Available online www.jsaer.com

Journal of Scientific and Engineering Research, 2020, 7(1):309-315

Research Article

ISSN: 2394-2630
CODEN(USA): JSERBR

Word Generation Using Recurrent Neural Network

Naga Sai Krishna Mohan Pitchikala[1], Saisuhas Kodakondla[2], Debargha Ghosh[3]

[1]master’s in computer science, University of Texas at Dallas, Texas, USA
nxp180022@utdallas.edu
[2]master’s in computer science University of Texas at Dallas, Texas, USA
sxk180114@utdallas.edu
[3]Master’s in computer science University of Texas at Dallas, Texas, USA
dxg170014@utdallas.edu

Abstract: Computers have influenced the life of humans to a very great extent. Natural Processing language is a field of computer science which helps to exchange information very efficiently between humans and machines with less human requirement. Text generation techniques can be applied for improving language models, machine translation summarizing and captioning. In this project we train a Recurrent Neural Network so that it can generate new words related to the words that are fed to it.

Keywords: Recurrent Neural Network, Computers

1. Introduction

Recurrent Neural Networks (RNNs) can also be used to create new data, making them generative models. This means that in addition to being used for predictive models they can learn from the sequence of the problem and then generate entirely new plausible sequence for the problem domain.

Recurrent Neural Networks are a flexible type of model used for tasks that involve sequences, like language or time series data. They are powerful because they can remember and process past information through their hidden states, which evolve in complex ways. Additionally, RNNs are efficient when calculating gradients using a method called backpropagation through time. Despite these advantages, RNNs haven't become widely used in machine learning because they are difficult to train effectively. The main issue is that the connection between their parameters and hidden states can be unstable, leading to problems like vanishing or exploding gradients. This has made training RNNs challenging, and as a result, there has been limited research on standard RNNs in recent years. Some still use them, for example, in word-level language models. This paper aims to show how an RNN can be used to generate new words.

2. Theory

A. Neural Network

Neural networks are a type of computer algorithm inspired by how the human brain works. They are designed to recognize patterns in data, like images, sounds, or text. To do this, they first convert real-world data into numbers (called vectors), which they can process and understand.

Neural networks are useful for two main tasks: grouping similar data together (clustering) and sorting data into categories (classification). If the data doesn’t have labels, the network can find patterns and group similar items. If the data is labeled, the network can learn from it and classify new data based on those labels. In short, neural networks help organize and make sense of the information you have.

A traditional neural network would be one that is a no-frills, Feedforward Multilayer Perceptron. They have 3 layers namely Input layer, Hidden layer and Output layer. Backpropagation is almost always included in this definition. With the help of back propagation algorithm, we train the neural network to achieve good results otherwise it's hard to use even these relatively simple networks to solve real problems. Hidden layers help in mapping input to the output, there can be a minimum of one hidden layer and maximum of 4 hidden layers. Beyond 4 the network transforms into deep neural network. The figure below shows the Neural Network with 1 Hidden layer. Each node in input and output layers is called a neuron. It sums up all the inputs and uses a activation function on the sum of inputs to classify them.

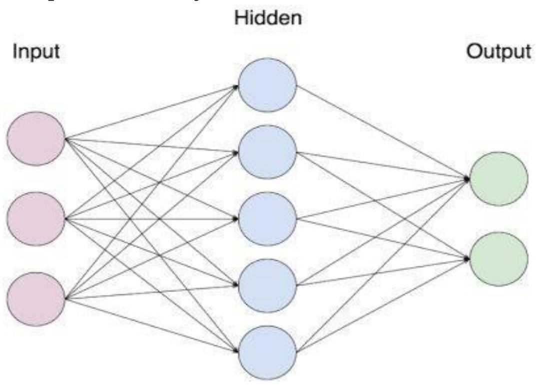


Fig. 1: *Neural Network*

B. Limitations of Neural Network

There are some limitations in a neural network due to its architecture. They are:

• Accept fixed size vector as an input

• Produce fixed size vector as an output

• Performs such input/output mapping using a fixed number of computational steps.

These limitations make it hard to model time series problems when input and output are real value problems. Recurrent Neural Networks are designed for this purpose.

C. Recurrent Neural Network

Traditional Neural Networks do everything from scratch. The output of a neuron depends only on the current input. But in real, we come across situations where the output of an event depends on the previous output along with the current input, in such cases the work of traditional neural network does not satisfy our need, so we developed Recurrent neural network.

