A Report for Assignment 1 of Machine Learning

(UCS611)

Submitted by:

101715001 **Aakriti Sachdeva** (ENC 1)

101715006 **Abhishek Goyal** (ENC 1)

Submitted to:

Dr. Ravi Kumar

Department of Electronics and Communication Engineering

THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY, (DEEMED TO BE UNIVERSITY), PATIALA, PUNJAB

OBJECTIVES -

1. Build an ML model and predict recovery rate and infection rate in India for the period 15th June to 31st of July.

2. Build an ML model and predict death rate for the countries for which data are available in the age group 35-50 till 31st July of 2020.

3. Build an ML model and predict infection rate for which data are available in the age group 35-50 till 31st of July 2020.

4. Using information obtained from the above models in this and the previous assignment and other data available on the link predict the tentative date (or interval spanning a week) after which the infection spread is 25% of its peak value in the world and in India.

DATA USED –

The data used in the model is fetched from official government API namely https://api.covid19india.org/data.json

```
import requests

api = 'https://api.covid19india.org/data.json'
data = requests.get(api)
with open('data.json', 'wb') as f:
    f.write(data.content)

time_series_data = data.json()['cases_time_series']
print(time_series_data)
```

This code fetches the data from API in JSON format and reads it into Python Dictionary. From this we get data in the following format -


```
{
    "cases_time_series": [
        {
            "dailyconfirmed": "1",
            "dailydeceased": "0",
            "dailyrecovered": "0",
            "date": "30 January ",
            "totalconfirmed": "1",
            "totaldeceased": "0",
            "totalrecovered": "0"
        },
        {
            "dailyconfirmed": "0",
            "dailydeceased": "0",
            "dailyrecovered": "0",
            "date": "31 January ",
            "totalconfirmed": "1",
            "totaldeceased": "0",
            "totalrecovered": "0"
        },
        .
        .
        .
    ]
}
```

The following code converts the above depicted data to a more usable format -

```
def transpose(list_of_dict):
    dict_of_list = {}
    for key in list_of_dict[0]:
        if key == 'date':
            dict_of_list[key] = [dic[key] for dic in list_of_dict]
        else:
            dict_of_list[key] = [int(dic[key]) for dic in list_of_dict]
    return dict_of_list

time_series_list = transpose(time_series_data)
print(time_series_list)

def get_data():
    return time_series_list
```

Output after data conversion –

```
{
    "dailyconfirmed": [. . . , 8, 10, 10, 11, 10, 14, 20, 25, 27, 58,
        78, 69, 94, 74, 86, 73, 153, 136, 120, 187, 309, . . .],
    "dailydeceased": [. . . , 0, 1, 0, 0, 1, 0, 1, 0, 0, 3, 2, 1, 1,
        5, 3, 5, 3, 14, 6, 6, 16, 14, 13, 22, 16, 27, 20, . . .],
    "dailyrecovered": [. . . , 3, 1, 1, 0, 5, 3, 0, 0, 2, 15, 3, 7,
        25, 10, 17, 35, 13, 19, 22, 39, 56, 43, 65, 75, 96, . . .],
    "date": [. . . , "13 February ", "14 February ", "15 February ",
        "16 February ", "17 February ", "18 February ", . . .],
    "totalconfirmed": [. . . , 5, 6, 28, 30, 31, 34, 39, 48, 63, 71,
        81, 91, 102, 112, 126, 146, 171, 198, 256, 334, . . .],
    "totaldeceased": [. . . , 2, 2, 3, 3, 4, 4, 4, 7, 9, 10, 11, 16,
        19, 24, 27, 41, 47, 53, 69, 83, 96, 118, 134, . . .],
    "totalrecovered": [. . . , 5, 15, 20, 23, 23, 23, 25, 40, 43, 50,
        75, 85, 102, 137, 150, 169, 191, 230, 286, 329, . . .]
}
```

Above given codes will be referred as dataloader.py after this.

Synthetic data generation:

For objective 2

Based on the data given at-
https://www.worldometers.info/coronavirus/coronavirus-age-sex-demographics/
It can be seen that for New York city:
on 14 April, 23.1% of deaths are from age group 45-64
on 13 May, 22.4% of deaths are from age group 45-64

Assuming same distribution in India, the deaths for the respective dates, for the age group 45-64 can be calculated as:

14 April -> 23.1% of Total deaths in India on this date = 0.231 * 353 = 81
13 May -> 22.4% of Total deaths in India on this date = 0.224 * 2415 = 541

Now, we assume, that for the days between 14 April and 13 May, the relation between Total deaths, and deaths from age group 45-64, remains uniform. Hence, if we can measure this relation, we can generate synthetic day-wise data for days between 14/04/2020 and 13/05/2020.
We'll measure this relation (or better, say, correlation), by simply measuring the covariance between them.

Date	% of Age grp deaths	Total deaths
14 April	23.1	353
13 May	22.4	2415

Measuring covariance for this matrix, where columns are considered to be features of the data that needs to be synthetically generated, so obviously our covariance matrix would be 2x2. The covariance matrix would be:

	% of Age grp confirmed	Total confirmed
% of Age grp confirmed	0.1225	-360.85
Total confirmed	-360.85	1062961.0

And mean of each feature =>

% of Age grp confirmed (22.75) Total confirmed (1384)

