Universiteit Leiden
Opleiding Informatica

Comparing Markov Chain
and Recurrent Neural Network Algorithms
for Text Generation

Name: Alenka Bavdaz

Date: 06/07/2016

1st supervisor: Dr. W. J. Kowalczyk
2nd supervisor: Dr. W. A. Kosters

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

Comparing Markov Chain and Recurrent Neural Network Algorithms for Text Generation

Alenka Bavdaz
Supervisor: Dr. W. J. Kowalczyk
Second reader: Dr. W. A. Kosters

Leiden University
6th of July, 2016

Abstract

In this paper, different models for text generation are compared. The performance of three different algorithms is evaluated: a Markov chain, a character-based Recurrent Neural Network (RNN) and a word-based RNN. They are evaluated under the same conditions and on the same data set. Three evaluation criteria have been chosen, based on which the algorithms are compared to each other, to select the algorithm most suited to this experimental setup. These three evaluation criteria are: the number of parameters of the model, the training time, and subjective evaluation of the generated output text. To test the RNN algorithms for overfitting, they will be trained on a randomly selected part of 80% of the data set, and tested on the remaining 20%. The training time of the Markov chain will be determined by the computation time it takes to create its database. During the comparison of results, we find that the character-based RNN outperforms both the Markov chain and the word-based RNN in the the number of parameters criterion, and ties with the word-based RNN on the last criterion. The Markov chain outperforms both RNNs in training time, followed by the character-based RNN. Concluding, the character-based RNN is most suited to this experimental setup.

Contents

1 Introduction

The problem that will be analyzed and discussed in this paper is text generation. Text generation is the process by which algorithms analyze a given data set of text and try to produce similar text. When algorithms are able to learn structures in a text, they can also be applied to learn from other structures. This ability can be useful in a variety of fields, such as handwriting and speech recognition. This will be discussed further in Section 2.

A simple model that can generate text is the Markov chain model. A Markov chain estimates probabilities of a word coming next based on the frequency of occurrence in the data set. This model can be extended by adding more depth: for example a depth-2 Markov chain calculates the frequency of words occurring after each pair of two consecutive words. This depth-2 Markov chain model will be used as a baseline for the Markov chain algorithm in this paper.

Recently, new methods have been developed that can be used for text generation. One of these is the Recurrent Neural Network (RNN) model. The RNN is a class of artificial neural networks, and is therefore a model inspired by biological neural networks. RNNs are known to be effective for text generation because they learn from structures over time. Usage of RNNs is increasing due to new algorithms for training the model. Although they are currently most commonly used in speech recognition [1], an increasing amount of papers suggest applying RNNs in several other fields such as music generation [2] and language understanding [3].

In this paper, the simple Markov chain will be compared to the RNN model for text generation. Moreover, experiments are also conducted between a character-based RNN and an altered version that is word-based instead of character-based, to see what effect this added complexity has on the result.

The goal of this paper is to compare different algorithms used in text generation, and to document the findings. The results will be assessed and com-

pared based on three criteria: the number of parameters of the model, the training time of the model, and subjective evaluation of the generated output text. The Markov chain and both RNNs are used to generate text based on a given data set with a specific format. The results of the experiments largely depend on the size of the data set, and therefore its selection will be elaborated in Subsection 5.4. The data set that is used in the experiments is the Bible [4].

This paper was written as part of the Computer Science bachelor programme at Leiden University, and has been supervised by W. J. Kowalczyk and W. A. Kosters.

2 Related Work

Recently, usage of RNNs for analyzing data structures has become more widespread. Therefore, new discoveries are made more frequently. Some of these discoveries will be discussed here. At first, RNNs seemed unfit for speech recognition as deep feed-forward networks showed better results. However Alex Graves, Abdel-rahman Mohamed and Geoffrey Hinton from the University of Toronto have shown that Deep Long Short-Term Memory Recurrent Neural Networks are effective in speech recognition [1], by recording the highest known score on the TIMIT phoneme recognition benchmark [5]. They compared RNNs varying among three dimensions: the training method, the number of hidden layer levels and the number of Long Short-Term Memory cells in each layer. They have found that the error rate (the phoneme error rate on the core test set) dropped significantly when using deep networks. In the field of language understanding, specifically opinion mining, Ozan Irsoy and Claire Cardie from Cornell University have found that deep narrow RNNs with multiple hidden layers outperform traditional shallow wide RNNs with a single hidden layer, while having the same number of parameters [6]. They have found that even smaller RNNs outperformed their other approach, Conditional Random Fields (CRFs), and therefore that powerful RNNs can be created in a compact manner.

3 Models

In this section, the two models used in this paper are explained.

3.1 Markov chains

The Markov chain associates a sequence of words (or any sequential data) with other words that can come directly after it in the data set. It can then be used to generate text based by selecting a next word based on probabilities calculated from each word's frequency of occurrence. A Markov chain is trained by creating a database, in which it writes all keys with all the possible words that come after it, anywhere in the training set, and their frequencies. This database is then used as a look-up table while generating text. A Markov chain is therefore very simple, since the probability of the next state depends only on the current state. This property is called the Markov property, or memorylessness. Markov chains can be of different depth. The depth of a Markov chain defines how long the key is. In this paper we will be using a Markov chain of depth $k = 2$. This means that the number of states is bigger, because we look at pairs of words.