Recurrent Neural Network (RNN) deals with sequence information. RNN makes use of available Sequence information. RNNs are called recurrent because they perform

the same task for every element of a sequence by that i mean, they have a "memory" which captures information about what has been calculated so far and pass that memory to the next node. Availability of memory helps in understanding the context of the sequence. RNN models are used widely used in NLP tasks because of their ability to handle sequences.

Normal neuron takes in some input, it can be multiple input so it can aggregate them and then once it aggregates those inputs it passes through some sort of activation function (RELU function in above example) and then an output is generated.

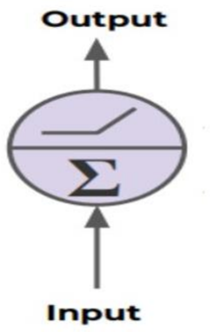


Fig. 2: *Traditional Neuron*

Recurrent Neuron is little different. In Recurrent neural network the output goes back into input of the same neuron. So, we can unroll it throughout time. This is what a Neuron in Recurrent Neural Network looks like.

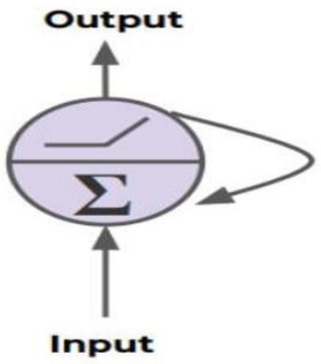


Fig. 3: *Recurrent Neuron*

A recurrent neural network can handle sequences of any length by repeatedly updating its internal memory (called the hidden state) as it processes each part of the sequence. At each step, the hidden state ht is updated based on the current input xt and the memory from the previous step ht−1. This allows the RNN to keep track of the sequence over time.

$$\mathbf{h}_t = \begin{cases} 0 & t = 0 \\ f(\mathbf{h}_{t-1}, \mathbf{x}_t) & \text{otherwise} \end{cases}$$

it's common to use a function that combines a simple transformation of both the current input (xt) and the previous memory (ht−1) with a non-linear function to update the hidden state.
Traditionally, the basic approach for working with sequences is to use an RNN to process the sequence into a fixed-size vector, which can then be passed to a softmax layer for tasks like classification.
However, a challenge with this method is that, during training, the gradients, which help the model learn can either grow too large or shrink too small over long sequences. This exploding or vanishing gradient problem makes it hard for the RNN to learn patterns over long distances in the sequence.
Recurrent Neural Network design for generating new words is shown below. The output from previous Softmax layer is fed to cell along with the new input. The words from the training data set are first preprocessed and after obtaining the unique characters we sort them and map them to their corresponding numbers. These numbers are fed as input to the neural network. The Softmax layer is used to get the probability of each next character.

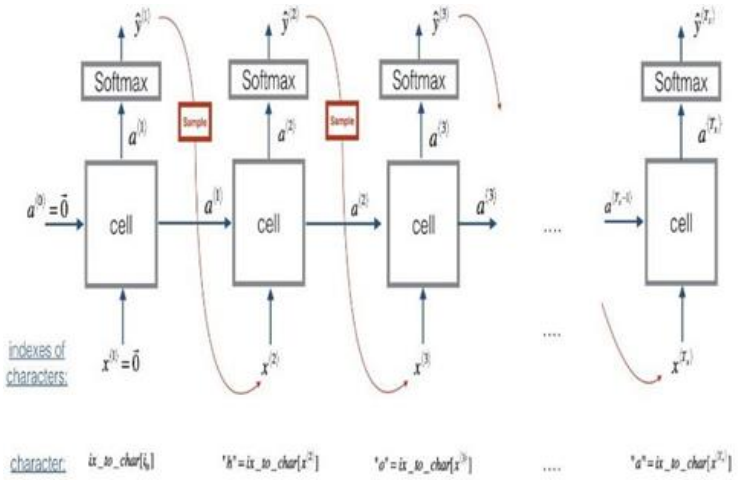

$\hat{y}^{\langle 1 \rangle}$
$\hat{y}^{\langle 2 \rangle}$
$\hat{y}^{\langle 3 \rangle}$
$\hat{y}^{\langle T_x \rangle}$
Softmax
Softmax
Softmax
Softmax
Sample
Sample
$a^{\langle 1 \rangle}$
$a^{\langle 2 \rangle}$
$a^{\langle 3 \rangle}$
$a^{\langle T_x \rangle}$
$a^{\langle 0 \rangle} = \vec{0}$
cell
$a^{\langle 1 \rangle}$
cell
$a^{\langle 2 \rangle}$
cell
$a^{\langle 3 \rangle}$
....
$a^{\langle T_x - 1 \rangle}$
cell
indexes of characters:
$x^{\langle 1 \rangle} = \vec{0}$
$x^{\langle 2 \rangle}$
$x^{\langle 3 \rangle}$
....
$x^{\langle T_x \rangle}$
character:
$ix\_to\_char[i_0]$
$'h' = ix\_to\_char[x^{\langle 2 \rangle}]$
$'o' = ix\_to\_char[x^{\langle 3 \rangle}]$
....
$'a' = ix\_to\_char[x^{\langle T_x \rangle}]$

