Department of Linguistics
University of Potsdam

BACHELOR THESIS

Generating Poetry with Recurrent Neural Networks

by

Aline Johanna Castendiek

in Partial Fulfillment of the Requirements for the Degree of

Bachelor of Science in Computational Linguistics

November 15, 2018

Thesis Supervisors:

Dr.-Ing. Sebastian Stober
Dr. Thomas Hanneforth

Abstract

Generating Poetry with Recurrent Neural Networks

Aline Johanna Castendiek

Automatic generation of poetry is a very interesting, yet complex research topic. Poems can be considered extremely well-defined messages that involve several levels of language simultaneously. This makes them particularly interesting from a linguistic point of view. In my work, I present an approach on generating limericks that involves training a character-level recurrent neural network with LSTM units. I also propose a method to evaluate the quality of the generated poems automatically based on their phonetic representations. The evaluation criteria incorporate several aspects of syntactic and phonetic well-formedness that apply to limericks, such as the number of verses, the number of syllables in a verse, the metric pattern, and the rhyme scheme. I found that the model is able to reliably learn all of the defined criteria at low temperatures, except for the rhyme scheme. I assume that this is due to the irregularities of English spelling that prevent a character-level model from inferring phonetic similarity in words that have strongly deviating orthographic representations.

Zusammenfassung

Die automatische Generierung von Poesie ist ein sehr interessantes, jedoch komplexes Forschungsthema. Gedichte lassen sich als extrem wohldefinierte Nachrichten betrachten, die mehrere Sprachebenen gleichzeitig miteinbeziehen. Dies macht sie vor allem aus linguistischer Perspektive interessant. In meiner Arbeit stelle ich einen Ansatz zur Generierung von Limericks vor, der auf einem rekurrenten neuronalen Netz mit LSTM-Units auf Buchstabenebene basiert. Des Weiteren entwickle ich eine Methode, um die Qualität der generierten Gedichte auf Basis ihrer phonetischen Repräsentationen automatisch zu evaluieren. Die Evaluation beinhaltet mehrere Aspekte syntaktischer und phonetischer Wohlgeformtheit im Bezug auf Limericks wie die Anzahl der Verse, die Anzahl der Silben pro Vers, die Metrik sowie das Reimschema. Es hat sich gezeigt, dass das Modell bei niedrigen Temperaturen alle definierten Kriterien bis auf das Reimschema verlässlich lernen kann. Dies liegt vermutlich an den Unregelmäßigkeiten der englischen Rechtschreibung, die es einem Modell auf Buchstabenebene nicht ermöglichen, die phonetische Ähnlichkeit von Wörtern mit stark voneinander abweichenden orthographischen Repräsentationen abzuleiten.

Declaration of Authorship

I certify that the work presented here is, to the best of my knowledge and belief, original and the result of my own investigations, except as acknowledged, and has not been submitted, either in part or whole, for another degree at this or any other university.

_____

(Signature)

Berlin, August 9, 2018

Contents

List of Figures

List of Tables

List of Algorithms

Introduction

In the last decade, neural networks and deep learning technology have become an integral part of computer science and machine learning research. Although deep learning seems to be a new, emerging field, the basic idea dates back to the 1940s and has been developed in the context of cybernetics and biological learning (McCulloch and Pitts, 1943; Hebb, 1949). The development of fundamental neural network architectures can be traced back to the 1950s and 1960s as well: With the implementation of the perceptron (Rosenblatt, 1958, 1962), a single trainable neuron capable of learning its own weights was introduced. Furthermore, the adaptive linear element (ADALINE) model (Widrow and Hoff, 1960) presented an algorithm for learning its weights very similar to stochastic gradient descent. To this day, the combination of stochastic gradient descent with backpropagation remains the predominant training algorithm for deep neural network architectures.

Knowing that these basic ideas have been around for decades, one might wonder why neural networks have only recently received this kind of extensive attention. The current research focus on neural network architectures and their success is mainly due to two technological advancements: Firstly, the amount of available training data has increased significantly. Nowadays it is very easy to automatically extract large amounts of data from the Internet and build a dataset from it. Since most machine learning algorithms rely on statistical estimation, this improvement helps these models to better generalize to new examples based on their advanced training datasets. And secondly, computational resources such as hardware and computing power have increased. We have computers available now that are much faster, and especially by using graphical processing units (GPUs), we can parallelize the computations for neural networks efficiently. In terms of model complexity, this effectively means that we can build larger neural networks with more neurons and more connections between these neurons as well.

Deep learning research across various fields also gives rise to new learning algorithms and architectures for neural networks that further improve their performance. If we take this into consideration, and also presume that the trend of increasing datasets and computational resources will continue, deep neural networks are indeed a very promising research area.

But besides their overall success, there is another reason that makes neural networks particularly interesting for machine learning researchers: In many cases, they make the time-consuming task of feature engineering unnecessary. Traditional machine learning techniques often required researchers to manually design features for each specific task. In the field of computational linguistics and natural language processing (NLP), this meant that it was necessary to hard-code knowledge about the world or grammatical restrictions in formal languages, for instance, which is a very tedious and costly task. But with the rise of neural networks, the burden of feature engineering has been eased, as it is now possible to use raw data instead and let the neural network extract the features automatically as it learns.

There are many NLP applications in which deep neural networks have been shown to outperform traditional approaches. Furthermore, there is a trend of improved neural network architectures outperforming previous approaches on a regular basis. This trend can be observed in areas such as speech recognition (Graves et al., 2013), machine translation (Cho et al., 2014), and text classification (Lai et al., 2015). However, the focus of this thesis lies on the task of natural language generation (NLG), or, more precisely, poetry generation, and the underlying architectures that are essential to tackle these tasks with neural networks.

Before I explain these architectures in detail, I will give a brief overview of motivational factors for research in poetry generation and shortly summarize previous approaches in this field as well as their limitations compared to deep neural networks.

Introduction to Poetry Generation

2.1 Motivation

Since most tasks in natural language generation have a clearly defined goal and real-world application, such as translating texts or creating summaries, it is not immediately evident why one should pursue the topic of poetry generation. The field seems to lack a predefined, measurable goal and a particularly useful application in real life. However, the motivation stems from an interesting question that arises in this context: Whether it is possible to teach computers to be creative.

AI systems that attempt to simulate artificial creativity have not only been implemented in the area of language generation, but also across various other domains: They have been used to generate paintings by reproducing different artistic styles (Gatys et al., 2015; Elgammal et al., 2017) or to compose folk (Sturm et al., 2015) and jazz music (Bickerman et al., 2010). But the topic of computational creativity remains a very challenging task, mainly because there is no clear definition on what creativity actually is. It appears to be a very vague concept that cannot be defined explicitly. This makes it impossible at the moment to create an all-encompassing model that is able to replicate human creativity and to evaluate the output of such a system objectively.

In general, NLG can already be seen as a well-established sub-field of computational linguistics and artificial intelligence. Nevertheless, the most successful and promising results in this field were achieved by generating texts that are generally not perceived as particularly creative – for example biographies, weather forecasts, or summaries of databases. In these kinds of text, neither creativity nor a high level of abstraction seems to be relevant. Naturally, some restrictions such as grammatical and semantic correctness and coherence apply here as well. But when it comes to automatically generating literature, many additional criteria need to be taken into consideration: Is there a theme, message, or tone present in the text? Is the text imaginative? Do rhetorical terms occur? Are there metaphors, analogies, puns, jokes, symbolism, irony, or sarcasm present?

In the context of poetry generation, even more criteria apply that we need to take into consideration if we want to produce aesthetically pleasing results: Consistent metrical patterns, rhyme schemes, alliterations, assonance, and many other poetic devices and terms of figurative language. Basically, it can be said that poems impose more restrictions in terms of semantics, syntax, phonetics and lexical choice than any other form of text. Since poetry generation involves several levels of language simultaneously, it is a very interesting, yet complex research topic. Although at present, there exists no automatic evaluation mechanism yet that could cover all these relevant aspects, there have been different approaches on how to automatically generate and (partially) evaluate poetry.

In the following, I will give a brief summary of earlier approaches on automatic poetry generation and outline their potential restrictions in terms of modeling capacity. After introducing neural network architectures and their terminology, I will then provide an overview of current, more promising approaches based on neural networks.

2.2 Early Approaches

Previous approaches on the topic of automatic poetry composition can be roughly divided into three basic categories: Template-based generation that places a strong focus on grammaticality, case-based reasoning generation that uses prespecified restrictions and inference rules, and generation based on evolutionary algorithms that tries to model the successive writing process of a human poet.

2.2.1 Template-based Generation

The Stochastische Texte system (Lutz, 1959) is commonly considered the first computational poetry generator. It consisted of a very small lexicon of sixteen subjects and sixteen predicates from Kafka's *Das Schloß*. These words were randomly fitted into a predefined grammatical template to create a new text. Over time, several similar systems that made use of sentence templates were implemented, such as Masterman's computerized haikus (Masterman, 1971) or the PROSE system (Hartman, 1996).

According to Manurung (2003), these template-based systems often made use of particular heuristics and tricks to give the appearance of coherence and poeticness: For instance, by using words assigned to "emotional categories" such as philosophy, nature, or love, by choosing lexical items repetitively to give a false sense of coherence, or by constructing complicated sentence templates with only a few holes.

Although template-generated poems are often perceived as grammatical by the reader, the resulting output texts of such systems are very limited in terms of creativity and expressiveness. It can be hard to distinguish such a poem from one written by an actual human poet – but this is not the result of an elaborate algorithm that can capture or simulate human creativity. Instead, this accomplishment can be attributed to the human template writers.

2.2.2 Case-based Reasoning

The WASP system (Gervás, 2000) can be seen as one of the first serious attempts on automatic poetry generation. It is a reasoning rule-based system that takes as input a set of words and verse patterns and outputs a generated set of verses. The generation process is guided by a set of construction heuristics that are obtained from formal literature on Spanish poetry. These heuristics include strategies for avoiding repetitions (e.g. simple random combination, eliminating used words) and methods for validating the current draft of a verse during generation (e.g. counting syllables, checking the position of stressed syllables). If the predefined restrictions in the current draft are not met, the verse is rejected. A corresponding study aims to investigate the relative importance of initial vocabulary, word choice, choice of verse patterns, and construction heuristics with respect to the acceptability of resulting verses. This is accomplished by extracting numerical parameters from the test cases and looking for significant correlations in the data. Whether or not a poem is deemed acceptable is decided by a team of volunteers who apply manual evaluation. While the system takes into account several metrical aspects such as syllable count, stress position, vowel position, and rhyme, the goal is to use these aspects as restrictions in the generation process. The focus lies on collecting more information about the effectiveness of different generation strategies and not on a differentiated method for evaluating the model output. It is also noteworthy that in Spanish, it is very easy to obtain the phonetic representations algorithmically from the written word. Most single letters have a particular pronunciation, not dependent on where they appear in a text. For English texts, however, this task proves to be more of a challenge because the pronunciation of a letter varies a lot depending on the letters that appear before and afterwards.

Over the years, several modifications and improvements of the original WASP system were implemented. One of these systems is ASPERA (Gervás, 2001), an application for composing Spanish poetry semiautomatically and interactively. The system incorporates most of the original heuristics of WASP, but applies an updated composing mechanism: It requires the user to input an intended message and specification of the type of poem and selects the most appropriate metric structure by applying a knowledge-based pre-processor. The given message is then translated into a poem of the defined specifications and constraints. In this manner, a draft poem is generated that needs to be validated or modified by the user, and afterwards, the system's database is updated with the validated information.

Another approach very similar to these two models is the COLIBRI model (Díaz-Agudo et al., 2002): It introduces an application-specific ontology called CBROnto. This framework incorporates explicit representations and general knowledge about the domain of case-based reasoning, which further improves the model's inference power.

While the WASP, ASPERA, and COLIBRI system aimed to generate Spanish poetry exclusively, such rule-based approaches have later also been transferred to the generation of haikus, a traditional Japanese poetic form that conveys seasonal or nature related themes (Tosa et al., 2008; Netzer et al., 2009).

2.2.3 Evolutionary Algorithms

Another approach on poetry generation is based on evolutionary algorithms. The MCGONAGALL system (Manurung, 2003) considers the process of generating poetry as a state space search problem. The goal state to be achieved is a text that satisfies the three properties of meaningfulness, grammaticality, and poeticness. To reach this goal state, stochastic hill-climbing search is applied, where a state in the search space is defined as a possible text, incorporating underlying semantic and phonetic representations. The model also uses the Lexicalized Tree Adjoining Grammar (LTAG) formalism to implement grammatical restrictions. The evolutionary algorithm consists of two stages, evaluation and evolution. During the evaluation process, a set of generated evolutionary individuals (represented as drafts of poems and their corresponding feature representations) is scored based on aspects such as phonetic patterns and semantics. During the evolution stage, the highest scored individuals are selected and mutations on the different levels of representations are applied, in hope of producing better versions of the poem. This approach tries to mimic the actual process of human poem composition, assuming that it generally involves drafting different versions of a poem until a satisfying result is achieved. It furthermore aims to model several aspects at once: Semantics, grammaticality, and poeticness in terms of phonetic restrictions such as rhythmic patterns and rhyme. In contrast, most previous approaches only focused on one particular aspect: Template-driven systems mainly concentrated on grammaticality, while the case-based reasoning approaches tried to model poeticness with regard to metrical restrictions.

2.3 Challenges

As we have seen, various challenges arise in the context of poetry generation. If we want to build a system capable of producing poems that are aesthetically pleasing to the human reader, we need to take several aspects into account. Most importantly, the criteria of grammaticality, semantics and poeticness must be satisfied.

