Recurrent Neural Network for Language Processing

1st John McDonald
Electrical and Computer Engineering Dept
North Carolina State University
Raleigh, North Carolina
jamcdon3@ncsu.edu

***Abstract*—This is an assignment for the ECE 542 course to become familiar with Recurrent Neural Networks (RNN) and the Long Short Term Memory (LSTM) unit. The paper is set up into two parts, with the first goal to design a LSTM for various counting tasks, and the second goal to design an RNN using LSTM for language processing in Words and characters. Alternatively, the goal is to become familiar with Tensorflow and in this case Keras Python libraries, for the use of RNN architecture. This paper includes the details of the delivered neural network structure and additionally performance analysis, using a high level API library. The performance of the RNN is described across various epoch training's over time.**

***Index Terms*—Recurrent Neural Networks, Long Short Term Memory, Machine Learning, Keras, Tensorflow**

I. Structure of LSTM

The LSTM cell and defined functionality is built in Python using instructor-provided scripts. All code for the cell and initiative is built from scratch, not using tensorflow. The class heirarchy is split between multiple files. Functionality of these files are listed below:

TABLE I
File Structure for part 1

File	Functionality
LSTM	build single LSTM cell
assign	initialize LSTM with weights and biases
count	define tasks and iterate steps through input
run	Define input and text interface

The structure of the LSTM is build exactly as described from the class lectures. Neurons with weights and biases are used to control inputs, gates, and outputs. The cell has a structure that supports manipulation of input, gate for input, gate for forgetting data from previous LSTM cells, and gate for submitting output. This structure of the cell can be used to process data in a time series, by maintaining previous and current states. To help describe this functionality an image of the LSTM structure is in Fig. 1.

This cell is constructed in the lstm.py file. Similar to the diagram, sigmoid and tanh activation functions are used to pass and process data. Sigmoids operate conveniently as gate activation functions, as it can support binary operations. Part one uses binary values to keep the structure of the LSTM

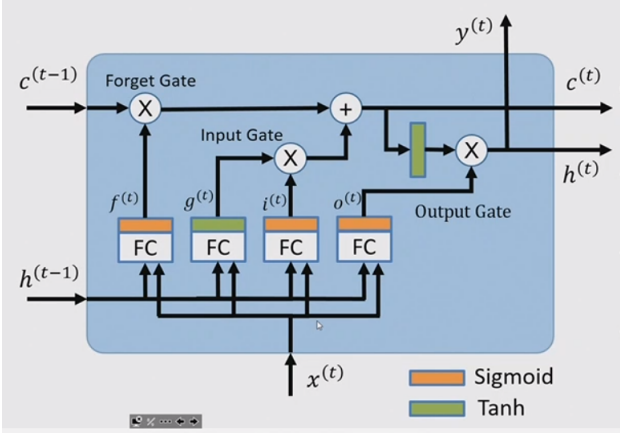

Fig. 1. Standard Long Short Term Memory cell as described in lecture. Used for applications in recurrent neural networks [1].

simple, in order to logically create a counter. The counter for this part will use the g(t) to read inputs from a series of numbers. The numbers of interest for this part are 0, 2, and 3, so all other numbers will be weighted at 0. Our input gate will use i(t) as the gate. This gate will be relevant when taking into account parameters from previous states, for example memorizing important parameters. The f(t) function will act as a forget gate, and is useful for removing data from memory. The o(t) function is often used to learn when to output memory; however, in this part we will not use the output function as our output is kept high to logically understand the flow and function of the LSTM. Also, Biases will not need to be used in this problem - only weights for layers between cells.

II. Parameters for Task 2 and Task 3

In this part I will include code to describe parameters chosen for Tasks 2 and Tasks 3. The structure of how weights and biases are assigned to the LSTM determine the parameterization, and how data is processed through a time series. The other parameters of significance are the number of in dimensions and out dimensions when building the LSTM. In dimensions determine the one hot encoding of data being input to the series - this will not change for any task and stays valued 10, for digits 0-9. The out dimensions determines the storage

capacity for the time series - this is directly proportional to complexity of functionality.

