Validation Accuracy Analysis on imdb movie review data

Abstract

In this Assignment, we are going to explore various hyper parameter tunings to observe the variations in the Validation Accuracy. We look at imdb movie reviews dataset, one with binary labels, which is already available in keras library. For the binary classification we subjected the model to regularizations methods, dropout method and observed changes by tuning hyperparameters. Most of the model findings are compared with the deep learning model that have downloaded from github.

Introduction

Hyperparameter tuning is a well-known task in the realm of Neural Networks. Given a set of reviews, the objective is to determine the type of reviews provides a comprehensive analysis of various methods, benchmarks, and loads of preprocessing. The review can consist of binary classes. It is whether A movie review is positive (+) or negative (-). There are several methods and layer embeddings that drastically improves the performance measure. But, for this analysis we will perform without any layer embeddings.

The challenge consists of three main parts. In the first part, we try a variety of basic tuning of layers, units, batch size. This provides a reasonable baseline to asses further complex methods. In the second part, we try different variants of the activation functions and loss functions. In the third part, we will try different models with dropout, regularizations. *Since the objective of this analysis is to only tune the hyperparameter to observe changes in the validation Accuracy of train data set and presented it using graphs and tables, we won't be using test set.*

Occam's razor Quote:

"*When you have two competing hypotheses that make the same predictions, the simpler one is better*"

With that being said, the model that I have downloaded from git is Simple yet efficient network. Hence, Each and every change/tweaking that I made is not a cumulative model building but an enhancement over the present.

```
library(keras)

## Warning: package 'keras' was built under R version 3.6.2

imdb <- dataset_imdb(num_words = 10000)
c(c(train_data, train_labels), c(test_data, test_labels)) %<-% imdb
```

```
#Padding list of integers to tensors and dividing the training set to training and va
lidation sets

vectorize_sequences <- function(sequences, dimension = 10000) {
  # Create an all-zero matrix of shape (len(sequences), dimension)
  results <- matrix(0, nrow = length(sequences), ncol = dimension)
  for (i in 1:length(sequences))
    # Sets specific indices of results[i] to 1s
    results[i, sequences[[i]]] <- 1
  results
}

# Our vectorized training data
x_train <- vectorize_sequences(train_data)
# Our vectorized test data
x_test <- vectorize_sequences(test_data)

#We should also vectorize our labels, which is straightforward:
# Our vectorized labels

y_train <- as.numeric(train_labels)
y_test <- as.numeric(test_labels)

#In order to monitor during training the accuracy of the model on data that it has ne
ver seen before, we will create a "validation set" by setting apart 10,000 samples fr
om the original training data:

val_indices <- 1:10000

x_val <- x_train[val_indices,]
partial_x_train <- x_train[-val_indices,]

y_val <- y_train[val_indices]
partial_y_train <- y_train[-val_indices]
```

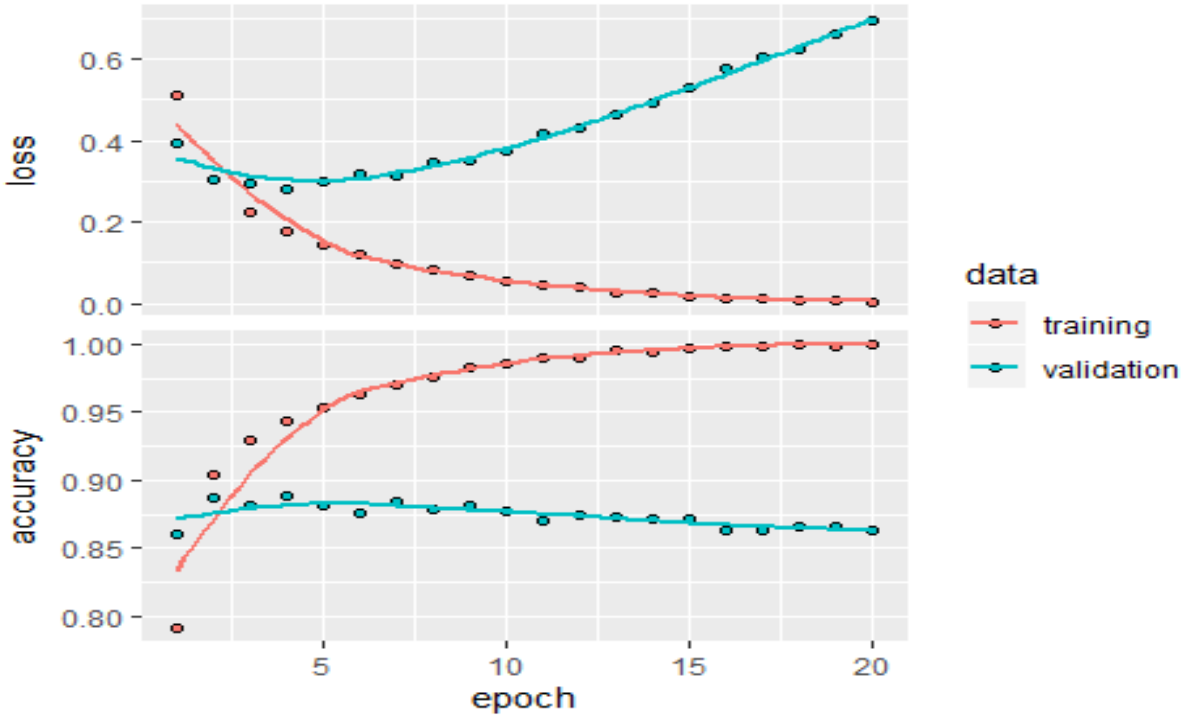
Building a simple and less complex network with one hidden layer:

```
library(keras)

model <- keras_model_sequential() %>%
  layer_dense(units = 16, activation = "relu", input_shape = c(10000)) %>%
  layer_dense(units = 16, activation = "relu") %>%
  layer_dense(units = 1, activation = "sigmoid")

model %>% compile(
  optimizer = "rmsprop",
  loss = "binary_crossentropy",
  metrics = c("accuracy")
)
```

```
history <- model %>% fit(
  partial_x_train,
  partial_y_train,
  epochs = 20,
  batch_size = 512,
  validation_data = list(x_val, y_val)
)
```

