house2   No. Observations:                   96
0
Model:                 ARIMA(60, 0, 2)   Log Likelihood                 415.29
4
Date:                Mon, 04 Apr 2022   AIC                           -702.58
8
Time:                        17:41:08   BIC                           -391.10
5
Sample:                             0   HQIC                          -583.96
9
                                - 960
Covariance Type:                  opg
==============================================================================
=
                 coef    std err          z      P>|z|      [0.025      0.97
5]
------------------------------------------------------------------------------
-
const          0.5152      0.120      4.304      0.000       0.281       0.75
0
ar.L1          0.5009      4.259      0.118      0.906      -7.847       8.84
9
ar.L2          0.2414      2.429      0.099      0.921      -4.519       5.00
2
ar.L3          0.1396      1.477      0.095      0.925      -2.754       3.03
4
ar.L4          0.0126      0.067      0.188      0.851      -0.119       0.14
4
ar.L5         -0.0472      0.043     -1.111      0.267      -0.131       0.03
6
ar.L6          0.0691      0.200      0.346      0.729      -0.322       0.46
1
ar.L7          0.0143      0.301      0.048      0.962      -0.576       0.60
5
ar.L8          0.0179      0.096      0.186      0.852      -0.170       0.20
6
ar.L9          0.0128      0.051      0.253      0.801      -0.086       0.11
2
ar.L10         0.0484      0.069      0.699      0.485      -0.087       0.18
4
ar.L11        -0.0338      0.207     -0.163      0.870      -0.439       0.37
2
ar.L12        -0.0666      0.173     -0.386      0.700      -0.405       0.27
2
ar.L13         0.0025      0.308      0.008      0.993      -0.601       0.60
6
ar.L14         0.0103      0.072      0.143      0.887      -0.131       0.15
2
ar.L15        -0.0060      0.089     -0.067      0.946      -0.180       0.16
8
ar.L16        -0.0815      0.069     -1.186      0.236      -0.216       0.05
3
ar.L17         0.0600      0.352      0.171      0.865      -0.630       0.75
0
```

```
ar.L18      -0.0189      0.284     -0.067      0.947      -0.575       0.53
8
ar.L19       0.0260      0.079      0.329      0.742      -0.129       0.18
1
ar.L20      -0.1682      0.078     -2.154      0.031      -0.321      -0.01
5
ar.L21      -0.0088      0.697     -0.013      0.990      -1.374       1.35
6
ar.L22       0.1099      0.090      1.219      0.223      -0.067       0.28
7
ar.L23      -0.0802      0.557     -0.144      0.885      -1.172       1.01
1
ar.L24       0.0044      0.405      0.011      0.991      -0.789       0.79
8
ar.L25       0.0896      0.076      1.185      0.236      -0.059       0.23
8
ar.L26      -0.0253      0.456     -0.056      0.956      -0.919       0.86
8
ar.L27       0.0727      0.167      0.435      0.663      -0.255       0.40
0
ar.L28      -0.0199      0.278     -0.072      0.943      -0.564       0.52
4
ar.L29      -0.0309      0.087     -0.358      0.721      -0.201       0.13
9
ar.L30       0.0150      0.177      0.085      0.933      -0.333       0.36
3
ar.L31       0.0786      0.101      0.779      0.436      -0.119       0.27
6
ar.L32       0.0063      0.351      0.018      0.986      -0.682       0.69
5
ar.L33       0.0100      0.055      0.183      0.855      -0.097       0.11
7
ar.L34       0.0102      0.042      0.242      0.809      -0.073       0.09
3
ar.L35      -0.0620      0.070     -0.885      0.376      -0.199       0.07
5
ar.L36       0.0058      0.283      0.021      0.984      -0.550       0.56
1
ar.L37      -0.0233      0.071     -0.328      0.743      -0.163       0.11
6
ar.L38      -0.0085      0.079     -0.107      0.915      -0.163       0.14
6
ar.L39       0.0227      0.068      0.335      0.738      -0.110       0.15
6
ar.L40      -0.1729      0.117     -1.481      0.139      -0.402       0.05
6
ar.L41       0.1178      0.737      0.160      0.873      -1.327       1.56
3
ar.L42      -0.0318      0.540     -0.059      0.953      -1.090       1.02
7
ar.L43       0.0527      0.118      0.447      0.655      -0.178       0.28
3
ar.L44      -0.0347      0.144     -0.242      0.809      -0.316       0.24
7
ar.L45       0.0254      0.126      0.201      0.841      -0.222       0.27
2
ar.L46       0.0285      0.098      0.290      0.772      -0.164       0.22
```

```
1
ar.L47         0.0102      0.144      0.071      0.944      -0.272       0.29
3
ar.L48         0.1247      0.039      3.216      0.001       0.049       0.20
1
ar.L49         0.0547      0.523      0.105      0.917      -0.970       1.08
0
ar.L50         0.0236      0.219      0.108      0.914      -0.406       0.45
3
ar.L51        -0.1540      0.043     -3.541      0.000      -0.239      -0.06
9
ar.L52        -0.0232      0.666     -0.035      0.972      -1.328       1.28
2
ar.L53         0.0048      0.106      0.045      0.964      -0.203       0.21
3
ar.L54        -0.0555      0.112     -0.497      0.619      -0.275       0.16
4
ar.L55         0.0162      0.256      0.063      0.950      -0.486       0.51
9
ar.L56        -0.0942      0.088     -1.072      0.284      -0.266       0.07
8
ar.L57         0.0843      0.369      0.228      0.819      -0.639       0.80
7
ar.L58        -0.0483      0.352     -0.137      0.891      -0.739       0.64
3
ar.L59         0.0682      0.195      0.350      0.726      -0.314       0.45
0
ar.L60        -0.0062      0.236     -0.026      0.979      -0.469       0.45
7
ma.L1          0.3815      4.258      0.090      0.929      -7.965       8.72
8
ma.L2          0.1410      1.460      0.097      0.923      -2.720       3.00
2
sigma2         0.0244      0.001     38.962      0.000       0.023       0.02
6
=============================================================================
======
Ljung-Box (L1) (Q):                  0.00   Jarque-Bera (JB):            23
699.13
Prob(Q):                             0.98   Prob(JB):
0.00
Heteroskedasticity (H):              0.00   Skew:
1.58
Prob(H) (two-sided):                 0.00   Kurtosis:
27.13
=============================================================================
======

Warnings:
[1] Covariance matrix calculated using the outer product of gradients (comple
x-step).
```

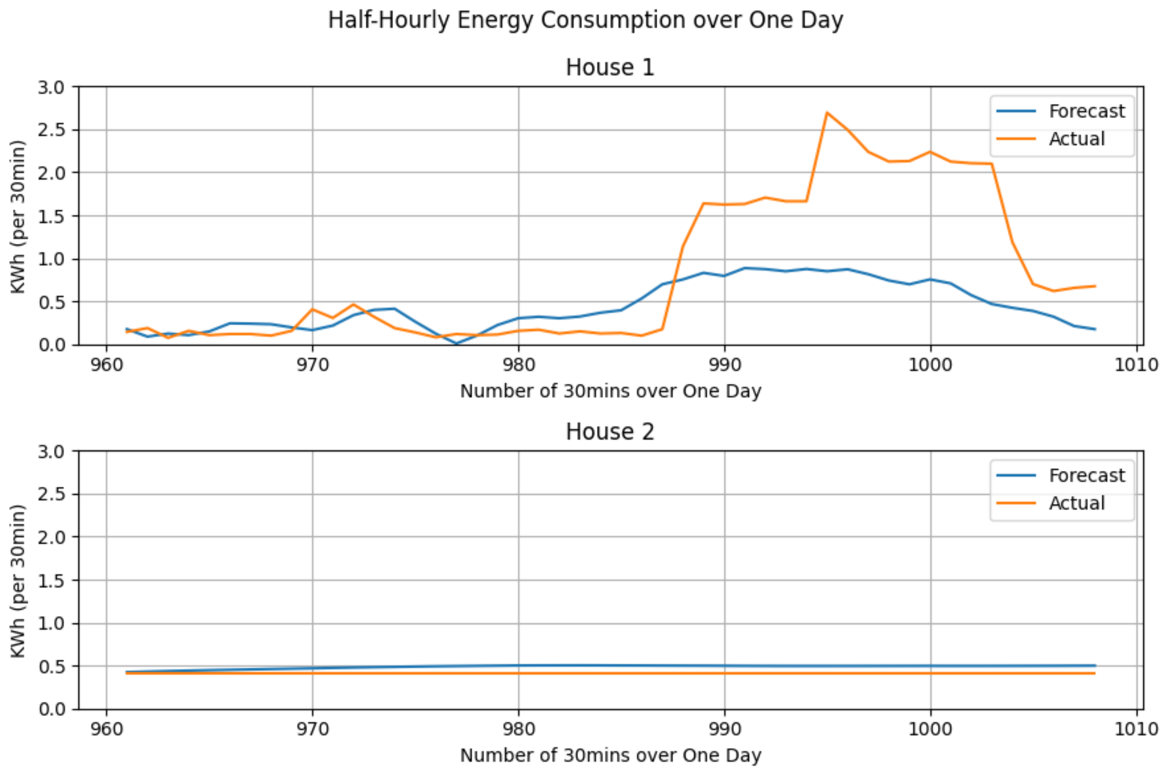
<IPython.core.display.Javascript object>



Out[89]: Text(0.5, 0.98, 'Fitted ARMA Model House 2')

In [90]:
```
from sklearn.metrics import mean_squared_error

#Forecast new data from the ARMA model
arma_h1_forecast = arma_res_h1.forecast(steps=48)
arma_h2_forecast = arma_res_h2.forecast(steps=48)
# print(arma_h1_forecast)

#Plot the HOuse 1 forcasted Vs actual
plt.figure(figsize=[9,6]).suptitle('Half-Hourly Energy Consumption over One Day
plt.subplot(211)
plt.plot(arma_h1_forecast,label="Forecast")
plt.plot(house1_test,label="Actual")
plt.title("House 1")
plt.xlabel('Number of 30mins over One Day')
plt.ylabel('KWh (per 30min)')
plt.ylim(0,3)
plt.grid()
plt.legend()

#Plot the HOuse 2 forcasted Vs actual
plt.subplot(212)
plt.plot(arma_h2_forecast,label="Forecast")
plt.plot(house2_test,label="Actual")
plt.title("House 2")
plt.xlabel('Number of 30mins over One Day')
plt.ylabel('KWh (per 30min)')
plt.ylim(0,3)
plt.legend()
plt.grid()
plt.tight_layout()
plt.show()

# evaluate forecasts
mse_h1 = mean_squared_error(house1_test, arma_h1_forecast)
print('H1 Forecast MSE: %.3f' % mse_h1)
mse_h2 = mean_squared_error(house2_test, arma_h2_forecast)
print('H2 Forecast MSE: %.3f' % mse_h2)
```

```
<IPython.core.display.Javascript object>
```

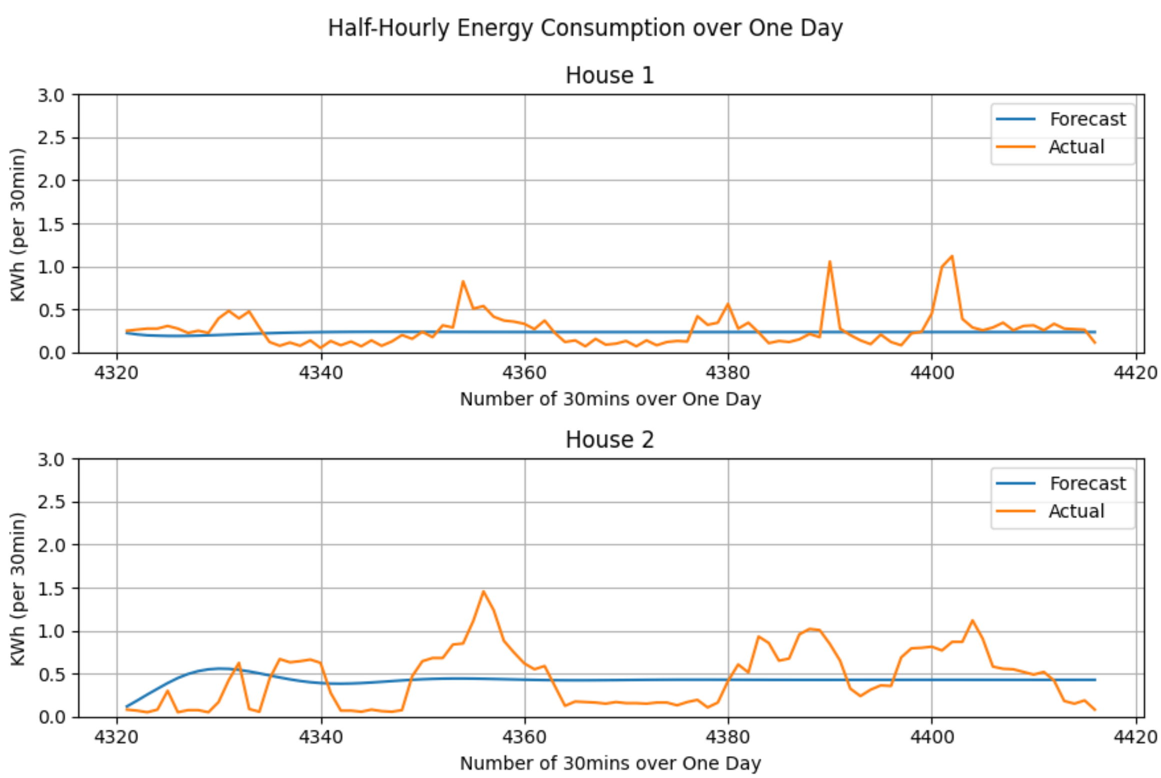



