ECE 657A: Data and Knowledge Modelling and Analysis

Assignment 2: Classification with Deep Learning on Covid-19 and FashionMNIST Datasets

CM1 (Data Pre-processing and Preparation)

Code Script:

```
df_covid= pd.read_csv("C:/Users/SAM/OneDrive - University of Waterloo/Spring 2021/ECE657A/Assignment 2/dkmacovid_train.csv")

df_covid_sorted = df_covid.sort_values(by=["State ID","Day"])
df_covid_sorted[['Confirmed', 'Deaths', 'Recovered']] = df_covid_sorted[['Confirmed', 'Deaths', 'Recovered']].astype(int)
df_covid_sorted = df_covid_sorted.round(2)

df_covid_sorted = df_covid_sorted.drop(['State'], axis=1)

df_covid_sorted['Resident Population 2020 Census']=df_covid_sorted['Resident Population 2020 Census'].str.replace(',','')
df_covid_sorted['Resident Population 2020 Census'] = df_covid_sorted['Resident Population 2020 Census'].astype(int)
df_covid_sorted['Population Density 2020 Census']=df_covid_sorted['Population Density 2020 Census'].str.replace(',','')

# Seperating the dependent and independent variables
X = df_covid_sorted.drop(['Confirmed', 'Deaths', 'Recovered'],axis=1)
y = df_covid_sorted.loc[:,['Confirmed', 'Deaths', 'Recovered']]

# declairing the y variable one for each label.
y_confirmed = y.loc[:,['Confirmed']]
y_deaths = y.loc[:,['Deaths']]
y_recovered = y.loc[:,['Recovered']]

#normalizing the training data before feeding it into the PCA and LDA algorithms.
sc = StandardScaler()
X = sc.fit_transform(X)
```

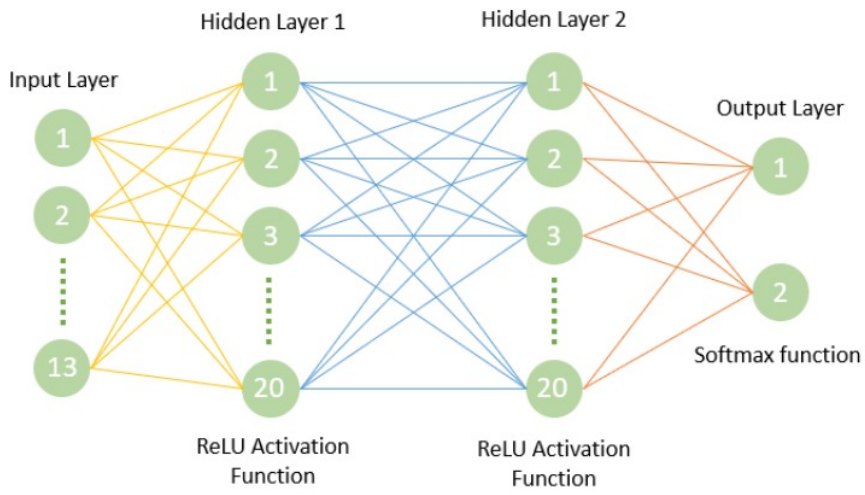
- We find the dataset to have sixteen columns which are a mixture of different data types. The three target features, i.e., our labels are of Boolean type which we will convert into a binary format (1 for TRUE and 0 for FALSE) further, for use in our ML models.
- No duplicate or missing values were found in our dataset.
- The data was firstly sorted by State IDs and Days in an ascending order for better understanding of the COVID cases for each state for a period of 30 days.
- We also rounded off the decimal values of the features to their second decimal place for reducing the complexity of our values.
- Further noise reduction is done by the Z score normalizing technique, as normalizing helps reduce any unwanted noise while preserving the original data.
- It is important in our case to perform normalization as it helps standardize the various features in our dataset, especially the population features which vary immensely for our states. The Active cases feature is another which required to be scaled as it varies significantly state wise and day wise.
- The conversion is done for the Resident Population and Population density features, to convert them into *int* and *float* data type for enabling our ML models to interpret the data.

ECE 657A: Data and Knowledge Modelling and Analysis

Assignment 2: Classification with Deep Learning on Covid-19 and FashionMNIST Datasets

CM2 (Default Network)

Model Structure:



Code Scripts with Explanation:

1. Libraries used are as follows:
    - Pandas
    - NumPy
    - Matplotlib
    - Scikit Learn
    - Keras

```
import pandas as pd
import numpy as np
from numpy import arange
from numpy import isnan
from matplotlib import pyplot as plt
from sklearn import preprocessing
from sklearn import metrics
from sklearn.metrics import accuracy_score
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.model_selection import train_test_split, cross_val_score
import warnings
warnings.filterwarnings('ignore')
from numpy import loadtxt
from keras.models import Sequential
from keras.layers import Dense
```

2. Splitting the data:

- We first separate the independent and independent variable.
- The Independent variables are all the features excluding the target labels.
- The dependent variables are the labels.
- Therefore, there are total **13 features and 3 labels**.
- Then we assign the target labels to their own variables in the code.
- We are doing this because we are training **3 separate model**, which are identical to each other, one for each label 'Confirmed', 'Deaths' and 'Recovered'.
- We then divide the features and each label into training and testing sets of 90% and 10% each.
- We also converted each split label (y_confirmed, y_deaths, y_recovered) into one hot encode data. We did this step because we want to have one unique binary value for each class.

```
# Seperating the dependent and independent variables
X = df_covid_sorted.drop(['Confirmed', 'Deaths', 'Recovered'],axis=1)
y = df_covid_sorted.loc[:,['Confirmed', 'Deaths', 'Recovered']]

# declairing the y variable one for each label.
y_confirmed = y.loc[:,['Confirmed']]
y_deaths = y.loc[:,['Deaths']]
y_recovered = y.loc[:,['Recovered']]

#normalizing the training data before feeding it into further algorithm.
sc = StandardScaler()
X = sc.fit_transform(X)
```

```
#One hot encoding the labels
ohe = OneHotEncoder()
yy = y_confirmed
yy = ohe.fit_transform(yy).toarray()
X_train,X_test,y_train,y_test = train_test_split(X,yy,test_size = 0.1, random_state = 98)
```

