COPENHAGEN UNIVERSITY

MASTERS THESIS

Shall I compare thee to a generated poem?

Author:
Morten LANTOW

Supervisor:
Manex AGUIRREZABAL ZABALETA

A thesis submitted in fulfillment of the requirements for the degree of Cand.scient.it in IT & Cognition

in the

Centre for Language Technology
Department of Nordic Studies and Linguistics

November 1, 2018

COPENHAGEN UNIVERSITY

Abstract

Centre for Language Technology
Department of Nordic Studies and Linguistics

Cand.scient.it in IT & Cognition

Shall I compare thee to a generated poem?

by Morten LANTOW

In this thesis I present a series of computer generated poems. The poetry is generated with a generative adversarial neural network. The network is trained on the poetry written by John Milton titled Paradise Lost. I will here explore three different ways of preprocessing the poetry before training - one of these being a novel method. I then conduct a series of qualitative and quantitative analyses, along with an experiment striving to empirically asses the perceived quality of the differently prepossessed poetry. These are then compared to the original poems. The motivation for this work is to contribute, as best possible, to the problems surrounding the creation of a language driven artificial intelligence. This is done through the attempt of generating poetry capable of passing a Turing test. In the first chapter of the thesis I will focus on aspects of natural language processing and poetry, as well as present the different processing methods. In the second chapter I will attempt to pressent the theory describing the inner workings allowing neural networks to automatically generate poetry. I will strive to do so by dialectically working my way from the most basic constituents of a neural network towards more and more complex problems and system architectures, culminating in the presentation of cutting edge research on generative adversarial neural networks. In the third chapter I pressent the specifics of both the experiment and the poetry generation. In chapter four I present the results. And lastly I conclude on the findings.

Contents

List of Figures

Chapter 1

Introduction and motivation

I will here introduce the field of natural language processing, as well as touch upon the difficulties of generating a language model capable of capturing the meaning of a poem. I will then present some thought about the Turing test as well as the thought behind picking Milton's poetry as the data source lastly I will touch upon the concept of preprocessing

1.0.1 What is NLP

Natural Language Processing (NLP) is a scientific field concerned primarily with understanding and generating natural language by utilizing methods from computer science. We mean by *natural language* that a language is humanly evolved - a language so to say, as we speak and write it. The field of NLP has a broad range of applications in real life ranging from machine translation over information extraction and fact checking, to spam filtering, and automatic text summarization.

In the intersection between computational linguistics and machine learning we find the subfield of Natural Language Generation (NLG) the focus of which is (much as the name suggests) to produce computer systems capable of putting words together so they seem like natural language. NLG has a few real-world applications, such as automated service chat-bots for question answering, and other task demanding routine based text creation. But it is still an emerging research area, that hasn't seen as much utilization as many other areas of NLP. Research in NLG, however, brings a lot of diverse problems into focus, which again brings with it many new perspectives useable beyond just the generation of language. Among these we find fundamental problems in cognitive science, computer human interactions, linguistics, and artificial intelligence. These profound questions include how to best represent language in mathematics, or how to best asses the quality of a text, or how best to model the human use of language (Reiter & Dale, 2000). Add to this the prospects of a user-interface completely navigated through conversing with a computer, and the real world prospects of NLG seems like taken right out of a science fiction novel.

Another field closely related to NLG is the area of NLP called Natural Language Understanding (NLU) also known as Natural Language Interpretation (NLI). This is the field concerned with making computer systems understand the content of language, which is achieved by mapping human language into an internal computer representation. The goal is in some sense to capture the latent information of a given text. This latent information is also referred to as the semantic, the reference, or the meaning of a text. That this part of NLP is also known as NL-*Interpretation* is for me especially apt, as extracting the latent information of a text, in a very real sense *is* a hermeneutic endeavour, an aspect best captured when referred to as interpretation. As such the idea of creating a machine based frame of interpretation is what underlies the undertaking of NLU/NLI. This undertaking thus also suffers from the seemingly impregnable issues we meet in the field of hermeneutics. Bluntly put the

problem arises because the reference of a word is not a constant entity, neither across context or across time. As such the precise reference of a word has a way of avoiding clear distinct measurement. This is a broad and well known issue about language and something that is discussed in every aspect of its use from theology, to law, to marketing, to translation, and every other corner of the language tree [1].

Thus to truly model the meaning, or to truly capture an understanding, of a sentence seems all but impossible. Not only does one need nearly unlimited computation power, but one must also have access to unthinkable amounts of natural language from as diverse a set of contexts as one can imagine. Capturing the meaning of a poem, is thus a project of another size, a hermeneutic endeavor of sorts. This would be an undertaking of mathematically modeling one coherent understanding of what the words refer to, which might even necessitate an actual agent acting in the world. This is conjecture though, but one thing is certain, and that is that context is very important when modeling the meaning of words and sentences, and how they relate to one another - something that will be thoroughly shown in later sections. For instance Milton's poetry was heavily connected to the British political history [2], one out of near uncountable examples of the lack of context that the neural network had to work around by filling in the gaps when approximating its understanding. The structure, however, is capturable - in the sense that the pattern in it can more readily be mathematically approximated by a neural network. This means that structural patterns like the count of syllables (a pattern already present in the original text) or iambic meters can be model comparatively easily.

1.0.2 Can digital computers think in poetry?

One could in many ways attribute the origin of the NLP to the paper (Turing, 1950). In this he presents a game of imitation and interrogation now infamously known as the Turing test. This test has as its goal to assert weather or not we can count a computer as being a thinking entity - and the test is very much predicated on language. The imitation game consists of a sort of text based chat-room where three people are chatting. One is a man (A), the other is a woman (B), and the last one is an interrogator (of any gender). It is the interrogators task to determine the correct gender of the other two. It is A's goal to fool the interrogator into making the incorrect identification, and it is B's objective to help the interrogator into making the correct identification. He then reframes the question to be "What will happen when a machine takes the part of A in this game?" - as such it is now only inadvertently about gender, and instead about seeing if there is a difference in judgments when A is a machine instead of a man.

In the same paper Turing made some example questions to show how mastering language alone would be a criterion with some advantages - the first of which very noticeably (at least to the author of this text) was about poetry:

> Q: Please write me a sonnet on the subject of the Forth Bridge.
> A : Count me out on this one. I never could write poetry.
>
> (Turing, 1950)

[1] In linguistics this feature is often talked about as semantics (Steinberg, 1971). In philosophy this relationship between the words themselves and what they refer to is known under a plurality of names including the study of meaning and reference (Putnam, 1974) or as in what has come to be known as semiotics the signifier and signified (De Saussure, 2011; Peirce, 1868)

[2] He was secretary for foreign tongues to the council of state from 1649 until 1659 under Oliver Cromwell's new republic (Maltzahn, 1994)

In its essence the idea of the Turing test is that it defines intelligence in such a way that we should consider a computer intelligent if it can carry a conversation with a human without the human realizing that they are talking to a computer. And as we saw, Turing himself perceived the need for poetry generation as part of solving intelligence. As such the project here unfolded is a project attempting to solve a small part of intelligence, by delving into the problems whose solutions allow a computer to write poetry - something that in many ways can be seen as the most human use of language.

1.0.3 Why Milton?

As previously mentioned it is tantamount to impossible to capture a meaningful semantic without access to excessive amounts of data and computational resources (and even then it seems far from trivial with the current technology). As such I choose to focus primarily on showing how it is possible to capture the structure of a text with a neural network. This demanded that the text had a clear and quantifiable structure allowing the generated results to be compared to the source text. Choosing poetry as the source text also allowed the semantic to have more variation and still be considered meaningful. Luckily a lot of poetry has a clear structure to be found. One common way that poetry has structure is in their rhyming schemes. The generation of rhymes however relies heavily on the pronunciation and thus the phonology of the words. A lot of work has been done in the field of generative phonology (Kenstowicz & Kisseberth, 2014), but for the sake of keeping the scope of the project manageable I chose instead to focus on another commonly occurring structure found in poems - namely the syllable counts and their stress patterns. It was because of these considerations I decided to work with Milton's Paradise Lost, as this has both a very clear syllabic structure and a poetic semantic.

On blank verse and poetic meters

Paradise Lost is written in a poetry style known as *blank verse*, which is a very commonly occurring poetry style in the English language. Blank verse has no clear rhyming patter, but instead has a clear stress pattern and syllabic count per line. This pattern, most frequently, takes the form of the poetic meter known as *iambic pentameter*. Milton did not dogmatically stick to this pattern though. He bent these poetic structures where he found it necessary. These divergences from the form, however, only occurs as exceptions to the rule in Paradise lost (Harvey, 1996).

Iambic pentameters consists of two distinct patterns both conveniently referred to in the name. Firstly the iambic part denotes that the line are made up of *iambs* - which is a foot (the basic rhythmic unit making up the meter) in poetic meter consisting of two syllables, with the first one being unstressed and the second being stressed. This generates a rhythm in the pronunciation like a man walking with a limb "da-DUM da-DUM da-DUM". Secondly the *pentameter* part refers to the count of five iambs per line. As such in a perfect Iambic pentameter there will be ten syllables with every other being stressed. An example of this would be the famous line from Shakespeare's Sonet 18 "Shall I compare thee to a summer's day?". A way of writing the stress pattern is with an $-$ denoting the unstressed syllable and a / denoting the stressed syllable. The above phrase can thus be written as " - / - / - / - / - /" or with the stressed pattern in bold as in:

Shall **I** com**pare** thee **to** a **sum**mer's **day**?

Other poetic meters used to some extend in Paradise Lost include the *trochee* which is the reverse of the iamb in that it has first a stressed and then an unstressed syllable as in "/ -". We also find the *trochaic* and the *spondaic* feet as being two unstressed and two stressed syllables respectively as in "//" and "–". More rarely we also find trisyllabic feet as the *dactyl* pattern consisting of one stressed syllable followed by two unstressed as in "/–". As well as its opposite the *anapestic* pattern consisting of two unstressed syllables followed by a stressed syllable as in "–/". The vast majority of the lines in Paradise Lost, however, contains Iambs (Harvey, 1996).

Previous attempts have been made at quantifying the stress patterns of poetry using probabilistic models on the text alone - including work on Paradise Lost Hirjee, 2010. I will however forego the use of these somewhat complicated methods here[3] and instead focus on quantifying the syllabic count per line.

Other attempts at measuring the syllabic count of Milton's poetry have been made, one of the more noticeable attempts was made in the 1918 paper entitled "An objective study of the syllabic quantity in English verse" (Verse & Snell, 1918). Here the author conducted a very thorough investigation into the iambic stress patterns and syllabic counts of 25 lines of Paradise Lost - all of which, in addition to the author's analysis, also was subjected to being read out loud by three other scholars from similar fields. These were asked to read out the poetry into a contraption devised by a Professor John F. Shepard so as to make the syllabic count measurable. The contraption consisted of *"various tambours covered with mica and rubber and mounted with pointers which record the vibrations and the outflow of air during speech"* - this in turn would remove *"[...] the soot from a revolving band of smoked paper"* thus noting down *"[...] the various vibrations and curves which represent speech"*(Verse & Snell, 1918)[4]. I will here also be counting the syllables of both the generated and the original text in an effort to compare these. For simplicity's sake, I will forego any use of mica covered tambours, and instead use an algorithm to count and compare the syllables per line in the generated and real text.

The generated poetry will thus be judged in three different ways. Firstly by a qualitative look at the stress patterns of the generated poetry compared to the source text. Secondly the syllabic count per line of the generated and the original text will be compared. And lastly an experiment is conducted asking participants to both asses how easily the content of the poem allows for a poetic interpretation as well as being asked to recognize the real from the generated poetry.

1.0.4 The syllabic algorithm

The algorithm I here will use was first presented in Franklin Mark Liang's PhD dissertation (F. M. Liang, 1983). It has since become the standard algorithm used for hyphenation in TeX based typesetting systems (eg. LaTeX). Hyphenations are not a precise science so there will always be exceptions to the rules, as such the aim of Liang's algorithm was to minimize a very extensive dictionary containing syllabic split words into a series of patterns, such that the number of exceptions needed was minimized while the accuracy was maximized. This was achieved by a language independent approach, creating an algorithm applicable to all languages with a word

[3] Partly for the sake of keeping the scope in check, but not least because we merely need a metric for assessing how well the generative model has approximated Milton's writing style - and a complicated metric tends to build in biases

[4] The same contraption was also used in (Snell, 1918) where the author purportedly expands more thoroughly upon its design. Most unsatisfyingly, however, I haven't been able to find a readable copy of this paper

syllable dictionary. The approach therefore consists of two components, the dictionary of exceptions[5] written manually and the learned patterns of syllables reducing the dictionary to a manageable size (F. Liang & Breitenlohner, 1991). It is important to note that this method is very far from perfect. However since the vocabulary is the same on both the generated and the original text the errors will be uniform across the samples. A possible bias could have been introduced if a wrongly hyphenated word occurred with abnormal frequency in one and not the rest of the compared texts - but no such words were found.

1.0.5 Preprocessing

Preprocessing is a very crucial step when dealing with language. As we have seen in previous sections, language has to be processed before it can be worked with mathematically. But even before this it has to be cleaned and pruned. The core idea of preprocessing is to remove noise from the data so as to enhance the latent pattern in it - this should make it easier to find for the neural network. After all "garbage in garbage out". What is considered noise and what is not depends on what pattern we want the system to approximate. Sometimes, however, preprocessing can have the opposite effect and instead end up adding more noise to the data. Another possible pitfall is to build in presuppositions and biases to the data, thus guiding the network towards a given assumption, that we ourselves have before we train it. As such preprocessing should be done with some care. Preprocessing can be done in many ways and it is thus important to think about what implications this might have for the pattern that the system finds. Punctuation is generally good to remove, but if one for some reason wants the system to model the use of punctuations they of course have to be left in. In this case I removed the punctuation, as if they were seen as tokens it would interfere with the syllabic count. In some cases it can be good to expand contractions from "isn't" to "is not" so as not to mistake these as having two different meanings. But in other cases we might want to keep them as two different cases for instance if we are trying to classify the writing style of the text. For classification we might want to remove *stop-words* to amplify the other words with stronger semantic. But when dealing with generation of text it is very important that the machine knows how to use the most frequently occurring words. Also *stemming* is useful sometimes as it amplifies the semantic of a text by making a lot of words the same, however when generating text we want grammatical patterns such as tense and plurality to be pressent. One might also want to transform all letters into lowercase, so the words for "cat" and "Cat" are not seen as two different tokens by the system. However, in the case of Milton's 17th century English, the uppercase letters actually had the added semantic meaning of telling the reader that the word was a noun and not a verb. This makes clear the distinction between "to object" and "an Object" or "to present" and "a Present". This is very important for the pronunciation and hence the stress pattern, which is why I choose not to change the words into lowercase. But doing so I oversaw that also the starting words of the lines have uppercase letter. A possible improvement to my initial preprocessing would thus be to lowercase all the beginning words but not the rest. One thing often done with sequence models is to pad the lines with semantically empty tokens, such that they all have the same length as the longest line. This is done for practical purposes - as will be touched upon later - and is basically a way to allow the corpus to be presented as a matrix, which makes it possible to feed it into the system.

[5] A list of exceptions can be found here

The three preprocessing methods

As such preprocessing can be used to amplify or reduce the latent patterns in the text. In the case of generating blank verse in iambic pentameter, we especially want the system to pick up on the syllabic count. An obvious way of doing so would be to preprocess the text by splitting it into syllables before feeding it into the network. It is, however, not clear that we can do so without sacrificing some clarity in the meaning of the text. The **first** preprocessing method to test will thus be that of splitting the words into syllables. This is measured against the **second** method of not amplifying any syllabic pattern and letting the algorithm attempt to find this by itself. And the **third** and novel method will be that of *syllabic padding* or *double padding*.