A. Task 2

The task 2 objective is to count the number of 0 after receiving the first 2 in the sequence. The out dimension for LSTM is increase from 1 to 2, as now the LSTM must remember two components: count of 0's and whether a 2 has been read. The assigned weights and biases are similar to task 1. The parameters are as follows:

```
input_count = [[100.] if i == 0 else [-0.] for i in range(10)]
input_gate = [[100.] if i == 2 else [-0.] for i in range(10)]
param_dict['wgx'] = np.concatenate((input_count, input_gate), axis=-1)
param_dict['wgh'] = np.zeros((out_dim, out_dim))
param_dict['bg'] = np.zeros((1, out_dim))

input_count = [[-100.] if i == 0 else [-100.] for i in range(10)]
input_gate = [[100.] if i == 2 else [-100.] for i in range(10)]
param_dict['wix'] = np.concatenate((input_count, input_gate), axis=-1)
param_dict['wih'] = [[0.,0.], [200., 200.]]
param_dict['bi'] = np.zeros((1, out_dim))

param_dict['wfx'] = np.zeros((in_dim, out_dim))
param_dict['wfh'] = np.zeros((out_dim, out_dim))
param_dict['bf'] = 100. * np.ones((1, out_dim))

param_dict['wox'] = np.zeros((in_dim, out_dim))
param_dict['woh'] = np.zeros((out_dim, out_dim))
param_dict['bo'] = 100. * np.ones((1, out_dim))
```

Fig. 2. Assigned weight and biases for task 2, LSTM.

We see that the weight for 'wgx' is now assigned to read inputs that are either a 0 or 2. The 'wix' weights for input gate are dependent upon the previous state, allowing the cell to turn on the gate once a 2 input value has been previously read. Until then, all inputs will be ignored.

B. Task 3

The task 2 objective is to count the number of 0 after receiving the first 2 in the sequence, but erase the counts after receiving 3, then continue to count from 0 after receiving another 2. The out dimension for LSTM is still 2, as the LSTM must remember two components: count of 0's and whether a 2 has been read. But now functionality will be added to the forget gate, for when a 3 is read. The assigned weights and biases are similar to task 2, except for the following differences shown in Fig. 4.

We see that now the 'wfx' weight is included to erase memory whenever a 3 is read from the input x. The cell continues to read and count 0's once a 2 is read, but forgets all memory once a 3 is read.

III. Plotted Output through Time Series

Values are plotted for the input node, internal state, input gate, forget gate and output gate as functions of time step for the example sequence [1, 1, 0, 4, 3, 4, 0, 2, 0, 2, 0, 4, 3, 0, 2, 4, 5, 0, 9, 0, 4]. Graphs are included in this section.

```
input_count = [[100.] if i == 0 else [-0.] for i in range(10)]
input_gate = [[100.] if i == 2 else [-0.] for i in range(10)]
param_dict['wgx'] = np.concatenate((input_count, input_gate), axis=-1)
param_dict['wgh'] = np.zeros((out_dim, out_dim))
param_dict['bg'] = np.zeros((1, out_dim))

input_count = [[-100.] if i == 0 else [-100.] for i in range(10)]
input_gate = [[100.] if i == 2 else [-100.] for i in range(10)]
param_dict['wix'] = np.concatenate((input_count, input_gate), axis=-1)
param_dict['wih'] = [[0.,0.], [200., 200.]]
param_dict['bi'] = np.zeros((1, out_dim))

input_count_num = [[-100.] if i == 3 else [100.] for i in range(10)]
input_gate_num = [[-100.] if i == 3 else [100.] for i in range(10)]
param_dict['wfx'] = np.concatenate((input_count_num, input_gate_num), axis=-1)
param_dict['wfh'] = np.zeros((out_dim, out_dim))
param_dict['bf'] = np.zeros((1, out_dim))

param_dict['wox'] = np.zeros((in_dim, out_dim))
param_dict['woh'] = np.zeros((out_dim, out_dim))
param_dict['bo'] = 100. * np.ones((1, out_dim))
```

Fig. 3. Assigned weight and biases for task 3, LSTM.

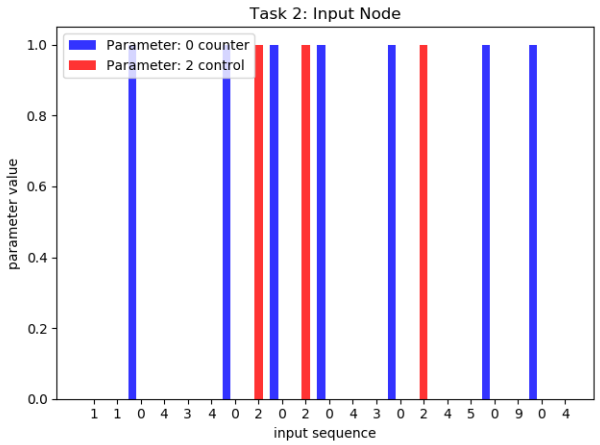

Fig. 4. Input node for the LSTM. Parameter activates 'high' when input is read for targeted values.