The training of the algorithm works as follows: first the algorithm initializes a database *cache* to the empty array. It splits the training set at each word to write it as a list of words into a new variable *words*. Then it goes through the *words* list creating a database into the *cache* array that contains, for each unique set of k consecutive words, a list of all possible words that follow (from now on called following words). If a following word was already recorded, rather than appending it, its frequency is increased. Therefore, for each set of $k + 1$ consecutive words, going through the training set, the algorithm does the following:

> Step 1: Assign the first k words as a key.
> Step 2a: If the key is already in the cache, and the following word is as well, increase the frequency of the following word by 1.

Step 2b: If they key is already in the cache, but the following word is not: append the following word to the key's results in the cache and set its frequency to 1.
Step 2c: If the key is not in the cache, add the key to the cache and append the following word to its results setting its frequency to 1.
Step 3: Take the next key.
Repeat step 2 and 3 until arriving at the end of the training set.

The creation of this database can be seen as the training of the algorithm, and once the database is created, the algorithm is able to generate its own text. It generates the first two words by selecting at random two consecutive words from the training set. Then generation of new text can start. For each key the algorithm randomly selects one of the words associated with this key in its database based on its probability. Then the algorithm shifts forward so that the second part of the key becomes the first, the new generation becomes the second part, from which it generates another following word. By recording the frequency, instead of just appending a word again when it occurs multiple times, probabilities can be easily calculated.

When computing the database based on a training set, and then generating text, we must keep in mind that just because a word does not follow a certain key in the training set, this does not mean that it should not be possible for this to happen. To account for this, each possible word should have at least a small chance to follow each key. This has been implemented with the Laplace estimator [7], where the probability of each word is defined by:

$$\frac{freq + 1}{total\_words + unique\_words}$$

Therefore, even if a word's frequency is zero (it does not occur after that key in the training set), it will have some small probability to be chosen. This way, a set of words could be generated that is not registered in the database as a key. When such a key is encountered, the following word will be randomly selected from all unique words.

3.2 Recurrent Neural Networks

To understand Recurrent Neural Networks (RNNs), first the concept of Artificial Neural Networks (ANNs) should be explained. The easiest way to do so is with an example of a feed-forward neural network. First of all, the concept of ANNs is based on neurons in the brain that transmit signals to each other. The McCulloch–Pitts model perceptron (see Figure 1), is an over-simplification of real biological neurons, however it illustrates the general behavior of neurons and is used as the basis of ANN models.

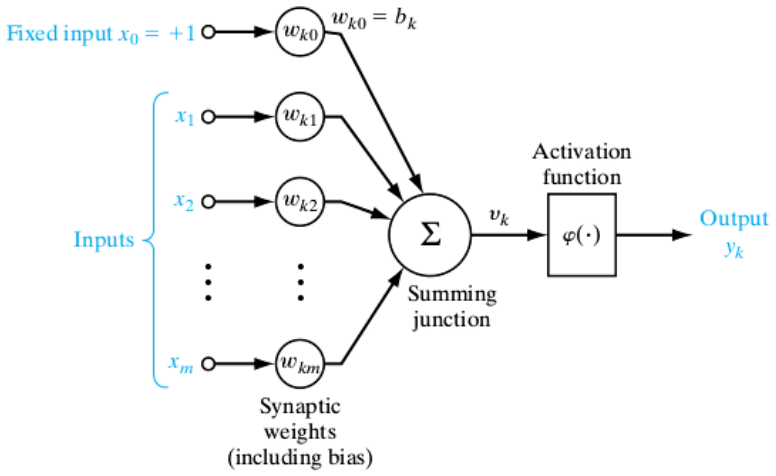

Figure 1: McCulloch-Pitts model perceptron [8]

In the ANN model, each input has a corresponding input node. The information obtained from all input nodes is combined to form an output in one or multiple output nodes. The way this is done in the McCulloch-Pitts model is the following: the values of the input nodes are multiplied by a weight, summed together and run through an activation function to produce the output. This makes up a linear perceptron. The weights of all the nodes are initially set to random numbers, and are updated after each cycle based on the output, to be able to produce a better output next time. This update is

the training process of the model, and serves to minimize the error (or loss) of the output. The loss is defined as the number of wrong answers in respect to the data set. The goal for such a model is to minimize this loss to be able to produce as many correct answers/good outputs as possible. For example, simple perceptrons can be used to model logic functions such as AND and OR. Here, we know what the correct answer should be, so we can check whether it is correct, and update the weights of each input node according to an update function, so that we make less mistakes next time. Usually, neural networks are trained on a training set, and tested on an independent test set. The reason for this is because the model can over-fit the training set, meaning it learns only exactly what is in the training set, and is not able to handle new data. We can test this by seeing how well it does (or how small the loss is) when applying its trained model to the test set, which should have never before seen data.