Grammatical analysis can be seen as well-established sub-field of computational linguistics, and there exist several state-of-the-art methods for modeling and evaluating grammaticality. Yet, evaluating semantic properties seems to be difficult, especially in the context of poetry: A poem might sometimes appear abstract and cryptic, but based on these exact properties it might be perceived as particularly poetic by a human reader. In this context, the term poeticness remains a vague, subjective concept that seems to lack a general definition. However, we can conclude that poetic well-formedness at least requires certain metrical constraints such as a consistent rhyme scheme and stress pattern, a certain number of lines per poem, and a specific number of syllables per verse, although these requirements vary depending on the desired poetic form.

With the previous approaches on this topic, this leaves us with the time-consuming task of handcrafting rules and restrictions for each of these features into our system; and for each language and each poetic form, we would need to design a different set of rules and restrictions. It is indeed not very tempting to just be able to reproduce one specific poem type in one particular language.

Ideally, we want to design a system that takes a dataset of poems of a particular poetic form and language as input, automatically learns the specific properties for this poetic form and language and outputs poems that satisfy these constraints. Since deep neural networks have proven effective in learning feature representations through training and are currently used in various natural language generation applications, they seem to be a very good fit for this task. Consequently, if we can successfully apply the generation process to neural networks, only the task of qualitatively evaluating the output remains (which still continues to be a difficult task for NLG in general).

In the following chapter, I will explain how neural networks work in general and then focus on two specific architectures, recurrent neural networks (RNNs) and long short term memory units (LSTMs) which have proven especially useful for this kind of task. Then, I will give a brief overview of ongoing research in the area of poetry generation with neural networks.

Neural Networks

3.1 Feedforward Neural Networks

Feedforward neural networks or *multilayer perceptrons (MLPs)*, sometimes also referred to as *vanilla neural networks*, can be considered one of the most basic deep neural network architectures. I will therefore use them to introduce general terminology and to explain how neural networks work in principle.

If we compare a feedforward network classifier to a conventional statistical classifier, their goal is very similar: They both aim to approximate some function f^* that maps input values to output values. More precisely, this means that $y = f^*(x)$ for an input x and an output category y. The feedforward network defines a mapping $y = f(x;\theta)$ with parameters θ, and the parameter values that correspond to the best function approximation are learned by the model during the training process.

The term *feedforward* stems from the direction of the information flow: The model starts from input x, performs several computations based on the definition of f and computes an output y. Compared to other neural network architectures, there are no feedback connections or loops in which the model outputs are fed back into the model itself.

A feedforward neural network can be represented as directed acyclic graph, as illustrated in figure 3.1. The network consists of an input layer, several hidden layers and an output layer. In the input layer, each node corresponds to an input value $x_1, ..., x_n$, when the training example is $x = x_1, ..., x_n$. Each node is called a neuron and can be seen as the basic processing unit of a neural network. The *depth* of the model is defined as the number of hidden layers, including the output layer. The number of neurons in each hidden layer, also referred to as the dimensionality of a hidden layer, indicates the *width* of the model.

Accordingly, the network illustrated in figure 3.1 depicts a feedforward neural network with three input values x_1, x_2 and x_3, two hidden layers with a dimensionality of four, and an output layer consisting of two output units. Hence, the model has a depth of three and a width of four.

For a better understanding, the feedforward network can also be explained in terms of function composition: If we represent the first layer as $f^{(1)}$, the second layer as $f^{(2)}$ and the third layer as $f^{(3)}$, we can represent the model function by composing these functions so that $f(x) = f^{(3)}(f^{(2)}(f^{(1)}(x)))$.

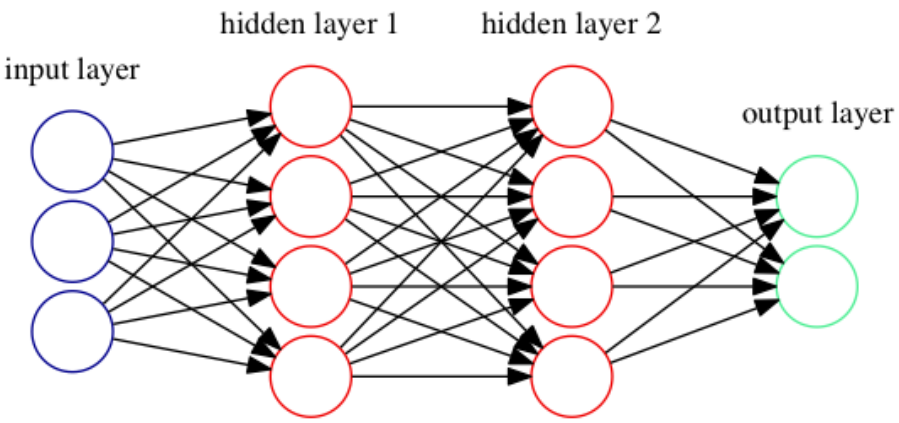


Figure 3.1: Example of a fully-connected feedforward neural network.

Each neuron in the hidden layer computes an output value that depends on the following properties: The node values from the previous layer that are connected to it, the associated weights (represented as edges in the graph representation), a bias term, and an activation function. Hence, between a layer of dimension i and a layer of dimension j, we have a weight matrix of size $i * j$ and a single bias term in a fully-connected network. All these weight matrices and biases combined are part of our model parameters θ. Accordingly, the network adjusts the weights and biases during training to produce the desired output.

While the behavior of the output layer is specified directly by the training examples – it must produce a value that is close to y for a given training example x – this is not the case for hidden layers. Instead, the learning algorithm has to decide how to use these layers to provide the best approximation of f^*. Based on this behavior, these layers are called *hidden*. We can also say that neural networks perform feature learning because they learn representations of their input at each hidden layer and subsequently use these representations for predictions in the output layer.

To understand how a network manages to learn its parameters θ involves knowledge about several components that play a role in this training process: Activation functions, a cost function, forward propagation to compute the cost, and gradient descent with backpropagation as actual mechanism to learn the optimal weights. In the following sections, I will introduce these concepts, and in the end we will have obtained the knowledge we need to train a feedforward neural network and to transfer this knowledge to train other network architectures.

3.1.1 Activation Functions

The *activation function* of a neuron computes the output of that neuron given its inputs. Its main purpose is to introduce non-linearity into the output of a neuron. This is a very important task since most real-world data is not linear, and we want our neurons to be able to learn these non-linear representations. By combining the activation functions of many hidden layers, the input of the model is transformed repeatedly and the network is able to model complex non-linear functions.

The choice of an appropriate activation function is part of the design decisions that need to be made when implementing a neural network. For modern neural networks, the recommended default is the *rectified linear unit (ReLU)* activation function (Goodfellow et al., 2016, ch. 6.1). It is defined as:

$$f(x) = max\{0, x\} \tag{3.1}$$

and plotted in figure 3.2a. This function takes a real-valued input and replaces negative values with zero. Although the application yields a non-linear transformation, it remains very close to linear; in fact, it is a non-linear function consisting of two linear pieces. Since rectified linear units are nearly linear, several useful properties of linear models are preserved: They are easy to optimize with gradient-based methods, for instance, which will prove useful later during training.

Another widely-used activation function is the *logistic sigmoid*:

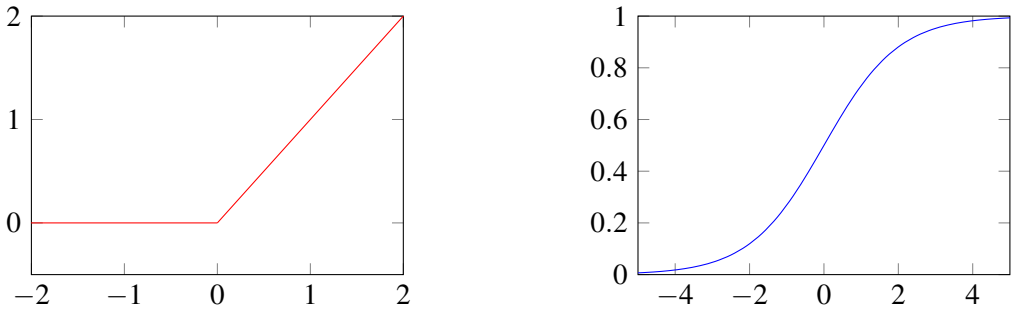
$$g(x) = \frac{1}{1 + e^{-x}}. \tag{3.2}$$

The sigmoid function takes a real-valued input and transforms it into a range between 0 and 1, as shown in 3.2b. This makes it particularly useful for models where we have to predict probabilities as output, for example in multi-class classification tasks. It is a smooth function that is continuously differentiable, which is also a very useful property in the context of computing gradients during training. However, the use of sigmoid activation functions can lead to the vanishing gradient problem. I will explain this phenomenon after introducing the computation of gradients in section 3.1.5.

The *hyperbolic tangent* or *tanh* function is another common choice for an activation function. It can be defined as:

$$tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}, \tag{3.3}$$

but can also be expressed as a rescaling of the logistic sigmoid function: $tanh(x) = 2g(2x) - 1$. Similarly, it transforms the input values into a range between -1 and 1. Hence, it is symmetric around the origin and



(a) Rectified linear unit (ReLU)

(b) Logistic sigmoid

Figure 3.2: Commonly used activation functions ReLU and logistic sigmoid.

can also output negative values, but apart from that, it shows the same properties and potential problems as the sigmoid function.

Besides these three functions that I introduced, there exist a number of other activation functions in practice. Through extensive research in this field, new activation functions that are better suited to particular neural network architectures are introduced from time to time as well. However, ReLU activation functions have been shown to outperform models with sigmoid or tanh units (Glorot et al., 2011).

3.1.2 Forward Propagation

We have now seen the different components that make up a feedforward neural network. To understand its training mechanism, we first need to define the terms forward propagation, cost function, and backpropagation.

First of all, the term *forward propagation* or *forward pass* simply refers to the process of calculating an output prediction, given an input. To calculate the output for a training example x, we simply propagate our input values $x_1, ..., x_n$ through the network and compute all activations for each hidden layer. The activation of a neuron based on the previous layer can be computed according to following formula:

$$a_j^i = \sigma(\sum_k (w_{jk}^i \cdot a_k^{i-1}) + b_j^i), \tag{3.4}$$

where a_j^i represents the activation of the j^{th} neuron in the i^{th} layer, σ is the activation function, w_{jk}^i is the weight from the k^{th} neuron in the $(i-1)^{th}$ layer to the j^{th} neuron in the i^{th} layer, and b_j^i is the bias of the j^{th} neuron in the i^{th} layer. In this notation, our input values would be represented as $a_1^1, ..., a_n^1$, respectively. Basically, we apply the activation function to the weighted sum of the inputs (represented as input value and corresponding weight) and an added bias b.

Bias terms are usually added to increase the model's prediction flexibility: While a change in the weights only affects the function's gradient in a sense that it gets steeper or flatter, an additional bias neuron can actually shift the function.

It is noteworthy that weights and hidden values are usually represented as vectors, so it is possible to rewrite equation 3.4 in a much more intuitive vectorized version:

$$a^i = \sigma(w^i \times a^{i-1} + b^i), \tag{3.5}$$

where a^i represents the vector of all activations in the i^{th} layer. This is also a very useful feature when implementing neural networks because we have several linear algebra libraries with fast implementations of matrix and vector operations at our disposal.

We now learned how to calculate the prediction value for a given input x in the feedforward neural network. For the network to be able to learn its optimal parameters, we need to define an objective function that we want to optimize - or, more precisely, a cost function we want to minimize.

3.1.3 Cost Function

A *cost* or *loss function* measures the difference between a model's actual and predicted output values. With linear models, the loss can be modeled as a convex function, so optimizers such as gradient descent are guaranteed to find the minimum after some training time and converge. However, loss functions of neural networks can not be assumed to be convex, especially since the network architecture involves many hyperparameters and non-linear activation functions. Therefore, if we want to optimize the parameters of a neural network during training, we usually settle for a very low, pre-defined cost value and stop the training once gradient descent has found that point.

For simplicity, I will choose *mean squared error* (also called *quadratic cost*) as the loss function for our neural network, but there exist several more loss functions that are used in this context, for example cross-entropy cost or exponential cost. We can define the mean squared error as

$$C = \frac{1}{n} \sum_x \| \, y(x) - a^L(x) \, \|^2, \tag{3.6}$$

where n denotes the total number of training examples, x is an individual training example, $y = y(x)$ is the desired output, L is the number of layers in the network, and $a^L = a^L(x)$ the vector of output activations when x is the input.

Two main requirements about our cost function C must be satisfied if we want to use it in combination with backpropagation: One property of backpropagation is that it computes the partial derivatives $\partial C_x/\partial w$ and $\partial C_x/\partial b$ for weights w and biases b for a single training example and then averages over training examples to obtain $\partial C/\partial w$ and $\partial C/\partial b$. Hence, we need to make sure that C can be expressed as an average $C = \frac{1}{n}\sum_x C_x$ over cost functions C_x for individual training examples x. Since it is possible to write down the cost function of a single training example as $C_x = \| \, y - a^L \, \|^2$, this requirement has been satisfied.

Furthermore, we need to show that it is possible to write C as a function of the outputs from our neural network, so that $C = C(a^L)$. We can express the mean squared error of a single training example as

$$C = \| \, y - a^L \, \|^2 = \sum_j (y_j - a_j^L)^2, \tag{3.7}$$

where C is a function of the model's output activation, so this criteria has been satisfied as well.

Lastly, the mean squared error cost function is continuously differentiable, which means that it has a derivative at every point. In contrast, the mean absolute error (same function without squaring) has an undefined derivative at 0.

3.1.4 Backpropagation

Now that we have become acquainted with the concept of forward propagation to compute the predictions of our model and the cost function as a method to measure the quality of our model predictions compared to the actual target values, we can introduce backpropagation, the mechanism used to calculate the gradients of the loss function. Obtaining the gradients is a necessary step in the training process of a neural network because an optimizer such as gradient descent needs to be provided with them in order to update the model weights to minimize the loss function.