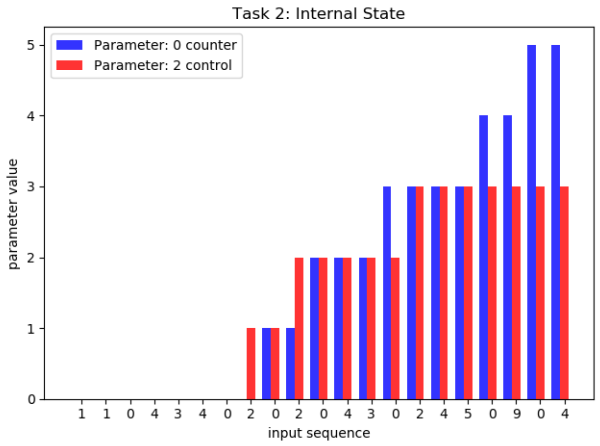

Fig. 5. Internal state for the LSTM. For this task we see that counter does not begin until the first 2 is read. Afterwards, the counter increments to five before finishing the input sequence.

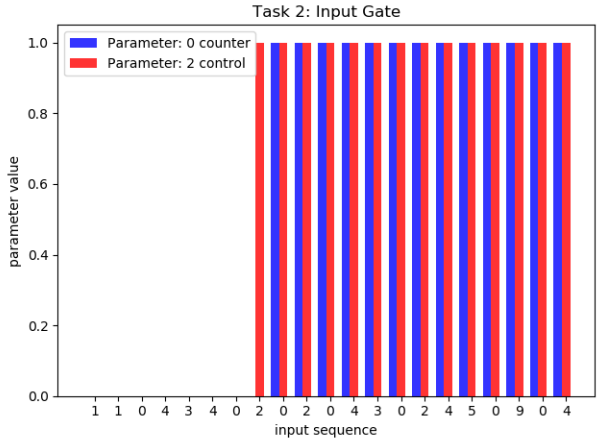

Fig. 6. Input gate for the LSTM. The input gate does not pass values to storage until the first 2 is read. Afterwards, the input gate stays high to read all 0 and 2 values to storage.

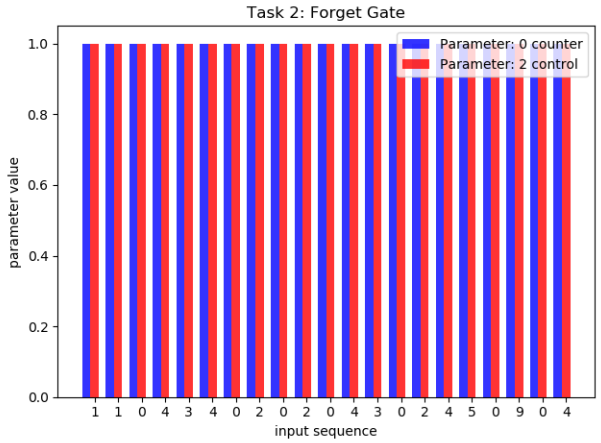

Fig. 7. Forget gate for the LSTM. Forget gate stays high for the whole sequence. In task 2, it is not necessary to remove information from storage. So the LSTM continues to pass counter values to next cell.

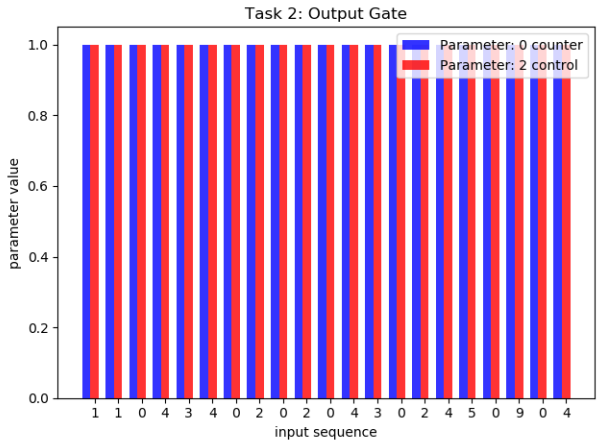

Fig. 8. Output gate for the LSTM. Output gate stays high, as we do not use this functionality in part 1.