```
H1 Forecast MSE: 0.572
H2 Forecast MSE: 0.005
```

In [99]:
```
from time import time
import warnings
warnings.filterwarnings('ignore')

def split_data(start,end, test_period):
    #Split the data to train and test
    house1_train,house1_test=house1.iloc[start:end],house1.iloc[end:end+test_p
    house2_train,house2_test=house2.iloc[start:end],house2.iloc[end:end+test_p
    return house1_train,house1_test,house2_train,house2_test

def fit_arma(p,q,house1_train,house2_train):
    print(f'Parameters p={p} q={q}')
    #Define and Fit the ARMA model for train data
    arma_mod_h1 = ARIMA(house1_train, order=(p, 0, q))
    t1=time()
    arma_res_h1 = arma_mod_h1.fit()
    t2=time()
    #Display the fit summary
#     print(arma_res_h1.summary())

    #Define and Fit the ARMA model for train data
    arma_mod_h2 = ARIMA(house2_train, order=(p, 0, q))
    t3=time()
    arma_res_h2 = arma_mod_h2.fit()
    t4=time()
    #Display the fit summary
#     print(arma_res_h2.summary())

    return t2-t1,t4-t3,arma_res_h1,arma_res_h2

def forecast_and_measure_performance(test_period,house1_test,house2_test,arma_
    #Forecast new data from the ARMA model
    arma_h1_forecast = arma_res_h1.forecast(steps=test_period)
    arma_h2_forecast = arma_res_h2.forecast(steps=test_period)
    # print(arma_h1_forecast)
    # evaluate forecasts
    mse_h1 = mean_squared_error(house1_test, arma_h1_forecast)
    print('H1 Forecast MSE: %.3f' % mse_h1)
    mse_h2 = mean_squared_error(house2_test, arma_h2_forecast)
    print('H2 Forecast MSE: %.3f' % mse_h2)

    #Plot the HOuse 1 forcasted Vs actual
    plt.figure(figsize=[9,6]).suptitle('Half-Hourly Energy Consumption over On
    plt.subplot(211)
    plt.plot(arma_h1_forecast,label="Forecast")
    plt.plot(house1_test,label="Actual")
    plt.title("House 1")
    plt.xlabel('Number of 30mins over One Day')
    plt.ylabel('KWh (per 30min)')
    plt.ylim(0,3)
    plt.grid()
    plt.legend()

    #Plot the HOuse 2 forcasted Vs actual
    plt.subplot(212)
    plt.plot(arma_h2_forecast,label="Forecast")
    plt.plot(house2_test,label="Actual")
```

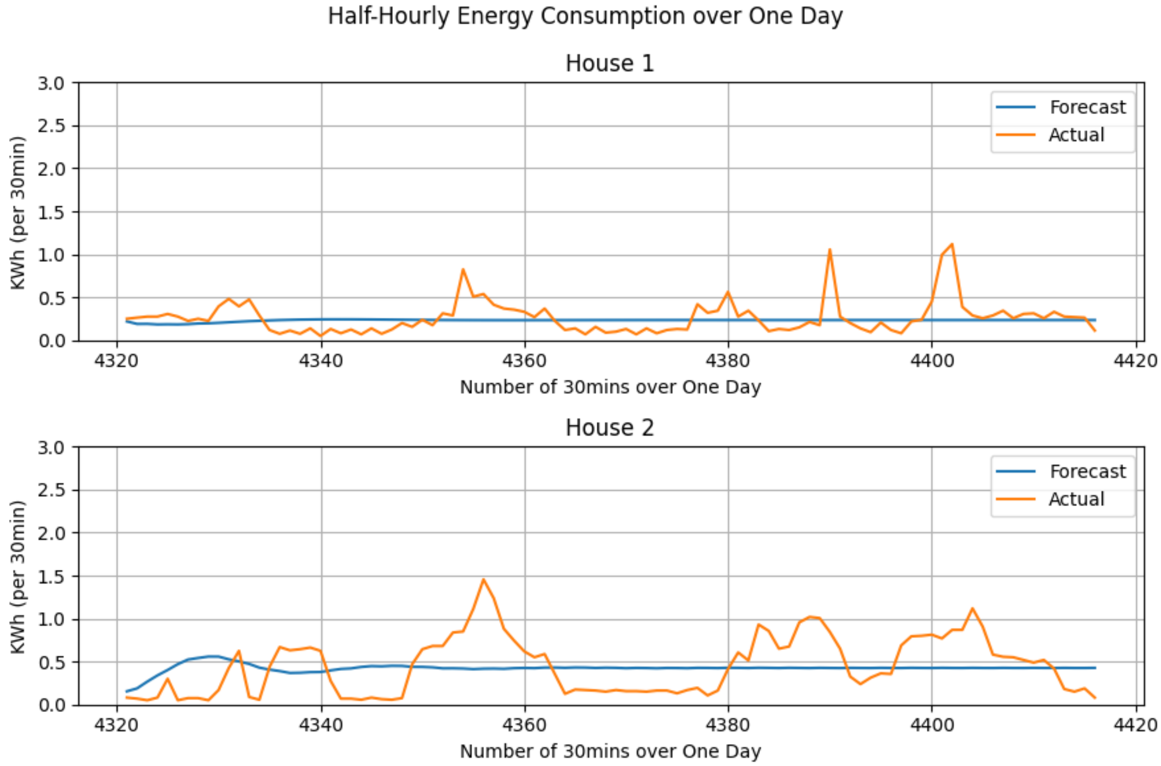
```
    plt.title("House 2")
    plt.xlabel('Number of 30mins over One Day')
    plt.ylabel('KWh (per 30min)')
    plt.ylim(0,3)
    plt.legend()
    plt.grid()
    plt.tight_layout()
    plt.show()
    return mse_h1,mse_h2

#Checking the effects of p,q on MSE and fit time of ARMA Model
start,end,test_period=48*60,48*90,48*2
p=[5,10,30,50]
q=[1,5,10,20]
h1_mse_list=[]
h2_mse_list=[]
h1_fittime_list=[]
h2_fittime_list=[]

for i in range(0,len(p)):
    house1_train,house1_test,house2_train,house2_test=split_data(start,end, te
    h1_fit_time,h2_fit_time,arma_res_h1,arma_res_h2=fit_arma(p[i],q[i],house1_
    h1_mse,h2_mse=forecast_and_measure_performance(test_period,house1_test,hou
    h1_fittime_list.append(h1_fit_time)
    h2_fittime_list.append(h2_fit_time)
    h1_mse_list.append(h1_mse)
    h2_mse_list.append(h2_mse)
```

```
Parameters p=5 q=1
H1 Forecast MSE: 0.039
H2 Forecast MSE: 0.111

<IPython.core.display.Javascript object>
```



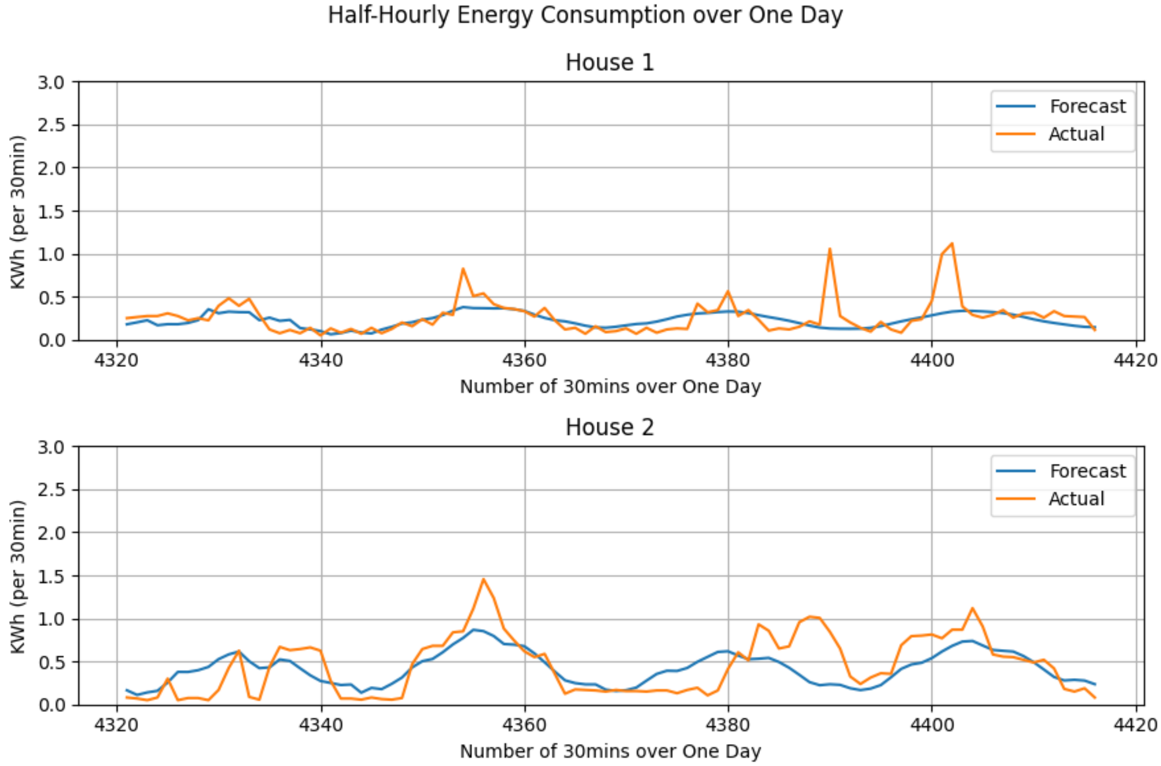
```
Parameters p=10 q=5
H1 Forecast MSE: 0.040
H2 Forecast MSE: 0.116
```

<IPython.core.display.Javascript object>



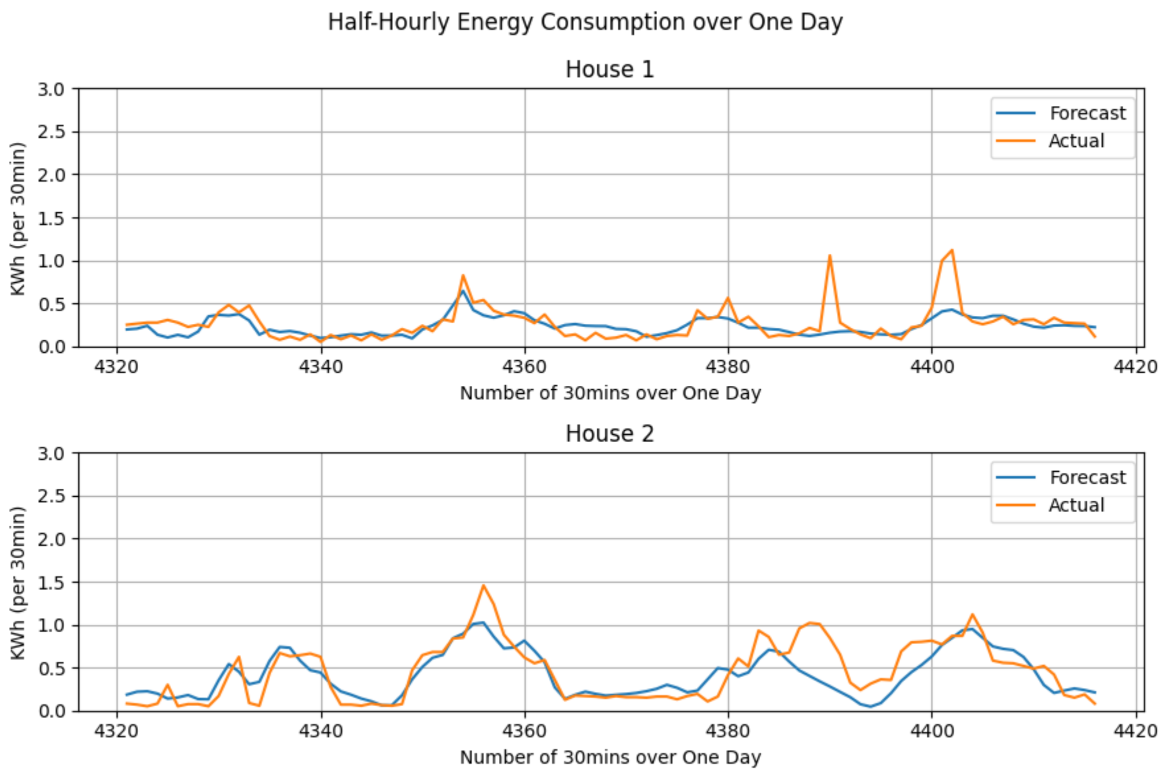