3. Defining the model:

- We are defining the models in Keras as a sequence of layers.
- We are creating sequential model and add layers one at a time as show in the code snippet below.
- Since we have 13 features, therefore we have 13 nodes in the input layer which is set with the **input_dim** argument.
- In this model we are asked to have 2 hidden and one (obvious) output layer.
- Both the hidden layers have 20 nodes each and the output has 2 nodes since there are two classes.
- we will use a fully connected network structure
- Fully connected layers are defined using the **Dense class**. We can specify the number of neurons or nodes in the layer as the first argument and specify the activation function using the activation argument.

- We will use the **Rectified linear unit activation** function referred to as ReLU on the two hidden layers and the **Softmax** function in the output layer, as directed in the instructions.
- The shape of the input to the model is defined as an argument on the first hidden layer. This means that the line of code that adds the first Dense layer is doing 2 things, defining the input or visible layer and the first hidden layer.

```
# define the keras model
model = Sequential()
model.add(Dense(20, input_dim=13, activation='relu'))
model.add(Dense(20, activation='relu'))
model.add(Dense(2, activation='softmax'))
```

4. Compiling the Keras Model:

- Now that the model is defined, we can compile it.
- Compiling the model uses the efficient numerical libraries in the backend such as Theano or TensorFlow. The backend automatically chooses the best way to represent the network for training and making predictions to run on your hardware, such as CPU or GPU or even distributed.
- We must specify some additional properties.
- We must specify the loss function to use to evaluate a set of weights. Here we will use cross entropy as the **loss** argument. This loss is for a binary classification problem and is defined in Keras as "**binary_crossentropy** ".
- We will define the **optimizer** as the efficient stochastic gradient descent algorithm "**Adam** ". This is a popular version of gradient descent because it automatically tunes itself and gives good results in a wide range of problems. Adam combines the best properties of the AdaGrad and RMSProp algorithms to provide an optimization algorithm that can handle sparse gradients on noisy problems.
- Metrics is used to specify the way we want to judge the performance of our neural network. Here we have specified it to **accuracy**.

```
#Compile the model
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
```

5. Fitting the Keras Model.

- We have defined our model and compiled it ready for efficient computation.
- Now we will execute the model on the data.
- We will train our model on our loaded data by calling the **fit ()** function on the model.
- Training occurs over epochs and each epoch is split into batches.
- We want to train the model enough so that it learns a good (or good enough) mapping of rows of input data to the output classification.

- We fit the model separately for each label. In the below code snippet, we have fit the model for the Confirmed label. We have implemented different models for different labels in our implementations.
- The fit function also has the feature of implementation of the validation set. Here we have split the training data into 80-20% as training-validation data.

```
# Fitting the model.
history_confirmed = model.fit(X_train, y_train, validation_split=0.20, epochs=100, batch_size=10)
```

6. Prediction and performance evaluation for the testing data.

- We can predict the labels for the testing data by using **predict ()** function.
- The resulted predictions are the probabilities in the range between 0 and 1. We converted these predictions into crisp binary predictions for this classification.
- We then converted the one hot encoded labels of the testing data to normal labels.
- We used **accuracy_score ()** function for the calculations of the performance of the model.
- Below we are showing the code for the confirmed label. We performed the same steps for the modeling of other labels as well.
- We used the **matplotlib.pyplot** to plot the graphs for the training and validations curves for the accuracy and loss for each epochs.
- Below is a code snippet shown for the confirmed label.

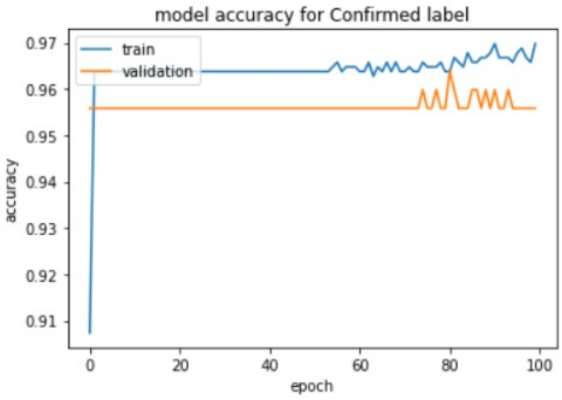
```
# summarize history for accuracy
plt.plot(history_confirmed.history['accuracy'])
plt.plot(history_confirmed.history['val_accuracy'])
plt.title('model accuracy for Confirmed label')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train', 'validation'], loc='upper left')
plt.show()
# summarize history for loss
plt.plot(history_confirmed.history['loss'])
plt.plot(history_confirmed.history['val_loss'])
plt.title('model loss Confirmed label')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'validation'], loc='upper left')
plt.show()
```

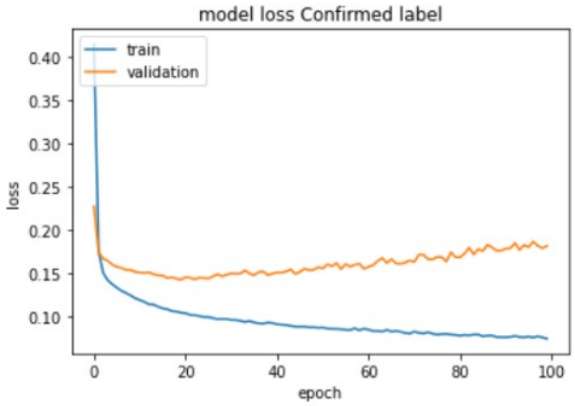
7. Results:

- *Confirmed Label:*

I. Output of last 5 epochs after fitting the model
II. Accuracy-epoch and Loss-epoch plots
III. Testing accuracy

```
Epoch 95/100
100/100 [==============================] - 0s 3ms/step - loss: 0.0754 - accuracy: 0.9658 - val_loss: 0.1823 - val_accurac
y: 0.9558
Epoch 96/100
100/100 [==============================] - 0s 3ms/step - loss: 0.0762 - accuracy: 0.9678 - val_loss: 0.1799 - val_accurac
y: 0.9558
Epoch 97/100
100/100 [==============================] - 0s 3ms/step - loss: 0.0752 - accuracy: 0.9688 - val_loss: 0.1864 - val_accurac
y: 0.9558
Epoch 98/100
100/100 [==============================] - 0s 3ms/step - loss: 0.0766 - accuracy: 0.9668 - val_loss: 0.1814 - val_accurac
y: 0.9558
Epoch 99/100
100/100 [==============================] - 0s 3ms/step - loss: 0.0753 - accuracy: 0.9658 - val_loss: 0.1787 - val_accurac
y: 0.9558
Epoch 100/100
100/100 [==============================] - 0s 3ms/step - loss: 0.0738 - accuracy: 0.9698 - val_loss: 0.1813 - val_accurac
y: 0.9558
```



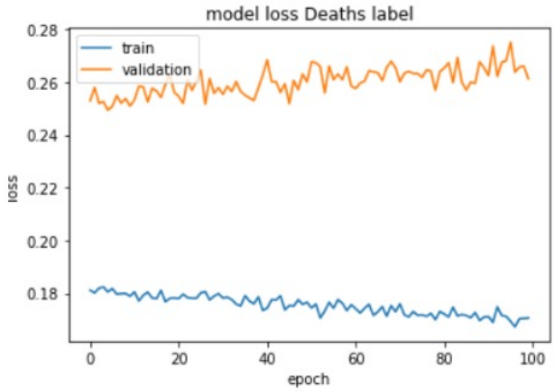


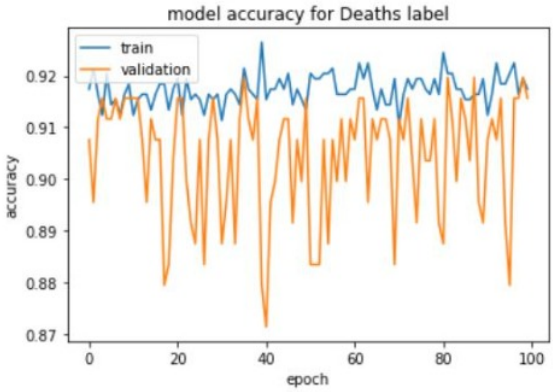

```
testing Accuracy for Confirmed label is: 97.10144927536231
```

- *Deaths Label:*

IV. Output of last 5 epochs after fitting the model
V. Accuracy-epoch and Loss-epoch plots
VI. Testing accuracy

```
Epoch 95/100
100/100 [==============================] - 0s 4ms/step - loss: 0.1825 - accuracy: 0.9154 - val_loss: 0.2542 - val_accurac
y: 0.8835
Epoch 96/100
100/100 [==============================] - 0s 3ms/step - loss: 0.1845 - accuracy: 0.9114 - val_loss: 0.2509 - val_accurac
y: 0.8795
Epoch 97/100
100/100 [==============================] - 0s 3ms/step - loss: 0.1839 - accuracy: 0.9094 - val_loss: 0.2567 - val_accurac
y: 0.9076
Epoch 98/100
100/100 [==============================] - 0s 3ms/step - loss: 0.1841 - accuracy: 0.9184 - val_loss: 0.2516 - val_accurac
y: 0.9157
Epoch 99/100
100/100 [==============================] - 0s 3ms/step - loss: 0.1838 - accuracy: 0.9104 - val_loss: 0.2527 - val_accurac
y: 0.9076
Epoch 100/100
100/100 [==============================] - 0s 3ms/step - loss: 0.1815 - accuracy: 0.9094 - val_loss: 0.2496 - val_accurac
y: 0.9076
```



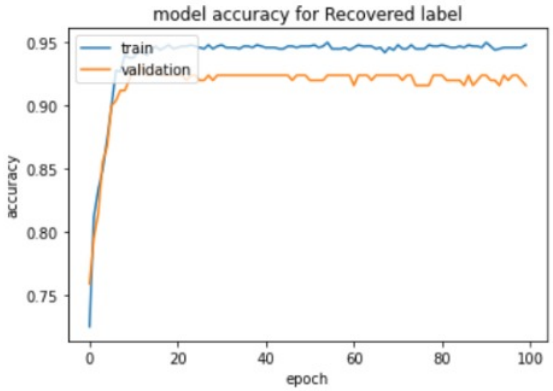


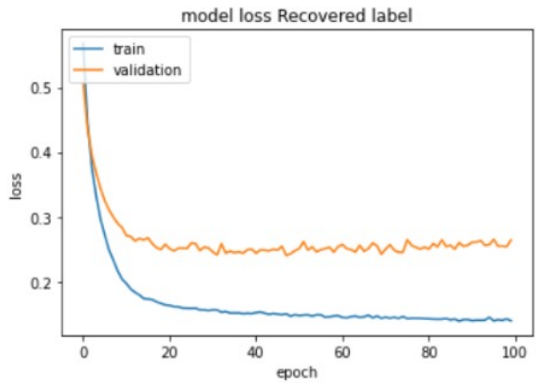

```
testing Accuracy for Deaths label is: 90.57971014492753
```