In this paper, models are used for text generation, which means that to check the predictions, they must be compared to the original text it is analyzing at that moment (be it the training set or test set). Cross-entropy loss is known to be most suited to algorithms performing classification, particularly neural networks [9]. This cross-entropy loss function will be explained later in this section.

There are multiple possible algorithms for updating the weights. In this paper the back-propagation (see [10]) algorithm will be used to determine in which direction the weights should be adjusted: if they should become bigger or smaller. First, the gradient of the error is calculated, which is defined as the difference between the network output and the expected output, over the change in weights. When updating the weights, we should not simply take this gradient as a measure, since the changes will be too drastic, and the model will take very long to find the optimized weight values. To soften the change, the gradient is multiplied by a set $learning\_rate$. This $learning\_rate$ therefore defines the rate at which the weights should be updated, according

to the error gradient. The weight update can therefore be expressed as:

$$W_{new} = W_{old} - learning\_rate \cdot \frac{\Delta E}{\Delta W_{old}}$$

Here, W_{new} and W_{old} contain the new and old weight vector values respectively, and ΔE is the error gradient. Therefore, if the gradient is positive, weights are decreased, and otherwise they are increased. This way the algorithm tries to get closer to the correct result in each iteration. As we see in Figure 1, there is a bias node with a fixed value. The indicated bias node's weight can be adjusted to shift the threshold location for the activation function. The activation function is what defines the output based on all the input nodes and their weights.

The linear perceptron described above is not very powerful. However, it can be used as a building block for making a multi-layer network. This is done by adding one or more layers of hidden nodes between the input and output layers. An example of a feed-forward network with one hidden layer and one output node (the output may contain many nodes) can be seen in Figure 2.

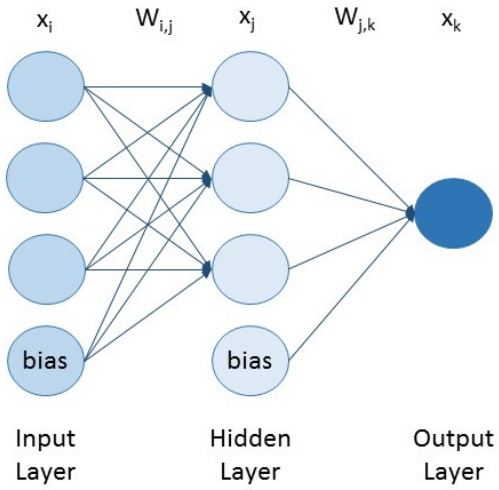

Figure 2: Simplified illustration of a feed-forward neural network with one hidden layer

The values for the weights of the hidden layers are calculated the same way as explained before, and their weights are updated with a similar formula. Each hidden layer also has its own bias node to shift the thresholds for the activation function. For feed-forward networks, the activation function is all that defines the hidden and output nodes, as they have no internal states. In contrast to feed-forward ANNs, RNNs are a class of ANNs that have a directed feedback connection and an internal hidden state. A simplified representation of an RNN can be seen in Figure 3.

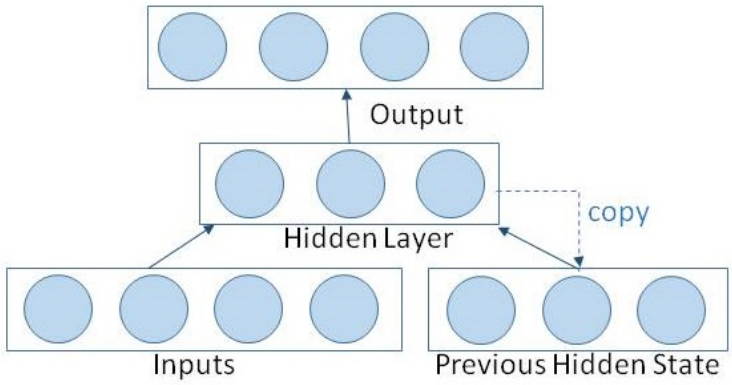

Figure 3: Simplified illustration of a recurrent neural network

The Previous Hidden State serves as the (short-term) memory of the model, and contains the contents of the previous Hidden Layer. The memory remembers the hidden state, which is used together with the input layer to serve as input for the hidden layer. This is done for a pre-defined number of times, in our algorithms called *seq_length*. The hidden layer is then used to compute the output, and is then memorized in the hidden state.

The directed feedback connection (copying the hidden layer to the hidden state) is a loop that is done for a predefined number of times before the output is calculated. As the loop may look confusing, we can unfold (or unroll) the RNN to show the loop as a finite amount of steps. A visual representation of this can be seen in Figure 4.