The idea behind double padding is to help the algorithm find the correct number of syllables per lines faster by adding information about the missing tokens. This is done by adding additional padding to the sequence that we feed in. If a line has ten syllables but only five words, we would thus pad the line with five extra tokens. The idea can be seen in figure 1.1

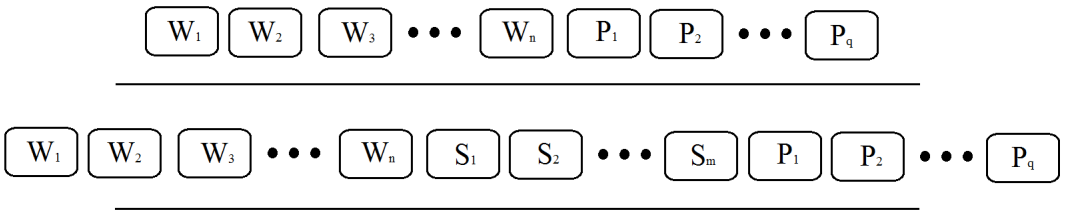

FIGURE 1.1: The top line shows conventional padding. The bottom line shows double padding

Where W_i is the $i'th$ word in the sequence, P_i is the padding, and S_i is the syllabic padding. I here added the syllabic padding at the end of the sentence, but in future work it would be interesting to see if adding the padding immediately after the word would enhance the performance further. The results are presented in chapter 4. After preprocessing the data was sent through a neural network. This will be expanded on in the following chapters

Chapter 2

Background

This chapter aims at explaining the essentials of machine learning based NLP. It does so with varying degrees of resolution, starting with the basics - which is intended as a sort of primer - as it works its way towards more complicated problems, culminating in resent work on generative adversarial neural networks and how these can be utilized for language generation

2.1 The basics of neural networks

In the following section I will start by outlaying the fundamentals of neural networks. I will here go from neurons to perceptions, through to loss functions and backpropagation.

2.1.1 Introduction to Neurons, Single and multilayer perceptrons

Neural networks in their most primitive form originated as a collaboration between a neurologist and a mathematician in the paper (McCulloch & Pitts, 1943). Together they proposed a mathematical model mapping how the basic unit of the brain - the neuron - might work. The paper (Hebb, 1949) brought the idea further by proposing that neural pathways become stronger every time they fire, especially when they fire between neurons that fire at the same time. This is where the famous phrase: 'neurons that fire together wire together' originated from. The mathematically simulated neurons became further developed in the paper (Rosenblatt, 1957) where they were also given the name perceptrons. The mathematical metaphor of the perceptron as a neuron builds on the idea of a neuron being in one of two discrete states either firing or not. In the simplified world of mathematics, we assume that the timing of these firings (action potential spikes) doesn't matter and that only the frequency of the firing is important for carrying the information (Cios, 2018).

The Neuron

In the simplest understanding of a neuron, the signal carrying part of the cell - known as dendrites - carry a signal from other neurons near the center of the cell (in the axon hillock) where all this potential energy - known as the action potential - is gathered. A model can be seen of this in figure 2.1

If then this potential is above some threshold, the neuron itself fires, thus sending a energy spike along its axon to be gathered by the dendrites of other neurons and potentially activating them. The link between the axon terminals and the dendrites, and thus the organ responsible for carrying over the signal, is known as a synapse.

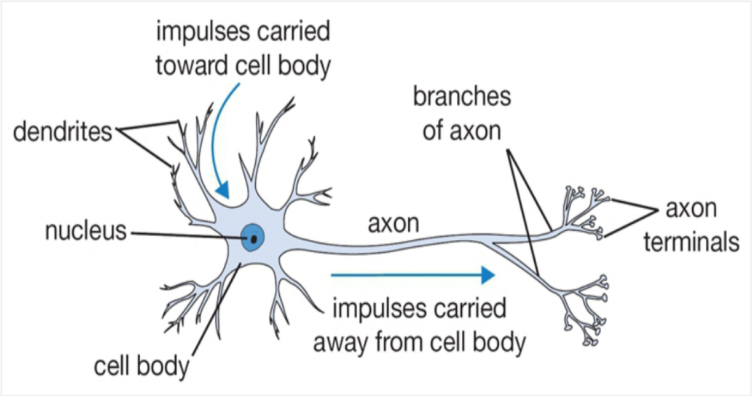

FIGURE 2.1: Simplistic model of a biological neuron

http://biology.reachingfordreams.com/biology/nervous-system/8-structure-of-nervous-system

The perceptron

Basically the perceptron is a function attempting to linearly separate labeled data points into two classes. However not all sets of data are linearly separable, and not all datasets only have two classes to distinguish, as such the lone standing perceptron is only useful in a limited amount of cases. But, datasets with multiple classes can be classified by having multiple perceptrons stacked in a layer, such that every perceptron receives the same input and each individual node is responsible for classifying one specific class in a binary member not-member fashion. Furthermore these layers can be stacked again creating what is known as multilayer perceptrons or with a more modern term artificial neural networks or simply neural nets.

The mathematical metaphor as described so far can be visualized in figure 2.2

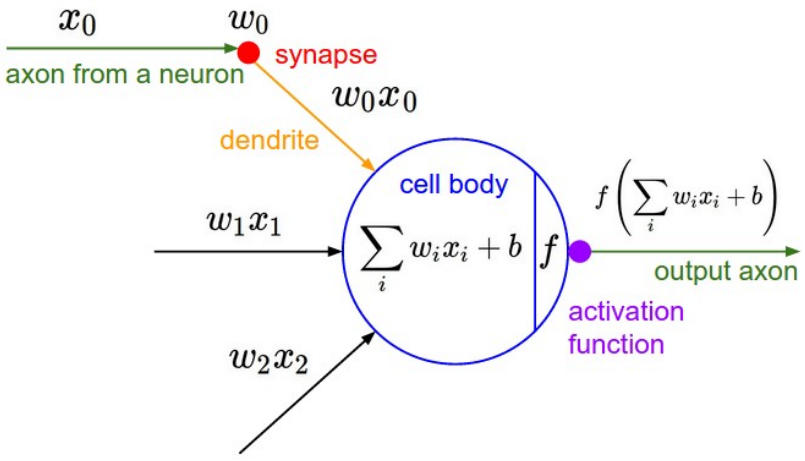

FIGURE 2.2: Simplistic mathematical model of a biological neuron

http://cs231n.github.io/

In the math of the perceptron the idea of 'fire together wire together' - or more specifically put: the representation of the learnable synaptic strength - is captured by the weights of the system. These are also learnable and control how much influence the activation of one neuron has on another neuron. This is achieved by determining how strong of an action potential is passed on from one neuron to the next. In

biological terms they talk of excitatory and inhibitory synapses, these are what we in the neural metaphor of machine learning instead call positive and negative weights.

Since the inception of the perceptron a lot have happened, especially in the availability of computational power now allowing for previously unimaginable feats - such as a computer writing poetry (Manurung, Ritchie, & Thompson, 2000). But, the basic unit of a neural network has in its essence stayed the same.

In a classifying neural network, there are thus many different layers all working together to model a function mapping between an input and its given labels.

The Layers

There are classically three different layers in a neural network, the input layer, the hidden layers, and the output layer. The input layer has as its purpose to feed in the data to the hidden layers. The input layers should thus have the same dimensionality as the data, meaning that each value or feature in the data has its own neuron attached (Goodfellow, Bengio, Courville, & Bengio, 2016). The hidden layers are where the bulk of the modeling happens, the more hidden layers are taken in use, the more complex a function our neural network will be able to model. However because of the added complexity, too many layers leads to overfitting, which in short is when the function maps onto the specific instances of the data, instead of finding an underlying pattern. This will mean that the model does not generalize well to unseen data. Lastly we have the output layer, where (in a classifying neural network) the prediction of which class the input data belongs to is given. The output in such a network thus typically has a neuron for each possible class, these neurons are connected to the last hidden layer. In figure 2.3 we see a three layer neural network with one hidden layer consisting of three nodes, an input layer with three nodes, and a binary output consisting of one node either firing or not.

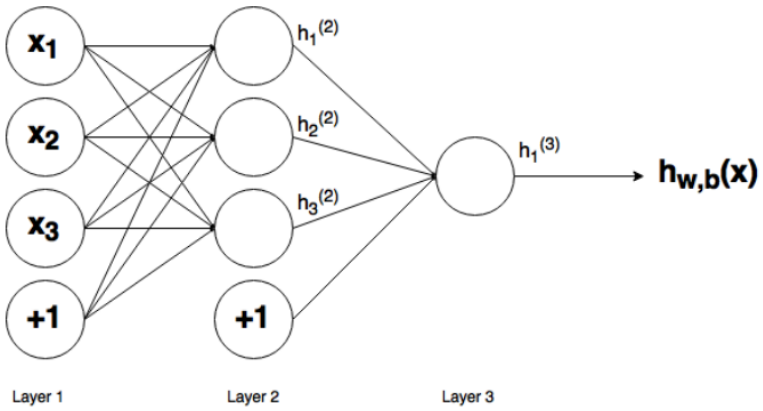

FIGURE 2.3: Simple three layer neural network

An Example

So, In the mathematics of neural networks the synapse is called a weight. This weight is a number intended to strengthen or weaken the output signal arriving from the "axon" of a previous neuron (or in case of the input neurons: a signal from the data itself). As previously mentioned the output of a neuron is binary, meaning it is either 0: not firing - or 1: firing. As such the weight multiplying the output signal from one neuron - let's call the neuron $h_1^{(2)}$ - before it is sent to be summed

up in the next neuron - let's call that neuron $h_1^{(3)}$ - signifies the importance of $h_1^{(2)}$ in determining whether or not $h_1^{(3)}$ is activated (see figure 2.3).

To make this perfectly clear we can imagine an image classifier determining if a picture is of a cat or not. Let's assume $h_1^{(3)}$ to be the output layer, and that the activation of $h_1^{(3)}$ means the neural net believes the input image to be of a cat. $h_1^{(2)}$ could then be a neuron looking for pointy ears, or some such. If $h_1^{(2)}$ fires it then means that the network believes the image to contain pointy ears. How significant the finding of a high level feature such as a pointy ear is to the output layer, is then determined by the weight (or "synapse") connecting $h_1^{(2)}$ and $h_1^{(3)}$. A high positive number means it is a very important find in determining that it is indeed an image of a cat, where a high negative number means it is very important in determining it is not a cat - and a zero means it does no difference either way. Many things have pointy ears, so the output neurons need to be connected to other neurons all looking for different things. The sum of all these weight-multiplied-signals then determines whether or not the perceptron is activated

Let us again take a look at figure 2.3. Here we see that all three input nodes are connected to three hidden nodes in the next layer. As such we have 3×3 weights (or "synapses") connecting the input and hidden layer. These weights can be written as a matrix using linear algebra notation as seen in equation 2.1

$$W^{(1)} = \begin{pmatrix} w_{11}^{(1)} & w_{12}^{(1)} & w_{13}^{(1)} \\ w_{21}^{(1)} & w_{22}^{(1)} & w_{23}^{(1)} \\ w_{31}^{(1)} & w_{32}^{(1)} & w_{33}^{(1)} \end{pmatrix} \tag{2.1}$$

Where: The $^{(1)}$ denotes that the weights are from the first layer, and the w_{ij} denotes that the weight w connects the i'th neuron in the first layer to the j'th neuron in the second layer. Similarly we can write the weights connecting the hidden layer to the output layer as seen in equation 2.2

$$W^{(2)} = \begin{pmatrix} w_{11}^{(2)} & w_{12}^{(2)} & w_{13}^{(2)} \end{pmatrix} \tag{2.2}$$

We now have all the weights of the network defined and accounted for. As mentioned earlier these weights are then multiplied by either the output of the previous layer, or in case of the first hidden layer by the input, whereupon the bias is added. Let us call the accumulated "action potential" of perceptron i in layer l for $z_i^{(l)}$ these are also known as the *logits*. With this we can now describe the "action potential" of a neuron in the next layer in terms of the logits of a previous layer as shown in 2.3

$$z_i^{(l+1)} = \sum_{j=1}^{n} w_{ji}^{(l)} x_i + b_i^{(l)} \tag{2.3}$$

Where: n is the number of nodes in layer l. This can be further simplified by using linear algebra notations such that the elements in equation 2.3 now become matrices and vectors and the operations become dot-products as seen in equation 2.4

$$z^{(l+1)} = w^{(l)T} x^{(l)} + b^{(l)} \tag{2.4}$$

The Activation Function

A last thing we need to explain is how the "synapse" decides whether or not the "action potential" is enough to activate and fire a signal to the next neuron - a decision mechanism in mathematics known as an activation function.

To follow the neural metaphor we can say that the activation function reflects the firing rate of the neuron, which represents the frequency of the action potential spikes along the axon.

The activation function thus has as a task to decide whether or not the summed up weighted potential of the neuron is sufficient for the neuron to be activated. This then becomes part of the input x for the neurons in the next layer $l+1$

$$f^{(l)}\left(w^{(l)T}x^{(l)} + b^{(l)}\right) = x_i^{l+1} \tag{2.5}$$

Let's make a quick jump back to the example of our logit described in equation 2.4. To see if the neuron fires or not we can now apply our activation function:

$$f^{(1)}\left(z^{(l+1)}\right) = f^{(1)}\left(w^{(l)T}x^{(l)} + b^{(l)}\right) \tag{2.6}$$

This activation can be done in many ways, however a thing to note is that if we choose a linear function as our activation function, this linear property is inherited all throughout the neural network and the network will only be able to approximate linear functions (Goodfellow et al., 2016).

2.1.2 On loss functions

I have so far only talked about how the data propagates through the network from input over hidden layers to output, with each layer being connected to the next. But in this story the most ingenious part of the neural network is left out - namely how the neural network learns. If humans had to manually tune every weight of a network - ie. manually decide how important pointy ears are in determining a cat photo from the rest - then neural networks would be impossible to use in practice as they would take ages to fine-tune. The ingenious part of the neural net comes with its ability to self correct and learn via a process called backpropagation. Backpropagation is from a top down view a mechanism of tuning the weights of a neural net - such that the network becomes better and better at its given task- but doing so in small incremental steps (Hecht-Nielsen, 1992) - I will expand on this in the next section. We talk about two different kinds of learning - supervised and unsupervised. In the case of supervised learning, we have annotated our data, such that what the neural network should learn is clearly defined. This is the case in classification algorithms - where the goal to be learned is a class from a set of distinct classes. If we do not have such clear definitions however, and instead we want the algorithm to learn complex and sometimes ill defined concepts - like understanding language or defining its own classes - often creative solutions working around this lack of clear categories must be found. In supervised learning though, the task needs to be given before the neural network can start to improve. More to the point, to guide the weights of the neural network towards improvement, backpropagation needs a heuristic defining the success criteria for the function that the neural networks is approximating. As such: to begin scratching into the idea of backpropagation we must firstly go through this idea of formally defining the success of a network.

When training a neural network, it is necessary to define a criteria for success (or in most cases, more accurately, a criteria for failure) that the neural net can use

as guidelines for its learning. When this criteria is formally defined we call it a loss function, and it dictates in which direction the weights of the network are changed during the backpropagation steps. In essence the loss function or the objective function is used to calculate the error between the predicted value and the correct value. This can formally be defined as $L(\hat{y}, y)$ where L is the loss function, $\hat{y}$ is the prediction from the neural net, and y is the true expected output. In other words the prediction of the neural net is fed into the loss function along with what we actually want the neural net to predict, whereupon the loss function spits out an error score, that we then try to minimize when we update the weights of the network. (Goldberg, 2016)

An example

Now, the loss function can be calculated in many different ways depending on the task, and even when the task is the same or similar, many different loss functions can be taken into use creating very different results.

Let us, as a relevant example, say that we have a simple poem about animals, and we want to train a generative model to create similar poems. To understand the loss function, we have to open up the hood of the machine during the learning phase and take a look at its inner workings. After zooming in on one iteration we find the algorithm at work trying to approximate the pattern found in the mathematical representations of the poems we feed it. Here we see that the original sentence from the animal poems, namely the sentence: "I like dogs not cats", has instead been predicted by our neural network to be "I like dogs not balloons". The algorithm is thus close, only one word away in fact, but it still has a ways to learn.

Now, the question arises how much of an error should we deem the change from "cats" to "balloons" to be? A human might think this through with many different aspects in mind. One could think that "balloons" and "cats" are similar in that they are both plural nouns, and thus "balloons" are grammatically correct, but lack the same semantic reference - where one refers to a thing the other refers to an animal. So intuitively we could say that "balloons" is a better word to mistake "cats" with than "running" is for instance, but probably worse than "kittens" or "kitties" - but the question arises: how can we make measurable the error of a word, and thus guide a neural net in the proper direction. Is one word ten better? maybe a million better? How do we make the understanding of language quantifiable?

This is the primary question of the loss function, and it arises when we try to formalize the aspects of language-understanding into a formula that can guide the neural network into learning a model of the language. When wanting to define a loss function, it is thus very important to make clear what the objective of the network is, as the loss function is the main function guiding the learning behavior of the network. Because of this, the loss function is sometimes interchangeably called the *objective function*. If the loss function is defined with error, the intended behavior and the actual behavior of the network will be two different things.

2.1.3 On Backpropagation

When we are talking about the learning part of a machine learning algorithm, what we are actually talking about is the process of finding the set of weights and biases that minimizes a certain loss function. The process of setting these weight via incremental steps is known as backpropagation.

