Deep Learning – Individual Assignment – Report

**Curated By - Paritosh Sinha**

# Part 1 – Predicting Customer Churn using Neural Networks

## Exploratory data analysis:

There are no null or missing values in the dataset. Other interesting insights from EDA are –

* 1. If the person is with no dependents, then they are more likely to churn as compared to people with dependents.
  2. There are very few customers with “No Phone Service”
  3. Consumers who have opted for Fibre Optic either Churn a lot or do not Churn at all. So, this might be dependent on quality of service
  4. Customers who have stayed with us for more than 2 years they are less likely to Churn
  5. Customers with payment method as electronic check they are most likely to churn as compared to other payment methods.
  6. MontlyCharges have a strong correlation with TotalCharges.
  7. Mostly Customers Churn post onboarding which is within a year’s time.
  8. Interestingly enough Customers who churned pay lesser charge then the customer who did not churn.

## Preprocessing:

* Removing the Gender variable as it ensures no gender related bias is learned by our model.
* Removing customerId as it will not help us in making the model.
* Outlier detection – but there were no outlier
* Class imbalance need to be treated we need to balance the two classes yes and no equally so that the model is balanced this is done using the sampling method. Output of this will balance the Yes/ No class equally for the Churn.
* Now we need to preprocess the data that is present in the dataset which is done differently for different type of data
  + MixMaxScaler – For the continuous or discrete variables like – MonthlyCharge, TotalCharge,
  + LabelBinarizer - Variables which have only binary category variables like Yes/No. SeniorCitizen
  + get\_dummies – For the variables which have more than two categories Features which are of this type are – MultipleLines, InternetService
* Aim for above process is to make sure all the inputs are of numerical type as the neural nets understand the numerical input only.

## Model architecture:

* **Model Class (Classifier):** Class is a PyTorch module that is a simple feedforward neural network for a binary classification task. Takes input\_size as a parameter representing the number of input features for the model which we are going to use for Classification.
* **Layers:** Hidden Layers (fully connected):
  + self.hidden\_n: Linear layer with m output features.
  + Each hidden layer is topped by ReLU (Rectified Linear Unit) activation function and dropout with a dropout rate) for regularization.

### Output Layer:

* self.output: Linear layer with 2 output features, which is for binary classification tasks like ours.
* The output layer is followed by log softmax activation.It is activation function that transforms the vector output of neural network outputs into probabilities.
* **Dropout**: Dropout is used to lessen overfitting after each layer of neural net.
* **Forward Function**: Comprises of activation function on each of the hidden layer which was defined in the init function. Applies dropout on the activation (**Relu**) through hidden layers. Applies log **softmax** to obtain the output probabilities for both the classes here.
* **Loss Function (Negative Log-Likelihood Loss):** nn.NLLLoss() (Negative Log-Likelihood Loss) is used as the loss function as it is in close association with the cross-entropy loss, which is a common loss function we use for classification problems.
* **Optimizer (Adam):** An adaptive optimization algorithm called the Adam optimizer effectively modifies learning rates during training to hasten convergence and improve model performance. This is achieved by updating the model's weights during training.