```
Parameters p=30 q=10
H1 Forecast MSE: 0.030
H2 Forecast MSE: 0.065
```

<IPython.core.display.Javascript object>



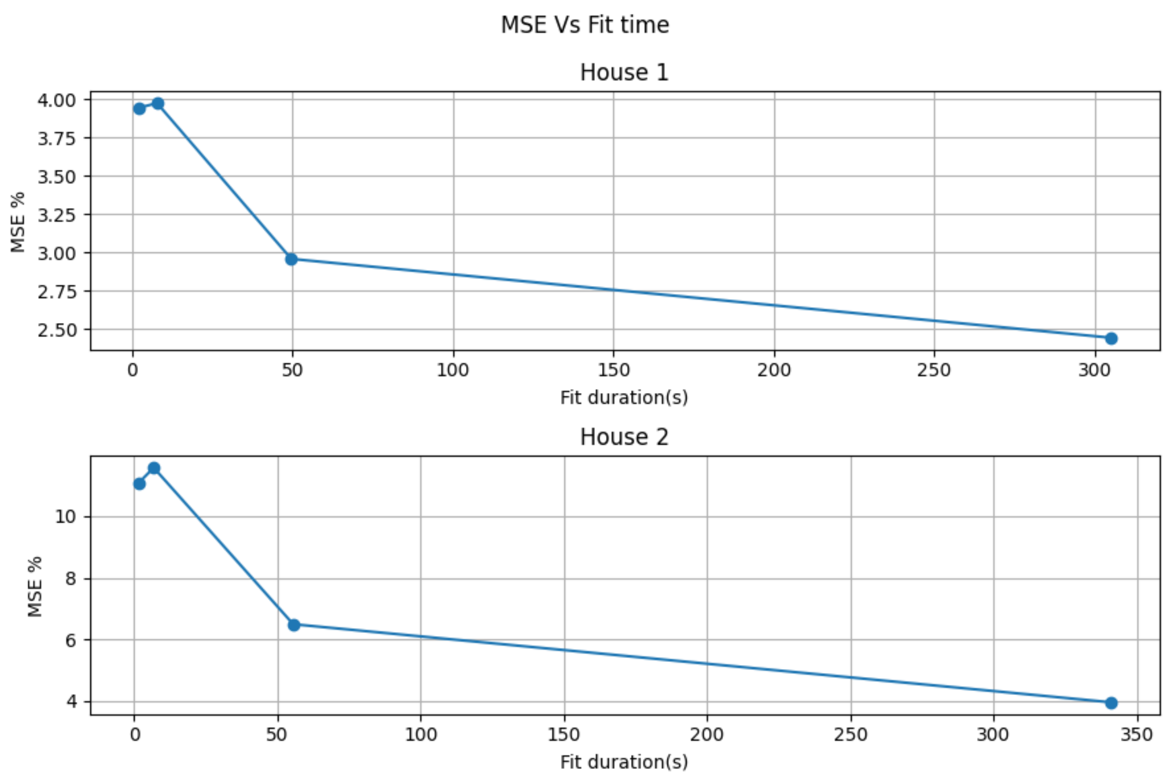

```
Parameters p=50 q=20
H1 Forecast MSE: 0.024
H2 Forecast MSE: 0.040
```

```
<IPython.core.display.Javascript object>
```



In [104]:
```
#Plot the HOuse 1 MSE vs Fit time
plt.figure(figsize=[9,6]).suptitle('MSE Vs Fit time')
plt.subplot(211)
plt.plot(h1_fittime_list,[x*100 for x in h1_mse_list],marker='o')
plt.title("House 1")
plt.xlabel('Fit duration(s)')
plt.ylabel('MSE %')
plt.grid()

#Plot the HOuse 2 MSE vs Fit time
plt.subplot(212)
plt.plot(h2_fittime_list,[x*100 for x in h2_mse_list],marker='o')
plt.title("House 2")
plt.xlabel('Fit duration(s)')
plt.ylabel('MSE %')
plt.grid()
plt.tight_layout()
plt.show()
```

```
<IPython.core.display.Javascript object>
```



- **It is seen that when p and q in increased, the MSE has reduced but at the expense of large fit time of ARMA model.**
- **Therefore we need to find a find best p,q values compromising on the MSE to achieve less fit time**

Question 2.2 [20%] Time Series Estimation using DNN/LSTM

Now, we will use DNNs, specifically LSTM to estimate the power consumption of house 1 and house 2. Specifically, we prepare our data to estimate the next 24 hour period based on the past 24 hours. Note that 24 hours mean 48 data points due to smart meters reporting half-hourly energy usage.

1. Define and train a Keras model that consists of LSTM and Dense layers with a 48 feature input and 48 feature output to forecast demand over the next 24 hour period based on past 24 hours. What type of activation function would you use at the output layer? Why? Try different (appropriate) loss functions and optimisers. You can use "mse" and "adam" as default choices. Choose `batch_size=128` and `epoch=20` as parameters to begin with. You can change these to your liking and are encouraged to experiment.
2. Provide model summary and keep track of training history to provide a plot of loss over epochs. Make predictions for different days and plot your predictions along with actual data. You can evaluate performance by calculating mean-squared error per day or over multiple days in the test set.
3. ***[optional, no points]*** you can try using 1-D CNN layer(s) before LSTM ones as a non-linear filter. Do you observe any improvements?

Useful documents and functions

- [Keras model api documentation (https://www.tensorflow.org/api_docs/python/tf/keras)](https://www.tensorflow.org/api_docs/python/tf/keras), [visualisation (https://www.tensorflow.org/guide/keras/train_and_evaluate#visualizing_loss_and_metrics_during_training)](https://www.tensorflow.org/guide/keras/train_and_evaluate#visualizing_loss_and_metrics_during_training), [sequential model (https://www.tensorflow.org/guide/keras/sequential_model)](https://www.tensorflow.org/guide/keras/sequential_model).
- `fit`, `summary`, `evaluate`, `predict`
- a few links to resources that may be useful:

> https://pandas.pydata.org/pandas-docs/stable/reference/frame.html#attributes-and-underlying-data (https://pandas.pydata.org/pandas-docs/stable/reference/frame.html#attributes-and-underlying-data)
> https://machinelearningmastery.com/time-series-forecasting-methods-in-python-cheat-sheet/ (https://machinelearningmastery.com/time-series-forecasting-methods-in-python-cheat-sheet/)
>
> https://machinelearningmastery.com/how-to-develop-lstm-models-for-multi-step-time-series-forecasting-of-household-power-consumption/ (https://machinelearningmastery.com/how-to-develop-lstm-models-for-multi-step-time-series-forecasting-of-household-power-consumption/)
>
> https://machinelearningmastery.com/multivariate-time-series-forecasting-lstms-keras/ (https://machinelearningmastery.com/multivariate-time-series-forecasting-lstms-keras/)
>
> https://blog.goodaudience.com/introduction-to-1d-convolutional-neural-networks-in-keras-for-time-sequences-3a7ff801a2cf (https://blog.goodaudience.com/introduction-to-1d-convolutional-neural-networks-in-keras-for-time-sequences-3a7ff801a2cf)
> https://github.com/ni79ls/har-keras-cnn (https://github.com/ni79ls/har-keras-cnn)

In [115]:
```
# sliding window function for next 24 hourly estimate
# see, e.g. https://towardsdatascience.com/using-lstms-to-forecast-time-series
# or https://machinelearningmastery.com/reframe-time-series-forecasting-probler
def house_data(inseries):
    window_size = 48+48
    series = inseries
    series_s = inseries.copy()
    for i in range(window_size):
        series = pd.concat([series, series_s.shift(-(i+1))], axis = 1)
    series.dropna(axis=0, inplace=True)
    X = series.iloc[:,0:48]
    yday = series.iloc[:,48:48+48] # next day
    return X, yday


# get the estimate data for house1 and house2
X1, yday1 = house_data(house1[0:8736])
X2, yday2 = house_data(house2[0:8736])

X1.shape, yday1.shape
```

Out[115]:
```
((8640, 48), (8640, 48))
```

In [188]:
```
# split into training and test sets for house 1
X1train, X1test, y1train, y1test = train_test_split(X1, yday1,random_state=132

X1train = np.array(X1train).reshape(X1train.shape[0],X1train.shape[1],1)
X1test = np.array(X1test).reshape(X1test.shape[0],X1test.shape[1],1)

X1train.shape, X1test.shape, y1train.shape, y1test.shape
```

Out[188]:
```
((6480, 48, 1), (2160, 48, 1), (6480, 48), (2160, 48))
```

In [189]:
```
import tensorflow as tf
from tensorflow.keras.layers import LSTM
import time
from sklearn.metrics import mean_squared_error
from sklearn.preprocessing import MinMaxScaler
from sklearn.preprocessing import StandardScaler
```

In [236]:
```
#for make model reproducable
np.random.seed(1320418)
tf.random.set_seed(1320418)

# #Reset Model
# tf.keras.backend.clear_session()
# tf.compat.v1.reset_default_graph()

# train the model
def create_model(train_x,train_y):
    # define parameters
    verbose, epochs, batch_size = 1, 128, 20
    n_inputs, n_features, n_outputs = train_x.shape[1], train_x.shape[2], trai
    # define model
    model = Sequential()
    model.add(LSTM(n_inputs, activation='relu'))
    model.add(Dense(48, activation='relu'))
    model.add(Dense(48, activation='relu'))
    model.add(Dense(n_outputs,activation='relu'))
    model.compile(
              optimizer=tf.keras.optimizers.Adam(),
              loss=tf.keras.losses.MeanSquaredError(),
              metrics=[tf.keras.metrics.MeanSquaredError()])

    # log results
    log_dir = "logs/LSTM/" + datetime.datetime.now().strftime("%Y%m%d-%H%M%S")
    tensorboard_callback = tf.keras.callbacks.TensorBoard(log_dir=log_dir, his

    # fit network
    train_hist=model.fit(train_x, train_y, epochs=epochs, batch_size=batch_siz

    #Plot Train Loss Vs epoch
    plt.figure()
    plt.plot(train_hist.history['loss'])
    plt.xlabel('Epoch number')
    plt.xlabel('MSE')
    plt.title('Training Loss Vs Epoch')
    plt.grid()
    plt.show()
    return model

#Create and train LSTM model
model=create_model(X1train,y1train)
```

```
Epoch 1/128
324/324 [==============================] - 7s 18ms/step - loss: 0.2849 - mean
_squared_error: 0.2849
Epoch 2/128
324/324 [==============================] - 8s 25ms/step - loss: 0.1559 - mean
_squared_error: 0.1559
Epoch 3/128
324/324 [==============================] - 10s 30ms/step - loss: 0.1463 - mea
n_squared_error: 0.1463
Epoch 4/128
324/324 [==============================] - 8s 24ms/step - loss: 0.1428 - mean
_squared_error: 0.1428
Epoch 5/128
324/324 [==============================] - 7s 22ms/step - loss: 0.1398 - mean
_squared_error: 0.1398
Epoch 6/128
324/324 [==============================] - 7s 22ms/step - loss: 0.1391 - mean
_squared_error: 0.1391
Epoch 7/128
324/324 [==============================] - 9s 29ms/step - loss: 0.1387 - mean
_squared_error: 0.1387
Epoch 8/128
324/324 [==============================] - 9s 29ms/step - loss: 0.1383 - mean
_squared_error: 0.1383
Epoch 9/128
324/324 [==============================] - 8s 26ms/step - loss: 0.1381 - mean
_squared_error: 0.1381 1s - loss: 0.1400 - mean_squ - ETA: 0s - loss: 0.1
Epoch 10/128
324/324 [==============================] - 9s 27ms/step - loss: 0.1374 - mean
_squared_error: 0.1374
Epoch 11/128
324/324 [==============================] - 7s 20ms/step - loss: 0.1351 - mean
_squared_error: 0.1351
Epoch 12/128
324/324 [==============================] - 9s 28ms/step - loss: 0.1258 - mean
_squared_error: 0.1258
Epoch 13/128
324/324 [==============================] - 7s 22ms/step - loss: 0.1174 - mean
_squared_error: 0.1174
Epoch 14/128
324/324 [==============================] - 7s 22ms/step - loss: 0.1097 - mean
_squared_error: 0.1097 1s
Epoch 15/128
324/324 [==============================] - 7s 23ms/step - loss: 0.1076 - mean
_squared_error: 0.1076
Epoch 16/128
324/324 [==============================] - 8s 23ms/step - loss: 0.1058 - mean
_squared_error: 0.1058
Epoch 17/128
324/324 [==============================] - 8s 23ms/step - loss: 0.1036 - mean
_squared_error: 0.1036
Epoch 18/128
324/324 [==============================] - 7s 22ms/step - loss: 0.1020 - mean
_squared_error: 0.1020
Epoch 19/128
324/324 [==============================] - 6s 20ms/step - loss: 0.1008 - mean
_squared_error: 0.1008
```