Historically the advent of backpropagation lead to the second boost in machine learning research (Goodfellow et al., 2016), but it has also found many other uses

besides neural networks, as it can be applied to all systems involving simultaneous equations (Werbos, 1990).

To dig a bit deeper into this concept, let us for a second think about the neural network as a simple function with an input and an output. Let the function be $f()$, the input be x, and the output be y giving us the function $f(x) = y$. Now, the first x represents the data, the $f()$ represent the whole neural network, and the y represent the output after the neural network has done its computations on x.

$f(x)$ is thus a function mapping from x to y. When we do the backpropagation we are trying to guide the neural network such that it approximates the function best explaining the data in this sense. We want the output y to approximate some expected output defined in supervised learning by the labels. Now, as the input x is given in form of the data we want to train the model on, and the output y is the labels we want to train our function to map to, the function we would like to learn is the function $f()$, that correctly maps from x to y with the lowest possible difference between the function output and y as measured by our loss function.

When we are trying to learn this function $f()$ we thus need to know how the loss changes when the parameters change. This means we need to find out what happens to our output loss if we change the weight connecting a given neuron to another. In a sense we are trying to find out how sensitive the loss function is to each weight and bias. This means looking at how the rate of change is in the output loss (defined by the objective function) when we change each individual weight and bias. As we want to figure out how a variable changes with respect to another, we can here use the field of mathematics interested in rate of continuous change which is calculus.

The idea hinges on the fact that layers later in the system have as input the output of layers earlier in the system. As such, the whole neural network can be described as one great nested function. In each layer of the neural net $f()$ we take the dot product between the previous layers output and the weights of the current layer, hereafter a bias is added, and the activation function is applied. As such the output of a previous layer is also a function of weights, biases, and activation - same as the current layer. This is what allows the neural net to be written as a series of nested functions.

Now, because of the chain rule in calculus we can propagate from the inner most nested function - which is ultimately the output - and from this iteratively go back through the layers to calculate how the loss value changes with respect to the weights and biases. This is essentially backpropagation (Pineda, 1987).

Now that we have the slope of each weight and bias describing how the loss value changes if we change the weights, we want to change the weights in a direction that minimizes the loss. But how big a step we want to take in that direction is not a trivial matter. This step is known as the learning rate.

One thing to note is that as we get closer and closer to the ideal configuration of the weights in the system, taking too large a step might overshoot and land us in a setting where the weight adds more error than before. Because of this we often make the learning rate smaller the more fine-tuned the network becomes. The way in which we change the learning rate during learning is known as a *learning schedule* (Fahlman et al., 1988).

Some specific activation functions

I will here quickly go through three of the most common activation functions and how these affect backpropagation. Firstly let us take a look at the sigmoid function.

This a broadly used activation function of the form $f(t) = \frac{1}{1+e^{-t}}$. One of the properties of sigmoid function is that it starts to taper out and become saturated when the function input becomes very positive or very negative. This makes the derivative (and thus the learning step on the weights during backpropagation) taper out and become insensitive to small changes in its input when the input is very large or very small. In the paper (LeCun, Bottou, Bengio, & Haffner, 1998) the sigmoid function was shown to slow down learning because it had a none-zero mean. The same was also investigated in the paper (Glorot & Bengio, 2010) where the authors showed that this activation function is unsuited for neural nets where the weights have been randomly initialized - which leads especially the top hidden layer to saturate.

Therefore when it is called for to use a sigmodal activation function - which can be necessary as some learning frameworks have additional requirements that doesn't allow for other types of activation functions - it is often better to us a tangent activation function of the formula $f(x) = \frac{2}{1+e^{-2x}} - 1$ (Goodfellow et al., 2016). The Tanh has a mean of zero - as can be seen in figure 2.4 - which somewhat mitigates the problem of the sigmoid function by firstly giving it steeper gradients and secondly by avoiding a bias in the gradients as the Tanh is similar to the identity function when the input is near zero. Lastly we have the rectified linear unit or simply ReLu activation function which in many ways has become the standard activation function in modern neural networks (Jarrett, Kavukcuoglu, LeCun, et al., 2009; Glorot, Bordes, & Bengio, 2011). The ReLu can be described with the formula $f(x) = \max(0, x)$ meaning that $f(x) = 0$ if $x < 0$ otherwise simply $f(x) = x$. Because the ReLu is a nearly linear function, it gets to keep a lot of the properties that makes linear functions easy to optimize with gradient based backpropagation.

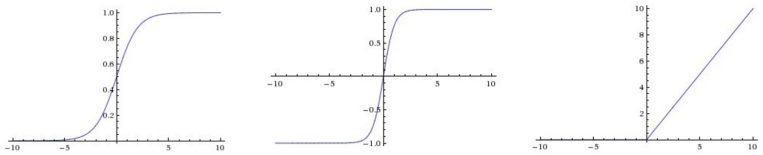

FIGURE 2.4: Three popular non-linear activation functions

2.1.4 On different output

Generally we can talk about two kinds of output to a neural network - discrete and continuous output. Mathematically speaking a continuous value is a value able to gradually change and take on infinitely many values within a range. When working with data this is often the stuff that can be measured - like a persons height, or the speed of an animal and so on. Inversely discrete data is data that is divided into boxes and thus doesn't have a smooth change in the transitions between the different chunks. As such discrete data is not measured but instead counted - like the amount of people on a bus, or the pebbles in a sandbox etc..

Now, when it comes to neural networks, if we are trying to train the neural net to classify some dataset into a bunch of classes - such as recognizing objects on an image, or determining whether a tweet is positive or not etc. - the outputs have to be in discrete boxes. As such there are generally speaking two kinds of output - discrete and continuous

Soft-max and discrete outputs

The softmax output function - or soft-argmax function as it is sometimes also (and arguably more aptly) known - is a way of forcing the outputs of a neural network to sum to one so that the output can represent a probability distribution across discrete alternatives (Goodfellow et al., 2016). As such the softmax is used when we want to turn the output of the last layer of the neural network into a set of probabilities of predicting what classes are most likely. The formula for this can be seen in equation 2.7

$$y_i = \frac{e^{z_i}}{\sum_{j \in G} e^{z_j}} \tag{2.7}$$

Where G is the group of logits arriving from the output layer of the network. Examples of its use will be shown shortly. But firstly we have to delve into some of core problems of language generation.

2.2 The core problems of language modeling - and how to solve them

In this section i will first explain the mystical art of transforming words to math known as *word embeddings*. I will then go on to talk about how to embed whole sequences, and then discuss some methods of how to asses sequences that have been generated. Along the way I will point to problems and their possible solutions. Lastly I will delve into the real meat of the GAN universe and report on some of the latest innovations in the field.

2.2.1 word embeddings

The necessity of representing words as math comes about when the neural network tries to exert the necessary linear algebra computations on the input in an attempt to find and replicate an underlying pattern. This can hardly be done with text alone, as text and math are not directly compatible without some intermediary translation from the system of language to the system of mathematics. As such we need somehow to embed the words of a language into mathematics - why this bridge, gaping between the logic of maths and the logic of language, has come to be known as word embeddings. Let us call the collection of all the words of a language the *vocabulary* of that language. The goal of word embeddings is thus to map every word in the vocabulary to a mathematical entity (such as a number or a series of numbers) in a way that allows the neural network to do its computations.

introduction

Without any previous knowledge of the subject one might be tempted to simply give every number a unique integer and then call the job done. In such a system every word would map to a number, which fulfills our first goal of mapping the vocabulary to a mathematical entity. However, as described above the input to a neural net must have the same dimensionality as the number of nodes in the first layer of that net - thus by using only integers we would have to make the first layer of our neural net consist of only a single node. The neural net would thus not be able to do its computations on such an embedding, making the system fail our second goal.

Another way of showing the lack of information about the words captured in such a simple system would be to visualize the words in a coordinate system. In our case since all the words are merely represented as integers, the coordinate system would simply be a number line spanning from 1 to n where n was the number of words in our vocabulary. Let us assume that we have given each word in the vocabulary a number according to the order in which the words occurred - such that the unique words seen first in our training set got the lowest number and vice versa. The only information captured in our embedding would then be a measurement, spanning from early towards late, of when the word occurred in our training text. This embedding would not capture enough information about the internal relation between the words for our neural net to start generating similar text. We thus need to add more dimension to the embedding - in essence make a larger space allowing for more relations between the words to be captured. How then, one might start to think, do we add this information to our embeddings such that the neural network can start to learn how a language functions.

Sparse embeddings

The simplest way to mathematically represent a word such that it firstly maps the word to a mathematical entity and secondly allows the neural net to do its computations on it, is via a vector called a one-hot encoded vector. Formally a one-hot word vector x can be defined as $\mathbf{x} = \{x_1, x_2, \ldots, x_V\}$ such that some $x_k = 1$ and all other $x_{k'} = 0$ for $k \neq k'$. In a more intuitive sense every dimension in the vector belongs to a word in the vocabulary and depending on what word we are trying to encode we place a 1 in its place and a 0 all other places. As an example if we have the sentence "I like dogs I do not like cats" we have a vocabulary set of cardinality 6 thus the above vector x would in this case become $\mathbf{x} = \{x_I, x_{like}, x_{dogs}, x_{do}, x_{not}, x_{cats}\}$ and as such the mapping from "cats" to word vector would be "**cats**" $= \{0, 0, 0, 0, 0, 1\}$. We now have a way to feed our words into the neural network.

2.2.2 Example of specific loss function

With this in mind, were now ready to look at a common objective functions used when modeling language. As we saw previously when dealing with language the output of a neural network typically is a distribution over the vocabulary acquired by way of a softmax layer. One popular method is then to train the model to predict the next word of the sentence given information about the previously occurring words in that sequence (Rosenfeld, 1996; Tomáš Mikolov, Karafiát, Burget, Černocky, & Khudanpur, 2010). As such we need a way to compare the output of the softmax with a one-hot representation of the word we are trying to predict. One popular way of measuring this is via categorical cross-entropy loss. This approach can be used when dealing with a distribution over different classes, which is useful in language modeling, where we often have a vocabulary of some size, and our neural net has given us an output of probabilities indicating how likely each word in our vocabulary is to be the actual word. The formal definition of cross-entropy has to do with the information theoretical notion of entropy proposed in the now infamous paper by Claude Shannon called "A Mathematical Theory of Communication" Shannon, 2001, according to which cross-entropy [1] is defined as a measure

[1] it is worth noting that many papers use the term *cross-entropy* specifically to denote the negative log-likelihood of a or softmax distribution, when this is better denoted as *Kullback-Leibler divergence*,

of distance between two distributions measured in bits. The function is described by equation 2.8.

$$H(p,q) = -\sum_i p(x_i) \log q(x_i) \tag{2.8}$$

where p denotes the actual distribution (here the one-hot vector), q denotes the predicted distribution (here the softmax output), and x_i is the ith entry in the vector. This measurement can then be used to guide the learning during backpropagation. Many other distance measurements can be taken into use (Lee, 2000). These include the Total Variation distance (Dahl, Adams, & Larochelle, 2012) and the Wasserstein (or Earth Mover) distance (Arjovsky, Chintala, & Bottou, 2017).

2.2.3 Beyond sparce embeddings

However, we are not quite there yet. Our one-hot vector carries no information about the word or how it relates to other words - like before if we assume each word in the vocabulary to be given a placement in the vector according to the order in which the word occurred - the only information gathered in this one-hot representation is how early or late the word occurred in the text we are training on. This is not ideal. Ideally the geometric relationship between word vectors should reflect the semantic relationships between the words in the word embedding space. The idea is that the embeddings should map human language into a geometric space with the vector representations being a placement in a vector space neighborhood with semantically similar words. Said in another way: What we aim to do is to encode the semantic relationships between the words as geometric relationships. In principal such a space would (as an arbitrary example) allow the same transformation to go both from "King" to "Queen" and from "Prince" to "Princess".

This idea of dense low dimensional embedding spaces for words - computed in an unsupervised way - is not a completely new idea. It originated around the turn of the millennium. Here a range of papers was published (Y. Bengio, Ducharme, Vincent, & Jauvin, 2003; S. Bengio & Bengio, 2000; Y. Bengio & Bengio, 2000). These presented the idea of "learning a distributed representation for words", instead of using a one-hot-encoding, thus avoiding the computationally expensive practise of working with sparse vectors. A practise the authors in their papers called the "curse of dimensionality". But the idea really started taking off in research and industry after the release of one of the most famous and successful word embedding schemes called the word2vec[2]

Word2Vec

In (Tomas Mikolov, Chen, Corrado, & Dean, 2013; Tomas Mikolov, Sutskever, Chen, Corrado, & Dean, 2013) the Google team presented two now canonical ways of representing words as vectors. These being the Continuous Bag-of-Words (CBOW) model and the Continuous Skip-gram Model - commonly known under the joint name of Word2Vec. The Word2Vec methods are used to create embedded dense representations from sparse representations of words. The basis of this approach is a very simple neural network only consisting of an input layer, an output layer, and

as any loss measured as the negative log-likelihood is a cross-entropy between the real and predicted distribution (Goodfellow et al., 2016; De Boer, Kroese, Mannor, & Rubinstein, 2005)

[2]Another popular embedding method - that I will not go into here - is the GloVe algorithm (Pennington, Socher, & Manning, 2014)

a single hidden layer. The function of the network as a whole is not useable in the context of embedding however, why we discard everything except the weights connecting the input layer and hidden layer after we are done training. This hidden layer then becomes our embedding space for the words we trained the model on.

To capture this idea a bit more thoroughly lets first take a deeper look at the objective function of the word2vec networks. Firstly we have the CBOW approach where the objective is to predict a focus word from its context. Secondly we have the Skip-gram model where the objective is to predict the context from the focus word. Both approaches have their application, however studies have shown that CBOW is more efficient with frequent words and that Skip-Gram is more efficient with infrequent words (Naili, Chaibi, & Ghezala, 2017). This is also concluded in the original paper (Tomas Mikolov, Sutskever, et al., 2013) where the authors write that the Skip-Gram works better with fewer training examples, and that this approach represents rare words or phrases fairly well. This is compared to the CBOW method that was found to have a much faster training time, and had a slight improvement in accuracy for more frequent words.

But let us dig deeper into each approach. In its simplest version the CBOW method only looks at a context of one word, meaning that it tries to predict the next word from the previous. As such in the simplest CBOW architecture (as seen in figure 2.5) the input is the word represented by a one-hot vector, and the output is the context word represented as a probability distribution over each word created by a softmax filter (this approach is very slow to train though, and a new approach called negative sampling will be introduced later).

The weights between the input layer and the hidden layer can be represented by a $V \times N$ matrix. Let's call this W - where V generally would be the cardinality of the vocabulary and N the number of dense dimensions we want to reduce to. The weights between the hidden layer and the output layer (later to be discarded) can be represented by an $N \times V$ matrix, which we can call W'

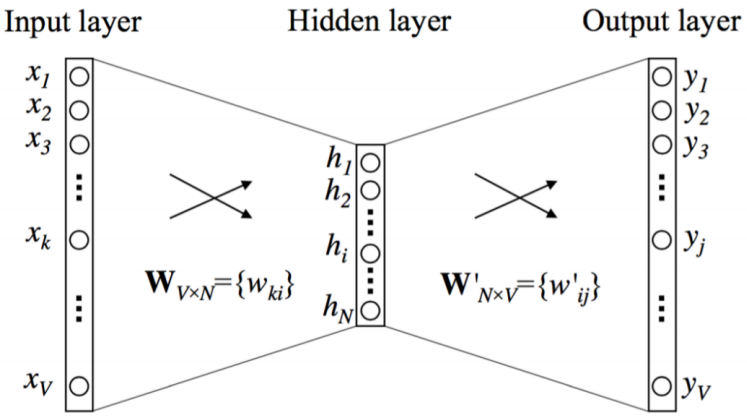

FIGURE 2.5: CBOW model with window size of 1

https://www.cse.unsw.edu.au/~cs9444/18s2/lect/09_Language4.pdf

The objective of this neural net is thus to maximize the probability of predicting the focus word when given as input the context words - given some window size defining the context. The objective function to be minimized is thus defined as the cross entropy between the predicted focus word and the actual focus word.

Formally we are given a corpus of context words w and their focus c, whereupon we consider the conditional probabilities $p(w|c)$ - that is, we consider what the probability of seeing the words is after we've been shown the context. Now, when given a text, the goal is to tweak the weights and biases - lets call these θ - in the formula $p(w|c;\theta)$ such that we maximize the probability of seeing the focus word given the context words and the parameters (Goldberg & Levy, 2014). To formalize this let D be the set containing all word and context pairs from the text - then the general goal of the neural net can be written as seen in equation 2.9

$$\arg\max_{\theta} \prod_{(w,c)\in D} p(w|c;\theta) \tag{2.9}$$

Now, lets increase the window size and move to a more general CBOW model. The architecture thus changes allowing for a larger window size and becomes as shown in figure 2.6

Here we see that the input layer changes such that it takes into account the now multiple words making up the context. But here a new problem arises - a subject we will devote its own subsection - namely the problem of representing multiple words in one vector. As can be seen in figure 2.6 the problem is one of concatenation. The reason for this is that we have a $V \times N \times C$ 3-D tensor - where C is the number of context words to be taken into account - instead of just having a $V \times N$ matrix. Thus, to avoid completely changing our architecture, we need to find a way to condense the C context words into the space were we before only had one word. For now let us forego a deeper discussion of this issue, and suffice it to say that there are many (more or less complicated) ways of achieving this. In the context of the Word2Vec it is simply achieved by adding up all the one-hot vectors, thus creating a Bag-Of-Words (BOW), whereafter an average is taken by dividing every entry in the vector with the number of concatenated context words C (As a twist of semantic irony the context between the context words are lost in this manner, but as i mentioned, we will return to these problems in the section about sequence modeling). After we've concatenated the one-hot-vectors into a BOW and taken the average, we can continue as normal with optimizing towards predicting the focus words from the context.

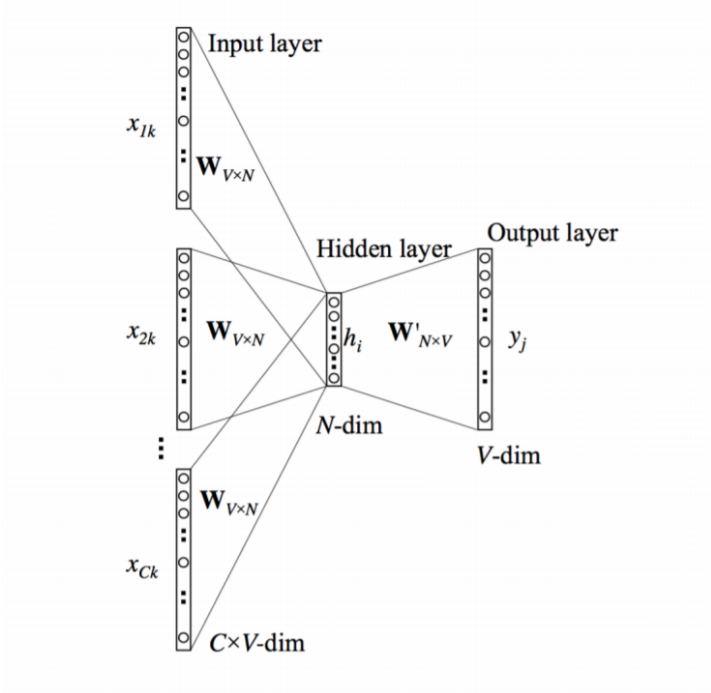

FIGURE 2.6: General CBOW model with total window size of C

https://www.cse.unsw.edu.au/~cs9444/18s2/lect/09_Language4.pdf

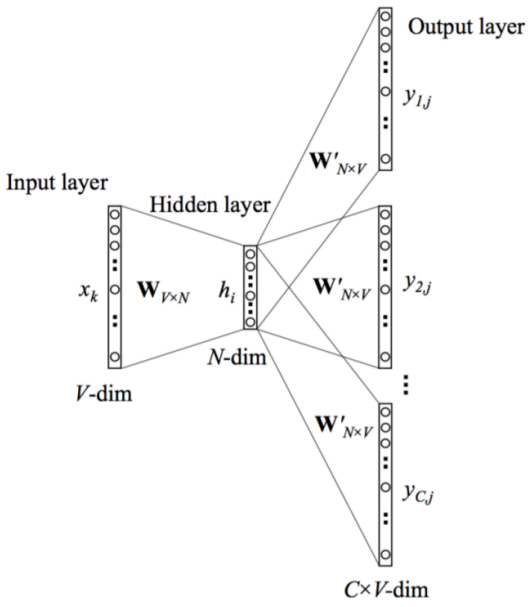

FIGURE 2.7: General Skip-Gram model with total window size of C

https://www.cse.unsw.edu.au/~cs9444/18s2/lect/09_Language4.pdf

Now, the Skip-Gram model is the opposite of the CBOW (as can be seen in figure 2.7) in the sense that the input is now the focus word and the output instead is the context words. The same problem now arises as before - namely the problem

of multiple context words - but it arises in the output layer instead of the input layer. The problem is now that we have multiple labels and only one prediction. The Skip-Gram solution is simply to produce a prediction vector for each context word, whereupon the losses calculated across the C training examples are added up (no average this time) and backpropagation is carried out as normally.

Similarly as before we can describe the goal of the Skip-Gram model to be the mirror image of the CBOW model - thus the overall goal of the algorithm can be described as seen in 2.10

$$\arg\max_{\theta} \prod_{(w,c)\in D} p(c|w;\theta) \tag{2.10}$$

Where D denotes the set of all pairs of focus and context words.

On intuitive embeddings and Word2Vec

One thing to note with many language embedding methods, is that even when the model arrives at an embedding space that manages to capture the semantic content of a vocabulary well enough to model a given language, there is no guarantee that either the relative placement between the words in that embedding space, or the embedding dimensions themselves has any intuitive interpretation. This means that neither the distance between the word vectors or the dimension they are measured along correlate to human intuitions. If for instance, a human was to design an embedding space from scratch and he or she attempted to do so in such a way that the words would be represented as comprehensively as possible, a reasonable starting point would be to start listing the properties that the enterprise of linguistics has taught us to attribute to words through the years. Foregoing a deeper discussion of what properties might be considered core or peripheral (along with the discussion of weather or not this distinction has any merit at all (Crain et al., 2010)), for the sake of example I will here just mention a few in no particular order, properties like: grammatical structure, phonetic characteristics etc.. If these were used as dimensions into which words could be embedded the way the words was represented in the embedding space could be directly interpretable: “the word is 0% a noun” or “95% a verb” or “the consonant sounds in the word-sound is produced 23% by the Bilabial segments”. The point is not that this would be the best embedding space, the point is that any word placed in this space could be interpreted back into lone standing statements about that word. This would correspond to human intuition. The problem, however, is that although some embedding methods have been shown to converge at a point where the dimension align with human intuition in very specific cases on carefully selected data sets (Blei, Ng, & Jordan, 2003) - none of the methods have been shown to consistently do so, especially not on large and diverse data sets.

But, even though the word embedding dimensions may not be interpretable so far, some algorithms such as the word2vec have been shown to capture an embedding space where the realation between the word vectors seemingly correspond to our intuitions (as can be seen in figure 2.8). A standard example of this is found in the subtraction of the "man" vector from the "king" vector, which yields the nearly exact same vector as when subtracting the "woman" vector from the "queen" vector. Another example is that the "swam" minus the "swimming" vector ends up at nearly the same place in the vector space as the "walk" minus "walking" vector. As such the dimensions themselves might not have any intuition attached, but the relation between the vectors seem to some degree to correspond to our intuition about both

the semantic and meaning, but also to some extent the structure and syntax of the vocabulary.

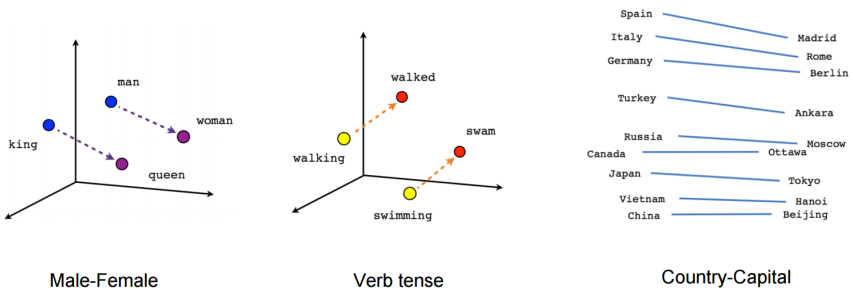

FIGURE 2.8: General Skip-Gram model with total window size of C

https://www.tensorflow.org/tutorials/representation/word2vec

Negative sampling

One of the problems with Word2Vec however is the slow training time. A lot of this has to do with the use of the computationally expensive softmax filter having each iteration to run a calculation for every word in the vocabulary. The reason the softmax is used is to create a distribution vector over the words that can then be used by the loss function to then guide the backprop. In (Tomas Mikolov, Sutskever, et al., 2013) the authors present a solution to this problem, via a new method called negative sampling. The idea is to cut down the computation power needed for the loss function to work as a guide for the backpropagation, while still having the guidance be useable. Negative-sampling solves this by not looking at the whole vocabulary on each iteration - instead it only considers a subsample. Negative-sampling is still used in the framework of the skip-gram model, however it is instead trying to optimize a very different objective.

The idea is to avoid training all the parameters of all the words for every iteration. Instead we create a bunch of fake examples of focus and context word pairs and then mix them in with the real pairs, whereafter we ask the neural net to only tell the real ones from the fake ones. This is done by sampling n random contexts from the vocabulary[3]

2.2.4 sentence embeddings

We have now looked at a few possible ways to embed words into a vector space in such a way as to mathematically represent the meaning of the words.

We saw how the Word2Vec algorithm was able to learn the meaning of words by learning to predict words from contexts and vice versa. This was done by concatenating all the context-words into a BOW whereafter an average was taken. As was shown this methods works fairly well at generating word embeddings but it has some shortcomings. By representing a sentence as a BOW we are loosing the order in which the words occurred. This means that in the representation of a BOW the sentence "I like dogs not cats" means the same as "I like cats not dogs". When the order of the data is an important factor for understanding the data, we call this data

[3]The paper suggests selecting 5-20 for smaller datasets, and only around 2-5 words for large datasets. Also there is a negligible probability that the sampled negative words might be part of the context words, but the computational cost of correcting this is not worth the gain (Tomas Mikolov, Sutskever, et al., 2013)

sequential data. An example of sequential data would be language, but also stock market prices, waves patterns and the like .

If we want to preserve the structure and context of the sentence we must thus move away from the BOW and find another solution.

One way one might think of solving this would be to simply concatenate all the one-hot representations of the words in the sentence. If we take the example above "I like cats not dogs" would thus become the vector:

$$\{1,0,0,0,0,0,1,0,0,0,0,0,1,0,0,0,0,0,1,0,0,0,0,0,1\}$$

By doing so we have preserved the structure of the sequence. This however, even with a tiny vocabulary, becomes very unwieldy as soon as we make just a slight increase in the sequence length. One could imagine a fairly small vocabulary with 10.000 words and a then feeding in the text 10 words at a time, which would mean that the first layer of our neural network would have to consist of 100.000 neurons - a computational cost that is not feasible. Another inconvenience with this representation is that now similar meanings becomes difficult to model. For instance the sentence "Tomorrow it will rain" and "It will rain tomorrow" mean the same but have two very different representations. So concatenating the one-hot vectors is not an option either.

Going back to the BOW representation as used in the word2vec algorithm, we find another range of problems occurring with the approach of only taking a specific window size into account. Firstly we see that we can find no guaranties of the context words having any relevant correlation to the prediction of the focus word, and secondly we see that this method has no chance of finding long-range interactions between words outside the context window (Dietterich, 2002). Especially this last problem is troublesome when trying to understand sentences or even whole texts, as for the most part in written language there are a lot of contextual information that stretches far beyond the effective range of a context window as described above. To predict the next word in a text we more often than not must have understood something going tens or even thousands of words back. As an example we could imagine trying to model a story about a person who gets introduced in the first paragraph as a passenger on a flight going to Paris. Now, the whole story could unfold on the plane whereupon we might find the last sentence telling us that: "the passenger landed safely in the city of *blank*". If the neural network didn't have a way of modeling this long-range dependency, it would have no chance of guessing the correct answer to be Paris. Thus to make a neural network capable of understanding and generating text we must first build into it a sort of memory. And here Recurrent Neural Networks (RNNs) enter as a method of -at least in some way - addressing all these problems (Tomáš Mikolov et al., 2010) .

On Recurrent Neural Networks

A RNN looks a lot like a basic neural network as the one introduced in figure 2.3 as can be seen in figure 2.9. There is just a slight but substantial difference in what the hidden nodes are computing.

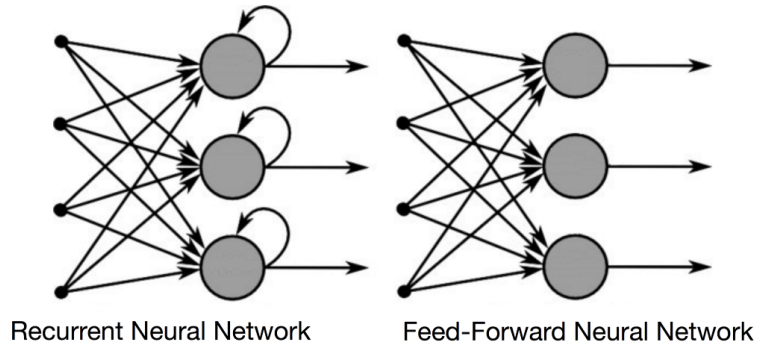

FIGURE 2.9: A Recurrent Neural Network compared to feed-forward nerual network

*https://cdn-images-1.medium.com/max/1080/0*mRHhGAbsKaJPbT21.png*

As the figure shows there is a recursive aspect to the calculation (hence the name), such that instead of merely calculating a function on the input words it is also taking into account the output word predicted from the previous time step - which we call its previous state.

As such at each iteration all neurons of that time step are connected to themselves in the next time step, where the exact same computations takes place, except the input is different. We call a neural network where all neurons in the network are connected to all neurons in the consecutive layer a fully connected neural network. Now, because RNN's share parameters across the different tokens in the sequence RNN's are invariant to the placement of the word in the sequence (Vinyals, Bengio, & Kudlur, 2015). This means that RNN's are very robust at dealing with sequences of different length (Goodfellow et al., 2016). This makes the recurrent neural network very good at dealing with time series data (or sequential data in general). The diagram in figure 2.10 shows a RNN being unfolded to a full network.

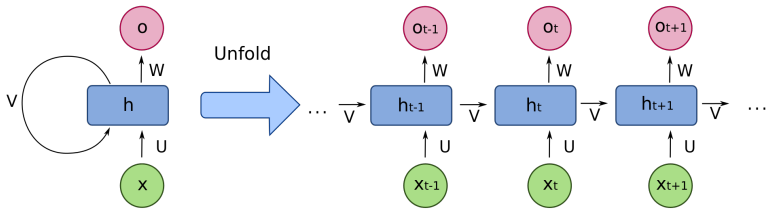

FIGURE 2.10: An unfolded Recurrent Neural Network

https://en.wikipedia.org/wiki/Recurrent_neural_network

Using the same notation as in figure 2.10 we can thus present how the RNN evolves over time as presented in equation 2.11

$$\begin{aligned} V_t &= f\left(h_t V_{t-1} + U x_t + b_V\right) \\ o^t &= g\left(W_t V_t + b_y\right) \end{aligned} \tag{2.11}$$

Where: V_t is the activation vector of the time step t, U is the weights connecting the input with the hidden layer, h_t is the hidden state at time t, o_t is the output at time t, and W is the weights connecting the hidden state with the output.

To unroll or unfold a RNN simply means to write out the network such that each computation in the sequence is accounted for. As an example if we want the RNN to learn how to model the poetry of John Milton we might choose to feed it in one line at a time, as such if we a looking at a line or sentence with 10 words we would unroll the network into a 10-layer fully connected neural network, where each layer was fed as input the previous output along with the word belong to that point in the sequence.

As the RNN computes its way through the sequence it thus takes a representation of the previous time step with it onto the next - this is done though the output merged with the next sequence word. And as the output of the previous time step is contingent of the one before that, while we go though the unfolded network we iteratively merge the words of the sequence into a placement in the vector space. As such we are able to represent a sequence consisting of many tokens in a single vector.

This way the RNN is able to remember words that was seen in past time steps. As such the state of the weights at any given time contains information about all previous time steps (Goodfellow et al., 2016). This somewhat takes care of the previously presented problem of accounting for the long range interactions between the words, however we still have to content with the problem of selecting what contextual words carry significance and which ones doesn't. Along with this we find the problem that recurrent neural networks are very difficult to train - especially when we want the network to learn the long-term dependencies between the words.