### Training Loop:

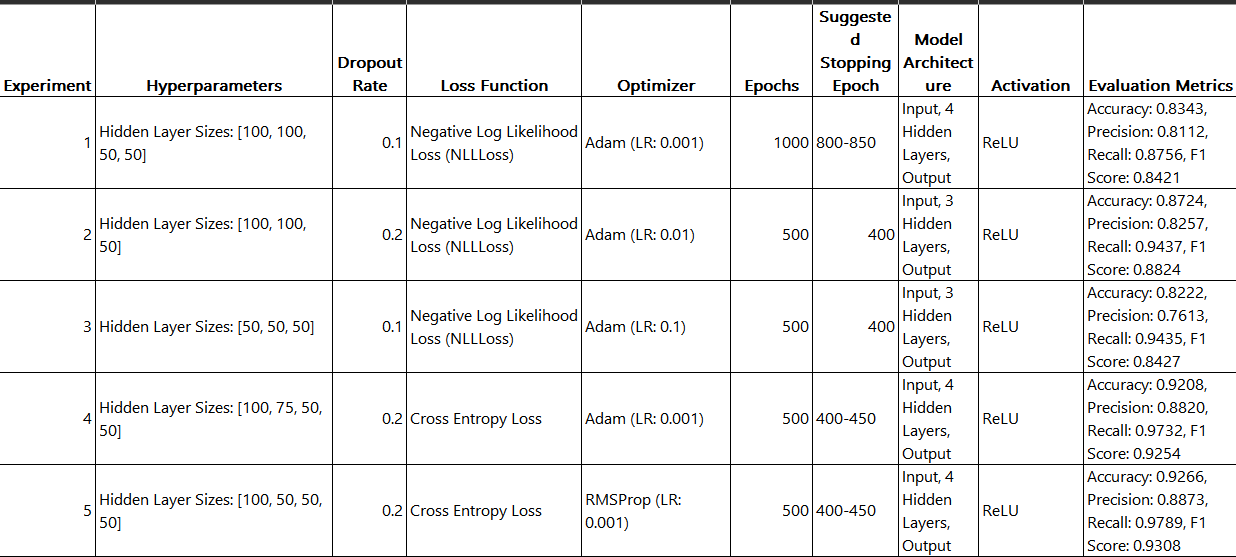
* A predetermined number of epochs are completed in the training loop.
* The training data is processed in mini-batches and is scrambled throughout each epoch.

### For each mini-batch:

* The model is run forward to get the predicted log probabilities.
* Loss is calculated using the defined loss function which is NLLLoss for this problem.
* Backpropagation and optimization (Adam optimizer) steps are performed.
* Validation loss and accuracy are computed after each epoch.
* Training and validation loss, as well as accuracy, are tracked which can help in studying the performance of the neural net over each epoch.

1. **Training and Evaluation:** Training**:** Splitting the dataset into Train (80%) and Test (20%) Evaluation: mentioned for different model in the matrix below

## Experimentation with different hyperparameters and architectures and Evaluation



### Summary: Keep the two major factors to consider while choosing the model –

**Based on the Principle of Parsimony** where I am aiming for simpler model but with good metrics – model from the following experiment can be considered –

From the parsimony aspect, Model complexity and performance are kept in check in these tests. In **Experiment 2**, overshooting during optimization is avoided by employing a slightly slower learning rate. For Adam, **Experiment 4** likewise exhibits a good learning rate and an ideal dropout rate. **Experiment 5** investigates RMSProp, a different optimizer that has been proven successful as well.

**Based on the problem statement which can be of minimising the Customer Churn** To minimize customer churn, we need to focus on models that give a good balance between false positives (customers predicted to churn but don't) and false negatives (customers predicted to stay but churn). Looking at the evaluation metrics from the experiments, we can focus on models with higher recall.

**Experiment 4: Recall 97.32% or 5(Best): Recall: 97.89%** These two experiments have the highest recall, making it a good candidate as it effectively identifies customers who are likely to churn. **From above two criteria – Any model from Experiment 4 or 5 can be considered good**.

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

# Part 2 – Spam Classification using Recurrent Neural Networks

1. **Data pre-processing:** After **importing** the dataset named “SPAM\_text\_message.csv”

### Text Preprocessing:

* Tokenizer Initialization: split the text into words or tokens.
* Fit on Texts: fits the tokenizer on the text data in the 'Message' column. This process learns the vocabulary and assigns a unique index to each word.
* Texts to Sequences: converts the text in 'Message' to sequences of numbers based on the tokenized words. Each word is represented by its corresponding index.
  + **Padding Sequences:** pad\_sequences are used to ensure that all sequences have the same length. If a sequence is shorter than max, it pads zeros. Since all the messages are not of same length.
  + **Convert Categories to Binary Labels:** Predicted (Y) is a binary categorical data which currently contains value as Spam or Ham. Spam = 1 and Ham = 0
  + **Importing the pretrained word embedding file -** I have used - glove.twitter.27B.100d

## Model architecture:

### Input Layer:

* + - **Embedding**: The first layer is an embedding layer that converts input sequences into dense vectors of fixed size. Here I am using pre-trained GloVe embeddings as initial weights.
    - **input\_dim**: Vocabulary size, the total number of unique words in the dataset.
    - **output\_dim**: Dimension of the dense embedding.
    - **embeddings\_initializer:** Constant initializer to load pre-trained embeddings.
    - trainable=False: The embeddings are frozen and not updated during training.