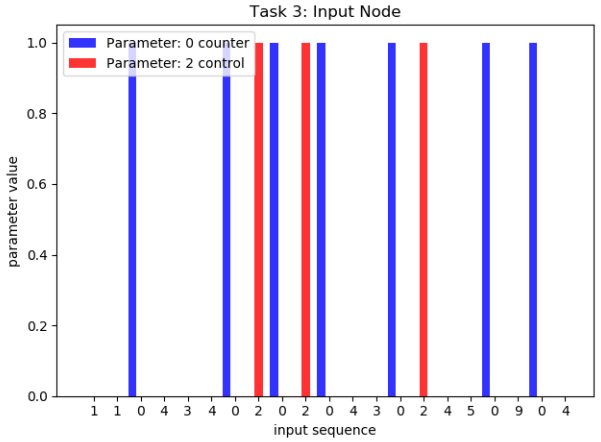

Fig. 9. Input node for the LSTM. Parameter activates 'high' when input is read for targeted values.

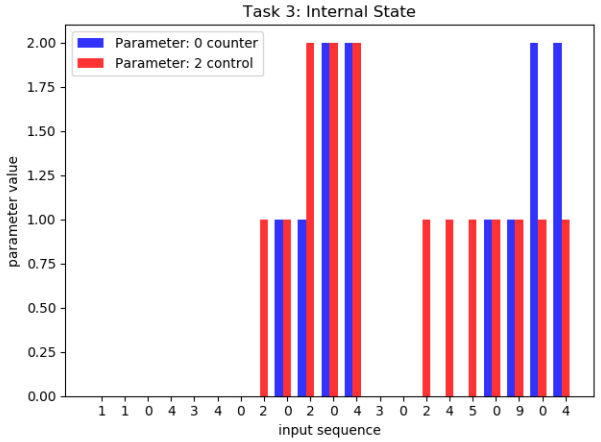

Fig. 10. Internal state for the LSTM. For this task we see that counter does not begin until the first 2 is read. Afterwards, the counter increments on reading 0's. Once a 3 is read, all memory is erased.

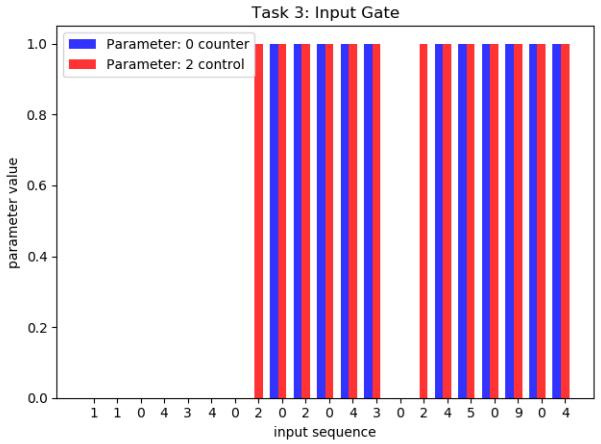

Fig. 11. Input gate for the LSTM. The input gate does not pass values to storage until the first 2 is read. Afterwards, the input gate stays high to read all 0 and 2 values to storage. This is reset after reading a 3.

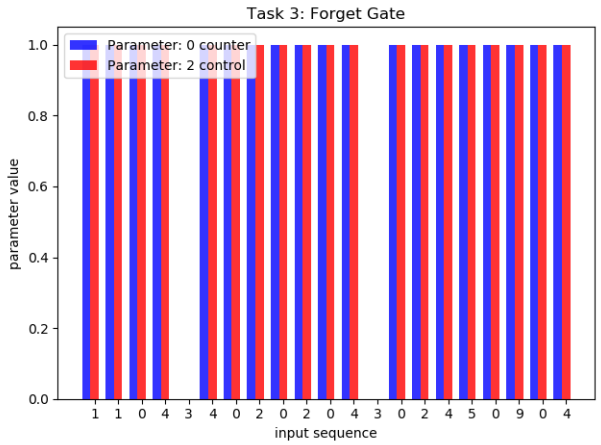

Fig. 12. Forget gate for the LSTM. Forget gate activates when a 3 is read. The gate turns 0, to block all memory from passing to the LSTM, thus forgetting memory.