One of the problems attributing to this is known as the *vanishing gradient* problem (Y. Bengio, Simard, & Frasconi, 1994). This problem arises when the gradients of the weights (with respect to the loss function applied to the output of the network) becomes too small early in the system. This means that the rate of change is so small that even if we change the weights a lot, we wont change the output loss very much (Hochreiter, 1998) - this can be visualized as a plateau in the gradient space. Some fairly elegant solutions have been suggested to solve this. For instance by simply normalizing the initialization or changing the activation functions (Glorot & Bengio, 2010). However this only greatly mitigates the problem, whereas to solve it a change in architecture is needed. Such a change is found in the idea of RNN's with Long Short-Term Memory (LSTM). The original LSTM added a memory-cell consisting of an input-gate and an output-gate to the RNN. As can be seen in equation 2.12 the gates are basically shallow (one layer) neural networks meant to filter out what values to update and what values to output (Hochreiter & Schmidhuber, 1997). The idea is to mitigate the vanishing gradient by introducing a series of gates to keep the error signal "fresh" during backprop. The input gate looks at the input - which is the concatenation of the output from previous step and the new token - and then decides how much of the input to commit to the memory state of the RNN - consisting of the internal RNN weights. The output does the same but instead decides how much to pass on to the next state. This is shown in equation 2.12.

$$\begin{aligned} InputGate_t &= \sigma\left(W_i \cdot [h_{t-1}, x_t] + b_i\right)) \\ OutputGate_t &= \sigma\left(W_o\,[h_{t-1}, x_t] + b_o\right) \end{aligned} \tag{2.12}$$

For succinctness sake I will presented a somewhat simplified formula here as there are a few more steps concatenating and scaling the input along with updating the

memory - but in its essence the idea is simply to add two more neural layers to decide what to update and what to pass on. Added to the memory cell was later the forget-gate meant to reset the memory blocks once their contents become unusable and out of date (Gers, Schmidhuber, & Cummins, 1999).

$$ForgetGate_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] + b_f\right) \tag{2.13}$$

In its current version the LSTM can look as seen in figure 2.11 where the first sigmoid layer - as seen in the middle box - is the forget gate, the second is the input gate, the tanh layer creates a vector of new candidate values, and the last sigmoid is the output layer. The magenta nodes are updates to the memory weights.

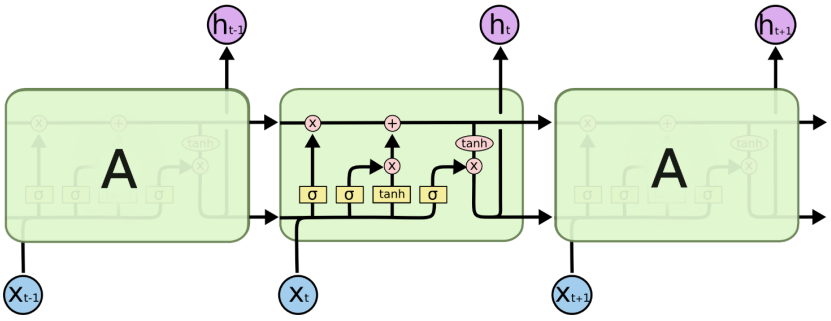

FIGURE 2.11: An RNN LSTM network unroled to trhee states

http://colah.github.io/posts/2015-08-Understanding-LSTMs/

This was again expanded on by adding peep-hole connections which allows the gates to look not only at the input but instead at both the input state and the cell memory state (Gers & Schmidhuber, 2000; Gers & Schmidhuber, 2001)

Other versions of the LSTM have gained popularity in recent years such as the GRU cell (Chung, Gulcehre, Cho, & Bengio, 2014)

Sequence to sequence models

A sequence to sequence model, as the name would suggest, is a neural network architecture taking a sequence as input and outputting a sequence as well. Such a setup can be used in translation, where we have a sequence of words and we want to make this into a sequence with the same meaning in another language (Sutskever, Vinyals, & Le, 2014). It can also be used in chat bots, where we have a sequence as input and we want to produce a meaningful output as a reply (Qiu et al., 2017). Or for poetry generation as in (Yi, Li, & Sun, 2017) where they use an encoder–decoder model to generate Chinese poems. This setup works by first boiling down the sequence to a single vector representation, and then from this vector generating a new sequence (Cho et al., 2014). More complex sequence to sequence also excicst such as variational autoencoders (VAEs) (Y. Bengio, Yao, Alain, & Vincent, 2013; Doersch, 2016; Sønderby, Raiko, Maaløe, Sønderby, & Winther, 2016) with dropout (Molchanov, Ashukha, & Vetrov, 2017; Srivastava, Hinton, Krizhevsky, Sutskever, & Salakhutdinov, 2014).

BLEU and ROUGE -Assessing the quality of a sequence

We saw above some of the ways we can generate sequences via different neural network architectures. Let me here shortly go through two methods of assessing

the quality of sequence, that historically has become somewhat of a community standard. First of these is the bilingual evaluation understudy or *BLEU* algorithm. This metric was developed specifically for machine translation and has as its primary goal to compare machine generated sequences to human generated sequences. The idea is basically a modified precision score that compares the overlap between the generated text as n-grams and the human generated text. Added to the precision score is a somewhat clever *brevity penalty* preventing very short generated sentences to have too high a score (basically what recall does for the f1-score) (Papineni, Roukos, Ward, & Zhu, 2002). Another commonly used set evaluation metrics for sequences are the recall-oriented understudy for gisting evaluation, commonly known as *ROUGE*. These algorithms are used for text summarization and are similar to BLEU in the sense that they are both attempting to map the f1-score (consisting of recall and precision) onto the language domain by redefining what is meant by precision and recall. In the ROUGE algorithms recall is defined as the relation between the number of words that are present in both the generated and the real summary, and the total number of words in the real summary such that: $\text{Recall} = \frac{Overlapping\ \ words}{Words\ \ in\ \ real\ \ summary}$. Said in another way it is a measure of what percentage of the real summary the generated summary is capturing. Precision is defined as the relation between firstly the number of words present in both the generated and the real summary, and secondly the total number of words in the generated summary such that: $\text{Precision} = \frac{Overlapping\ \ words}{Words\ \ in\ \ generated\ \ summary}$. Which simply put is a percentage measure of how many of the generated words was to be found in the real summary (Lin, 2004). There are different ROUGE metrics using n-grams, skip-gram co-occurrences, and longest common sub-sequence measurements. Common to these are that they all rely on labels to do a comparative analysis. However when dealing with many other practical use-cases, besides translation and summarization, it is not always possible to design a task specific loss to asses a generated sequence consistently. This is very much the case when dealing with the generation of poetry (X. Zhang & Lapata, 2014) why domain specific assessment tools (like BLEU and ROUGE) are not always applicable.

2.2.5 The problems of exposure bias and sentence level loss function

As we saw above the RNN style neural network solves some of the issues we had when working with sequential data. But a RNN does not solve *all* the problems arising when attempting to generate text. Most noticeably there are two major drawback. Firstly let us address what is known as exposure bias - a problem not only occurring with RNN's but also with n-gram models such as in (Kneser & Ney, 1995) and feed-forward neural networks such as in (Morin & Bengio, 2005). The reason for this problem arises when we generate text from the trained model. As we saw in the section on loss functions for language models, the models are trained by showing it words from the real data and asking it to predict the next word, whereafter the loss is measured (in some way) and the model updated. Now, when we generate text with these models they output entire sequences, and they are doing so by predicting one word at a time. Therefore when the next word in the generated sequence is produced it is done so with the other generated words as its input. This causes a problem, as the model was not trained on the distribution of the generated words, but instead on the distribution of the real words. This causes the word-losses of every time step to rapidly accumulate. As such we call it exposure bias when the model is never exposed to its own errors during training (Y. Bengio, Yao, et al., 2013; Ranzato, Chopra,

Auli, & Zaremba, 2015). This problem have been attempted solved in many ways such as with the Search-based structured prediction algorithm (or SEARN for short) (Daumé, Langford, & Marcu, 2009), or more recently with curriculum learning inspired ideas such as scheduled sampling presented in (S. Bengio, Vinyals, Jaitly, & Shazeer, 2015). However none of these have shown to be a stable solution to the problem, and scheduled sampling was shown in (Huszár, 2015) not to address the real problem directly. The exposure bias could possibly be solved by opting for a task specific loss function - thus judging the sentence as a whole instead of using a token-level maximum likelihood approach (Y. Zhang, Gan, & Carin, 2016). But as was discussed above it is very difficult to define a domain specific loss function for sequences. The second major drawback is in extension of this. The problem is that we are training the network on word-level losses, while after we have generated the sentence we judge it as a whole. This means that we are training for something else than we are testing for. This problem was dubbed "Loss-Evaluation Mismatch" in (Wiseman & Rush, 2016). One idea might be to use a metric like BLEU directly as the loss function, but this isn't trivial as the BLEU algorithm is not differentiable (Rosti, Zhang, Matsoukas, & Schwartz, 2011). As such we need to find a way to learn the algorithm what we are testing for, as well as making the loss function differentiable.

The Generative Adversarial Neural-network (GAN) architecture seems promising in having possible solutions to both the problem of exposure bias as well as addressing the loss-evaluation mismatch by having the ability to learn a domain specific sequence-level training objective by itself (Rajeswar, Subramanian, Dutil, Pal, & Courville, 2017). I will delve into the specifics of the GAN architecture shortly, but let me first address the problem now arising as we are shifting from a word level loss function to a sequence level loss function. The problem being that we can only get a loss-score when an entire sequence has been generated.

A loss function for a generative sequence model

In most practical cases where a sequence needs to be generated, we find that the sequence is not prone to evaluation before the sequence is complete. In chess for instance, one might need to sacrifice the queen to get a check mate further down the sequence of moves. As such, if we only judge the sequence from time step to time step, our neural network will miss out on learning more complex and nuanced series of moves. This means that the reward from the objective function only makes sense when judged on the entire sequence. In chess this would be all the moves from start to finish. In these tasks methods to simulate future gains from current decision are important, as there can otherwise be given no loss function for individual times steps but only for whole sequences (Bod, 1998; Brill, 1995). This is not only the case where we have a somewhat manageable search space like in chess, but also very relevant when dealing with machine translation as in (Sutskever et al., 2014) or in other games where the search spaces are large such as described in (Silver et al., 2016).

However if we define the objective function to give a loss score on the whole sequence and not on each token, we might solve the problem of short sightedness, but instead we now have the problem that we can not give intermediary loss scores. As such the sequence level loss function will distribute the loss equally on all the tokens. Because of this the backpropagation step will lack some crucial information when updating the weights, and it is not obvious that learning is possible at all in this scenario. As such we need a way to get a loss function both for the whole sequence, but also at every time step. This problem is what the authors of (Li et al.,

2017) dubbed *REGS* standing for reward for every generation step. In cases like this many have chosen to adopt methods like Monte Carlo search (Nivre, 2001; Browne et al., 2012; Y. Zhang et al., 2016; Yu, Zhang, Wang, & Yu, 2017).

2.2.6 On Generative Advesarial Neural Networks

The Generative Advesarial Neural-network is an architecture for generating new data similar to a distribution that was trained on. It was firstly presented in the paper (Goodfellow et al., 2014). The original intent of the GAN was to generate images but has later been attempted modified to work with discreet output like language also. The crux of the idea lies in the *advesarial* part of the name. Firstly we have some neural network - any will do but of course the best one depends on the problem to be solved - this neural network has as a job in this architecture to generate new samples from that approximates the original data that we are training on. Now, with language this would normally have been one with a loss function such as the cross entropy between the softmax output and the one-hot representation of the target word (as previously discussed). This analytically defined loss function however carries with it a lot of assumptions (such as the idea that the best way to understand a language is to predict it). The main point of the GAN is to avoid building a bunch of assumptions into the model via the loss function, by instead letting another neural network learn how best to design the loss function for the specific task. This means that in the GAN architecture the loss function of the generative neural network is not chosen by a human but instead learned by a neural network, as such the loss function of the generator network *is* another neural network.

As can be seen in figure 2.12 the forward pass for the GAN starts with a noise vector being feed into the generative network. The generative network in our case produces a range of poems. The poems are then passed on to the discriminator network along with a range of poems from the original source. The discriminator then has as a job to asses the probability of whether or not any given poem is from the real distribution (ie. the Milton poetry) or the generated distribution. When updating the networks in the backpropagation step, the discriminator has as its objective function to optimize towards putting the highest probabilities on samples from the real distribution. In direct opposition the generative network has as its objective function to try and fool the discriminator network, so as to optimize the discriminator networks output-probability that the generated text is from the real distribution. This is the adversarial part of the GAN[4]

The GAN architecture has shown tremendous results in areas spanning from the generation of realistic images (Goodfellow et al., 2014) to style transfer between images, and even video generation (Tulyakov, Liu, Yang, & Kautz, 2017). These areas are dealing with continuous functions why every part of the computation is fully differentiable - allowing the gradients to propagate through from the discriminator network to the generator network. This means that backpropagation is possible and thus that learning is possible. But when dealing with language we are typically working with it as discrete outputs, making it impossible directly to do backpropagation from the discriminator through to the generator. The crux of this problem lies in the sampling action choosing what outputs from the generator to feed the discriminator. This is a discrete sampling process, which means that the sampling operation is non-differentiable, and thus backpropagation becomes non-trivial

[4]It is worth noting that the GAN - with it's setup of a discriminator and a generator - theoretically will converge at a 50/50 judgment for fake/real, which is the same as the ideal passed test for a Turing test

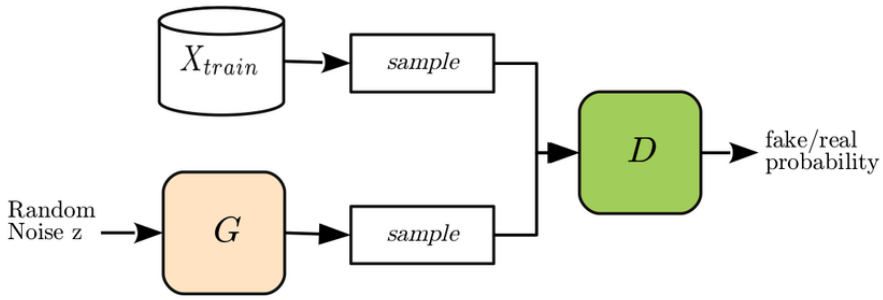

FIGURE 2.12: Simplistic Generative Adversarial Neural network

https://www.researchgate.net/figure/Generative-Adversarial-Network-GAN_fig1_317061929

To update the weights of the generative network via backpropagation means to guide the network such that it incrementally changes the weights in a direction that makes the generated sentences more realistic (as judged by the discriminator). However, as the generated data consists of discrete tokens, chances are that this incremental change does not correspond to a change in tokens. As such, if we insist on working with language as a series of discrete tokens, we must find some sort of gradient estimator for our system to be able to learn[5].

Previous work on GAN's for sequence generation

Many attempts have been made at solving the problem of differentiability. Some of the approaches rely on the idea of using policy gradients to solve the issue. Policy gradients is a technique originating from reinforcement learning (RL) (Sutton, McAllester, Singh, & Mansour, 2000). Following this framework we can see the generative model as an agent interacting with an environment defined as the input at every time step. The parameters of the agent then defines a policy, where this policy decides what words to chose given the input words. After the agent has reached the end of the sequence a reward is given according to some loss function. Using RL for language models was explored in (Bachman & Precup, 2015). RL was also used in (Ranzato et al., 2015) where the authors introduce the Mixed Incremental Cross-Entropy Reinforce (MIXER) algorithm. The MIXER is a text specific learning objective utilizing first cross-entropy training to find the optimal starting policy for the agent, whereafter a variant of the REINFORCE algorithm continues to guide the learning. Here the authors chose to use the BLEU algorithm as the loss function. Neither of these approached relied on the GAN architecture however. But other gradient estimators have also been proposed, as for example in (Jang, Gu, & Poole, 2016) where the non-differentiable sampling method from a categorical distribution (such as the output of a softmax layer) is replaced with a differentiable sample from the output of a novel gradient estimator dubbed a Gumbel-softmax estimator. Other methods include the straight-through estimator presented in (Y. Bengio, Léonard, & Courville, 2013).

In (Bahdanau et al., 2016) the authors address the problem of exposure bias by introducing what they dubbed a *critic* network. The output of the critic network is

[5]Another range of problems encountered when training a GAN has to do with convergence issues - one of the more noticeable ones being *mode collapse* as discussed in (Arjovsky et al., 2017) - unfortunately a deeper discussion will have to fall outside the scope of this work

used to approximate the gradient of the expected returns with respect to the parameters of the actor. As such the critic is feed the ground-truth training data of some time step, along with the output of the generator at the same time step (added to this is an input summary calculated using an attention mechanism), whereafter the critic outputs an approximated loss-score for that time step. In the paper the BLEU function was also used as the objective function that the critic was approximating. The method of using a critic has the benefits of having a greatly lower variance than the REINFORCE algorithms alone have, as well as exploiting the ground-truth of the input words.