Fig. 4: *Recurrent Neural Network*

In every iteration we perform forward pass with the cost and computation using entropy loss, backward pass, clipping the gradients if needed and update the parameters
ŷi ⟨t+1⟩ indicates the probability indexed by I is the next character

D. Gradient exploding:

Gradient descent is a method used to optimize a model by gradually adjusting its parameters to minimize a certain error or loss. In machine learning, we use gradient descent to tweak the model's parameters—like the coefficients in linear regression or the weights in neural networks—to make the model more accurate.

During training, an "error gradient" is calculated, which shows the direction and amount by which the model's weights should be updated to reduce the error. However, in deep neural networks or recurrent neural networks, these error gradients can sometimes grow too large. When this happens, it causes the weights to be updated by a large amount, making the network unstable. In extreme cases, the weights can become so large that they produce invalid (NaN) values. This happens because the gradients are multiplied across many layers, and if their values are greater than 1, they can grow exponentially with each step. This growth must be controlled to prevent the model from becoming unstable and to ensure it learns correctly.

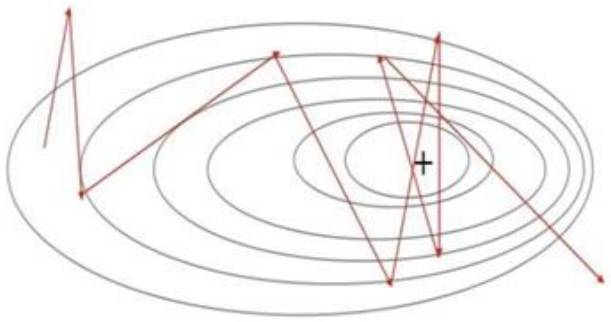


Fig. 5: *Gradients with coefficients greater than 1.*

The issue of exploding gradients in recurrent neural networks happens when the gradients become too large during training. This can lead to an unstable network that either fails to learn from the training data or struggles with long sequences of data. There are a few signs that may indicate your model is experiencing exploding gradients:

- **The model can't learn properly from the training data:** No matter how much you train it, the model makes no progress.
- **The model's loss becomes NaN:** The loss (a measure of how well the model is learning) might become "NaN" (Not a Number), meaning something went wrong during training.
- **The model is unstable:** The loss fluctuates wildly from one update to the next, showing inconsistency.
- **The weights grow too large:** As the model trains, the weights (parameters that adjust during learning) can become extremely large.
- **Gradients are too high:** The error gradient values stay consistently above 1.0 for each node and layer, showing that the gradients are too large for proper learning.

These are clear indicators that the network is affected by exploding gradients, making it difficult or impossible to train effectively.

E. How to overcome Gradient Exploding

There are many ways to overcome exploding gradients problem. The best among those for Recurrent neural networks are addressed below

1. Redesign the Neural Network:

One way to address exploding gradients in RNNs is to reduce the number of time steps the model updates across during training. This technique is called "Truncated Backpropagation Through Time." By limiting how far back the network looks when updating, you can reduce the likelihood of gradients becoming too large.

2. Use LSTM Networks:

RNNs are prone to exploding gradients due to the instability in their training process, especially when using a method called Backpropagation Through Time, which turns the RNN into a very deep network.

A great solution to this is using Long Short-Term Memory (LSTM) units. LSTMs are designed to maintain more stable gradients, making them much better at handling longer sequences without running into exploding gradients. This has become a common best practice when working with RNNs for tasks like sequence prediction.

3. Gradient Clipping:

If exploding gradients still occur, you can directly manage them by checking and limiting the size of the gradients during training. This method is called Gradient Clipping. It works by capping the gradient values to a specific maximum (threshold), preventing them from getting too large and destabilizing the model.

4. Use Weight Regularization:

Another technique to combat exploding gradients is to control the size of the network's weights. If the weights grow too large, you can apply a penalty to the network's loss function, which discourages large weight values. This process is called Weight Regularization. You can apply either an L1 penalty (which uses the absolute value of weights) or an L2 penalty (which squares the weights), helping to keep the network stable during training.

These strategies can help you train recurrent neural networks more effectively, avoiding the instability caused by exploding gradients.

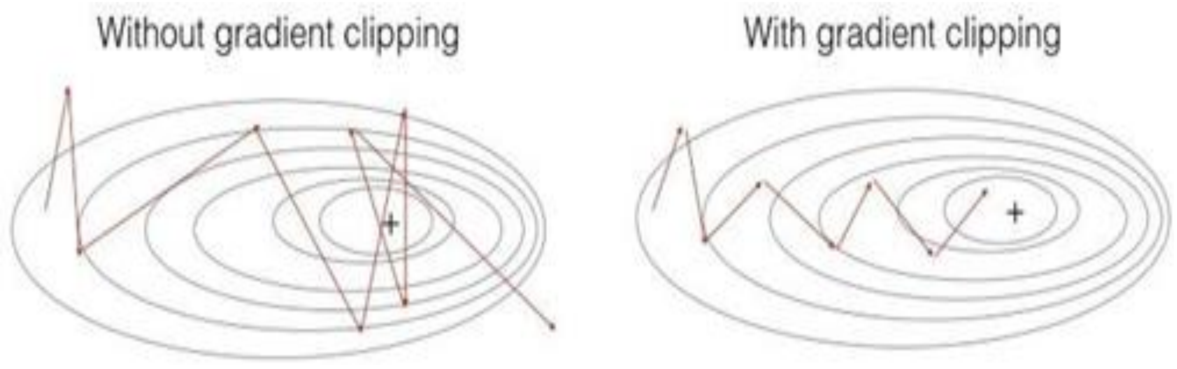


Fig. 6: *Gradients with and without gradient clipping*

F. Cross Entropy Loss:

Cross-entropy is a better way to measure error than Mean Squared Error (MSE) for classification tasks because classification involves making clear distinctions between categories, and cross-entropy penalizes incorrect guesses more effectively. MSE doesn't punish mistakes enough in classification, but it works well for regression tasks, where the predicted values are closer together.

Cross-entropy measures how different two probability distributions are for a set of possible outcomes. In classification tasks, it's useful because it helps adjust the model's predictions to better match the actual categories

Here, we are focusing on Cross-Entropy Loss (CE Loss), which is the main loss function used for classification. The CE Loss is defined as:

$$CE = -\sum_{i}^{C} t_i log(s_i)$$

Where ti and si are the groundtruth and the CNN score for each class i in C. As usually an activation function (Sigmoid/Softmax) is applied to the scores before the CE Loss computation. In the specific (and usual) case of Multi-Class classification the labels are one-hot, so only the positive class CpCp keeps its term in the loss. There is only one element of the Target vector tt which is not zero ti=tpti=tp. So discarding the elements of the summation which are zero due to target labels, we can write:

$$CE = -log\left(\frac{e^{s_p}}{\sum_{j}^{C} e^{s_j}}\right)$$

Where Sp is the CNN score for the positive class.

3. About The Dataset

The data set consists of names of people in different languages. We use people names in two languages namely English and Japanese names. All the names are in alphabetical order.

English dataset consists of 3668 names of which “Thistlethwaite” is longest with 15 characters and “Rose” is smallest with 4 characters. Japanese dataset contains 991 names of which “Mushanaokoji” being longest with 13 characters and “Abe” is smallest with 3 characters.

These names are preprocessed into distant characters and each character is mapped with a unique numerical value and is fed to the network. The output numerical values are again mapped with corresponding characters.

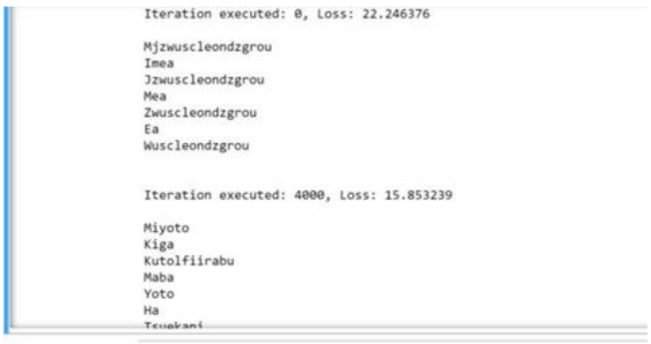
4. Analysis And Results

Learning Rate	Clipped Value	Text File	Cross-loss	Entropy
	5	English	15.736322	
0.01		Japanese	12.436578	
	5	English	16.575021	
0.001		Japanese	12.746988	
0.005	5	English	15.206230	
		Japanese	11.590827	
	10	English	15.731513	
0.01		Japanese	12.802580	
	10	English	16.586822	
0.001		Japanese	12.729625	
0.005	10	English	15.218383	
		Japanese	11.715368	

We tried for different values of the learning rate and clipped value and study the cross-entropy loss. We have used element wise clipping. It means that if the Clipped Value is N then every element in the gradient vector must lie between
+N and -N.