Backpropagation is a technique for calculating and evaluating derivatives in a neural network efficiently by applying the chain rule, and should therefore not be confused with an optimizer. It can therefore be considered as a necessary step in the optimization process, but the actual adjustments of the weight parameters are generally applied through methods such as gradient descent.

To fully explain all mechanisms involved in backpropagation requires numerous calculation steps and rich equations. Therefore I will only illustrate the basic intuition behind it and omit some definitions and certain equations because these mechanisms should not be the focus of this thesis.

I have provided a pseudocode version of the backpropagation algorithm following the introduced notation according to Nielsen (2015), depicted in algorithm 1. The algorithm can be broken down into three stages: Firstly, we set the input values in the input layer, and apply a forward pass to obtain the model predictions for our given input, as explained in chapter 3.1.2. By doing so, we obtain the weighted

input to the neurons in the current layer z^i for each hidden layer $i = 2$ to $i = L$ as well as the vector of activations a^i.

Secondly, we compute the output error δ^L in the output layer L: This is done by calculating the Hadamard product (elementwise product) of vector $\nabla_a C$ and $\sigma'(z^L)$. $\nabla_a C$ consists of the partial derivatives $\partial C/\partial a_j^L$, hence it expresses the rate of change of the cost function C with respect to the output activations a, while $\sigma'(z^L)$ measures the rate of change of the activation function σ in the output layer L.

In the third phase, we backpropagate the error from the final hidden layer to the first hidden layer in order to obtain the gradient of the cost function: For all hidden layers $i = L-1$ to $i = 2$, we multiply the error vector of the previous layer δ^{i+1} with the transposed weights of the previous layer $(w^{i+1})^T$ and then take the Hadamard product with $\sigma'(z^i)$.

By applying the transpose operation we simply ensure that the product of the multiplication results in the correct dimensionality. The multiplication operation is what can intuitively be understood as "propagating the error backwards" through the network: If we have a connection that is associated with a high weight and observed a high error value, we also end up with a high value due to multiplication, and hence this particular weight is penalized.

The actual computations involve applying the chain rule repeatedly to obtain all the derivatives, but to keep it short, I will not dive deeper into differential calculus at this point.

Algorithm 1 Backpropagation algorithm for a single training example.

1: **Input:** x. ▷ Vector of input values $x_1, ..., x_n$ for training example x
2: **set** $a^1 = x$. ▷ Set input values
3: **for each** $i = 2, 3, ..., L$ **compute:**
4: $z^i = w^i a^{i-1} + b^i$ **and** $a^i = \sigma(z^i)$. ▷ Forward propagation
5: **compute** $\delta^L = \nabla_a C \odot \sigma'(z^L)$. ▷ Compute output error
6: **for each** $i = L-1, L-2, ..., 2$ **compute:**
7: $\delta^i = ((w^{i+1})^T \delta^{i+1}) \odot \sigma'(z^i)$. ▷ Backpropagate error
8: **output:** $\frac{\partial C}{\partial w_{jk}^i} = a_k^{i-1} \delta_j^i$ **and** $\frac{\partial C}{\partial b_j^i} = \delta_j^i$. ▷ Gradient of cost function

3.1.5 Stochastic Gradient Descent

We have now seen how to apply backpropagation to calculate the gradient of the cost function for weights and biases for a single training example. We obtained a vector which consists of partial derivatives with respect to each parameter. Each partial derivative shows us the direction of the steepest increase for this parameter in our cost function, based on this training example.

Since training a neural network means finding a set of weights and biases that make the cost as small as possible, we have now acquired all the knowledge we need in order to train our network. The general approach to achieve this goal is to use gradient descent as optimization method.

Basically, the idea of gradient descent is to move into the opposite direction of the gradient to minimize the cost function. The step size we take is determined by a hyperparameter called *learning rate*. I will hereby refer to the learning rate as η. Hence, η dictates how fast or slow we move into the optimal direction. Choosing an appropriate value for η is crucial: If the parameter is too small, our training process will take a long time, and if it is too high, we might overshoot the minimum so that the algorithm never converges, or even diverges. Note that the cost function of neural networks usually does not come with a convergence guarantee, as explained in section 3.1.3.

In the context of calculating gradients, another phenomenon can arise that I briefly mentioned: The *vanishing gradient* problem. When we compute the gradient by backpropagating errors, we repeatedly multiply very small numbers, especially if we use activation functions in the range (0,1), such as sigmoid. Accordingly, the deeper the computational graph of our network is, the more backpropagation steps we take, and hence the smaller our gradients tend to become. And since small gradients indicate little or no

impact on C, this effectively means that the earlier layers cannot learn their optimal weights properly. As already mentioned, we can address this problem by using ReLU activation functions instead.

Similarly, if we have very large values for weights and biases, the problem of *exploding gradients* can occur: The repeated multiplications can lead to far too high weight updates, which in practice can result in overflowing numbers and NaN values. To prevent exploding gradients, methods such as gradient clipping can be applied.

Thus, using a learning rate that is too small as well as the fundamental problem of unstable gradients can result in very long training times. However, if we try to learn our model's optimal weights using gradient descent, another problem arises that concerns the duration of the training process: Regular gradient descent (also known as *batch gradient descent*) computes ∇C based on the entire training dataset. In other words, it iterates over all training examples in order to perform just one weight update. As soon as we deal with large amounts of data, which is common practice in most current tasks, training a deep model with gradient descent can become intractable. And assumed there is a certain inherent similarity between the training examples, this seems to be quite an inefficient and possibly redundant approach.

Fortunately, there exist two variations of this algorithm: *Stochastic gradient descent* and *mini-batch gradient descent*. In earlier research, the term stochastic gradient descent has been used to describe methods that perform a weight update after every single training example (also sometimes referred to as *on-line gradient descent*), and the term mini-batch gradient descent referred to methods that take a randomly chosen mini-batch of size m from the training set and perform a weight update after an iteration over these m training examples (Goodfellow et al., 2016, ch. 8.1.3). In contrast, now it seems to be the general consensus that stochastic gradient descent includes sampling a mini-batch of training examples. I will therefore use the terms *stochastic gradient descent* and *mini-batch gradient descent* interchangeably.

A pseudocode version of the training process of a neural network using stochastic gradient descent is given in algorithm 2. Note that compared to backpropagation illustrated in algorithm 1, three additional hyperparameters occur: Learning rate η, batch size m, and number of epochs e. Aside from the additional loops for e, m and x and the weight and bias update introduced through gradient descent, not much has changed. The bias and weight update for a layer is applied in the following way: The gradient of the cost function with respect to the bias (or weights) of this layer is computed by averaging over all training examples x in the mini-batch. Since this gives us the direction of the steepest increase of C, we want to move into the opposite direction, which is represented by subtracting this vector. We multiply the vector by learning rate η because this determines by what amount we move into this direction.

Although the mechanism of stochastic gradient descent with mini-batches is used for training most neural networks, there are several optimizers available that expand this algorithm and therefore improve the model performance, for example by automatically adapting the learning rate during the course of learning. Among the most noteworthy of these optimizers are the AdaGrad (Duchi et al., 2011), RMSProp (Tieleman and Hinton, 2012), and Adam (Kingma and Ba, 2014) algorithm.

Lastly, I will address an issue that I have not covered yet, namely parameter initialization. For the sake of space, the corresponding steps for parameter initialization are not depicted in algorithm 1 and 2, but of course it is necessary in practice to include this step. While parameter initialization strategies are a topic of much debate and ongoing research in the context of optimizing deep neural networks, it can be noted that initializing the weights in a network with zero or another constant value is generally not a good idea: If all the weights in a layer are the same, then the same function is calculated and learned for every neuron. Instead, it is common practice to initialize the weights randomly from a high-entropy distribution over a high-dimensional space and the biases with heuristically chosen constants (Goodfellow et al., 2016, ch. 8.4).

3.1.6 Summary

By introducing general neural network terminology and architecture, the choice of a cost function, forward propagation, backpropagation, and stochastic gradient descent with mini-batches, we have now learned how to train deep feedforward neural networks. Essentially, we have learned how to train most modern deep neural networks, since most deep models use these underlying basic mechanisms, although sometimes with slight variations depending on the model architecture.

Algorithm 2 Neural network training algorithm using stochastic gradient descent with mini-batches.

Input: complete training data set X.
for each epoch e: ▷ One full training cycle
  generate batch of size m **from** X. ▷ Randomly generate mini-batches
  for each batch $x_1, ..., x_m$: ▷ Batch with training examples $x_1, ..., x_m$
    for each x: ▷ Training example x
      set $a^{x,1} = x$. ▷ Set input activation
      for each $i = 2, 3, ..., L$ **compute:**
        $z^{x,i} = w^i a^{x,i-1} + b^i$ **and** $a^{x,i} = \sigma(z^{x,i})$. ▷ Forward propagation
      compute $\delta^{x,L} = \nabla_a C_x \odot \sigma'(z^{x,L})$. ▷ Compute output error
      for each $i = L-1, L-2, ..., 2$ **compute:**
        $\delta^{x,i} = ((w^{i+1})^T \delta^{x,i+1}) \odot \sigma'(z^{x,i})$. ▷ Backpropagate error
    Gradient descent:
    for each $i = L, L-1, ..., 2$:
      update weights according to rule:
      $w^i \to w^i - \frac{\eta}{m} \sum_x \delta^{x,i} (a^{x,i-1})^T$. ▷ Update weights
      update biases according to rule:
      $b^i \to b^i - \frac{\eta}{m} \sum_x \delta^{x,i}$. ▷ Update biases
  start new epoch if all training examples in X were used.

While feedforward neural networks are well suited for tasks where we process fixed-size grids of input values, such as image processing, they show inherent limitations: It is not possible to model different timesteps of an input sequence (as opposed to Hidden Markov Models, for example). However, when we are dealing with natural language, we can assume that a word or character in an input sequence heavily depends on the previously occurred words or characters. This makes feedforward neural networks not particularly useful for language modeling. But there exists another type of neural network that is able to model these dependencies and that is therefore widely used in natural language and text processing tasks: The recurrent neural network.

3.2 Recurrent Neural Networks

A *recurrent neural network (RNN)* is a type of network that incorporates the concept of time steps into the model architecture. This makes them particularly useful for processing sequential information, and they are therefore used in many NLP tasks, such as language modeling (Mikolov et al., 2010), machine translation (Auli et al., 2013), and speech recognition (Graves et al., 2013).

When we use feedforward networks, we are restricted to a fixed length and a fixed number of computational steps. However, with RNNs, we are able to model input sequences of variable length, and, if desired, we can also operate over output sequences.

RNNs share their model parameters across different timesteps. Compared to feedforward networks that use different weight matrices for every layer, with RNNs we have the same weight matrices at every time step. These shared weight matrices are classified depending on their connection type: A network can have separate matrices for input-to-hidden connections, hidden-to-hidden connections, and hidden-to-output connections, for example, depending on the RNN version.

For a better understanding, we could also interpret RNNs as fixed implemented functions with several inputs (input sequence) and internal variables (weights). In fact, it has been shown that RNNs are Turing complete (Siegelmann and Sontag, 1995), which means that there exists a finite RNN for any computable function.

This also means that we compute the same function at every time step, which corresponds to the idea of generalizing over sequences. In language modeling, for instance, if we encounter the sequence

"I went swimming on Sunday", we also want to be able to model sequences such as "On Sunday I went swimming" without explicitly encountering this additional sentence construction. This can be achieved by RNNs due to their parameter sharing properties.

The basic structure of an RNN is shown in figure 3.3. Note that the depicted model maps an input sequence of x values to a corresponding sequence of output o values and hence shows synced sequences of input and output. But this representation is not mandatory - we could also map a fixed input to a sequence output or a sequence input to a fixed output. Or we could rearrange the connections between input, hidden layers and output, depending on the task: In machine translation, for example, the depicted structure would not necessarily make sense; Instead of associating every word with a corresponding word in another language, it would make more sense to first read the sequence in one language and then output the corresponding sequence in another language.

On the left, we can see the computational graph of the synced sequence RNN. The recurrent connection represents the hidden state calculation that is not dependent on a particular sequence length. The diagram on the right shows the same model unfolded into a network with separate timesteps: x_t represents input, s_t hidden state and o_t output at step t. U, W and V are the input-to-hidden, hidden-to-hidden and hidden-to-output weight matrices of the model, respectively.

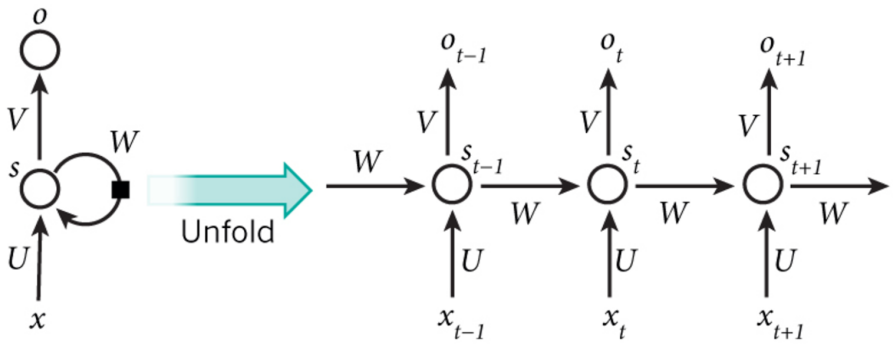


Figure 3.3: Example of a synced sequence RNN. (Figure source: LeCun et al. (2015))

Forward propagation of the depicted RNN can be computed according to following equations:

$$s^{(t)} = \sigma(b + Ws^{(t-1)} + Ux^{(t)}), \tag{3.8}$$

$$o^{(t)} = c + Vs^{(t)}, \tag{3.9}$$

where we introduce two additional parameters b and c as bias vectors for hidden layers and output layers. Note that the activation computation in equation 3.8 is very similar to forward propagation in feedforward neural networks as shown in equation 3.5: Although I slightly changed the notation and use a matrix notation now instead of a vectorized implementation, the basic mechanism to obtain the current hidden state activation $s^{(t)}$ is still to apply activation function σ to the product of weights W and previous hidden state activation $s^{(t-1)}$ and an added bias b. However, some changes arise due to the different architecture of the model: We also add the input sequence $x^{(t)}$ at the corresponding time step, multiplied by the input-to-hidden weight matrix U. Accordingly, to compute output activation $o^{(t)}$, we multiply current activation $s^{(t)}$ by hidden-to-output weights V and add bias c, as illustrated in equation 3.9.