```
Epoch 20/128
324/324 [==============================] - 8s 23ms/step - loss: 0.1000 - mean
_squared_error: 0.1000
Epoch 21/128
324/324 [==============================] - 8s 24ms/step - loss: 0.0987 - mean
_squared_error: 0.0987
Epoch 22/128
324/324 [==============================] - 7s 23ms/step - loss: 0.0959 - mean
_squared_error: 0.0959
Epoch 23/128
324/324 [==============================] - 7s 22ms/step - loss: 0.0957 - mean
_squared_error: 0.0957 1s -
Epoch 24/128
324/324 [==============================] - 7s 21ms/step - loss: 0.0953 - mean
_squared_error: 0.0953
Epoch 25/128
324/324 [==============================] - 8s 26ms/step - loss: 0.0946 - mean
_squared_error: 0.0946
Epoch 26/128
324/324 [==============================] - 8s 26ms/step - loss: 0.0942 - mean
_squared_error: 0.0942
Epoch 27/128
324/324 [==============================] - 7s 23ms/step - loss: 0.0942 - mean
_squared_error: 0.0942
Epoch 28/128
324/324 [==============================] - 7s 22ms/step - loss: 0.0939 - mean
_squared_error: 0.0939
Epoch 29/128
324/324 [==============================] - 7s 23ms/step - loss: 0.0934 - mean
_squared_error: 0.0934
Epoch 30/128
324/324 [==============================] - 8s 23ms/step - loss: 0.0929 - mean
_squared_error: 0.0929
Epoch 31/128
324/324 [==============================] - 7s 22ms/step - loss: 0.0929 - mean
_squared_error: 0.0929
Epoch 32/128
324/324 [==============================] - 7s 23ms/step - loss: 0.0923 - mean
_squared_error: 0.0923
Epoch 33/128
324/324 [==============================] - 8s 24ms/step - loss: 0.0922 - mean
_squared_error: 0.0922
Epoch 34/128
324/324 [==============================] - 9s 29ms/step - loss: 0.0922 - mean
_squared_error: 0.0922
Epoch 35/128
324/324 [==============================] - 7s 23ms/step - loss: 0.0915 - mean
_squared_error: 0.0915
Epoch 36/128
324/324 [==============================] - 8s 26ms/step - loss: 0.0914 - mean
_squared_error: 0.0914 1s -
Epoch 37/128
324/324 [==============================] - 8s 26ms/step - loss: 0.0910 - mean
_squared_error: 0.0910 1s - l
Epoch 38/128
324/324 [==============================] - 7s 22ms/step - loss: 0.0917 - mean
_squared_error: 0.0917
```

```
Epoch 39/128
324/324 [==============================] - 7s 22ms/step - loss: 0.0908 - mean
_squared_error: 0.0908
Epoch 40/128
324/324 [==============================] - 7s 20ms/step - loss: 0.0903 - mean
_squared_error: 0.0903
Epoch 41/128
324/324 [==============================] - 10s 30ms/step - loss: 0.0899 - mea
n_squared_error: 0.0899
Epoch 42/128
324/324 [==============================] - 7s 21ms/step - loss: 0.0896 - mean
_squared_error: 0.0896
Epoch 43/128
324/324 [==============================] - 7s 21ms/step - loss: 0.0901 - mean
_squared_error: 0.0901
Epoch 44/128
324/324 [==============================] - 7s 22ms/step - loss: 0.0894 - mean
_squared_error: 0.0894
Epoch 45/128
324/324 [==============================] - 8s 23ms/step - loss: 0.0887 - mean
_squared_error: 0.0887
Epoch 46/128
324/324 [==============================] - 7s 22ms/step - loss: 0.0892 - mean
_squared_error: 0.0892
Epoch 47/128
324/324 [==============================] - 7s 23ms/step - loss: 0.0884 - mean
_squared_error: 0.0884 1s - loss: 0.088
Epoch 48/128
324/324 [==============================] - 7s 21ms/step - loss: 0.0892 - mean
_squared_error: 0.0892
Epoch 49/128
324/324 [==============================] - 7s 23ms/step - loss: 0.0887 - mean
_squared_error: 0.0887
Epoch 50/128
324/324 [==============================] - 8s 24ms/step - loss: 0.0875 - mean
_squared_error: 0.0875
Epoch 51/128
324/324 [==============================] - 7s 21ms/step - loss: 0.0881 - mean
_squared_error: 0.0881
Epoch 52/128
324/324 [==============================] - 8s 24ms/step - loss: 0.0860 - mean
_squared_error: 0.0860
Epoch 53/128
324/324 [==============================] - 7s 21ms/step - loss: 0.0868 - mean
_squared_error: 0.0868
Epoch 54/128
324/324 [==============================] - 8s 24ms/step - loss: 0.0863 - mean
_squared_error: 0.0863
Epoch 55/128
324/324 [==============================] - 7s 21ms/step - loss: 0.0841 - mean
_squared_error: 0.0841
Epoch 56/128
324/324 [==============================] - 7s 22ms/step - loss: 0.0819 - mean
_squared_error: 0.0819
Epoch 57/128
324/324 [==============================] - 9s 29ms/step - loss: 0.0806 - mean
_squared_error: 0.0806
```

```
Epoch 58/128
324/324 [==============================] - 8s 25ms/step - loss: 0.0797 - mean
_squared_error: 0.0797 0s - loss: 0.0797 - mean_squar
Epoch 59/128
324/324 [==============================] - 7s 22ms/step - loss: 0.0801 - mean
_squared_error: 0.0801 0s - loss: 0
Epoch 60/128
324/324 [==============================] - 7s 22ms/step - loss: 0.0785 - mean
_squared_error: 0.0785
Epoch 61/128
324/324 [==============================] - 7s 22ms/step - loss: 0.0788 - mean
_squared_error: 0.0788 0s - loss: 0.0788 - mean_squared_error: 0.
Epoch 62/128
324/324 [==============================] - 6s 20ms/step - loss: 0.0799 - mean
_squared_error: 0.0799
Epoch 63/128
324/324 [==============================] - 8s 24ms/step - loss: 0.0771 - mean
_squared_error: 0.0771
Epoch 64/128
324/324 [==============================] - 7s 21ms/step - loss: 0.0765 - mean
_squared_error: 0.0765
Epoch 65/128
324/324 [==============================] - 7s 23ms/step - loss: 0.0764 - mean
_squared_error: 0.0764
Epoch 66/128
324/324 [==============================] - 7s 22ms/step - loss: 0.0761 - mean
_squared_error: 0.0761
Epoch 67/128
324/324 [==============================] - 8s 23ms/step - loss: 0.0751 - mean
_squared_error: 0.0751
Epoch 68/128
324/324 [==============================] - 6s 19ms/step - loss: 0.0760 - mean
_squared_error: 0.0760
Epoch 69/128
324/324 [==============================] - 7s 22ms/step - loss: 0.0748 - mean
_squared_error: 0.0748
Epoch 70/128
324/324 [==============================] - 7s 21ms/step - loss: 0.0742 - mean
_squared_error: 0.0742
Epoch 71/128
324/324 [==============================] - 7s 21ms/step - loss: 0.0739 - mean
_squared_error: 0.0739
Epoch 72/128
```

```
324/324 [==============================] - 8s 24ms/step - loss: 0.0737 - mean
_squared_error: 0.0737
Epoch 73/128
324/324 [==============================] - 7s 20ms/step - loss: 0.0733 - mean
_squared_error: 0.0733
Epoch 74/128
324/324 [==============================] - 8s 25ms/step - loss: 0.0700 - mean
_squared_error: 0.0700
Epoch 75/128
324/324 [==============================] - 8s 24ms/step - loss: 0.0710 - mean
_squared_error: 0.0710 1s - loss: 0.070
Epoch 76/128
324/324 [==============================] - 8s 25ms/step - loss: 0.0685 - mean
_squared_error: 0.0685
Epoch 77/128
324/324 [==============================] - 7s 21ms/step - loss: 0.0676 - mean
_squared_error: 0.0676
Epoch 78/128
324/324 [==============================] - 7s 23ms/step - loss: 0.0675 - mean
_squared_error: 0.0675
Epoch 79/128
324/324 [==============================] - 7s 22ms/step - loss: 0.0681 - mean
_squared_error: 0.0681
Epoch 80/128
324/324 [==============================] - 8s 25ms/step - loss: 0.0666 - mean
_squared_error: 0.0666
Epoch 81/128
324/324 [==============================] - 6s 20ms/step - loss: 0.0672 - mean
_squared_error: 0.0672
Epoch 82/128
324/324 [==============================] - 7s 21ms/step - loss: 0.0660 - mean
_squared_error: 0.0660
Epoch 83/128
324/324 [==============================] - 8s 25ms/step - loss: 0.0665 - mean
_squared_error: 0.0665
Epoch 84/128
324/324 [==============================] - 7s 22ms/step - loss: 0.0675 - mean
_squared_error: 0.0675
Epoch 85/128
324/324 [==============================] - 7s 22ms/step - loss: 0.0644 - mean
_squared_error: 0.0644
Epoch 86/128
324/324 [==============================] - 7s 21ms/step - loss: 0.0652 - mean
_squared_error: 0.0652
Epoch 87/128
324/324 [==============================] - 8s 23ms/step - loss: 0.0654 - mean
_squared_error: 0.0654
Epoch 88/128
324/324 [==============================] - 7s 21ms/step - loss: 0.0647 - mean
_squared_error: 0.0647
Epoch 89/128
324/324 [==============================] - 8s 25ms/step - loss: 0.0640 - mean
_squared_error: 0.0640
Epoch 90/128
324/324 [==============================] - 8s 24ms/step - loss: 0.0640 - mean
_squared_error: 0.0640
Epoch 91/128
```