- *<u>Recovered Label:</u>*

VII. Output of last 5 epochs after fitting the model
VIII. Accuracy-epoch and Loss-epoch plots
IX. Testing accuracy

```
Epoch 95/100
100/100 [==============================] - 0s 3ms/step - loss: 0.1448 - accuracy: 0.9456 - val_loss: 0.2569 - val_accurac
y: 0.9237
Epoch 96/100
100/100 [==============================] - 0s 3ms/step - loss: 0.1393 - accuracy: 0.9456 - val_loss: 0.2651 - val_accurac
y: 0.9197
Epoch 97/100
100/100 [==============================] - 0s 3ms/step - loss: 0.1411 - accuracy: 0.9456 - val_loss: 0.2547 - val_accurac
y: 0.9237
Epoch 98/100
100/100 [==============================] - 0s 3ms/step - loss: 0.1401 - accuracy: 0.9456 - val_loss: 0.2550 - val_accurac
y: 0.9237
Epoch 99/100
100/100 [==============================] - 0s 3ms/step - loss: 0.1423 - accuracy: 0.9456 - val_loss: 0.2540 - val_accurac
y: 0.9197
Epoch 100/100
100/100 [==============================] - 0s 3ms/step - loss: 0.1394 - accuracy: 0.9476 - val_loss: 0.2642 - val_accurac
y: 0.9157
```






```
testing Accuracy for Recovered label is: 94.20289855072464
```

8. Analysis:

- We observed that the training and validation accuracies is coming highest for the Confirmed label (approx. 96%) and intermediate for Recovered (approx. 94%) and lowest for the Deaths label (approx. 89%).
- According to us the Confirmed label is experiencing Overfitting and that is why it is showing the highest accuracy. This, according to our understanding, is happening because the Confirmed label is most unbalanced label among all the three labels.
- The same happened with the Deaths label, except for the fact that it did not experience overfitting, hence the poorest performance among all the three labels.
- The Recovered label is showing the best performance because it did not fall prey to overfitting. It happened because it is the most balanced label among all the three labels.
- The similar patter is observed among the testing accuracy as it is observed for the training accuracies.
- The accuracy plot for the confirmed and recovered labels is showing decent behavior. The validation accuracy is lower that the training accuracy. And it increases with increase of number of epochs.
- The loss plot for the confirmed and recovered labels is showing the desired trend with the increase of number of epochs but only till a limit. Then the validation loss starts increasing but the training loss continues to decrease. According to our understanding and learnings this behavior is coming due to the unbalanced nature of these labels.
- The accuracy and loss plots for the deaths label is showing undesirable results. The validation accuracy is coming way bigger that the training accuracy and the loss for the validation accuracy is fluctuating very much unlike the training loss plot.
- Rest of the analysis is done in CM4.

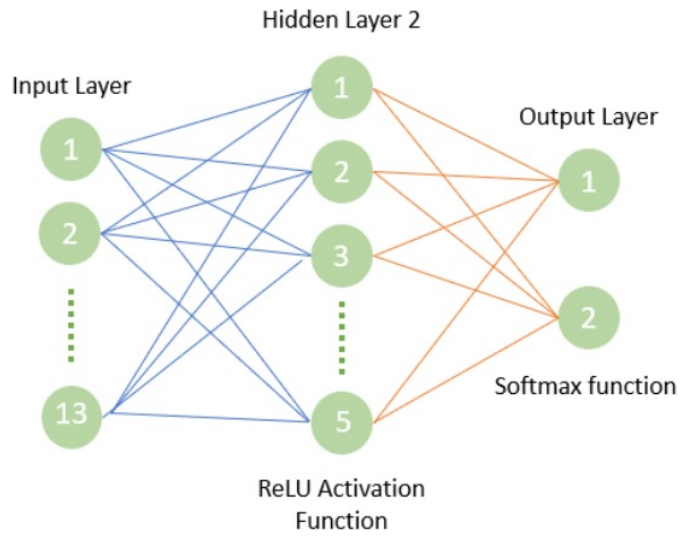
ECE 657A: Data and Knowledge Modelling and Analysis

Assignment 2: Classification with Deep Learning on Covid-19 and FashionMNIST Datasets

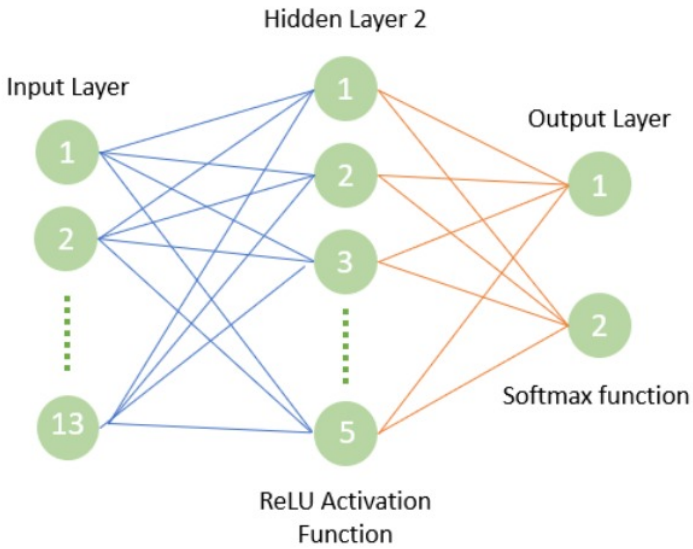
CM3 (Our Own Network)

Model Structure:

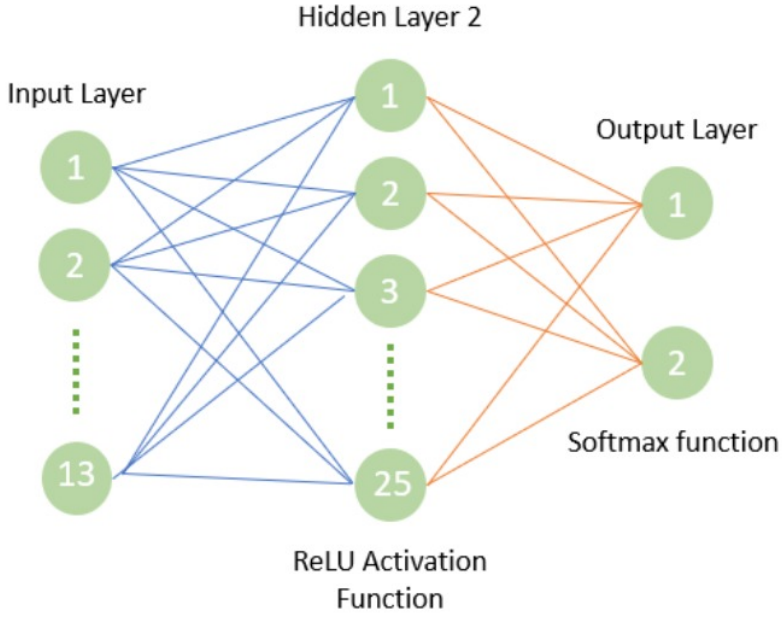
Confirmed Label



Deaths Label



Recovered Label



The model structure is designed as below.

- We are using the back propagation fully connected deep neural network here instead of just feed forward network.
- There are 3 different models for different labels.
- We have implemented everything from scratch and on our own without using existing libraries like TensorFlow, Keras, or PyTorch.
- Here we are using one input layer, one hidden layer, and one output layer in our implementation.
- The labels are one-hot encoded just like the previous model.
- We used 5 nodes for Confirmed and Deaths labels models each in their hidden layers.
- We used 25 nodes for the Recovered label model in its hidden layer.
- Each model has 13 nodes in input layer and 2 nodes in output layer.
- The hidden layer in each model used ReLU activation function
- The output layer uses Softmax function
- In addition to these variations, we have also used L2 regularization technique to deal with the overfitting problem.
- We have implemented each of the above-mentioned techniques and features from scratch by ourselves.

Code Scripts with Explanation:

1. Defining some required functions:

- We are selecting Rectified Linear Unit (ReLU) function as the activation function for getting the outputs at the hidden layer. It is a simple function that gives output = 0 for any negative values of input and output = input for values greater than 0.

```
def ReLU_activation_function(a):
    return np.maximum(a, 0)
```

- In the case of multi-class classification, we represent the choice of class as a probability distribution over other possible class. Therefore, we need to transform the output layer. We can do this by using Softmax function as an activation function of out output layer. Also, the derivative of Softmax function is computationally easy and cheap to calculate.

```
def softmax_activation_function(b):
    return np.exp(b) / np.sum(np.exp(b), axis = 1, keepdims = True)
```

- In the algorithm of backpropagation , an important step is to find the loss function to calibrate the future weight and bias vectors. Since we are using Softmax function as an activation function for output layer, we will calculate the loss using cross entropy loss function.

```
def cross_entropy_loss(probability_arr, y_out):
    indices = np.argmax(y_out, axis = 1).astype(int)
    pred_probability = probability_arr[np.arange(len(probability_arr)), indices]
    log_preds = np.log(pred_probability)
    loss = -1.0 * np.sum(log_preds) / len(log_preds)
    return loss
```

- To avoid overfitting in predictive modeling, we will use L2 regularization. this will have the weights come closer to 0. it is always advised to use regularization for complex problems.

```
def L2_regularization(reg_lambda, w1, w2):
    w1_loss = 0.5 * reg_lambda * np.sum(w1 * w1)
    w2_loss = 0.5 * reg_lambda * np.sum(w2 * w2)
    return w1_loss + w2_loss
```

- Our very own function for measuring the accuracy score.

```
def accuracy(predictions, labels):
    preds_correct_boolean =  np.argmax(predictions, 1) == np.argmax(labels, 1)
    correct_predictions = np.sum(preds_correct_boolean)
    accuracy = correct_predictions / predictions.shape[0]
    return accuracy
```

2. One Hot encoding

- We changed yy = y_deaths or y_confirmed for the Deaths and Confirmed labels respectively and RERUN the entire model for correct scores.

```
ohe = OneHotEncoder()
yy = y_recovered
yy = ohe.fit_transform(yy).toarray()
x_train,x_test,y_train,y_test = train_test_split(X,yy,test_size = 0.10, random_state = 98)
```

3. Initializing few variables before we go to model training.

- Here we initialize few variables like number of features, number of nodes in hidden layer, number of classes, learning rate, reg-lambda.
- They are initialized as follows.

```
np.random.seed(12)
# number of different features to decide the number of nodes in input layer (LAYER 0).
num_feat = x_train.shape[1]
# number of nodes in the hidden layer (LAYER 1).
hidden_nodes = 5
# number of different classes to decide the number of nodes in output layer (LAYER 2). Here, 4. We want
num_class = y_train.shape[1]
#learning rate
lrng_rate = .01
reg_lmbda = .01
```

4. Initializing the starting weight and bias vectors for hidden and output layers each.

- we have to multiply the hidden layer/layer 1 weights to the input data. Therefore (num_feat X hidden_nodes) will become the dimension of the layer 1 weight vector.
- we have to multiply the output layer/layer 2 weights to the output of hidden layer. Therefore (hidden_nodes X num_class) will become the dimension of the layer 2 weight vector.
- The layer 1 bias has to be added up to the input layer, so it will have a dimension of (1 X hidden_nodes)
- Same happens with the bias vector of output layer.

```
l1_w = np.random.normal(0, 1, [num_feat, hidden_nodes])
l2_w = np.random.normal(0, 1, [hidden_nodes, num_class])
l1_b = np.zeros((1, hidden_nodes))
l2_b = np.zeros((1, num_class))
```

