Project Report-IMDB Sentiment Analysis

Curios_i

11/04/2021

Contents

1 Introduction

Sentiment Analysis is a common natural language processing task that Data Scientist need to perform. In this project, a sentiment classifier is built which evaluates the polarity of piece of text being positive or negative. We have selected IMDB dataset: a set of 50,000 reviews from Internet Movie Database. We shall be using different deep machine learning (Deep Neural Network) methods to classify movie reviews as positive or negative. The main objective of this project is to learn different deep learning techniques, though they were not taught in the HarvardX Data Science course.

This report in .rmd and pdf form as well as R code can also be downloaded from github repository.

2 Implementation Notes

2.1 Stochastic Nature of Neural Networks

Neural networks use randomness by design to ensure they effectively learn the function being approximated for the problem. Randomness is used because this class of machine learning algorithm performs better with it than without.

The most common form of randomness used in neural networks is the random initialization of the network weights. Although randomness can be used in other areas, here is just a short list:

- Randomness in Initialization, such as weights.
- Randomness in Regularization, such as dropout.
- Randomness in Layers, such as word embedding.
- Randomness in Optimization, such as stochastic optimization.
- Randomness introduced due to parallel processing in CPU and GPU.

These sources of randomness, and more, mean that when you run the exact same neural network algorithm on the exact same data, you are guaranteed to get different results. The model tried in this study are no exception, so definitely when this code is run, you may get a little different results than reported here.

The keras FAQ article How can I obtain reproducible results using keras during development discusses methods to get reproducible results. However, one source of randomness is GPU computation and CPU parallelization, and they must be disabled in the method mentioned to get reproducible results. Since execution of deep neural network requires a lot of computation power and much slower on CPUs (as we used in most of the models in this study), the method was not tried in this study.

2.2 CPU Vs GPU Computation

DNN Applications—in particular, image processing with convolutional networks and sequence processing with recurrent neural networks will be excruciatingly slow on CPU, even a fast multicore CPU. Even for applications that can realistically be run on CPU, you'll generally see speed increase by a factor or 5 or 10 by using a modern GPU. Most of the models in this study were run on CPU, only the LSTM model was finally run on GPU, since it was painfully slow on CPU and even was crashing with 8G of memory. That's why you will notice that every few epochs are used in training the models. This is also discussed in Conclusion as an opportunity.

We are not covering the installation of Cuda, cuDNN in this report, though it would have been an interesting topic considering very little and correct information is available on this topic for RStudio; most of the guidelines are for python/anaconda applications. All the models in this project were run on CPU, so it may not be required to test the code, except for the last LSTM model which was run on GPU. Even if Cuda and cuDNN is not installed, that model will still run on CPU, but it will be very slow, resource and time consuming.

2.3 Reproducing the Results

It is not simple and direct to reproduce the results, as mentioned in this report. Since it is not easy to write a rmarkdown report with DNN code by executing the code again and again, the models in this report were executed and results were saved in `rda` files, which were loaded in rmarkdown to generate report. The size of `rda` files and downloaded database is too large that github doesn't allow pushing them, so they were not synced with github. However, interested readers can download the database from Stanford University link provided and run this code on their environment. There will also be some variation required in the code running it on CPU Vs GPU.

3 Data Analysis

3.1 Grabbing the Data

IMDB movie review dataset is available from different sources. However, we have used the dataset from Stanford University in this project.

3.1.1 Stanford University

In the project we have used the dataset from Stanford University. It can be downloaded from:

http://ai.stanford.edu/~amaas/data/sentiment/aclImdb_v1.tar.gz.

The first step is to download the aclImdb_v1.tar.gz file and unzip it into ~/Downloads/ directory. The original aclImdb folder has separate folders for train and test data.

3.1.2 Kaggle

IMDB dataset is also available from Kaggle in CSV form. It can be downloaded from :

https://www.kaggle.com/lakshmi25npathi/imdb-dataset-of-50k-movie-reviews

3.1.3 Keras Library

Another source of IMDB dataset is Keras R library. Keras is the package we have used in this project for deep neural networks.

3.2 Data Analysis

The core dataset contains 50,000 reviews split evenly into 25k train and 25k test sets. The overall distribution of labels is balanced (25k pos and 25k neg). The data also includes an additional 50,000 unlabeled documents for unsupervised learning (not relevant to this project).

In the entire collection, no more than 30 reviews are allowed for any given movie because reviews for the same movie tend to have correlated ratings. Further, the train and test sets contain a disjoint set of movies, so no significant performance is obtained by memorizing movie-unique terms and their associated with observed labels. In the labeled train/test sets, a negative review has a score <= 4 out of 10, and a positive review has a score >= 7 out of 10. Thus reviews with more neutral ratings are not included in the train/test sets.

3.2.1 Files

There are two top-level directories [train/, test/] corresponding to the training and test sets. Each contains [pos/, neg/] directories for the reviews with binary labels positive and negative. Within these directories, reviews are stored in text files named following the convention [[id]_[rating].txt] where [id] is a unique id and [rating] is the star rating for that review on a 1-10 scale. For example, the file [test/pos/200_8.txt] is the text for a positive-labeled test set example with unique id 200 and star rating 8/10 from IMDb. The [train/unsup/] directory has 0 for all ratings because the ratings are omitted for this portion of the dataset.

Since machine learning algorithms work better when we have more training data as compared to test data, we have merged the training and test sets into one folder, data, and then we shall use caret package to randomly generate a training set of 60% and a test set of 40%.

While merging train and test folders, we also found that there were 3,540 reviews with the same file names in both pos and neg folders. So, after merging train and test folders, the data folder contains pos and neg folders, each with 21,461 text files (each file containing a review).

3.3 Preparing the Data

In preparing the data, we transfer reviews texts from ~/Downloads/aclImdb folder into a "texts" string variable, while the sentiment of the review is stored in the "labels" variable. Following is the code.

```
if (!require(keras)) install.packages('keras')
if (!require(tidyverse)) install.packages('tidyverse')
if (!require(caret)) install.packages('caret')
if (!require(knitr)) install.packages('knitr')
if (!require(tensorflow)) install.packages('tensorflow')

library(keras)
library(tidyverse)
library(caret)
library(tensorflow)

# Installs tensorflow and keras
install_keras()
install_tensorflow()
# setting virtual memory size for CPU training
memory.limit(size=200000)
#setting maximum words limit for text tokenization
max_words <- 20000
#Preparing data
imdb_dir <- "~/Downloads/aclImdb"
data_dir <- file.path(imdb_dir, "data")
labels <- c()
texts <- c()
# downloads reviews' text in texts while corresponding
#sentiments are stored in labels, 0=negative, 1= positive
for (label_type in c("neg", "pos")) {
  label <- switch(label_type, neg = 0, pos = 1)
  dir_name <- file.path(data_dir, label_type)
  for (fname in list.files(dir_name, pattern = glob2rx("*.txt"),
                           full.names = TRUE)) {
    texts <- c(texts, readChar(fname, file.info(fname)$size))
    labels <- c(labels, label)
```

```
  }
}
```

Next, we use keras functions text_tokenizer() and fit_text_tokenizer() to turn each of `texts` into a vector of integers based on keras dictionary. We keep maximum number of words to 20000, i.e. only most common 20000 words will be kept while generating integer vectors out of `texts`. Reviews in `texts` are then converted into integer vectors `sequences`. By default, all punctuation is removed, turning the texts into space-separated sequences of words (words maybe include the ' character). These sequences are then split into lists of tokens. They will then be indexed or vectorized. 0 is a reserved index that won't be assigned to any word.

The object `tokenizer` has an attribute `word_index`, which is a named list mapping words to their rank/index(int).

```
# Now we tokenize the texts read from the files
# We restrict it to max 20000 words

tokenizer <- text_tokenizer(num_words = max_words)%>%
  fit_text_tokenizer(texts)
# tokenize sequences (vectors of integers) are generated
# and stored in a list sequences
sequences <- texts_to_sequences(tokenizer, texts)
word_index = tokenizer$word_index
cat("Found", length(word_index), "unique tokens.\n")
#Found 115970 unique tokens.
```

Using the word_index attribute of tokenizer, we display that the tokenizing process has found 115970 unique tokens in all reviews.

Please note that though we have found 115970 only most frequent 20000 words are used in the analysis. Here, number of tokens

```
max(sapply(sequences,max))
```

```
## [1] 19999
```

which is what we set in our `max_words` variable.

Here is a quick look on the `word_index`

```
head(word_index)
```

```
## $the
## [1] 1
##
## $and
## [1] 2
##
## $a
## [1] 3
##
## $of
## [1] 4
##
```

```
## $to
## [1] 5
##
## $is
## [1] 6
```

Now we transfer sequences to x and labels to y, then remove extra variable which will not be used anymore.

```
x<-sequences
y<-as.numeric(labels)
rm(texts,sequences,labels)
```

3.3.1 Dividing Data into Train and Test Set

In this project, we are dividing the data downloaded from Stanford database into three portions:

Initially we shall set 60% of data for training and 40% for testing using the Caret package. The train set is further divided into 80% for training and 20% for validation. In this way, 48% of the data will be used for training, 12% for validation and 40% for final testing. The reasons for doing that are

- We need to tune hyperparameters of our model, for which we require a validation data set. The test dataset will be left alone only for testing.
- We want to use more data for training then testing. If we have used the original division of Stanford IMDB data, which was 50% for training and 50% for testing, after dividing training set into validation, there would have been less data for training and more data for testing (including validation set).

Following is the code we have used to breakdown data into train and test set, whereas, we shall use keras functions to further divide the data from train set to validation set at the time of fitting the model.

Please note that since x is a list of integer vectors, we are using y to generate test index.

```
set.seed(100)
#since x is a list of integer vectors, we are using y
# to generate test_index
test_index<-createDataPartition(y,times=1,p=0.4,list=FALSE)
x_train<-x[-test_index]
y_train<-y[-test_index]
x_test<-x[test_index]
y_test<-y[test_index]
```

Rest of the data preparation depends on the type of model we shall be using, so we shall deal it separately in different model subsections.

4 Dense Layers DNNs

4.1 A Brief Introduction

Neural networks originated in the computer science field to answer questions that normal statistical approaches were not designed to answer at the time.

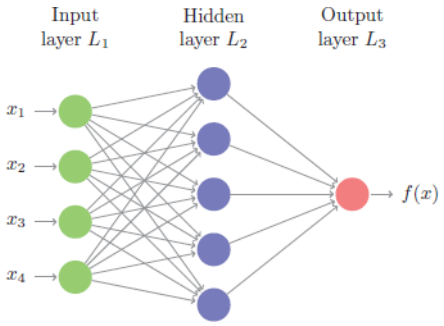


Figure 1: Representation of a simple feedforward neural network

At their most basic levels, neural networks have three layers: an input layer, a hidden layer, and an output layer. The input layer consists of all of the original input features. The majority of the learning takes place in the hidden layer, and the output layer outputs the final predictions.

Neurons are the basic units of a neural network. In an ANN, each neuron in a layer and is connected to each neuron in the next layer. When the inputs are transmitted between neurons, the weights are applied to the inputs along with the bias.

Let us understand the skeleton of a neuron.