### Recurrent Layer:

* + - **SimpleRNN:** A simple RNN layer with 32 units (or neurons). It processes the embedded sequences.

### Output Layer:

* + - **Dense**: A dense layer with a single neuron and a sigmoid activation function, which will be used for binary classification tasks like spam detection.
    - **activation**='sigmoid': Sigmoid activation squashes the output between 0 and 1, making it suitable for binary classification.
  + **Compilation**: Compiles the model with the **RMSprop** optimizer and **binary cross-entropy loss function** with metrics as **accuracy**

## Training and evaluation:

* **X\_train**: Training input data (tokenized and padded sequences).
* **y\_train**: Training labels (binary: 0 for 'ham', 1 for 'spam').
* **Embedding:** layer which is frozen while training the model.
* **epochs**=10: Number of training epochs.
* **batch\_size**=128: Batch size for training.
* **validation\_split**=0.2: 20% of the training data is used for validation during training.

## Experimentation with different hyperparameters and architectures:

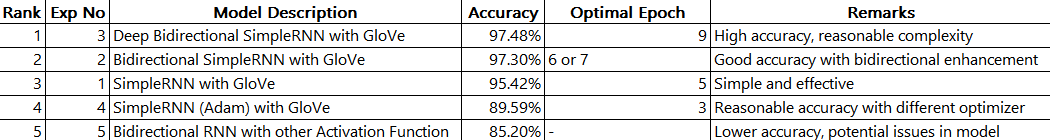
***Architectures and Hyperparameters and Accuracy***

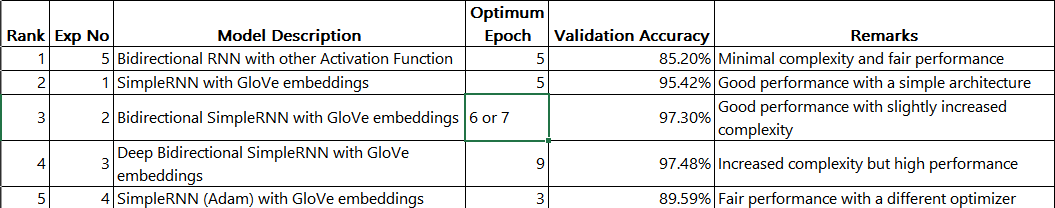
Embedding Layer: Pretrained GloVe embeddings

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Experiment** | **Model Type** | **Hidden Layer**  **Sizes** | **Activation** | **Optimizer** | **Loss Function** | **Epochs** | **Accuracy** | **Suggested Stopping Epoch** |
| 1 | Sequential Model | Simple RNN  (32) | Sigmoid | RMSprop | Binary Crossentropy | 5 | 95.42% | Epoch 5 |
| 2 | Bidirectional SimpleRNN | [32,  32] | Sigmoid | RMSprop | Binary Cross Entropy | 10 | 97.30% | Epoch 6 or Epoch 7 |
| 3 | Deep Bidirectional  SimpleRNN | [32,  32] | Sigmoid | RMSprop | Binary Cross Entropy | 10 | 97.48% | Epoch 9 |
| 4 | Sequential Model | Simple RNN  (32) | Sigmoid | Adam (LR:  0.001) | Binary Crossentropy | 5 | 89.59% | Epoch 3 |
| 5 | Bidirectional RNN | Simple RNN  (32) | Sigmoid | SGD (LR: 0.01) | Binary Crossentropy | 5 | 85.20% | After 1st Epoch |