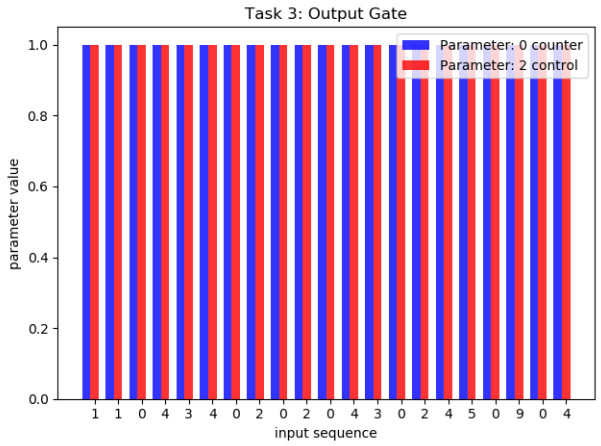

Fig. 13. Output gate for the LSTM. Output gate stays high, as we do not use this functionality in part 1.

IV. Structure of the RNN

The objective of the RNN is to read data from a Penn Tree Bank dataset, and use it for language processing and prediction for both words and characters. This RNN uses multiple LSTM units stacked on top of each other to learn in a time series. This is shown in Fig. 4.

The code uses Tensorflow high level library to set up and create this RNN. Tensorflow also provides template code for this RNN build, to aid with complexity of code [3]. The parameters chosen mainly based on my local computers processing demands. The RNN training can become very cumbersome on CPU performance. Parameters are chosen as shown in Table II.

Note, to handle computational intensity for character prediction, I reduced the number of LSTM units to 100.

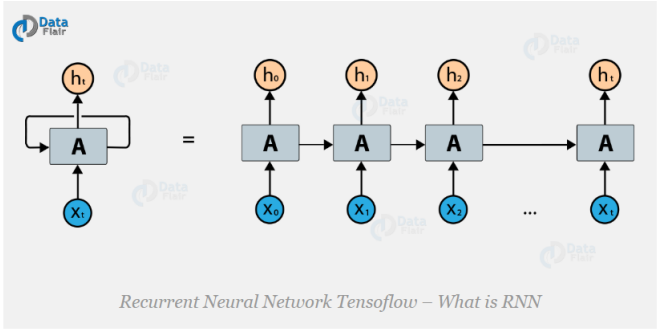

Fig. 14. Structure of RNN using LSTM's [2].

TABLE II
Parameters for RNN

Parameter	Description	Value
init_scale	the initial scale of the weights	0.1
learning_rate	the initial value of the learning rate	1.0
max_grad_norm	the maximum permissible norm of the gradient	5
nu_layers	the number of LSTM layers	2
num_steps	the number of unrolled steps of LSTM	20
hidden_size	the number of LSTM units	200
max_epoch	number of epochs trained w/ learning_rate	4
max_max_epoch	the total number of epochs for training	10
keep_prob	probability of keeping weights in dropout	1.0
lr_decay	the decay of the learning rate	0.5
batch_size	the batch size	20
rnn_mode	the low level implementation of lstm cell	block

V. Perplexity on Training and Validation

Using the chosen parameters in section 5, I trained the RNN for word-level and character-level predictions. Perplexity of the model is measured over time per epoch and included in this section. Optimization was rough, provided this computational intensity for my computer. However, I managed to get steady Perplexity and Performance curves for 10 epochs each.

A. Word Level Prediction

Time for training for all 10 epochs took approximately 35 minutes on my local computer. As specified by hyper parameter, the initial learning rate was 1, and by epoch 10 reduced to 0.016. The results for perplexity curve is included in image 15.

The validation perplexity stabilizes at around 120 after training 5 epochs. Training perplexity continues to decrease to around 41. At the 7th epoch, the validation perplexity starts to increase slightly indicating that the model is being over trained. During the beginning epochs of the model, there is an exponential decrease in perplexity showing rapid progress for training.