#Storing the history metrics for further analysis

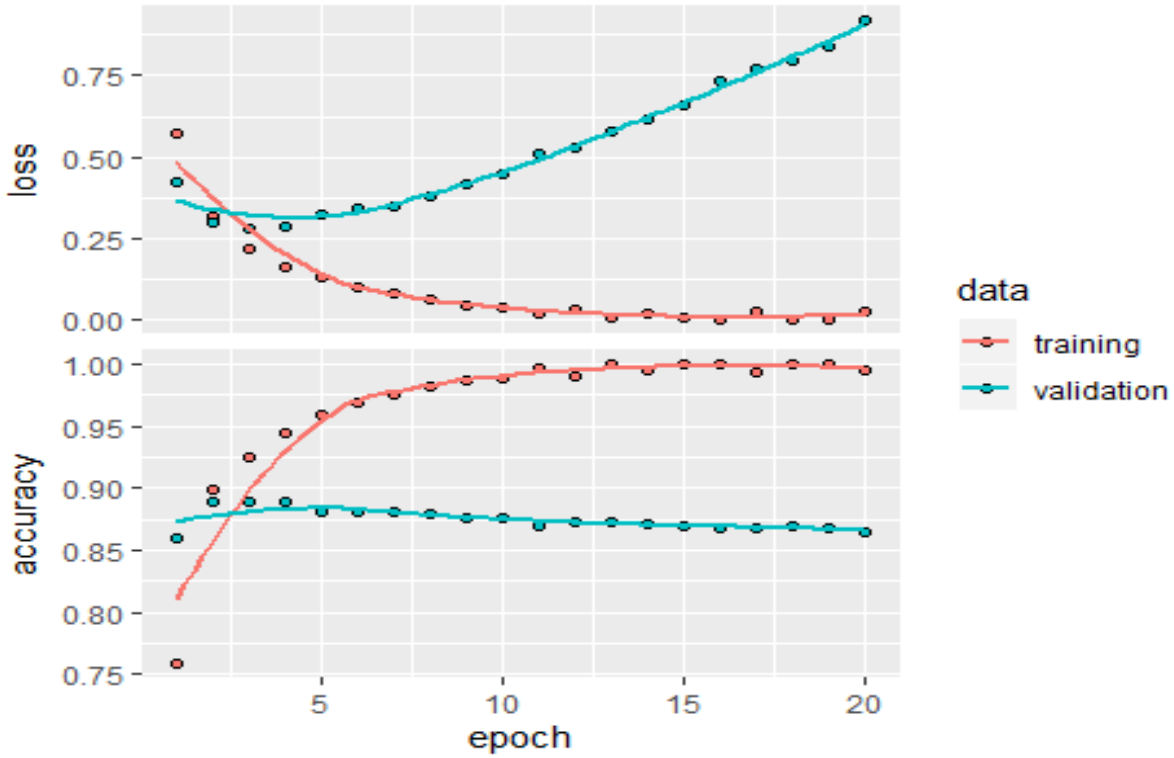
```
History<-data.frame(history$metrics)
plot(history)
```



1. Try using one or three hidden layers and see how doing so affects validation and test accuracy.

```
library(keras)

model_Layers <- keras_model_sequential() %>%
  layer_dense(units = 16, activation = "relu", input_shape = c(10000)) %>%
  layer_dense(units = 16, activation = "relu") %>%
  layer_dense(units = 16, activation = "relu") %>%
  layer_dense(units = 16, activation = "relu") %>%
  layer_dense(units = 1, activation = "sigmoid")
```

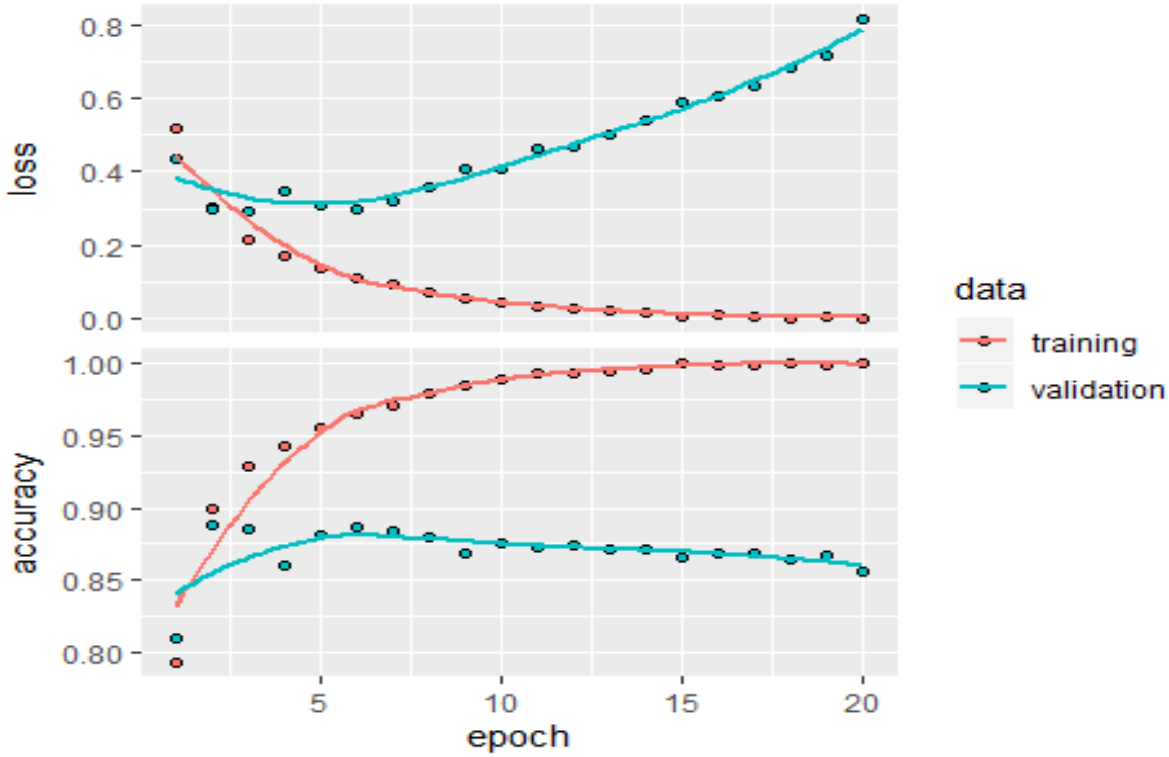
```
model_Layers %>% compile(
  optimizer = "rmsprop",
  loss = "binary_crossentropy",
  metrics = c("accuracy")
)

history_Layers <- model_Layers %>% fit(
  partial_x_train,
  partial_y_train,
  epochs = 20,
  batch_size = 512,
  validation_data = list(x_val, y_val)
)
Hist_Layers<-data.frame(history_Layers$metrics)
plot(history_Layers)
```