## Summary:

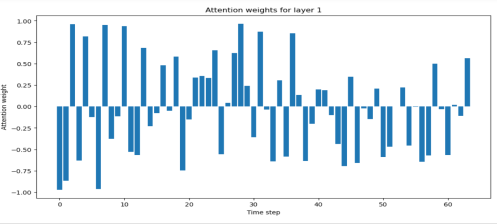
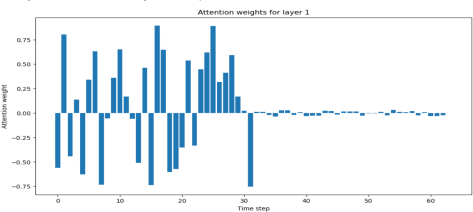
* + One major problem with training the model is that data of both the classes is not equally present. Ham i.e. Not Spam is dominating the dataset which is posing the problem of training the dataset properly.
  + Keeping in mind the **Validation Accuracy** we can comment on the above model as following



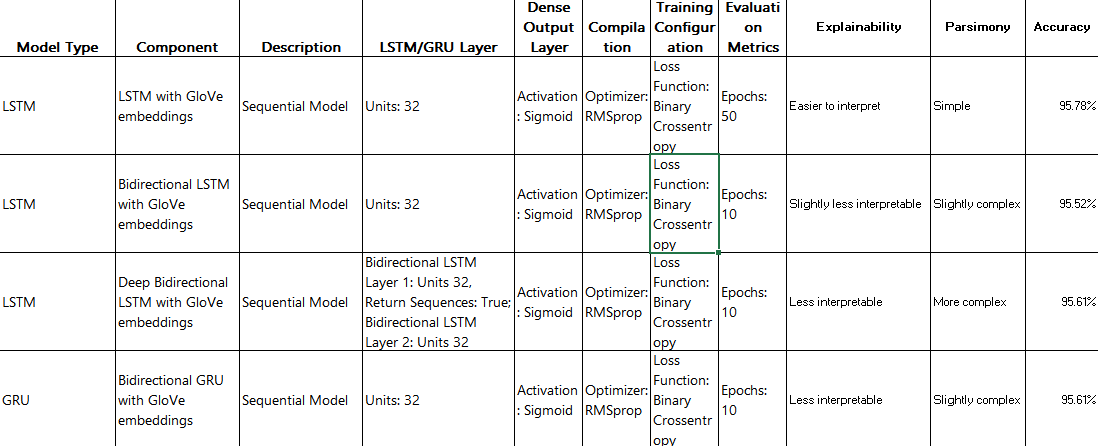
* + Now with respect to the **principle of parsimony** we can conclude the following:
  + It can be concluded that for RNN models Adam and SGD Optimisers aren’t suitable when it comes to high performing models.

## Visualize the attention weights of the RNN model: (In the. ipynb)

Bidirectional\_gru\_model Bidirectional RNN Model:



## Use different RNN architectures, such as LSTM or GRU, to improve the performance of the model:

Different RNN architectures - Summary for LSTM and GRU

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

# Part 3 – Image Classification using Transfer Learning

1. **Data preprocessing:** After importing the datasets.

Looking for the distribution of all the 6 classes in the training set Visual inspection of the classes.

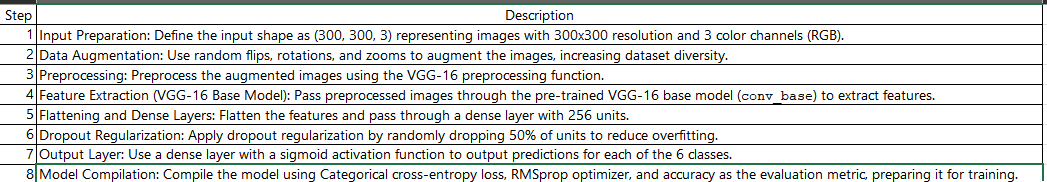
Splitting the dataset into Train and Test making sure the images of all type in both Train and Test

1. **Pre-trained model selection:** Load a pre-trained model **VGG-16.** (With 16 weight layers, the VGG-16 convolutional neural network design is renowned for its simplicity and depth. Due to its hierarchical and reliable structure of convolutional layers, it performs exceptionally well in tasks involving picture recognition.)