As mentioned, weight matrices W, U and V as well as bias vectors b and c are shared parameters across the whole network. This greatly reduces the number of parameters the model has to learn, compared to feedforward neural networks.

In figure 3.3, we have not yet made any assumptions about how to represent output probabilities or about how to compute the loss for a sequence x. One possibility would be to apply the softmax operation as a post-processing step.

Softmax is another activation function that is often used in the final layer of neural network: It represents the probability distribution over k different classes:

$$softmax(z)_i = \frac{e^{z_i}}{\sum_j^k e^{z_j}} \text{ for } j = 1, 2, ..., k. \tag{3.10}$$

As with the sigmoid function, the output range will be between 0 and 1, but all probabilities in an output vector will sum up to one as well, which is not the case for sigmoid. Therefore, if we want to obtain a vector of normalized probabilities over the output of our RNN, we simply calculate:

$$\hat{y}^{(t)} = softmax(o^{(t)}), \tag{3.11}$$

where $\hat{y}^{(t)}$ represents such an output vector of normalized probabilities for the output $o^{(t)}$ at time step t. To compute the loss, we choose a cost function as described in section 3.1.3, for example mean squared error, and compute the difference between target $y^{(t)}$ and output $\hat{y}^{(t)}$. The total cost for an input sequence x can then simply be calculated as the sum of these losses over all time steps. In practice, however, the recommended default for a cost function in combination with softmax is negative log-likelihood.

RNNs are also trained with backpropagation and gradient descent, as described in the previous sections. They use a variant of the algorithm called backpropagation through time (BPTT) which uses the same basic mechanisms as standard backpropagation. It is referred to as backpropagation *through time* to denote that we compute the gradients of the unfolded RNN from time step to time step – just as we did with feedforward networks from layer to layer. But the procedure is the same: We apply the chain rule to the underlying computational graph, calculate the gradients of our loss function with respect to the model parameters, and gradient descent uses these gradients to adjust the parameters. The only difference is that RNNs have shared parameters across the whole model, so additionally to summing up the errors, we also sum up the gradients at each time step.

Like feedforward networks, RNNs can suffer from the vanishing gradient problem. In fact, when we work with natural language, it is common to have very deep unrolled computational graphs, for instance if one training example represents a complete sentence. To that effect, vanishing gradients are even more common with RNNs. Similarly, RNNs have difficulties learning long-term dependencies. However, there are particular extensions of vanilla RNNs that are specifically designed to deal with these problems, most notably long short-term memory (LSTM) units and gated recurrent units (GRUs). To this day, they are the most used RNN variants in NLP applications (Young et al., 2017).

In the following section, I will focus on LSTMs and explain how they can be integrated into a recurrent network to build a modern RNN capable of delivering state-of-the-art results.

3.3 Long Short-term Memory Units

Long short-term memory networks (LSTMs) are recurrent neural networks that incorporate LSTM units. Accordingly, the underlying network architecture is still an RNN, but the hidden layer units are replaced by LSTM units.

While vanilla RNN layers only apply a single transformation function to their input, LSTM units apply a more complex transformation: Each unit consists of several sublayers, gates, and a cell state. The gates, consisting of an input, output and forget gate, regulate the amount of information that flows to the cell state. Each of these gates is composed of a sigmoid layer and a pointwise multiplication operation. The cell state serves as the network's internal memory and can remember values over arbitary time intervals. In addition to the outer recurrent structure of RNNs, cells of LSTM units introduce internal recurrence by applying self-loops.

A single LSTM unit that corresponds to a single time step is depicted in figure 3.4. In the diagram, x_t represents the input vector to the LSTM unit – it is used to calculate the vector of input gate activations i_t, output gate activations o_t, and forget gate activations f_t. The $\times$ in the following nodes denotes the pointwise vector multiplication (also introduced as Hadamard product, see section 3.1.4). Additionally, we have an input unit, depicted on the left: Compared to the gates, which all use the sigmoid function, the

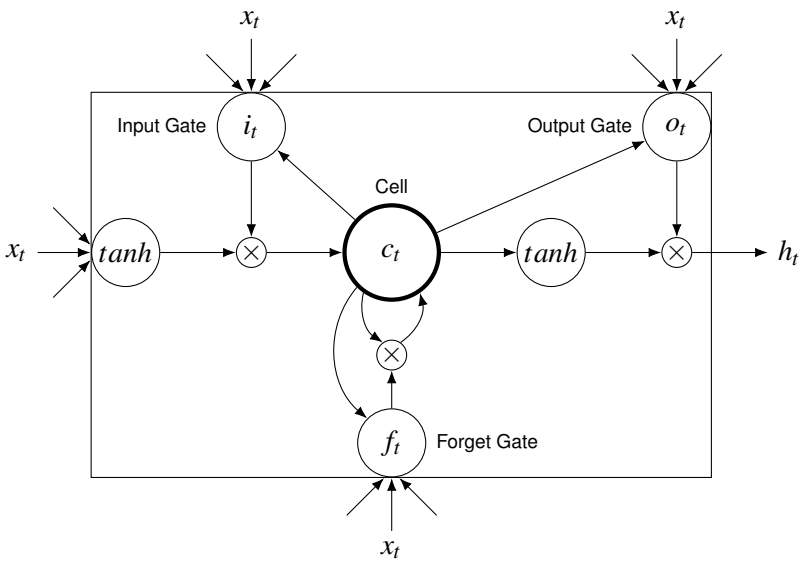


Figure 3.4: Representation of a single LSTM unit.

transformation applied at the input unit could also be any other squashing nonlinear activation function, for example *tanh*. The cell state c_t has several ingoing and outgoing connections, but most noteworthy we can see that its output is connected to all gates i_t, o_t, and f_t as well as to the internal recurrent self-loop. Lastly, there is another *tanh* layer on the right that is used to compute the final output vector h_t of the whole LSTM unit.

The computations performed to calculate the output vector h_t can be expressed gradually. First of all, the forget gate layer, as the name suggests, determines what information should be thrown away from the cell state. Because sigmoid is applied, the resulting output will be a vector with values between 0 and 1, and each position in the vector denotes the degree of importance for the corresponding value. This output vector f_t can be calculated according to the equation

$$f_t = \sigma(U_f x_t + W_f h_{t-1} + b_f), \tag{3.12}$$

where h_{t-1} represents the output vector of the previous LSTM unit, U_f and W_f are the input and recurrent connection weight matrices of the forget gate, and b_f is the bias vector of the forget gate. In other words, the forget gate looks at the current input and the previous unit output, applies weights and biases and outputs a vector consisting of values between 0 and 1. Each entry in this output vector corresponds to a value in the cell state c_t, and by computing the element-wise product of f_t and c_t in the next step, the gate can influence the memory of the cell state.

The output activation vectors i_t and o_t of the input and output gate are calculated similarly:

$$i_t = \sigma(U_i x_t + W_i h_{t-1} + b_i), \tag{3.13}$$

$$o_t = \sigma(U_o x_t + W_o h_{t-1} + b_o). \tag{3.14}$$

Note that each gate has its own set of parameters U, W, and b. The new cell state c_t can then be computed according to following equation:

$$c_t = f_t \circ c_{t-1} + i_t \circ tanh(U_c x_t + W_c h_{t-1} + b_c), \tag{3.15}$$

where c_{t-1} represents the previous cell state. The parameters U_c, W_c and b_c introduce a new set of parameters, the weights and bias of the cell state. While this equation is somewhat less intuitive, the single steps can easily be compared to the diagram in figure 3.4: First, the LSTM unit computes the element-wise multiplication of the forget gate activations f_t and the previous cell state c_{t-1}. This represents the scaling of information in the cell's internal memory.

Then, the calculated vector is added to another vector, namely the element-wise product of the input gate activations i_t and the output vector of a *tanh* activation. The *tanh* layer output can be considered as vector of new candidate values that will be added to the state, and i_t scales by how much we should update each value. After these values were obtained, the output of the LSTM unit can be calculated accordingly:

$$h_t = o_t \circ tanh(c_t). \tag{3.16}$$

Note that due to the weights and biases of the input gate, output gate, forget gate, and the cell state, we have four times as many parameters as with vanilla RNNs.

As mentioned in the beginning, the underlying structure of LSTM networks is still an RNN. But instead of just computing a single activation function, the LSTM network carries out the computations I just described at each time step. Accordingly, we can simply replace the hidden state s_t of our vanilla RNN at each time step with an LSTM unit that instead produces the output h_t. Then, the same mechanisms and computations in terms of output calculations, cost function, forward propagation, BPTT, and stochastic gradient descent can be applied.

To simplify the equations, I illustrated the calculations of a shallow (one layer) RNN, but of course it is more common to use deep architectures with many layers stacked on top of each other. The mechanism of stacking several layers in an LSTM does not differ from traditional feedforward neural networks: The output of a previous layer is simply used as the input of the next layer.

3.4 Summary

In this chapter, I have introduced general terminology and explained the underlying mechanisms that are necessary to train a neural network: Using forward propagation to obtain output predictions, choosing a cost function to calculate the difference between predictions and target values, using backpropagation to propagate the error backwards through the network and to compute the gradients in respect to the model parameters, and lastly, using stochastic gradient descent to adjust the model parameters accordingly.

I have explained the architecture of feedforward neural networks, recurrent neural networks, and long short-term memory networks. I also addressed several problems and limitations that arise from the specific model architectures: Feedforward neural networks are not able to model sequences with respect to different time steps and are therefore not a very good choice for processing natural language. In contrast, recurrent neural networks introduce the concept of time steps into their model architecture and as a result, they are able to capture the structure of natural language very well. However, they often have difficulties learning long-term dependencies and suffer from the vanishing gradient problem.

LSTMs were specifically designed to address these problems, and they are widely used to obtain state-of-the-art performance in learning problems related to sequential data. Accordingly, LSTMs are very well suited for modeling the constraints inherent in natural language. I therefore assume that they might also be able to tackle the challenging task of generating poetry and learn the inherent structure of poems, although this form of text imposes even more constraints that have to be learned by the model. In the next chapter, I will briefly review current research in the field of poetry generation with recurrent neural networks. I will specifically focus on the model architecture and evaluation mechanisms that are implemented in these papers, and then draw conclusions on which model and evaluation method I should choose for my own implementation.

Current Approaches on Poetry Generation

4.1 Chinese Poetry Generation

First of all, I found that current poetry generation approaches involving (recurrent) neural networks almost exclusively aim to generate Chinese poetry. I assume that this is due to the fact that in China, poetry is traditionally held in extremely high regard. Every paper on this topic also stresses the importance of Chinese poetry as a cultural heritage.

However, Chinese and English are two fundamentally different languages. On the phonetic level, for instance, the Chinese language incorporates additional constraints: To distinguish between words, the word pitch or tone is an important feature, whereas the speaker's tone only conveys paralinguistic information in the English language. This additional phonetic feature results in even more phonetic constraints on Chinese poems: Besides rhyme schemes and metric patterns, which also apply to English poetry, we also have restrictions in terms of tone patterns that can vary depending on the poetic form.

Another radical difference occurs in the context of language modeling: The following Chinese poetry generation systems employ character-level language models. However, the representation of English and Chinese characters is fairly different. In English, a character corresponds to an alphabet letter as the smallest writing system unit, and characters are stringed together in order to produce words. A single character does usually not convey any semantic information and is not considered a complete word (other than very few exceptional cases, such as *I* and *a*). Chinese characters, however, are logograms and each character represents a single word or phrase. This makes it generally difficult to compare implementations of character-level language models in English and Chinese.

Before neural networks were applied to this task, there have been several approaches where the generation of Chinese poetry is treated as a statistical machine translation problem and each new poem line is generated as a "translation" of the previous line (Jiang and Zhou, 2008; He et al., 2012). The first neural network based poetry generator was then introduced by Zhang and Lapata (2014). The generator aims to produce Chinese quatrains, a poetic form consisting of four lines that are each five or seven characters long. These poems furthermore include a single rhyme and a particular tonal pattern. It is worth mentioning that almost all of the following approaches try to generate Chinese quatrains.

The generation process is executed in the following manner: First, the user supplies keywords that determine around which concepts the poem should revolve. Note that this keyword-based approach where a user interactively defines the poetic theme also occurred in earlier systems such as ASPERA, as described in section 2.2.2. The keywords are restricted to words occurring in the ShiXueHanYing poetic phrase taxonomy, a corpus containing 1,016 manually annotated clusters of poetic phrases. The first line of the poem is generated based on these keywords: Several phrases corresponding to the keywords and the tonal pattern constraints are generated, and a character-based language model chooses the best ranked candidate as first line of the poem. Then, a convolutional sentence model is used to convert the poem line into a vector. A recurrent context model takes these line vectors as input and compresses them to one vector, which is consequently used as hidden layer for the next model. This model's output vector represents the context up to the current character – which is then used by another RNN, the generation model, to estimate the probability distribution of the next character.

To evaluate the model predictions and the quality of the output poems, three different methods are applied: A perplexity calculation to measure the model predictions with respect to the gold standard, a BLEU score to evaluate the model's ability to generate the second, third and fourth line given previous gold standard lines, and lastly a human judgment study. For this study, the authors invited 30 experts on

Chinese poetry that rated output poems on a one to five scale on four dimensions: Fluency, coherence, meaningfulness, and poeticness. The approach has been shown to outperform previous approaches on Chinese poetry generation, but the human evaluation study revealed that the quality of the machine-generated poems cannot yet be compared to the human-generated ones. Although this system is not template-based, the general expressivity of the resulting poems is limited, since the user can only choose from a set of certain words in the taxonomy.

There have been several similar approaches on generating Chinese poetry: The iPoet model (Yan, 2016) introduces a system where the user can input arbitrary keywords and their meaning is captured via a convolutional or recurrent neural network over characters. It further incorporates an iterative polishing scheme, where the single-pass generation process is replaced by a multi-pass generation. In other words, the RNN uses a first poem draft that gets further improved over multiple iterations. This incremental generation tries to capture the actual process of human poem writing, an idea that was already present in the earlier evolutionary approaches mentioned in section 2.2.3. As evaluation method, perplexity measurements, BLEU score, and a human evaluation study are conducted as well. However, the focus lies on comparing their model to different baseline approaches, as well as comparing different generation strategies. The author does not seem to aim to answer the question to what extend the generated poems can actually be compared to human-written poems, as he states in the conclusion that the iPoet model can "generate rather good poems and outperform baselines".

Wang et al. (2016) use a planning-based approach where the input writing intent of the user can be any document described in natural language. From this input document, sub-topics are derived, and each poem line is generated to include such a sub-topic. An attention based RNN encoder-decoder framework encodes both the sub-topic and the preceding text. This approach especially improves the coherence and semantic consistency across lines, since the previously mentioned systems only generated the first line based on input keywords, and the other lines based on all other previously generated lines.

Additionally to applying human evaluation according to the previously mentioned criteria, the authors conducted a study that resembles a Turing test: Actual poems were randomly selected from the test set, and the titles of these poems were used as input to the generation model in order to obtain similar-looking, machine-generated poems. A normal group and an expert group then had to identify the human-written poem from this pair. The study showed that 49.9% of the machine-generated poems were either wrongly identified as the human-written poems or could not be distinguished by the normal evaluators. For the expert group, this number dropped to 16.3%. The authors therefore argue that at least under the standard of normal users, the quality of the system's generated poems is very close to human-written poetry.

While these RNN-based approaches relying on keywords and other textual information have dominated the field of Chinese poetry generation in the last few years, I briefly want to mention that very recently, a novel approach has been proposed: Xu et al. (2018) introduced a memory based neural model that aims to generate poetry by extracting visual features from pictures. Since then, similar work focusing on the extraction of poetic clues from images has been proposed, for example RNN-based adversarial training via policy gradient (Liu et al., 2018a) and a multi-modal approach consisting of a hierarchy-attention seq2seq and Latent Dirichlet allocation (LDA) model (Liu et al., 2018b).

4.2 English Poetry Generation

As already mentioned, the literature on poetry generation involving neural networks in other languages than Chinese is quite sparse - I was only able to find two approaches that aim to generate English poems with RNNs.

The Hafez model proposed by Ghazvininejad et al. (2016) combines an LSTM with finite-state machinery to generate 14-line classical sonnets in English. The generation process can be divided into three stages: In the first stage, a scored list of words related to the user-supplied topic is generated by training a continuous-bag-of-words model. These related words are hashed into rhyme classes. In the second stage, a finite-state acceptor (FSA) encoding all word sequences that use these rhyme words and obey formal sonnet constraints is created. In the third stage, a path through the FSA that corresponds to a meaningful poem is extracted: This is achieved by implementing an LSTM language model and employing a beam

search guided by the FSA. Furthermore, an additional encoder-decoder sequence-to-sequence model is implemented as well that preselects the rhyme words before the generation process. In this way, the generation model already knows all rhyme words prior to the generation process, which leads to a better knowledge of the overall topic of the poem. To show the generality of their approach, the authors modify their system to generate Spanish poetry for the classical Spanish poem form soneta, which corresponds to a different rhyme scheme. They also conduct human judgment studies to test different variants of the generation process: Out of two poems, the participants are asked to choose a preferred one. However, no evaluation study that tries to determine the overall quality of the generated poems, for example compared to human-written poems, is applied.

The second approach by Hopkins and Kiela (2017) introduces two novel models for generating different kinds of English poetry. The first model consists of an LSTM that is trained on the phonetic representations of a poetry corpus. In order to to convert the generated phoneme symbols back to orthographic symbols, the phonetic encoding is interpreted as homophonic cipher and modeled via a Hidden Markov Model and an n-gram language model, where the Viterbi algorithm is applied to find the most likely sequence of words. However, the authors argue that this kind of model requires internal phoetic consistency and cannot generalize to other forms of poetry. They therefore introduce a second model consisting of a generative language model representing content, and a discriminative model representing form. The generative character-level model is trained on a generic corpus consisting of different forms of poetry. The discriminative model is then used to constrain the output during the generation process. It consists of a probabilistic classifier that was extracted via Expectation Maximization and determines the most likely stress patterns for each word. To ensure a consistent theme throughout the poem, the probability of generating words that are semantically related to a theme word is heuristically boosted. To achieve this, a distributional semantic model is used to obtain its semantic neighbors. To increase the likelihood of sampling these words, their sampling probability is multiplied by the cosine similarity to the key word. The authors state that the same mechanism can be used to increase the occurrence of poetic devices such as assonance, consonance and alliteration, since they can be described as the repetition of identical sequences of characters. Poetic devices contribute considerably to the reader's perception of a poem being poetically well-formed as well as creative, and to my knowledge, this approach is the first one to actually address this level of language in practice.

4.3 Conclusions

In this chapter, I have introduced current research in the field of poetry generation with recurrent neural networks. Since all of these approaches involve the implementation of several models simultaneously and at least one evaluation method where a human judgment study is conducted, this left me with two problems: Firstly, my resources in terms of computational power were limited. I therefore decided to implement a stand-alone model, and I assumed that an RNN with LSTM units would be the best choice for this task. And secondly, I did not have a team of human evaluators at my disposal to conduct such a study, let alone a team of poetry experts. My evaluation approach therefore had to be be fully automatic.

Since the well-formedness of a poem depends on many restrictions in terms of syntax, semantics, and phonetics, it is also necessary to decide which aspects of well-formedness we want to incorporate into the evaluation method. Obviously, we cannot evaluate all criteria, since some of them seem to be only captured through human judgment studies.

I therefore decided to focus on the aspect of formal poetic well-formedness. In other words, I want to automatically evaluate whether an output poem corresponds to the syntactic and phonetic restrictions imposed by its poetic form. To that effect, such an evaluation method covers two of the three mentioned levels of language, but it does not incorporate semantic evaluation. Although semantic well-formedness is undoubtedly a highly important influence on the quality of an output poem, semantic consistency was mostly covered by human judgment studies in current research, as it involves partially subjective impressions that are difficult to formalize.

In the following chapter, I will now explain my evaluation approach step by step, starting with the introduction of limericks, the poetry genre I chose to generate with my neural network.

Evaluation Approach

5.1 Limericks

Limericks are a poetic form, often humorous and sometimes including some kind of punch line or twist at the end. The structure of a limerick imposes several restrictions on language in terms of syntax and phonetics. Such a poem can be classified as *poetically well-formed* if the following formal criteria apply:

(a) consists of **five lines** or verses (metrical lines)

(b) the **stress positions** in each verse correspond to the **anapestic meter** (`001`)

(c) has a strict **rhyme scheme** (AABBA)

(d) **verse three** and **four** are **shorter** than the other verses (measured in syllable counts)

(e) **verse one, two,** and **five** consist of **8-11 syllables** and **verse three** and **four** consist of **5-7 syllables**.

To illustrate the structure of such a limerick, an example from my actual training data is given in figure 5.1. According to the first and second criterion, the poem consists of five verses and has the anapest as underlying metrical pattern. To show this, we can also represent the first, second and fifth line in terms of stress positions as the string `001 001 001`, where `0` corresponds to an unstressed syllable and `1` to a stressed one, and line three and four as `001 001`, accordingly. Thus, we can see that all lines incorporate the anapest as underlying metrical pattern (`001`). We can also see that the fourth criterion is met because in terms of syllable counts, verse three and four are shorter than the other verses. The restrictions of the rhyme scheme AABBA as defined in criterion three are met as well because *net* rhymes with *sweat* and *babette* and *chose* rhymes with *knows*. And lastly, if we count the syllables of each line, we see that they correspond to the number of allowed syllables defined in criterion five.

a security break on the **net**
has me stunned, breaking out in a **sweat**.
i'm a man voters **chose**,
and now somebody **knows**
that i chat on the web as "**babette.**"

Figure 5.1: Example training data limerick.

Criteria one to four express the universally defined structure of limericks that can be found all throughout literature on this topic. Criterion five is an additional criterion I defined after working with limericks for a while - how this criterion was derived is explained in section 5.3.1 in detail.

These five properties are the criteria I use to evaluate whether a limerick is well-formed and hence corresponds to a good output example. As previously mentioned, the evaluation approach focuses on evaluating poetic well-formedness that can be expressed on the syntactic and phonetic level, and does not take other levels of language into account. In section 6.3, I will address this issue as well as other potential limitations.

5.2 Arpabet Notation

In order to evaluate phonetic properties of words, we need to obtain their corresponding phoneme representations. The most common way to represent the pronunciation of a word in general is via the International Phonetic Alphabet (IPA) - however, I chose the Arpabet phonetic transcription code because it works best with the tools I decided to use.

In Arpabet notation, phonemes are encoded as a sequence of ASCII characters. The vowel representations also include stress information: Each vowel contains an additional `1` if it is stressed. Depending on the variant of the Arpabet notation, vowels with secondary stress are sometimes represented as `2` and unstressed vowels as `0` - however, in the version I use, `1` simply stands for a stressed syllable and a vowel without any additional symbol represents an unstressed syllable.

The set of Arpabet symbols also varies among different applications - in my case, the set consists of 39 symbols, namely 15 vowel sounds and 24 consonants. Some selected symbols are shown in table 5.1. In the word *father* (`F AA1 DH ER`), for example, we can see the stress information I just described: The vowel `AA` is represented as `AA1` because it corresponds to the stress position of the word, and the vowel `ER` contains no such information because it is unstressed.

Arpabet	**Word Example**
AO	frost (`F R AO1 S T`)
AA	father (`F AA1 DH ER`)
AW	how (`HH AW1`)
UW	you (`Y UW1`)
IY	she (`SH IY1`)
OY	boy (`B OY1`)
JH	gym (`JH IH1 M`)
TH	thanks (`TH AE1 NG K S`)
M	man (`M AE1 N`)
Z	zoo (`Z UW1`)

Table 5.1: Some selected Arpabet symbols with word examples.

5.3 Evaluating Limericks

We now have the components at our disposal that we need in order to design an evaluation method: A set of criteria we want to evaluate, and a way to obtain phonetic representations in order to test these criteria. I will now explain how to check and implement each criterion one by one.

5.3.1 Number of Verses

Evaluating the number of verses for a limerick proves to be a trivial task. Since we are only interested in whether a poem consists of the appropriate number of five lines or not, we can simply read in the limerick in a way that preserves its structure and implement a boolean test to check this condition. By doing do, we either obtain `True` or `False` for each Limerick.

5.3.2 Metric Pattern

Evaluating whether a poem fulfills the restrictions of the anapestic meter is a more complex task. First of all, we should be aware that our phonetic representations would need to be one hundred percent correct in order to be able to check this condition reliably. We count the syllables of a verse based on the occurring vowels, and the stress information lies in the vowels as well. A word with an incorrect phonetic representation, for example one that is erroneously represented with one vowel too many, will consequently mess up the whole metric pattern of the verse, and accordingly, of the whole poem. But we cannot assume that the generated phonetic representations are always correct - especially if we want to evaluate limericks generated by a neural network. These poems potentially include a number of neologisms or non-words for which the system might fail to generate accurate phonetic representations. However, just because of one erroneous representation, we probably do not want to reject the whole poem. So instead of just checking whether all verses in a poem correspond to the anapestic meter or not, we could check this condition for each verse separately and compute an overall metric score for the poem based on that. This way we can obtain a degree of metric correctness rather than a simple, boolean expression, which also corresponds to a more fine-grained analysis.

To find out whether a particular verse corresponds to the anapestic meter or not, we must first make a few more assumptions about the general structure of limericks.

Extracting an Extended Stress Pattern for Limericks

The standard form of a limerick consists of three metric feet in the first, second and fifth line and two metric feet in the third and fourth line. A *metric foot* describes the basic repeating rhythmic unit of a verse, which is the anapest (`001`) for limericks. Accordingly, three metric feet correspond to three anapests - and to a verse with nine syllables, for which the stress pattern can be represented as `001 001 001`. In the same way, two metric feet represent a verse with six syllables and the stress pattern `001 001`. This standard stress pattern of a limerick is depicted in figure 5.2a. Note that the example limerick from my training data depicted in figure 5.1 exhibits this exact structure.

However, I found many variations of this structure in poems that can still be considered well-defined limericks. For example, a verse can end with an incomplete foot and still be considered to incorporate the anapestic meter. In poetic prosody, this phenomenon is called *catalexis*. The same applies to verses that drop an unstressed syllable from the beginning of the line – such a line is called an *acephalous* or headless line.

With these rules in mind, I worked with the limericks in my training data and designed a stress pattern that includes catalectic and acephalous lines, hence allowing more variance than with the traditional limerick structure. The pattern is shown in figure 5.2b: A bracketed zero represents an optional unstressed syllable that can either occur in the line or be omitted.

```
001 001 001
001 001 001
001 001
001 001
001 001 001
```

(a) Standard stress pattern

```
(0)01 001 001 (0)(0)
(0)01 001 001 (0)(0)
(0)01 001 (0)
(0)01 001 (0)
(0)01 001 001 (0)(0)
```

(b) Extended stress pattern

Figure 5.2: Stress patterns for limericks.