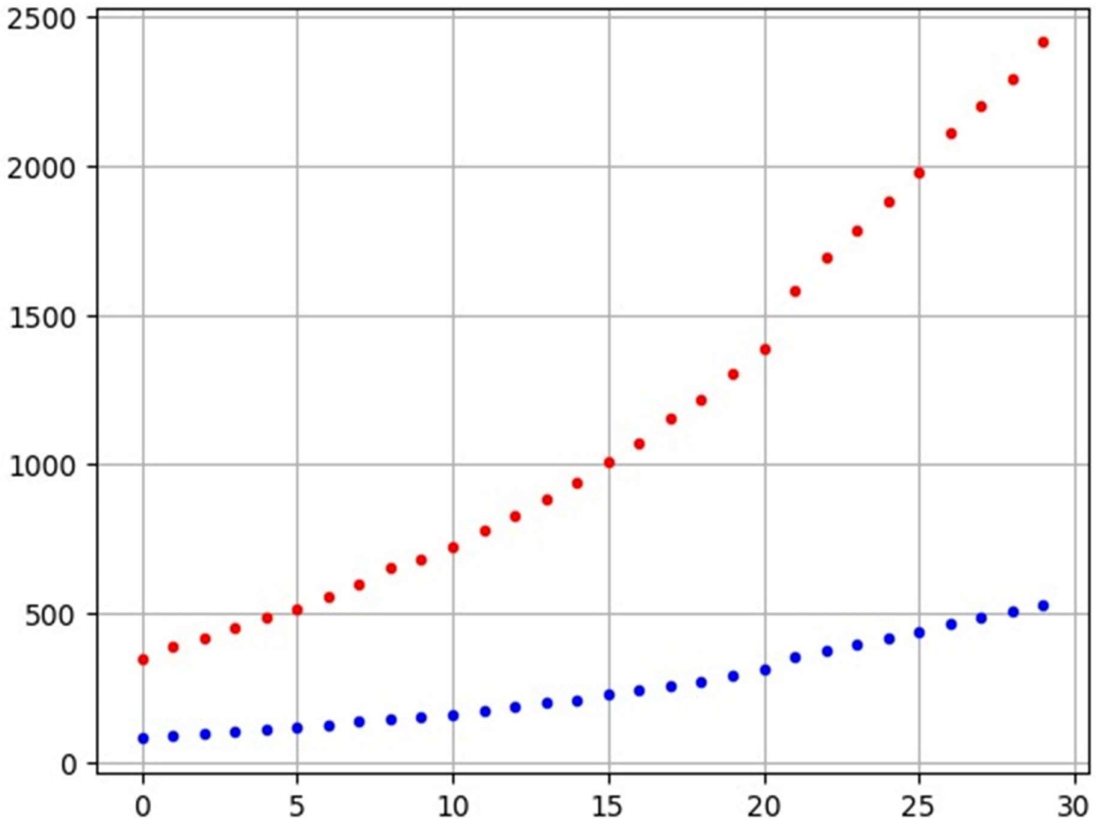
Now, we can generate data (% of deaths in the age group 45-64) that follows this property of covariance with total deaths, and is centred around this mean, and sort that data descending (assuming monotonic change from 14 April to 13 May).

Code used to generate points:

```
import numpy as np
import matplotlib.pyplot as plt

def gen_points(num_samples, mu, r):
    y = np.random.multivariate_normal(mu, r, size=num_samples)
    a=y[:,0]
    t=y[:,1]
    a=sorted(a, reverse=True)
    plt.plot(a, 'b.')
    plt.savefig('./age_deaths_data.png')
    plt.grid(True)
    plt.show()

num_samples = 30
# The desired mean values of the sample.
mu = np.array([22.75, 1384.0])
# The desired covariance matrix.
r = np.array([
        [  0.1225, -360.85],
        [ -360.85,  1062961.0]
    ])
gen_points(num_samples, mu, r)
```



This graph represents data values from 14th April to 13th May. The red dots represent the total no. Of deaths in this period while blue dots represent synthetic generated deaths for age group 39-50.

For objective 3

Based on the data given at-
https://www.worldometers.info/coronavirus/coronavirus-age-sex-demographics/
It can be seen that for New York city:
on 14 April, 23.1% of deaths are from age group 45-64
on 13 May, 22.4% of deaths are from age group 45-64

Assuming same distribution for confirmed cases in India, the confirmed for the respective dates, for the age group 45-64 can be calculated as:

14 April -> 23.1% of Total confirmed in India on this date = 0.231 * 10815 = 2498
13 May -> 22.4% of Total confirmed in India on this date = 0.224 * 74281 = 16639

Now, we assume, that for the days between 14 April and 13 May, the relation between Total confirmed, and confirmed from age group 45-64, remains uniform. Hence, if we can measure this relation, we can generate synthetic day-wise data for days between 14/04/2020 and 13/05/2020.
We'll measure this relation (or better, say, correlation), by simply measuring the covariance between them.

	% of Age grp confirmed	Total confirmed
14 April	23.1	10815
13 May	22.4	74281

Measuring covariance for this matrix, where columns are considered to be features of the data that needs to be synthetically generated, so obviously our covariance matrix would be 2x2. The covariance matrix would be:

	% of Age grp confirmed	Total confirmed
% of Age grp confirmed	0.1225	-11106.55
Total confirmed	-11106.55	1006983289.0

And mean of each feature =>

% of Age grp confirmed (22.75) Total confirmed (42548)

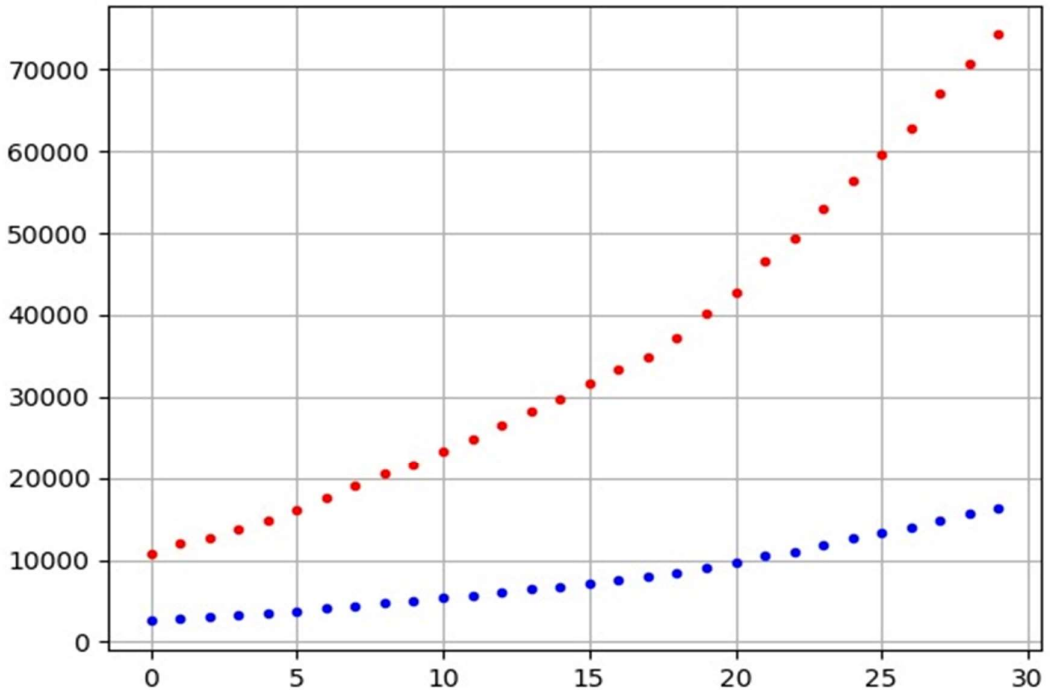
Now, we can generate data (% of confirmed in the age group 35-64) that follows this property of covariance with total confirmed, and is centred around this mean, and sort that data descending (assuming monotonic change from 14 April to 13 May).

