LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN
CENTRUM FÜR INFORMATIONS- UND SPRACHVERARBEITUNG
STUDIENGANG COMPUTERLINGUISTIK

Bachelorarbeit

im Studiengang Computerlinguistik

an der Ludwig- Maximilians- Universität München

Fakultät für Sprach- und Literaturwissenschaften

Exploring Genetic Algorithms to optimize Convolutional Architectures for Slot Filling Tasks

vorgelegt von
Yannick Kaiser

Betreuer:	Nina Pörner
Prüfer:	Prof. Dr. Schütze
Bearbeitungszeitraum:	19. März - 28. Mai 2018

Selbstständigkeitserklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig angefertigt, alle Zitate als solche kenntlich gemacht sowie alle benutzten Quellen und Hilfsmittel angegeben habe.

München, den 28. Mai 2018

.....
Yannick Kaiser

Abstract

English

This bachelors thesis explores the possibility of optimizing Convolutional Neural Network architectures with the means of Genetic Algorithms. For this task, the example domain of Slot Filling Tasks is chosen. The theoretical backgrounds of Slot Filling Tasks and Convolutional Neural Networks are briefly discussed, before a more thourough examination of the components of a Genetic Algorithm is carried out. Finally, one such algorithm is developed, implemented and evaluated. These evaluations suggest that Genetic Algorithms are capable of carrying out a successful search for optimum solutions in a difficult solution space. The empirical results show that the genetic approach has succeeded in optimizing the structure of Convolutional Neural Networks.

Deutsch

Diese Arbeit untersucht die Möglichkeit, die Architektur von Convolutional Neural Networks mithilfe von Genetischen Algorithmen zu optimieren. Für diese Aufgabe wurden die Slot Filling Tasks als Beispieldomäne herangezogen. Zunächst werden die theoretischen Hintergründe von Slot Filling Tasks und Convolutional Neural Networks behandelt, bevor eine gründliche Untersuchung der Bausteine von Genetischen Algorithmen folgt. Schließlich wird im Rahmen der Arbeit ein solcher Algorithmus entwickelt, implementiert und evaluiert. Die Auswertung lässt darauf schließen, dass Genetische Algorithmen fähig sind, eine erfolgreiche Suche nach optimalen Lösungen auch in schwierigen Lösungsräumen durchzuführen. Die empirischen Ergebnisse zeigen, dass der Ansatz nützliche Architekturentscheidungen zur Optimierung von Convolutional Neural Networks getroffen hat.

Contents

1 Introduction

Scientific progress has many key components. Able researchers need to be equipped with time and sufficient funding. Machinery needs to be avaiable and reliable, for experiments are getting ever more sophisticated. But the golden key to scientific progress is, and has always been, inspiration.

Nature has always been an inspiring factor in many scientific areas, as one thing is trivially understood: If a structure or mechanism survives the evolutionary selectionism that is the history of our planet, it is probably robust and useful towards reaching a certain goal or towards providing a key advantage [8, 31, 40]. Especially evolution itself has repeatedly sparked scientific interest, up to a point where researchers take it into their own hand in labs around the world.

Ironically, by taking over evolution, one does strip it from its greatest asset: Because the ability to mimic evolutionary processes would lead to problem-solving techniques that require less creative human input instead of more. The desire to harness this asset has lead to the exploration and exploitation of so called Genetic Algorithms, algorithms that try to recreate evolutional mechanisms and try to be able to apply these to arbitrary problem domains.

On a parallel stream of events, nature stood model for one of the biggest contemporary fields of research and progress: Neuronal Networks and Deep Learning. With the advances in processing power during the last decade and the widespread adaptation of Deep Learning techniques, amazing progress has been made.

A very impressive recent example is Google Duplex, a Dialogue System so natural, that the called person is unable to tell it from a real person. And this is only 28 years after one of the first shared tasks on Spoken Language Understanding was given out to researchers.

This paper will combine all three: Genetic Algorithms, Deep Learning and a Spoken Language Understanding Task. The goal however is rather technical: Improving the architecture of a Deep Learning Neural Network, letting it solve the Spoken Language Understanding Task.

