Dataset - Covid-19 Outcomes in Ontario

This dataset is about the confirmed COVID-19 cases in Ontario. The dataset has 14860 datapoints (rows) and 12 features. The dataset includes the features age group, gender, case acquisition info, city, outbreak, latitude, and longitude. It also has the date features. The categorical features have been converted to one-hot encoded features. The feature Outcome1 is the label.

Pre-processing:

- Dropping rows with NaN values in the 'Age_Group' column
- Replacing NaN values in the 'Outbreak_Related' column with value 'No'
- Replacing NaN values in the 'Test_Reported_Date' column with value in the 'Case_Reported_Date' column. It was observed that for 85 % of the rows (datapoints) in the dataset, these two columns had the same value.
- Replacing NaN values in the 'Specimen_Date' column with value in the 'Accurate_Episode_Date' column. It was observed that for 50 % of the rows (datapoints) in the dataset, these two columns had the same value.
- One-hot encoding the categorical features.
- Ordinal encoding the 'Age_Group' feature.
- Converting dates to integer format and scaling down numerical features using Robust scalar. Three different scalers were tried. Robust scalar performed the best whereas Min-Max scalar performed the worst. The accuracy for Robust scalar and Standard scalar were similar, slightly higher for Robust.

Train - Val - Test Split:

Splitting the data into Training, Validation and Test set in the ratio 80-10-10 % with random state 0.

[CM 1] Design and Implementation Choices of your Model

5 models with different architectures and parameters were implemented and their performances compared.

Model 0: Fully Connected Neural Network – 3 Hidden Layers

- **Fully Connected** Input Layer (32 units) - Activation - **ReLU**
- 2 **Fully Connected** Hidden Layers (32 units) - Activation - **ReLU**
- 1 **Fully Connected** Hidden Layer (16 units) - Activation - **ReLU**
- **Fully Connected** Output Layer (3 units) - Activation - **Softmax**
- Adam Optimizer

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense (Dense)                (None, 32)                1600
_________________________________________________________________
dense_1 (Dense)              (None, 32)                1056
_________________________________________________________________
dense_2 (Dense)              (None, 32)                1056
_________________________________________________________________
dense_3 (Dense)              (None, 16)                528
_________________________________________________________________
dense_4 (Dense)              (None, 3)                 51
=================================================================
Total params: 4,291
Trainable params: 4,291
Non-trainable params: 0
_________________________________________________________________
```

Model 1: Fully Connected Neural Network – Deeper - 5 Hidden Layers

- **Fully Connected** Input Layer (32 units) - Activation - **ReLU**
- 4 **Fully Connected** Hidden Layers (32 units) - Activation - **ReLU**
- 1 **Fully Connected** Hidden Layer (16 units) - Activation - **ReLU**
- **Fully Connected** Output Layer (3 units) - Activation - **Softmax**
- Adam Optimizer

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_5 (Dense)              (None, 32)                1600
_________________________________________________________________
dense_6 (Dense)              (None, 32)                1056
_________________________________________________________________
dense_7 (Dense)              (None, 32)                1056
_________________________________________________________________
dense_8 (Dense)              (None, 32)                1056
_________________________________________________________________
dense_9 (Dense)              (None, 32)                1056
_________________________________________________________________
dense_10 (Dense)             (None, 16)                528
_________________________________________________________________
dense_11 (Dense)             (None, 3)                 51
=================================================================
Total params: 6,403
Trainable params: 6,403
Non-trainable params: 0
_________________________________________________________________
```

Model 2: Fully Connected Neural Network – 3 Hidden Layers - With Batch Normalization and Dropout

- **Fully Connected** Input Layer (32 units) - Activation - **ReLU**
- 2 **Fully Connected** Hidden Layers (32 units) - Activation - **ReLU**
- 1 **Fully Connected** Hidden Layer (16 units) - Activation - **ReLU**
- **Fully Connected** Output Layer (3 units) - Activation - **Softmax**
- Adam Optimizer

```
Model: "sequential_2"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_12 (Dense)             (None, 32)                1600
_________________________________________________________________
batch_normalization (BatchNo (None, 32)                128
_________________________________________________________________
dropout (Dropout)            (None, 32)                0
_________________________________________________________________
dense_13 (Dense)             (None, 32)                1056
_________________________________________________________________
batch_normalization_1 (Batch (None, 32)                128
_________________________________________________________________
dropout_1 (Dropout)          (None, 32)                0
_________________________________________________________________
dense_14 (Dense)             (None, 32)                1056
_________________________________________________________________
batch_normalization_2 (Batch (None, 32)                128
_________________________________________________________________
dropout_2 (Dropout)          (None, 32)                0
_________________________________________________________________
dense_15 (Dense)             (None, 16)                528
_________________________________________________________________
batch_normalization_3 (Batch (None, 16)                64
_________________________________________________________________
dropout_3 (Dropout)          (None, 16)                0
_________________________________________________________________
dense_16 (Dense)             (None, 3)                 51
=================================================================
Total params: 4,739
Trainable params: 4,515
Non-trainable params: 224
_________________________________________________________________
```

Batch normalization, as the name suggests, seeks to normalize the activations of a given input volume before passing it into the next layer. It has been shown to be effective at reducing the number of epochs required to train a CNN at the expense of an increase in per-epoch time.

Dropout is a form of regularization that aims to prevent overfitting. Random connections are dropped to ensure that no single node in the network is responsible for activating when presented with a given pattern.

An experimental test seems to suggest that ordering does matter. I ran the same network twice with only the batch norm and dropout reverse. When the dropout is before the batch norm, validation loss seems to be going up as training loss is going down. They're both going down in the other case.

RNN – A recurrent neural network (RNN) is a type of artificial neural network in which node connections form a directed graph along a time series. As a result, it may exhibit temporal dynamic behaviour. RNNs, which are derived from feedforward neural networks, can process variable length sequences of inputs by using their internal state (memory). While RNNs learn similarly while training, in addition, they remember things learnt from prior input(s) while generating output(s).