```
324/324 [==============================] - 8s 24ms/step - loss: 0.0629 - mean
_squared_error: 0.0629
Epoch 92/128
324/324 [==============================] - 7s 21ms/step - loss: 0.0607 - mean
_squared_error: 0.0607
Epoch 93/128
324/324 [==============================] - 7s 22ms/step - loss: 0.0609 - mean
_squared_error: 0.0609
Epoch 94/128
324/324 [==============================] - 7s 22ms/step - loss: 0.0638 - mean
_squared_error: 0.0638
Epoch 95/128
324/324 [==============================] - 7s 23ms/step - loss: 0.0604 - mean
_squared_error: 0.0604
Epoch 96/128
324/324 [==============================] - 7s 23ms/step - loss: 0.0576 - mean
_squared_error: 0.0576
Epoch 97/128
324/324 [==============================] - 7s 21ms/step - loss: 0.0575 - mean
_squared_error: 0.0575
Epoch 98/128
324/324 [==============================] - 7s 23ms/step - loss: 0.0574 - mean
_squared_error: 0.0574
Epoch 99/128
324/324 [==============================] - 7s 23ms/step - loss: 0.0567 - mean
_squared_error: 0.0567
Epoch 100/128
324/324 [==============================] - 7s 22ms/step - loss: 0.0564 - mean
_squared_error: 0.0564
Epoch 101/128
324/324 [==============================] - 6s 19ms/step - loss: 0.0576 - mean
_squared_error: 0.0576
Epoch 102/128
324/324 [==============================] - 8s 25ms/step - loss: 0.0553 - mean
_squared_error: 0.0553
Epoch 103/128
324/324 [==============================] - 7s 22ms/step - loss: 0.0553 - mean
_squared_error: 0.0553
Epoch 104/128
324/324 [==============================] - 7s 22ms/step - loss: 0.0544 - mean
_squared_error: 0.0544
Epoch 105/128
324/324 [==============================] - 7s 21ms/step - loss: 0.0546 - mean
_squared_error: 0.0546
Epoch 106/128
324/324 [==============================] - 8s 23ms/step - loss: 0.0544 - mean
_squared_error: 0.0544
Epoch 107/128
324/324 [==============================] - 9s 27ms/step - loss: 0.0554 - mean
_squared_error: 0.0554
Epoch 108/128
324/324 [==============================] - 7s 22ms/step - loss: 0.0558 - mean
_squared_error: 0.0558
Epoch 109/128
324/324 [==============================] - 7s 22ms/step - loss: 0.0532 - mean
_squared_error: 0.0532
Epoch 110/128
```

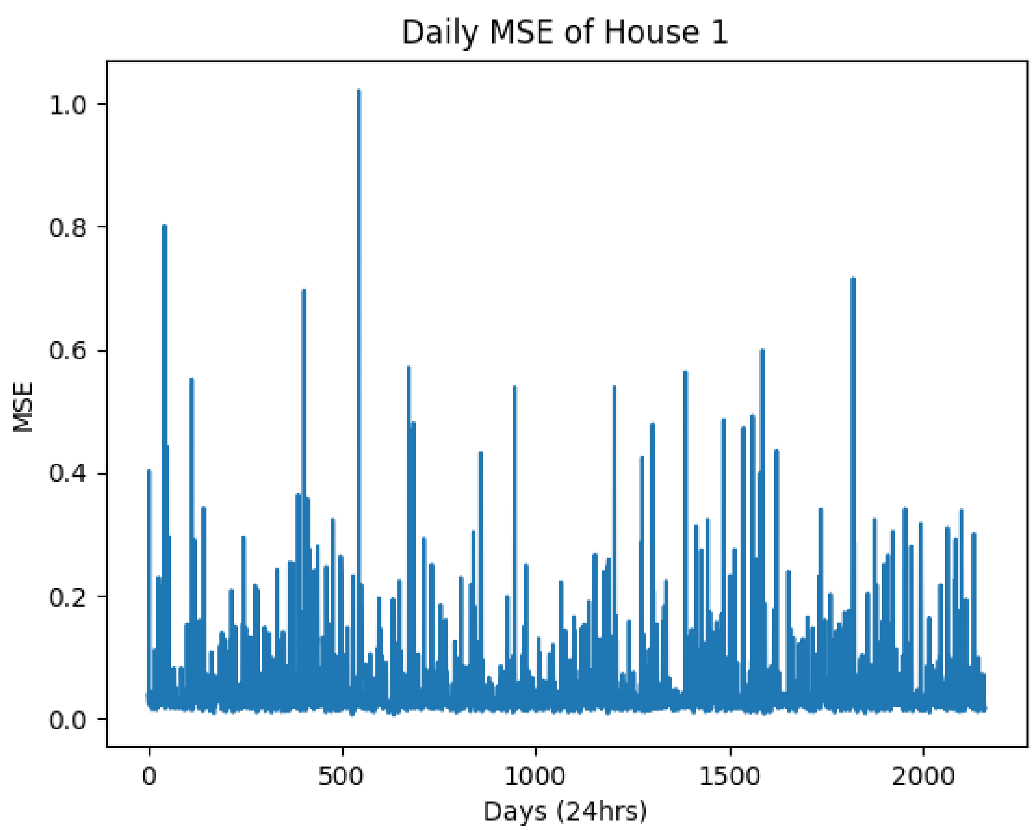
```
324/324 [==============================] - 7s 23ms/step - loss: 0.0539 - mean
_squared_error: 0.0539
Epoch 111/128
324/324 [==============================] - 7s 22ms/step - loss: 0.0531 - mean
_squared_error: 0.0531
Epoch 112/128
324/324 [==============================] - 7s 21ms/step - loss: 0.0523 - mean
_squared_error: 0.0523
Epoch 113/128
324/324 [==============================] - 8s 24ms/step - loss: 0.0533 - mean
_squared_error: 0.0533
Epoch 114/128
324/324 [==============================] - 6s 20ms/step - loss: 0.0521 - mean
_squared_error: 0.0521
Epoch 115/128
324/324 [==============================] - 7s 20ms/step - loss: 0.0517 - mean
_squared_error: 0.0517
Epoch 116/128
324/324 [==============================] - 8s 24ms/step - loss: 0.0513 - mean
_squared_error: 0.0513
Epoch 117/128
324/324 [==============================] - 7s 22ms/step - loss: 0.0510 - mean
_squared_error: 0.0510
Epoch 118/128
324/324 [==============================] - 7s 21ms/step - loss: 0.0498 - mean
_squared_error: 0.0498
Epoch 119/128
324/324 [==============================] - 7s 22ms/step - loss: 0.0506 - mean
_squared_error: 0.0506
Epoch 120/128
324/324 [==============================] - 10s 29ms/step - loss: 0.0497 - mea
n_squared_error: 0.0497
Epoch 121/128
324/324 [==============================] - 8s 24ms/step - loss: 0.0493 - mean
_squared_error: 0.0493
Epoch 122/128
324/324 [==============================] - 10s 30ms/step - loss: 0.0495 - mea
n_squared_error: 0.0495
Epoch 123/128
324/324 [==============================] - 7s 22ms/step - loss: 0.0492 - mean
_squared_error: 0.0492
Epoch 124/128
324/324 [==============================] - 8s 24ms/step - loss: 0.0488 - mean
_squared_error: 0.0488
Epoch 125/128
324/324 [==============================] - 7s 21ms/step - loss: 0.0478 - mean
_squared_error: 0.0478
Epoch 126/128
324/324 [==============================] - 7s 23ms/step - loss: 0.0481 - mean
_squared_error: 0.0481
Epoch 127/128
324/324 [==============================] - 7s 21ms/step - loss: 0.0471 - mean
_squared_error: 0.0471
Epoch 128/128
324/324 [==============================] - 7s 22ms/step - loss: 0.0475 - mean
_squared_error: 0.0475
```

```
<IPython.core.display.Javascript object>
```

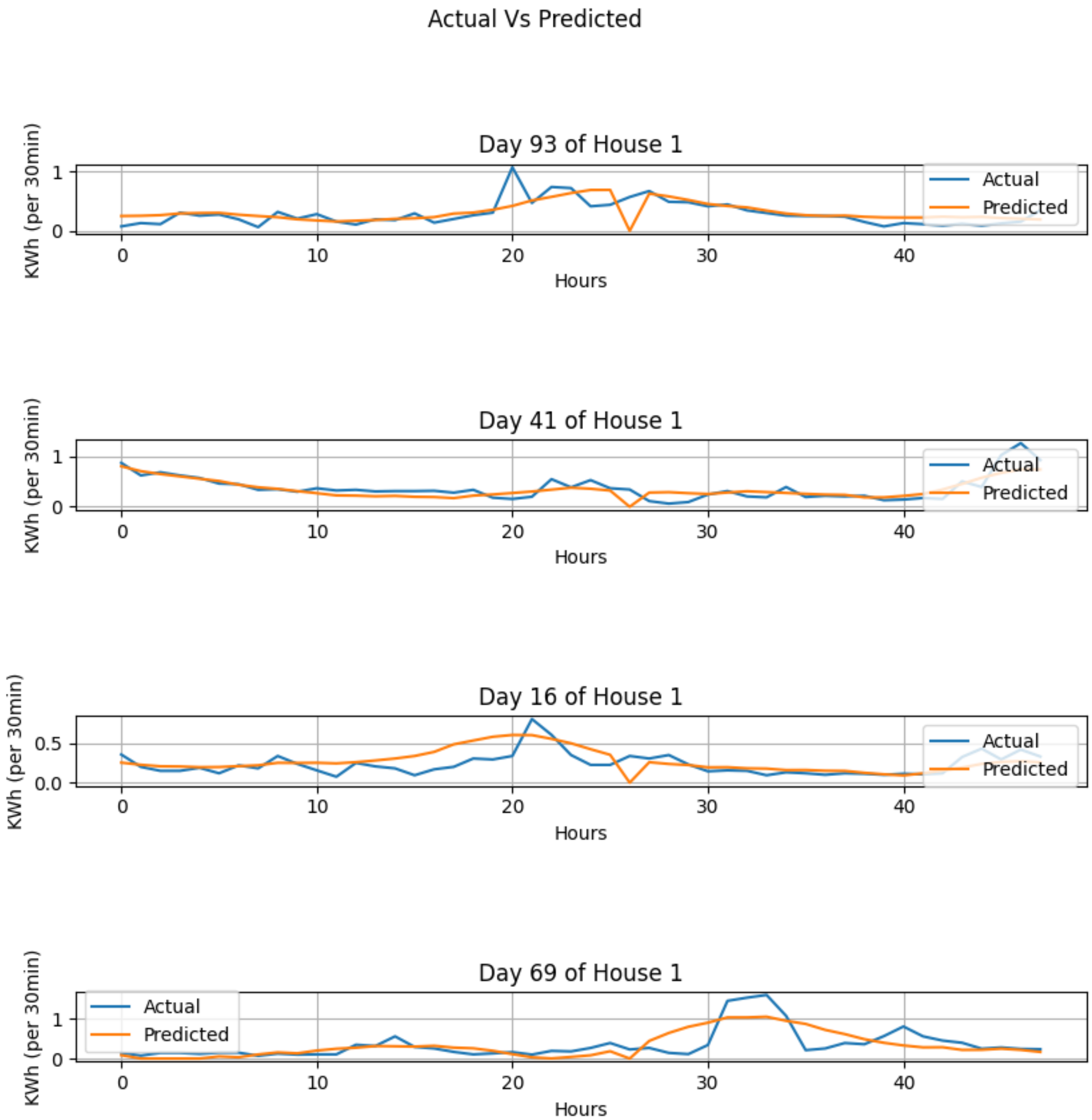
In [232]:
```
print(model.summary())

#Evaluate the predicted data from LSTM using MSE
def evaluate_forecasts(actual, predicted):
    scores = list()
    s = 0
    # calculate an MSE score for each day and overall MSE
    for i in range(actual.shape[0]):
        # calculate mse
        mse = mean_squared_error(actual[i], predicted[i])
        s+=mse
        scores.append(mse)
    score = s/(actual.shape[0])
    return score, scores


def plot_actual_predicted(actual, predicted):
    random_days=np.random.randint(100,size=(4))
    plt.figure().suptitle("Actual Vs Predicted")
    for i,day in enumerate(random_days):
        plt.subplot(4,1,i+1)
        plt.plot(actual[day],label="Actual")
        plt.plot(predicted[day],label="Predicted")
        plt.title(f"Day {day} of House 1")
        plt.xlabel('Hours')
        plt.ylabel('KWh (per 30min)')
        plt.legend()
        plt.grid()
    plt.tight_layout()
    plt.show()

#Predict from test data
y1pred=model.predict(X1test)

#get MSEs
mse_avg,mse_daily=evaluate_forecasts(np.array(y1test), y1pred)
print(f"Avg MSE per day =  {round(mse_avg,4)}")

#Plot prediction MSE daily
plt.figure()
plt.plot(mse_daily)
plt.title("Daily MSE of House 1")
plt.xlabel('Days (24hrs)')
plt.ylabel('MSE')
plt.show()

#Plot Actual Vs Predicted
plot_actual_predicted(np.array(y1test), y1pred)
```

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                Output Shape              Param #
=================================================================
lstm_1 (LSTM)               (None, 48)                9600

dense_2 (Dense)             (None, 48)                2352

dense_3 (Dense)             (None, 48)                2352

=================================================================
Total params: 14,304
Trainable params: 14,304
Non-trainable params: 0
_________________________________________________________________
None
Avg MSE per day =  0.052
```

```
<IPython.core.display.Javascript object>
```




```
<IPython.core.display.Javascript object>
```



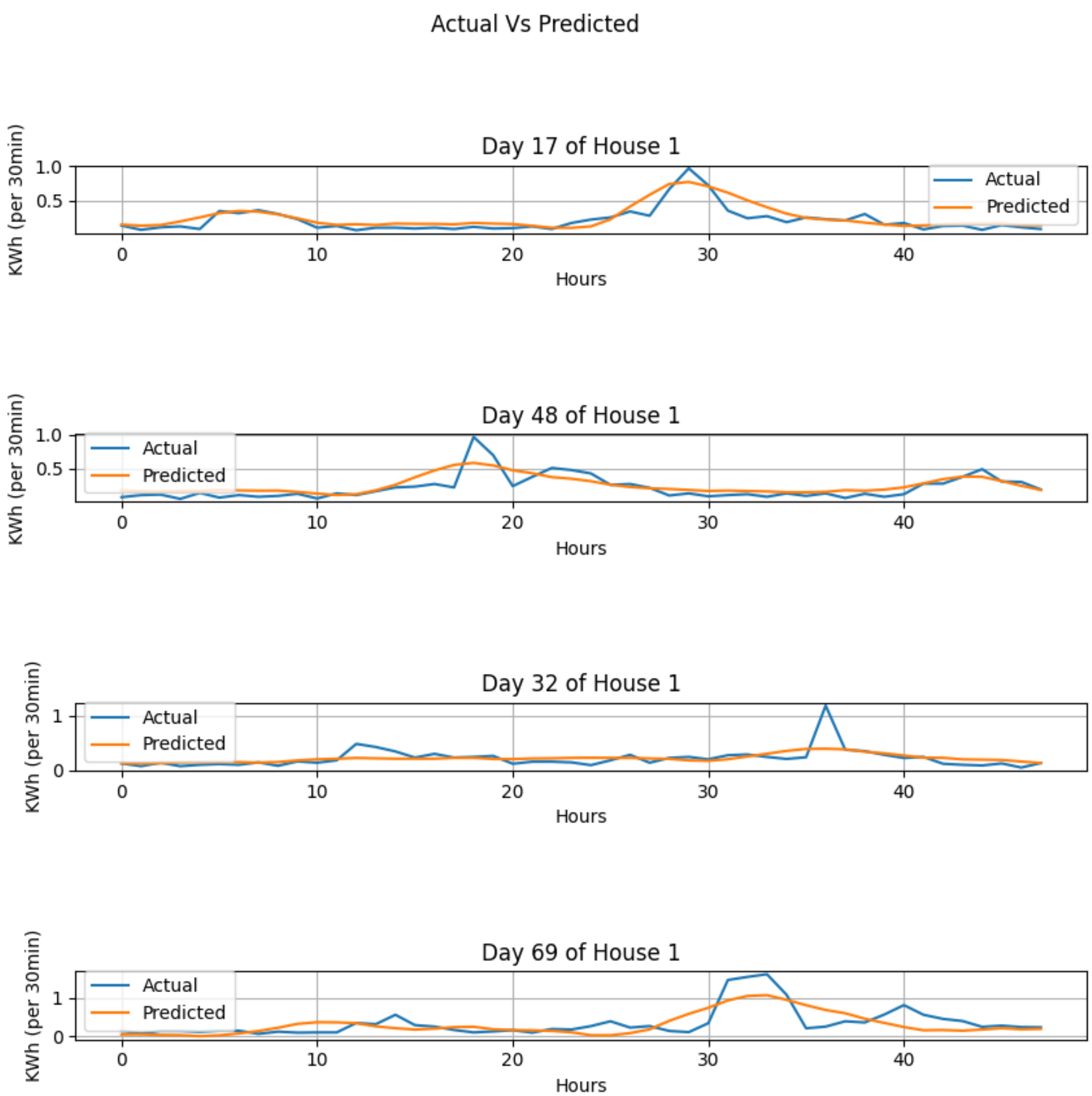
- **Since we are predicting a numerical variable the activation function at the output layer should be either none or a linear activation function. In this case i chose "ReLu" as the output should be positive**