Initially it was generating random combination of characters but after 100,000 iterations it generated somewhat meaning combinations.

Initial:



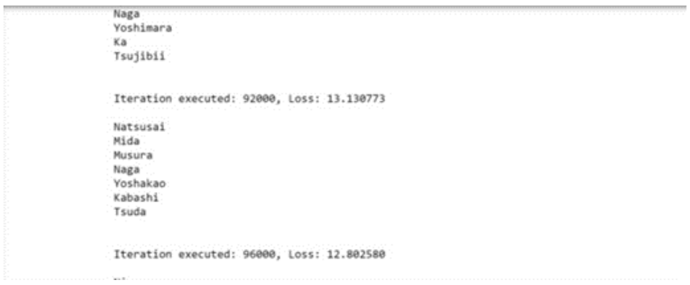
```
Iteration executed: 0, Loss: 22.246376

Mjzwuscleondzgrou
Imea
Jzwuscleondzgrou
Mea
Zwuscleondzgrou
Ea
Wuscleondzgrou


Iteration executed: 4000, Loss: 15.853239

Miyoto
Kiga
Kutolfiirabu
Maba
Yoto
Ha
Tsuekani
```

fter 100,000 iterations:



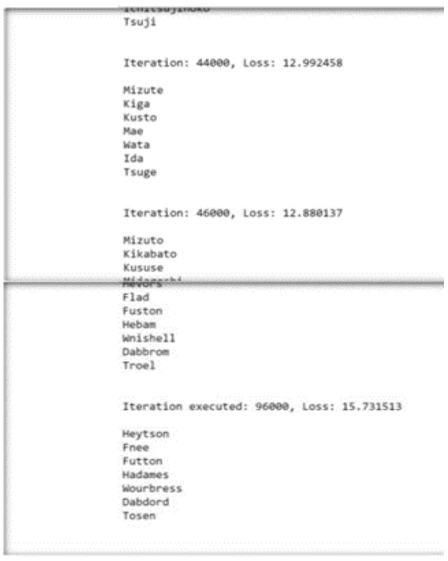
```
Naga
Yoshimara
Ka
Tsujibii


Iteration executed: 92000, Loss: 13.130773

Natsusai
Mida
Musura
Naga
Yoshakao
Kabashi
Tsuda


Iteration executed: 96000, Loss: 12.802580
```

The Japanese text data fared better than the English one in all the cases. It gave best result for learning rate 0.005 and Clipped value of 5. The English text data was more error prone. The loss was minimum when learning rate was 0.005 and Clipped value was 5. It has however generated interesting names that exist in real life but are not present in the dataset like Ida, Mae, Tosen.



```
Ichitsujinoko
Tsuji


Iteration: 44000, Loss: 12.992458

Mizute
Kiga
Kusto
Mae
Wata
Ida
Tsuge


Iteration: 46000, Loss: 12.880137

Mizuto
Kikabato
Kususe
Midagashi
Mevors
Flad
Fuston
Hebam
Wnishell
Dabbrom
Troel


Iteration executed: 96000, Loss: 15.731513

Heytson
Fnee
Futton
Hadames
Wourbress
Dabdord
Tosen
```

It produced interesting English names like Vough and Guston.

5 Conclusion and Future Work

In this project we have used Recurrent Neural Network for generating innovative names. We have also used gradient clipping to tackle the exploding gradient problem. Experimental results show that after many iterations our model was performing well.

In future work, we would like to implement GRU and LSTM and look forward to sentence generation that are grammatically correct.

References

[1]. https://machinelearningmastery.com/exploding-gradients-in-neural- networks/

[2]. https://medium.com/mlrecipies/recurrent-neural-networks-theory- f81d59c2add7

[3]. Generating Text with Recurrent Neural Networks by Ilya Sutskever, James Martens, Geoffrey Hinton, University of Toronto, CANADA.

[4]. Recurrent Neural Network for Text Classification with Multi- Task Learning by Pengfei Liu Xipeng Qiu, Xuanjing Huang, Shanghai Key, Fudan University, Shanghai, China

[5]. http://slazebni.cs.illinois.edu/spring17/lec20_rnn.pdf

[6]. https://gombru.github.io/2018/05/23/cross_entropy_loss/