These ideas have been expanded upon in a long range of papers utilizing RL techniques for the GAN setup. This is done such that the discriminator output becomes the loss function of the agent, with the agent being the generator. In (Yu et al., 2017) the authors introduced the SeqGAN where policy gradients was used to train the generator, and a Convolutional Neural Network (CNN) was used as discriminator. This addressed the problem of making the setup differentiable, and Monte Carlo rollouts was introduced to solve the problem of getting useful loss signals per time step. In (Dai, Fidler, Urtasun, & Lin, 2017) the authors also utilized Policy Gradient with Monte Carlo rollouts, but instead used it for image descriptions, yielding better results than state-of-the-art encoder-decoder models. In (Che et al., 2017) the authors introduced the maximum-likelihood augmented discrete generative adversarial network or simply MaliGAN for short. The main novelty of the MaliGAN was in its use of importance sampling to lower the variance of the discriminator. A thing to note about importance sampling is that it is *not* a sampling method for probability distributions - as the name otherwise might suggest- instead it is a variant of Monte Carlo approximation. In short, the basic idea of using importance sampling is to optimize the generator, not with the standard high variance output of the discriminator, but instead with a lower variance approximation. A very similar idea was independently developed for image generation in (Hjelm et al., 2017). In (Rajeswar et al., 2017) the authors solve the problem of a discrete output space without turning to gradient estimators such as RL. This is tested on poetry generation of Chinese poetry. The problem is attempted solved by allowing the output, being passed to the discriminator from the generator, to be a sequence of distributions over every token in the vocabulary. The real distribution is then fed to the discriminator as a sequence of one-hot vectors from the true data distribution. This is similar to the embedding style discussed earlier in the section about sentence embeddings, where a sentence was represented as the concatenation of its constituent words represented as a one-hot vector. To avoid the high computational cost however the model in the paper was set to use characters instead of words. The idea is that because the generator has to learn how to imitate the real distribution to fool the discriminator, it will learn to output one-hot vectors. This setup therefore allows the generative model to learn how to directly output one-hot word embeddings form a latent vector, thus making the setup fully differentiable by mitigating the need for a discrete sampling step. The same approach is also found in (Gulrajani, Ahmed, Arjovsky, Dumoulin, & Courville, 2017).

In (Fedus, Goodfellow, & Dai, 2018) the authors - among them the original inventor of the GAN architecture Ian Goodfellow - introduce the MaskGAN. This architecture combines many of the previously mentioned approaches and adds its own novel method in an attempt to modify the GAN to work with discrete data. The main contribution of the MaskGAN is in the way that the text is fed into the generator. The authors used a sequence to sequence model as the generator. As such it consists of an encoder and a decoder part. The masking part of the MaskGAN happens

when the text is feed to the encoder. For each sequence of text $x = (x_1, x_2, \ldots, x_n)$ a binary mask is created $m = (m_1, m_2, \ldots, m_n)$ to determine what tokens stay and what tokens are replaced with a mask token $< m >$. The decoder then fill in the masked tokens auto-regressively as normal - meaning that it uses the accumulated values from the previous time steps to predict the masked values. The encoder is being feed both the masked text and the filled in text. This setup also uses policy gradients to make the sampling step differentiable, however, it does so while using a *critic* to approximate the output of the discriminator network at every time step. This masked infilling of text allows the discriminator to give an error signal for each time step, mitigating the need for ideas such as Monte Carlo rollouts. The discriminator is identical to the generator except that the output isn't a distribution over the vocabulary, but instead a probability of the input being real at each time step given the true context of the masked sequence.

At time of writing the MaskGAN setup seems to be the pinnacle of text based GANs. As such I am basing my generation of poetry on this architecture.

Chapter 3

Methodology

Now that the tools to generate the poetry have been laid out, we need a way to asses it. In an attempt to judge the poetic artistry of the authors - both human and machine - a Turing-type test was conducted. In this chapter we go through the details of this experiment, as well as delve into how they were generated

3.1 Experiment setup

To asses the generated poetry I setup a partition on the internet for people to fill out a survey. This was written in JavaScript and HTML, and utilized a firebase database to automatically store the data[1]. The introductory text introduced the experiment, and defined to the participant what scale they were to judge the poetry on. I here decided to measure it on two scales. The first being grammatical soundness and the other being the poetic meaning. Grammatical soundness is to be understood as a measurement of whether the poetry adhere to the participants intuitions about grammar. The participants was thus asked to question themselves if anything in the poetry sounded like its was not grammatically correct (ie. was there a verb where a noun should be etc.). They were then instructed to give a low score if nothing - or close to nothing - was grammatically readable, and vice versa. The poetic meaning is to be understood as the semantics or the content of the poem. The participants was thus asked to question themselves if the poem made sense. The participants was also asked to question themselves if each line came together to form a whole, and weather or not there was a hidden or direct meaning in the verse. They where then asked to give a low score if the words seemed made up or if nothing - or close to nothing made any sense, and vice versa.
The experiment setup was as follows: Online participants were shown 40 different four line verses of poetry. These consisted of 10 verses from the three different generated corpora along with 10 verses from the real Milton corpus. Participants were then asked to rate the perceived quality of the grammar and the perceived quality of the poetic content of the verses. These were measured on two scales ranging from 0 to 6. Participants were also asked to asses weather the verses were made by Milton or by an algorithm. The 40 verses was selected randomly, by grouping all the corpora together in lines of four, whereafter 10 random poems were selected with the random sample algorithm from the python library simply called *random*. All participants was shown the same poems. As the experiment was anonymous I do not known the gender distribution. To make sure that participants concentrated on the task I also measured the time they took in making their assessments. When the experiment concluded there were entries from 61 separate participants - they were not all useable. The results will be discussed in the following chapter.

[1]The experiment (and the poetry) can be found here

3.2 The generative part

To generate the poetry I followed the maskGAN presented in (Fedus et al., 2018). I used a GAN setup were both the generator and the discriminator was a variational autoencoder. A similar setup was also used in a DTU based research project where the authors utilized a merged architecture between a VAE and a GAN with some success, but instead for generating images of faces (Larsen, Sønderby, Larochelle, & Winther, 2015).

3.2.1 Library used

The GAN architecture here used is one of very high complexity. It has many different moving parts interlocking and influencing each other. All in all it consists of five different neural networks (two encoders, two decoders, and a critic), and they are jointly codependent as the forward pass and the backpropagation needs to flow through all of them. The most substantial problem of implementation, however, is that when the generator is learning, the discriminator has to be static, and when the discriminator is learning the generator has to be static. This setup showed itself to be highly problematic as tensorflow does not yet have an inbuilt mechanism for this. As such I choose to not reinvent the wheel, and instead utilized the library already presented in (Fedus et al., 2018) [2] with only some minor tweaks. Because of my somewhat limited computational resources[3] i choose to run every model for a maximum of 10000 iteration. The syllabic language model was for some reason cut short at about 9000 iteration[4]. Most of the computation hours went with trying out different architectures and hyperparameter selection.

Hyperparameter setup

I changed around the hyperparamters when doing the initial tests. I also tried different generator and discriminator combinations - among these two variational bidirectional models as the generator and the discriminator. All these test however was done in the spirit of exploration, and any rigorous experiment of both hyperparameter-selection and model architecture will therefore have to wait until future work.

The generator and discriminator networks where mirrors of each other, as this allowed for both of them using the same word embedding space. They both consisted of two LSTM-RNN's (the encoder and decoder respectively) with 650 neurons. I set the learning rate of both networks to the relatively low starting point of 0.0005 as any values larger than this appeared to create a higher variance in the loss-output. I feed in the data in relatively small batches with a size of 30. The exact number was a trail and error based on intuition, but the benefits of not using too large a batch size are well documented (Keskar, Mudigere, Nocedal, Smelyanskiy, & Tang, 2016). I also found some inspiration for selecting initial values for batch size and learning rate in (Smith, 2018). I did a little experimentation on what the best sequence length was, and found that around four lines of text was seemingly the maximum length before the generated output seemed to become less coherent. The sequence length varied between the plain single padded text and the two preprocessing methods taking the number of syllables into account - these being 48 and 60 tokens respectively. I set the

[2] MaskGan code can be found here

[3] I was lucky enough to get 200 hours of computation on the abacus2 server - link to website here

[4] but as can be seen on the graphs in the appendix, this did not seem to have any big effect

dropout rate of both the generative and discriminative VAE to 0.5 following the advice of (Baldi & Sadowski, 2013). As a learning schedule i use an exponential moving average with a decay rate of 0.99. As I was using RL I also needed to set a discount rate reflecting the relationship between short-term rewards relative to those further in the sequence. I here set it to 0.95 following (Guo, 2015)

Chapter 4

Results

The machinery have thus exposed its lyrical prowess (or lack thereof) to the world, and in this chapter we will look at what the critics say. Firstly I will conduct a comparative analysis of some lines taken from Milton and some from the machine. Secondly we will look at how well the structure was approximated in form the syllabic count. Lastly we will look at what the experiment yielded

4.1 Qualitative analysis of syllabic stress patterns

Let us for comparison here look closer at some of the ways Milton put together his syllables. The following verses are excerpts chosen randomly by a generator. All poetry here presented was taken from the parts shown in the experiment. In the following i will present the generated poetry as well as the pieces from Paradise Lost. After this i will select a few lines and give an interpretation of their stress patterns.

Excerpts from Paradise Lost

With purpose to resign them in full time
Up to a better Cov'nant disciplin'd
From shadowie Types to Truth from Flesh to Spirit
From imposition of strict Laws to free

To mortal men above which only shon
Filial obedience as a sacrifice
Glad to be offer'd he attends the will
Of his great Father Admiration seis'd

To Judgement he proceeded on th' accus'd
Serpent though brute unable to transferre
The Guilt on him who made him instrument
Of mischief and polluted from the end

Milton

First lets take a look at the syllabic count of the poetry above. If we run the text through the previously described syllabic counter, we can obtain the following list where the $i'th$ entry in the list corresponds to the syllabic count of the $i'th$ line going from top to bottom: 10 10 10 10 8 10 11 10 10 9 10 9 [1]. We can see that the standard deviation of this sample is a relatively low 0.62 centered around a mean of 9.67 which

[1] It is also worth noting again that the syllabic algorithm makes quite some mistakes. The syllabic split poetry can be seen in the appendix for comparison

is very close to what we expect to see. I will later do a more thorough analysis on the whole text. There are also a few lines in the excerpts above that does not obviously adhere to the structure of an iambic pentameter. Let us take a closer look and see how Milton bends this structure. One such line is the third line of the second verse :

From **shad**-owie **Types** to **Truth** from **Flesh** to **Spir***it*

The stress pattern of this line can be written as: " - / - / - / - / - / (-) ". the brackets here show that there are no stress put on the last syllable. It is often referred to as a *weak ending* when a line has an extra unstressed syllable as seen here [2] . Another thing to note is that the syllabic algorithm didn't count "spirit" as containing to syllables, thus inadvertently ignoring the unstressed syllable in the syllabic count.

Another good example of Milton's inclination towards bending the meter is found in the second line of the second verse:

Fil-**ial obe**-di-**ence** as a **sa**-cri-**fice**

With a stress pattern of " - / / - / - - / - / ". We here see that Milton reverses the rhythm so the second and third feet becomes trochees instead of iambs.

If we instead direct our attention towards the generated poems we can try and get a first look at the patterns here emerged. Let us first look at the poetry with the normal preprocessing method.

Excerpts from standard processing

Training his hand rebuke him to remorse
With painful Emploid rowling first were bowing Air
Heard on the fence of Heavn the starrie train
To hottus passd which Satan filld

Who to the powerful Destiny and pure
Then high else to explode in fact of Warr
Did what I be to God who grew that weak
Then thou in Center for posson as easie

Nor many a berrie and of a kerhole turns
Among hisship Farr how first in Heavn or Night
Out in high face and to a Trees immense
Inbrokn and in seden chivalry

Source

We here see that the syllabic count is a bit off compared to Milton with a syllable count looking like: 10, 12, 10, 8, 10, 10, 10, 10, 10, 9, 10, 7. Although the middle verse seems good, and with the mean on this sample coincidentally being the exact same as Milton, we still see a range between seven and twelve syllables with a substantially higher standard deviation of 1.20.

If we dig a bit deeper we can see that the generated poetry is not completely off. Most of the lines seem to fit into a very clear iambic pattern. Even the semantic seem to be somewhat poetically interpretable. The third line of the second verse could easily be a line taken straight out of a monologue from the Devil in Paradise Lost:

[2] An argument could also be made that Milton mixed the pattern in the second foot. As **sha**-dowie could be read **sha**-dow-ie with two consecutive unstressed syllables and one stressed. This would make it a specific trisyllabic foot known as a *anapaest* instead of an iamb

Did **what** I **be** to **God** who **grew** that **weak**

Most of the lines also seem to be splittable into iambs, although the syllabic count is off on this sample. This somewhat indicates that the pentameter part of the iambic pentameter is not captured as well by the model preprocessed with standard preprocessing.

To take a look at some of the stress patterns generated besides the iambs lets look at the line:

Who **to** the **pow**-er-ful **Des**-ti-ny and **pure**

With what can be seen as a stress pattern of "- / - / - - / - - - /". We here see a strong divergence from the iambic pentameters. Firstly as the line has eleven syllables with a strong ending (even though the syllabic algorithm didn't count it as such). And secondly we see that the line contains three unstressed syllable in a row.

Let us move on and instead look at the poetry generated with the syllabic split language model.

Excerpts from syllabic processing

Our then my just exageing to that shape
Which as I namd speed we may easie to send
I know to him by this deep may have made
So felt us rose unvaild the wamer sense

Bedsing Victorie from every Trees Earth blow
Rebellion in her flaming sheer but makes
Beholding as the frowning and his wings
Canst from the Book of God structure by Heavn'

Thegressive but th ungodly but to save
Rousd as the day though vitive rest how stood
The Nor side high in batter to the Starrs
And wrought my throne If to his Senple view

Source

We can here see that on this sample the syllabic count is pretty spot on with a syllabic count of: 10, 10, 10, 9, 10, 10, 10, 10, 10, 10, 10, 10. The mean is 9.9 and the standard deviation is a very low 0.28.

This however is only when counted with the syllabic algorithm as we see that some of the lines actually contains more than ten syllables. Take for instance the first lines in verse two and three:

Beds-**ing** Vic-**to**-rie **from** eve-**ry Trees** Earth **blow**
The-**gres**-sive **but** th un-god-ly **but** to **save**

The stress patterns are not at all clear in these cases, and I here applied the pronunciation that seemed to flow the best. It is clear though that even though the syllabic count, as measured with the algorithm, shows the generated poem to have caught the syllabic structure well, with human assessment this is not the case. It is also noteworthy that some words such as "Thegressive" and "bedsing" seem to be a meaninglessly spliced together bunch of syllables.

Let us therefore instead move on to the last preprocessing method and the poetry generated from the double padded input:

Excerpts from double padded processing

Moist nutriment or the shrill Matin light
bended Dolphins play with his great World to
To have express it best the Angelic Angel
Ere in your friendly East it by our

But so thou stoodst in Heav'n and to be last
At my right hand and made them onely
A Heav'n and dance in all a moment wide
With dreadful Artillery had combustion here

His bountie Sovran to spend him with rage
Or all his place by Heav'n and fair the place
Of flight of Ages at her greater large
And with parts dayes and moves from the Skie

Source

We can here see that the syllabic count on this sample is fairly decent with a syllabic count of: 10, 10, 11, 8, 10, 8, 10, 10, 10, 10, 10, 9. The mean is 9.7 with a relatively higher standard deviation of 0.87.

Continuing on to the poetry itself, we see that - besides the rather bad last line of the second verse (coursed by the algorithms inability to split both "artillery" and "combustion" correct) - this time the read out syllabic count seem to correspond better to the count of the algorithm. It could thus be the case above (in the syllabic split poetry) that the syllables, when combined after the generation step, does not combine back into words with the same syllabic splits (and therefore not the same syllable count) when judged by a human reader. The poetry also has a slightly less muddled rhythmic structure than the previous one, as most of the lines can be put into iambic pentameter. And added to this is the seemingly more meaningful semantic with less made-up words and more coherence between the used words. The third line of the second verse seem to be slightly more poetic than the rest, so let us here take a look at its meter:

A **Heav'n** and **dance** in **all** a **mo**-ment **wide**

This line could be interpreted to mean that the the good things in life are fleeting, as the heaven and the dance are no more but a moment wide. Besides this it adheres to the iambic pentameter by having the stress pattern " - / - / - / - / "

4.2 Quantitative measurement of syllabic count

I will here use the algorithm described above to compare the syllabic count per line in the different poetry. This comparison is shown in figure 4.1. The first thing that stands out in the box plot is that preprocessing with plain words seem to result in a much higher spread, with much more radical outliers than any of the other processing methods. Secondly we can see at a glance that both the double padded and the syllabic language model has generated poetry with a fairly similar syllabic count as the source poetry.