2. Try using layers with more hidden units or fewer hidden units: 32 units, 64 units, and so on.We will initially build a model with

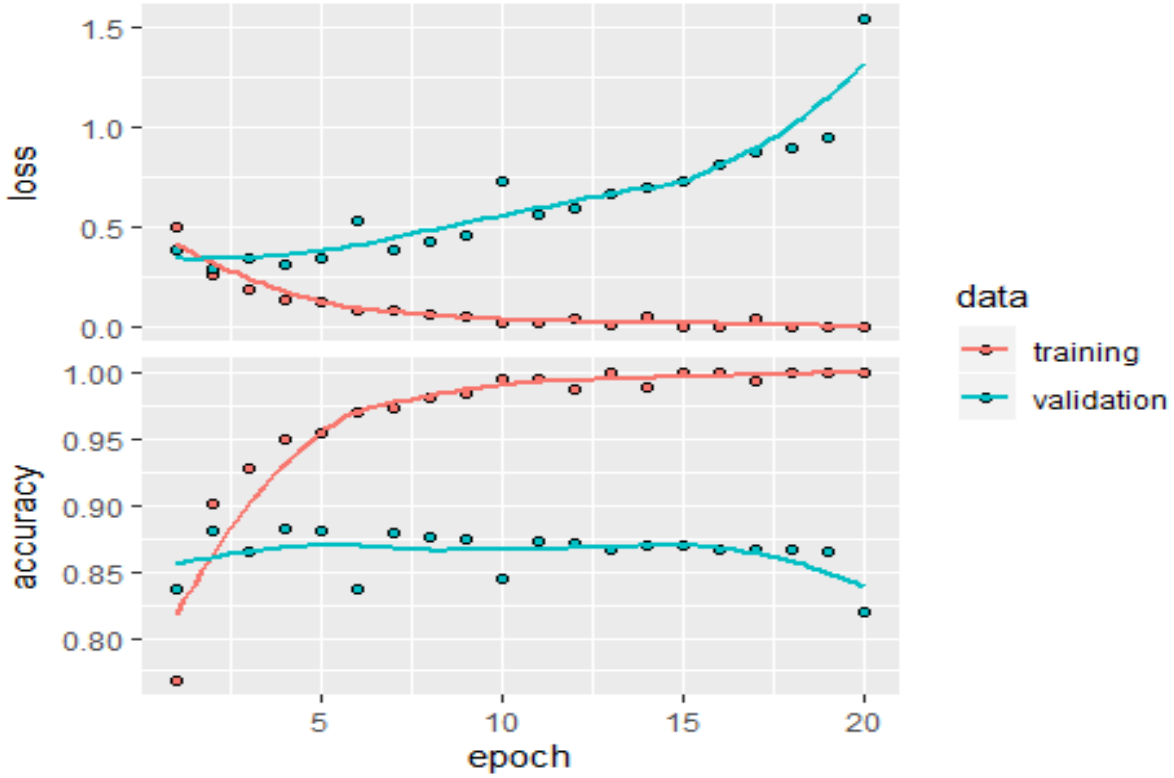
```
library(keras)

model_32Units <- keras_model_sequential() %>%
  layer_dense(units = 16, activation = "relu", input_shape = c(10000)) %>%
  layer_dense(units = 32, activation = "relu") %>%
  layer_dense(units = 1, activation = "sigmoid")

model_32Units %>% compile(
  optimizer = "rmsprop",
  loss = "binary_crossentropy",
```

```
  metrics = c("accuracy")
)

history_32Units <- model_32Units %>% fit(
  partial_x_train,
  partial_y_train,
  epochs = 20,
  batch_size = 512 ,
  validation_data = list(x_val, y_val)
)
Hist_32Units<-data.frame(history_32Units$metrics)
plot(history_32Units)
```




```
library(keras)

model_64Units <- keras_model_sequential() %>%
  layer_dense(units = 16, activation = "relu", input_shape = c(10000)) %>%
  layer_dense(units = 64, activation = "relu") %>%
  layer_dense(units = 64, activation = "relu") %>%
  layer_dense(units = 1, activation = "sigmoid")

model_64Units %>% compile(
  optimizer = "rmsprop",
  loss = "binary_crossentropy",
  metrics = c("accuracy")
)

history_64Units <- model_64Units %>% fit(
```

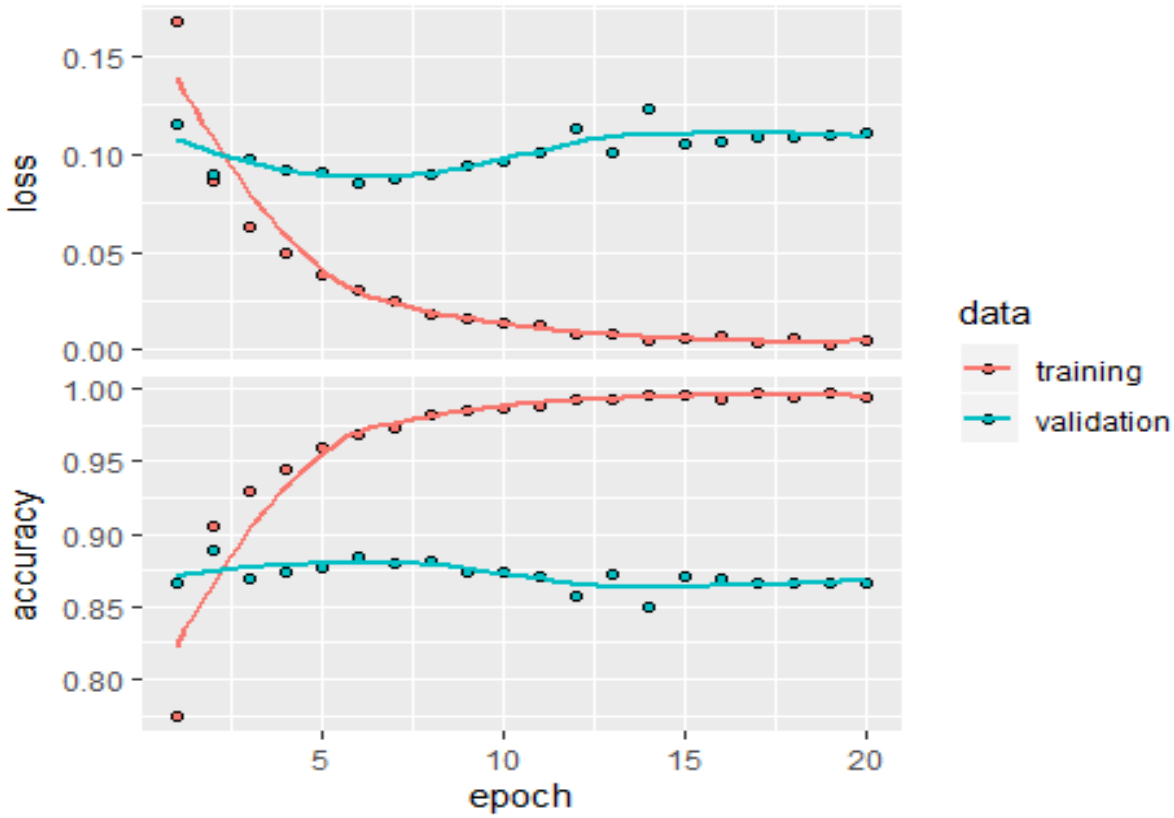
```
  partial_x_train,
  partial_y_train,
  epochs = 20,
  batch_size = 512 ,
  validation_data = list(x_val, y_val)
)
Hist_64Units<-data.frame(history_64Units$metrics)
plot(history_64Units)
```