5. Model Training:

- We will be training the model again and again to decrease the loss and increase the accuracy. Each time we train the entire model, we will call it an epoch.
- Since we are using 2 different activation functions for the hidden and output layers, we will divide the feed-forward phase into 2 phases
- This process involves the Feed Forward section of the algorithm.

```
## Phase 1: Calculating output of the hidden layer.
ip_layer = np.dot(x_train, l1_w)
hidden_layer = ReLU_activation_function(ip_layer + l1_b)

## Phase 2: Calculating output of the output layer.
op_layer = np.dot(hidden_layer, l2_w) + l2_b
op_probs = softmax_activation_function(op_layer)
```

- The process now onwards is the beginning of Backpropagation section of the algorithm.
- We calculate the loss at output layer with cross_entropy_loss function defined earlier and then we regularize it. For details, please refer to respective function definitions.

```
loss = cross_entropy_loss(op_probs, y_train)
loss += L2_regularization(reg_lmbda, l1_w, l2_w)
```

- We will use the softmaxed output of the network and labels array to calculate the error signal at the output. Since the model's loss is an average of all the observation's losses, we need to divide it by number of observations to find the actual error signal.

```
op_error = (op_probs - y_train) / op_probs.shape[0]
```

- Using op_error calculated above we can get error signal at hidden layer. since the derivative of ReLU functions is 1 when the value is greater than 0 and otherwise, we updated the error signal at hidden layer to be 0 for the values less than 0 and leaving the rest as it is for those greater than 0.

```
hidden_error = np.dot(op_error, l2_w.T)
hidden_error[hidden_layer <= 0] = 0
```

- Then we continue by calculating the gradient on layer 2 weight and biases.

```
gradient_l2_w = np.dot(hidden_layer.T, op_error)
gradient_l2_b = np.sum(op_error, axis = 0, keepdims = True)
```

- Then we do the same thing for the hidden layer only this time error signal will be hidden_error for input values as x_train.

```
gradient_l1_w = np.dot(x_train.T, hidden_error)
gradient_l1_b = np.sum(hidden_error, axis = 0, keepdims = True)
```

- Now we start adding regularization contribution to respective gradients for each weight

```
gradient_l2_w += reg_lmbda * l2_w
gradient_l1_w += reg_lmbda * l1_w
```

- We are ready to update the weights and biases. we are subtraction the gradients times the earning rate from weights and biases because we want to move in negative gradient direction.

```
l1_w -= lrng_rate * gradient_l1_w
l1_b -= lrng_rate * gradient_l1_b
l2_w -= lrng_rate * gradient_l2_w
l2_b -= lrng_rate * gradient_l2_b
```

- Here we complete the Backpropagation section if the algorithm.
- Now we print the loss for each epoch and fill in the loss and accuracy list which will be used in for plotting the accuracy and loss plots. lesser the loss, greater will be the accuracy of the model.

```
if epoch % 500 == 0:
    print('Epoch {0} - Loss = {1}'.format(epoch, loss))
    Loss.append(loss)
    input_layer = np.dot(x_test, l1_w)
    hidden_layer = ReLU_activation_function(input_layer + l1_b)
    scores = np.dot(hidden_layer, l2_w) + l2_b
    probs = softmax_activation_function(scores)
    a = accuracy(probs, y_test)
    acc.append(a)
```

6. Plotting the graphs

```
# summarize model for accuracy
plt.plot(epoch,acc)
plt.title('model accuracy for Deaths label')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.show()

# summarize model for loss
plt.plot(epoch,Loss)
plt.title('model loss for Deaths label')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.show()
```

7. Results:

I. Results and the analysis are done in CM4.

ECE 657A: Data and Knowledge Modelling and Analysis

Assignment 2: Classification with Deep Learning on Covid-19 and FashionMNIST Datasets

CM4 (Results Analysis)

1. Runtime performance for training and testing on both networks for all three labels:

    - Runtime performance for the Default model was slower than our own backpropagation deep neural network for both the training and testing sets.
    - This faster performance came despite the huge number of epochs in our model (50000 epochs) as compared to the default network (100 epochs).
    - Our model is performing better even with far more number from epochs.
    - Now between the training and testing runtimes of the Default network, the training section took more time than the testing section because the training data and training method is much more than the testing section.
    - Same is the case with our deep neural model.

2. Comparisons of different algorithms we tried:

    - We tried increasing and decreasing the number of nodes in the default network i.e., 30 nodes in each hidden nodes and found out that the model was underperforming for all the 3 labels. The weights stopped changing after certain epochs and some weights started becoming zero. We immediately knew that we needed to change the design of our model.
    - After selecting Backpropagation for our deep neural network, we tried running one with a single input, hidden and output layer.
    - We tried running the model with only 5 nodes in the hidden layer and the model performed decently.
    - Upon increasing the number of nodes in the hidden layer for all the three label models separately we found that the accuracy for recovered label was increasing with increase of number of nodes in hidden layer. But this was not the case for always. We found that the accuracy for the recovered label increased only till we increased the nodes in hidden layer till 25. As we started increasing the hidden nodes future from 25, the accuracy started decrease and model started underperforming.
    - Interestingly this was not the case with the confirmed and deaths labels. Their models performed best with only 5 nodes in the hidden layer. As we increased the number of nodes in the hidden layer, there was not difference in the performance of the models for Confirmed and Deaths label as such. Moreover, the runtime started decreasing as we increased the nodes in hidden layer only to give little to no improvement in the model performance.