Code used to generate points:

```
import numpy as np
import matplotlib.pyplot as plt

def gen_points(num_samples, mu, r):
    y = np.random.multivariate_normal(mu, r, size=num_samples)
    a=y[:,0]
    t=y[:,1]
    a=sorted(a, reverse=True)
    plt.plot(a, 'b.')
    plt.savefig('./age_deaths_data.png')
    plt.grid(True)
    plt.show()

num_samples = 30
# The desired mean values of the sample.
mu = np.array([22.75, 42548.0])
# The desired covariance matrix.
r = np.array([
        [  0.1225, -11106.55],
        [ -11106.55,  1006983289.0]
    ])
gen_points(num_samples, mu, r)
```



This graph shows data of confirmed cases from date 14th April to 13th may. The red dots represent total number of confirmed cases while the blue dots represent confirmed cases between age 39-50.

MODELS USED –

MLPs (Multi Layered Perceptron(s)) of different sizes are used for prediction of the given time series data. There are five different sizes used.

The MLPs (aka Vanilla NNs) are completely implemented from scratch without using any ML / DL Library.

Each model is run for 400 Epochs spitting out a graph every 100 Epochs. So in total there are 80 (4 graphs per model x 5 models per task x 4 tasks) graphs. So only graphs depicting something useful or best performing are shown in the report.

All graphs can be seen in the zip file in high resolution.

For training the model the data is transformed into X (input), Y (output) pairs. X being a 1 x 10 sized vector and Y being a 1 x 1 vector. X is 10 consecutive days of input data and Y is the 1 following day output data.

Following code does the stated transformation –

```
import data_loader as dtl
import numpy as np

def process_data(raw_data):
    max_case = max(raw_data)
    max_case += max_case / 3
    raw_data = [case / max_case for case in raw_data]
    train_data = [(np.array(raw_data[i-11:i-
1]).reshape(1, 10), np.array(raw_data[i]).reshape(1, 1)) for i in range(11, le
n(raw_data))]
    return train_data, max_case

data_loader = dtl.get_data()
raw_train_data = data_loader['totalconfirmed']
train_data, max_case = process_data(raw_train_data)
print(train_data)
```

Output Format –

A list of (array (1 x 10), array (1 x 1)) tuples denoting X and Y respectively.

```
[
    (array([[3.93446751e-06, 3.93446751e-06, 3.93446751e-06, 7.86893502e-06,
    1.18034025e-05, 1.18034025e-05, 1.18034025e-05, 1.18034025e-05,
    1.18034025e-05, 1.18034025e-05]]), array([[1.18034025e-05]])),
    (array([[3.93446751e-06, 3.93446751e-06, 7.86893502e-06, 1.18034025e-05,
    1.18034025e-05, 1.18034025e-05, 1.18034025e-05, 1.18034025e-05,
    1.18034025e-05, 1.18034025e-05]]), array([[1.18034025e-05]])),
    .
    .
    .
]
```

Now we have our training data completely ready. The values were normalised by dividing the data by (max(data) * 4 / 3) effectively giving values between 0 (min) to 0.75 (max).

CODE –

MLPs code is written from scratch in python using numpy & matplotlib is used to draw graphs.

It is as follows –

1. Importing all the required libraries for the MLP –

```
import numpy as np
import random
import math
import time
import os
import matplotlib.pyplot as plt
from matplotlib.ticker import MultipleLocator
```

2. The Layer class which makes a single layer of the perceptron. It takes Input Size, Output Size and Learning Rate as arguments and implements all the basic functions required by the model to train.

```
def sigmoid(z):
    return 1.0/(1.0+np.exp(-z))

def sigmoid_prime(z):
    return sigmoid(z)*(1-sigmoid(z))

class layer:
    def __init__(self, inp_size, out_size, ETA):
        self.ETA = ETA
        self.b = np.random.randn(1, out_size)
        self.w = np.random.randn(inp_size, out_size)
        self.bgrad = 0.0
        self.wgrad = 0.0
    def forward(self, inp):
        self.x = inp
        self.y = np.dot(self.x, self.w) + self.b
        self.a = sigmoid(self.y)
        return self.a
    def compgrad(self, a_grad):
        y_grad = sigmoid_prime(self.y) * a_grad
        self.bgrad += y_grad
        self.wgrad += np.dot(self.x.T, y_grad)
        return np.dot(y_grad, self.w.T)
    def backprop(self):
        self.w -= self.ETA * self.wgrad
        self.b -= self.ETA * self.bgrad
        self.bgrad = 0
        self.wgrad = 0
```

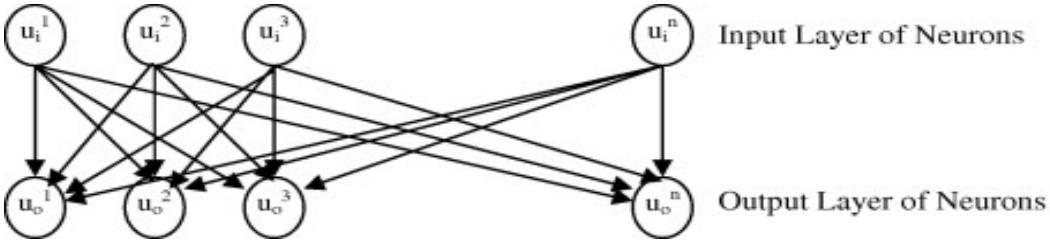
The '__init__' function takes 3 arguments 'inp_size', 'out_size' & 'ETA' and creates a bias vector 'b' and weight vector 'w' of size (1 x 'out_size') and ('inp_size' x 'out_size') respectively. Both of these are randomly initialised using normal distribution. Two variables 'bgrad' and 'wgrad' are also initialised which will be used later to accumulate the gradient.

The 'forward' function takes a 'inp' vector of size (1, 'inp_size) and stores it in 'x'. It then computes sigmoid(wx + b) and returns it as the output 'a'.

The ‘compgrad’ function takes an argument ‘a_grad’ which is gradient vector of the loss function on ‘a’ with respect to loss $\left(\frac{dloss}{da}\right)$. It than computes ‘y_grad’ which is sigmoid_prime(‘y’) * ‘a_gard’ $\left(\frac{dloss}{dy} = \frac{da}{dy} * \frac{dloss}{da}\right)$. It than accumulates ‘bgrad’ which is same as ‘ygrad’. Than it accumulates ‘wgrad’ which is ‘x’ dot ‘y_grad’ $\frac{dloss}{dw} = \left(\frac{dy}{dw} * \frac{dloss}{dy}\right)$. Finally, it returns ‘xgrad’ which is ‘w’ dot ‘y_grad’ $\left(\frac{dloss}{dx} = \frac{dy}{dx} * \frac{dloss}{dy}\right)$ and is to be used by previous layer as its ‘a_grad’.

Note that actual weights weren’t updated in this function. They are only accumulated in two class variables. This will be useful in accumulating the gradient for a ‘batch’.