3. Try using the mse loss function instead of binary_crossentropy.

```
library(keras)

model_LF <- keras_model_sequential() %>%
  layer_dense(units = 16, activation = "relu", input_shape = c(10000)) %>%
  layer_dense(units = 16, activation = "relu") %>%
  layer_dense(units = 1, activation = "sigmoid")

model_LF %>% compile(
  optimizer = "rmsprop",
  loss = "mse",
  metrics = c("accuracy")
)

history_LF <- model_LF %>% fit(
  partial_x_train,
  partial_y_train,
  epochs = 20,
```

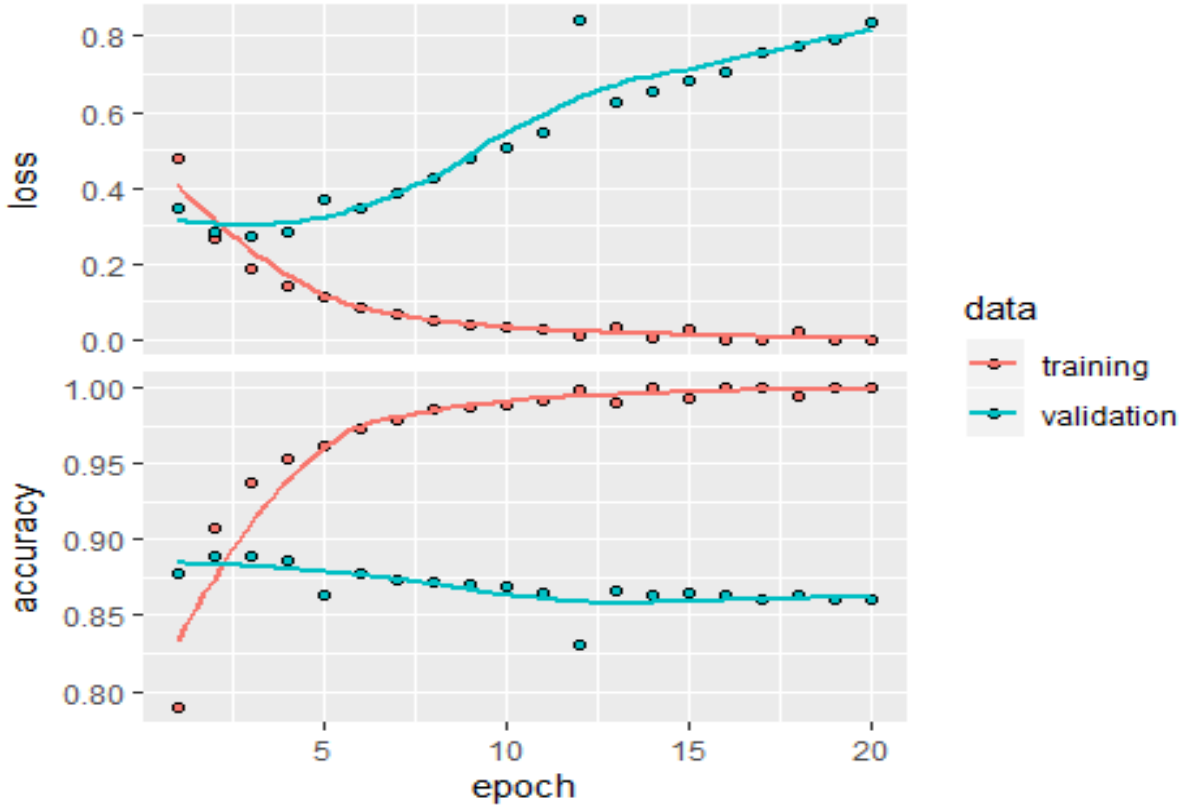
```
  batch_size = 512,
  validation_data = list(x_val, y_val)
)
Hist_LF<-data.frame(history_LF$metrics)
plot(history_LF)
```



4. Try using the tanh activation (an activation that was popular in the early days of neural networks) instead of relu.

```
library(keras)

model_AF <- keras_model_sequential() %>%
  layer_dense(units = 16, activation = "tanh", input_shape = c(10000)) %>%
  layer_dense(units = 16, activation = "tanh") %>%
  layer_dense(units = 1, activation = "sigmoid")

model_AF %>% compile(
  optimizer = "rmsprop",
  loss = "binary_crossentropy",
  metrics = c("accuracy")
)

history_AF <- model_AF %>% fit(
  partial_x_train,
  partial_y_train,
  epochs = 20,
  batch_size = 512,
```