In [227]:
```
from tensorflow.keras.layers import Conv1D

#for make model reproducable
np.random.seed(1320418)
tf.random.set_seed(1320418)

# #Reset Model
# tf.keras.backend.clear_session()
# tf.compat.v1.reset_default_graph()

# train the model
def create_model_CNN_LSTM(train_x,train_y):
    # define parameters
    verbose, epochs, batch_size = 1, 128, 20
    n_inputs, n_features, n_outputs = train_x.shape[1], train_x.shape[2], trai
    # define model
    model = Sequential()
    model.add(Conv1D(filters=64, kernel_size=3, activation='relu', input_shape
    model.add(LSTM(48, activation='relu'))
    model.add(Dense(48, activation='relu'))
    model.add(Dense(n_outputs,activation='relu'))
    model.compile(
              optimizer=tf.keras.optimizers.Adam(),
              loss=tf.keras.losses.MeanSquaredError(),
              metrics=[tf.keras.metrics.MeanSquaredError()])

    # log results
    log_dir = "logs/LSTM/" + datetime.datetime.now().strftime("%Y%m%d-%H%M%S")
    tensorboard_callback = tf.keras.callbacks.TensorBoard(log_dir=log_dir, his

    # fit network
    model.fit(train_x, train_y, epochs=epochs, batch_size=batch_size, verbose=
    return model


#Create and train LSTM model
model_CNN_LSTM=create_model_CNN_LSTM(X1train,y1train)
```

```
Epoch 1/128
324/324 [==============================] - 29s 84ms/step - loss: 0.1540 - m
ean_squared_error: 0.1540
Epoch 2/128
324/324 [==============================] - 18s 54ms/step - loss: 0.1195 - m
ean_squared_error: 0.1195
Epoch 3/128
324/324 [==============================] - 21s 64ms/step - loss: 0.1036 - m
ean_squared_error: 0.1036
Epoch 4/128
324/324 [==============================] - 22s 68ms/step - loss: 0.0953 - m
ean_squared_error: 0.0953
Epoch 5/128
324/324 [==============================] - 19s 60ms/step - loss: 0.0898 - m
ean_squared_error: 0.0898
Epoch 6/128
324/324 [==============================] - 22s 67ms/step - loss: 0.0867 - m
ean_squared_error: 0.0867
Epoch 7/128
```

In [237]:
```python
#for make model reproducable
np.random.seed(1320418)
tf.random.set_seed(1320418)

#Predict from test data
y1pred_CNN_LSTM=model_CNN_LSTM.predict(X1test)

#get MSEs
mse_avg,mse_daily=evaluate_forecasts(np.array(y1test), y1pred_CNN_LSTM)
print(f"Avg MSE per day =  {round(mse_avg,4)}")

#Plot Actual Vs Predicted
plot_actual_predicted(np.array(y1test), y1pred_CNN_LSTM)
```

```
Avg MSE per day =  0.0312

<IPython.core.display.Javascript object>
```



Yes, the avg MSE has improved by using a 1D Convolution layer before LSTM

Reinforcement Learning Overview

Reinforcement Learning (RL) has been making headlines the last few years and there are good reasons for it! Extensions of the methods you will see in this workshop have been used to make computers learn [how to play Atari games by themselves (https://openai.com/blog/openai-baselines-dqn/)](https://openai.com/blog/openai-baselines-dqn/) [(see also this) (https://deepmind.com/research/publications/playing-atari-deep-reinforcement-learning/)](https://deepmind.com/research/publications/playing-atari-deep-reinforcement-learning/). The recent advances in [solving most challenging board games (https://deepmind.com/research/alphago/)](https://deepmind.com/research/alphago/) have been very impressive. Until even ten years ago, many people believed that computers would never learn how to play the game "Go" due to its combinatorial complexity. Today, AlphaGo variants are the first computer program to defeat a professional human Go player, the first program to defeat a Go world champion, and arguably the strongest Go player in history. It is a testament to the power of RL that [AlphaGo Zero (https://deepmind.com/blog/alphago-zero-learning-scratch/)](https://deepmind.com/blog/alphago-zero-learning-scratch/) learns to play simply by playing games against itself, starting from completely random play.

The theoretical foundations of RL have been known for a long while as presented in lectures. Today's successes basically come from well-engineered or designed software that runs on powerful computing systems. Multiple heuristic algorithms and designs verified through extensive experimentation seem to be the key methodology. Despite introducing state-of-the-art concepts, tools, and implementations, this workshop provides only an initial starting point to the world of modern RL.

> Learning more on RL requires good coding skills and a powerful computer (often with a good CUDA-supporting graphics card) or a cloud computing account with one of the major providers. Computer and board games have been the natural playground of modern RL. However, [application of RL to engineering disciplines (https://blog.insightdatascience.com/using-reinforcement-learning-to-design-a-better-rocket-engine-4dfd1770497a)](https://blog.insightdatascience.com/using-reinforcement-learning-to-design-a-better-rocket-engine-4dfd1770497a) remains an under-explored and very exciting domain!

Section 3: RL with Multi-armed Bandits

In a **k-armed (multi-armed) bandit** problem, a decision-making agent repeatedly chooses one of k different actions. Each action can be interpreted as pulling one of the k different levers. After each choice, the agent receives a reward obtained from a probability distribution that depends on the selected action. The objective is to maximise the expected total reward over a time horizon, for example, over $1,000$ action selections, or time steps. Multi-armed bandits have [a variety of important applications (https://medium.com/@CornellResearch/whats-behind-your-navigation-app-79d2754e6878)](https://medium.com/@CornellResearch/whats-behind-your-navigation-app-79d2754e6878) ranging from clinical trials and routing (including navigation) to recommender systems.

As a special case of ***reinforcement learning***, the [multi-armed bandit (https://en.wikipedia.org/wiki/Multi-armed_bandit)](https://en.wikipedia.org/wiki/Multi-armed_bandit) problem has actually only a single state. The agent still has to learn the environment represented by the underlying probability distributions and rewards. The problem provides a nice introduction to ***reinforcement learning*** and an opportunity to explore the fundamental **exploration versus exploitation** trade-offs involved.

Hint: Example implementations online (randomly selected, not guaranteed to be correct):

- https://www.analyticsvidhya.com/blog/2018/09/reinforcement-multi-armed-bandit-scratch-python/ (https://www.analyticsvidhya.com/blog/2018/09/reinforcement-multi-armed-bandit-scratch-python/)
- https://peterroelants.github.io/posts/multi-armed-bandit-implementation/ (https://peterroelants.github.io/posts/multi-armed-bandit-implementation/)
- https://lilianweng.github.io/lil-log/2018/01/23/the-multi-armed-bandit-problem-and-its-solutions.html (https://lilianweng.github.io/lil-log/2018/01/23/the-multi-armed-bandit-problem-and-its-solutions.html)
- https://towardsdatascience.com/comparing-multi-armed-bandit-algorithms-on-marketing-use-cases-8de62a851831 (https://towardsdatascience.com/comparing-multi-armed-bandit-algorithms-on-marketing-use-cases-8de62a851831)

In [3]:
```
%matplotlib notebook
import pandas as pd
import numpy as np
import time
import random
import matplotlib.pyplot as plt
import matplotlib
from collections import deque
from tensorflow.keras.optimizers import Adam
```

10-armed bandit data set

Let's first (create or) ***load*** a random 10-armed data set that approximately matches the description in Section 2.3 of Sutton and Barto RL book. (http://incompleteideas.net/book/the-book-2nd.html)

In [7]:
```
def gen_data(num_bandits=10, T=2000, filename='10armdata'):
    ## function generates a synthetic data set with given parameters
    ## and saves the result to files folder under the given name

    # init data array
    tenarm_data = np.zeros((T,num_bandits))

    # random mean awards
    mean_rewards = np.random.normal(size=num_bandits)
#     tenarm_data[0,:]=np.random.normal(mean_rewards,1,num_bandits)
    print(tenarm_data)
#     print(np.random.normal(mean_rewards,1,num_bandits))
    for t in range(T):
        tenarm_data[t,:]=np.random.normal(mean_rewards,1,num_bandits)

    np.save('./files/'+filename, tenarm_data)


# No need to set the random seed again if you did it in above cells.

# gen_data()
#tenarm_data.shape
#tenarm_data[0:10,:]

# use generated data
tenarm_datal = np.load('./files/10armdata.npy')
tenarm_datal.shape
```

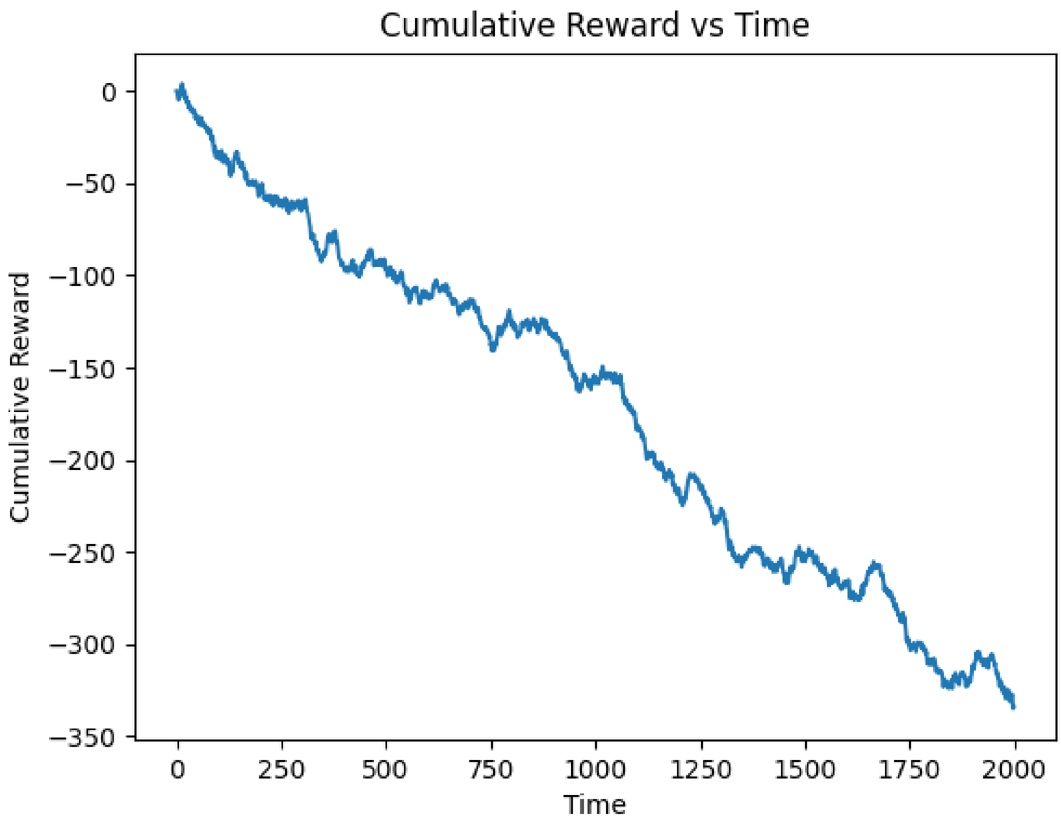
Out[7]:
```
(2000, 10)
```

Multi-armed Bandit Algorithms

We now implement a simple random strategy for selecting actions. The results are also random as expected. This can be considered as pure **exploration** since the algorithm keeps randomly choosing actions. However, note that we do not make proper use of the randomly collected observations yet.