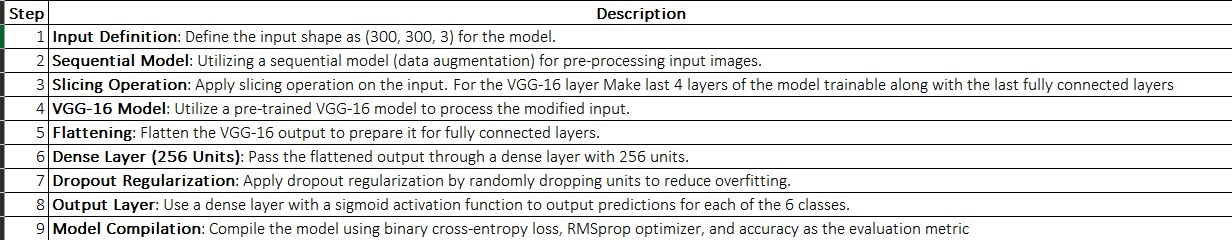
Pre-processed images of train and validate set using VGG-16's preprocessing function. Then generating high-level features that capture intricate patterns, textures, and shapes within each image of the datasets. These features are then utilized to train additional models, enhancing performance of classification.

### Model architecture modification:

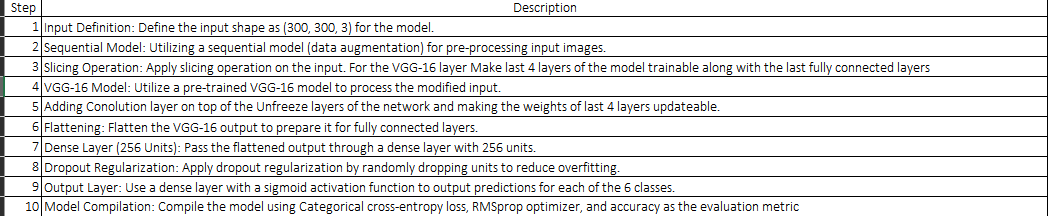
**Defining and Training the model:** This structure sets up an end-to-end model ready for training on image data, leveraging VGG-16 for feature extraction to enhance dataset diversity.



## Training and evaluation of the new model and fine-tuning:

Fine tuning of a pre trained model (VGG-16) by introducing freezing Freeze the last 4 layers of the model.

**Training and evaluation of the new model and fine-tuning: /** Adding a convolution network on top of the VGG-16 for better output:



## Summary:

Considering both accuracy and model parsimony, Model 3 would be the recommended choice as it achieves the highest accuracy with a simpler model structure. As Model 2 computationally heavy might have achieved better results if trained

for 10 or more epochs. Due to computation limitations, it has achieved 69.64% As it also has the minimum loss so might be a good choice for complex problems where accuracy is required.

 Note: Issue while running Imagenet. Multiple crashes and dead kernels.

1. **Bonus Points: Visualization of feature maps:** Code in the ipynb

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

# Part 4 – Fraud Detection using Graph Neural Networks

1. **Data pre-processing:** After importing the datasets.

Check for the balance of the classes with respect to the Target variable – Fraudulent and Non-Fraudulent. It can be concluded that the sample of Fradulent data is very less as compared to the other classes which might induce bias in our model training.

* + **Elimination of columns:** Removing the columns feature columns containing Id as they will not be contributing in the training of the Neural Net. Transaction Class – Fraud /Non-Fraud as that is the predicted or class label. Transaction\_dt as this feature is also not relevant for training our model.
  + **Creating the dummy variables:** For the categorical variables we will be creating dummy variables
  + **Handling missing values:** For the missing values with 0. In case of making of dummy variables this will make sure that any missing data in the original features are represented as zeros in the dummy variables.
  + **Scaling and Normalisation of features like Transaction Amount** we will be taking log of the amount to scale down
  + **Standardise** all the input which we will passing to the neural network’s input dimension layer
  + Create pytorch tensor consisting of features for each node as this will be the final input

## Model architecture:

**Input Dimensions:**INPUT\_DIM = 8 (dimensionality of the input features for each node in the graph)

**Hidden Dimensions**: HIDDEN\_DIM = 8(dimensionality of the hidden layers)

**Output Dimensions:** TARGET\_OUT\_DIM = 8 (target nodes)

**Convolutional Layers:** CONV\_LAYERS = 2 (There are 2 convolutional layers in this architecture.)

**Target Node Preprocessing:** TARGET\_PREPROCESSING\_HIDDEN\_DIM = 32

TARGET\_PREPROCESSING\_NO\_LAYERS = 3 (preprocessing of target nodes with 3 hidden layers of dim 32.)