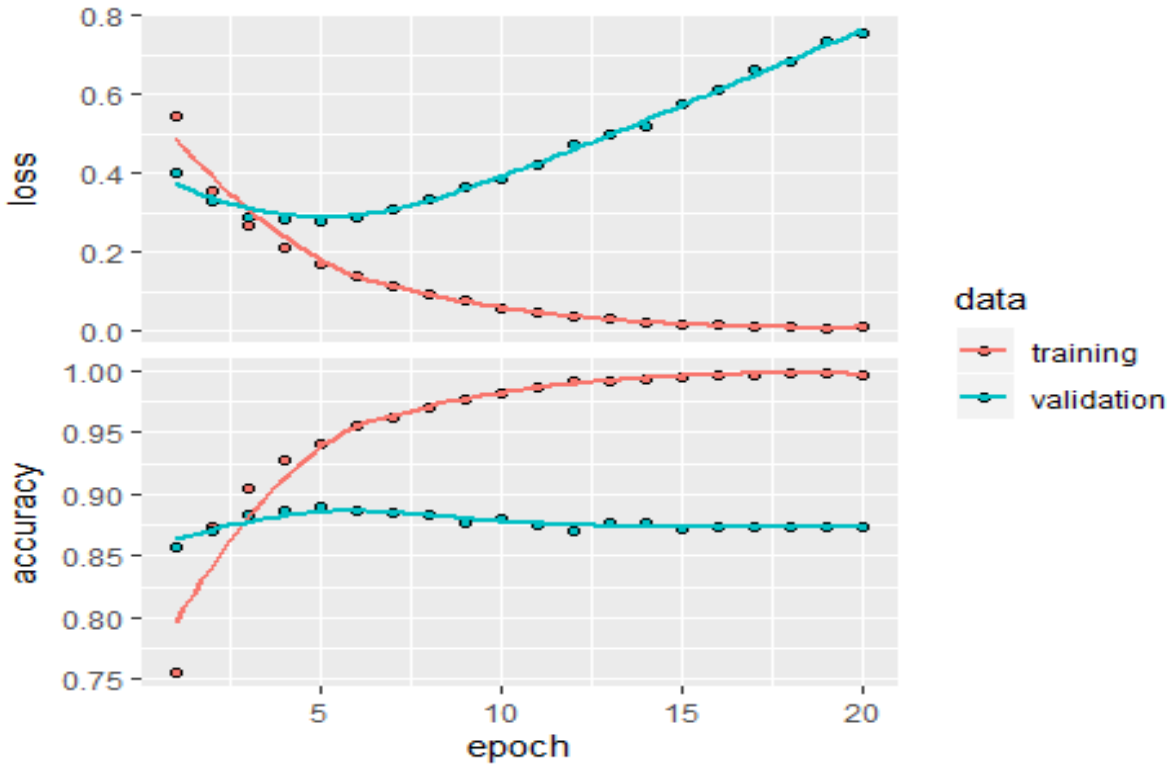
```
  validation_data = list(x_val, y_val)
)
Hist_AF<-data.frame(history_AF$metrics)
plot(history_AF)
```



- Best performing models tends to be large but trained is such a way that restricts the utilization of their entire potential
- One of the reasons we subject our models to regularization is to restrict the model to Fewer parameters which in-turn demands less computational power
- Regularization encourages models to have a preference towards simpler models and reduces the risk of overfitting

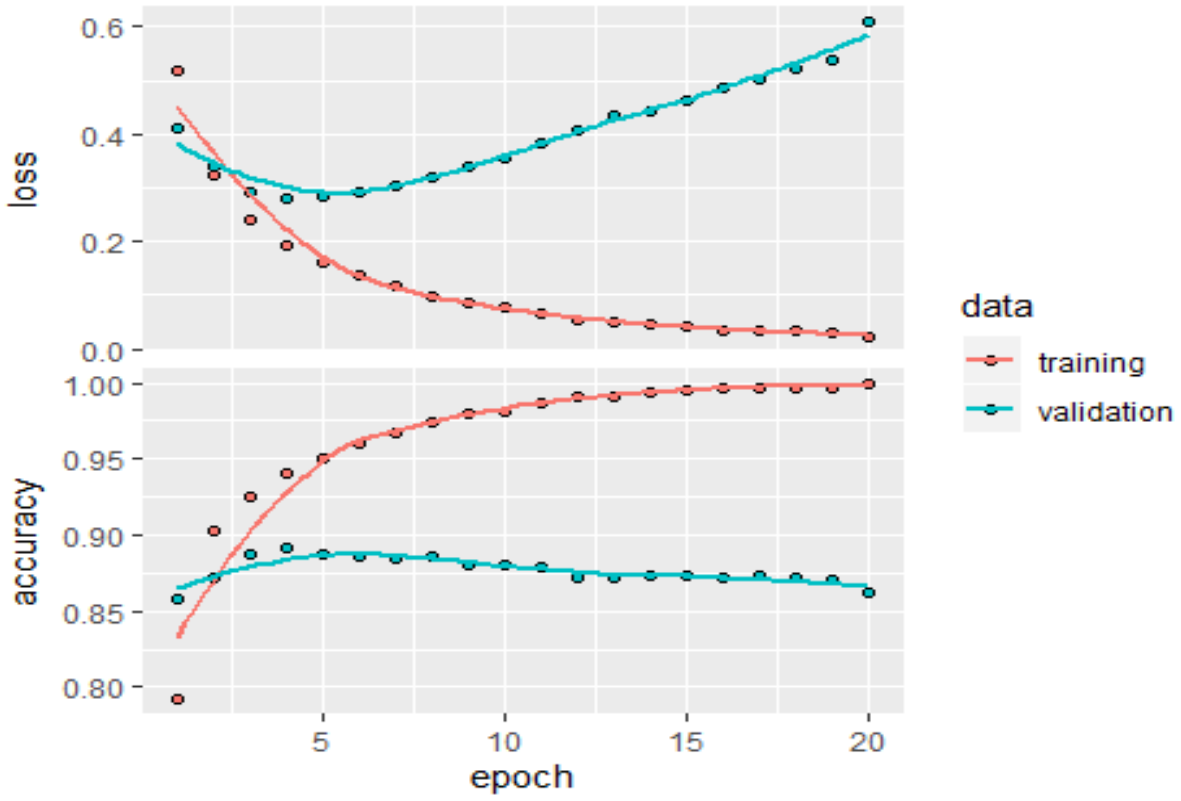
```
5. Use any technique we studied in class, and these include regularization, dropout,
etc., to get your model to perform better on validation.

##With Dropout Layer

model_Drop <- keras_model_sequential() %>%
  layer_dense(units = 16, activation = "relu", input_shape= c(10000)) %>% layer_dropo
ut(rate = 0.15) %>%
  layer_dense(units = 16, activation = "relu") %>% layer_dropout(rate = 0.15) %>%
  layer_dense(units = 1, activation = "sigmoid")
```

```
model_Drop %>% compile(
  optimizer = "rmsprop",
  loss = "binary_crossentropy",
  metrics = c("accuracy")
)

history_Drop <- model_Drop %>% fit(
  partial_x_train,
  partial_y_train,
  epochs = 20,
  batch_size = 512,
  validation_data = list(x_val, y_val)
)
Hist_Drop<-data.frame(history_Drop$metrics)
plot(history_Drop)
```