In [4]:
```
random.seed(1320418)
def bandit_random(data=tenarm_datal):
    # random selection bandit algorithm

    num_bandits = tenarm_datal.shape[1]
    T = tenarm_datal.shape[0]
    # init storage arrays
    selections = np.zeros(T) # sequence of lever selections
    step_rewards = np.zeros(T) # sequence of step selections
    cum_rewards = np.zeros(T) # sequence of cumulative rewards
    # main loop
    for t in range(T):
        sel = random.randrange(num_bandits)
        selections[t] = sel
        step_rewards[t] = data[t,sel]
        if t>0:
            cum_rewards[t] = step_rewards[t]+cum_rewards[t-1]
        else:
            cum_rewards[t] = step_rewards[t]

    total_reward = cum_rewards[-1] # the last one is total reward!

    return (selections, step_rewards, cum_rewards, total_reward)



(selections, step_rewards, cum_rewards, total_reward) = bandit_random()

print(total_reward)
plt.figure()
plt.title('Cumulative Reward vs Time')
plt.xlabel('Time')
plt.ylabel('Cumulative Reward')
plt.plot(cum_rewards)
plt.show()
```

```
-333.8348661464376

<IPython.core.display.Javascript object>
```

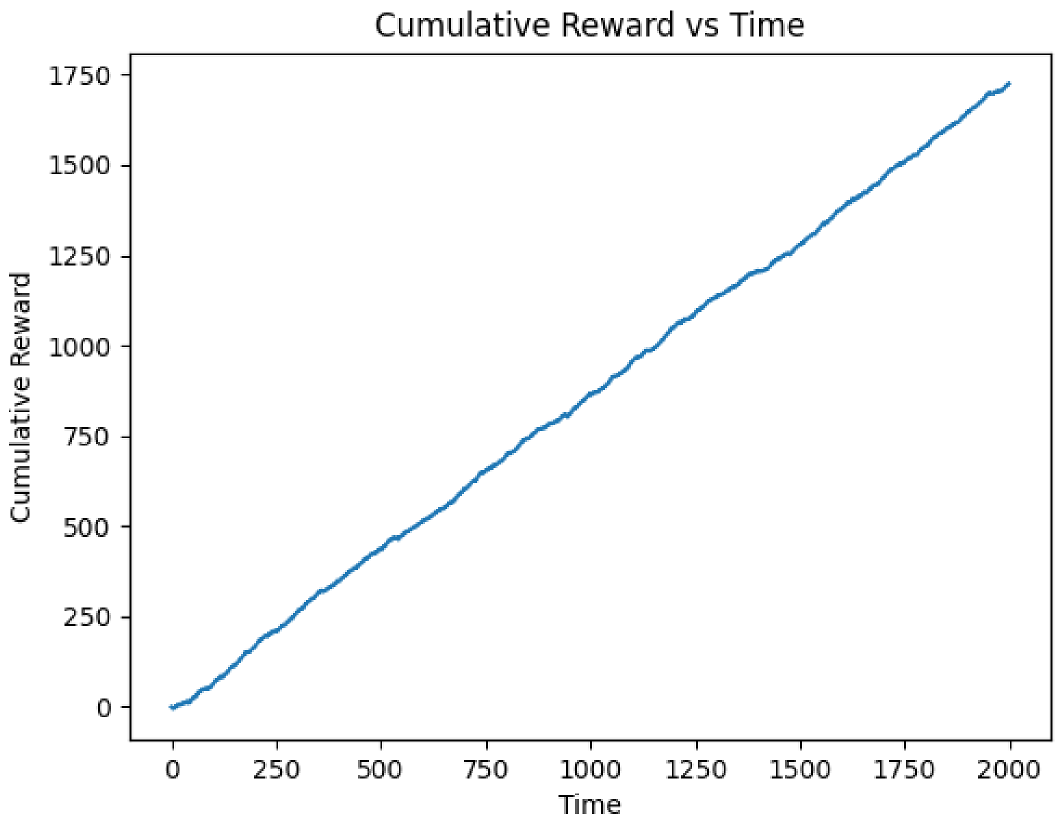


Let us consider next a more meaningful strategy, known as ε**-greedy algorithm**. The idea is to explore with a pre-determined, fixed probability $\varepsilon < 1$ and exploit, i.e. get the maximum reward given the current knowledge, with probability $1 - \varepsilon$. The observations are now used to estimate the values of actions by averaging. This well-known algorithm is discussed in Section 2.7 of [Sutton and Barto book (http://incompleteideas.net/book/the-book-2nd.html)](http://incompleteideas.net/book/the-book-2nd.html) and described below:

We provide a rudimentary implementation below as a single run.

In [34]:
```
def bandit_epsgreedy(data=tenarm_datal, eps=0.1):
    # epsilon-greedy bandit algorithm

    # parameters
    num_bandits = data.shape[1]
    T = data.shape[0]

    # init storage arrays
    Q = np.zeros(num_bandits)
    N = np.zeros(num_bandits)
    selections = np.zeros(T) # sequence of lever selections
    step_rewards = np.zeros(T) # sequence of step selections
    cum_rewards = np.zeros(T) # sequence of cumulative rewards
    # main loop
    for t in range(T):

        # pull lever
        if np.random.rand() < eps:
            # make a random selection
            sel = random.randrange(num_bandits)
        else:
            # choose the best expected reward
            sel = np.argmax(Q)

        # update nbr of selections made
        N[sel] = N[sel] + 1
        # update mean reward estimate
        Q[sel] = Q[sel] + (1/N[sel])*(data[t,sel]- Q[sel])

        # store values
        selections[t] = sel
        step_rewards[t] = data[t,sel]
        if t>0:
            cum_rewards[t] = step_rewards[t]+cum_rewards[t-1]
        else:
            cum_rewards[t] = step_rewards[t]

    total_reward = cum_rewards[-1] # the last one is total reward!

    return (selections, step_rewards, cum_rewards, total_reward)



(selections, step_rewards, cum_rewards, total_reward) = bandit_epsgreedy(eps=0

print(total_reward)
plt.figure()
plt.title('Cumulative Reward vs Time')
plt.xlabel('Time')
plt.ylabel('Cumulative Reward')
plt.plot(cum_rewards)
plt.show()
```

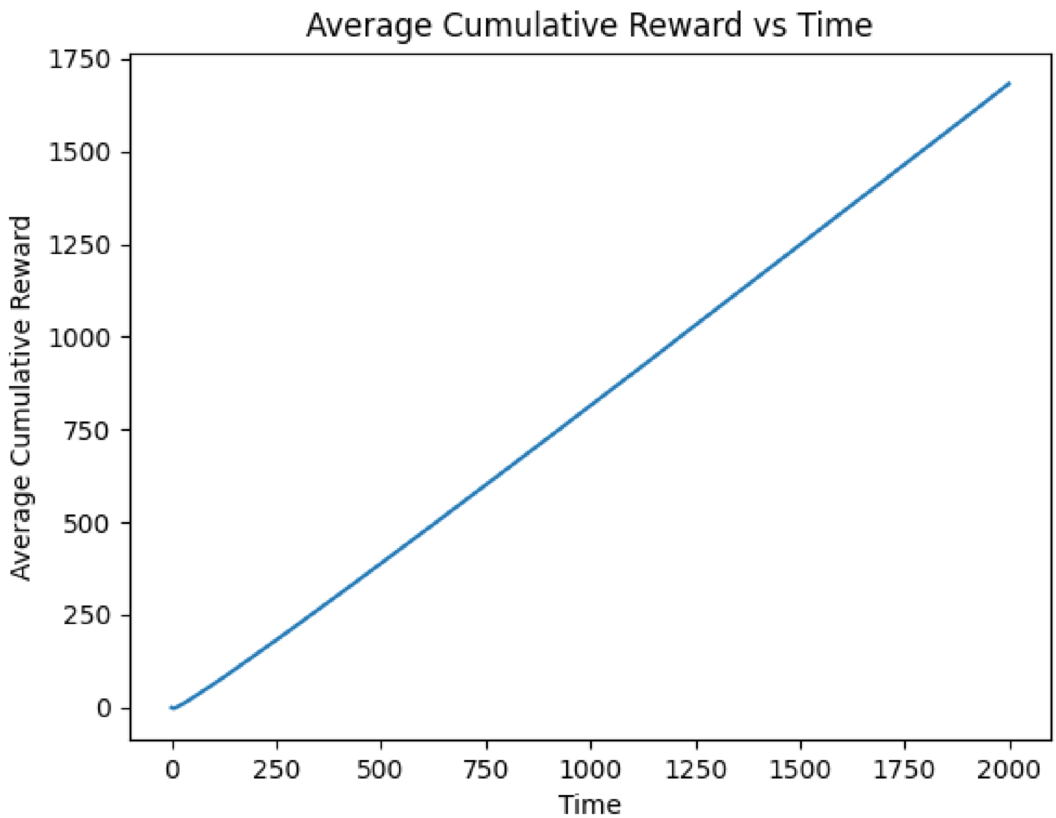
```
1724.1414861976564

<IPython.core.display.Javascript object>
```



Next, we run the algorithm over multiple simulations, which we generate by permutating the input data. The obtained average results are naturally less "noisy". **It may take many simulations to get low-variance, averaged results.**

In [35]:
```
def bandit_epsgreedy_sims(datasim=tenarm_datal, epsilon=0.1, nbr_sims=10):
    # parameters
    num_bandits = datasim.shape[1]
    T = datasim.shape[0]

    # store values
    sim_cum_rewards = np.zeros((nbr_sims,T))
    sim_total_rewards = np.zeros(nbr_sims)

    for s in range(nbr_sims):
        (dummy,dummy, cum_rewards, total_reward) = bandit_epsgreedy(data=np.ra
                                                                    eps=epsilo
        sim_cum_rewards[s,:] = cum_rewards
        sim_total_rewards[s] = total_reward

    return (sim_cum_rewards, sim_total_rewards)


(sim_cum_rewards, sim_total_rewards) = bandit_epsgreedy_sims(epsilon=0.15, nbr_
print('Average total reward = ', np.average(sim_total_rewards))

sim_avg_rewards = np.average(sim_cum_rewards, axis=0)
sim_avg_rewards.shape
plt.figure()
plt.title('Average Cumulative Reward vs Time')
plt.xlabel('Time')
plt.ylabel('Average Cumulative Reward')
plt.plot(sim_avg_rewards)
plt.show()
```

```
Average total reward =  1681.9248085941124

<IPython.core.display.Javascript object>
```



Exploration vs Exploitation Trade-off

It is important to investigate the relationship between the outcome (average cumulative reward over time) and ε parameter. For small ε, the algorithm is more greedy and chooses the best action (given knowledge level, here Q estimate) most of the time. This is called **exploitation** in [reinforcement learning (RL) (https://en.wikipedia.org/wiki/Reinforcement_learning)](https://en.wikipedia.org/wiki/Reinforcement_learning). For large ε, the algorithm spends more time in **exploration** mode and obtains better Q estimates. This **exploration vs exploitation** trade-off is [fundamental to all RL approaches (https://www.coursera.org/learn/fundamentals-of-reinforcement-learning)](https://www.coursera.org/learn/fundamentals-of-reinforcement-learning), not just multi-armed bandits. The same concepts are also relevant to [dual control (https://en.wikipedia.org/wiki/Dual_control_theory)](https://en.wikipedia.org/wiki/Dual_control_theory) as well as [adaptive control (https://en.wikipedia.org/wiki/Adaptive_control)](https://en.wikipedia.org/wiki/Adaptive_control).

Question 3.1 [30%] A Multi-armed bandit for CDN Optimisation

In this question, the problem of real-world data retrieval from multiple redundant sources is investigated. This communication network problem is commonly known as the Content Distribution Network (CDN) problem [(see a relevant paper, right click to download) (./files/performance_of_CDN.pdf)](./files/performance_of_CDN.pdf). An agent must retrieve data through a network with several redundant sources available. For each retrieval, the agent selects one source and waits until the data is retrieved. The objective of the agent is to minimize the sum of the delays for the successive retrievals. This problem is investigated in Section 4.2 of [this paper, (right click to download) (./files/bandit.pdf)](./files/bandit.pdf) and this related [project (http://bandit.sourceforge.net/)](http://bandit.sourceforge.net/) as well as discussed in this [practical book (http://shop.oreilly.com/product/0636920027393.do)](http://shop.oreilly.com/product/0636920027393.do).