Finally, the ‘backprop’ function. It takes no argument and just uses internally stored gradients (‘wgrad’ and ‘bgrad’) to update weights ‘w’ and biases ‘b’. It subtracts learning rate * gradient of vector from the vector (‘w’ -= ‘ETA’ * ‘wgrad’ and ‘b’ -= ‘ETA’ * ‘bgrad’) and sets ‘wgrad’ and ‘bgrad’ to 0.0.

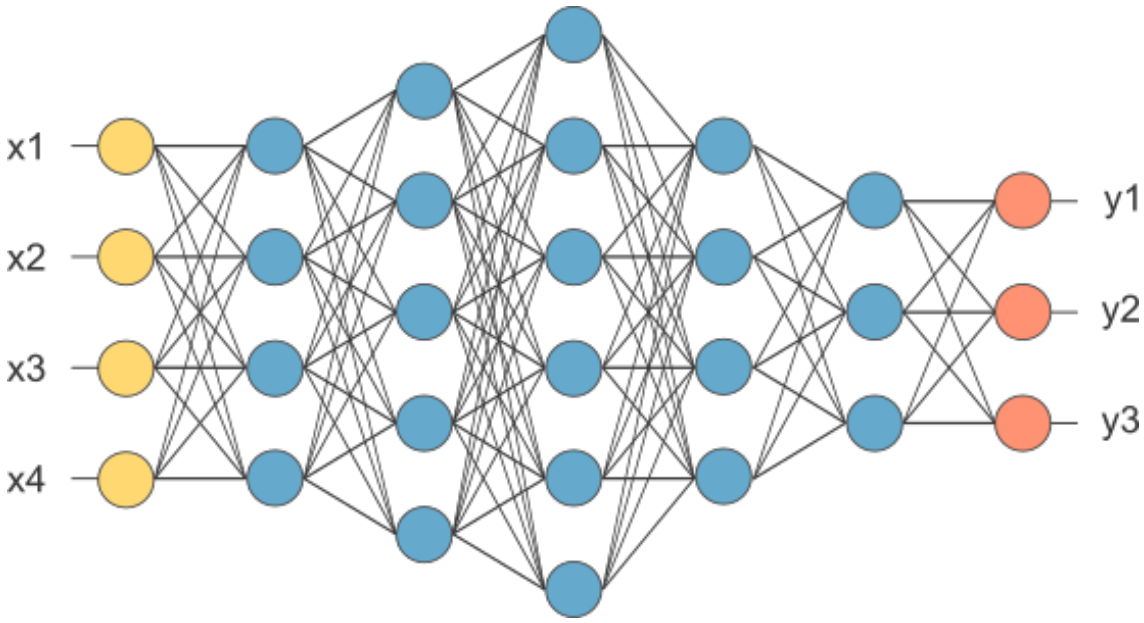


3. The Model class makes it easier to define and train a multi-layered network. As we can see the above class only makes a 2-layered network with no hidden layer. We can make hidden layers by stacking input output layers. Doing the manually for every model will not be very convenient. That’s what the model class does.

```
class model:
    def __init__(self, net_size, ETA):
        self.layers = []
        for i in range(len(net_size) - 1):
            self.layers.append(layer(net_size[i], net_size[i + 1], ETA))
    def forward(self, x):
        for l in self.layers:
            x = l.forward(x)
        return x
    def compgrad(self, a_grad):
        for l in reversed(self.layers):
            a_grad = l.compgrad(a_grad)
    def backprop(self):
        for l in reversed(self.layers):
            l.backprop()
```

It takes input a list called 'net_size' and learning rate 'ETA'. The input list contains a list of integers defining the size of layers in the network.

For example list[4, 4, 5, 6, 4, 3, 3] will make the following network-



All the blue neurons are hidden layers and yellow layer must be equal to input size and orange layer denotes the output size of our data.

4. The ‘train’ function takes input the training data and batch size called ‘data’ and ‘BATCH_SIZE’ respectively.

```
def train(data, BATCH_SIZE= 10):
    data_size = len(data)
    # random.shuffle(data)
    batches = (data[i : i + BATCH_SIZE] for i in range(0, data_size, BATCH_
SIZE))
    loss = 0.0
    for batch in batches:
        for x, y in batch:
            a = network.forward(x)
            loss += (a - y) ** 2 / BATCH_SIZE
            a_grad = 2 * (a - y) / BATCH_SIZE
            network.compgrad(a_grad)
        network.backprop()
    return np.sum(loss) * BATCH_SIZE / data_size
```

Train function divides the data into mini batches of size ‘BATCH_SIZE’ and the for each batch it updates the model using ‘backprop’ function. The inner loop for each batch goes through every single training example accumulates the gradients for the batch using ‘compgrad’ function.

5. The test function called ‘predict’ takes quite a few variables.

```
def int_round(n, p=0):
    return int(round(n / 10**int(math.log10(n)), p) * 10**int(math.log10(n)))

def predict(data, line='-', title='', file_path='./', future=10):
    fig = plt.figure(figsize=(13,5))
    fig = plt.figure(figsize=(16,4.5))
    ax = plt.axes()
    ax.set_title(title)
    ax.xaxis.set_major_locator(MultipleLocator(5))
    ax.xaxis.set_minor_locator(MultipleLocator(1))
    ax.yaxis.set_major_locator(MultipleLocator(int_round(max_case) // 10))
    ax.yaxis.set_minor_locator(MultipleLocator(int_round(max_case) // 50))
    ax.grid(which='major')
    ax.grid(which='minor', alpha=0.3)
    plt.plot(raw_train_data, line)
    size = len(raw_train_data)
    out = [network.forward(x).item() for x,_ in data]
    plt.plot(list(range(11, size)), [c * max_case for c in out], line)
    pred = []
    for i in range(future):
        pred.append(network.forward(np.array(out[-10:]).reshape(1, 10)).item())
```

```
        out.append(pred[-1])
    plt.plot(list(range(size, size+future)), [c * max_case for c in pred], line)
    plt.legend(['Ground Truth', 'Testset Validation', 'Future Prediction'])
    os.makedirs(os.path.dirname(file_path), exist_ok=True)
    plt.savefig(file_path, bbox_inches='tight', dpi=600)
    plt.close()
```

It plots a graph using matplotlib. We can see 3 graphs are plotted on same figure in this function. The 3 graphs are 'Ground Truth', 'Testset Validation' and 'Future Prediction'. ‘Ground truth’ is the actual dataset plotted, ‘Testset Validation’ is the models output on training data and ‘Future Prediction’ is the models output on ‘future’ future days. Here future variable controls the number of days to predict. For this model we have use future equal to 30.

6. Finally, the main code which calls and runs all these functions and classes together.

```
data_loader = dtl.get_data()
tasks = list(data_loader.keys())
tasks.remove('date')
network_list = [
    [10, 100, 32, 10, 1],
    [10, 100, 100, 1],
    [10, 100, 32, 100, 1],
    [10, 200, 130, 200, 150, 200, 130, 200, 150, 1],
    [10, 80, 100, 100, 120, 140, 140, 120, 100, 100, 25, 1]
]
for task in tasks:
    EPOCHS = 500
    BATCH_SIZE = 1
    ETA = 0.06
    raw_train_data = data_loader[task]
    train_data, max_case = process_data(raw_train_data)
    line = 'x' if task.find('total') != -1 else '-'
    for i in range(len(network_list)):
        network = model(network_list[i], ETA)
        network_str = ' x '.join(map(str, network_list[i]))
        for itr in range(EPOCHS):
            if itr % 100 == 0:
                title = f'Task: {task}    Model: {network_str}    Epochs: {itr}'
                file_path = f'./graphs/{task}/{i+1}/MLP_predction{itr}.png'
                predict(train_data, line=line, title=title, file_path=file_path, future
=30)
            loss = train(train_data, BATCH_SIZE)
            print(f'Loss: {loss}\tTask: {task}\tModel: {i+1}\tEpoch: {itr}')
```

First the ‘data_loader’ get all the data from the data_loader.py. Then ‘tasks’ extract the keys from the dictionary and removes the ‘date’ key.

‘tasks’ now have the following keys [‘dailyconfirmed’, ’dailydeceased’, ‘dailyrecovered’, ‘totalconfirmed’, ‘totaldeceased’, ‘totalrecovered’]. This is from the data provided by the government. This was explained before in the document in data_loader.py. The data with ‘daily’ in key corresponds to its rate (per day) and the data with ‘total’ in key corresponds to total cumulative infections till that date.

Network list defines the sizes of the models used, these are –
10 x 100 x 32 x 10 x 1
10 x 100 x 100 x 1
10 x 100 x 32 x 100 x 1
10 x 200 x 130 x 200 x 150 x 200 x 130 x 200 x 150 x 1
10 x 80 x 100 x 100 x 120 x 140 x 140 x 120 x 100 x 100 x 25 x 1

Then a loop runs for all the 6 tasks taking 1 of them as training data.

Inside the loop we have variables defining the epochs = 500, batch size = 1, learning rate ‘ETA’ = 0.06.

‘raw_train_data’ gets the data from ‘data_loader’ in time-series list format.

‘train_data’, ‘max_case’ gets data in list of (input, output) tuples where input and output are numpy arrays of size (1, 10) and (1, 1) respectively. This was also explained above in the document.

Then I set the line variable which controls the design of line on the graph.

Then a loop runs for ‘network_list’ to choose a model and fit the given tasks on it.

Inside it is the usual loop for ‘EPOCHS’ and calls to ‘train’ and ‘predict’ functions and a few variables to set title and save directory of the graphs.

RESULTING GRAPHS –

Following shown are some of the graphs which showed a good predict. Other graphs can be seen in the zip file in high res.

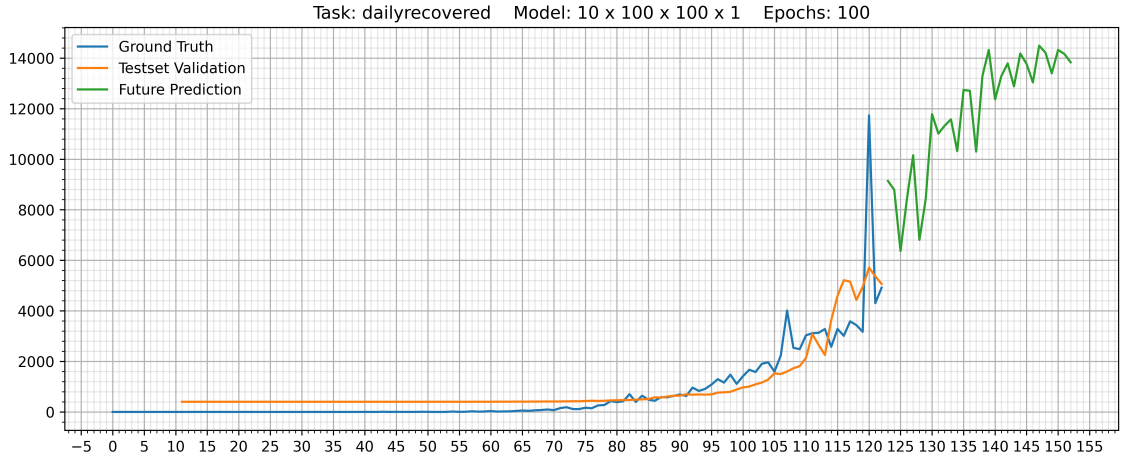
The first 3 models defined in the code are shallow models and the last two are deep models.

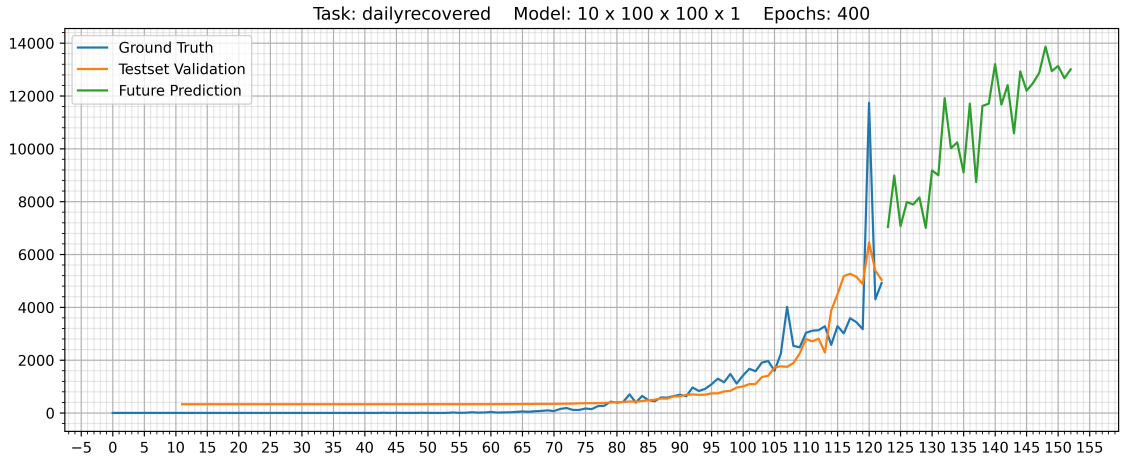
Generally, the shallow models performed well on the cumulative data (as it is smooth) while the deep models performed well on the daily data (rate per day) as the data changes frequently.

Deep models failed miserably on the cumulative data as the just over-fitted the training data predicting random values for future.

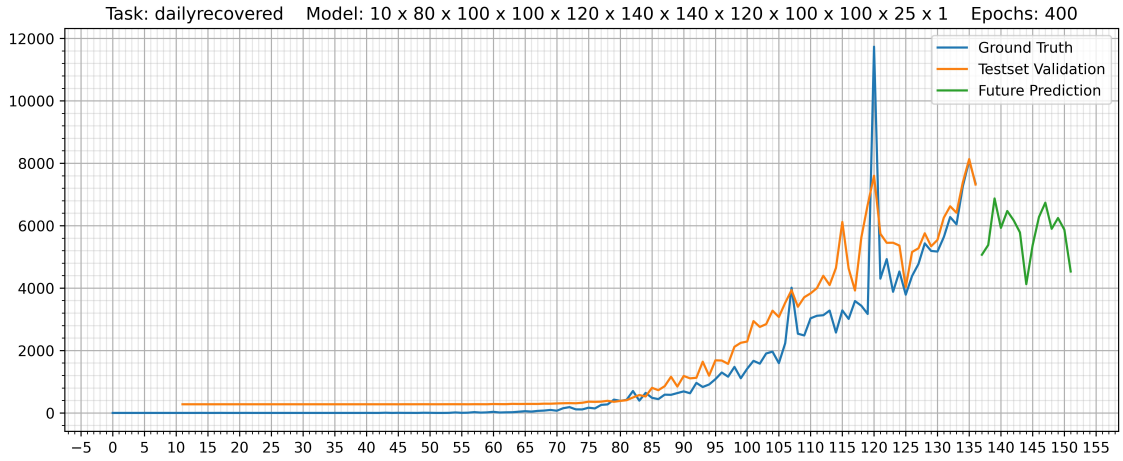
Graphs of ML model for predicting recovery rate and infection rate in India till 31st of July 2020.