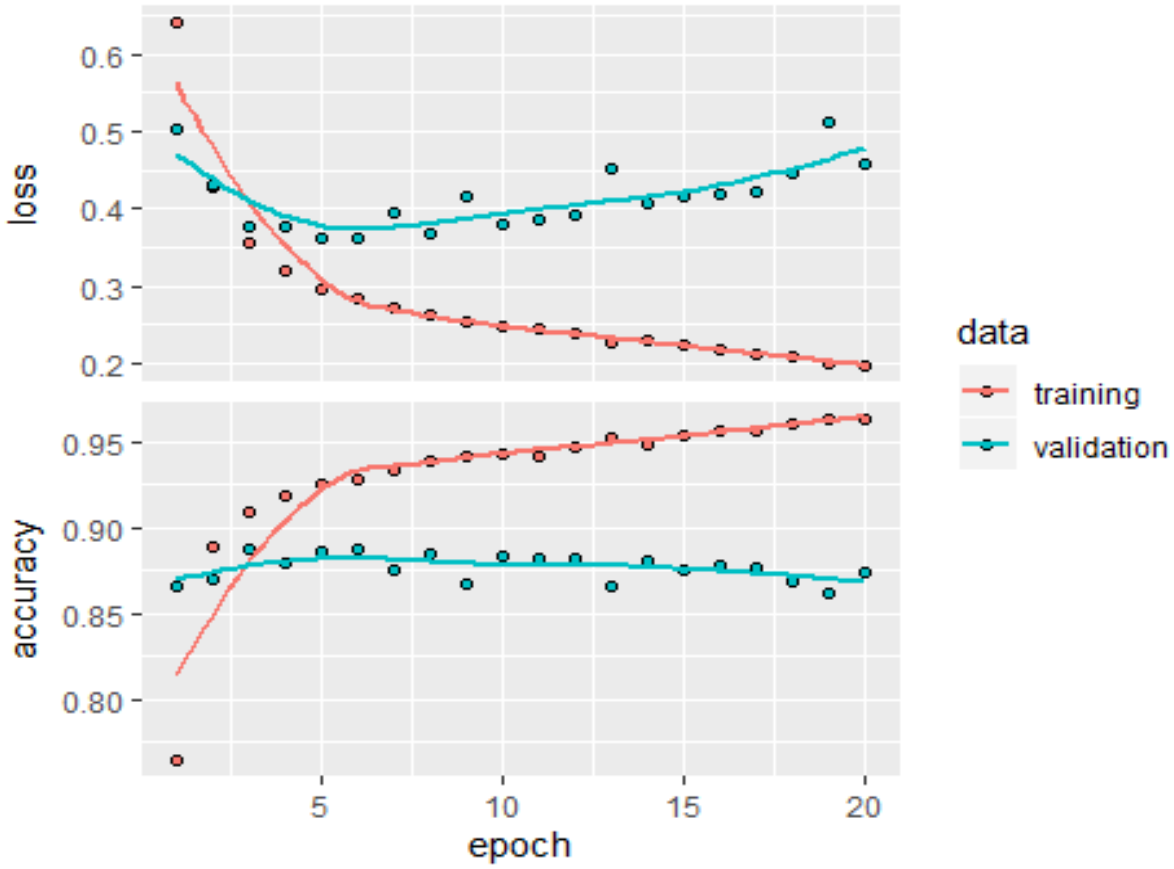

```
##With Ridge Regression as penalty

model_Ridge <- keras_model_sequential() %>%
  layer_dense(units = 16,kernel_regularizer = regularizer_l2(0.00005),activation = "r
elu", input_shape= c(10000)) %>%
  layer_dense(units = 16,kernel_regularizer = regularizer_l2(0.00005), activation = "
relu") %>%
  #layer_dense(units = 8,kernel_regularizer = regularizer_l2(0.001), activation = "re
lu") %>%
  layer_dense(units = 1, activation = "sigmoid")

model_Ridge %>% compile(
```

```
  optimizer = "rmsprop",
  loss = "binary_crossentropy",
  metrics = c("accuracy")
)

history_Ridge <- model_Ridge %>% fit(
  partial_x_train,
  partial_y_train,
  epochs = 20,
  batch_size = 512,
  validation_data = list(x_val, y_val)
)
Hist_Ridge<-as.data.frame(history_Ridge$metrics)
plot(history_Ridge)
```




```
##With Lasso as penalty

model_Lasso <- keras_model_sequential() %>%
  layer_dense(units = 16,kernel_regularizer = regularizer_l1(0.0001),activation = "re
lu", input_shape= c(10000)) %>%
  layer_dense(units = 16,kernel_regularizer = regularizer_l1(0.0001), activation = "r
elu") %>%
  #layer_dense(units = 8,kernel_regularizer = regularizer_l1(0.0001), activation = "r
elu") %>%
  layer_dense(units = 1, activation = "sigmoid")

model_Lasso %>% compile(
```

```
  optimizer = "rmsprop",
  loss = "binary_crossentropy",
  metrics = c("accuracy")
)

history_Lasso <- model_Lasso %>% fit(
  partial_x_train,
  partial_y_train,
  epochs = 20,
  batch_size = 512,
  validation_data = list(x_val, y_val)
)
Hist_Lasso<-data.frame(history_Lasso$metrics)
plot(history_Lasso)
```



Visualizing the Findings Through Line Graphs:

```
Models_Val_Accuracy<-data.frame(epochs=1:10,History[1:10,4],Hist_32Units[1:10,4],Hist
_64Units[1:10,4],Hist_Layers[1:10,4],Hist_AF[1:10,4],Hist_LF[1:10,4],Hist_Drop[1:10,4
],Hist_Ridge[1:10,4],Hist_Lasso[1:10,4])

colnames(Models_Val_Accuracy)<-c("epochs","Model Val Acc","Units 32","Units 64","Hidd
en Layers 3","Tanh AF","MSE LF","DrouOut","Ridge regularizer","Lasso regularizer")
```

```
library(data.table)
Model_ValAcc_transpose <- transpose(Models_Val_Accuracy)
rownames(Model_ValAcc_transpose)<-colnames(Models_Val_Accuracy)
colnames(Model_ValAcc_transpose)<-rownames(Models_Val_Accuracy)
```