We will use a subset of the [publicly available (./files/license.txt)](./files/license.txt) university web latency data set from the [bandit project (http://bandit.sourceforge.net/)](http://bandit.sourceforge.net/), which contains retrieval delay/latency measurements from over 700 universities' homepages in milliseconds. Let's decrease the number of options (columns) randomly to 20 to make it computationally less time consuming (but you can change this later if you wish). The rewards are the negatives

In [4]:
```
univ_data = pd.read_csv('./files/univ-latencies.csv')
univ_data_samp = -univ_data.sample(n=20, axis=1) #choose 20 columns randomly fo
print(univ_data_samp.shape)
univ_data_samp.head()
```

```
(1361, 20)
```

Out[4]:

	ucf-edu	ua-ac-be	graceland-edu	sou-edu	canisius-edu	skidmore-edu	asbury-edu	baruch-cuny-edu	buffalo-edu	ccon-edu	aum-edu
0	-244	-703	-332	-671	-349	-38	-305	-1651	-1430	-1172	-111
1	-317	-448	-411	-723	-119	-33	-307	-1020	-975	-295	-227
2	-9231	-440	-514	-710	-97	-46	-312	-1341	-129	-328	-102
3	-314	-391	-362	-666	-98	-3427	-325	-1546	-132	-378	-126
4	-240	-389	-439	-717	-112	-52	-600	-1503	-169	-909	-102

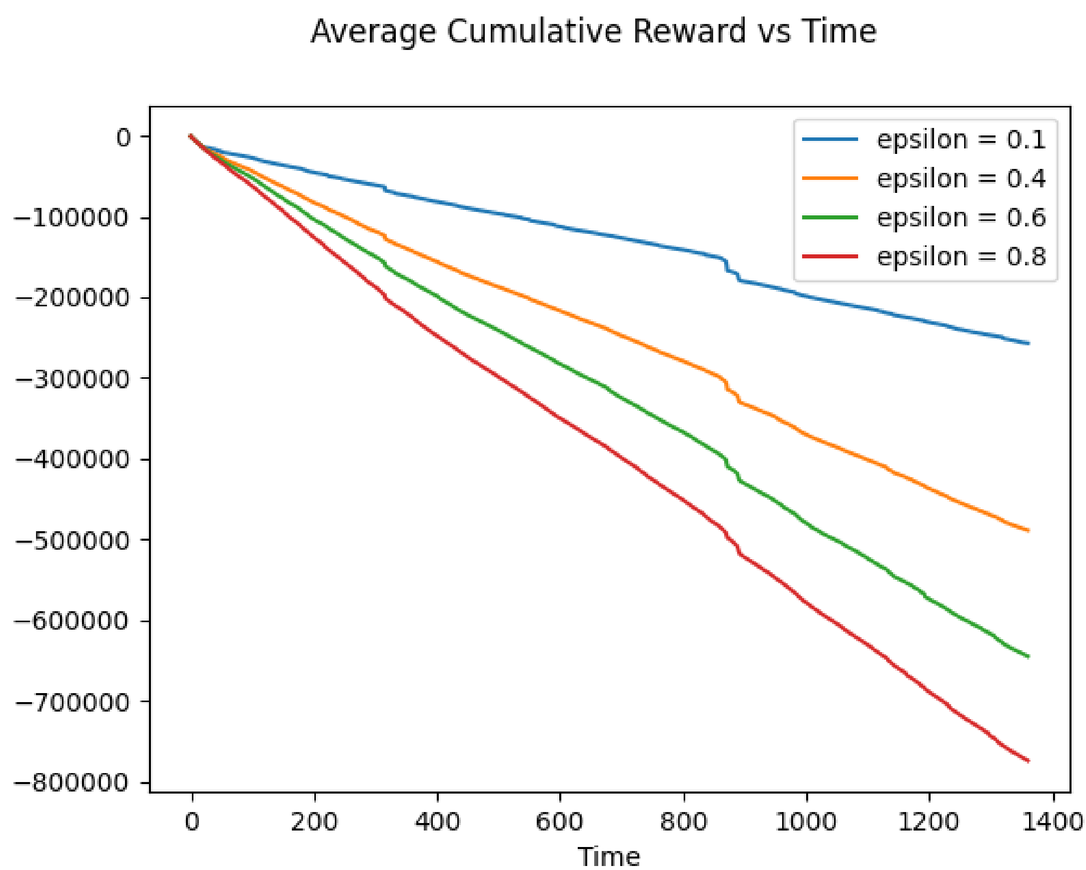
Answer the following by implementing and simulating well-known multi-armed bandit algorithms.

1. Apply the ε-greedy algorithm to the CDN problem. Sample randomly with replacement from the ***univ-latencies*** dataset in order to simulate latencies. You should use negative of latencies as rewards here since high latency is not desirable. Try different ε values to investigate the exploration vs exploitation trade-off and the best total average reward.
2. Implement and apply the ***upper confidence bound*** (UCB) action selection algorithm to the same data set. Compare your results and briefly discuss your findings.

Answer as text here

In [55]:
```
def bandit_epsgreedy_CDN_sim(datacdn=tenarm_datal, epsilon=0.1, nbr_cdn=10):
    # parameters
    num_bandits = datacdn.sample(n=20, axis=1).shape[1]
    T = datacdn.sample(n=20, axis=1).shape[0]

    # store values
    cdn_cum_rewards = np.zeros((nbr_cdn,T))
    cdn_total_rewards = np.zeros(nbr_cdn)

    for s in range(nbr_cdn):
        (dummy,dummy, cum_rewards, total_reward) = bandit_epsgreedy(data= np.a
                                                                    eps=epsilo
        cdn_cum_rewards[s,:] = cum_rewards
        cdn_total_rewards[s] = total_reward
    return (cdn_cum_rewards, cdn_total_rewards)

epsilon=[0.1,0.4,0.6,0.8]
plt.figure().suptitle('Average Cumulative Reward vs Time')
for eps in epsilon:
    (cdn_cum_rewards, cdn_total_rewards) = bandit_epsgreedy_CDN_sim(datacdn=un
    print(f'Average total reward at epsilon {eps} = {np.average(cdn_total_rewa

    cdn_avg_rewards = np.average(cdn_cum_rewards, axis=0)
    # sim_avg_rewards.shape
    plt.plot(cdn_avg_rewards,label=f'epsilon = {eps}')

plt.xlabel('Time')
plt.ylabel('Average Cumulative Reward')
plt.legend()
plt.show()
```

```
<IPython.core.display.Javascript object>
```



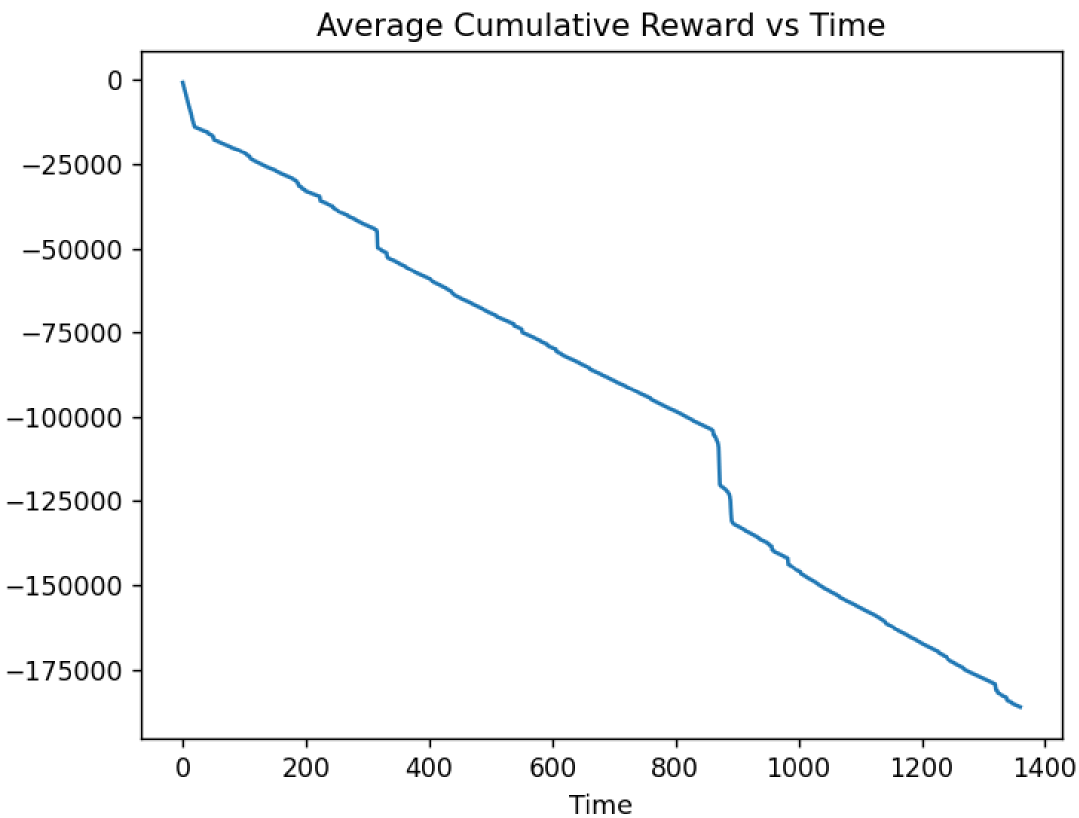

```
Average total reward at epsilon 0.1 = -257129.767
Average total reward at epsilon 0.4 = -488836.494
Average total reward at epsilon 0.6 = -645041.677
Average total reward at epsilon 0.8 = -774348.077
```

- **The best total average reward achieved with epsilon 0.1 (more exploitation) = -257129.769**

In [14]:
```
def bandit_UCB(dataset):
    # parameters
    num_bandits = dataset.shape[1]
    T = dataset.shape[0]

    # store values
    N = np.zeros(num_bandits) # numbers of selections
    cum_rewards = np.zeros(T) # sequence of cumulative rewards
    sums_of_reward = np.zeros(num_bandits)
    bandit_selected = [] # sequence of selections

    for n in range(0, T):
        bandit = 0
        max_upper_bound = -1E700
        for i in range(0, num_bandits):
            if (N[i] > 0):
                average_reward = sums_of_reward[i] / N[i]
                confidence = np.sqrt(np.log(n+1) / N[i])
                upper_bound = average_reward + confidence
            else:
                upper_bound = 0
            if upper_bound > max_upper_bound:
                max_upper_bound = upper_bound
                bandit = i
        bandit_selected.append(bandit)
        N[bandit] += 1
        reward = dataset[n, bandit]
        sums_of_reward[bandit] += reward
        if n>0:
            cum_rewards[n] = reward+cum_rewards[n-1]
        else:
            cum_rewards[n] = reward
    total_reward = cum_rewards[-1]
    numbers_of_selections=N
#     print(N)
#     print(sums_of_reward)
    return total_reward,cum_rewards,numbers_of_selections,bandit_selected


def bandit_UCB_sim(datasim,nbr_sim=10):
    # parameters
    num_bandits = datasim.sample(n=20, axis=1).shape[1]
    T = datasim.sample(n=20, axis=1).shape[0]

    # store values
    sim_cum_rewards = np.zeros((nbr_sim,T))
    sim_total_rewards = np.zeros(nbr_sim)

    for s in range(nbr_sim):
        (total_reward,cum_rewards,dummy,dummy) = bandit_UCB(np.array(-datasim.

        sim_cum_rewards[s,:] = cum_rewards
        sim_total_rewards[s] = total_reward
    return (sim_cum_rewards, sim_total_rewards)

(sim_cum_rewards, sim_total_rewards) = bandit_UCB_sim(univ_data,1000)
```

```
print('Average total reward = ', np.average(sim_total_rewards))
sim_avg_rewards = np.average(sim_cum_rewards, axis=0)
plt.figure()
plt.title('Average Cumulative Reward vs Time')
plt.xlabel('Time')
plt.ylabel('Average Cumulative Reward')
plt.plot(sim_avg_rewards)
plt.show()
```

```
Average total reward =  -186170.812

<IPython.core.display.Javascript object>
```



- **It is seen that with UCB the reward has increased (minimized total latency) compared to epsilon greedy method**
- **The UCB method selects non greedy action based on their potential for being optimal and uncertainity achieved over time. That is the reason for the results to be better with UCB**

Workshop Assessment Instructions

You should complete the workshop tasks and answer the questions within the allocated session! **Submission deadline is the the end of second Week of the workshop. Please check Canvas for exact deadline!**

It is **mandatory to follow all of the submissions guidelines** given below. ***Don't forget the Report submission information on top of this notebook!***

1. The completed Jupyter notebook and its Pdf version (you can simply print-preview and then print as pdf from within your browser) should be uploaded to the right place in Canvas. ***It is your responsibility to follow the announcements!*** **Late submissions will be penalised (up to 100% of the total mark depending on delay amount)!**
2. Filename should be "ELEN90088 Workshop **W: StudentID1-StudentID2** of session **Day-Time**", where **W** refers to the workshop number, **StudentID1-StudentID2** are your student numbers, **Day-Time** is your session day and time, e.g. ***Tue-14***.
3. Answers to questions, simulation results and diagrams should be included in the Jupyter notebook as text, code, plots. ***If you don't know latex, you can write formulas/text to a paper by hand, scan it and then include as image within Markdown cells.***
4. Please submit your report individually. Partners can submit the same report.

Workshop Marking

- **Each workshop has 10 points corresponding to 10% of the total subject mark.** You will receive 3 points from the submitted report and 7 points from an **individual oral examination.**
- Individual oral quizzes will be scheduled within the next two weeks following the report submission. They will be during workshop hours. Therefore, it is important that you attend the workshops!
- The individual oral examination will assess your answers to workshop questions, what you have done in that workshop, and your knowledge of the subject material in association with the workshop.

Additional guidelines for your programs:

- Write modular code using functions.
- Properly indent your code. But Python forces you do that anyway ;)
- Heavily comment the code to describe your implementation and to show your understanding. No comments, no credit!
- Make the code your own! It is encouraged to find and get inspired by online examples but you should exactly understand, modify as needed, and explain your code via comments. If you resort to blind copy/paste, you will

In [1]: