RNN model using IMDB Dataset

Introduction:

Artificial neural networks called recurrent neural networks (RNNs) are made to identify patterns in data sequences including spoken language, handwriting, genomes, and text. RNNs are very successful for jobs where context is important because, unlike feedforward neural networks, they can handle sequences of inputs by using their internal state, or memory.

Text classification tasks such as sentiment analysis, subject labeling, spam detection, and intent detection have all been effectively handled by RNNs. Their capacity to identify temporal connections and context in the supplied data makes them very helpful in these tasks.

Dataset:

One widely used dataset for sentiment analysis is the IMDB Large Movie Review Dataset. It has fifty thousand very polar movie reviews (twenty-five thousand train and twenty-five thousand test sets). Of the reviews, half have a rating of seven or higher on a ten-point scale, and the other half have a rating of four or below.

Data preprocessing:

First, the lines train_data \- train_data[1:100,] and train_labels <- train_labels[1:100] limit the training data and labels to the first 100 samples. In the development phase, this is done to expedite the training process.

Next, the line train_data <-pad_sequences(train_data, maxlen = 150) restricts the training data to the first 150 words of each review. This is done in order to guarantee that, as is necessary for training RNNs, every sequence in the input data is the same length.

Next, test_data <-imdb$test$x and test_labels <-imdb$test$y are loaded to provide the test data for validation. Test_data <- pad_sequences(test_data, maxlen = 150) restricts the test data to the first 150 words of every review.

Model Building:

The keras_model_sequential() method is used to build the model, producing a linear stack of layers. Three layers make up the model:

An **embedding layer** that creates dense vectors of a predetermined size from positive integers (indexes). This layer is limited to being utilized as the model's top layer. The vocabulary size is given by the input_dim parameter, and the dense vector size is given by the output_dim parameter.

A layer called **SimpleRNN** completes the implementation of a recurrent neural network. The dimensionality of the output space is the units parameter.

For binary classification tasks, a **Dense layer**, a typical densely-connected neural network layer—with a sigmoid activation function is employed.

The binary cross-entropy loss function, appropriate for binary classification, the RMSprop optimizer, and accuracy as the evaluation measure are used in the compilation of the model.

Next, the model is trained for 10 epochs with a batch size of 512 using the fit() method on the training data. This function also receives the validation data, which is used to assess the model's performance at the conclusion of each epoch.

Lastly, the history object that the fit() method provided is used to extract and report the validation accuracy of the most recent epoch. This helps you gauge the model's effectiveness using the validation data.

Changing the number of Different Samples:

In order to test alternative numbers of training samples, the code first constructs a vector called num_samples. The validation accuracies for each quantity of training samples are then stored in a vector called val_acc, which is initialized.

The amount of training samples is then looped over by the algorithm. It determines if the number of samples is fewer than or equal to the total number of samples in the training data for each number of samples. If so, it sets a cap on the total number of labels and training samples.

It then uses the same steps to define, assemble, and train an RNN model. On the other hand, the subset of training data is used to train the model this time. The validation data doesn't change.

In order to test alternative numbers of training samples, the code first constructs a vector called num_samples. The validation accuracies for each quantity of training samples are then stored in a vector called val_acc, which is initialized.

The amount of training samples is then looped over by the algorithm. It determines if the number of samples is fewer than or equal to the total number of samples in the training data for each number of samples. If so, it sets a cap on the total number of labels and training samples.

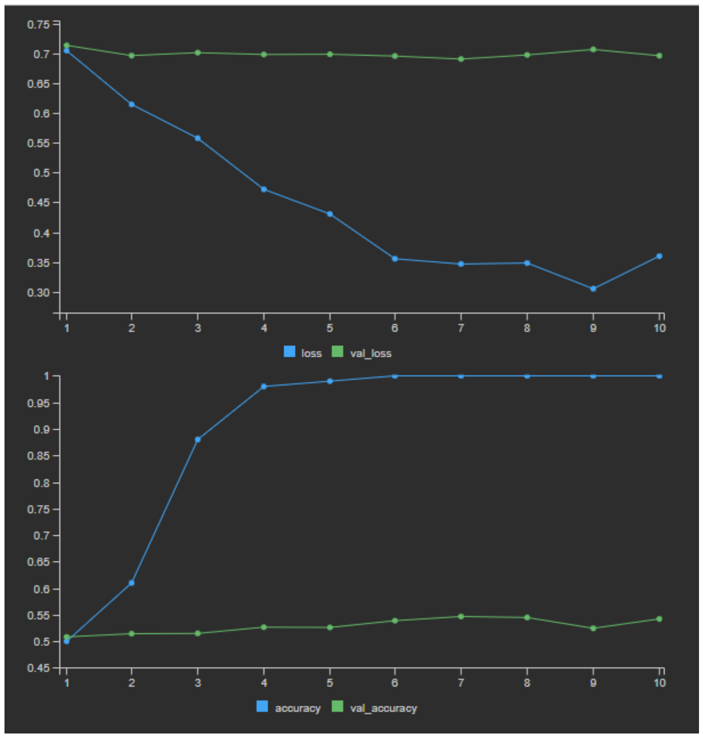
It then uses the same steps to define, assemble, and train an RNN model. On the other hand, the subset of training data is used to train the model this time. The validation data doesn't change. The results data frame is utilized to initialize the plot using the ggplot() function, and the num_samples and val_acc variables are mapped to the x- and y-axes, respectively, using the aes() function.

For every combination of num samples and val acc, the geom_point() method adds a point to the plot, and the geom_line() function joins these points with lines. This makes the trend in validation accuracy as the quantity of training samples rises easier to see.

The labs() function is used to add labels to the x-axis, y-axis, and the title of the plot. In this case, the x-axis is labeled as “Number of Training Samples”, the y-axis is labeled as “Validation Accuracy”, and the title of the plot is “Model Performance”. Finally, the theme_minimal() function is used to apply a minimalistic theme to the plot, which removes most of the non-data ink and provides a clean and focused visualization.

Results and Conclusion:

Model Performance:

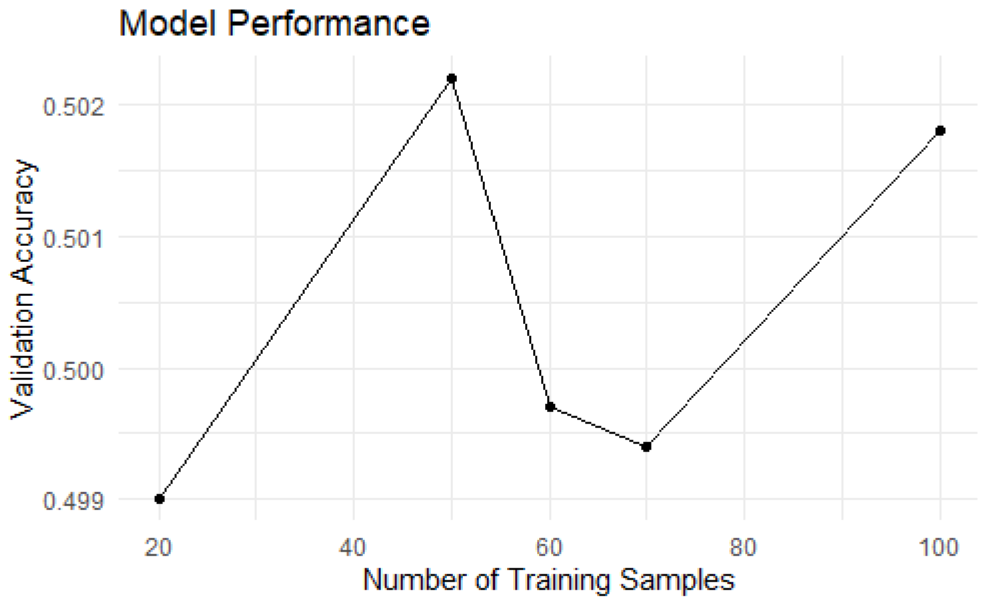


The blue line, which represents the model's loss, shows that it begins at 0.7 and drops rapidly to about 0.3 at epoch 5, after which it gradually increases. This suggests that the model is picking up new skills and becoming more efficient. Nonetheless, the validation loss (shown by the green line) stays largely steady at 0.7, indicating that the model may not be adequately generalizing to new data.

The model's accuracy (blue line) and validation accuracy (green line) are displayed in the bottom graph. By epoch ten, the accuracy has increased substantially from just over 0.5 to just below 1, suggesting that the model is getting more and more accurate with the training set. Nonetheless, the validation accuracy is steady over all epochs, roughly between.55 and.65, indicating that the model might not be generalizing well to unseen data.

Using different samples:

The training sample is then divided into number of samples to see the performance of embedding layer at different number of samples. For loop iterates the model on different number of samples one by one and at last we receive the accuracy of last epochs of all the samples which is plotted using the ggplot2 package to see the performance od the layer.



After looking at the above graph we can interpret that Embedding layer works better when the number of training samples are 60.

Due to the restrictions in R deep learning packages, it was not possible t compare a model utilizing an embedding layer with a model with a model using pre-trained embedding (such as

GloVe or Word2Ves). However, because pretrained embeddings make use of information gleaned from vast quantities of data, it is well recognized that they frequently enhance performance, particularly in situation when the amount of training data is constrained. Although the model was able to learn from the training data to some extent, the accuracy indicates that there is still a great deal of space for improvement.