This section gives a brief overview over the content of this paper. After the general introduction in Chapter 1, Chapter 2 introduces the class of Slot Filling Tasks, picking the ATIS task as a standout example, because it will provide the input data for the practical parts of this thesis. Chapter 3 then outlines the special architecture of Convolutional Neural Networks. Chapter 4 gives an in-depth introduction to Genetic Algorithms. The natural processes that inspired these Algorithms are taken as a starting point in exploring the components of the Genetic Algorithm. Afterwards, some interesting use cases are depicted. Chapter 5 is dedicated to illustrating the experimental goal, theoretical buildup, and obstacles in the way of the Genetic Algorithm which will be used and evaluated in Chapter 6. Finally, Chapter 7 summarizes the experiment and lays out approaches for future work.

2 Slot Filling Tasks

In this section the concept of Slot Filling Tasks is being presented. The ATIS task is chosen as an example Slot Filling Task, as it will provide the dataset used in the practical evaluation of this paper.

2.1 Task Description

There are several archetypes of tasks in natural language processing (NLP), such as text tokenization, Part-Of-Speech tagging, named entity recognition, event extraction, relationship extraction and speech recognition.

Many of the tasks, especially those further down the NLP pipeline [18], can be interpreted as Slot Filling Tasks where a predefined *template* of information slots is to be filled with information from natural language text. In case of event extraction, the template could gather features describing any single event instance, like *event location* and *event date* (Table 2.1). In case of spoken language understanding systems [34], the template could try to gather all information needed to convert the utterance into a structured representation on wich the machine is actionable, e.g. by responding to the utterance in a useful manner.

Slot Filling Tasks can be solved using different approaches like instance extraction per template slot, sequence-to-sequence labelling or multi-layered combinations of those and other approaches.

[1]	Big detonation: Red car exploded yesterday in Birmingham
[2]	Prince Harry wed Meghan Markle in May 2018, historians remember.

Sentence	Event Type	Trigger Phrase	Event Location	Event Date
[1]	attack	car exploded	Birmingham	yesterday
[2]	marriage	Harry wed Meghan	London	May 2018

Table 2.1: Event extraction Slot Filling example

2.2 ATIS

Beginning 1990 the US Defence Advanced Research Agency (DARPA) put up the task of extracting airline schedules and related information from the Air Travel Information System (ATIS) dataset. It was built from sentences taken from spoken queries on flight-related information. [47, p. 19]

The ATIS task therefore is a spoken language understanding task (SLU) that requires semantic parsing. "The semantic parsing of input utterances in spoken language understanding typically consists of three [sub]tasks: domain detection, intent determination, and slot filling." [34, p. 1]

- **Domain Detection**

  In case of the ATIS dataset, domain detection is not necessary, as all data stems from a single known domain. Domain detection is normally treated as a semantic utterance classification problem. [34]

- **Intent Determination**

  Intent determination for the ATIS dataset has been a subject of steady improvements over the years. [47] give a concise overview of the related works.

  The ATIS dataset hosts a very skewed collection of intents, as can be seen in Figure 2.1. Over 70% of all utterances can be attributed to the *Flight* intent, while other intents, such as *Day_Name* or *Restriction* provide very sparse data. This imballance in data distribution poses a challenge for designing intent detection systems for the ATIS dataset, as well as for the actual Slot Filling Task further down the line.

Intent	Training Set	Test Set
Abbreviation	2.4%	3.6%
Aircraft	1.6%	0.9%
Airfare	9.0%	5.8%
Airline	3.4%	4.3%
Airport	0.5%	2.0%
Capacity	0.4%	2.4%
City	0.3%	0.6%
Day_Name	0.1%	0.1%
Distance	0.4%	1.1%
Flight	73.1%	71.6%
Flight_No	0.3%	1.0%
Flight_Time	1.2%	0.1%
Ground_Fare	0.4%	0.8%
Ground_Service	5.5%	4.0%
Meal	0.1%	0.6%
Quantity	1.1%	0.9%
Restriction	0.3%	0.1%

Figure 2.1: The frequency of intents for the training and test sets. [47, p. 20]

- **Slot Filling**

  Each intent presents a template to fill. Slot Filling hereby is "typically treated as a sequence classification problem in which contiguous sequences of words are assigned semantic class labels." [34] and has been approached with a wide range of discriminative and generative approaches. [33, 50, e.g.]