From this extended stress pattern, we can draw conclusions concerning two parameters that need to be defined in order to evaluate metric properties: The allowed number of syllables per verse and the required stress pattern for a verse, based on its syllable counts. Based on this pattern, we can see that the first, second and fifth line must consist of 8-11 syllables and the third and fourth line of 5-7 syllables. Accordingly, I defined criterion (e) in abstract 5.1 based on these findings.

So we now have the additional information we needed about the allowed number of syllables per verse. Furthermore, we can obtain the possible stress patterns for each allowed syllable count from figure 5.2b as well: For a verse with five syllables, for example, there is only one possibility, `01 001`. For a verse with six syllables, there exist two allowed stress patterns, namely `001 001` and `01 001 0`. In table 5.2, all possible stress patterns for each allowed syllable count are depicted, including an actual example verse from the training data as well as its phonetic representation.

Metric Score

We now want to implement a metric score that is able to capture by how much a limerick fulfills the metric restrictions imposed by the anapest.

I will illustrate my implementation by using an example verse from table 5.2: Suppose our current verse is *of the clashing of armor*. First, we count the syllables of this verse. Luckily, with phonetic representations, each vowel sound represents exactly one syllable, so we just have to count the occurring vowels in the phonetic representation. After obtaining the syllable count, in this case seven, we look up the metric pattern for this particular syllable count, which is `001 001 0`. We then extract the obligatorily stressed syllables in the sequence, in this case three and six, and check these positions in the phonetic representation of the verse. If all these obligatory positions are actually stressed, we say that this verse incorporates the correct metric pattern.

Syl	Stress Pattern	Example Verse	Phonetic Representation
5	01 001	has **star**ted to **spoil**	[HH AE1 Z, S T **AA1** R T AH D, T UW1, S P **OY1** L]
6	001 001	not a **flow**er was **left**	[N AA1 T, AH1, F L **AW1** ER, W AA1 Z, L **EH1** F T]
	01 001 0	it's **not** an il**lu**sion	[IH1 T S, N **AA1** T, AE1 N, IH L **UW1** ZH AH N]
7	001 001 0	of the **cla**shing of **ar**mor	[AH1 V, DH AH1, K L **AE1** SH IH NG, AH1 V, **AA1** R M ER]
8	01 001 001	who **drove** the in**ha**bitants **mad**	[HH UW1, D R **OW1** V, DH AH1, IH N HH **AE1** B AH T AH N T S, M **AE1** D]
9	001 001 001	I a**wait** the a**larm** clock's shrill **tug**	[AY1, AH W **EY1** T, DH AH1, AH L **AA1** R M, K L AA1 K S, SH R IH1 L, T **AH1** G]
10	001 001 001 0	ursa **mi**nor's the **best** constel**la**tion	[ER1 S AH, M **AY1** N ER Z, DH AH1, B **EH1** S T, K AA N S T AH L **EY1** SH AH N]
	01 001 001 00	are **apt** to be**come** ange**lo**logists	[AA1 R, **AE1** P T, T UW1, B IH K **AH1** M, AE N JH EH L **AA1** L AH JH IH S T S]
11	001 001 001 00	they com**plained** of a **man** from con**nect**icut	[DH EY1, K AH M P L **EY1** N D, AH1 V, AH1, M **AE1** N, F R AH1 M, K AH N **EH1** T AH K AH T]

Table 5.2: Possible stress patterns for each syllable count that result in the anapestic meter.

If a necessarily stressed vowel in the representation is not stressed, and there exists another possible metric pattern, we try out this pattern. Otherwise, and in case the second pattern also does not match, we say that this verse does not incorporate the correct metric pattern.

To compute the metric score of a complete limerick, we apply this test to all five lines in the poem. For each erroneous pattern in a line, we subtract 0.2 from the perfect score. If the poem consists of less than five lines, 0.2 are subtracted for each missing line, accordingly. For example, if one of our poems receives a metric score of 0.6, we know that the metric pattern in 3 out of 5 verses is correct. The possible score values are illustrated in table 5.3.

Verse Count Score

Similarly, I compute a score to measure to what degree the syllable counts in each poem are appropriate. This corresponds to the fifth criterion concerning the allowed number of syllables per verse. For each verse in the poem, I examine whether the syllable count lies in the range of allowed syllable counts, which is 8-11 syllables for verse one, two, and five and 5-7 syllables for verse three and four.

If the syllable count is too high or too low in a verse, we again subtract 0.2 from the perfect score, just as we did when calculating the metric score. Accordingly, a limerick with a verse count score of 0.8 would represent that 4 out of 5 verses in this poem fulfill their corresponding length restrictions in terms of syllable counts. The possible score values are depicted in table 5.3 as well.

Score value	Metric score	Verse count score
	Number of verses with correct stress pattern	Number of verses with correct number of syllables
1.0	5/5	
0.8	4/5	
0.6	3/5	
0.4	2/5	
0.2	1/5	
0.0	0/5	

Table 5.3: Possible values for the metric score and verse count score of a limerick.

5.3.3 Rhyme Score

To to be able to evaluate the rhyme scheme of a poem, we must first formalize what a rhyme actually is. I found that in order for two words to rhyme with each other, the part from the stressed syllable up to the end of the word must be the same in their phonetic representations. For example, if we have the rhyme *dedication* (D EH D AH K EY1 SH AH N) and *expectation* (EH K S P EH K T EY1 SH AH N), the stressed syllable is EY1, and the identical substring from there up to the end of both words is EY1 SH AH N. I found this property to be true for all perfect rhymes. From now on, I will refer to this particular substring from the stressed syllable up until the end as the *rhyming part* of a word. Accordingly, if we want to test whether two words rhyme with each other, we have to extract their rhyming parts and compare them.

Similar to the metric score and verse count score, if we want to test whether a limerick meets the required rhyme scheme, we might be interested in a more fine-grained analysis. I therefore decided to represent this criterion with a score as well. However, I also wanted to be able to model each single rhyme in a way that reflects a degree of rhyme rather than a simple boolean value (as opposed to the two scores I explained above, where I compute an overall score for a poem based on boolean values for each verse). So in order to explain how the overall rhyme score of a limerick is derived, I must first define how I compute a single rhyme between two words.

As already mentioned, the rhyme computation of two words is carried out by comparing their rhyming parts. But there are several relevant cases that need to be covered separately: Firstly, a *perfect rhyme* between two words occurs if both words have the exact same rhyming part, but the substring that corresponds to the rest of the word before the rhyming part is different. If this is the case, we return the best possible rhyme score, 1.0.

If the rhyming part as well as the substring before the rhyming part are equal in both words, this effectively means that we have the exact same phonetic representation – and probably the same word as well. Just using the same word twice would not be considered rhyming, so in this case, we return the worst score, 0.0.

Similar phonemes	Imperfect rhyme example
AE	bet (B **AE1 T**)
AH	but (B **AH1 T**)
AA	bot (B **AA1 T**)
AW	bout (R **B AW1 T**)
AO	bought (B **AO1 T**)
UW	loot (L **UW1 T**)
UH	foot (F **UH1 T**)
IY	beat (B **IY1 T**)
IH	bit (B **IH1 T**)
M	sum (S **AH1 M**)
N	sun (S **AH1 N**)
NG	sung (S **AH1 NG**)
T	seat (S **IY1 T**)
D	seed (S **IY1 D**)
DH	breathe (B R **IY1 DH**)
S	fuss (F **AH1 S**)
Z	buzz (B **AH1 Z**)
P	map (M **AE1 P**)
B	cab (K **AE1 B**)
K	speak (S P **IY1 K**)
G	league (L **IY1 G**)
F	thief (TH **IY1 F**)
V	leave (L **IY1 V**)

Table 5.4: Similar phoneme groups.

Apart from perfect rhymes, there are also *imperfect rhymes*, sometimes referred to as half-rhymes. There exists no formal definition of imperfect rhymes, but the general consensus is that such a rhyme is characterized by similar-sounding phonemes. To that effect, if two words incorporate an imperfect rhyme, the phonemes in their corresponding rhyming parts are not the same, but similar. I therefore formally defined an imperfect rhyme between two words based on two properties: Firstly, the length of both rhyming parts must be the same. And secondly, if phonemes at the same position in these two rhyming parts differ, they have to be similar phonemes. Which phonemes can be considered similar was defined manually. In table 5.4, all similar phoneme groups are depicted, including an example imperfect rhyme.

I decided to give an imperfect rhyme a perfect score of 1.0 as well: The qualitative difference in perfect and imperfect rhymes is often not attached any importance or not even perceived by the untrained ear, especially in the case of unvoiced consonants and their voiced counterparts.

For all other cases that are neither a perfect nor an imperfect rhyme, I compute the edit distance to measure by how much the rhyming parts differ from each other. The edit distance of two strings is generally defined as the minimum number of operations required to transform one string into the other. Usually, the allowed operations include removing, inserting or substituting characters in the string – the only difference in my implementation is that instead of characters, I use phonemes (or more precisely, indices that correspond to phonemes) that we can remove, insert or substitute. Each edit distance value

for two words corresponds to a rhyming score: An edit distance of one yields a 0.9 rhyming score because it is still very similar - but from there on, we again subtract 0.2 for each additional edit distance operation. This means that for an edit distance of six or higher, the score will be 0.0. The rhyme scores of border cases and some additional examples are illustrated in 5.5. For the sake of space, I did not include an example for every possible rhyming score.

Score	First word	Second word	Comment
1.0	blue (B L **UW1**)	do (D **UW1**)	perfect rhyme (best score)
1.0	thief (TH **IY1 F**)	leave (L **IY1 V**)	imperfect rhyme (best score)
0.9	betray (B IH T R **EY1**)	fail (F **EY1 L**)	edit distance = 1
0.7	doubt (D **AW1 T**)	court (K **A01 R T**)	edit distance = 2
0.0	blue (B L **UW1**)	blue (B L **UW1**)	same word (worst score)
0.0	blue (B L **UW1**)	[ ]	one line missing (worst score)

Table 5.5: Example rhyme scores for two words.

Accordingly, we obtain a score that describes the degree of rhyme between two words. Now we still have to use this mechanism to compute an overall rhyming score for the complete poem that measures whether it fulfills the AABBA rhyme scheme. Evaluating this scheme involves four cases: Whether verse one rhymes with verse two, whether verse three rhymes with verse four, whether verse one rhymes with verse five, and whether verse two rhymes with verse five. Therefore we can simply compute these partial scores and compute an overall rhyme score according to following formula:

$$\frac{score_{1,2} + score_{3,4} + score_{1,5} + score_{2,5}}{4}. \tag{5.1}$$

For example, if we have a poem with three perfect rhymes and one rhyme with edit distance one, we would compute $(1.0 + 1.0 + 1.0 + 0.9)/4 = 0.9857$, which would correspond to the overall rhyme score of this limerick.

5.3.4 Verse Three and Four Shorter

The last criterion still missing is whether verse three and four in a poem are shorter in terms of syllable counts than the other verses. However, this is very easy to implement - we already obtained the syllable counts for each verse and just have to compare them. I defined that each other verse has to be at least one syllable longer than verse three and four. This also corresponds to the extended stress pattern in 5.2b and the allowed number of syllables per verse, where verse three and four can consist of seven syllables at most and the other verses have to consist of at least eight syllables. By testing these conditions, we obtain a boolean value for the limerick.

5.4 Summary

In this chapter, I have explained my approach on evaluating the quality of limericks and explained how I derive and implement the corresponding criteria. An overview of the five final criteria is given in table 5.6. This means that we now have a method to evaluate the generated poems of different trained models and maybe, we will see how the choice of different model parameters influences the evaluation scores.

	Criterion	Possible values
1	five verses	true, false
2	verse 3 and 4 shorter	true, false
3	verse count score	0.0 - 1.0
4	metric score	0.0 - 1.0
5	rhyme score	0.0 - 1.0

Table 5.6: Final criteria for evaluating the well-formedness of a limerick.

Implementation and Experiments

In this chapter, I illustrate my neural network implementation, present the experiments I conducted, and discuss the results. Furthermore, I will address the limitations of my approach and suggest possible improvements.

6.1 Processing Pipeline

The single processing steps can be summarized as follows: After preprocessing the training data, the neural network is trained, saved to a file, and loaded as often as needed to generate any number of limericks for any sampling temperature. To evaluate the generated limericks, I obtain their phonetic representations and run an evaluation script that automatically computes the evaluation scores.

6.1.1 Preprocessing

I obtained a pre-built limerick corpus from Github[1] that was assembled by using Scrapy to collect limericks from various poetry websites. All words were already converted to lowercase. As a preprocessing step, I removed all poems that did not consist of exactly five lines because I considered them bad examples for the limerick structure. Interestingly, I found that the poems with more or less than five lines were deliberately mocking the traditional structure of limericks.

I also remove all punctuation except the hyphen and apostrophe prior to training. Whether a neural network is able to learn punctuation is not part of my evaluation, and by removing it we can reduce the vocabulary size. The hyphen is preserved because it is used for compound words, so the network might use it during generation to make up more creative neologisms. The apostrophe represents possession or omission and hence, grammatical functions. I also keep newline symbols, since they specify line breaks and therefore express the explicit limerick structure we try to learn. After preprocessing, my training data consists of 90,395 limericks and a vocabulary of 33 characters.

6.1.2 Model Implementation

I decided to employ a character-level language model for this task. In principle, it would also make sense to use a word-level model, but I assumed that by combining characters instead of words, such a model would yield more interesting and creative results.

The system was implemented using the deep learning library Keras[2] with TensorFlow as backend. The architecture corresponds to a multi-layer LSTM with an LSTM input layer, a time distributed output layer, softmax[3] activation functions, categorical crossentropy as loss measure, and RMSprob as optimizer. The model has 500 hidden dimensions in each layer, a fixed batch size of 50, and was trained for 200 epochs.