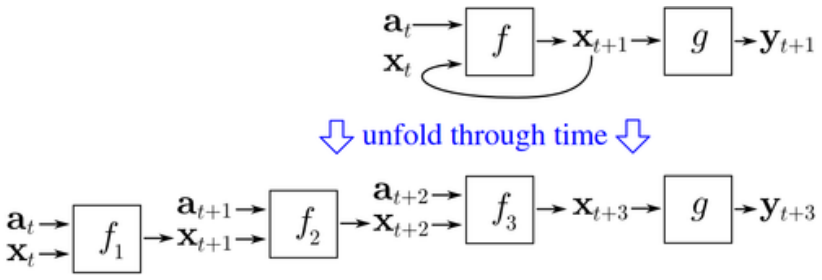

Figure 4: Visualization of unfolding a RNN [11]

Here, the a's represent the input, x's the hidden layer (and therefore hidden state), and y's the output. In our case, the input nodes represent all unique characters (or words) in the vocabulary of the training set. Therefore a word is represented by precisely one input node having the value 1, and all others 0.

The accuracy of the algorithm is calculated with a loss function, since the loss function is a measure for the error of a network's outputs with respect to its input. To measure the accuracy on sequences of inputs, the cross-entropy loss function is chosen. This calculated loss is used to adapt the model, by shifting its weights, in aim of reducing the loss to a minimum. From iteration 0 until the algorithm reaches a local minimum loss, this indicates an estimation of the training time of the algorithm, which will be used during the assessment of the algorithms.

The variable *loss* is used to represent the cross-entropy loss of each iteration, and is initialized to 0. To calculate this loss, the network is applied to each t in the inputs: X_t. For a record X_t, the output values form a vector of

non-negative values $ys[]$ that are normalized by:

$$ps[t] = \left(\frac{e^{ys[t]}}{\sum e^{ys[t]}} \right)$$

Since each output node represents a character (or word) in the vocabulary, these values are interpreted as probabilities. Then, the *loss* variable is updated by adding the term $-\ln(ps[t][targets[t]])$, where $targets[t]$ is 0 or 1, depending on the target value Y_t. The loss is calculated at each iteration, and it is reset to 0 before each calculation of loss.

Just as we use the *learning_rate* to soften the weight changes, we also a rate to equally soften the representation of loss. The representation of loss is expressed in the variable *smooth_loss*, and this is the value referred to when discussing loss in the results. While training on the training set, the training set *smooth_loss* is a smooth line that approaches a local minimum loss, and normalizes afterwards. The initial *smooth_loss* at iteration 0 is defined as:

$$smooth\_loss = -\log\left(\frac{1.0}{vocab\_size}\right) \cdot seq\_length$$

where *seq_length* is the amount of steps to unroll the RNN for. This defines the starting "maximum" loss. It is then updated after each calculation of *loss* with the following formula:

$$smooth\_loss = smooth\_loss \cdot 0.999 + loss \cdot 0.001$$

Here it shows that the rate of change should be chosen appropriately. If the rate is too small, it will take very long for the algorithm to reach a local minimum loss, as there is not much change between iterations. On the other hand, if it is too large, the model will change too drastically, and it may also take very long for the algorithm to reach a local minimum loss. With a satisfactory rate, the model will keep improving, until it reaches a local minimum loss and normalizes. Once the local minimum loss is reached and the value normalizes, the model is said to be trained.

As RNNs can model sequential data, they can be used for text generation. Text generation is done is by feeding a character (or word) into the model, which then generates the distribution of possible characters (or words) to come next, which then get fed into the model again. By this method, new text is generated. The RNN learns from the data set over time to try to generate output which resembles the original data set.

Two versions of RNNs are used in the experiments. First of all, we use Andrej Karpathy's [12] RNN code, which is character-based. Secondly, an altered version will be made to be word-based. The purpose of this is to test which of these two is more successful in which assessment category, and how this difference in the model will affect results.

4 Approach

In the interest of understanding, both the Markov chain and RNN models are studied thoroughly and experimented with. Shabda Raaj's Markov chain example [13] is studied to understand the Markov chain model. For this paper, a new Markov chain algorithm is written to include probabilities and a calculation of loss. To study and understand the RNN model, Karpathy's (character-based) RNN algorithm [12] is used. The character-based RNN algorithm by Karpathy will be modified to be word-based. Both RNNs will be modified to train on the training set for a set number of iterations, calculating the loss at each iteration, and then perform a loss calculation of the trained model on the test set. The algorithms being compared are therefore:

1. Markov chain
2. Character-based RNN (Karpathy [12])
3. Word-based RNN (Modification of Karpathy [12])

All three algorithms are run on the training set multiple times, and an average of their loss values is taken. Based on these initial results, the parameters of both RNN versions are optimized by comparing results on a trial and error basis. After the final selection of parameter values has been made, the algorithms are run on the test set, and their loss is recorded. The loss on the test set is compared to the smallest loss during the training. This way we can see whether the algorithms are over-fitting the training set or not. Then the number of parameters and the training time of each algorithm will be calculated. Finally, all algorithms will be assessed and compared to each other based on their results.

5 Experimental Setup

To be able to judge results in a fair way, all algorithms will use the same training set, test set, and will be run for the same amount of iterations on the same CPU system.