Recovery Rate –

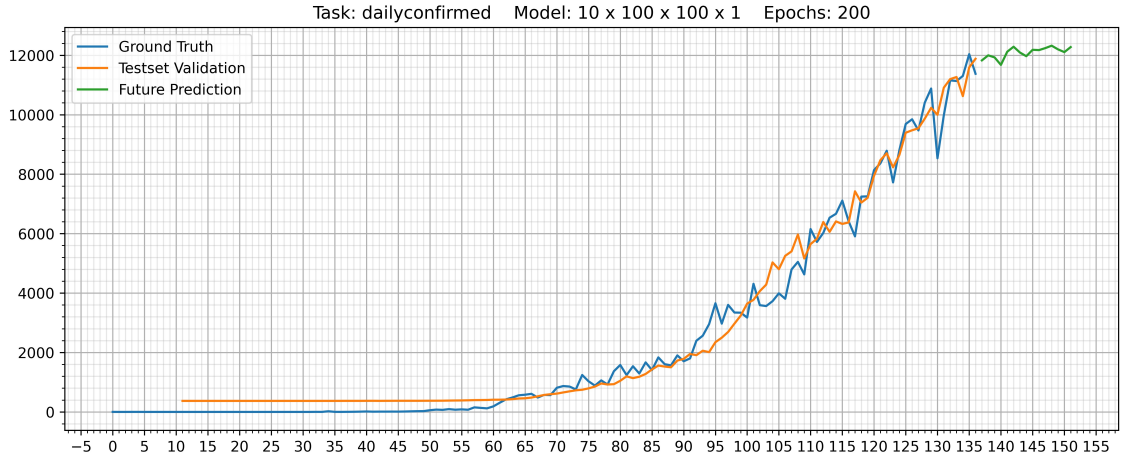


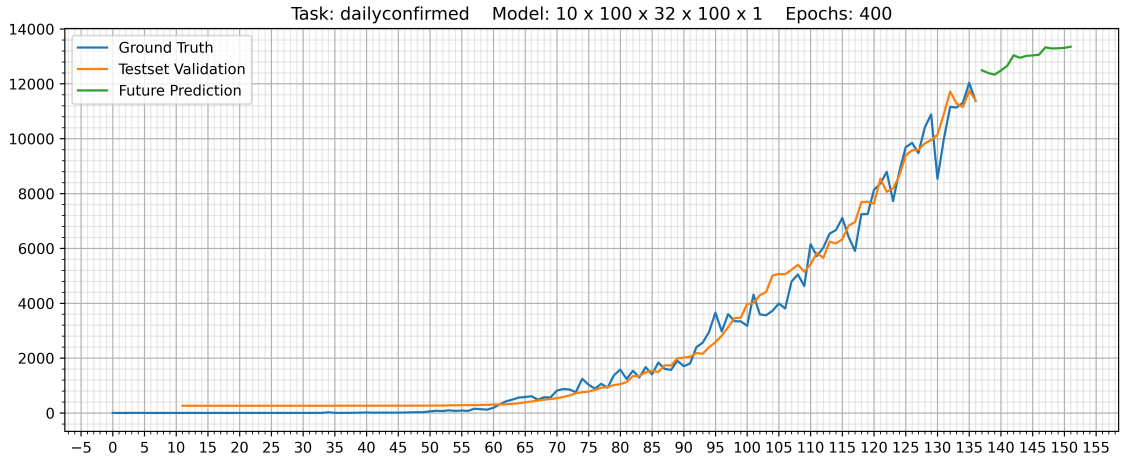


Deep model prediction –

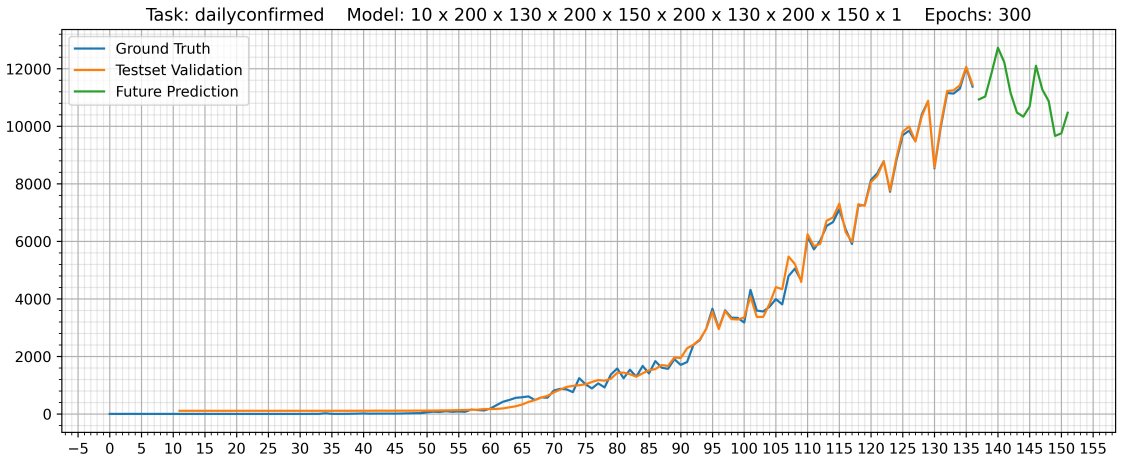


Infection rate –

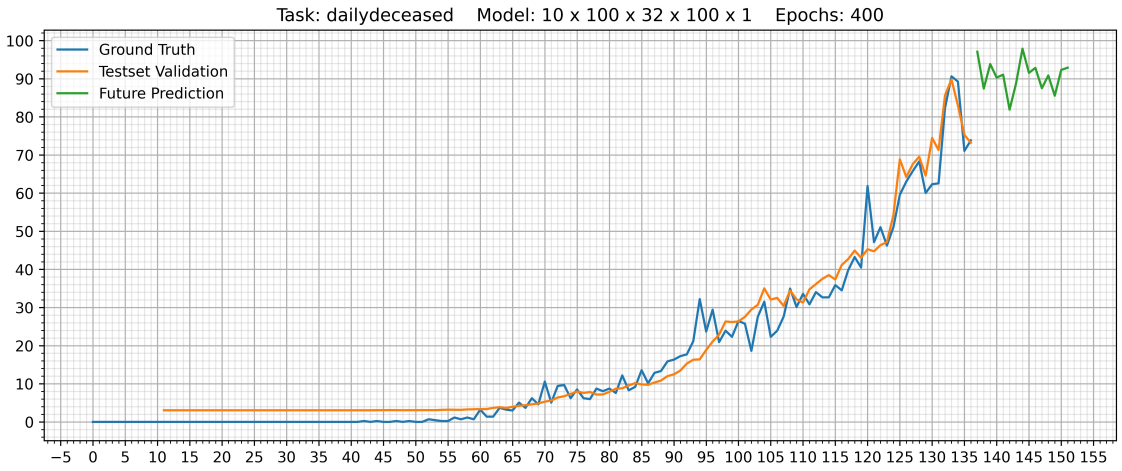


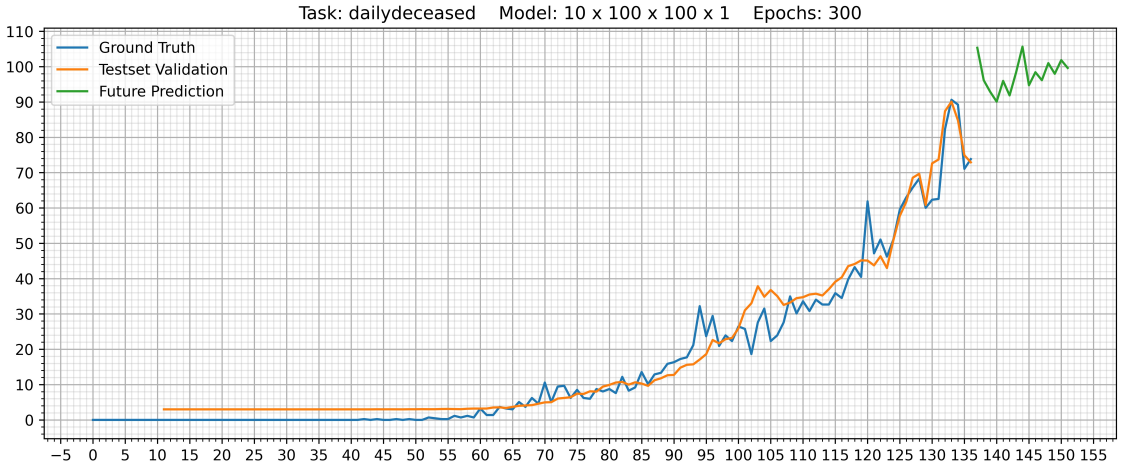


Deep models prediction –

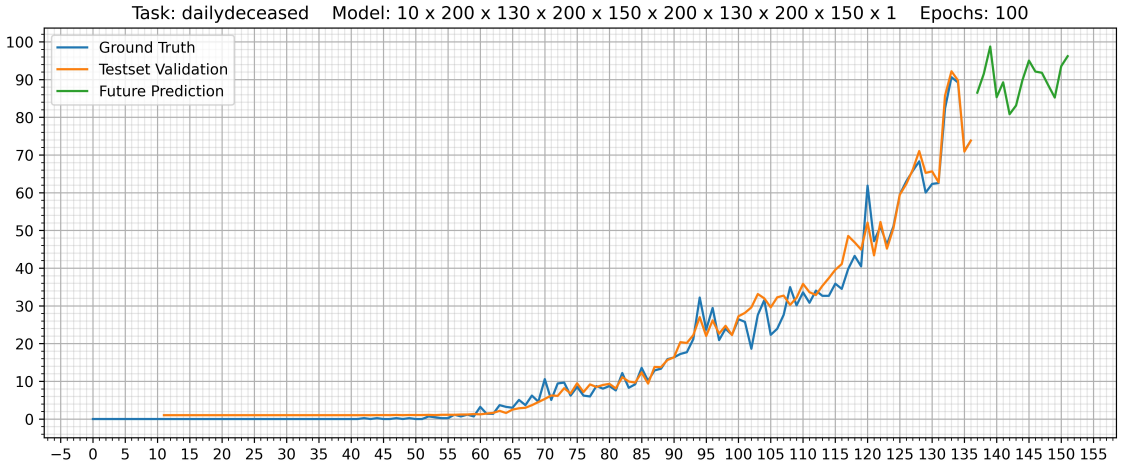


Graphs of ML model predicting death rate in India in the age group 35-50 till 31st July of 2020-