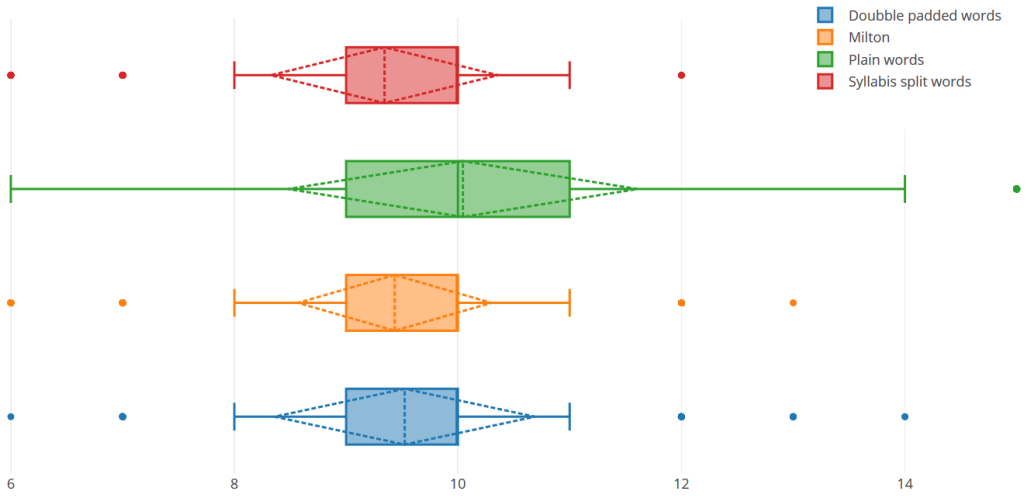

FIGURE 4.1: Boxplot comparing syllabic count of poetry generated with different preprocessing methods to Milton

https://lantow.github.io/Data_and_plots/SYL-COUNT.html

Let us dig a little deeper and take a look at the specifics of the distributions presented the table below.

	Mean	STD	Median
Syllabic	9.34	1.02	10
Plain	10.04	1.56	10
Double	9.52	1.16	10
Milton	9.52	0.67	10

Here we again see that the plainly processed poetry have a higher standard deviation than the other methods. It is, however, centered around a similar mean, indicating that it has found a somewhat similar pattern but simply haven't been able to approximate the pattern[3]. I presume that it would have converged on the correct syllabic distribution after more iterations, as all the information necessary for the algorithm to model the syllabic count is pressent in the text [4]. It also shows that the median of all the different distributions is 10, indicating that none of the methods got the algorithm completely confused. This, interestingly enough, seem to indicate that the double padding method actually did help the GAN algorithm to approximate the syllabic count faster.

4.3 Experiment results

The initial data consisted of 457 data entries distributed on 61 participants. After a closer inspection of the data it was clear that some people had used less time on

[3] I would have liked a plot showing how well the syllabic count was approximated over time. But unfortunately I couldn't find a way to make the syllabic algorithm (implemented in JavaScript) work with the tensorflow summary creating the graphs.

[4] Every word can be seen as mapping to an integer of the syllabic count, and the sum of each line is 10

the judgments than others. I thus started by discarding the data where the participant had used less than 30 seconds on the judgment. After this there where 292 entries left. It is worth noting that the judgments under 30 seconds on average took 14.1 seconds (after removing outliers farther away that two standard deviation of the original mean) - which in some cases might be a reasonable amount of time to asses a verse in. I initially choose to omit these as I was wary that the quality of the assessment might be compromised. However, after a closer inspection of the 292 entries left, it became apparent that the bulk of the data a came from just seven participants who had assessed all the 40 verses. These seven participants had on average used 43.8 seconds per judgment (after again removing outliers). I then isolated this data and found that on the entries were these participants had used less than 30 seconds the average was a relatively high 19.0 seconds with no assessments being under five seconds. As such I instead choose the data from these seven participants to be the starting point of the data analysis. This also had the benefit that there were equally many judgments in each category, alleviating any problems of an unbalanced sample. As such the data analysis is performed on a dataset of 280 entries with 70 assessments from each category. The standard preprocessing and the double padding was guessed to be human at the same error rate of 27.1% which is 19 out of 70. And unexpectedly this was also the case of the poetry from the syllabic language model and the poetry from Milton. These were both guessed to be wrong 13 out of 70 meaning that Milton was thought to be a computer and the syllabic language model was thought to be a human 18.6 percent percent of the time. All in all 77 percent was guessed correct, which is a ways away from a passed Turing test.

The results of the participants grammatical judgments can be seen in figure 4.2

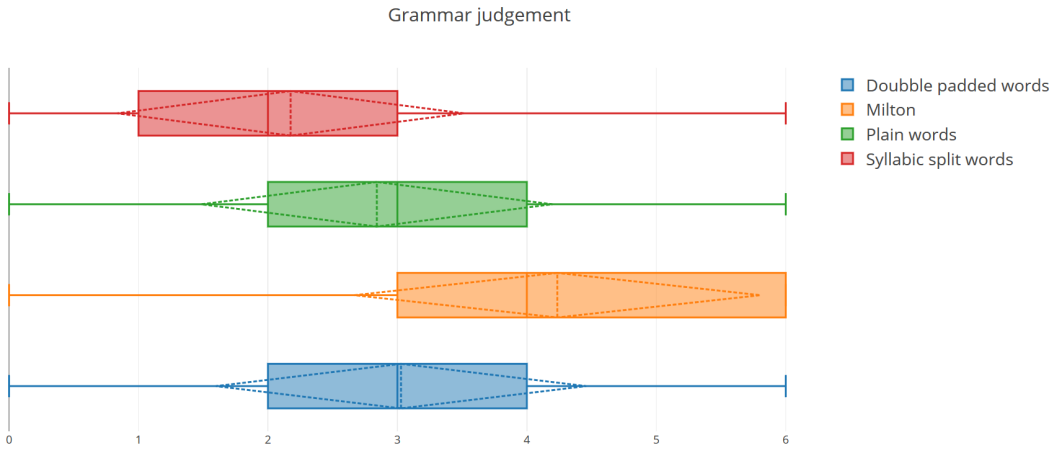

FIGURE 4.2: Boxplot of grammatical judgments for different preprocessing methods

https://lantow.github.io/Data_and_plots/grammar_judgement1.html

Here we see that participants consistently judged Milton higher than the generated poetry. We also see an indication that generating poetry with a syllabic language model seemingly will achieve worse grammar than a word level language model. The details are presented in the table below

	Mean	STD	Median
Syllabic	2.18	1.34	2
Plain	2.84	1.36	3
Double	3.03	1.43	3
Milton	4.24	1.57	4

Here we see that the syllabic padding method seemingly didn't contribute too a grammatical confusion even though it did contribute to a better syllabic approximation.

If we move on to the semantic judgments we see a very similar pattern as shown in figure 4.3.

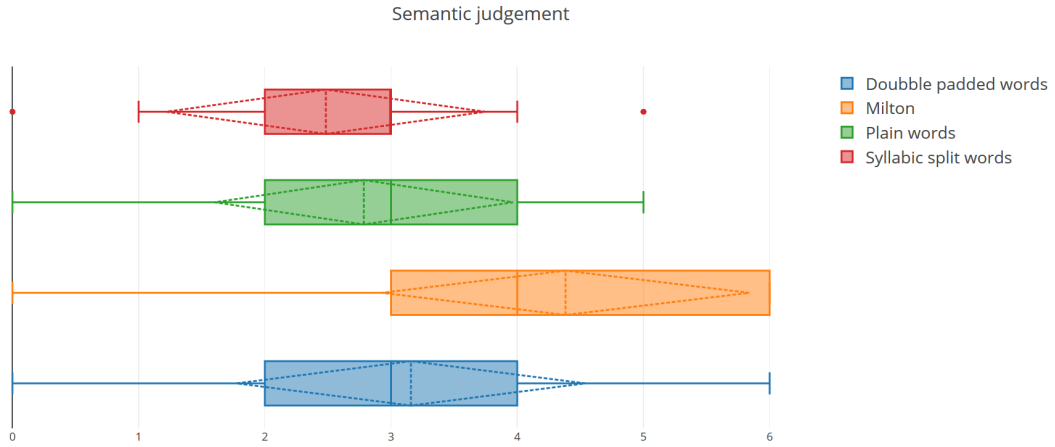

FIGURE 4.3: Boxplot of semantic judgments for different preprocessing methods

https://lantow.github.io/Data_and_plots/semantic_judgement1.html

Now, if this similarity is because the participants confused the two concepts or because they in some way are correlated is not obvious. The details are presented in the table below

	Mean	STD	Median
Syllabic	2.48	1.26	3
Plain	2.78	1.18	3
Double	3.16	1.38	3
Milton	4.38	1.36	4

Here we see that the semantic of the poetry generated with the syllabic language model has a slightly lower standard deviation and a slightly higher assessment with a median semantic of three. This could be interpreted as indicating that the model have captured the semantic slightly better than it did the grammar. Besides this both of the other corpora of generated poetry have the same judgment of around three.

Chapter 5

Conclusion and Future work

In conclusion . . . look to the future

5.1 Conclusion

The qualitative analysis seemed to indicate that the errors of the syllabic splitting algorithm was actually what was learned. Because of this the measurements of syllabic count made on the poetry generated with the syllabic split shows the model to have found the same distribution of syllables as Milton - and this is very much true, but with a slightly erroneous definition of syllables. This means that the modeled learned to consistently generate ten syllables per line - but only to the extent that the syllabic algorithm had defined the syllables correctly - which it hadn't. This is a good example of faulty assumptions being build into the model, and why we must be very careful with how the preprocessing is done. Also word embeddings with syllables seems to capture the semantic very poorly, and sometimes gibberish words are created. On a more uplifting note there were indications that the syllabic padding might worked as hypothesized. More specifically both the syllabic count and the semantic seemed to be better or as good as any of the other methods. However it wasn't shown that this would necessarily be the case after more iterations - however it indicates that by adding information to via the preprocessing might help the model converge faster - even if it converges faster to faulty assumptions about how to split syllables as in this case. As an overarching thought the poetry didn't seem to be as bad as I initially expected it to be

5.2 Future Work

I would have liked to experiment more rigorously with different setups. For instance a RNN as the generative part and the CNN might work for poetry as shown in (Yu et al., 2017). Another idea I would like to explore is one based on changing the philosophical view of words as clear cut categories. By laxing the assumption that language must be a series of discrete tokens, and instead see words as references fussy areas in the embedding space - there might be a way to get around the troublesome sampling function in the output of the generator. We thus might be able to make the entire GAN differentiable. This would mean a change from Shannon's language philosophy towards seeing words as on a continuum with many slight variations caught inside the probabilistic defined boundaries of word.

Chapter 6

APENDIX

6.1 Poetry split after the syllabic alogrithm

Syllabic split excerpts from Milton

With pur pose to re sign them in full time
Up to a bet ter Cov nant dis ci plind
From shad owie Types to Truth from Flesh to Spirit
From im po si tion of strict Laws to free
To mor tal men above which only shon
Fil ial obe di ence as a sac ri fice
Glad to be of ferd he at tends the will
Of his great Fa ther Ad mi ra tion seisd
To Judge ment he pro ceeded on th ac cusd
Ser pent though brute un able to trans ferre
The Guilt on him who made him in stru ment
Of mis chief and pol luted from the end

Syllabic split excerpts from standard processing

Our then my just ex age ing to that shape
Which as I namd speed we may easie to send
I know to him by this deep may have made
So felt us rose un vaild the wamer sense
Beds ing Vic to rie from every Trees Earth blow
Re bel lion in her flam ing sheer but makes
Be hold ing as the frown ing and his wings
Canst from the Book of God struc ture by Heavn
The gres sive but th un godly but to save
Rousd as the day though vi tive rest how stood
The Nor side high in bat ter to the Starrs
And wrought my throne If to his Sen ple view

Syllabic split excerpts from doubble padded processing

Moist nu tri ment or the shrill Ma tin light
ben ded Dol phins play with his great World to
To have ex press it best the An gelic An gel
Ere in your friendly East it by our
But so thou stoodst in Heavn and to be last

At my right hand and made them onely
A Heavn and dance in all a mo ment wide
With dread ful Ar tillery had com bus tion here
His boun tie Sov ran to spend him with rage
Or all his place by Heavn and fair the place
Of flight of Ag es at her grea ter large
And with parts dayes and moves from the Skie

Syllabic split excerpts from syllabic processing

Our then my just ex age ing to that shape
Which as I namd speed we may easie to send
I know to him by this deep may have made
So felt us rose un vaild the wamer sense
Beds ing Vic to rie from every Trees Earth blow
Re bel lion in her flam ing sheer but makes
Be hold ing as the frown ing and his wings
Canst from the Book of God struc ture by Heavn
Our then my just ex age ing to that shape
Which as I namd speed we may easie to send
I know to him by this deep may have made
So felt us rose un vaild the wamer sense

6.2 Some graphs from the GAN

Figure 6.1 show how well the language models was learned with different prepossessing methods.

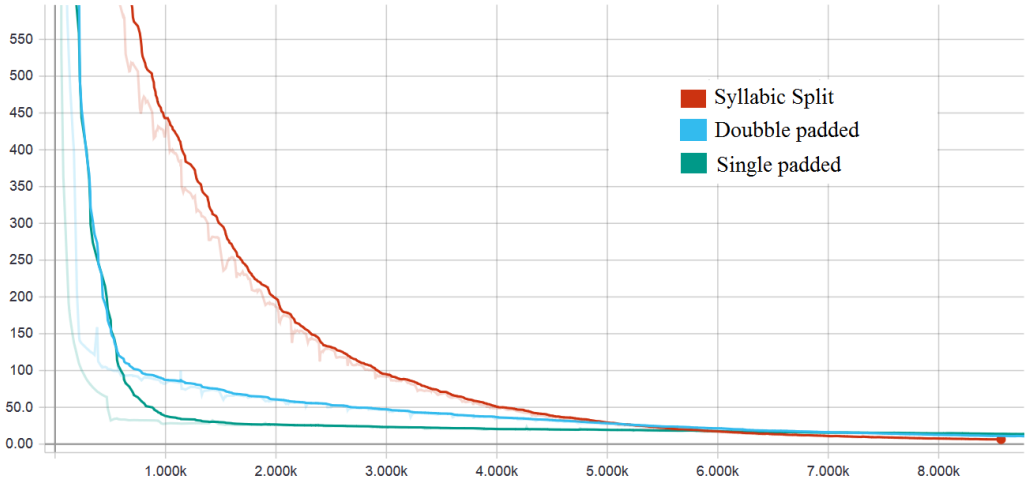

FIGURE 6.1: A comparison between the development in perplexity across language models generated with different prepossessing methods

We can here see that the perplexity of model trained on syllables fell with a slower rate than the models using words. We also see that the double padding seem to have added a bit of noise (although not as much as the syllables) as it converged slightly slower than the words alone.

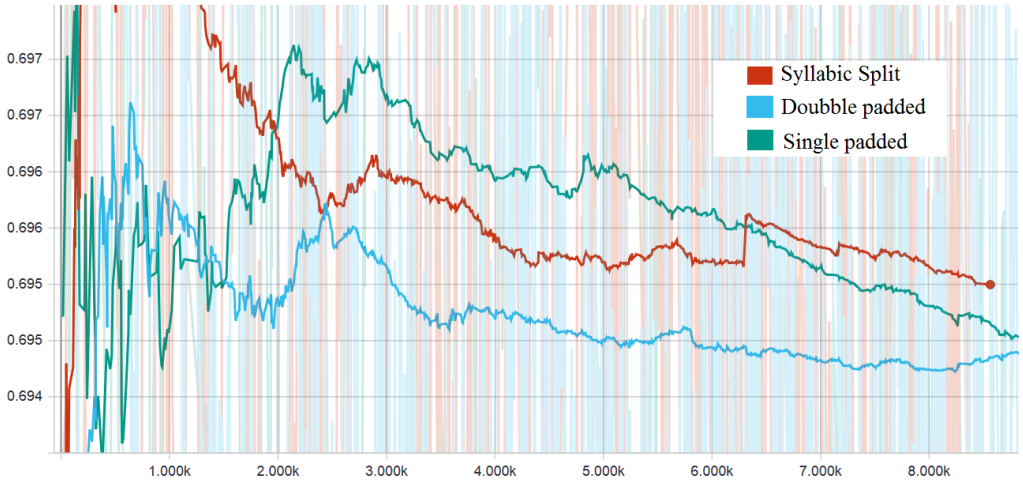

FIGURE 6.2: A comparison between the development in the discriminators arability to tell the generated poetry from the real across language models generated with different prepossessing methods

As was mentioned in previous sections the GAN setup suffers from high variance in the discriminator outputs. The plots in figure in 6.2 shows the discriminator outputs when smoothed with 99.99 percent. We see that the smoothed plots only changes very slightly on the Y-axis. The discriminator outputs show how well the generator has learned to fool the discriminator into seeing its outputs as real poetry. This should in theory converge at 0.5 if the training went on long enough.