5.1 System

All algorithms will be run on the same computer system with the following specifications:

System type: 64-bit Operating System, x64 based processor
Processor: Intel(R) Core(TM) i5-2310 CPU @ 2.9 GHz
RAM: 8 GB
Operating System: Windows 10

5.2 Algorithms

In this subsection, the three algorithms will be briefly explained.

5.2.1 Markov chain

For the first of the three algorithms a depth-2 Markov chain is used. This algorithm is written in Python, and uses the standard math library and the Numpy library [14]. The Numpy library is used to select a word based on its probability.

5.2.2 Character-based RNN

Karpathy's character-based RNN algorithm [12] is used as the second algorithm. The algorithm is written in Python, and uses the Numpy library.

Weights are stored in matrices and updated with the use of matrix functions from this library.

5.2.3 Word-based RNN

The third algorithm is a word-based modification of Karpathy's RNN [12]. The first modification is the splitting of the data set, which now instead at each character, will happen at every space. The rest of the modifications are the follow-up changes to replace each reference to characters with words. This way, the enumeration of each unique input is adjusted according to the modification.

5.3 Data set

The training set which the algorithms learn from should satisfy a number of general requirements. To ensure sufficient (and more accurate) results, a large data set is needed. The data should also have a specific format, so that qualitative experiments are more easily conducted. Finally, since the run-time of the experiments is limited, it will help if the text in the data set has some structure, as association can be more easily made. A commonly used data set text analysis and generation is the entire content of Wikipedia. Based on these requirements, the following options have been selected: movie scripts and the Bible. Popular movie's scripts would be interesting and entertaining, however since obtaining the rights to them was not possible, the final choice is the Bible. The Bible is expected to most strongly separate the two RNN versions due to its specific sentence structure and uncommon word choices. The version of the Bible that is used is the King James Bible, and is 4.48MB in size [4]. In this version, each line begins with a definition of the verse, written as "Book Chapter:Verse". Because of this, we additionally have a "grammar rule" on which we can assess the algorithms' generated output text. This version of the Bible has 75 unique characters, and 4 556 377 characters in total. For the word-based algorithms, the data set is modified

to surround symbols with spaces, so that the algorithm will see them as separate words. It is also modified to have only lower case letters. This is done to decrease the amount of unique words and simplify the data set, otherwise computation for the word-based RNN could take too long. The data set modified for the word-based algorithms has 12 624 unique words and 998 824 words in total. A random 80% portion of the data set is used as the training set. The remaining 20% of the data set makes up the test set.

5.4 Assessment criteria

The three algorithms will be assessed on the following three criteria:

1. Number of parameters
2. Training time
3. Subjective quality of the generated text

The number of parameters is the number of values that need to be kept for the algorithm to compute an output. The loss factor can be roughly explained to be the difference between the generated text and the initial text, so our goal is to have a small loss. RNNs learn over time, which means that we will usually see an improvement in Smooth Loss. Therefore, if we plot the Smooth Loss against the number of iterations, we will see how many iterations it takes to get the smallest loss, and can therefore calculate how long the computation time for training that algorithm is. For the Markov chain, the loss factor is constant, as there are no parameters to be adjusted during the training process. Since it only has one loss value, the training time of the Markov chain cannot be defined the same way as the RNNs training time, so it will instead be defined by how long building its database takes in terms of computation time. For a Markov chain, training the model is the creation of the database. The loss values themselves cannot be compared between different algorithms. An example for this is that the log of a value very close to 0 will give a large "punishment", and this is more likely for the word-based RNN since there are many more unique words (than unique

characters).

Subjective quality is assessed based on readability, and will be tested by myself personally based on observing many output texts. First of all, the character-based RNN should be able to produce words to be readable. All algorithms will be evaluated on the extent of their application of the added grammar rule.

6 Results

In this section, a selection of results from all three algorithms will be discussed.

6.1 Markov chain

The Markov chain is a simple model, but has many parameters. A length 2 Markov chain can have up to n^3 parameters, with n being the amount of unique words: $nn = n^2$ for all possible keys, and $n^2 n = n^3$ if all possible words follow all possible keys. In our case, with a data set of 12 624 unique words, this would mean the number of parameters can be up to $12\,624^3 \approx 2 \cdot 10^{12}$. Realistically, however, not all possible pairs of words will occur in a text, and not all possible 3rd words will occur after every key. The actual number of parameters for this data set has been calculated to be approximately $6 \cdot 10^5$, counting only the keys and 3rd words that actually occur in the text. It takes approximately 5 seconds to compute the database, and about 5 seconds to compute a text generation of length 50 (words). Here follows a sample text generation:

> if ye can discern the noise of the fathers and the lord will cast out with great power , and by the sword , and his weapons in his own reward according to all that joab the captain of the syrians to hear thee .