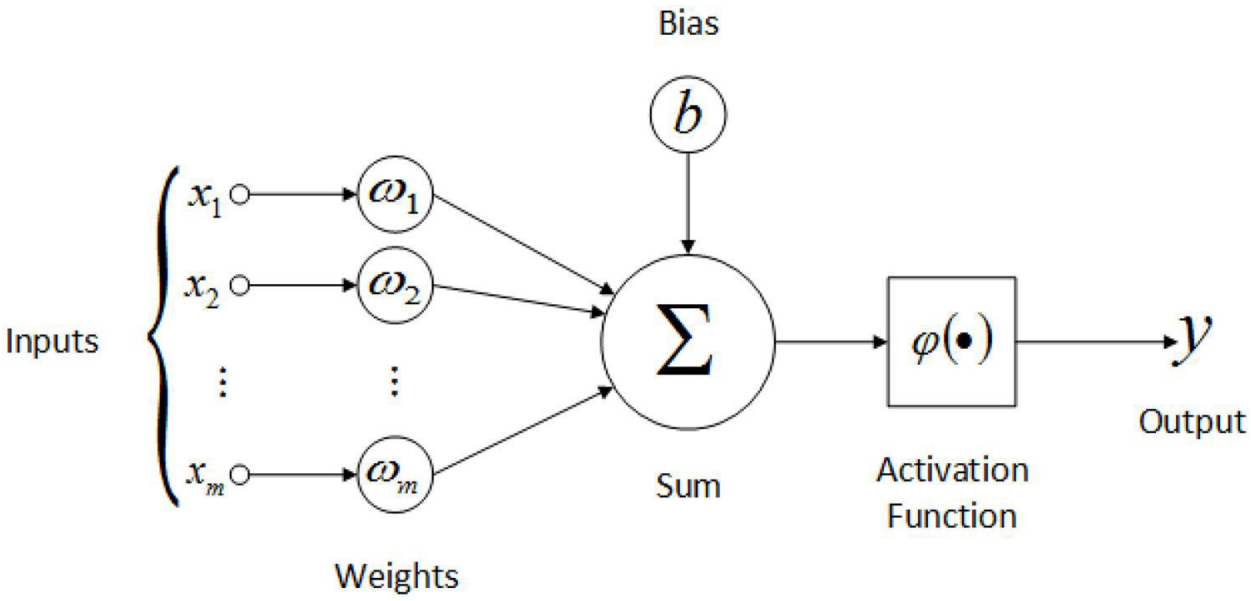


Figure 2: Skeleton of a Neuron

A neuron has following components.

- Inputs: Inputs are the set of values for which we need to predict the output value. They can be viewed as features or attributes in a dataset.
- Weights: weights are the real values that are associated with each feature which tells the importance of that feature in predicting the final value. (we will know more about in this article)
- Bias: Bias is used for shifting the activation function towards left or right, it can be referred to as a y-intercept in the line equation. (we will know more about this in this article)
- Summation Function: The work of the summation function is to bind the weights and inputs together and find their sum.

- Activation Function: It is used to introduce non-linearity in the model.

In R we can write this as:

```
output <- activation_function(dot(W, input)+b)
```

Where W is a 2D tensor of weights and b is a vector.

4.1.1 Layers and Nodes

The layers are made of nodes. A node is just a place where computation happens, we also called it a neuron. The layers and nodes are the building blocks of our DNN and they decide how complex the network will be. Layers are considered dense (fully connected) when all the nodes in each successive layer are connected. Consequently, the more layers and nodes you add the more opportunities for new features to be learned (commonly referred to as the model's capacity). Beyond the input layer, which is just our original predictor variables, there are two main types of layers to consider: hidden layers and an output layer.

4.1.2 Activation Function

As stated previously, each node is connected to all the nodes in the previous layer. Each connection gets a weight and then that node adds all the incoming inputs multiplied by its corresponding connection weight plus an extra bias parameter (w_0). The summed total of these inputs become an input to an activation function.

The activation function is simply a mathematical function that determines whether or not there is enough informative input at a node to fire a signal to the next layer. There are multiple activation functions to choose from but the most common ones include:

$$\texttt{Linear (identity):}\ f(x) = x$$

$$\texttt{Rectified linear unit (ReLU):}\ f(x) = \begin{cases} 0, & \text{for } x < 0. \\ x, & \text{for } x \geq 0. \end{cases}$$

$$\texttt{Sigmoid:}\ f(x) = \frac{1}{1+e^{-x}}$$

$$\texttt{Softmax:}\ f(x) = \frac{e^{x_i}}{\sum_j e^{x_j}}$$

Figure 3: Activation Functions

When using rectangular data, the most common approach is to use ReLU activation functions in the hidden layers. The ReLU activation function is simply taking the summed weighted inputs and transforming them to a 0 (not fire) or > 0 (fire) if there is enough signal. For the output layers we use the linear activation function for regression problems, the sigmoid activation function for binary classification problems, and softmax for multinomial classification problems.

4.1.3 Backpropogation

On the first run (or forward pass), the DNN will select a batch of observations, randomly assign weights across all the node connections, and predict the output. The engine of neural networks is how it assesses its own accuracy and automatically adjusts the weights across all the node connections to improve that accuracy. This process is called backpropagation. To perform backpropagation we need two things:

1. An Objective (loss) Function
2. An Optimizer

4.1.3.1 Objective (loss) Function First, you need to establish an objective (loss) function to measure performance. For regression problems this might be mean squared error (MSE) and for classification problems it is commonly binary and multi-categorical cross entropy. Following is the brief description of some loss functions used in keras for regression and classification models.

4.1.3.1.1 Regression Models

- MSE: Mean squared error is the average of the squared error. The squared component results in larger errors having larger penalties. This (along with RMSE) is the most common error metric to use.
- RMSE: Root mean squared error. This simply takes the square root of the MSE metric so that your error is in the same units as your response variable. If your response variable units are dollars, the units of MSE are dollars-squared, but the RMSE will be in dollars.
- MAE: Mean absolute error. Similar to MSE but rather than squaring, it just takes the mean absolute difference between the actual and predicted values.This results in less emphasis on larger errors than MSE.
- RMSLE: Root mean squared logarithmic error. Similar to RMSE but it performs a log() on the actual and predicted values prior to computing the difference. When your response variable has a wide range of values, large response values with large errors can dominate the MSE/RMSE metric. RMSLE minimizes this impact so that small response values with large errors can have just as meaningful of an impact as large response values with large errors.

4.1.3.1.2 Classification Models

- Crossentropy: crossentropy is usually the best choice when you're dealing with models that output probabilities. Crossentropy is a quantity from the field of Information Theory that measures the distance between probability distributions or, in this case, between the ground truth distribution and your predictions. this metric disproportionately punishes predictions where we predict a small probability for the true class.
  - Binary Crossentropy Use this cross-entropy loss when there are only two label classes (assumed to be 0 and 1).
  - Categorical Crossentropy Use this crossentropy loss function when there are two or more label classes. We expect labels to be provided in a one_hot representation.
  - Sparse Categorical Crossentropy Use this crossentropy loss function when there are two or more label classes. We expect labels to be provided as integers.

4.1.3.2 Optimizer Optimizers are algorithms or methods used to change the attributes of your neural network such as weights and learning rate in order to reduce the losses.

In this project, we have used RMSprop in all deep learning models, however, we have covered gradient descent, stochastic gradient descent and Adam in this report as well.

- Gradient Descent: Though keras doesn't provide any optimizer only with gradient descent, rest of all optimizers are some variations of gradient descent, so it is worthwhile to start with it.

Gradient descent is a first-order optimization algorithm which is dependent on the first order derivative of a loss function. It calculates that which way the weights should be altered so that the function can reach a minima. Through backpropagation, the loss is transferred from one layer to another and the model's parameters also known as weights are modified depending on the losses so that the loss can be minimized.

Advantages:

1. Easy computation.
2. Easy to implement.
3. Easy to understand.

Disadvantages:

1. May trap at local minima.
2. Weights are changed after calculating gradient on the whole dataset. So, if the dataset is too large than this may take years to converge to the minima.
3. Requires large memory to calculate gradient on the whole dataset.

- Stochastic Gradient Descent: It's a variant of Gradient Descent. It tries to update the model's parameters more frequently. In this, the model parameters are altered after computation of loss on each training example. So, if the dataset contains 1000 rows SGD will update the model parameters 1000 times in one cycle of dataset instead of one time as in Gradient Descent.

As the model parameters are frequently updated parameters have high variance and fluctuations in loss functions at different intensities.

Advantages:

1. Frequent updates of model parameters hence, converges in less time.
2. Requires less memory as no need to store values of loss functions.
3. May get new minima's.

Disadvantages:

1. High variance in model parameters.
2. May shoot even after achieving global minima.
3. To get the same convergence as gradient descent needs to slowly reduce the value of learning rate.

- RMSProp: RMSprop, or Root Mean Square Propogation has an interesting history. It was devised by the legendary Geoffrey Hinton, while suggesting a random idea during a course in a class.

Referring to the following figure, RMSProp tries to dampen the oscillations in w1 direction, as would be taken by a gradient descent algorithm, at the same time changing the learning rate automatically. RMSProp adjust the learning rate for each parameter (node).

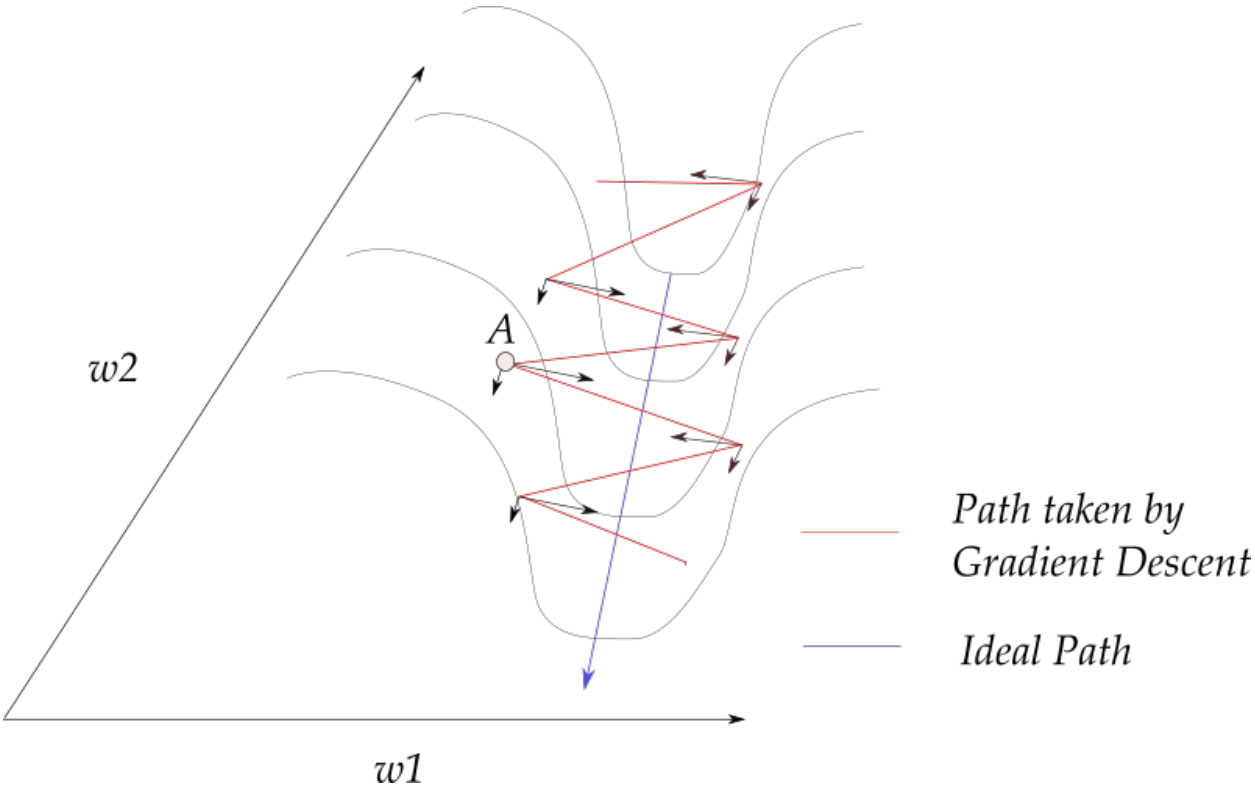


Figure 4: Grad Descent Vs Ideal Path

In RMS prop, each update is done according to the equations described below. This update is done separately for each parameter.

For each Parameter w^j (j subscript dropped for clarity)

Eq_1: $v_t = \rho v_{t-1} + (1-\rho) * g_t^2$

Eq_2: $\Delta w_t = -\frac{\eta}{\sqrt{v_t+\epsilon}} * g_t$

Eq_3: $w_{t+1} = w_t + \Delta w_t$

Here:

η : Initial Learning Rate

w_t : Exponential Average of squares of gradients

g_t : Gradient at time t along w^j

ϵ : A small positive number to make sure when v_t is zero or close to zero, Δw_t doesn't become too large

Equation 2 implies that the change in parameter w^J in the current iteration depends on the current gradient as well as a variable learning rate. The variable learning rate is equal to initial learning rate η divided by square root of exponential average of squares of gradient.