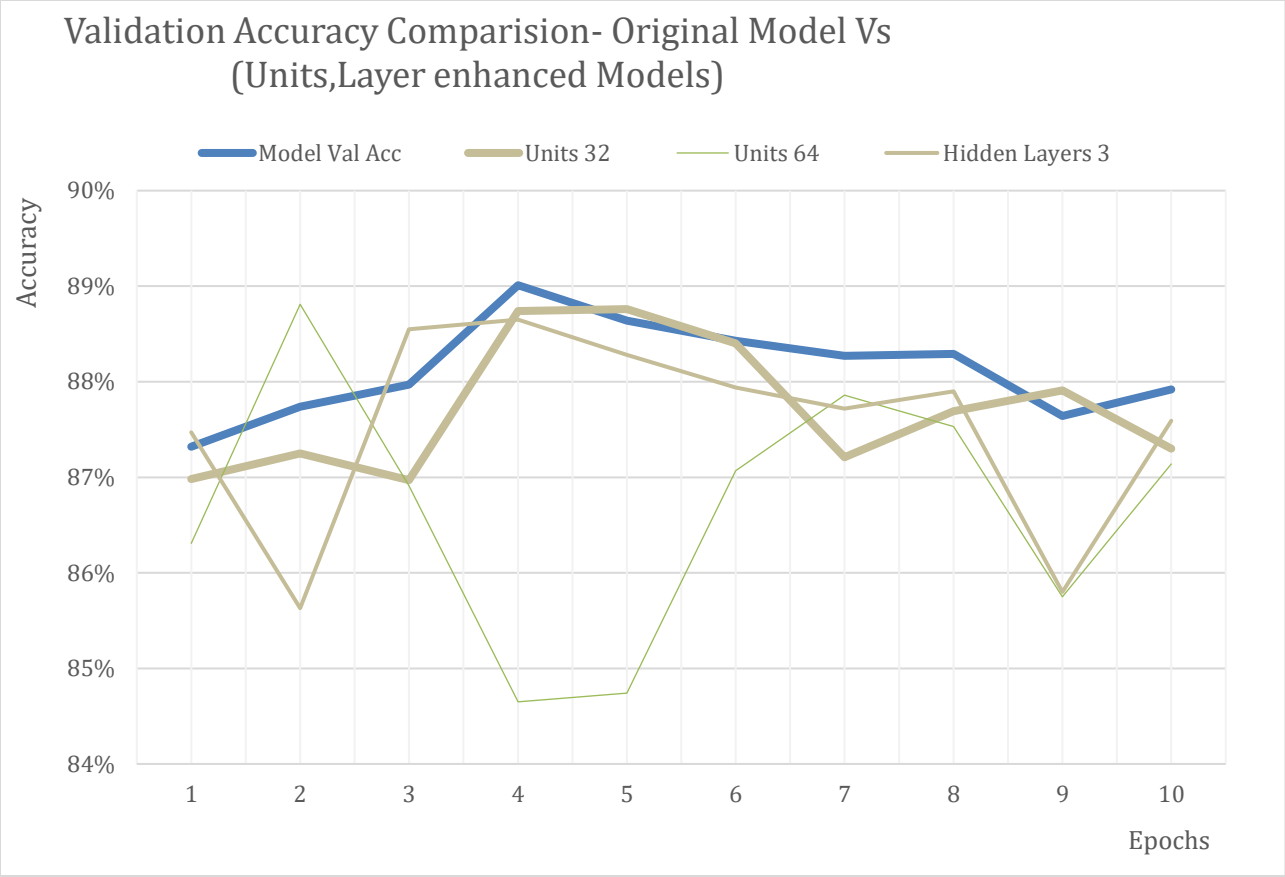
The Table below Summarizes the Validation Accuracy for 10 Epochs for different Tunings of H yper Parameter. Here, I have considered Up 10 epochs because, the Fluctuation in Validation acc uracy either be constant or too dynamic.

** Validation accuracies above 10% were highlighted in the table

Epoch	Model Val Acc	Units 32	Units 64	Hidden Layers 3	Tanh AF	MSE LF	DropOut	Ridge regularizer	Lasso regularizer
1	87.32%	86.98%	86.31%	87.47%	87.77%	86.29%	83.91%	87.12%	82.28%
2	87.74%	87.25%	88.81%	85.63%	88.47%	88.90%	88.37%	88.39%	88.43%
3	87.97%	86.97%	86.91%	88.55%	88.90%	89.26%	88.30%	89.04%	88.13%
4	89.01%	88.74%	84.65%	88.65%	88.85%	87.89%	89.13%	88.06%	88.66%
5	88.64%	88.76%	84.74%	88.28%	88.63%	87.26%	89.29%	88.78%	87.65%
6	88.43%	88.40%	87.07%	87.94%	87.93%	88.28%	88.77%	88.58%	88.16%
7	88.27%	87.21%	87.86%	87.72%	86.41%	87.94%	88.73%	88.44%	87.92%
8	88.29%	87.69%	87.53%	87.90%	86.43%	88.06%	88.63%	88.21%	88.35%
9	87.64%	87.91%	85.75%	85.80%	87.32%	87.49%	88.21%	87.51%	88.10%
10	87.92%	87.30%	87.14%	87.59%	87.07%	87.52%	88.20%	87.28%	88.20%

We can further evaluate the table through Line graphs. There are three significant comparisons m ade between the validation accuracies of nine different models. Each and every model is compar ed with the original model to get a perspective of better performing models after and before enha ncing.

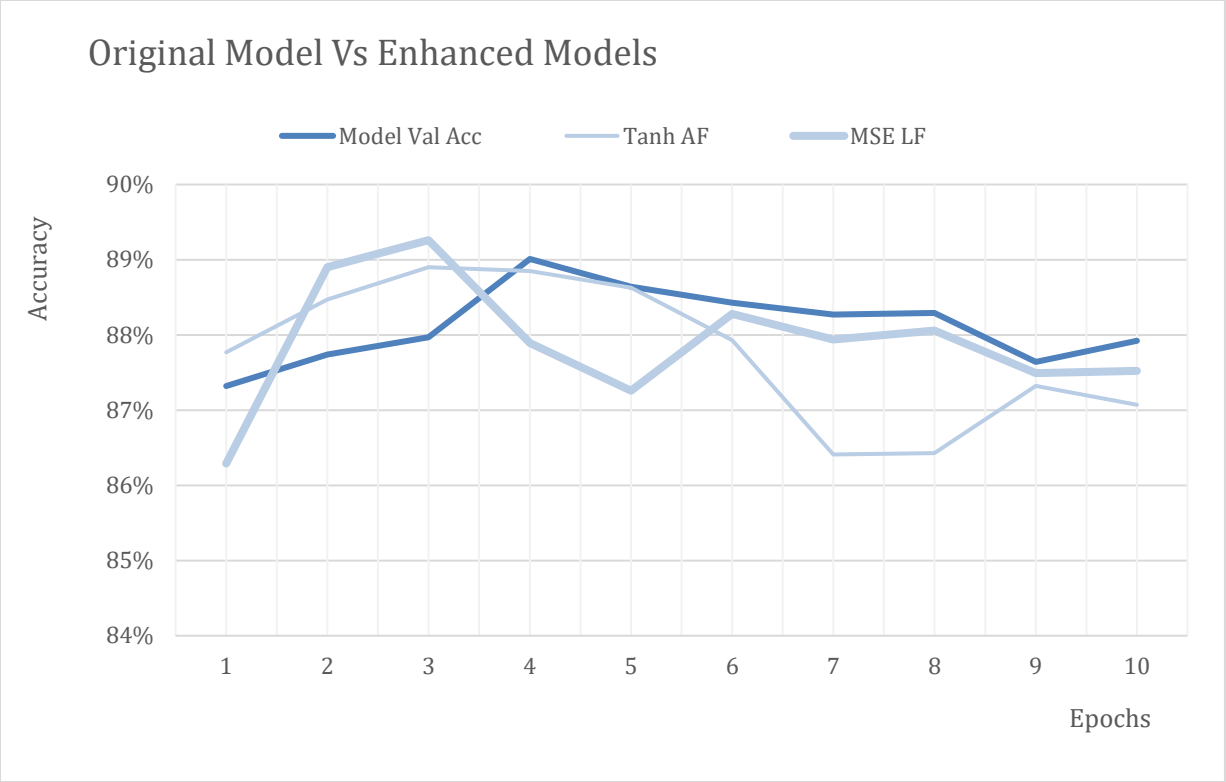
The first comparison is made between the original standard model and models with enhanced uni ts, hidden layers. Below line graph, we can observe the consistencies through every epoch.