We see in the output text of the word algorithms that symbols and punctuation are put between spaces, since they are counted as words. This generation seems almost as if it applies grammatical rules. However, the algorithm only occasionally produces such output. Another result is as follows:

> unto thee shooteth low plates few frequent asherites scant ossifrage rattleth lacketh hence severity eli avenge prospered lionesses pekah strake upper thirtyfold eternal heresy shemer chew

This text generation, on the other hand, shows no signs of grammar or any other format. This is a consequence of the simple approach used by the Markov chain algorithm.

6.2 Character-based RNN

Here, the results of the character-based RNN are discussed per evaluation criterion.

6.2.1 Parameters

The best set of parameter values found are:

$$hidden\_size = 100$$

$$seq\_length = 25$$

$$learning\_rate = 0.1$$

Here $hidden\_size$ defines the number of nodes in the hidden layer, and seq_length is the number of steps to unroll the RNN for. The $learning\_rate$ defines how much the calculated loss will be applied to modify the weights. To calculate the amount of parameters, we first define the weight arrays:

$$W_{xh} = hidden\_size \cdot vocab\_size$$

$$W_{hy} = vocab\_size \cdot hidden\_size$$

$$W_{hh} = hidden\_size \cdot hidden\_size$$

Here W_{xh} is the matrix of weights from the input layer to the hidden layer, W_{hy} the weights from the hidden layer to the output layer, and W_{hh} the weights from the (previous state) hidden layer to the hidden layer. Based on the model of the RNN (see Figure 3), the total amount of parameters will

then be:

$$W_{xh} + W_{hh} + W_{hy}$$

With the chosen parameter values, the character-based RNN therefore has $7500 + 10000 + 7500 \approx 2.5 \cdot 10^4$ parameters.

6.2.2 Training time and over-fitting

Computation time for 25 000 iterations takes approximately 20 minutes. The loss factor representation can be observed in Figure 5.

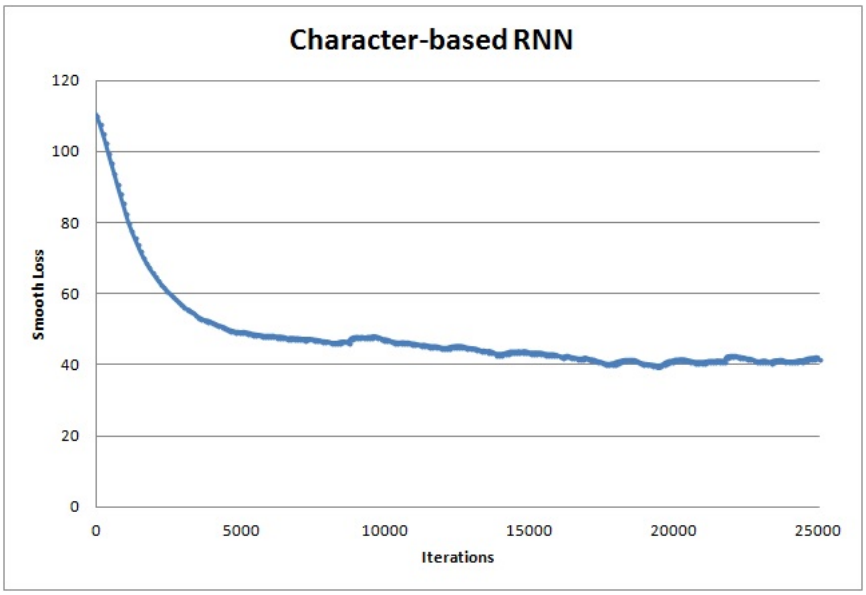

Figure 5: Character-based RNN: Smooth Loss over iterations

Here we see that the smallest loss is approximately 40. The average loss recorded on the test set is about 60. Since this is not a large difference from the smallest loss recorded on the training set, we can say that the algorithm is not over-fitting the training set. The loss reaches its minimum loss at approximately 20 000 iterations.

6.2.3 Subjective analysis of generated output

Although the loss normalizes, and therefore the loss does not get much smaller anymore, the output is still not that readable. This can be seen in an example text generation during iteration 25 000:

> reauts To the LORD mardall und the LORDp shill thes id in Mothe deowe sriend. Nuvebly silsed: tiom the unore At sper bealjaro shall deut hivon ynor sile's clooy sake thamst gom eid;

Despite not having a significant change in loss, an example text generation during iteration 100 000 shows a large improvement in readability:

> Proverbs 11:16 May: Proverbs 14:5 That no exall, shall there of that at those people with how they rested as is that foot of the cyst.

We see that the network is now able to construct words. However, word choice and grammar are still not good enough to give any meaning to the text. It did apply the structure of the Bible, meaning that each verse (line) starts with the book, chapter and verse number in the form of "Book Chapter:Verse".

6.3 Word-based RNN

Here, the results of the word-based RNN are discussed per evaluation criterion.

6.3.1 Parameters

For the word-based RNN the *vocab_size* is equal to the amount of unique words. Since there are more unique words than unique characters, there are more parameters. However, the total number of words is less than the total number of characters, and therefore the order of magnitude has changed.