Now referring again to Fig. 4, $\sqrt{v_t+\epsilon}$ is relatively large number in w_1 direction as compared to w_2 direction. As a result Δw_t in w_1 is relatively small as compared to that in w_2 direction, as we desire to minimize oscillation of learning rate and get to the minima soon, the ideal path marked in the Fig. 4.

- Adam : Adaptive Moment Estimation (Adam) is another method that computes adaptive learning rates for each parameter. In addition to storing an exponentially decaying average of past squared

gradients v_t like Adadelta and RMSprop, Adam also keeps an exponentially decaying average of past gradients m_t, similar to momentum. Whereas momentum can be seen as a ball running down a slope, Adam behaves like a heavy ball with friction, which thus prefers flat minima in the error surface. We compute the decaying averages of past and past squared gradients m_t and v_t respectively as follows:

$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$

$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$

m_t and v_t are are estimates of the first moment (the mean) and the second moment (the uncentered variance) of the gradients respectively, hence the name of the method. As m_t and v_t are initialized as vectors of 0's, the authors of Adam observe that they are biased towards zero, especially during the initial time steps, and especially when the decay rates are small (i.e. β_1 and β_2 are close to 1).

These biases are counterated by computing bias-corrected first and second moment estimates:

$\hat{m}_t = \frac{m_t}{1-\beta_1^t}$

$\hat{v}_t = \frac{v_t}{1-\beta_2^t}$

These are then used to update parameters just as we have seen in RMSprop.

$w_t = w_{t-1} - \eta \frac{\hat{m}_t}{\sqrt{\hat{v}} + \epsilon}$

The default values of hyperparameters are:

$\beta_1 = 0.9$

$\beta_2 = 0.999$

$\epsilon = 10^{-8}$

4.2 Loss Vs Accuracy

All the models we shall work on using keras will give us two metrics at the end: accuracy and loss.

Loss can be seen as a distance between the true values of the problem and the values predicted by the model. Greater the loss is, more huge is the errors you made on the data.

Accuracy can be seen as the number of error you made on the data.

That means:

- a low accuracy and huge loss means you made huge errors on a lot of data
- a low accuracy but low loss means you made little errors on a lot of data
- a high accuracy with low loss means you made low errors on a few data (best case)
- a high accuracy but a huge loss, means you made huge errors on a few data.

4.3 Preparing Data for Dense Layer DNN Models

For Dense Layer model, we need to convert integer vectors representing review texts into a tensor of shape (samples, vector_sequence) where samples are no of rows of matrix and vector_sequence is the vector in sequences list converted into a one-hot encoding row of dimension=max_words. Only those columns will be 1 in a row (representing a review) whose corresponding word index is found in the row, all other columns will be 0.

First we define a function to convert list of integer vectors into a matrix (tensor).

```
vectorize_sequences <- function(sequences, dimension = max_words) {
  # 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
}
```

Now convert x_train and y_train list of integer vectors into a tensor or shape (samples, vector_sequence)

```
x_train <- vectorize_sequences(x_train)
x_test<-vectorize_sequences(x_test)
```

4.4 Simple Dense Layer Feedforward model

The first DNN model we are going to build is simple 3 dense layers' model, one input, one hidden and one output layer. We shall use "relu" as activation function in input and hidden layers, and "sigmoid" for output layer. For explanation about activation function refer to section Activation Function.

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

Here is the summary of the model.

```
summary(model)
```

```
## Model: "sequential"
## ________________________________________________________________________________
## Layer (type)                        Output Shape                    Param #
## ================================================================================
## dense_2 (Dense)                     (None, 16)                      320016
## ________________________________________________________________________________
## dense_1 (Dense)                     (None, 16)                      272
## ________________________________________________________________________________
## dense (Dense)                       (None, 1)                       17
## ================================================================================
## Total params: 320,305
## Trainable params: 320,305
## Non-trainable params: 0
## ________________________________________________________________________________
```

Next we shall compile the model using loss function as "binary_crossentropy" (refer to section Objective (loss) Function) and using RMSprop optimizer (refer to section Optimizer).

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

4.4.1 Model Training

After creating the model, we shall train it to train dataset we have prepared earlier. To do so, we feed our model to a `fit()` function, at the same time we shall split the train data into a 20% validation part, which we shall use to tune hyper parameters. Results are stored in a variable `history` so that we can access hyper parameters later.

```
history <- model %>% fit(
  x_train, y_train,
  epochs = 20,
  batch_size = 512,
  validation_split = 0.2
)
```

Following parameters passed to `fit()` function are worth mentioning.

- `batch_size`: DNNs take a batch of data to run through the mini-batch SGD process. Batch sizes can be between one and several hundred. Small values will be more computationally burdensome while large values provide less feedback signal. Values are typically provided as a power of two that fit nicely into the memory requirements of the GPU or CPU hardware like 32, 64, 128, 256, and so on.
- `epoch`: An epoch describes the number of times the algorithm sees the entire data set. So, each time the algorithm has seen all samples in the data set, an epoch has completed. The more complex the features and relationships in your data, the more epochs you'll require for your model to learn, adjust the weights, and minimize the loss function.
- `validation_split` : The model will hold out XX% of the data so that we can compute a more accurate estimate of an out-of-sample error rate.

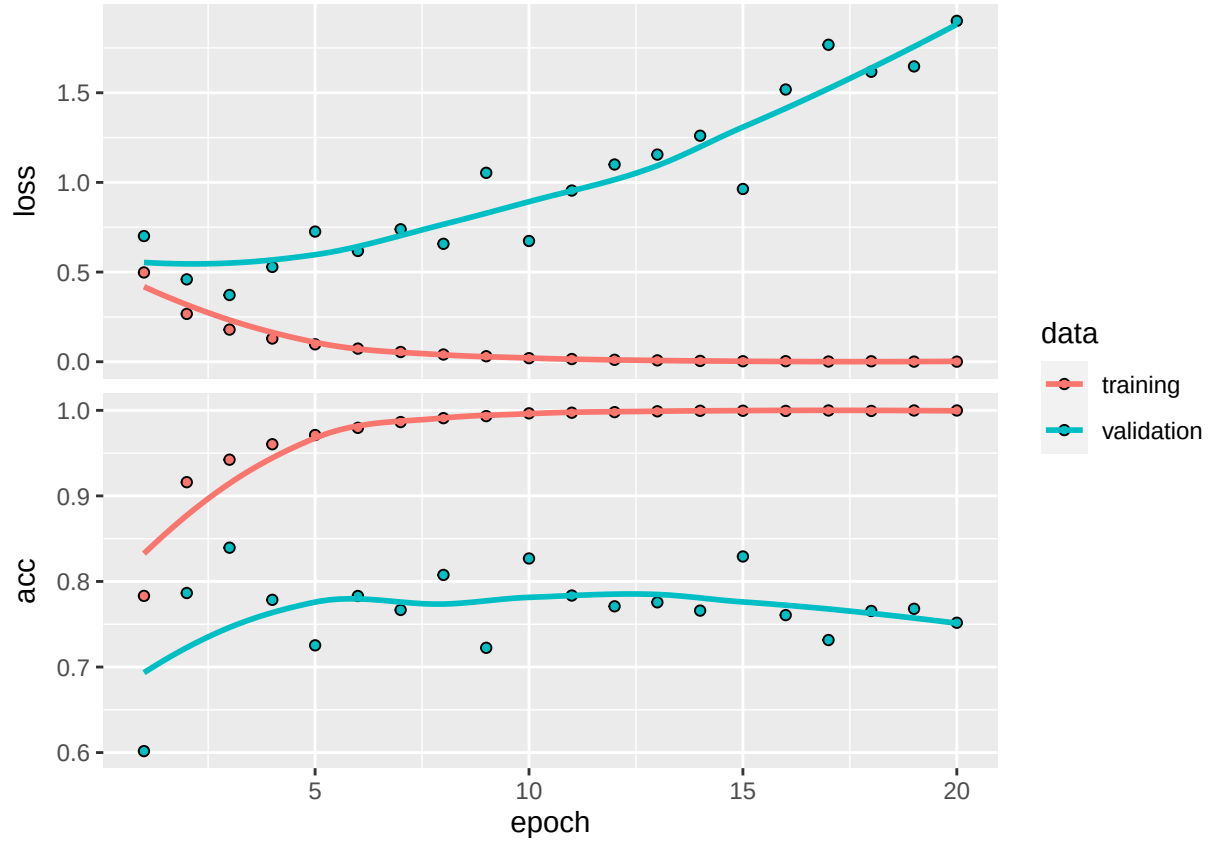
Call to `fit()` returns `hisotry` object. The `history` object includes parameters used to fit the model (history$params) as well as data for each of the metrics being monitored (history$metrics).

```
str(history)
```

```
## List of 2
##  $ params :List of 3
##   ..$ verbose: int 1
##   ..$ epochs : int 20
##   ..$ steps  : int 41
##  $ metrics:List of 4
##   ..$ loss    : num [1:20] 0.498 0.267 0.179 0.129 0.097 ...
##   ..$ acc     : num [1:20] 0.783 0.916 0.942 0.96 0.971 ...
##   ..$ val_loss: num [1:20] 0.701 0.459 0.372 0.529 0.726 ...
##   ..$ val_acc : num [1:20] 0.602 0.786 0.839 0.779 0.725 ...
##  - attr(*, "class")= chr "keras_training_history"
```

The history object has a plot() method that enables you to visualize training and validation metrics by epoch:

```
plot(history)
```



The epoch at which maximum validation accuracy is achieved is

```
epoch<-which.max(history$metrics$val_acc)
epoch
```

```
## [1] 3
```

While the maximum validation accuracy achieved is

```
max(history$metrics$val_acc)
```

```
## [1] 0.8394798
```

The `history` plot also shows us that after 3 epoch, the model starts overfitting.

Now, let's fit our model for the whole train dataset for epoch = 3

```
model %>% fit(x_train, y_train, epochs = epoch, batch_size = 512)
```

4.4.2 Results

We use keras evaluate function to evaluate our model on test data.

Finally, our simple dense layer model without any regularization gives us following accuracy and loss on the test data.

```
results <- model %>% evaluate(x_test, y_test)
```

```
results
```

```
##      loss       acc
## 0.8891655 0.8626754
```

In this way, this model achieved an accuracy of 86.26754.

We can print results of our first model as.

```
library(knitr)
model_results<-tibble(method="Dense Layer",accuracy=results[[2]],loss=results[[1]])
kable(model_results)
```

method	accuracy	loss
Dense Layer	0.8626754	0.8891655

4.5 Dense Layer Model with Dropout

Dropout is one of the most effective and most commonly used regularization techniques for neural networks, developed by Geoff Hinton and his students at the University of Toronto. Dropout, applied to a layer, consists of randomly dropping out (setting to zero) a number of output features of the layer during training.

The dropout rate is the fraction of the features that are zeroed out. It's usually set between 0.2 and 0.5. At test time, no units are dropped out; instead, the layer's output values are scaled down by a factor equal to the dropout rate, to balance for the fact that more units are active than at training time.

We can write a psuedo code for 0.5 dropout as following.

```
layer_output <-layer_output * sample(0:1, length(layer_output),replace = TRUE)
```

At test time, we scale down the output by the dropout rate. Here, we scale by 0.5 (because we previously dropped half the units):

```
layer_output <-layer_output * 0.5
```

In the first dense layer model, we saw that the model overfitting started with 7th layer. Let's add dropout to our dense layer and see how it improves the accuracy.

```
library(keras)
model <- keras_model_sequential() %>%
  layer_dense(units = 16, activation = "relu", input_shape = c(max_words )) %>%
  layer_dropout(rate=0.5)%>%
  layer_dense(units = 16, activation = "relu") %>%
  layer_dropout(rate=0.5)%>%
  layer_dense(units = 1, activation = "sigmoid")
```

Here is the summary of the model.

```
summary(model)
```

```
## Model: "sequential_1"
## _________________________________________________________________________________
## Layer (type)                        Output Shape                    Param #
## =================================================================================
## dense_5 (Dense)                     (None, 16)                      320016
## _________________________________________________________________________________
## dropout_1 (Dropout)                 (None, 16)                      0
## _________________________________________________________________________________
## dense_4 (Dense)                     (None, 16)                      272
## _________________________________________________________________________________
## dropout (Dropout)                   (None, 16)                      0
## _________________________________________________________________________________
## dense_3 (Dense)                     (None, 1)                       17
## =================================================================================
## Total params: 320,305
## Trainable params: 320,305
## Non-trainable params: 0
## _________________________________________________________________________________
```

Now, let's compile the model.

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

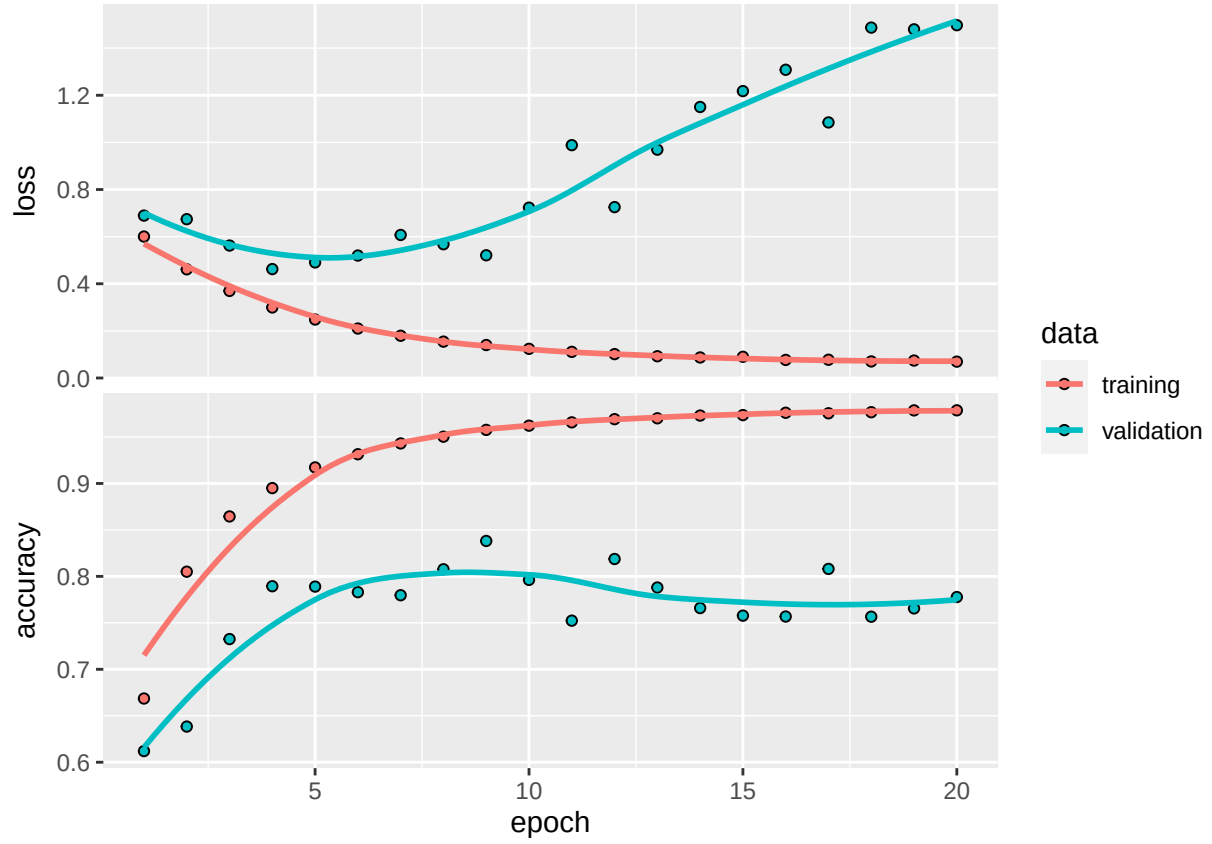
4.5.1 Model Training

Again we shall go and fit the model on training dataset first by dividing it into 80% training and 20% validation. Here is the code.

```
history <- model %>% fit(
  x_train, y_train,
  epochs = 20,
  batch_size = 512,
  validation_split = 0.2
)
```

To visualize training and validation metrics by epoch we shall plot the `history`:

```
plot(history)
```



The epoch at which maximum validation accuracy is achieved is

```
epoch<-which.max(history$metrics$val_acc)
epoch
```

```
## [1] 9
```

While the maximum validation accuracy achieved is

```
max(history$metrics$val_acc)
```

```
## [1] 0.8381211
```

Now, let's fit our model for the whole train dataset for epoch = 9

```
model %>% fit(x_train, y_train, epochs = epoch, batch_size = 512)
```

4.5.2 Results

We use keras evaluate function to evaluate our model on test data.

Finally, this dense layers with dropouts models gives us following accuracy and loss on the test data.

```
results <- model %>% evaluate(x_test, y_test)
```

```
results
```

```
##      loss  accuracy
## 0.5989639 0.8790402
```

In this way, this model achieved an accuracy of 87.90402.

Following is the comparison of different model results so far.

```
model_results2<-rbind(model_results,data.frame(method="Dense Layer with Dropout",accuracy=results[[2]],l
kable(model_results2)
```

method	accuracy	loss
Dense Layer	0.8626754	0.8891655
Dense Layer with Dropout	0.8790402	0.5989639

4.6 Dense Layer Model with Weight Regularization

A common way to mitigate overfitting is to put constraints on the complexity of a network by forcing its weights to take only small values, which makes the distribution of weight values more regular. This is called weight regularization, and it's done by adding to the loss function of the network a cost associated with having large weights.

This cost comes in two flavors:

- L1 regularization— The cost added is proportional to the absolute value of the weight coefficients (the L1 norm of the weights).
- L2 regularization— The cost added is proportional to the square of the value of the weight coefficients (the L2 norm of the weights). L2 regularization is also called weight decay in the context of neural networks.

In Keras, weight regularization is added by passing weight regularizer instances to layers as keyword arguments. The weight regularization is called kernal regularization.

Let's add kernal regularization to our dense layer model instead of drop out.

```
################################################################
# Dense Layer Feedforward model with kernal L2 regularization
################################################################
k_clear_session()
model <-keras_model_sequential() %>%
  layer_dense(units = 16, kernel_regularizer = regularizer_l2(0.01),
  activation = "relu", input_shape = c(max_words)) %>%
```

```
  layer_dense(units = 16, kernel_regularizer = regularizer_l2(0.01),
  activation = "relu") %>%
  layer_dense(units = 1, activation = "sigmoid")
```

Here is the summary of the model.

```
summary(model)
```

```
## Model: "sequential"
## _________________________________________________________________________
## Layer (type)                    Output Shape                  Param #
## =========================================================================
## dense_2 (Dense)                 (None, 16)                    320016
## _________________________________________________________________________
## dense_1 (Dense)                 (None, 16)                    272
## _________________________________________________________________________
## dense (Dense)                   (None, 1)                     17
## =========================================================================
## Total params: 320,305
## Trainable params: 320,305
## Non-trainable params: 0
## _________________________________________________________________________
```

We compile the model with same parameters.

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

4.6.1 Model Training

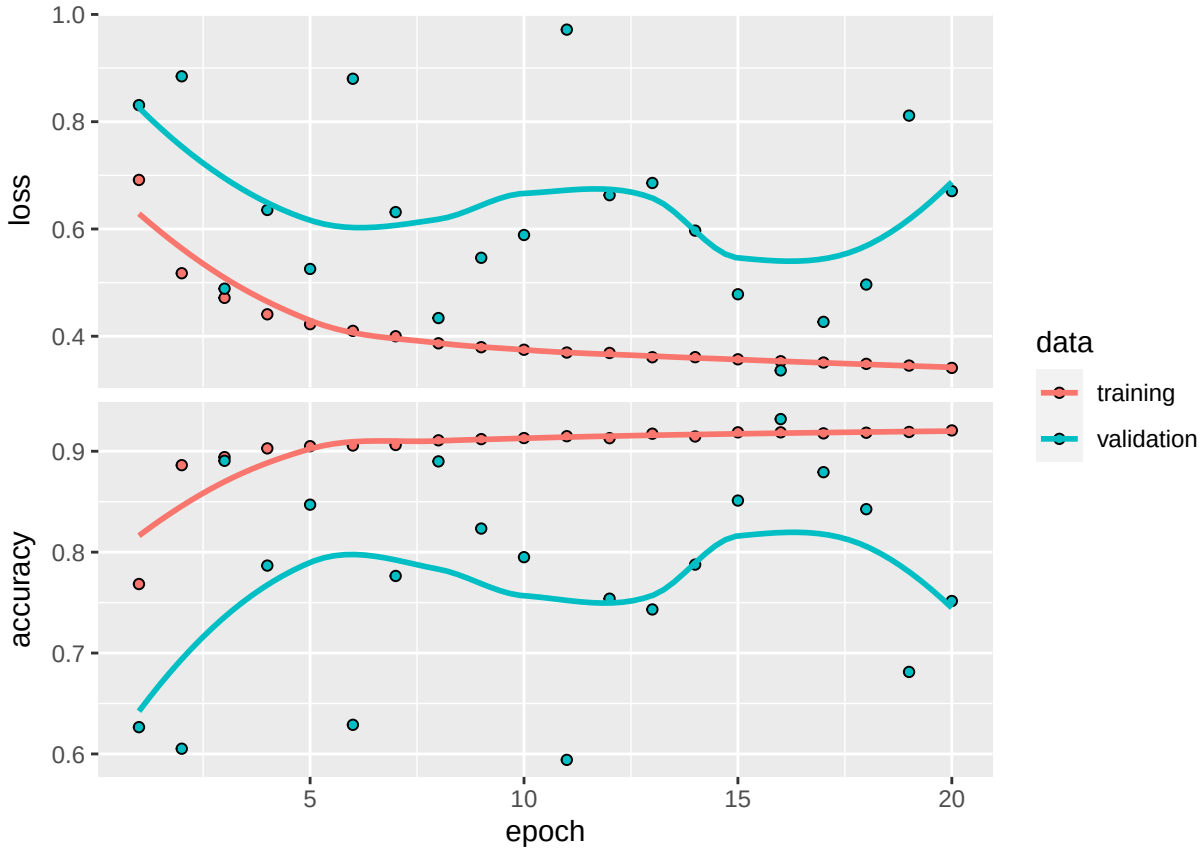
We shall go and fit the model on training dataset first by dividing it into 80% training and 20% validation. Here is the code.

```
load("./rdas/FFL2RegHistory.rda")
```

```
history <- model %>% fit(
  x_train, y_train,
  epochs = 20,
  batch_size = 512,
  validation_split = 0.2
)
```

To visualize training and validation metrics by epoch we shall plot the `history`:

```
plot(history)
```



The epoch at which maximum validation accuracy is achieved is

```
epoch<-which.max(history$metrics$val_acc)
epoch
```

```
## [1] 16
```

While the maximum validation accuracy achieved is

```
max(history$metrics$val_acc)
```

```
## [1] 0.9318711
```

Now, let's fit our model for the whole train dataset for epoch = 16

```
model %>% fit(x_train, y_train, epochs = epoch, batch_size = 512)
```

4.6.2 Results

We use keras evaluate function to evaluate our model on test data.

Finally, this dense layers with L2 regularization on weights models gives us following accuracy and loss on the test data.

```
results <- model %>% evaluate(x_test, y_test)
```

```
results
```

```
##      loss  accuracy
## 0.3880340 0.8787491
```

In this way, this model achieved an accuracy of 87.87491.

Following is the comparison of different model results so far.

```
model_results3<-rbind(model_results2,data.frame(method="Dense Layer with L2 Weight Regularization",accu
kable(model_results3)
```

method	accuracy	loss
Dense Layer	0.8626754	0.8891655
Dense Layer with Dropout	0.8790402	0.5989639
Dense Layer with L2 Weight Regularization	0.8787491	0.3880340

5 Advanced Deep Learning Models

5.1 Preparing Data for Rest of the Models

In the following sections, we shall be using advanced DNN models like 1D Convolution Neural Networks, LSTM or combination of both. For dense layer models, we use one hot encoding to convert 42927 reviews, each into a vector of length 20000. Another way to prepare text data is to pad your lists so that they all have the same length, turn them into an integer tensor of shape `(samples, word_indices)`, and then use as the first layer in your network a layer capable of handling such integer tensors (embedding layer).

The movie review database we downloaded has 42927 reviews. The average length of these revives is.

```
mean(unlist(lapply(x,length),use.names = FALSE))
```

```
## [1] 228.8617
```

While the maximum length of any review in this list is.

```
max(unlist(lapply(x,length),use.names = FALSE))
```

```
## [1] 2323
```

Following is the code we use to prepare the data.

```
####################################################
# Preparing data for rest of the models
####################################################
# For rest of the models, we don't use one-hot encoding
# rather we use list of integer vectors in x_train and x_test
```

```
# each representing a review and pad them with trailing 0's
# if they are short of max_len, if the no of words in review
# are more than max_len, we truncate rest of the words
x_train<-x[-test_index]
y_train<-y[-test_index]
x_test<-x[test_index]
y_test<-y[test_index]
max_features <- 20000  # Number of words to consider as features
maxlen <- 500          # Cut texts after this number of words
# Pad sequences
x_train<-pad_sequences(x_train, maxlen = maxlen)
x_test<-pad_sequences(x_test, maxlen = maxlen)
```

Since we are running these models on CPU, where time of execution is not that fast, we decided to limit our reviews length to maximum 500 words. That means any review longer that 500 words would be truncated and shorter will be padded with zeros at the end.

In this way, if we select maximum length of 500 words per review, % of reviews truncated will be.

```
data.frame(nwords=unlist(lapply(x,length),use.names = FALSE))%>%
  filter(nwords>500)%>%summarise(n=n()/length(x)*100)
```

```
##          n
## 1 7.531391
```

5.2 Word Embedding Vs One-hot Encoding

Another popular and powerful way to associate a vector with a word is the use of dense word vectors, also called word embeddings. Whereas the vectors obtained through one-hot encoding are binary, sparse (mostly made of zeros), and very high dimensional (same dimensionality as the number of words in the vocabulary), word embeddings are low dimensional floating point vectors (that is, dense vectors, as opposed to sparse vectors). Unlike the word vectors obtained via one-hot encoding, word embeddings are learned from data. It's common to see word embeddings that are 256-dimensional, 512-dimensional, or 1,024-dimensional, when dealing with very large vocabularies. On the other hand, one-hot encoding words generally leads to vectors that are 20,000-dimensional or greater (capturing a vocabulary of 20,000 tokens, in this case). So, word embeddings pack more information into far fewer dimensions.

An embedding layer takes as input a 2D tensor of integers, of shape `(samples,sequence_length)`, where each entry is a sequence of integers.This layer returns a 3D floating-point tensor of shape `(samples,sequence_length, embedding_dimensionality)`. Such a 3D tensor can then be processed by an RNN layer or a 1D convolution layer.

When you instantiate an embedding layer, its weights (its internal dictionary of token vectors) are initially random, just as with any other layer. During training, these word vectors are gradually adjusted via back-propagation, structuring the space into something the downstream model can exploit. Once fully trained, the embedding space will show a lot of structure—a kind of structure specialized for the specific problem for which you're training your model.

5.3 Bidirectional LSTM Model

5.3.1 Recurrent Neural Networks

A major characteristic of all neural networks you've seen so far, such as densely connected networks and convnets, is that they have no memory. Each input shown to them is processed independently, with no

state kept in between inputs. Humans don't start their thinking from scratch every second. As you read this report, you understand each word based on your understanding of previous words. You don't throw everything away and start thinking from scratch again. Your thoughts have persistence.

As an example, consider the following statements:

Mark was born in Quebec and grew up their until the age of 15. He can speak _____ fluently.

A human can easily fill in the blank above with the word "French", however, a densely connected numeral network or a CNN can't do that.

Recurrent neural networks address this issue. They are networks with loops in them, allowing information to persist.

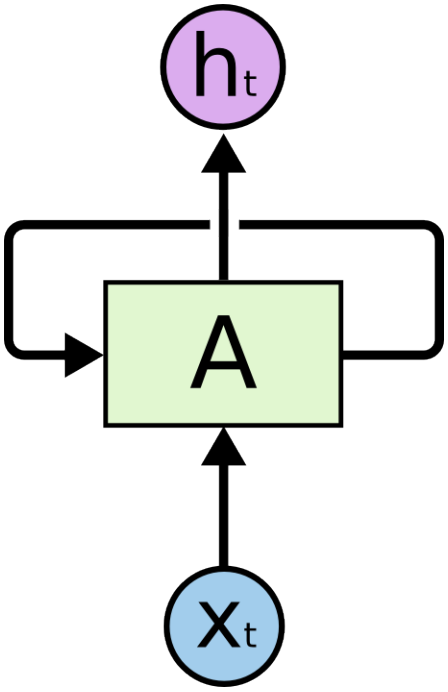


Figure 5: RNN have loops

A recurrent neural network can be thought of as multiple copies of the same network, each passing a message to a successor. Consider what happens if we unroll the loop:

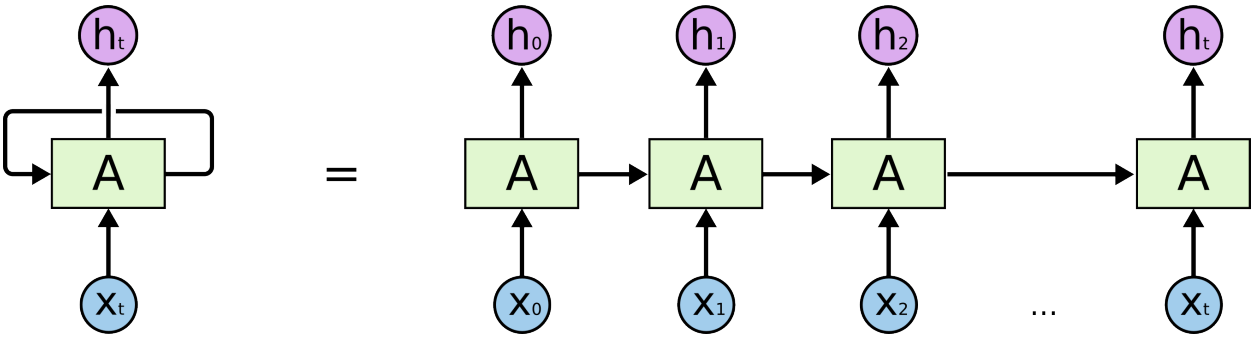


Figure 6: An Unrolled Recurrent Neural Network

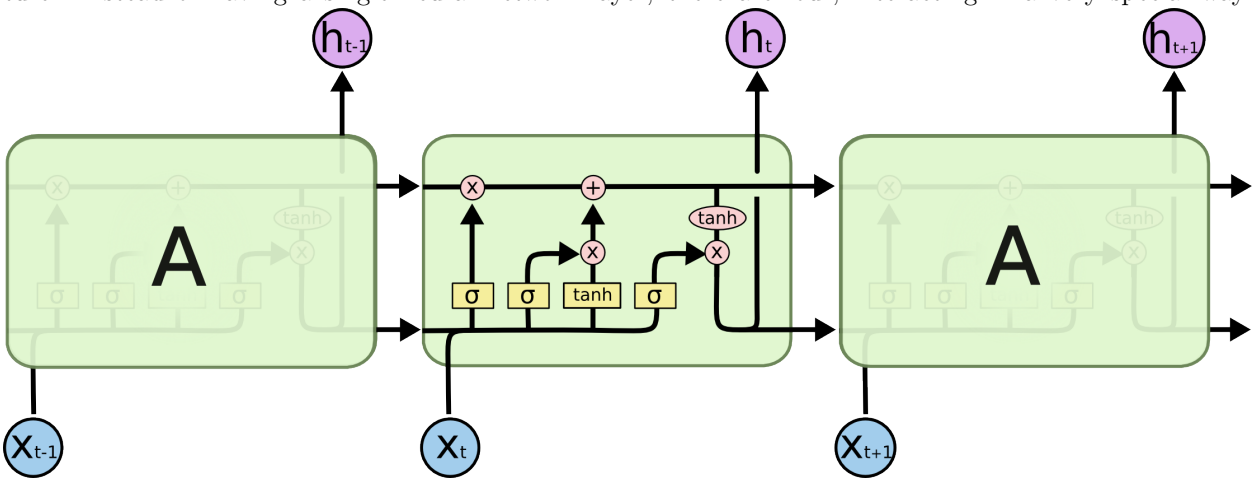
This chain-like nature reveals that recurrent neural networks are intimately related to temporal data and sequences. That makes RNNs specially useful for natural language processing.

One major issue with layer_simple_rnn is that although it should theoretically be able to retain at time t information about inputs seen many timesteps before, in practice, such long-term dependencies are impossible to learn. This is due to the vanishing gradient problem, an effect that is similar to what is observed with non-recurrent networks (feedforward networks) that are many layers deep: as you keep adding layers to a network, the network eventually becomes untrainable. The theoretical reasons for this effect were studied by Hochreiter, Schmidhuber, and Bengio in the early 1990s. The LSTM and GRU layers are designed to solve this problem.

5.3.2 Long Short-Term Memory (LSTM)

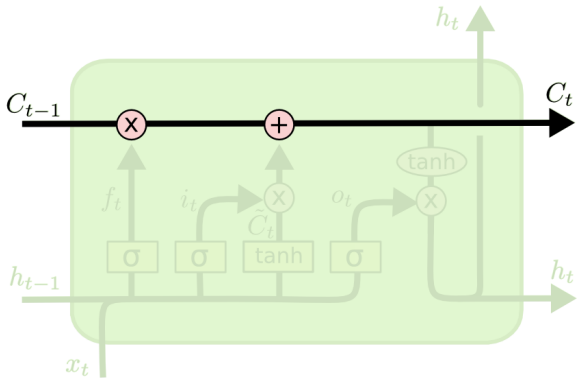
Long Short Term Memory networks – usually just called "LSTMs" – are a special kind of RNN, capable of learning long-term dependencies. They were introduced by Hochreiter & Schmidhuber (1997), and were refined and popularized by many people in following work. They work tremendously well on a large variety of problems, and are now widely used.

LSTMs also have a chain like structure like RNNs, but the repeating module has a different structure. Instead of having a single neural network layer, there are four, interacting in a very special way.



The key to LSTMs is the cell state, the horizontal line running through the top of the diagram.

The cell state is kind of like a conveyor belt. It runs straight down the entire chain, with only some minor linear interactions. It's very easy for information to just flow along it unchanged.



The LSTM does have the ability to remove or add information to the cell state, carefully regulated by structures called gates.

Gates are a way to optionally let information through. They are composed out of a sigmoid neural net layer and a point wise multiplication operation.

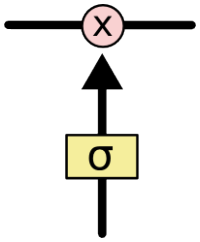
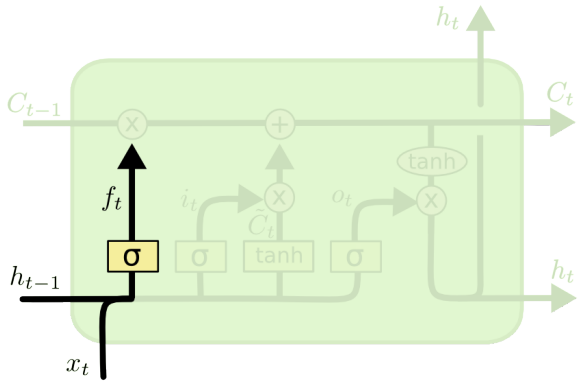


Figure 7: Gate

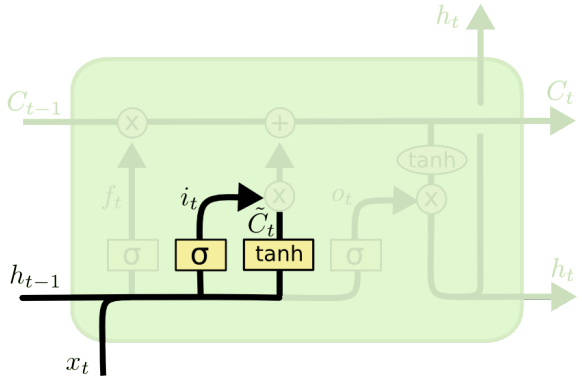
The first step in our LSTM is to decide what information we're going to throw away from the cell state. This decision is made by a sigmoid layer called the "forget gate layer." It looks at h_{t-1} and x_t, and outputs a number between 0 and 1 for each number in the cell state C_{t-1}. A 1 represents "completely keep this" while a 0 represents "completely get rid of this."



$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$$

Figure 8: f_t Forget Gate Layer

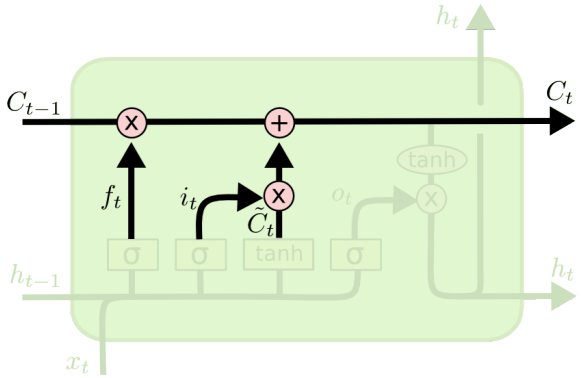
The next step is to decide what new information we're going to store in the cell state. This has two parts. First, a sigmoid layer called the "input gate layer" decides which values we'll update. Next, a tanh layer creates a vector of new candidate values, $\tilde{C}_t$, that could be added to the state. In the next step, we'll combine these two to create an update to the state.



$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \; + \; b_i\right)$$
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \; + \; b_C)$$

Figure 9: $\tilde{C}_t$ and i_t

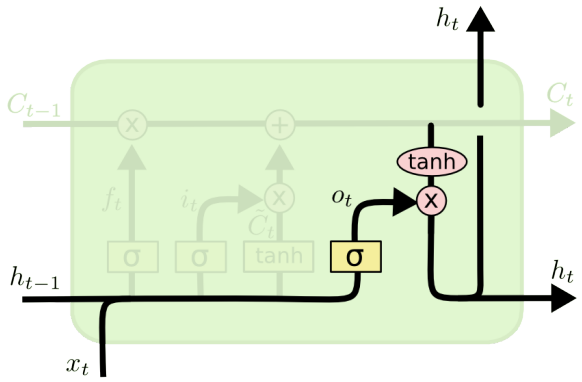
It is now time to update the old cell state, C_{t-1}, into the new cell state C_t. The previous steps already decided what to do, we just need to actually do it. We multiply the old state by f_t, forgetting the things we decided to forget earlier. Then we add $i_t * \hat{C}_t$. This is the new candidate values, scaled by how much we decided to update each state value.



$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

Figure 10: New Cell State

Finally, the output will be based on our cell state, but will be a filtered version. First, we run a `sigmoid` layer which decides what parts of the cell state we're going to output. Then, we put the cell state through `tanh` (to push the values to be between -1 and 1) and multiply it by the output of the sigmoid gate, so that we only output the parts we decided to.



$$o_t = \sigma\left(W_o\left[h_{t-1}, x_t\right] + b_o\right)$$
$$h_t = o_t * \tanh\left(C_t\right)$$

Figure 11: LSTM Cell Output

5.3.3 Bidirectional RNN

A bidirectional RNN is a common RNN variant that can offer greater performance than a regular RNN on certain tasks. It's frequently used in natural language processing.

RNNs are notably order dependent, or time dependent: they process the timesteps of their input sequences in order, and shuffling or reversing the timesteps can completely change the representations the RNN extracts from the sequence. This is precisely the reason they perform well on problems where order is meaningful, such as the temperature forecasting problem. A bidirectional RNN exploits the order sensitivity of RNNs: it consists of using two regular RNNs, such as layer_gru and layer_lstm , each of which processes the input sequence in one direction (chronologically and antichronologically), and then merging their representations. By processing a sequence both ways, a bidirectional RNN can catch patterns that may be overlooked by a unidirectional RNN.

5.3.4 Stacked LSTMs

Another architecture feature we have used in our model is stacking of two LSTM layers. Stacking LSTM hidden layers makes the model deeper, more accurately earning the description as a deep learning technique.

Given that LSTMs operate on sequence data, it means that the addition of layers adds levels of abstraction of input observations over time. In effect, chunking observations over time or representing the problem at different time scales.

Stacked LSTMs or Deep LSTMs were introduced by Graves, et al. in their application of LSTMs to speech recognition, beating a benchmark on a challenging standard problem.

Each LSTMs memory cell requires a 3D input. When an LSTM processes one input sequence of time steps, each memory cell will output a single value for the whole sequence as a 2D array.

To stack LSTM layers, we need to change the configuration of the prior LSTM layer to output a 3D array as input for the subsequent layer.

We can do this by setting the `return_sequences` argument on the layer to True (defaults to False). This will return one output for each input time step and provide a 3D array.

By default, the `return_sequences` is set to `False` in Keras RNN layers, and this means the RNN layer will only return the last hidden state output h_t. The last hidden state output captures an abstract representation of the input sequence. In some case, it is all we need, such as a classification or regression model where the RNN is followed by the Dense layer(s) to generate logic for news topic classification or score for sentiment analysis, or in a generative model to produce the softmax probabilities for the next possible char.

In the case when LSTM layer is followed by another LSTM layer, setting `return_squence` to TRUE is necessary so that h_t of each LSTM cell is forwarded to subsequent LSTM layer.

5.3.5 Building the Model

Following is the code to apply an embedding layer followed by bidirectional LSTM layer to IMDB data.

```
k_clear_session() # clears keras session

model <- keras_model_sequential() %>%
  layer_embedding(input_dim = max_features, output_dim = 128) %>%
  layer_dropout(0.9)%>%
  bidirectional(
    layer_lstm(units = 64, return_sequences=TRUE,recurrent_activation = "sigmoid")
  ) %>%
```

```
  bidirectional(
    layer_lstm(units = 64,recurrent_activation = "sigmoid")
  ) %>%
  layer_dense(units = 1, activation = "sigmoid")
```

Here is the summary of the model.

```
summary(model)
```

```
## Model: "sequential"
## _________________________________________________________________________________
## Layer (type)                         Output Shape                     Param #
## =================================================================================
## embedding (Embedding)                (None, None, 32)                 640000
## _________________________________________________________________________________
## dropout (Dropout)                    (None, None, 32)                 0
## _________________________________________________________________________________
## bidirectional_1 (Bidirectional)      (None, None, 128)                49664
## _________________________________________________________________________________
## bidirectional (Bidirectional)        (None, 128)                      98816
## _________________________________________________________________________________
## dense (Dense)                        (None, 1)                        129
## =================================================================================
## Total params: 788,609
## Trainable params: 788,609
## Non-trainable params: 0
## _________________________________________________________________________________
```

Please note that originally this model was run on CPU, while the above model was run on nVidia GPU with `cuDNN`. The model that ran on CPU is also included in the code file and achieved similar but a little less accuracy.

Here to regularize the model we are using a dropout layer before LSTM. It has long been known that applying dropout before a recurrent layer hinders learning rather than helping with regularization. In 2015, Yarin Gal, as part of his PhD thesis on Bayesian deep learning, determined the proper way to use dropout with a recurrent network: the same dropout mask (the same pattern of dropped units) should be applied at every timestep, instead of a dropout mask that varies randomly from timestep to timestep. In keras, the `layer_lstm` function has a parameter `recurrent_dropout` to achieve this. The problem is that if we use that parameter, `cuDNN` won't support that model and we'll have to run it on `CPU` or `GPU kernel` (not `cuDNN kernel`) which is painfully slow and takes hours. A compromise is that we have used a very light dropout before LSTM layer.

Another important note to mention here is the use of `sigmoid` as `recurrent_activation` instead of default `hard_sigmoid`. This is necessary requirement to train the model using cuDNN.

Next we shall compile the model using loss function as "binary_crossentropy" (refer to section Objective (loss) Function) and using RMSprop optimizer (refer to section Optimizer).

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

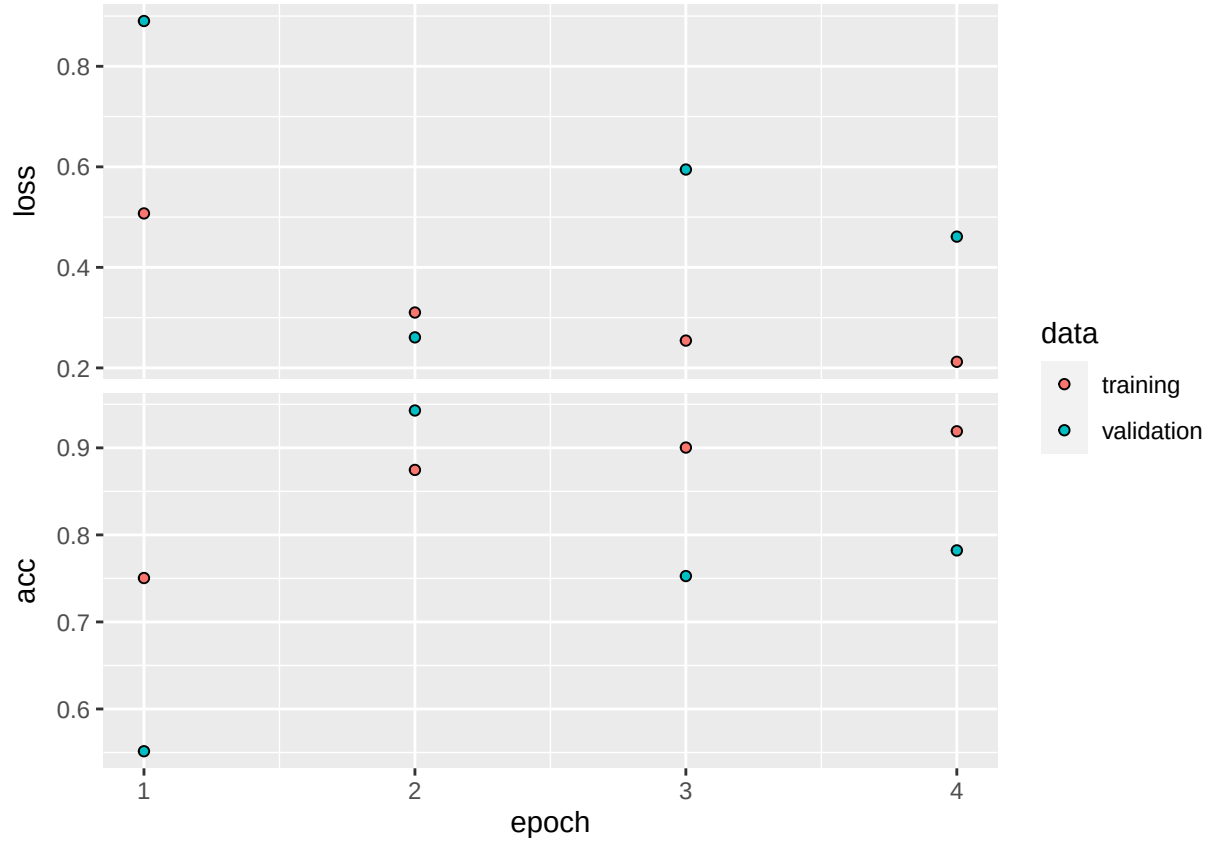
5.3.6 Model Training

We shall fit the model on training dataset first by dividing it into 80% training and 20% validation. Here is the code.

```
history <- model %>% fit(
  x_train, y_train,
  epochs = 15,
  batch_size = 128,
  validation_split = 0.2
)
```

To visualize training and validation metrics by epoch we shall plot the `history`:

```
plot(history)
```



The epoch at which maximum validation accuracy is achieved is

```
epoch<-which.max(history$metrics$val_acc)
epoch
```

```
## [1] 2
```

While the maximum validation accuracy achieved is

```
max(history$metrics$val_acc)
```

```
## [1] 0.9429348
```

Now, let's fit our model for the whole train dataset for epoch = 2

```
model %>% fit(x_train, y_train, epochs = epoch, batch_size = 128)
```

5.3.7 Results

We use keras evaluate function to evaluate our model on test data.

Finally, this model gives us following accuracy and loss on the test data.

```
results <- model %>% evaluate(x_test, y_test)
```

```
results
```

```
##      loss       acc
## 0.2761915 0.8995982
```

In this way, this model achieved an accuracy of 89.95982.

Following is the comparison of different model results so far.

```
model_results4<-rbind(model_results3,data.frame(method="Bidirectional LSTM", accuracy=results[[2]],loss=
kable(model_results4)
```

method	accuracy	loss
Dense Layer	0.8626754	0.8891655
Dense Layer with Dropout	0.8790402	0.5989639
Dense Layer with L2 Weight Regularization	0.8787491	0.3880340
Bidirectional LSTM	0.8995982	0.2761915

5.4 1D Convnet Model

Convolutional neural networks (CNNs) perform particularly well on computer vision problems, due to their ability to operate convolutionally, extracting features from local input patches and allowing for representation modularity and data efficiency. The same properties that make CNNs excel at computer vision also make them highly relevant to sequence processing. Time can be treated as a spatial dimension, like the height or width of a 2D image.

Such 1D CNNs can be competitive with RNNs on certain sequence-processing problems, usually at a considerably cheaper computational cost. Recently, 1D CNNs, typically used with dilated kernels, have been used with great success for audio generation and machine translation. In addition to these specific successes, it has long been known that small 1D CNNs can offer a fast alternative to RNNs for simple tasks such as text classification and timeseries forecasting.

The 2D convolutions extract 2D patches from image tensors and applying an identical transformation to

every patch. In the same way, you can use 1D convolutions, extracting local 1D patches (sub-sequences) from sequences.

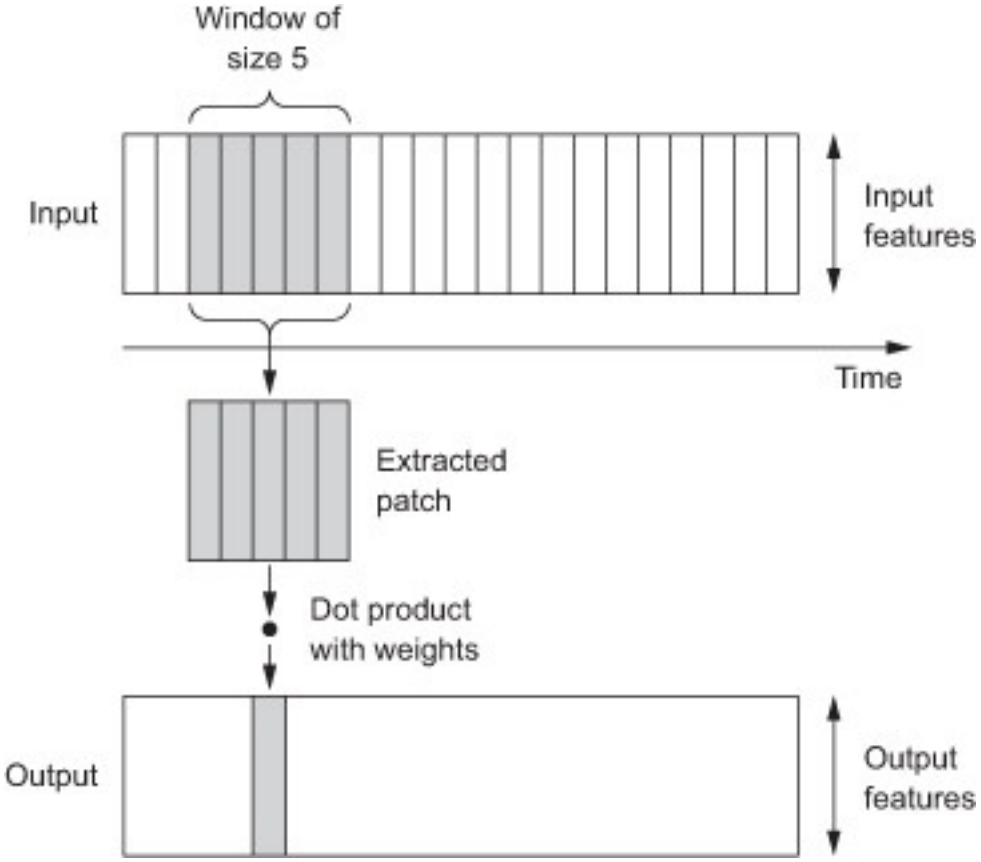


Figure 12: How 1D convolution works: each output timestep is obtained from a temporal patch in the input sequence

Such 1D convolution layers can recognize local patterns in a sequence. Because the same input transformation is performed on every patch, a pattern learned at a certain position in a sentence can later be recognized at a different position, making 1D CNNs translation invariant (for temporal translations). For instance, a 1D CNN processing sequences of characters using convolution windows of size 5 should be able to learn words or word fragments of length 5 or less, and it should be able to recognize these words in any context in an input sequence. A character-level 1D CNN is thus able to learn about word morphology.

The 2D pooling operation has a 1D equivalent: extracting 1D patches (sub-sequences) from an input and outputting the maximum value (max pooling) or average value (average pooling). Just as with 2D CNNs, this is used for reducing the length of 1D inputs (sub-sampling).

5.4.1 Max Pooling

Max pooling layer is used to downsample results of previous 1D CNN layer. The block or pool size is determined by the parameter `pool_size`. For each block, or "pool", the operation simply involves computing the max value. Doing so for each pool, we get a nicely downsampled outcome, greatly benefiting the spatial hierarchy we need.

5.4.2 Global Max Pooling

Another type of pooling layer is the Global Max Pooling layer. Here, we set the pool size equal to the input size, so that the max of the entire input is computed as the output value (Dernoncourt, 2017).

Global pooling layers can be used in a variety of cases. Primarily, it can be used to reduce the dimensionality of the feature maps output by some convolutional layer, to replace Flattening and sometimes even Dense layers in your classifier (Christlein et al., 2019). In our case of 1D CNN, it is used before the output layer to replace flattening.

5.4.3 Building the Model

We build our model using the following code.

```
k_clear_session() # clears keras session

model <- keras_model_sequential() %>%
  layer_embedding(input_dim = max_features, output_dim = 128,
                  input_length = maxlen) %>%
  layer_dropout(0.2)%>%
  layer_conv_1d(filters = 250, kernel_size = 3,
                activation = "relu") %>%
  layer_max_pooling_1d(pool_size = 5) %>%
  layer_conv_1d(filters = 250, kernel_size = 3,
                activation = "relu") %>%
  layer_global_max_pooling_1d() %>%
  layer_dense(units = 1,activation = "sigmoid")
```

Here is the summary of the model.

```
summary(model)
```

```
## Model: "sequential"
## ________________________________________________________________________________
## Layer (type)                        Output Shape                    Param #
## ================================================================================
## embedding (Embedding)               (None, 500, 128)                2560000
## ________________________________________________________________________________
## dropout (Dropout)                   (None, 500, 128)                0
## ________________________________________________________________________________
## conv1d_1 (Conv1D)                   (None, 498, 250)                96250
## ________________________________________________________________________________
## max_pooling1d (MaxPooling1D)        (None, 99, 250)                 0
## ________________________________________________________________________________
## conv1d (Conv1D)                     (None, 97, 250)                 187750
## ________________________________________________________________________________
## global_max_pooling1d (GlobalMaxPool (None, 250)                     0
## ________________________________________________________________________________
## dense (Dense)                       (None, 1)                       251
## ================================================================================
## Total params: 2,844,251
## Trainable params: 2,844,251
## Non-trainable params: 0
## ________________________________________________________________________________
```

Next we shall compile the model using loss function as "binary_crossentropy" (refer to section Objective (loss) Function) and using RMSprop optimizer (refer to section Optimizer).

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

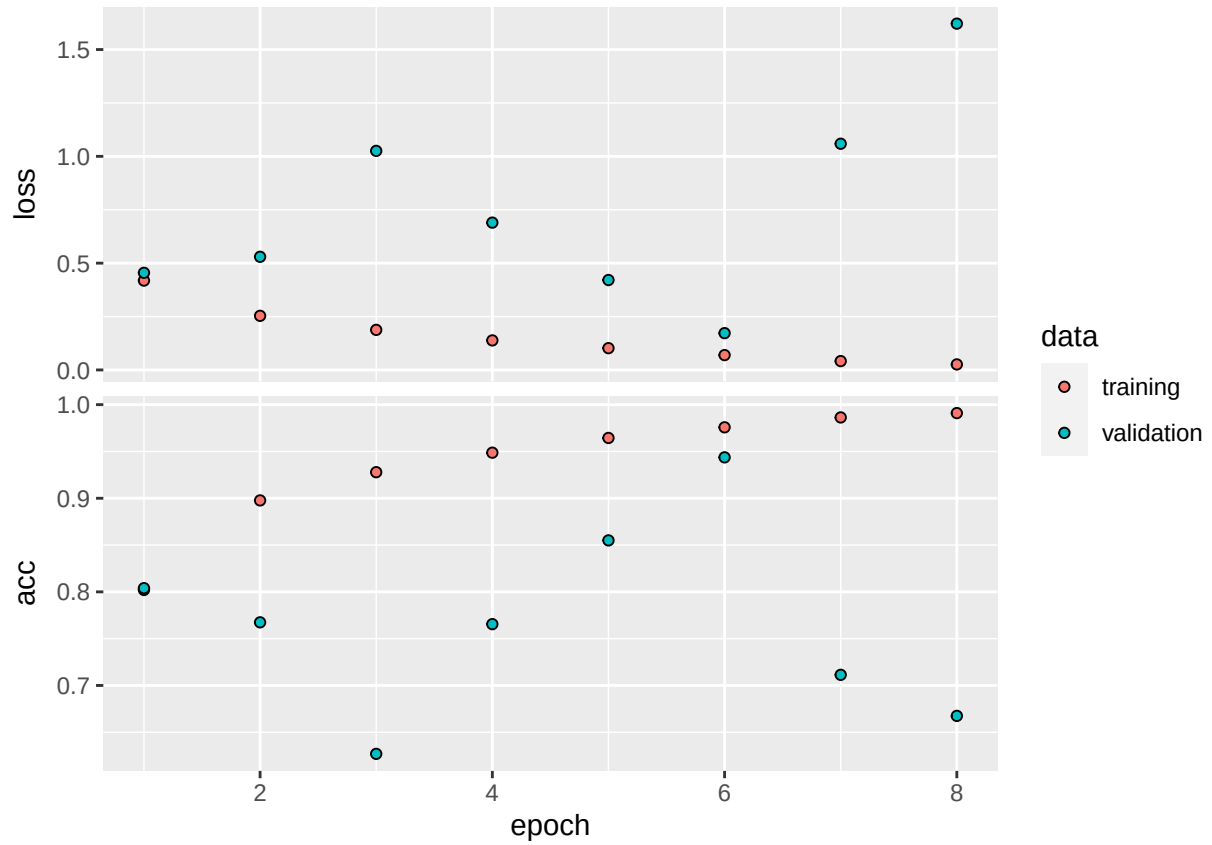
5.4.4 Model Training

We shall fit the model on training dataset first by dividing it into 80% training and 20% validation. Here is the code.

```
history <- model %>% fit(
  x_train, y_train,
  epochs = 8,
  batch_size = 32,
  validation_split = 0.2
)
```

To visualize training and validation metrics by epoch we shall plot the `history`:

```
plot(history)
```



The epoch at which maximum validation accuracy is achieved is

```
epoch<-which.max(history$metrics$val_acc)
epoch
```

```
## [1] 9
```

While the maximum validation accuracy achieved is

```
max(history$metrics$val_acc)
```

```
## [1] 0.8381211
```

Now, let's fit our model for the whole train dataset for epoch = 9

```
model %>% fit(x_train, y_train, epochs = epoch, batch_size = 32)
```

5.4.5 Results

We use keras evaluate function to evaluate our model on test data.

Finally, this model gives us following accuracy and loss on the test data.

```
results <- model %>% evaluate(x_test, y_test)
```

```
results
```

```
##      loss       acc
## 0.7355296 0.8763613
```

In this way, this model achieved an accuracy of 87.63613.