It seems, none of the above model tend to perform well with the change in the units or layers in the model network. Hence, one will only complicate the model if he/she tris to enhance the validation accuracy.

below Line graph is compared between the original model and models with Tanh activation function and ' Mean Squared Error' loss function. Choosing the right activation function could yield greater enhanceme nt in accuracy but, 'ReLu' is by far the best in terms of finding local/global minima thus, minimizing the error.

Let's have look at the comparison between models with 'ReLu', Tanh and MSE.



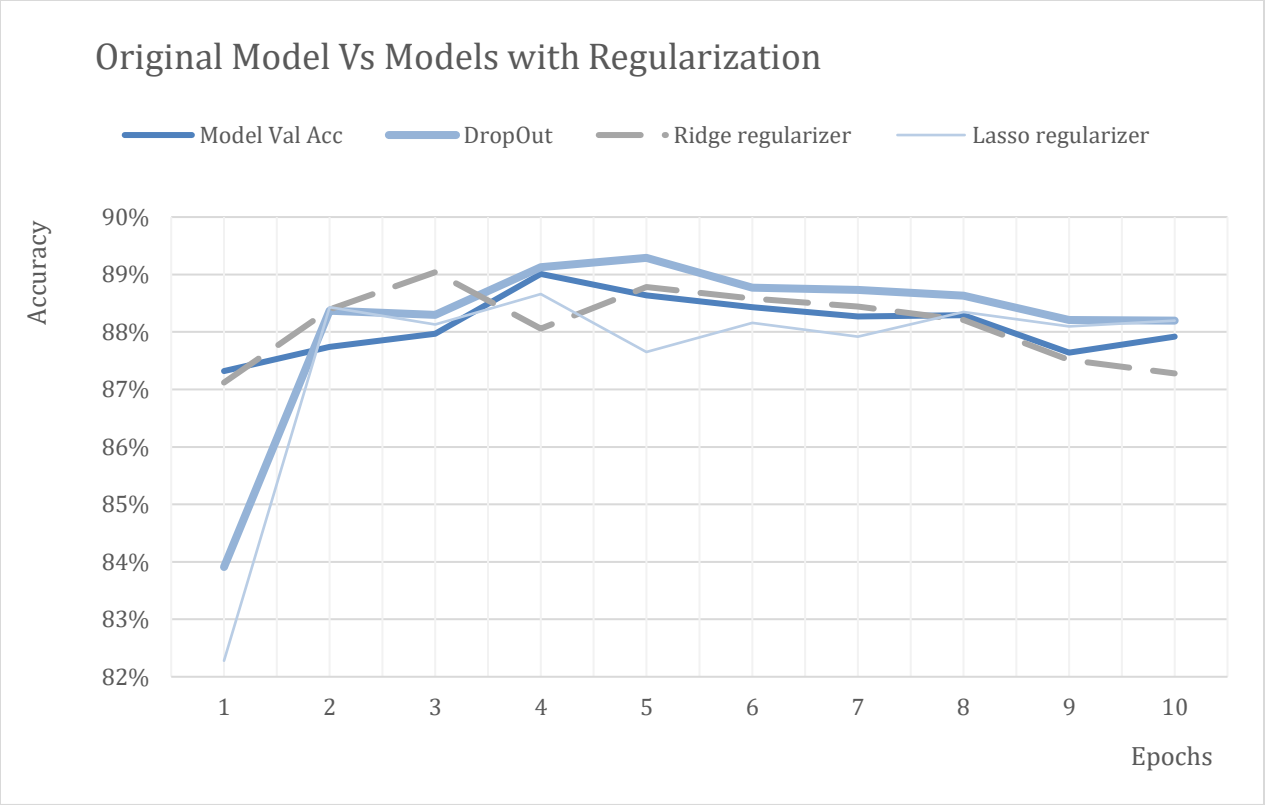
Though the model with MSE loss function yielded good Validation accuracy at epoch 3, its is not a good practice to use MSE for a classification prediction.

Tanh Activation function is not consistent through out the epochs which suggests it is dynamic. The fore most reason for this function to be disapproved is because of the trouble caused by vanishing gradients.

Finally, we will apply regularization methods to the original model and compare it with the original mode l to see if there is any significant change in accuracy.

few reasons we regularize our model is,

- To restrict the model to Fewer parameters which in-turn demands less computational power
- Regularization encourages models to have a preference towards simpler models and reduces the ri sk of overfitting



CONCLUSIONS:

- As we observed, applying regularizations to the models not only improves the Accuracy of the model but also helps the model to capture the Non-Linearities of the function by pe nalizing it.
- In our case, Drop-out layer performed well above all the models including the original m odel.
- Accuracies of other models can be improved but, as I said we end up making the model more complex and computationally inefficient. That's a trade-off and is subjective for ma ny reasons.
- Since our dataset is small, for all the models the best accuracies can be observed in less th en six epochs with out any over/under fitting(Bias-Variance Tradeoff)
- The could be even more scope of increasing the model accuracies by adding an Embeddi ng Layer(GLoVe)