One aspect of my implementation that is particularly important to mention concerns the structure of the training examples: I treat the training data as one large chunk of text, and split it at a particular length specified by a parameter called `SEQ_LENGTH`. This way, the resulting training examples are all of the same length, namely the number of chars specified by this parameter. This also means that poems

[1] `https://github.com/sballas8/PoetRNN`

[2] `https://keras.io/`

[3] ReLU would actually be the recommended default, but I did not run into any problems with exploding or vanishing gradients with this model, and therefore kept the softmax activation function.

will be split up in the middle. However, I added a begin-of-limerick and end-of-limerick marker before and after every poem to capture the structure of the underlying poems in the text. This approach is actually a strong disadvantage that should be addressed: We can assume that each single limerick in the data corresponds to a single training example of varying length, and it should be treated that way. However, after implementing such a model for variable-length training examples, including padding to fill up the shorter training examples with dummy zero vectors, I ran into problems with Keras: It started to only produce NaN values as error during the training process. I suspected that this behavior was due to exploding gradients and tried different methods such as applying gradient clipping, using different activation functions, and trying out different optimizers. Unfortunately, none of these methods could solve this problem. I therefore decided to abandon this approach because my first implementation already seemed to yield interesting results.

I trained several different models where the previously mentioned parameters were held constant, but a varying number of hidden layers and different values for `SEQ_LENGTH` were used. These experiments and the best trained model are described in section 6.2.2.

6.1.3 Generation Process

The generation process is executed character by character: We first start with the begin-of-limerick marker and predict the most probable next char based on our trained model weights. We then append this char to the sequence, and predict the most probable next char for this sequence of length two, and append it again. This character-wise generation is carried out until either the end-of-limerick marker is generated, or we reach a length of 200 chars. This default length was chosen because most limericks consist of about 150 to 170 characters. If the model should sometimes erroneously generate limericks with more than five lines, we can capture this fact for evaluation with this preset length, but we do not end up with sequences that could potentially be very long. The begin and end markers are just considered an aid in the generation process, so they are removed from the sequence before returning the complete poem.

The generation introduces a parameter that I briefly mentioned, namely the *sampling temperature*. In text generation tasks, we usually include a mechanism for controlling the variety in the output. Otherwise, if we start with a begin-of-limerick char and only predict the most probable next char, we would always end up with the same poem being produced - because there is just this one most probable sequence for the learned model parameters. To prevent this, we reweight the probability distribution to a certain temperature, and sample the next character according to this reweighted distribution. Accordingly, a low temperature makes the predictions more confident, but also more repetitive. In contrast, a high temperature will yield diversity, but at cost of more mistakes: It might invent creative neologisms, but also make spelling or grammar mistakes.

6.1.4 Evaluation Process

To obtain phonetic representations, I use the G2P tool (Reichel, 2012, 2014) that is provided via a web service[4]. I found that the tool does not preserve newline characters in the generated limerick text files - therefore I convert the limericks to a format in which single lines and single poems are annotated with HTML tags. When I upload these reformatted poems to the website, these tags are interpreted as annotations and preserved.

It is important to note there is a capacity limit for uploading data on the website. In my case, I found this limit to be roughly equal to 2,000 limericks. To evaluate a model, I therefore generate 2,000 limericks, obtain the phonetic representations from the website, and then run my evaluation script on the data that computes an overall score for this model by averaging over all limericks.

[4] `https://clarin.phonetik.uni-muenchen.de/BASWebServices/interface/Grapheme2Phoneme`

6.2 Experiments

6.2.1 Predictions

Before presenting the experimental results, I would like to establish a few assumptions regarding the limerick properties the network might be able to learn. I presume that it should definitely be able to learn three of the five criteria: The appropriate number of lines per poem, the appropriate number of syllables per verse, and that line three and four are always shorter.

The first criterion just corresponds to a pattern of newline characters in the data that can be learned through training. The other two criteria involve the syllable count of a line, which is not explicitly visible in the training data – but it is likely that there also exists an identifiable pattern between char sequences and their corresponding syllable counts in a line. Accordingly, at least in most cases, a shorter sequence of chars in a line should also correspond to fewer syllables.

It is however not clear whether the model will be able to learn the correct stress pattern and rhyme scheme. These criteria are based on the pronunciation of words, and the character-level model does not receive any information about phoneme representations at training time. For most languages, it is reasonable to assume that the pronunciation of a word corresponds to its spelling. But unfortunately, English spelling is very irregular. In terms of stress patterns, we have the additional problem that the evaluation mechanism is not very robust – as soon as one erroneous pronunciation is generated for a word, even if the actual pronunciation would be correct, the whole line will not match the required stress pattern anymore. We can check this assumption by computing a metric score for the training data corpus – these limericks should in principle yield better metric results than the generated limericks. If this is not the case, we will know that it is probably due to incorrect phonetic representations.

I furthermore suspect that the character-level model will not be able to learn the inherent rhyme scheme due to the irregularities of English spelling. I found several examples of such irregular rhyme pairs in the training data, such as: *through - true*, *said - spread*, *pride - cried*, *by - eye*, *alone - known*, *guessed - test*, *mean - obscene*, *applied - guide*, *roots - brutes*, *coward - powered*, *bear - where*, and many more. I would argue that it is not possible to infer from this spelling that these words actually rhyme.

6.2.2 Experiments

I carried out three different kinds of experiments. First, I computed the evaluation scores for the training data. Since the G2P web service has an upload limit that roughly equals 2,000 limericks, I could not upload the complete training data corpus. Instead, I randomly extracted 2,000 different limericks from the data to obtain three different training data subsets. I reformatted these data sets and uploaded them to the website to obtain their phonetic representations. Afterwards, I computed the evaluation scores for each data set separately and averaged over them to get an overall estimate of the training data scores.

Then, I trained six different neural networks that varied in their number of hidden layers and values for `SEQ_LENGTH`, while the other parameters were held constant. Afterwards, I sampled 2,000 poems with a temperature of 0.2 for each of these models and evaluated the generated poems. The best scores were obtained by a model with four hidden layers and a `SEQ_LENGTH` of 400 chars.

Lastly, I used this model to sample more poems at higher temperatures (0.5, 0.8, and 1.0) to see how this parameter influences the evaluation scores. For each temperature, I generated 2,000 more poems and computed their corresponding scores. The evaluation scores of my experiments are depicted in table 6.1.

	Training data	**Trained model**			
Evaluation criterion		temp = 0.2	temp = 0.5	temp = 0.8	temp = 1.0
5 verses	**100%**	99.00%	98.7%	96.05%	94.31%
line 3 and 4 shorter	97.22%	**98.50%**	96.95%	92.56%	86.48%
verse count score	98.35%	**99.28%**	98.56%	96.37%	93.87%
metric score	81.57%	**92.98%**	87.56%	77.95%	69.10%
rhyme score	**91.65%**	62.31%	58.21%	52.87%	48.20%

Table 6.1: Evaluation scores of training data and best trained model at different temperatures.

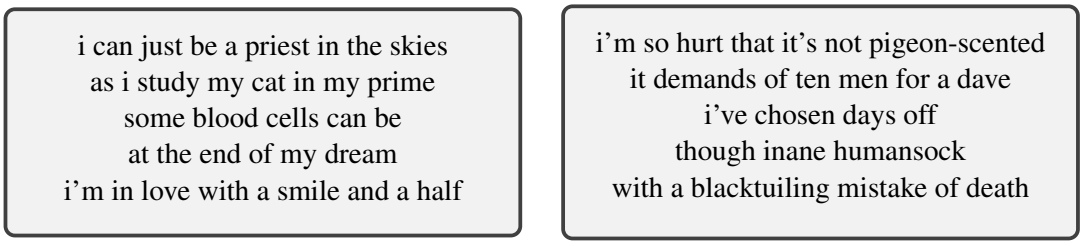
6.2.3 Results and Discussion

As estimated, at least for low temperatures, the model is able to learn four of the five limerick criteria, but fails to learn the rhyme score. At a temperature of 0.2, 99% of the 2,000 generated limericks consist of five verses, and in 98.5% line three and four are shorter than the other lines, which is even slightly better than in the randomized training data subsets. The overall verse count score of 99.28% is also slightly better than the score for the training data.

It is important to note that in the training data, both metric score as well as rhyme score are relatively low, compared to the other values. I assume that this is due to frequent incorrect phonetic representations generated by the G2P tool. Some of these errors are plausible: For words such as *every* and *our*, the tool generates the representations `EH1 V ER IY` and `AW1 ER`, respectively. However, they are usually pronounced `EH1 V R IY` and `AW1 R`, and are therefore intended to have one syllable less in the metric pattern. To that effect, the metric pattern that the author tried to convey cannot be captured. But there are also errors occurring in phonetic representations that are somewhat less plausible: For example, for the word *schoolgirls*, the representation `S1 K L G ER L Z` is generated, although `S` is a consonant and hence cannot be stressed. There are several of these invalid stress positions on consonants that get generated by the tool. In such cases, the word will erroneously have one syllable less, and exhibit an incorrect stress representation. Of course, the metric score is much more sensitive to these errors than the rhyme score: The rhyme score for a line will be low if such an error occurs in the last word of the line, but the metric score will be low if an error occurs anywhere in the line.

Interestingly, the metric score is significantly higher in the poems generated by the neural network as long as the temperature is not too high. I suspect that this is due to the network's tendency to mainly sample short words at lower temperatures. Accordingly, the corresponding short phonetic representations yield a lower error potential. Furthermore, we can observe that all scores decline for higher temperatures. This is not surprising, since a higher temperature introduces more variety to the generation process, and the resulting poems will differ more noticeably from the training data, and possibly, the actual limerick structure. At a temperature of 1.0, 94.31% of the generated poems still consist of five lines, and the verse counts still have a relatively high score of 93.87% as well. The metric score drops to 69.10%, most likely because it is the criterion most sensitive to errors: High temperature poems contain longer words with longer corresponding phonetic representations that have a greater potential of not matching the required stress pattern. This might either be due to the tool that produces more incorrect representations, or because such a high variance simply does not enable the model parameters to capture this criterion anymore.

So we can conclude that at a low temperature, the best trained model is able to learn all criteria except the rhyme scheme. Even at the lowest temperature of 0.2, the rhyme score amounts to 62.31%. This score roughly corresponds to a poem with a phonetic edit distance of 2.5 for every occurring rhyme. At best, this can be interpreted as partial phonetic similarity, but it does not come close to incorporating a strict rhyme scheme. Two example limericks are depicted in figure 6.1: We can observe the network's tendency to sample longer words at higher temperatures, which leads to creative neologisms, as well as the phonetic similarity in word pairs such as *skies - prime*, *be - dream*, and *off - sock*. More generated poems in which the model actually managed to generate a few rhymes are included in appendix A.



(a) Example with temperature = 0.2

(b) Example with temperature = 1.0

Figure 6.1: Limericks generated by the best model at different temperatures.

6.3 Limitations and Future Work

Lastly, I want to address the limitations of my approach, especially in terms of model architecture and evaluation, and suggest some improvements.

As already mentioned, splitting the limericks during training instead of treating each limericks as single training example is a considerable disadvantage. I found that the model was sometimes able to generate isolated rhymes and similar-sounding words, and therefore assume that improving this mechanism would also yield better rhyme scores. When we treat the corpus as one large text, effectively all other criteria can be learned nevertheless because they correspond to repeating patterns in the data. However, rhymes cannot be captured that way because they are different in every poem. In order to reliably reproduce rhymes in English, it would probably be necessary to train the model on phonetic representations, due to the irregularities of English spelling. But then, we would also need a separate model to convert the generated phonemes back into their corresponding orthographic representations. If we want to avoid training the model on phoneme representations and nevertheless learn the rhyme scheme, we might also try to modify the evaluation function in a way that it can be used as a cost function for the model.

There are also several limitations that concern the evaluation function: Firstly, we need to address the errors occurring in phonetic representations. We could try out other tools that possibly produce less errors, or include additional tools. For instance, we could integrate the CMU Pronouncing Dictionary[5], which can be downloaded as a list. This list contains multiple representations for many words, for example the additional pronunciations of *our* and *every* I previously mentioned. To implement this, we could look up words in the dictionary first, and in case they do not occur there, use the G2P method to generate unknown representations. By doing so, there should occur fewer mistakes in the data.

Secondly, we could refine the rhyme score, particularly in reference to two aspects: We could adjust the score values that are assigned to a particular edit distance, since two rhyming parts need an edit distance of six to receive the worst score. In practice, such a high value occurs infrequently because the rhyming parts of words are often shorter. Furthermore, we could integrate the process of comparing similar phonemes into the edit distance computation. For example, we could introduce a lower substitution cost of 0.5 if a phoneme can be replaced by a similar-sounding phoneme.

There is a potential problem with the metric score as well I have not addressed yet: To test whether a line corresponds to the required stress pattern, we extract the obligatory stress positions and compare them to these positions in the poem line. The pronunciation of a line is obtained by combining all phonetic representations. This means that a line only consisting of one-syllable words will always be classified as correct. To illustrate this, let us compare this hypothetical example: The lines "I don't **care** what you **say**" and "I do **not** care what **you** say at **all**" would both formally incorporate the correct stress pattern for their respective syllable count and receive a perfect score. However, the second example actually sounds strange to the reader because the stress positions do not correspond to the semantically relevant words in the line. In order to distinguish between such cases, we could include a stop word list and subtract a penalty from the score each time the required stress position corresponds to such a stop word.

It is also important to stress that my approach on evaluating the quality of poetry only incorporates the aspects of poetic well-formedness that I defined in section 5.1. This includes most aspects of syntax and phonetics that apply to limericks, but for instance excludes grammaticality, semantics, textual coherence and many more aspects that contribute to the conception of poetic well-formedness. While a grammaticality test might be relatively easy to implement and to evaluate automatically, the other aspects would very likely require a human judgment study.