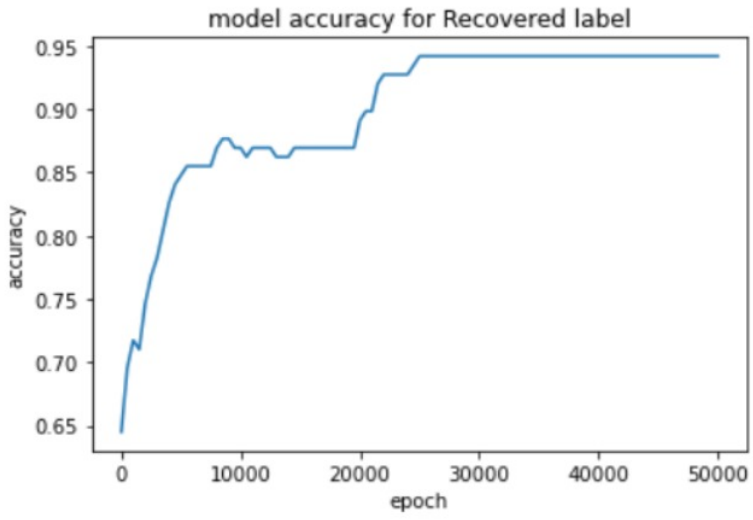
- With the above design we also tried increasing the epochs in the efforts to increase the model performances but found out that the performance was indifferent to this design change.
- at last, we found out that there is a need to deal with the imbalance nature of the two unbalanced labels, Confirmed and Deaths labels. Therefore, we decided to introduce our model with the L2 Regularization. After implementing the L2 Regularization the model was found to be performing drastically well than the previous versions.

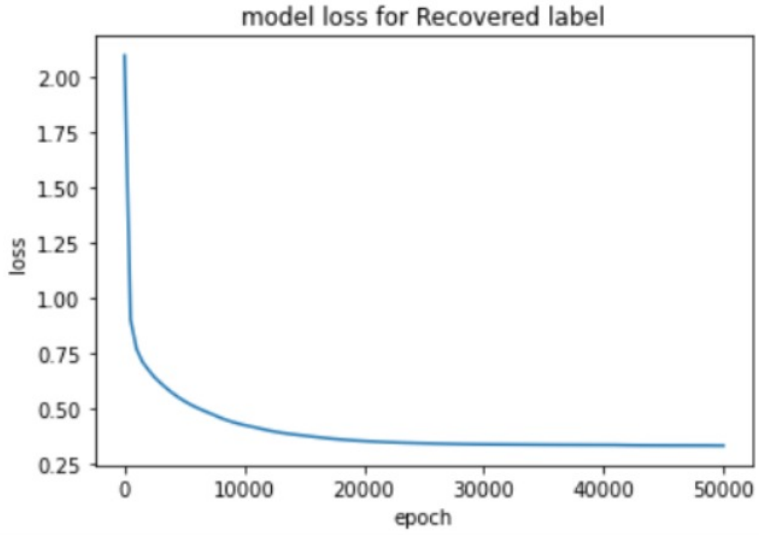
3. Performance evaluation.

- The results of all the three labels are shown below. They contain:
  I. Output of last few epochs.
  II. classification accuracy vs. training epoch curve
  III. training loss vs. training epoch curve

Recovered label

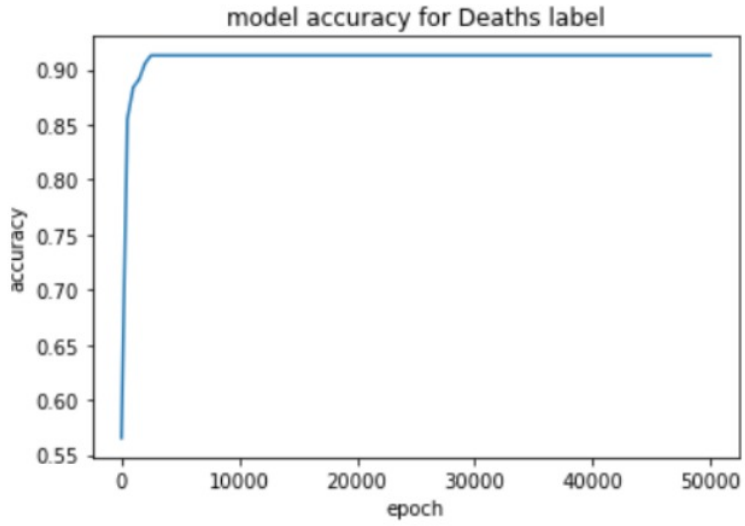
```
Epoch 44000 - Loss = 0.3361131213579313
Epoch 44500 - Loss = 0.33600889599970046
Epoch 45000 - Loss = 0.33592457345338655
Epoch 45500 - Loss = 0.33585440168138025
Epoch 46000 - Loss = 0.3357954122021926
Epoch 46500 - Loss = 0.3357453365369627
Epoch 47000 - Loss = 0.33570189790806076
Epoch 47500 - Loss = 0.33566413351447216
Epoch 48000 - Loss = 0.33563184582277616
Epoch 48500 - Loss = 0.33560411379198585
Epoch 49000 - Loss = 0.33531307927444376
Epoch 49500 - Loss = 0.3350428799867842
Epoch 50000 - Loss = 0.334832585354094
```