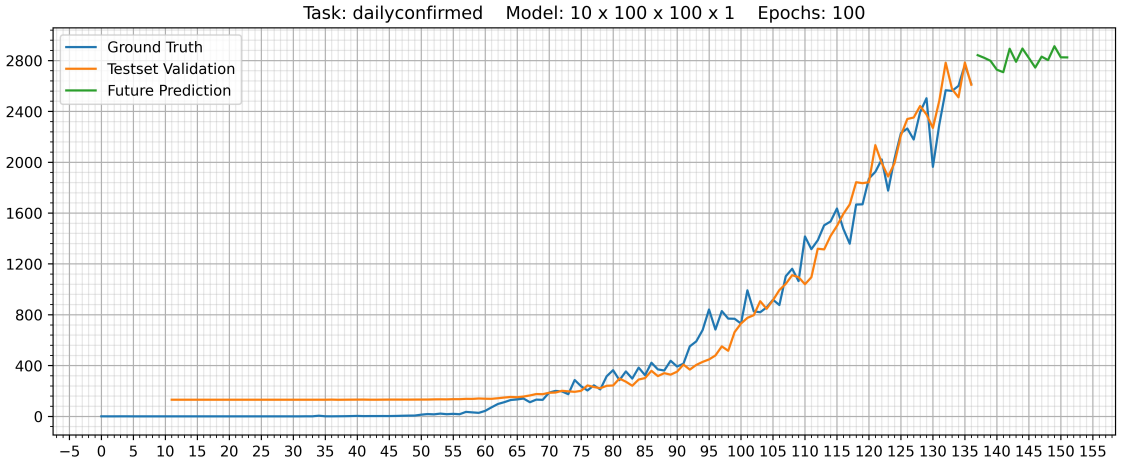


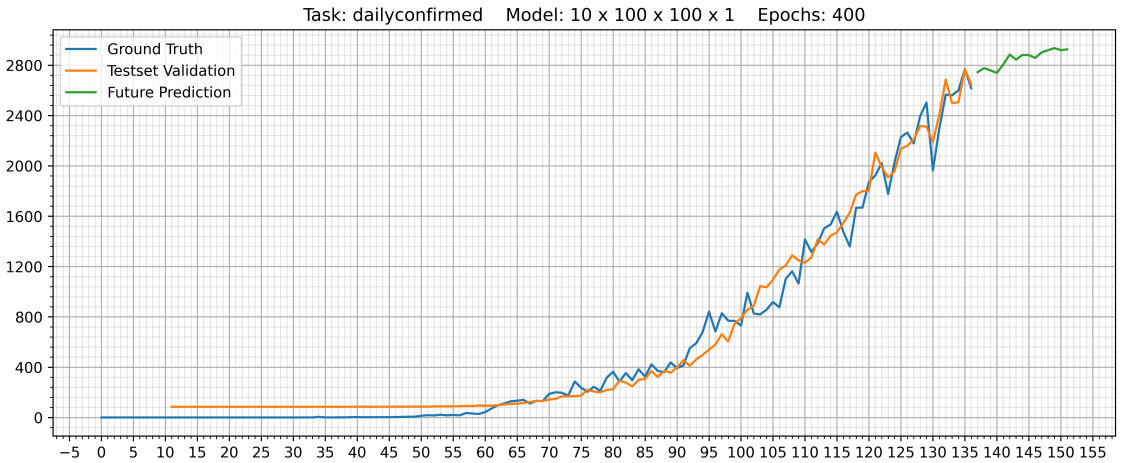


Deep model -

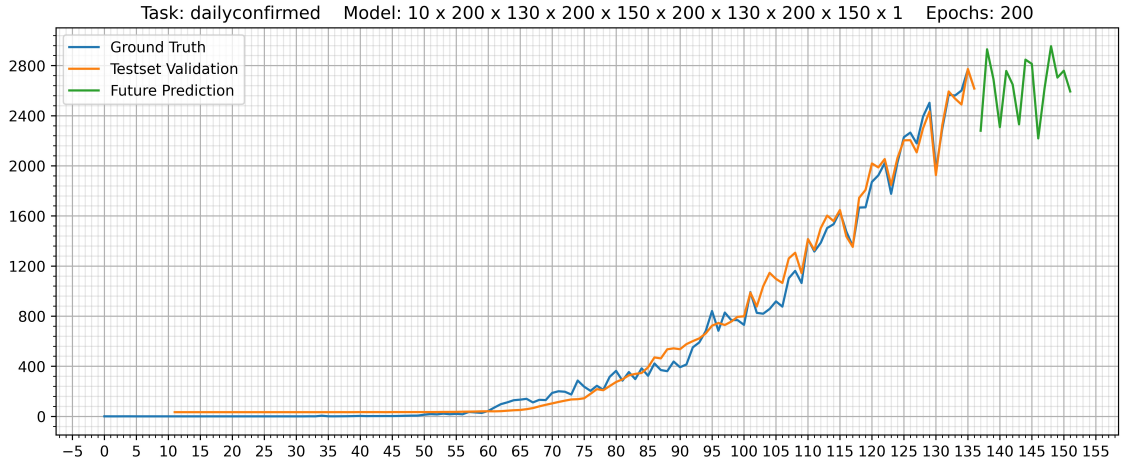


Graphs of ML model predicting infection rate in India in the age group 35-50 till 31st of July 2020.

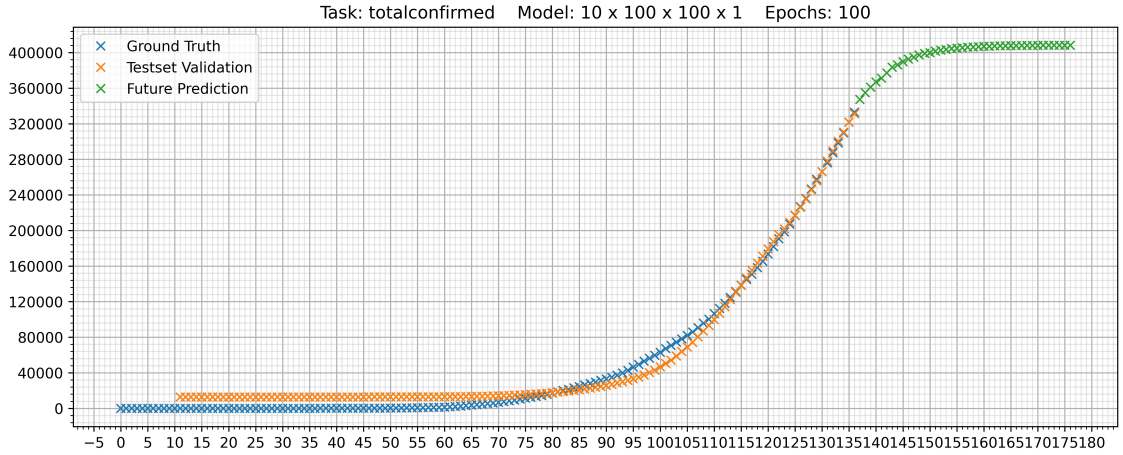




Deep model –



Objective 4 -: Using information obtained from the above models in this and the previous assignment and other data available on the link predict the tentative date (or interval spanning a week) after which the infection spread is 25% of its peak value in the world and in India.



The above graph shows the prediction for total confirmed cases till 10 August for India.

The prediction says, the number of confirmed cases may reach a saturation near 2 August, at a value just above 4,12,000 confirmed cases.

Considering the effects by government, it can be said, that after this saturation, the graph can fall down; a decrease in number of confirmed cases.

Assuming that, the nature of decrease is similar to that of increasing (as before 31July), (flipping the graph around x=2 August), the expected time-period when confirmed cases become 25% of that at 31 July (4,12,000) = 412000 * 0.25 = 103000, will be, somewhere between **7 October 2020 - 14 October 2020**