Bibliography

Arjovsky, M., Chintala, S., & Bottou, L. (2017). Wasserstein gan. *arXiv preprint arXiv:1701.07875*.

Bachman, P. & Precup, D. (2015). Data generation as sequential decision making. In *Advances in neural information processing systems* (pp. 3249–3257).

Bahdanau, D., Brakel, P., Xu, K., Goyal, A., Lowe, R., Pineau, J., ... Bengio, Y. (2016). An actor-critic algorithm for sequence prediction. *arXiv preprint arXiv:1607.07086*.

Baldi, P. & Sadowski, P. J. (2013). Understanding dropout. In *Advances in neural information processing systems* (pp. 2814–2822).

Bengio, S. & Bengio, Y. (2000). Taking on the curse of dimensionality in joint distributions using neural networks. *IEEE Transactions on Neural Networks*, *11*(3), 550–557.

Bengio, S., Vinyals, O., Jaitly, N., & Shazeer, N. (2015). Scheduled sampling for sequence prediction with recurrent neural networks. In *Advances in neural information processing systems* (pp. 1171–1179).

Bengio, Y. & Bengio, S. (2000). Modeling high-dimensional discrete data with multi-layer neural networks. In *Advances in neural information processing systems* (pp. 400–406).

Bengio, Y., Ducharme, R., Vincent, P., & Jauvin, C. (2003). A neural probabilistic language model. *Journal of machine learning research*, *3*(Feb), 1137–1155.

Bengio, Y., Léonard, N., & Courville, A. (2013). Estimating or propagating gradients through stochastic neurons for conditional computation. *arXiv preprint arXiv:1308.3432*.

Bengio, Y., Simard, P., & Frasconi, P. (1994). Learning long-term dependencies with gradient descent is difficult. *IEEE transactions on neural networks*, *5*(2), 157–166.

Bengio, Y., Yao, L., Alain, G., & Vincent, P. (2013). Generalized denoising auto-encoders as generative models. In *Advances in neural information processing systems* (pp. 899–907).

Blei, D. M., Ng, A. Y., & Jordan, M. I. (2003). Latent dirichlet allocation. *Journal of machine Learning research*, *3*(Jan), 993–1022.

Bod, R. (1998). Beyond grammar: an experience-based theory of language.

Brill, E. (1995). Transformation-based error-driven learning and natural language processing: a case study in part-of-speech tagging. *Computational linguistics*, *21*(4), 543–565.

Browne, C. B., Powley, E., Whitehouse, D., Lucas, S. M., Cowling, P. I., Rohlfshagen, P., ... Colton, S. (2012). A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in games*, *4*(1), 1–43.

Che, T., Li, Y., Zhang, R., Hjelm, R. D., Li, W., Song, Y., & Bengio, Y. (2017). Maximum-likelihood augmented discrete generative adversarial networks. *arXiv preprint arXiv:1702.07983*.

Cho, K., Van Merriënboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., & Bengio, Y. (2014). Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*.

Chung, J., Gulcehre, C., Cho, K., & Bengio, Y. (2014). Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv preprint arXiv:1412.3555*.

Cios, K. J. (2018). Deep neural networks—a brief history. In *Advances in data analysis with computational intelligence methods* (pp. 183–200). Springer.

Crain, S. et al. (2010). What are core linguistic properties?

Dahl, G. E., Adams, R. P., & Larochelle, H. (2012). Training restricted boltzmann machines on word observations. *arXiv preprint arXiv:1202.5695*.

Dai, B., Fidler, S., Urtasun, R., & Lin, D. (2017). Towards diverse and natural image descriptions via a conditional gan. *arXiv preprint arXiv:1703.06029*.

Daumé, H., Langford, J., & Marcu, D. (2009). Search-based structured prediction. *Machine learning*, *75*(3), 297–325.

De Boer, P.-T., Kroese, D. P., Mannor, S., & Rubinstein, R. Y. (2005). A tutorial on the cross-entropy method. *Annals of operations research*, *134*(1), 19–67.

De Saussure, F. (2011). *Course in general linguistics*. Columbia University Press.

Dietterich, T. G. (2002). Machine learning for sequential data: a review. In *Joint iapr international workshops on statistical techniques in pattern recognition (spr) and structural and syntactic pattern recognition (sspr)* (pp. 15–30). Springer.

Doersch, C. (2016). Tutorial on variational autoencoders. *arXiv preprint arXiv:1606.05908*.

Fahlman, S. E. et al. (1988). An empirical study of learning speed in back-propagation networks.

Fedus, W., Goodfellow, I., & Dai, A. M. (2018). Maskgan: better text generation via filling in the _. *arXiv preprint arXiv:1801.07736*.

Gers, F. A. & Schmidhuber, E. (2001). Lstm recurrent networks learn simple context-free and context-sensitive languages. *IEEE Transactions on Neural Networks*, *12*(6), 1333–1340.

Gers, F. A. & Schmidhuber, J. (2000). Recurrent nets that time and count. In *Proceedings of the ieee-inns-enns international joint conference on neural networks. ijcnn 2000. neural computing: new challenges and perspectives for the new millennium* (Vol. 3, pp. 189–194). IEEE.

Gers, F. A., Schmidhuber, J., & Cummins, F. (1999). Learning to forget continual prediction with lstm.

Glorot, X. & Bengio, Y. (2010). Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics* (pp. 249–256).

Glorot, X., Bordes, A., & Bengio, Y. (2011). Domain adaptation for large-scale sentiment classification: a deep learning approach. In *Proceedings of the 28th international conference on machine learning (icml-11)* (pp. 513–520).

Goldberg, Y. (2016). A primer on neural network models for natural language processing. *Journal of Artificial Intelligence Research*, *57*, 345–420.

Goldberg, Y. & Levy, O. (2014). Word2vec explained: deriving mikolov et al.'s negative-sampling word-embedding method. *arXiv preprint arXiv:1402.3722*.

Goodfellow, I., Bengio, Y., Courville, A., & Bengio, Y. (2016). *Deep learning*. MIT press Cambridge.

Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., ... Bengio, Y. (2014). Generative adversarial nets. In *Advances in neural information processing systems* (pp. 2672–2680).

Gulrajani, I., Ahmed, F., Arjovsky, M., Dumoulin, V., & Courville, A. C. (2017). Improved training of wasserstein gans. In *Advances in neural information processing systems* (pp. 5767–5777).

Guo, H. (2015). Generating text with deep reinforcement learning. *CoRR*, *abs/1510.09202*. arXiv: 1510.09202. Retrieved from http://arxiv.org/abs/1510.09202

Harvey, M. L. (1996). *Iambic pentameter from shakespeare to browning: a study in generative metrics*. Mellen.

Hebb, D. (1949). The organization of behaviour (hoboken, nj. Wiley.

Hecht-Nielsen, R. (1992). Theory of the backpropagation neural network. In *Neural networks for perception* (pp. 65–93). Elsevier.

Hirjee, H. (2010, October). Rhyme, rhythm, and rhubarb using probabilistic methods to analyze hip hop, poetry, and misheard lyrics.

Hjelm, R. D., Jacob, A. P., Che, T., Trischler, A., Cho, K., & Bengio, Y. (2017). Boundary-seeking generative adversarial networks. *arXiv preprint arXiv:1702.08431*.

Hochreiter, S. (1998). The vanishing gradient problem during learning recurrent neural nets and problem solutions. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, *6*(02), 107–116.

Hochreiter, S. & Schmidhuber, J. (1997). Long short-term memory. *Neural computation*, *9*(8), 1735–1780.

Huszár, F. (2015). How (not) to train your generative model: scheduled sampling, likelihood, adversary? *arXiv preprint arXiv:1511.05101*.

Jang, E., Gu, S., & Poole, B. (2016). Categorical reparameterization with gumbel-softmax. *arXiv preprint arXiv:1611.01144*.

Jarrett, K., Kavukcuoglu, K., LeCun, Y., et al. (2009). What is the best multi-stage architecture for object recognition? In *Computer vision, 2009 ieee 12th international conference on* (pp. 2146–2153). IEEE.

Kenstowicz, M. & Kisseberth, C. (2014). *Generative phonology: description and theory*. Academic Press.

Keskar, N. S., Mudigere, D., Nocedal, J., Smelyanskiy, M., & Tang, P. T. P. (2016). On large-batch training for deep learning: generalization gap and sharp minima. *arXiv preprint arXiv:1609.04836*.

Kneser, R. & Ney, H. (1995). Improved backing-off for m-gram language modeling. In *Icassp* (Vol. 1, 181e4).

Larsen, A. B. L., Sønderby, S. K., Larochelle, H., & Winther, O. (2015). Autoencoding beyond pixels using a learned similarity metric. *arXiv preprint arXiv:1512.09300*.

LeCun, Y., Bottou, L., Bengio, Y., & Haffner, P. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, *86*(11), 2278–2324.

Lee, L. (2000). Measures of distributional similarity. *arXiv preprint cs/0001012*.

Li, J., Monroe, W., Shi, T., Jean, S., Ritter, A., & Jurafsky, D. (2017). Adversarial learning for neural dialogue generation. *arXiv preprint arXiv:1701.06547*.

Liang, F. & Breitenlohner, P. (1991). Pattern generation program for the tex82 hyphenator. *Electronic documentation of PATGEN program version*, *2*.

Liang, F. M. (1983). *Word hy-phen-a-tion by com-put-er*. Calif. Univ. Stanford. Comput. Sci. Dept.

Lin, C.-Y. (2004). Rouge: a package for automatic evaluation of summaries. *Text Summarization Branches Out*.

Maltzahn, N. V. (1994). John t. shawcross. john milton: the self and the world. (studies in the english renaissance.) lexington, ky.: university press of kentucky. 1993. pp. vii, 358. 59.00. *Albion*, *26*(1), 138–140. doi:10.2307/4052118

Manurung, H., Ritchie, G., & Thompson, H. (2000). *Towards a computational model of poetry generation*. The University of Edinburgh.

McCulloch, W. S. & Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, *5*(4), 115–133.

Mikolov, T. [Tomas], Chen, K., Corrado, G., & Dean, J. (2013). Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*.

Mikolov, T. [Tomáš], Karafiát, M., Burget, L., Černocky, J., & Khudanpur, S. (2010). Recurrent neural network based language model. In *Eleventh annual conference of the international speech communication association*.

Mikolov, T. [Tomas], Sutskever, I., Chen, K., Corrado, G. S., & Dean, J. (2013). Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems* (pp. 3111–3119).

Molchanov, D., Ashukha, A., & Vetrov, D. (2017). Variational dropout sparsifies deep neural networks. *arXiv preprint arXiv:1701.05369*.

Morin, F. & Bengio, Y. (2005). Hierarchical probabilistic neural network language model. In *Aistats* (Vol. 5, pp. 246–252). Citeseer.

Naili, M., Chaibi, A. H., & Ghezala, H. H. B. (2017). Comparative study of word embedding methods in topic segmentation. *Procedia Computer Science*, *112*, 340–349.

Nivre, J. (2001). On statistical methods in natural language processing. In *Proceedings of the 13th nordic conference of computational linguistics (nodalida 2001)*.

Papineni, K., Roukos, S., Ward, T., & Zhu, W.-J. (2002). Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting on association for computational linguistics* (pp. 311–318). Association for Computational Linguistics.

Peirce, C. S. (1868). Questions concerning certain faculties claimed for man. *The Journal of Speculative Philosophy*, *2*(2), 103–114.

Pennington, J., Socher, R., & Manning, C. (2014). Glove: global vectors for word representation. In *Proceedings of the 2014 conference on empirical methods in natural language processing (emnlp)* (pp. 1532–1543).

Pineda, F. J. (1987). Generalization of back-propagation to recurrent neural networks. *Physical review letters*, *59*(19), 2229.

Putnam, H. (1974). Meaning and reference. *The journal of philosophy*, *70*(19), 699–711.

Qiu, M., Li, F.-L., Wang, S., Gao, X., Chen, Y., Zhao, W., ... Chu, W. (2017). Alime chat: a sequence to sequence and rerank based chatbot engine. In *Proceedings of the 55th annual meeting of the association for computational linguistics (volume 2: short papers)* (Vol. 2, pp. 498–503).

Rajeswar, S., Subramanian, S., Dutil, F., Pal, C., & Courville, A. (2017). Adversarial generation of natural language. *arXiv preprint arXiv:1705.10929*.

Ranzato, M., Chopra, S., Auli, M., & Zaremba, W. (2015). Sequence level training with recurrent neural networks. *arXiv preprint arXiv:1511.06732*.

Reiter, E. & Dale, R. (2000). *Building natural language generation systems*. Cambridge university press.

Rosenblatt, F. (1957). *The perceptron, a perceiving and recognizing automaton project para*. Cornell Aeronautical Laboratory.

Rosenfeld, R. (1996). A maximum entropy approach to adaptive statistical language modeling.

Rosti, A.-V. I., Zhang, B., Matsoukas, S., & Schwartz, R. (2011). Expected bleu training for graphs: bbn system description for wmt11 system combination task. In *Proceedings of the sixth workshop on statistical machine translation* (pp. 159–165). Association for Computational Linguistics.

Shannon, C. E. (2001). A mathematical theory of communication. *ACM SIGMOBILE mobile computing and communications review*, *5*(1), 3–55.

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., Van Den Driessche, G., ... Lanctot, M., et al. (2016). Mastering the game of go with deep neural networks and tree search. *nature*, *529*(7587), 484.

Smith, L. N. (2018). A disciplined approach to neural network hyper-parameters: part 1–learning rate, batch size, momentum, and weight decay. *arXiv preprint arXiv:1803.09820*.

Snell, A. L. F. (1918). *Pause: a study of its nature and its rhythmical function in verse, especially blank verse*. Ann Arbor Press.

Sønderby, C. K., Raiko, T., Maaløe, L., Sønderby, S. K., & Winther, O. (2016). How to train deep variational autoencoders and probabilistic ladder networks. *arXiv preprint arXiv:1602.02282*.

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). Dropout: a simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, *15*(1), 1929–1958.

Steinberg, D. D. (1971). *Semantics: an interdisciplinary reader in philosophy, linguistics and psychology*. CUP Archive.

Sutskever, I., Vinyals, O., & Le, Q. V. (2014). Sequence to sequence learning with neural networks. In *Advances in neural information processing systems* (pp. 3104–3112).

Sutton, R. S., McAllester, D. A., Singh, S. P., & Mansour, Y. (2000). Policy gradient methods for reinforcement learning with function approximation. In *Advances in neural information processing systems* (pp. 1057–1063).

Tulyakov, S., Liu, M.-Y., Yang, X., & Kautz, J. (2017). Mocogan: decomposing motion and content for video generation. *arXiv preprint arXiv:1707.04993*.

Turing, A. M. (1950). Computing machinery and intelligence. *Mind*, *59*(236), 433.

Verse, B. & Snell, A. L. (1918). An objective study of syllabic quantity in english verse. *Publications of the Modern Language Association of America*, 396–408.

Vinyals, O., Bengio, S., & Kudlur, M. (2015). Order matters: sequence to sequence for sets. *arXiv preprint arXiv:1511.06391*.

Werbos, P. J. (1990). Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, *78*(10), 1550–1560.

Wiseman, S. & Rush, A. M. (2016). Sequence-to-sequence learning as beam-search optimization. *arXiv preprint arXiv:1606.02960*.

Yi, X., Li, R., & Sun, M. (2017). Generating chinese classical poems with rnn encoder-decoder. In *Chinese computational linguistics and natural language processing based on naturally annotated big data* (pp. 211–223). Springer.

Yu, L., Zhang, W., Wang, J., & Yu, Y. (2017). Seqgan: sequence generative adversarial nets with policy gradient. In *Aaai* (pp. 2852–2858).

Zhang, X. & Lapata, M. (2014). Chinese poetry generation with recurrent neural networks. In *Proceedings of the 2014 conference on empirical methods in natural language processing (emnlp)* (pp. 670–680).

Zhang, Y., Gan, Z., & Carin, L. (2016). Generating text via adversarial training. In *Nips workshop on adversarial training* (Vol. 21).