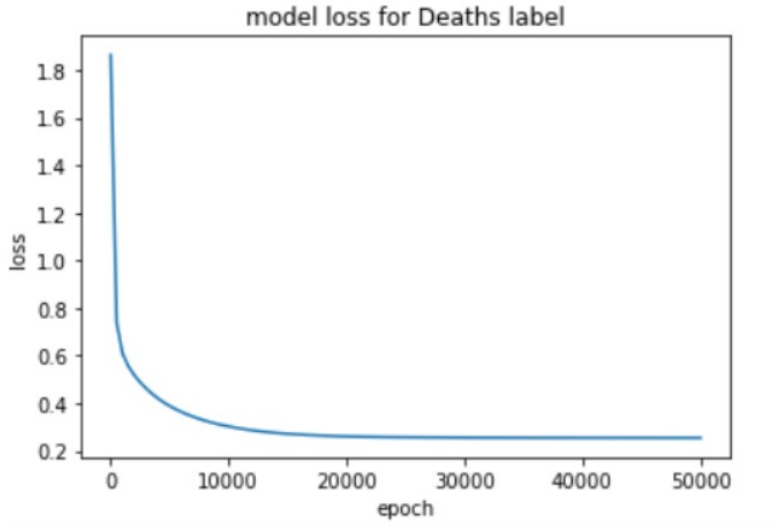




Deaths Label

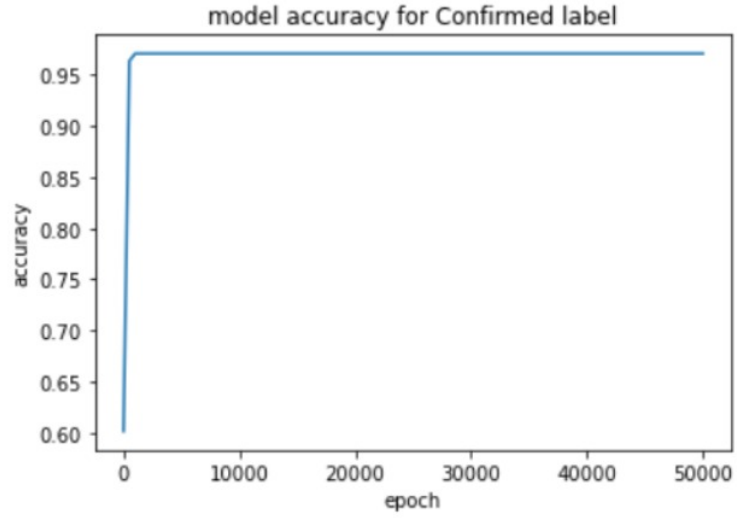
```
Epoch 43000 - Loss = 0.25503981509333756
Epoch 43500 - Loss = 0.2550229628922029
Epoch 44000 - Loss = 0.25500632423919195
Epoch 44500 - Loss = 0.25499056892321886
Epoch 45000 - Loss = 0.2549756440747493
Epoch 45500 - Loss = 0.254961672942263
Epoch 46000 - Loss = 0.25494795738030185
Epoch 46500 - Loss = 0.25493475047619285
Epoch 47000 - Loss = 0.25492190558378847
Epoch 47500 - Loss = 0.2549094544073037
Epoch 48000 - Loss = 0.2548973851730806
Epoch 48500 - Loss = 0.2548855188119254
Epoch 49000 - Loss = 0.25487392278942766
Epoch 49500 - Loss = 0.2548623200739291
Epoch 50000 - Loss = 0.25485067718695736
```

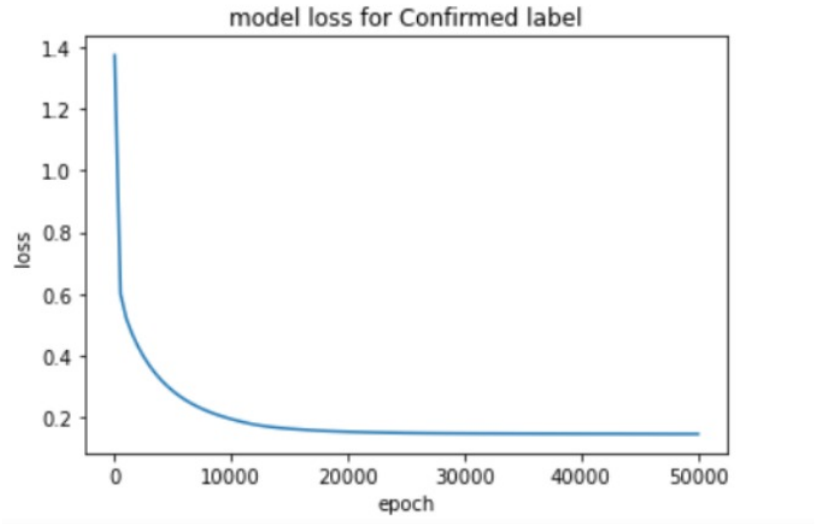




Confirmed Label:

```
Epoch 44500 - Loss = 0.14640615115463793
Epoch 45000 - Loss = 0.146354980059555
Epoch 45500 - Loss = 0.14630949605696167
Epoch 46000 - Loss = 0.14626652793112802
Epoch 46500 - Loss = 0.14622494046488746
Epoch 47000 - Loss = 0.14618341658921336
Epoch 47500 - Loss = 0.14614902502694233
Epoch 48000 - Loss = 0.14611862702937364
Epoch 48500 - Loss = 0.14609169563007032
Epoch 49000 - Loss = 0.14606782472261387
Epoch 49500 - Loss = 0.1460461508882249
Epoch 50000 - Loss = 0.14602662420334195
```





IV. Final testing accuracies of the three models are as follows.
- Recovered Label: 95.18%
- Deaths label: 92.36%
- Confirmed Label: 97.01%

V. These accuracies can be different for different runs or multiple restarts of the model.

VI. Here we can observe that the accuracy of all the labels improved, and loss continues to decrease with each epoch

VII. We used a greater number of epochs and found that the model got better with every epoch.

VIII. We tried running 50000 thousand epochs.

IX. Compared to the last model our current model seems to perform comparatively better under the threat of overfitting and avoids it better that the last model.

X. We used many new techniques like L2 regularization, backpropagation etc.