Since the trained model only relies on statistical estimation about character distributions that are extracted automatically from the training data, I assume that it would also be able to reproduce different poems in different languages, as long as it is provided with enough training data. In the future, I would like to test this model for other languages and poetic forms. However, this involves developing another evaluation method specifically designed for this task, as for now, due to the greatly differing formal requirements of different poetic forms, it does not seem possible to cover more than one poetic form with such an automatic approach.

[5] `http://www.speech.cs.cmu.edu/cgi-bin/cmudict`

Conclusion

In this thesis, I have developed an automatic approach on poetry generation with recurrent neural networks by employing a character-level multi-layer LSTM to generate limericks.

I found that due to the strong constraints that poems impose in terms of semantics, syntax, phonetics, and lexical choice, the automatic generation of poetry proves to be a very challenging task. However, this additional difficulty makes it even more interesting to design a system that could potentially learn, capture and reproduce all these language constraints just by automatically extracting patterns from the training data. That such a system furthermore tries in some way to replicate human creativity makes the task particularly interesting as well. Yet, it can be concluded that such an approach will not teach us anything about the properties of creativity, especially since the term currently lacks a general definition and cannot be defined objectively.

Concerning the output evaluation of a poetry generation model, I found that it is possible to automatically evaluate several properties that correspond to the poetic well-formedness of a particular poetic form. However, capturing the overall quality of poems in terms of fluency, coherence, meaningfulness, and general poeticness seems to require a human judgment study, as the current literature on this topic has shown as well.

In reference to my own implementation, I found that my approach mainly suffers from three occurring problems: Firstly, the automatically generated phonetic representations are often incorrect. This distorts the evaluation scores, especially in terms of metric properties, as an evaluation on the gold standard data has shown. Secondly, I treat the training data corpus as one large text that is split up into training examples at a particular char length instead of treating every individual limerick as single training example. Although the evaluation criteria that correspond to repeating patterns in this text can be learned nevertheless, the rhyme scheme cannot be captured this way, because the occuring rhymes are different in every poem. And lastly, even if every poem would be represented as single training example in the implementation, it is not clear whether a character-level model could learn the rhyme scheme reliably due to the irregularities of English spelling. It might be necessary to train the model on phonetic representations instead.

However, I conclude that a character-based model might still be sufficient to capture the rhyme scheme of poetry in other languages that exhibit a phonemic orthography and a more regular spelling, respectively, such as Spanish, Italian, Turkish, or Finnish.

References

The CMU pronouncing dictionary. http://www.speech.cs.cmu.edu/cgi-bin/cmudict. [accessed: 09-August-2018].

Grapheme2Phoneme web service. https://clarin.phonetik.uni-muenchen.de/BASWebServices/interface/Grapheme2Phoneme. [accessed: 09-August-2018].

Michael Auli, Michel Galley, Chris Quirk, and Geoffrey Zweig. Joint language and translation modeling with recurrent neural networks. In *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing (EMNLP 2013)*, pages 1044–1054, 2013.

Sam Ballas. PoetRNN. GitHub Repository. https://github.com/sballas8/PoetRNN. [accessed: 09-August-2018].

Greg Bickerman, Sam Bosley, Peter Swire, and Robert M. Keller. Learning to create jazz melodies using deep belief nets. In *Proceedings of the International Conference on Computational Creativity (ICCC 2010)*, pages 228–237, 2010.

Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using RNN encoder-decoder for statistical machine translation. In *Proceedings of 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP 2014)*, pages 1724–1734, 2014.

François Chollet et al. Keras: The python deep learning library. https://keras.io, 2015. [accessed: 09-August-2018].

Belén Díaz-Agudo, Pablo Gervás, and Pedro A. González-Calero. Poetry generation in COLIBRI. In *Proceedings of the 6th European Conference on Advances in Case-Based Reasoning (ECCBR 2002)*, pages 73–87, 2002.

John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12:2121–2159, 2011.

Ahmed Elgammal, Bingchen Liu, Mohamed Elhoseiny, and Marian Mazzone. CAN: Creative adversarial networks, generating "art" by learning about styles and deviating from style norms. *arXiv preprint arXiv:1706.07068*, 2017.

Leon A. Gatys, Alexander S. Ecker, and Matthias Bethge. A neural algorithm of artistic style. *arXiv preprint arXiv:1508.06576*, 2015.

Pablo Gervás. WASP: Evaluation of different strategies for the automatic generation of spanish verse. In *Proceedings of the AISB'00 Symposium on Creative & Cultural Aspects and Applications of AI and Cognitive Science*, pages 93–100, 2000.

Pablo Gervás. An expert system for the composition of formal spanish poetry. *Journal of Knowledge-Based Systems*, 14:181–188, 2001.

Marjan Ghazvininejad, Xing Shi, Yejin Choi, and Kevin Knight. Generating topical poetry. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing (EMNLP 2016)*, pages 1183–1191, 2016.

Xavier Glorot, Antoine Bordes, and Yoshua Bengio. Deep sparse rectifier neural networks. In *Proceedings of the 14th International Conference on Artificial Intelligence and Statistics (AISTATS 2011)*, pages 315–323, 2011.

Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT Press, 2016. `http://www.deeplearningbook.org` [accessed: 09-August-2018].

Alex Graves, Abdel-rahman Mohamed, and Geoffrey Hinton. Speech recognition with deep recurrent neural networks. In *Proceedings of the 38th International Conference on Acoustics, Speech and Signal Processing (ICASSP 2013)*, pages 6645–6649. IEEE, 2013.

Charles O. Hartman. *Virtual muse: experiments in computer poetry*. Wesleyan University Press, 1996.

Jing He, Ming Zhou, and Long Jiang. Generating chinese classical poems with statistical machine translation models. In *Proceedings of the 26th Conference on Artificial Intelligence (AAAI 2012)*, pages 1650–1656, 2012.

D. O. Hebb. *The organization of behavior*. Wiley, New York, 1949.

Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997.

Jack Hopkins and Douwe Kiela. Automatically generating rhythmic verse with neural networks. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 168–178, 2017.

Long Jiang and Ming Zhou. Generating chinese couplets using a statistical MT approach. In *Proceedings of the 22nd International Conference on Computational Linguistics (COLING 2008)*, pages 377–384, 2008.

Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

Siwei Lai, Liheng Xu, Kang Liu, and Jun Zhao. Recurrent convolutional neural networks for text classification. In *Proceedings of the 29th Conference on Artificial Intelligence (AAAI 2015)*, pages 2267–2273, 2015.

Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521:436–444, 2015.

Bei Liu, Jianlong Fu, Makoto P Kato, and Masatoshi Yoshikawa. Beyond narrative description: Generating poetry from images by multi-adversarial training. *arXiv preprint arXiv:1804.08473*, 2018a.

Dayiheng Liu, Quan Guo, Wubo Li, and Jiancheng Lv. A multi-modal chinese poetry generation model. *arXiv preprint arXiv:1806.09792*, 2018b.

Theo Lutz. Stochastische texte. *Augenblick*, 4(1):3–9, 1959.

Hisar Manurung. *An evolutionary algorithm approach to poetry generation*. PhD thesis, University of Edinburgh, 2003.

Margaret Masterman. Computerized haiku. *Cybernetics, Art and Ideas*, pages 175–183, 1971.

Warren S. McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, 5:115–133, 1943.

Tomáš Mikolov, Martin Karafiát, Lukáš Burget, Jan Černockỳ, and Sanjeev Khudanpur. Recurrent neural network based language model. In *Proceedings of the 11th Annual Conference of the International Speech Communication Association (INTERSPEECH 2010)*, pages 1045–1048, 2010.

Yael Netzer, David Gabay, Yoav Goldberg, and Michael Elhadad. Gaiku: Generating haiku with word associations norms. In *Proceedings of the Workshop on Computational Approaches to Linguistic Creativity (CALC 2009)*, pages 32–39. ACL, 2009.

Michael A. Nielsen. *Neural networks and deep learning*. Determination Press, 2015. `http://neuralnetworksanddeeplearning.com/` [accessed: 09-August-2018].

Uwe D. Reichel. PermA and Balloon: Tools for string alignment and text processing. In *Proceedings of the 13th Annual Conference of the International Speech Communication Association (INTERSPEECH 2012)*, pages 1874–1877, 2012.

Uwe D. Reichel. Language-independent grapheme-phoneme conversion and word stress assignment as a web service. In *Elektronische Sprachverarbeitung. Studientexte zur Sprachkommunikation*, volume 71, pages 42–49. TUDpress, 2014.

Frank Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, 65(6):386–408, 1958.

Frank Rosenblatt. *Principles of neurodynamics*. Spartan Books, New York, 1962.

Hava T. Siegelmann and Eduardo D. Sontag. On the computational power of neural nets. *Journal of Computer and Systems Sciences*, 50(1):132–150, 1995.

Bob Sturm, João F. Santos, and Iryna Korshunova. Folk music style modelling by recurrent neural networks with long short term memory units. In *16th International Society for Music Information Retrieval Conference, late-breaking demo session*, 2015.

Tijmen Tieleman and Geoffrey Hinton. Lecture 6.5 - rmsprop. *COURSERA: Neural networks for machine learning*, 2012.

Naoko Tosa, Hideto Obara, and Michihiko Minoh. Hitch haiku: An interactive supporting system for composing haiku poem. In *Proceedings of the 7th International Conference on Entertainment Computing (ICEC 2008)*, pages 209–216, 2008.

Zhe Wang, Wei He, Hua Wu, Haiyang Wu, Wei Li, Haifeng Wang, and Enhong Chen. Chinese poetry generation with planning based neural network. *arXiv preprint arXiv:1610.09889*, 2016.

Bernard Widrow and Marcian E. Hoff. Adaptive switching circuits. *In 1960 IRE WESCON Convention Record*, 4:96–104, 1960.

Linli Xu, Liang Jiang, Chuan Qin, Zhe Wang, and Dongfang Du. How images inspire poems: Generating classical chinese poetry from images with memory networks. *arXiv preprint arXiv:1803.02994*, 2018.

Rui Yan. i, Poet: Automatic poetry composition through recurrent neural networks with iterative polishing schema. In *Proceedings of the 25th International Joint Conference on Artificial Intelligence (IJCAI 2016)*, pages 2238–2244, 2016.

Tom Young, Devamanyu Hazarika, Soujanya Poria, and Erik Cambria. Recent trends in deep learning based natural language processing. *arXiv preprint arXiv:1708.02709*, 2017.

Xingxing Zhang and Mirella Lapata. Chinese poetry generation with recurrent neural networks. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP 2014)*, pages 670–680, 2014.

Appendix

Some more selected limericks generated by the best model at temperatures 0.2, 0.5, 0.8, and 1.0 are presented here. Note that the network sometimes manages to produce a similar-sounding word pair or even a rhyme. However, there is almost never more than one of these word pairs or rhymes per poem, which suggests that the model suffers from the implementation approach where the corpus is split into training examples of a particular char length.

A.1 Temperature = 0.2

```
the candidate said with a smile
i adore yourself into a tide
they are hard to relate
to a start to convey
the archbishop will show us a treat

a distraction starts up to the ground
with a fine conversation of sound
all the state of a string
is a problem for wick
i think that your face is all right

the burger was angry and smart
and it wasn't a pain in the neck
he said what the hell
it's a change in my brain
the great butter is making me smile

a canadian sailor named death
was a star of the congress he said
it relates to your sins
and it's true that it sings
when you're driving a steery black thing
```

A.2 Temperature = 0.5

```
an angel has labeled my soul
in the air force and cons and my car
it's a deal of delight
and a pain in the night
that is deep in the oven of space
```

in the alps to the back of the pier
his father is lost to the sin
when he spoke as a fish
and the french disapprove
as he speaks from the cheeks and his mind

i'm a beaver but still i would choose
to define a fall cancel this soup
we're like parts of a group
from a high-pressure door
a few days of descending to cheese

i control all the windows i run
but it's deep in a bottomless treat
i would say what i need
it's called cornish indeed
and the concept of acid involved

i'm not anytime with a guy
who is famous for food in the sky
the rich mushroom booze
it seemed nasty dear friend
it's a source that's a concept i fear

a causal transparent professor
was enslaved at the moon in the north
the expression a fight
will be horribly bright
so white fracture the cargo when running

there once was a foot named skin death
who removed a surprising event
he's a criminal glass
that will wish to confuse
and advance on the way that you'll find

i sought a new plant and a boat
and a town of sweet cats to explore
the cramp walked to flow
and i now have a cow
she demands near the strain of my bowel

A.3 Temperature = 0.8

david sings for a contract committee
had a name called the converted eyegrand
it succeeds and enthralled
you don't like it you're screwed
as your horseyes can fly and amaze

after lunch for a limerick i practice
with candybiaste linguist doorbaskian
you ask what i am
for our priest came to town
i believe that i'm left in the bucket

my uncle's a fish since his teens
a man with a view still to thank
i'd dispendingly quit
as the lover you've signed
contrological still has been drydable

the warden with dinner was blind
seeking nuts in a simple attire
all those guys there's a word
fertile amplitude hill
made of crops the potatoes and salt

A.4 Temperature = 1.0

at the zoo i was married for years
unrendering whistles and pains
when the sheep is decayed
the destroyer jack-nancy
make a felon with god and it's neat

a bolt-lynic pallet is found
a deciduous marriage to plough
a circular hair
or a blue colored dart
which is also a quacropod here

the bartender follows the yard
of the ailmend in which called a maiden
in her business or spice
it's corruption is nice
and it gave me a clue in the past

a catfisher prof guards the fact
and funds nipple-as-diving degrees
the exchange guarantee
is tamango with squee
it's implied because anything sparks