Model 3: Recurrent Neural Network – 3 Hidden Layers - With Batch Normalization and Dropout

- **SimpleRNN** Input Layer (32 units) - Activation - **ReLU**
- 2 **SimpleRNN** Hidden Layers (32 units) - Activation - **ReLU**
- 1 **SimpleRNN** Hidden Layer (16 units) - Activation - **ReLU**
- **Fully Connected** Output Layer (3 units) - Activation - **Softmax**
- Adam Optimizer

```
Model: "sequential_3"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
simple_rnn (SimpleRNN)       (None, 1, 32)             2624
_________________________________________________________________
dropout_4 (Dropout)          (None, 1, 32)             0
_________________________________________________________________
batch_normalization_4 (Batch (None, 1, 32)             128
_________________________________________________________________
simple_rnn_1 (SimpleRNN)     (None, 1, 32)             2080
_________________________________________________________________
dropout_5 (Dropout)          (None, 1, 32)             0
_________________________________________________________________
batch_normalization_5 (Batch (None, 1, 32)             128
_________________________________________________________________
simple_rnn_2 (SimpleRNN)     (None, 1, 32)             2080
_________________________________________________________________
dropout_6 (Dropout)          (None, 1, 32)             0
_________________________________________________________________
batch_normalization_6 (Batch (None, 1, 32)             128
_________________________________________________________________
simple_rnn_3 (SimpleRNN)     (None, 16)                784
_________________________________________________________________
dropout_7 (Dropout)          (None, 16)                0
_________________________________________________________________
batch_normalization_7 (Batch (None, 16)                64
_________________________________________________________________
dense_17 (Dense)             (None, 3)                 51
=================================================================
Total params: 8,067
Trainable params: 7,843
```

```
Non-trainable params: 224
_________________________________________________________________
```

LSTM - Long Short-Term Memory (LSTM) networks are a type of recurrent neural network capable of learning order dependence in sequence prediction problems. LSTM was created as the solution to short-term memory. They have internal mechanisms called gates that can regulate the flow of information. The two technical problems overcome by LSTMs are vanishing gradients and exploding gradients, both related to how the network is trained.

Model 4: Recurrent Neural Network (**LSTM**) – 3 Hidden Layers - With Batch Normalization and Dropout

- **LSTM** Input Layer (32 units) - Activation - **ReLU**
- 2 **LSTM** Hidden Layers (32 units) - Activation - **ReLU**
- 1 **LSTM** Hidden Layer (16 units) - Activation - **ReLU**
- **Fully Connected** Output Layer (3 units) - Activation - **Softmax**
- Adam Optimizer

```
Model: "sequential_4"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
lstm (LSTM)                  (None, 1, 32)             10496
_________________________________________________________________
dropout_8 (Dropout)          (None, 1, 32)             0
_________________________________________________________________
batch_normalization_8 (Batch (None, 1, 32)             128
_________________________________________________________________
lstm_1 (LSTM)                (None, 1, 32)             8320
_________________________________________________________________
dropout_9 (Dropout)          (None, 1, 32)             0
_________________________________________________________________
batch_normalization_9 (Batch (None, 1, 32)             128
_________________________________________________________________
lstm_2 (LSTM)                (None, 1, 32)             8320
_________________________________________________________________
dropout_10 (Dropout)         (None, 1, 32)             0
_________________________________________________________________
batch_normalization_10(Batch (None, 1, 32)             128
_________________________________________________________________
lstm_3 (LSTM)                (None, 16)                3136
_________________________________________________________________
dropout_11 (Dropout)         (None, 16)                0
_________________________________________________________________
batch_normalization_11(Batch (None, 16)                64
_________________________________________________________________
dense_18 (Dense)             (None, 3)                 51
=================================================================
Total params: 30,771
Trainable params: 30,547
Non-trainable params: 224
```

DNN Model 0: (Best model)

We use a **Sequential** model to build the DNN model.

The Sequential model is a linear stack of layers. It can be first initialized and then we add layers using add method or we can add all layers at initialization stage. The layers added are as follows:

Dense is a Fully Connected Layer. Dense implements the operation: output = activation (dot (input, kernel) + bias) where activation is the element-wise activation function passed as the activation argument, kernel is a weights matrix created by the layer, and bias is a bias vector created by the layer. The parameters used are:

- units - this is a positive integer, with the meaning: dimensionality of the output space; in this case is: 32 or 16 for hidden layers.
- activation - is the activation function used, in this case Rectified Linear Activation (ReLU / ReLU); ReLU has been proven to work well in neural networks. The **rectified linear activation** function or **ReLU** is a piecewise linear function that will output the input directly if it is positive, otherwise, it will output zero. In this model, for the input layer and hidden layer, ReLU has been used as the activation function. For the output layer, Softmax has been used.

Batch normalization, as the name suggests, seeks to normalize the activations of a given input volume before passing it into the next layer. It has been shown to be effective at reducing the number of epochs required to train a CNN at the expense of an increase in per-epoch time.

Dropout is a form of regularization that aims to prevent overfitting. Random connections are dropped to ensure that no single node in the network is responsible for activating when presented with a given pattern.

Input Dense Layer - This layer is a regular fully connected NN layer. It is used without parameters.

- units - this is a positive integer, with the meaning: dimensionality of the output space; in this case is: 32;
- activation - activation function: ReLU.

Dense - This layer is a regular fully connected NN layer. It is used without parameters.

- units - this is a positive integer, with the meaning: dimensionality of the output space; in this case is: 32;
- activation - activation function: ReLU.

Dense - This layer is a regular fully connected NN layer. It is used without parameters.

- units - this is a positive integer, with the meaning: dimensionality of the output space; in this case is: 32;
- activation - activation function: ReLU.

Dense - This layer is a regular fully connected NN layer. It is used without parameters.

- units - 16;
- activation - activation function: ReLU.

Dense Output Layer - This is the final layer (fully connected). It is used with the parameters:

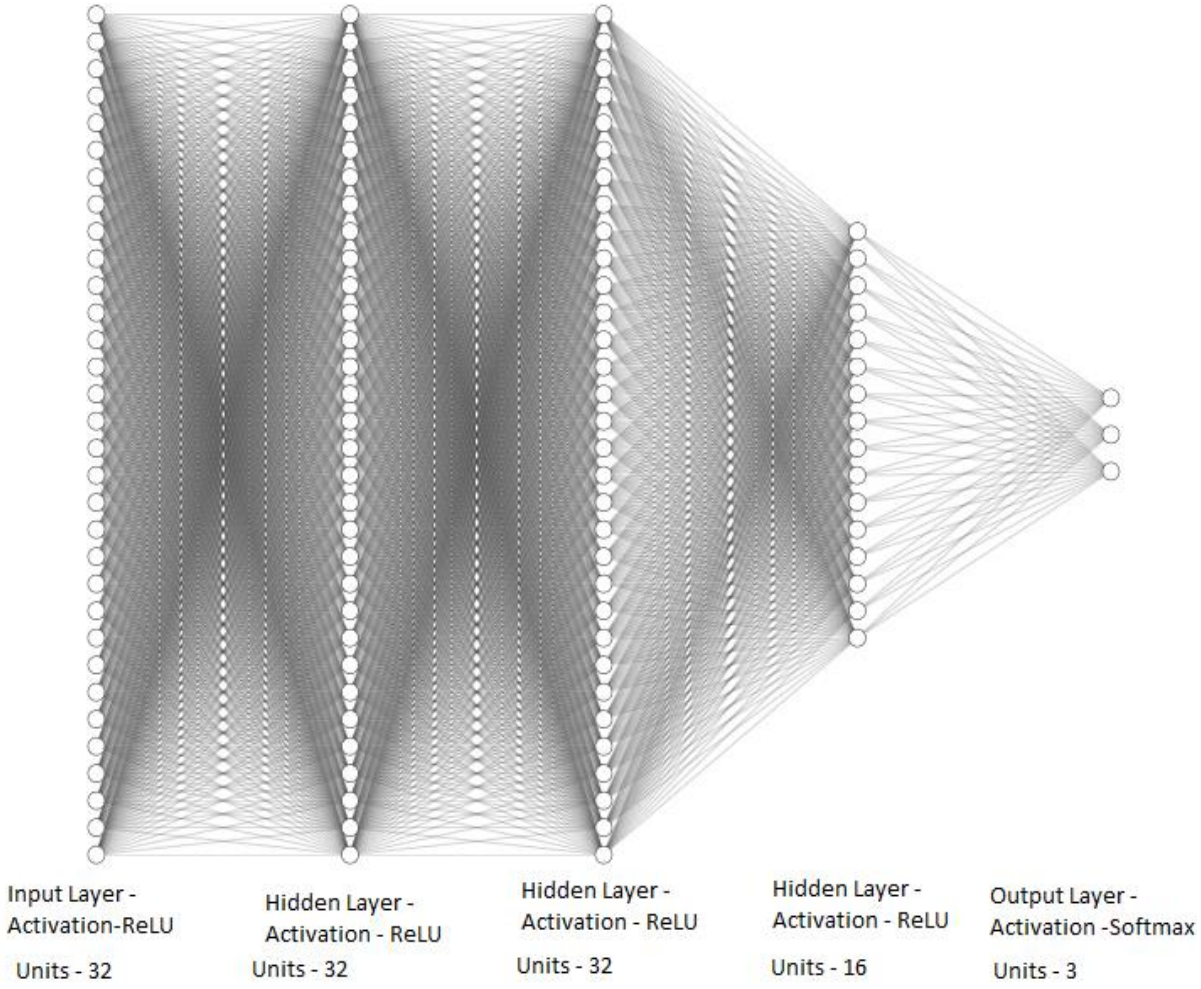
- units: the number of classes (in our case 3, one for each possible outcome)
- activation: Softmax; for this final layer it is used Softmax activation (standard for multiclass classification). Softmax makes the output sum up to 1 so the output can be interpreted as probabilities.

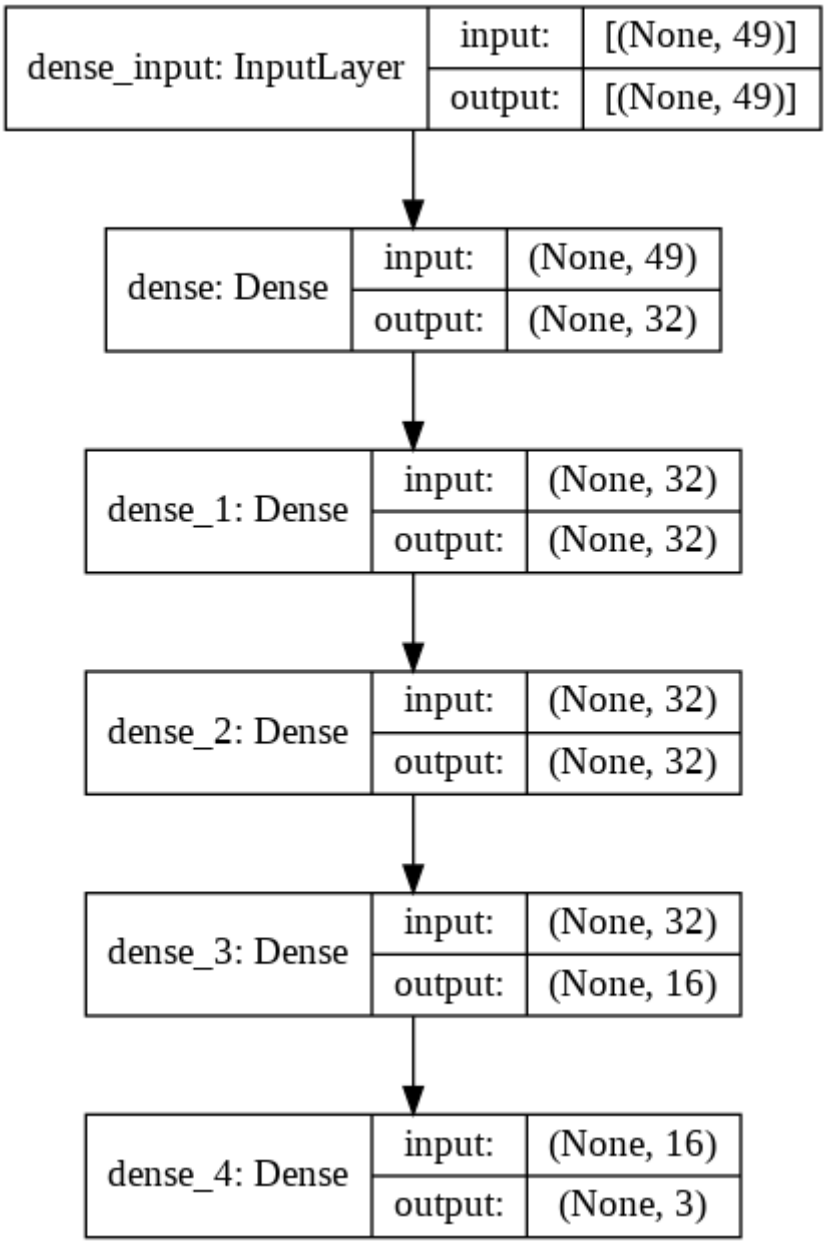
Then we compile the model, specifying the following parameters:

- loss - cross-entropy.
- optimizer – Adam.
- metrics - accuracy.

Optimizer is the algorithm used by the model to update weights of every layer after every iteration. Popular choices are SGD, RMSProp and Adam. SGD works well for shallow networks but cannot escape saddle points and local minima in such cases RMSProp could be a better choice, AdaDelta/AdaGrad for sparse data whereas Adam is a general favourite and could be used to achieve faster convergence. Hence, we use Adam for our model.

Model Visualization




dense_input: InputLayer
input: [(None, 49)]
output: [(None, 49)]
dense: Dense
input: (None, 49)
output: (None, 32)
dense_1: Dense
input: (None, 32)
output: (None, 32)
dense_2: Dense
input: (None, 32)
output: (None, 32)
dense_3: Dense
input: (None, 32)
output: (None, 16)
dense_4: Dense
input: (None, 16)
output: (None, 3)

[CM2] Implementation of your Design Choices

Loading the dataset -

```
df = pd.read_csv('COVID_dataset.csv')
```

Data Exploration -

df.head(5)

Accurate_Episode_Date	Case_Reported_Date	Test_Reported_Date	Specimen_Date	Age_Group	Client_Gender	Case_AcquisitionInfo	Reporting_PHU_City	Outbreak_Related	Reporting_PHU_Latitude	Reporting_PHU_Longitude	Outcome1
2020-03-30	2020-03-31	2020-03-31	2020-03-30	70s	MALE	OB	Stratford	Yes	43.368662	-81.001913	Fatal
2021-01-22	2021-01-24	2021-01-24	2021-01-23	50s	FEMALE	NO KNOWN EPI LINK	Newmarket	NaN	44.048023	-79.480239	Not Resolved
2020-03-24	2020-04-14	2020-04-14	2020-04-13	70s	FEMALE	OB	Toronto	Yes	43.656591	-79.379358	Resolved
2021-01-18	2021-01-21	2021-01-21	2021-01-18	<20	MALE	CC	Mississauga	NaN	43.647471	-79.708893	Not Resolved
2020-12-26	2020-12-28	2020-12-28	2020-12-26	60s	MALE	OB	Windsor	Yes	42.308796	-83.033670	Resolved

Pre-processing the dataset -

```
# Removing NAn values from age group and replacing in other features
df = df[df['Age_Group'].notna()]
df.loc[df['Test_Reported_Date'].isnull(),'Test_Reported_Date'] = df['Case_Reported_Date']
df.loc[df['Specimen_Date'].isnull(),'Specimen_Date'] = df['Accurate_Episode_Date']
df['Outbreak_Related'].fillna(value='No', inplace=True)
```

```
# converting date to integers
for column in ('Accurate_Episode_Date', 'Case_Reported_Date', 'Test_Reported_Date', 'Specimen_Date'):
        df[column] = pd.to_datetime(df[column]).astype(int) // int(1e9)
```

```
# scaling down numerical features
from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import RobustScaler
from sklearn.preprocessing import MinMaxScaler

scaled_features = df.copy()
col_names = ['Accurate_Episode_Date', 'Case_Reported_Date', 'Test_Reported_Date',
             'Specimen_Date', 'Reporting_PHU_Latitude', 'Reporting_PHU_Longitude']
features = scaled_features[col_names]
scaler = RobustScaler().fit(features.values)
# scaler = StandardScaler().fit(features.values)
# scaler = MinMaxScaler().fit(features.values)
features = scaler.transform(features.values)

scaled_features[col_names] = features
df = scaled_features
```

Splitting the dataset into Train, Validation and Test set (80% - 10% - 10%) with random seed as 0.

```
from sklearn.model_selection import train_test_split

# splitting data and target
X = df.drop(['Outcome1'], axis=1)
y = df['Outcome1']

# dividing the data into train, validation and test sets (80%, 10%, 10%) with random_state=0
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=0)
X_test, X_val, y_test, y_val = train_test_split(X_test, y_test, test_size=0.5, random_state=0)
```

Defining and compiling the models -

```
# Model
EPOCHS = 50
BATCH_SIZE = 64
```

Model 0 – DNN – 3 HIDDEN LAYERS

```
# Dependencies
import tensorflow as tf
import keras
from keras.models import Sequential
from keras.layers import Dense
from keras import activations
from tensorflow.keras.optimizers import Adam, SGD

# Deep Neural network
model = Sequential()
model.add(Dense(32, activation=activations.relu))
model.add(Dense(32, activation=activations.relu))
model.add(Dense(32, activation=activations.relu))
model.add(Dense(16, activation=activations.relu))
model.add(Dense(3, activation=activations.softmax))

model.compile(loss='sparse_categorical_crossentropy',
              optimizer=Adam(learning_rate=0.001), metrics=['accuracy'])
```

Model 1 - DNN (DEEPER- 5 HIDDEN LAYERS)

```
# Deep Neural network
model1 = Sequential()
model1.add(Dense(32, activation=activations.relu))
model1.add(Dense(32, activation=activations.relu))
model1.add(Dense(32, activation=activations.relu))
model1.add(Dense(32, activation=activations.relu))
model1.add(Dense(32, activation=activations.relu))
model1.add(Dense(16, activation=activations.relu))
model1.add(Dense(3, activation=activations.softmax))

model1.compile(loss='sparse_categorical_crossentropy',
               optimizer=Adam(learning_rate=0.001), metrics=['accuracy'])
```

Model 2 – DNN WITH DROPOUT & BATCH NORMALIZATION

```
from tensorflow.keras.layers import Dropout
from tensorflow.keras.layers import BatchNormalization

# Model
model2 = Sequential()
model2.add(Dense(32, activation=activations.relu))
model2.add(BatchNormalization())
model2.add(Dropout(0.2))
model2.add(Dense(32, activation=activations.relu))
model2.add(BatchNormalization())
model2.add(Dropout(0.3))
model2.add(Dense(32, activation=activations.relu))
model2.add(BatchNormalization())
model2.add(Dropout(0.3))
model2.add(Dense(16, activation=activations.relu))
model2.add(BatchNormalization())
model2.add(Dropout(0.2))
model2.add(Dense(3, activation=activations.softmax))

model2.compile(loss='sparse_categorical_crossentropy',
               optimizer=Adam(learning_rate=0.001), metrics=['accuracy'])
```

Model 3 – SIMPLE RNN

```
# Reshaping X_train, X_val and X_test for efficient modelling
X_train = np.expand_dims(X_train, 1)
X_val = np.expand_dims(X_val, 1)
X_test = np.expand_dims(X_test, 1)
```

```
from tensorflow.keras.layers import SimpleRNN

# Recurrent Neural network
model3 = Sequential()

# First SimpleRNN layer
model3.add(SimpleRNN(32, return_sequences=True, activation=activations.tanh,
                     use_bias=True))
model3.add(Dropout(0.2))
model3.add(BatchNormalization())
# Second SimpleRNN layer
model3.add(SimpleRNN(32, return_sequences=True, activation=activations.tanh))
model3.add(Dropout(0.3))
model3.add(BatchNormalization())
# Third SimpleRNN layer
model3.add(SimpleRNN(32, return_sequences=True, activation=activations.tanh))
model3.add(Dropout(0.3))
model3.add(BatchNormalization())
# Fourth SimpleRNN layer
model3.add(SimpleRNN(16, activation=activations.tanh))
model3.add(Dropout(0.2))
model3.add(BatchNormalization())
# The output layer
model3.add(Dense(3, activation=activations.softmax))

model3.compile(loss='sparse_categorical_crossentropy',
               optimizer=Adam(learning_rate=0.001), metrics=['accuracy'])
```

Model 4 – LSTM

```
from tensorflow.keras.layers import LSTM

# Recurrent Neural network - LSTM
model4 = Sequential()

# First LSTM layer with Dropout regularisation
model4.add(LSTM(32, return_sequences=True, activation=activations.tanh))
model4.add(Dropout(0.2))
model4.add(BatchNormalization())
# Second LSTM layer
model4.add(LSTM(32, return_sequences=True, activation=activations.tanh))
model4.add(Dropout(0.3))
model4.add(BatchNormalization())
# Third LSTM layer
model4.add(LSTM(32, return_sequences=True, activation=activations.tanh))
model4.add(Dropout(0.3))
model4.add(BatchNormalization())
# Fourth LSTM layer
model4.add(LSTM(16, activation=activations.tanh))
model4.add(Dropout(0.2))
model4.add(BatchNormalization())
# The output layer
model4.add(Dense(3, activation=activations.softmax))

model4.compile(loss='sparse_categorical_crossentropy',
               optimizer=Adam(learning_rate=0.001), metrics=['accuracy'])
```

Training the model to fit the processed Covid dataset with Batch-Size of 64, and Epochs 50. The start time and the end time is noted to calculate the total time taken for model training.

```
# Train the the model
start_time = time.time()
train_model = model.fit(X_train, y_train, batch_size=BATCH_SIZE,
                        epochs=EPOCHS, verbose=1, validation_data=(X_val, y_val),
                        shuffle=True)
end_time = time.time()
print("Total training time : {:0.2f} minute".format((end_time - start_time)/60.0))
```

Performance of the trained model (best model – in terms of accuracy and training time) on the Test set -

```
%%time
score = model.evaluate(X_test, y_test, verbose=0)
print('Test accuracy:', score[1])
print('Test loss:', score[0])
```

```
%%time
y_pred = np.argmax(model.predict(X_test), axis=-1)
```

```
from sklearn.metrics import classification_report

target_names = [" {}:".format(i)  for i in ['Resolved', 'Not Resolved', 'Fatal']]
print(classification_report(y_pred, y_test, target_names=target_names))
```

Method for Accuracy – Loss plots for the trained model

```
def accuracy_loss_plot(model):

  hist = model.history
  acc = hist['accuracy']
  val_acc = hist['val_accuracy']
  loss = hist['loss']
  val_loss = hist['val_loss']
  epoch = range(50)

  fig = plt.figure(figsize = (12,10))
  plt.subplot(2,2,1)
  sns.lineplot(acc,loss)
  plt.xlabel('Accuracy')
  plt.ylabel('Loss')
  plt.legend(['train set'], loc='upper right')
  plt.title('Training Accuracy vs Loss')

  plt.subplot(2,2,2)
  sns.lineplot(val_acc,val_loss)
  plt.xlabel('Accuracy')
  plt.ylabel('Loss')
  plt.legend(['validation set'], loc='upper right')
  plt.title('Validation Accuracy vs Loss')

  plt.subplot(2,2,3)
  sns.lineplot(epoch, acc)
  sns.lineplot(epoch, val_acc)
  plt.xlabel('Epoch')
  plt.ylabel('Accuracy')
  plt.legend(['train set', 'validation set'], loc='lower right')
  plt.title('Accuracy vs Epoch')

  plt.subplot(2,2,4)
  sns.lineplot(epoch, loss)
  sns.lineplot(epoch, val_loss)
  plt.xlabel('Epoch')
  plt.ylabel('Loss')
  plt.legend(['train set', 'validation set'], loc='upper right')
  plt.title('Loss vs Epoch')

  plt.show()
```

[CM3] Results Analysis

Run-time performance for training and testing -

Training time for all the models for Batch-Size 64 and 50 epochs

Model 0: DNN

```
Epoch 50/50
186/186 [==============================] - 0s 2ms/step - loss: 0.2489 - accuracy: 0.9153 - val_loss: 0.2454 - val_accuracy: 0.9098
Total training time : 0.34 minute
```

Model 1: DNN - Deeper

```
Epoch 50/50
186/186 [==============================] - 0s 2ms/step - loss: 0.2423 - accuracy: 0.9125 - val_loss: 0.2873 - val_accuracy: 0.8964
Total training time : 0.37 minute
```

Model 2: DNN – Dropout and Batch Normalization

```
Epoch 50/50
186/186 [==============================] - 1s 3ms/step - loss: 0.2893 - accuracy: 0.9008 - val_loss: 0.2448 - val_accuracy: 0.9118
Total training time : 0.50 minute
```

Model 3: RNN

```
Epoch 50/50
186/186 [==============================] - 1s 5ms/step - loss: 0.3005 - accuracy: 0.8994 - val_loss: 0.2511 - val_accuracy: 0.9145
Total training time : 0.87 minute
```

Model 4: LSTM

```
Epoch 50/50
186/186 [==============================] - 1s 8ms/step - loss: 0.2983 - accuracy: 0.8974 - val_loss: 0.2461 - val_accuracy: 0.9071
Total training time : 1.42 minute
```

The training cost/time for DNN models mainly depends on the model complexity (depth, parameters etc.) and the training sample size.

- We observe that as the model complexity increases (number of layers, dropouts, Batch normalization), the total training time increases as well.
- The training time increases as the number of trainable parameters increase.
- LSTM is seen to take the longest training time as it has a lot of parameters to be processed. LSTM require 4 linear layer (MLP layer) per cell to run at and for each sequence time-step. Linear layers require large amounts of memory bandwidth to be computed.

Testing time of Model 0 -

```
%%time
score = model.evaluate(X_test, y_test, verbose=0)
print('Test accuracy:', score[1])
print('Test loss:', score[0])
```

```
Test accuracy: 0.9178451299667358
Test loss: 0.2335989624261856
CPU times: user 150 ms, sys: 9.7 ms, total: 160 ms
Wall time: 150 ms
```

```
%%time
y_pred = np.argmax(model.predict(X_test), axis=-1)
```

```
CPU times: user 222 ms, sys: 6.47 ms, total: 228 ms
Wall time: 224 ms
```

The model takes 150 ms to evaluate on the test dataset, and 224 ms to predict the classes of the test dataset images.

Comparison of the different algorithms and parameters -

Model	Hidden Layers	Optimizer	Train Accuracy	Val Accuracy	Train Loss	Val Loss	Test Accuracy	Training Time
Model 0	**3**	**Adam**	**0.9166**	**0.9098**	**0.2344**	**0.2454**	**0.9178**	**0.34 minute**
Model 1	5	Adam	0.9059	0.8964	0.2719	0.2873	0.9044	0.37 minute
Model 2	3	Adam	0.9116	0.9118	0.2453	0.2448	0.9192	0.50 minute
Model 3	3	Adam	0.9121	0.9145	0.2497	0.2511	0.9138	0.87 minute
Model 4	3	Adam	0.9160	0.9071	0.2440	0.2461	0.9192	1.42 minute

Table 1: Model performance for different algorithms and parameters

We observe that:

- All the models, show similar accuracy with the train set, but the training time goes higher with increase in complexity as the number of trainable parameters increase.
- All the models are generalizing well with respect to unseen data, hence none of them really overfits.
- In terms of training time, the simpler DNN model takes the minimum amount to train, hence it is chosen as the best model considering both accuracy and training time.
- Model 1 with 5 hidden layers, has the highest train and validation loss among all models. The differences are not too significant among other models.

Comparison of the different parameters on best model – Model 0

Optimizer	Train Accuracy	Val Accuracy	Train Loss	Val Loss	Test Accuracy
SGD	0.7401	0.7510	0.5786	0.5724	0.7273
Adam	**0.9166**	**0.9098**	**0.2344**	**0.2454**	**0.9178**

Table 2: Model 0 Performance of Optimizers (Epochs - 50, Batch Size - 64)

From Table 2, upon trying different optimizers (SGD and Adam), we observe that SGD optimizer gives much lower training and test accuracy compared to Adam. Thus, Adam is a better optimizer for the model. Train and Validation loss is also much higher for SGD as compared to Adam. Also, SGD slightly overfits the data.

Batch Size	Train Accuracy	Val Accuracy	Train Loss	Val Loss	Test Accuracy	Training Time
16	0.9062	0.8984	0.2721	0.2859	0.9017	1.10 minute
32	0.9044	0.8970	0.2713	0.2870	0.9077	0.58 minute
64	**0.9166**	**0.9098**	**0.2344**	**0.2454**	**0.9178**	**0.28 minute**
128	0.9191	0.9145	0.2323	0.2441	0.9151	0.22 minute

Table 3: Model 0 Batch Size Experimentation (Epochs - 50, Optimizer - Adam)

Batch size is the number of samples processed before the model is updated. From Table 3, we observe that as the batch size increases, the training time decreases. Using larger batch sizes would allow the model to parallelize computations to a greater extent, speeding up the model training significantly. Batch Size 64 and 128 perform better in terms of training time, accuracy, train loss and validation loss as compared to batch size 16 and 32.

Learning Rate	Train Accuracy	Val Accuracy	Train Loss	Val Loss	Test Accuracy	Training Time
0.01	0.9159	0.9145	0.2358	0.2441	0.9118	0.35 minute
0.02	0.9088	0.9091	0.2495	0.2511	0.9104	0.34 minute
0.001 (default)	**0.9166**	**0.9098**	**0.2344**	**0.2454**	**0.9178**	**0.28 minute**
0.005	0.9172	0.9125	0.2395	0.2482	0.9118	0.31 minute

Table 4: Model 0 Learning Rate Experiment (Epochs -50, Batch Size -64, Optimizer - Adam)

The learning rate is a hyperparameter that determines the amount of change in the model in response to the estimated error each time the model weights are updated. A small learning rate value may result in a long training process that could get stuck, whereas a value too large may

lead to learning a sub-optimal set of weights too fast or an unstable training process. From Table 4, we observe that the model has slightly low training accuracy for higher learning rate. Also, as the learning rate decreases, the training time decreases. We observe the best training accuracy for the default training rate (0.001) for Adam optimizer.

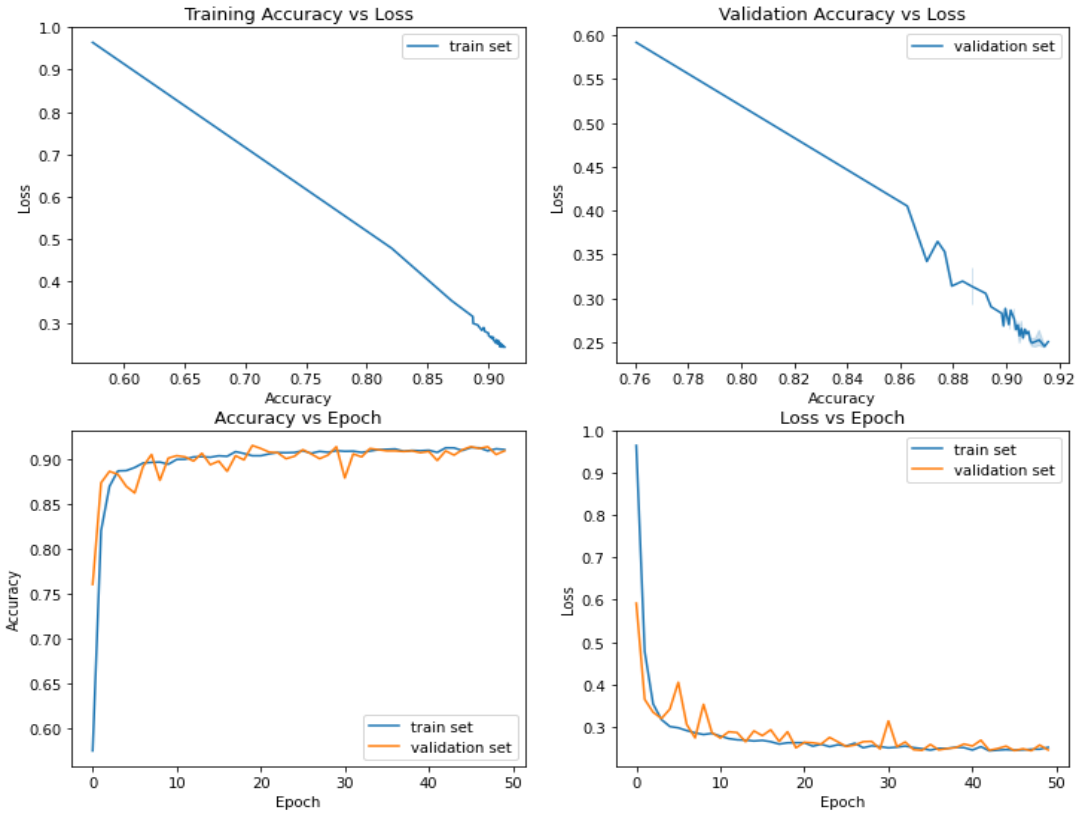
Epochs	Train Accuracy	Val Accuracy	Train Loss	Val Loss	Test Accuracy	Training Time
10	0.9082	0.9098	0.2677	0.2865	0.9124	0.09 minute
30	0.9126	0.9071	0.2488	0.2536	0.9138	0.21 minute
50	**0.9166**	**0.9098**	**0.2344**	**0.2454**	**0.9178**	**0.28 minute**
70	0.9148	0.9098	0.2377	0.2546	0.9124	0.47 minute

Table 5: Model 4 Epoch Experimentation (Batch Size - 64, Optimizer - Adam)

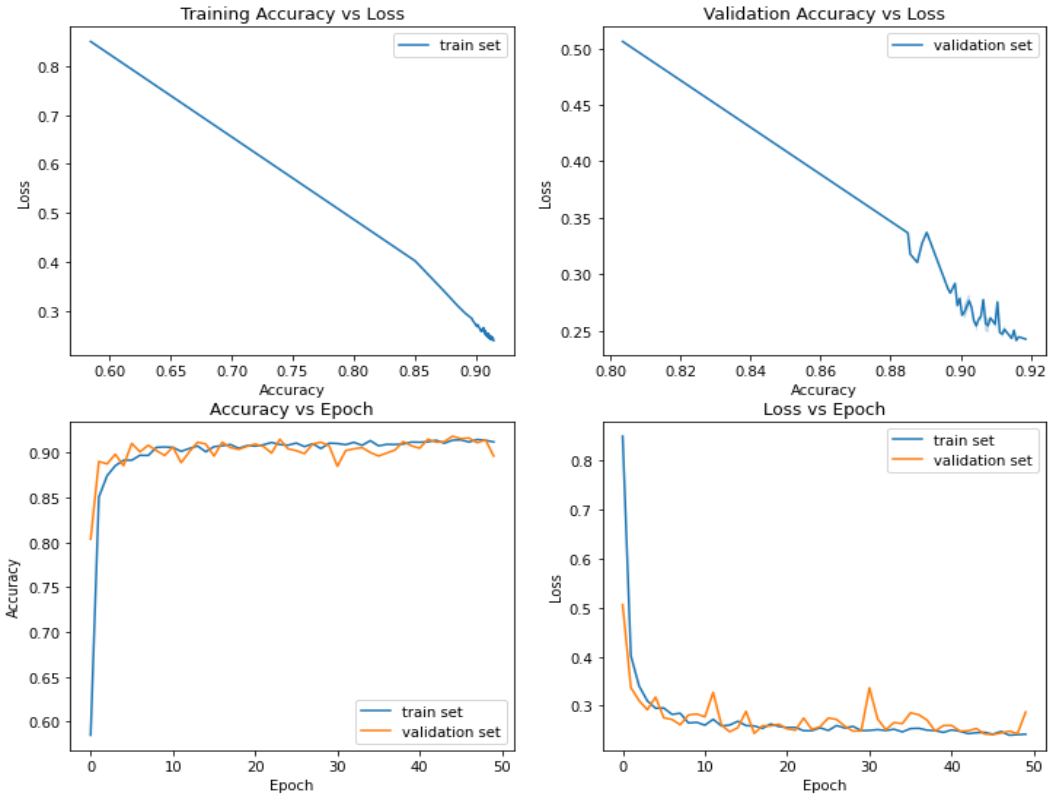
The number of epochs is the number of complete passes through the training dataset. From Table 5, we notice that the accuracy of the model increases with more epochs. For 50 and 70 epochs, the model has similar accuracy but different training time. The training time increases as the number of epochs increase. With 10 epochs, the train and validation loss is slightly higher.

Accuracy – Loss plots

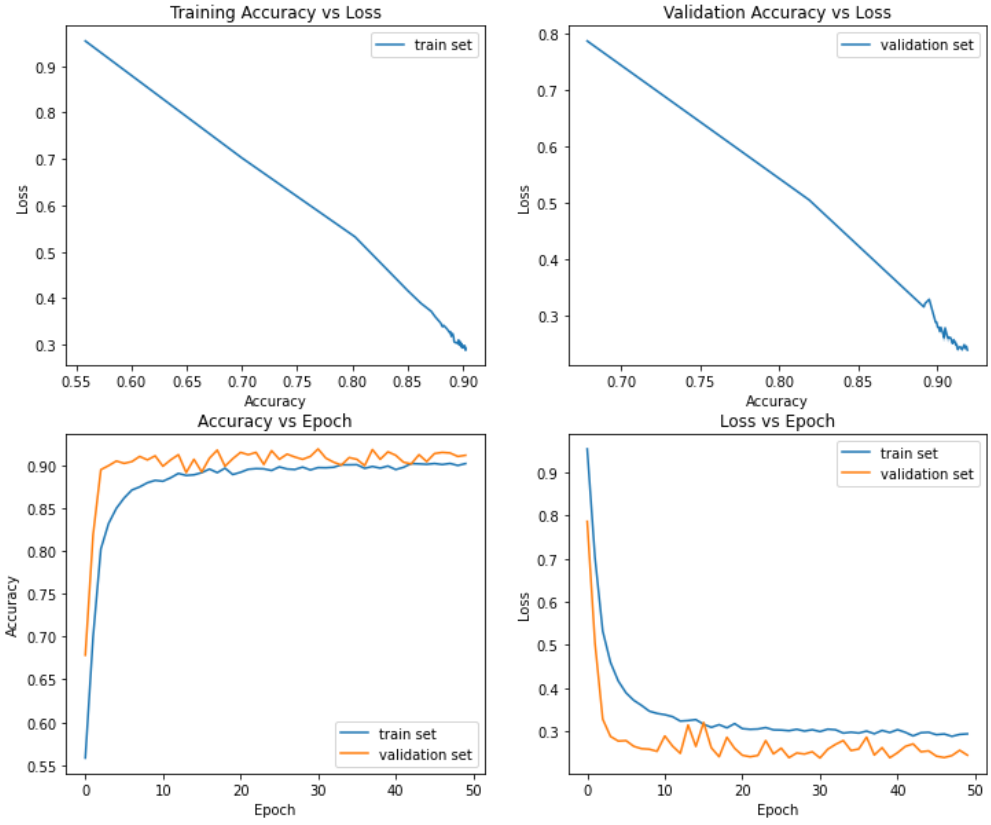
Model 0:



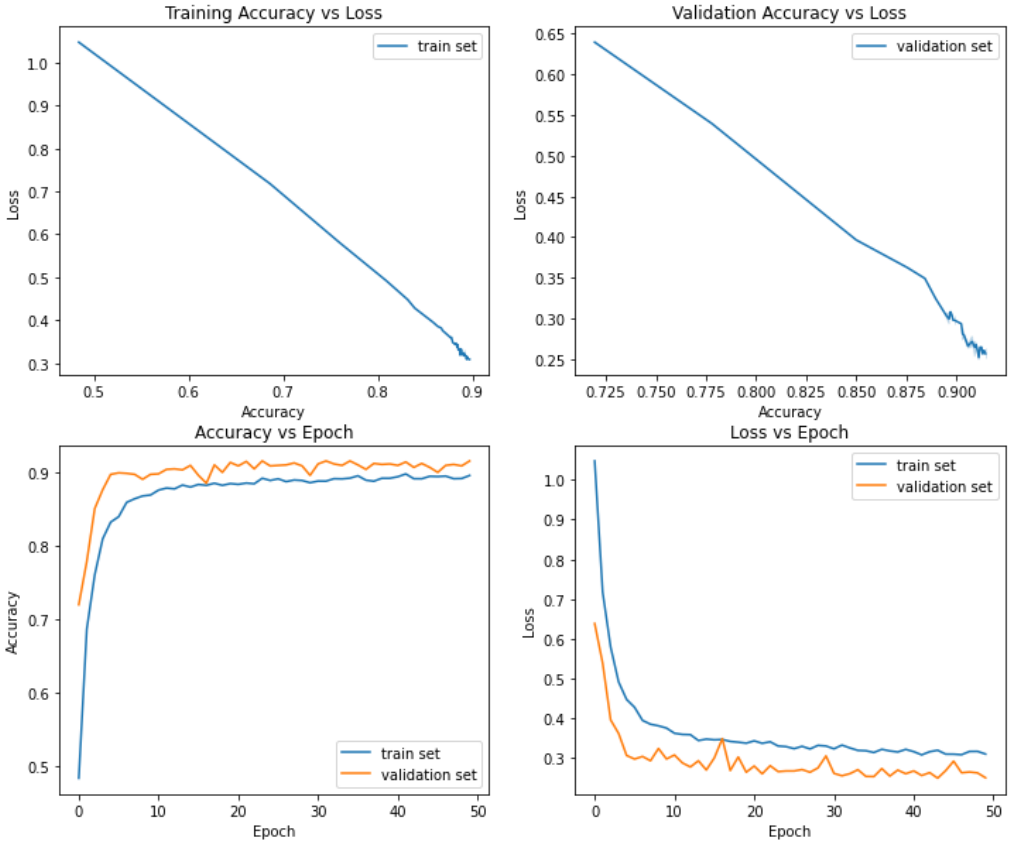
Model 1:



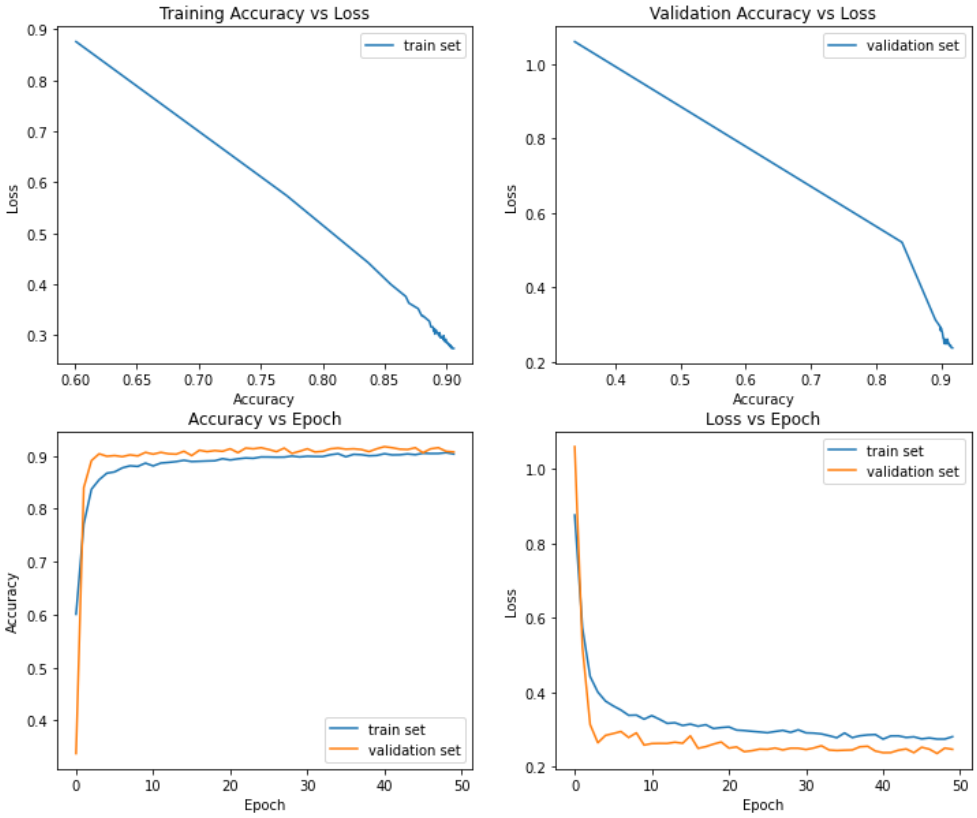
Model 2:



Model 3:



Model 4:



From the Accuracy – Loss plots, we observe:

- For all the Models, the Loss decreases as the accuracy increases for both the training and validation dataset.
- Similarly, for all the Models, the Accuracy increases with increase in epochs, and the Loss decreases with increase in epochs for both the training and test dataset.
- Model 0 and Model 1, the training and validation accuracy and loss curves are overlapping, indicating a good fit.
- Model 2, 3 & 4, the validation accuracy is slightly higher than the training accuracy and the validation loss lower than the training loss, indicating the model is able to generalize better.
- The fluctuations in the Validation Accuracy and Loss plots can be attributed to small size of the validation dataset.

Performance of the Model on the test set –

```
%%time
score = model.evaluate(X_test, y_test, verbose=0)
print('Test accuracy:', score[1])
print('Test loss:', score[0])
```

```
Test accuracy: 0.9178451299667358
Test loss: 0.2335989624261856
CPU times: user 150 ms, sys: 9.7 ms, total: 160 ms
Wall time: 150 ms
```

```
%%time
y_pred = np.argmax(model.predict(X_test), axis=-1)
```

```
CPU times: user 222 ms, sys: 6.47 ms, total: 228 ms
Wall time: 224 ms
```

Evaluating the performance of the best Model 0 on the Test dataset (10% of whole dataset), we get an accuracy of ~92%, which is good for a classifier. The model is able to generalize i.e., classify unseen data well. In addition to classification accuracy, the metrics that are commonly required for a neural network model on a classification problem are:

- Precision
- Recall
- F1 Score

	precision	recall	f1-score	support
Resolved:	0.86	0.94	0.90	450
Not Resolved:	0.97	0.95	0.96	502
Fatal:	0.93	0.87	0.90	533
accuracy			0.92	1485
macro avg	0.92	0.92	0.92	1485
weighted avg	0.92	0.92	0.92	1485

From the above model classification report, we observe:

- Model has the highest precision for class ‘Not Resolved’ & ‘Fatal’, and the lowest for ‘Resolved’. Precision tells how precise/accurate the model is, out of the predicted positive, how many are actual positive. We see the Model is able to correctly classify cases with outcome as ‘Not Resolved’ and ‘Fatal’ and misclassifies many ‘Resolved’ cases.
- Model has highest recall for class ‘Not Resolved’, and the lowest for ‘Fatal’. Recall tells how many of the Actual Positives the model capture through classifying it as Positive (True Positive).
- Model has the highest f1-score for ‘Not Resolved’ and the same score for class ‘Resolved’ & ‘Fatal’. F1-score seek a balance between Precision and Recall. We see the model has better balance between precision-recall for class ‘Not Resolved’ than all other classes.

References:

https://machinelearningmastery.com/gentle-introduction-long-short-term-memory-networks-experts/

https://www.tutorialspoint.com/keras/keras_dense_layer.htm#:~:text=Dense%20layer%20is%20the%20regular,input%20and%20return%20the%20output.

https://datascience.stackexchange.com/questions/12851/how-do-you-visualize-neural-network-architectures

https://towardsdatascience.com/lstm-by-example-using-tensorflow-feb0c1968537

https://becominghuman.ai/a-noobs-guide-to-implementing-rnn-lstm-using-tensorflow-1907a5bbb1fa?gi=e30fd8019fc6

https://towardsdatascience.com/understanding-and-implementing-dropout-in-tensorflow-and-keras-a8a3a02c1bfa

https://www.machinecurve.com/index.php/2020/01/15/how-to-use-batch-normalization-with-keras/

https://www.kaggle.com/gpreda/cnn-with-tensorflow-keras-for-fashion-mnist

https://www.analyticsvidhya.com/blog/2018/04/fundamentals-deep-learning-regularization-techniques/

https://towardsdatascience.com/illustrated-guide-to-lstms-and-gru-s-a-step-by-step-explanation-44e9eb85bf21