The parameters should be in the same order of magnitude ([12]), therefore even though less adaptation of parameters was possible due to the long computation time, the following parameters were chosen:

$$hidden\_size = 150$$

$$seq\_length = 25$$

$$learning\_rate = 0.1$$

The number of parameters with these values, calculated in the same way as in Subsection 6.2, is $1\,893\,600 + 22\,500 + 1\,893\,600 \approx 3.8 \cdot 10^6$.

6.3.2 Training time and over-fitting

Still we can see that the number of parameters is very high, which results in the computation time for training for 10000 iterations taking approximately 2.5 hours. The loss factor representation can be observed in Figure 6.

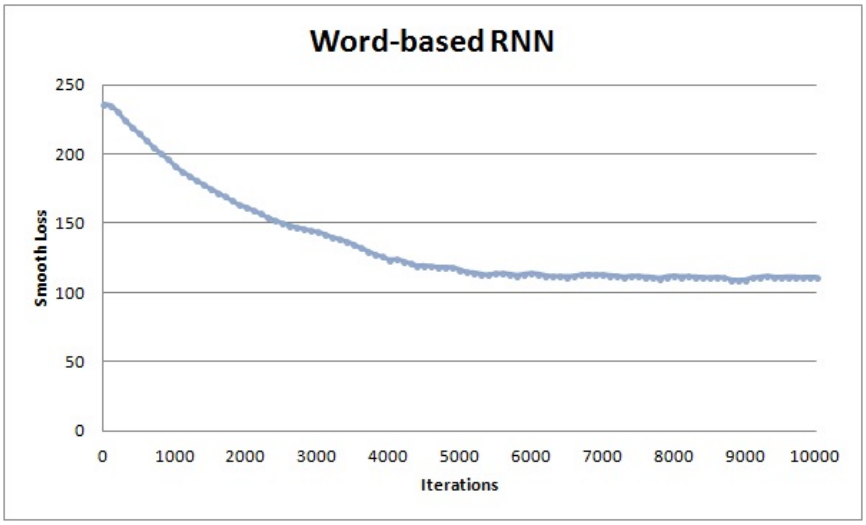

Figure 6: Word-based RNN: Smooth Loss over iterations

Figure 6 shows that the best recorded loss is approximately 110 at around iteration 9 000. The average loss recorded on the test set is around 135. Since this is not a large difference with the smallest loss on the training set, we can confirm that the algorithm is not over-fitting.

6.3.3 Subjective analysis of generated output

It is interesting to note that sometimes the algorithm is stuck trying to construct a sentence with the same starting word. This happened during a run around iteration 1 000:

> genesis trough , until house . genesis house . genesis that their their 33 : 11 take , and house . genesis so house . genesis the son land , husband , they shall them house . genesis house . genesis died ; all house . genesis savory and abraham . genesis house . genesis sister house . genesis my house .

This could be a result from a weight being set too high. Here follows an output after 9 000 iterations:

> genesis 31 : 30 he shall keep the land to not keep this land , and the departed into the congregation . the people was it upon the house is fallen , whom tree within you , then found and the priests of the third shall give you zeboiim : 1 joshua , i be not , and the seest which ye be seest fear only pass , and say unto the land of naphtali .

We can see that it is readable, since the words were already known to the algorithm, but no meaning can be deducted from the text. We can see that the word-based RNN also adapted the bible structure of starting a line with "Book Chapter:Verse", however it does not implement this on every new line.

6.4 Comparing results

In this subsection, the results of the three algorithms are compared for each criterion.

6.4.1 Number of parameters

For the Markov chain, the number of parameters is $6{\cdot}10^5$. This is significantly more than $2.5 \cdot 10^4$, which is the number of parameters the character-based RNN has. An even greater difference is the number of parameters of the character-based RNN compared to the word-based RNN, which has $3.8{*}10^6$. In this case, the character-based RNN outperforms the others, followed by the Markov chain.

6.4.2 Training time

Despite having the largest number of parameters, the computation time needed to train the Markov chain is the shortest, as it takes only 5 seconds to create its database. The character-based RNN reaches its minimum loss around iteration 20000, and takes about 20 minutes for 25000 iterations. This means that the character-based RNN needs approximately 16 minutes to train. The word-based RNN reaches its minimum loss around iteration 9000, and takes about 2.5 hours for 10000 iterations. This means that the word-based RNN needs more than 2 hours to train. Therefore, between the two RNNs, the character-based model requires less training time.