**Target Node Postprocessing:** TARGET\_POSTPROCESSING\_HIDDEN\_DIM = 8 TARGET\_POSTPROCESSING\_NO\_LAYERS =2

(postprocessing of target node with 2 hidden layers of dimension 8)

**Using Training Configuration:** \* LEARNING\_RATE = 0.001 \* LOSS\_MULTIPLIER = 15 \* NUM\_EPOCHS = 30

## Training and evaluation:

**\* train\_one\_epoch:** For every epoch

Computes **forward pass** to get logits using the model.

Computes **probabilities using a sigmoid activation** function on logits. Computes training and validation loss using **Binary Cross Entropy Loss**. Computes training and validation **accuracy**.

**Backpropagates** and updates model parameters using the optimizer.

**\* train:** Iterates through the specified number of epochs. Calls train\_one\_epoch for each epoch. Keeps track of training loss, validation loss, validation accuracy, and the best validation accuracy seen so far.

## Experimentation:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Parameter** | **Experiment 1** | **Experiment 2** | **Experiment 3** | **Experiment 4** | **Experiment 5** | **Experiment 6** |
| INPUT\_DIM | 8 | 8 | 8 | 16 | 16 | 32 |
| HIDDEN\_DIM | 8 | 8 | 8 | 16 | 16 | 32 |
| TARGET\_OUT\_DIM | 8 | 8 | 8 | 8 | 8 | 8 |
| CONV\_LAYERS | 2 | 2 | 2 | 2 | 2 | 4 |
| TARGET\_PREPROCESSING\_HIDDEN\_DIM | 32 | 32 | 32 | 32 | 32 | 32 |
| TARGET\_PREPROCESSING\_NO\_LAYERS | 3 | 3 | 3 | 4 | 4 | 4 |
| TARGET\_POSTPROCESSING\_HIDDEN\_DIM | 8 | 8 | 8 | 8 | 8 | 8 |
| TARGET\_POSTPROCESSING\_NO\_LAYERS | 2 | 2 | 2 | 2 | 2 | 2 |
| LEARNING\_RATE | 0.001 | 0.1 | 0.001 | 0.001 | 0.1 | 0.1 |
| NUM\_EPOCHS | 30 | 30 | 30 | 30 | 30 | 30 |
| Optimiser | Adam | Adam | RMSprop | Adam | RMSprop | SGD |
| Loss function | BCELoss | BCELoss | BCEWithLogitsLoss | BCELoss | BCEWithLogitsLoss | BCEWithLogitsLoss |
| Best validation accuracy | 0.955 | 0.966 | 0.948 | 0.856 | 0.966 | 0.964 |
| Final loss | 0.965 | 1.005 | 1.022 | 0.922 | 1.058 | 1.028 |

Note: As for few combinations run was giving CPU usage error.

1. **Visualization of learned embeddings:** Code and Visuals are in the. ipynb file

## Summary:

Parsimonious model would imply a simpler model that achieves good results.

The model from **Experiment 1** lower complexity comparatively low validation accuracy **– 0.955**. **Experiment 1, Experiment 2, Experiment 3: 696 parameters**

With respect to the Validation Accuracy **in terms of validation accuracy, the ranking is Experiment 2, Experiment 6, Experiment 1, Experiment 3, Experiment 4.**

Most Complex with best validation accuracy – Experiment 6 – 0.964 So SGD Optimiser can be considered with other combinations hyperparameters.

### Considering both parsimony and validation accuracy following models can be considered Experiment 2, Experiment 1, Experiment 6.

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***