B. Character Level Prediction

Time for training for all 10 epochs took 1.5 hours. To handle computational intensity, I reduced the number of hidden LSTM units from 200 to 100. Similar to the word prediction model,

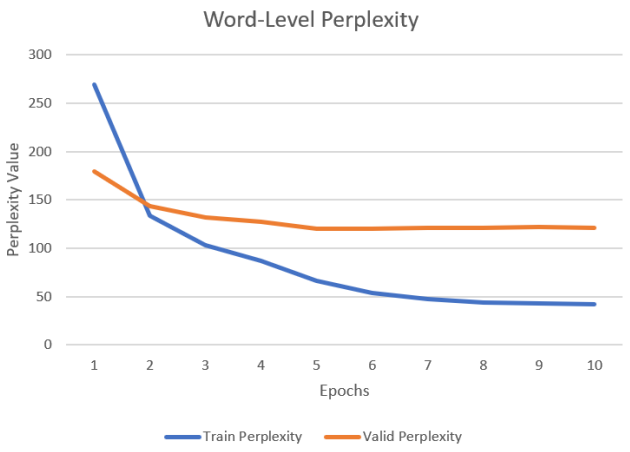

Fig. 15. Perplexity for RNN Word-level prediction over number of epochs.

the learning rate for Epoch 1 started at 1.0, and was reduced to 0.016 by Epoch 10. Note that the perplexity for character prediction is significantly lower than the word-prediction. This is because the model is only developing the prediction based on a character index bank of 49. So, the possible outputs provided an input prediction sequence are significantly lower, contributing to a lower perplexity.

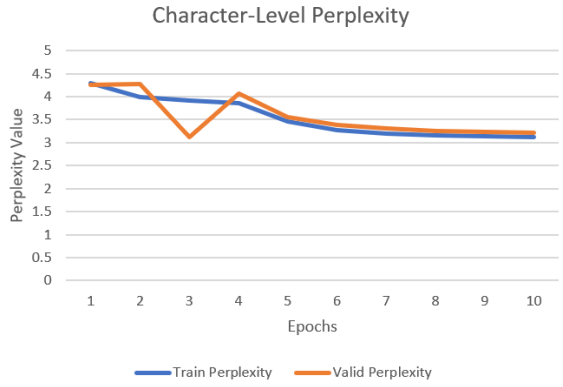

Fig. 16. Perplexity for RNN Character-level prediction over number of epochs.

The validation perplexity continues to decrease until the last epoch, ending at a value of 3.223. The training perplexity also decreases until around 3.1. The continual decrease in validation and training perplexities indicates that the model can be further trained, for additional accuracy. Another interesting characteristic of the graph is the sudden dip in validation perplexity for epoch 3. The training curve also looks significantly different than the word perplexity model. This curve learns at a much more linear rate, whereas the word-perplexity model learned exponentially fast at the beginning.

VI. RNN Performance

The accuracy of the model was tested against the ptb.test.txt and ptb.char.test.txt files included in the data folder. Test perplexity is included below.

TABLE III
Test Perplexity for PTB Datasets

Prediction	Perplexity
Word-level	115.429
Char-level	3.110

The test perplexity was measured to be lower than validation perplexity for both word processing and character processing models. However, for word-processing, the training perplexity was significantly lower than both training and test perplexity. This indicates that the model is heavily over trained, and can be further optimized. Also, the fact that the training and validation perplexities increasingly decreased for character processing model, indicates that the model may be under trained, and can be further optimized.

In order to display predictions, an input prediction sequence of text ID's needs to be fed to the model. Inputs can be selected using word ID's as created by the reader.py for the PTB dataset. In figure 17 and 18, the text ID's are printed. These text ID's can be cross referenced with displayed predictions to read performance of the model.

Predictions for the model can freely be generated by using these word maps to select an input, or beginning part of the sequence, then using the model to predict the rest of the sequence for a target number of words or characters.

Fig. 17. Word to ID map for word PTB builder.

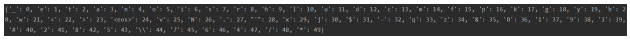

Fig. 18. Word to ID map for character PTB reader.

References

[1] E. Lobaton, "ECE 542 - CSC 591 Lecture 018," in ECE 542 - CSC 591 Lecture 018, 11-Nov-2019.

[2] "Recurrent Neural Networks," TensorFlow - Recurrent Neural Networks. [Online]. Available: https://chromium.googlesource.com/external/github.com/tensorflow/tensorflow/-/refs/heads/0.6.0/tensorflow/g3doc/tutorials/recurrent/index.md. [Accessed: 11-Nov-2019].

[3] D. F. Team, "Recurrent Neural Network TensorFlow: LSTM Neural Network," DataFlair, 15-Sep-2018. [Online]. Available: https://data-flair.training/blogs/tensorflow-recurrent-neural-network/. [Accessed: 11-Nov-2019]. https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6. [Accessed: 15-Oct-2019]."Why use Keras?,"