6.4.3 Subjective assessment of generated output text

We can clearly see that the character-based RNN takes more iterations than the word-based RNN to generate something readable. However, the total computation time to generate something readable is still shorter than the word-based RNN. Still quicker, the Markov chain takes only seconds of

computation time, and computes a readable output. The Markov chain does sometimes show correct usage of grammar, however other times produces incomprehensible phrases. Of course, the Markov chain, like the word-based RNN, also works with words, so readability is more easily achieved. The output texts from the RNNs, much like those mentioned above, are all similar in readability, provided that the character-based RNN runs for at least 100000 iterations. They all consist of proper words, and thus are somewhat readable. Some words seem to connect to form sentences, but a full correct sentence is never constructed. Therefore, none of the texts have meaning and thus are not understandable. This was to be expected considering the relative simplicity of the algorithms and their lack of intelligence, since they are not provided with any grammatical rules. However, unlike the Markov chain, both RNNs successfully adapt the structure of the data set (or added grammar rule), meaning the starting of each line with "Book Chapter:Verse". This shows that the RNNs capture deeper structures more than the Markov chain. They are therefore considered to outperform the Markov chain. The character-based RNN can be said to generate higher quality output text than word-based RNN, since it applies the added grammar rule at every line, rather than occasionally.

6.4.4 Final remarks

All in all, the character-based RNN seems most suited for this data set, as it uses the fewest parameters, and slightly outperforms the Markov chain (together with the word-based RNN) when it comes to subjective assessment of the quality of the output text. Seeing its improvement in output text even after reaching its minimum loss indicates that, given more computation time and iterations, it would most possibly be able to produce not only a readable but also understandable text.

7 Conclusion

In this paper an application of a Markov chain and recurrent neural networks to the task of text generation of a specific format has been explored. The experiments showed that the simple Markov chain is quicker and sometimes returns more correct output, however most of the time it does not perform well. The character-based RNN is able to produce readable text after a certain amount of iterations. Due to having less parameters and needing shorter computation time, the character-based RNN seems more suitable to this data set than the word-based RNN. Therefore, the character-based RNN outperforms both the Markov chain and the word-based RNNs on the Bible as a data set.

7.1 Future research

In further study, these experiments could be conducted on a text with less unique words. The word-based RNN may then outperform the character-based RNN as less unique words would have the effect of decreasing its computation time. Also, a larger data set could be used. If the data set were shorter, this could make it more difficult for the character-based RNN to learn even to the extent of creating words.

To be able to conduct more experiments, the RNN algorithms could be rewritten to make them quicker and more efficient, using languages and libraries such as Lua, Torch and Theano. Also, a Markov chain of a different depth could be used in comparison, to see how this affects results.

Besides the analysis of text for text generation, due to their ability to analyze sequences, RNNs could also be used in a variety of growing fields, as was mentioned in Section 2. More examples of such fields are handwriting recognition and speech recognition.

Bibliography

[1] A. Graves, A. Mohamed, and G. Hinton. *Speech Recognition With Deep Recurrent Neural Networks.* IEEE International Conference on Acoustics, Speech and Signal Processing, 2013.

[2] C. J. Chen and R. Miikkulainen. *Creating Melodies with Evolving Recurrent Neural Networks.* Proceedings of the International Joint Conference on Neural Networks, 2001.

[3] K. Yao, G. Zweig, M. Hwang, Y. Shi, and D. Yu. *Recurrent Neural Networks for Language Understanding.* Interspeech, 2013.

[4] King James Bible Authorized Version. `http://www.truth.info/download/bible.htm`. [Accessed 03/07/2013].

[5] J. Garofolo, L. Lamel, W. Fisher, J. Fiscus, D. Pallett, N. Dahlgren, and V. Zue. *TIMIT Acoustic-Phonetic Continuous Speech Corpus LDC93S1.* `https://catalog.ldc.upenn.edu/LDC93S1`. [Accessed 03/07/2013].

[6] O. Irsoy and C. Cardie. *Opinion Mining with Deep Recurrent Neural Networks.* Proceedings of the Conference on Empirical Methods in Natural Language Processing, 2014.

[7] D. De Cao and R. Basili. *Probability Estimation.* `http://www.uniroma2.it/didattica/WmIR/deposito/estimation_handout.pdf`. [Accessed 03/07/2013].

[8] Illustration of McCulloch-Pitts neuron. `https://themenwhostareatcodes.wordpress.com/2014/03/02/neural-networks-in-a-nutshell/`. [Accessed 03/07/2013].

[9] J. McCaffrey. *Why You Should Use Cross-Entropy Error Instead Of Classification Error Or Mean Squared Error For Neural Network Classifier Training.* `https://jamesmccaffrey.wordpress.com/2013/11/05/`. [Accessed 03/07/2013].

[10] R. Rojas. *Neural Networks A Systematic Introduction*. Springer-Verlag, Berlin, New-York, 1996.

[11] Illustration of unfolding a Recurrent Neural Network. `https://en.wikipedia.org/wiki/Backpropagation_through_time`. [Accessed 03/07/2013].

[12] A. Karpathy. *The Unreasonable Effectiveness of Recurrent Neural Networks*. `http://karpathy.github.io/2015/05/21/rnn-effectiveness/`. [Accessed 03/07/2013].

[13] S. Raaj. *Generating pseudo random text with Markov chains using Python*. `http://agiliq.com/blog/2009/06/generating-pseudo-random-text-with-markov-chains-u/`. [Accessed 03/07/2013].

[14] Scientific computing package for Python. `http://www.numpy.org/`. [Accessed 03/07/2013].