Following is the comparison of different model results so far.

```
model_results5<-rbind(model_results4,data.frame(method="1D Convent",
    accuracy=results[[2]],loss=results[[1]]))
kable(model_results5)
```

method	accuracy	loss
Dense Layer	0.8626754	0.8891655
Dense Layer with Dropout	0.8790402	0.5989639
Dense Layer with L2 Weight Regularization	0.8787491	0.3880340
Bidirectional LSTM	0.8995982	0.2761915
1D Convent	0.8763613	0.7355296

5.5 1D conv with 2 bidirectional LSTMs Model

One strategy to combine the speed and lightness of convnets with the order sensitivity of RNNs is to use a 1D convnet as a preprocessing step before an RNN (See Fig 13)

This is especially beneficial when you're dealing with sequences that are so long they can't realistically be processed with RNNs, such as sequences with thousands of steps. The convnet will turn the long input sequence into much shorter (downsampled) sequences of higher-level features. This sequence of extracted features then becomes the input to the RNN part of the network.

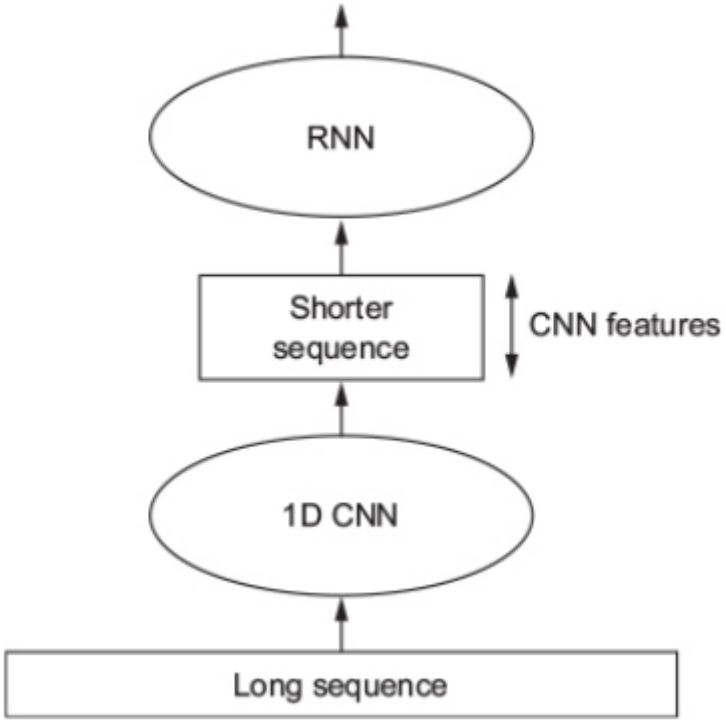


Figure 13: Combining a 1D Convent and an RNN for processing long sequences

5.5.1 Building the Model

We build our model using the following code.

```
k_clear_session() # clears keras session

model <- keras_model_sequential() %>%
  layer_embedding(input_dim = max_features, output_dim = 128,
                  input_length = maxlen) %>%
  layer_dropout(0.5)%>%
  layer_conv_1d(filters = 250, kernel_size = 3, activation = "relu") %>%
  layer_max_pooling_1d(pool_size = 5) %>%
  layer_conv_1d(filters = 250, kernel_size = 3, activation = "relu") %>%
  bidirectional(
    layer_lstm(units = 64, return_sequences=TRUE)
  ) %>%
```

```
  bidirectional(
    layer_lstm(units = 64)
  ) %>%
  layer_dense(units = 1, activation = "sigmoid")
```

Here is the summary of the model.

```
summary(model)
```

```
## Model: "sequential"
## _________________________________________________________________________________
## Layer (type)                        Output Shape                    Param #
## =================================================================================
## embedding (Embedding)               (None, 500, 128)                2560000
## _________________________________________________________________________________
## dropout (Dropout)                   (None, 500, 128)                0
## _________________________________________________________________________________
## conv1d_1 (Conv1D)                   (None, 498, 250)                96250
## _________________________________________________________________________________
## max_pooling1d (MaxPooling1D)        (None, 99, 250)                 0
## _________________________________________________________________________________
## conv1d (Conv1D)                     (None, 97, 250)                 187750
## _________________________________________________________________________________
## bidirectional_1 (Bidirectional)     (None, 97, 128)                 161280
## _________________________________________________________________________________
## bidirectional (Bidirectional)       (None, 128)                     98816
## _________________________________________________________________________________
## dense (Dense)                       (None, 1)                       129
## =================================================================================
## Total params: 3,104,225
## Trainable params: 3,104,225
## Non-trainable params: 0
## _________________________________________________________________________________
```

Next we shall compile the model using loss function as "binary_crossentropy" (refer to section Objective (loss) Function) and using RMSprop optimizer (refer to section Optimizer).

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

5.5.2 Model Training

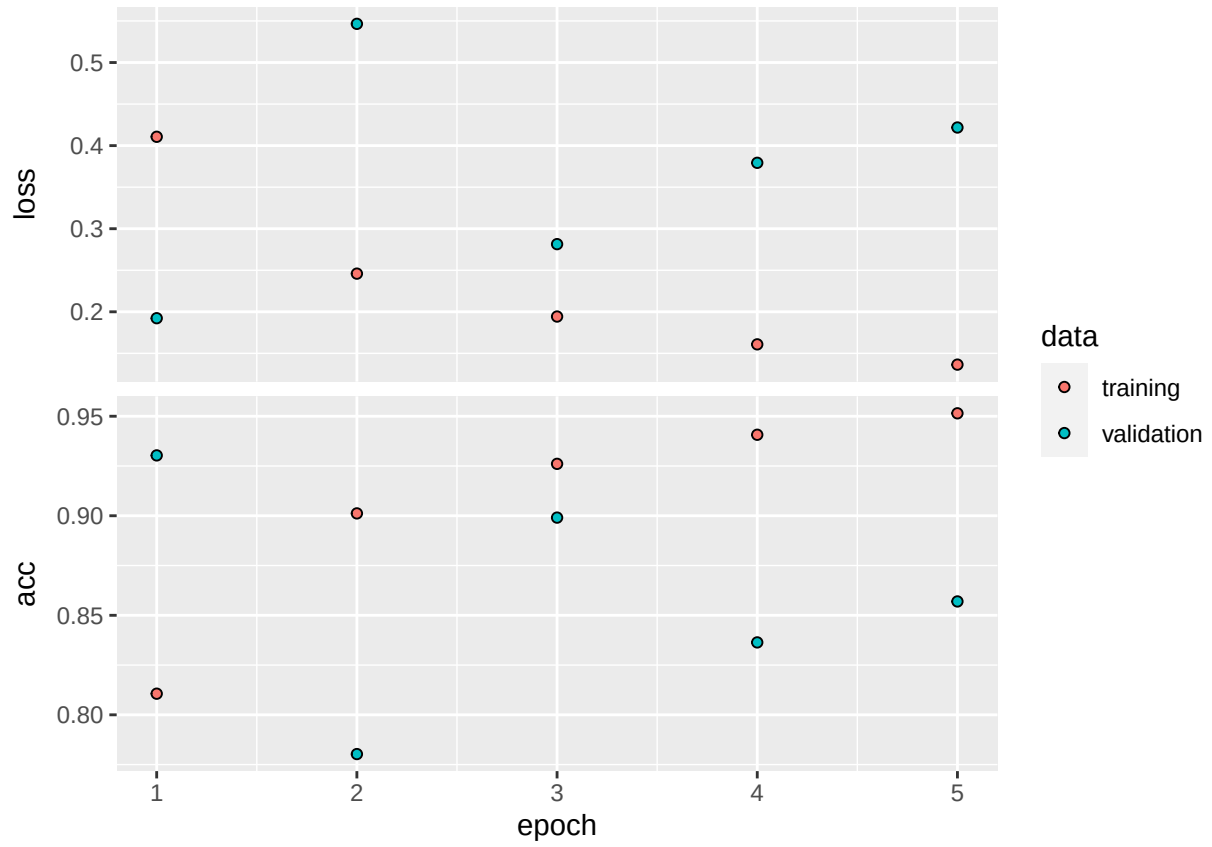
We shall fit the model on training dataset first by dividing it into 80% training and 20% validation. Here is the code.

```
history <- model %>% fit(
  x_train, y_train,
  epochs = 5,
```

```
  batch_size = 32,
  validation_split = 0.2
)
```

To visualize training and validation metrics by epoch we shall plot the `history`:

```
plot(history)
```



The epoch at which maximum validation accuracy is achieved is

```
epoch<-which.max(history$metrics$val_acc)
epoch
```

```
## [1] 1
```

While the maximum validation accuracy achieved is

```
max(history$metrics$val_acc)
```

```
## [1] 0.9303183
```

Now, let's fit our model for the whole train dataset for epoch = 1

```
model %>% fit(x_train, y_train, epochs = epoch, batch_size = 32)
```

5.5.3 Results

We use keras evaluate function to evaluate our model on test data.

Finally, this model gives us following accuracy and loss on the test data.

```
results <- model %>% evaluate(x_test, y_test)
```

```
results
```

```
##      loss       acc
## 0.3024800 0.8924931
```

In this way, this model achieved an accuracy of 89.24931.

Following is the comparison of different model results so far.

```
model_results6<-rbind(model_results5,data.frame(method="1D Convent with two bidirectional LSTMs",
    accuracy=results[[2]],loss=results[[1]]))
kable(model_results6)
```

method	accuracy	loss
Dense Layer	0.8626754	0.8891655
Dense Layer with Dropout	0.8790402	0.5989639
Dense Layer with L2 Weight Regularization	0.8787491	0.3880340
Bidirectional LSTM	0.8995982	0.2761915
1D Convent	0.8763613	0.7355296
1D Convent with two bidirectional LSTMs	0.8924931	0.3024800

6 Conclusion and Final Thoughts

Looking into the table we created for each model accuracy and loss, we found that stacked bidirectional LSTM model performed the best. Compared to that, combination of 1D Convent with LSTMs was also close. In this way, the later was more effective considering the less computational cost and quick speed of execution.

Though our models achieved a good accuracy, we still think that a much better performance can be obtained. Following are some final thoughts which could not be implemented due to lack of time and resources.

6.1 GPU Execution and Use of Callbacks

As mentioned in Implementation Notes, most of these models were run on CPU, due to which we only tried a limited number of epochs. We believe a better result can be obtained by running these models on GPU, trying more epochs during training and even use of callbacks.

6.1.1 Callbacks

When you are training a model, there are many things you can't predict from the start. In particular, you can't tell how many epochs will be needed to get to an optimal validation loss. The models in this study have adopted the strategy of training for enough epochs that you begin overfitting, using the first run to figure out the proper number of epochs to train for, and then finally launching a new training run from scratch using this optimal number. Of course, this approach is wasteful.

A much better way to handle this is to stop training when you measure that the validation loss in no longer improving. This can be achieved using a Keras callback. A callback is an object that is passed to the model in the call to fit and that is called by the model at various points during training. It has access to all the available data about the state of the model and its performance, and it can take action: interrupt training, save a model, load a different weight set, or otherwise alter the state of the model.

Here are some examples of ways you can use callbacks:

- Model checkpointing—Saving the current weights of the model at different points during training.
- Early stopping—Interrupting training when the validation loss is no longer improving (and saving the best model obtained during training).
- Dynamically adjusting the value of certain parameters during training—Such as the learning rate of the optimizer.
- Logging training and validation metrics during training, or visualizing the representations learned by the model as they're updated.

6.2 Batch Normalizaton

Normalization is a broad category of methods that seek to make different samples seen by a machine-learning model more similar to each other, which helps the model learn and generalize well to new data. The most common form of data normalization is centering the data on 0 by subtracting the mean from the data, and giving the data a unit standard deviation by dividing the data by its standard deviation. In effect, this makes the assumption that the data follows a normal (or Gaussian) distribution and makes sure this distribution is centered and scaled to unit variance.

Data normalization should be a concern after every transformation operated by the network, even if the data entering `layer_dense` or `layer_conv_2d` has a 0 mean and unit variance, there's no reason to expect a priori that this will be the case for the data coming out.

Batch normalization is a type of layer (layer_batch_normalization in Keras) introduced in 2015 by Ioffe and Szegedy. It can adaptively normalize data even as the mean and variance change over time during training. It works by internally maintaining an exponential moving average of the batch-wise mean and variance of the data seen during training. The main effect of batch normalization is that it helps with gradient propagation—much like residual connections—and thus allows for deeper networks. Some very deep networks can only be trained if they include multiple batch normalization layers.