  Figure 2.2 shows an example ATIS utterance and the corresponding IOB[1] annotated slot categories. The label classes can have subclasses via linking them with a dot character.

please	list	all	first
o	o	o	B-class_type
class	flights	on	United
I-class_type	o	o	B-airline_name
from	Denver	to	Baltimore
o	B-fromloc.city_name	o	B-toloc.city_name

Figure 2.2: The frequency of intents for the training and test sets. [47, p. 20]

While the ATIS taks is largely regarded as solved, the main challenges that still remain to the ATIS Slot Filling Task are previously unseen sequences, very sparse training data for some labels, and (human) annotation errors in the provided data set. [47]

[1] Inside/Outside/Beginning of a continuous sequence of the same label

3 Convolutional Neural Networks

The Idea of Convolutional Neural Networks dates back into the early 1980s, when previous ideas of convolutional operations were taken out of the domain of biological neurocognition into the field of Neural Networks [19]. Human visual cells and their ability of pattern recognition (e.g. edges, corners) were the inspiration behind a convolutional operation that scans parts of a multi-dimensional input and applies a pattern recognition filter on each scanned area [9]. Figure 3.1 illustrates the convolutional operation, where a kernel of size (x, y) is shifted along a 2D input matrix to produce the output or feature matrix via matrix multiplication.

The following sections will present the general architecture of Convolutional Neural Networks, as well as highlight some of their use cases. A short discussion on the usefulness for Slot Filling Tasks will be given.

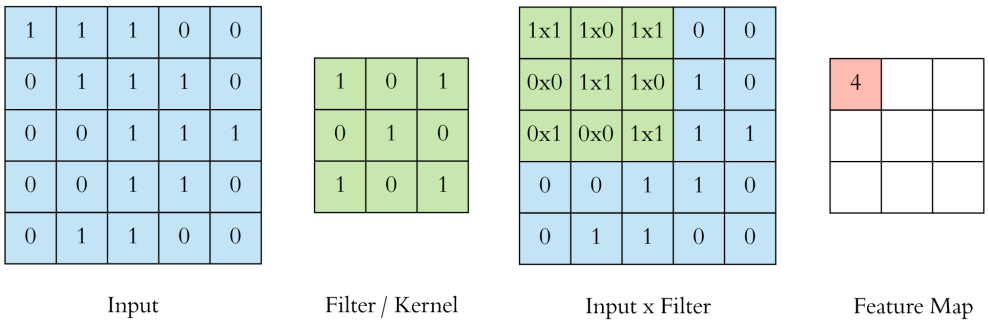


Figure 3.1: The basic convolutional operation. [14]

3.1 Architecture

Convolutional Neural Networks typically consist of one or more layers of stacked convolution and max pooling operations, followed by generally at least one densly connected layer before the output layer is reached. Figure 3.2 illustrates the aforementioned structure on the example of a simple network for sentiment analysis, omitting the hidden dense layer for clarity.

3.1.1 Embedding Layer

Convolutional Neural Networks for text processing typically use an embedding layer, either word-based or character-based. Word based embeddings project a word into a multi-dimensional vector space where each word gets mapped to a distinct feature vector. Due to the way these word vectors are trained, syntactic and semantic information is being encoded. Similar words get mapped to similar vectors. Therefore, the distance between resulting semantically or syntactically related vectors is small, letting the resulting vector space exhibit semantic and syntactic structures that can be exploited [35, p. 2].

A good example for word embeddings is word2vec[1] [35], a popular and competititve system for generating word embeddings. Here, word vector training is achieved either by letting a Neural Network predict a word from a bag-of-words context (CBOW) or by letting the network predict the context of a given word (skip-gram). Other approaches to

[1] `https://code.google.com/archive/p/word2vec/`

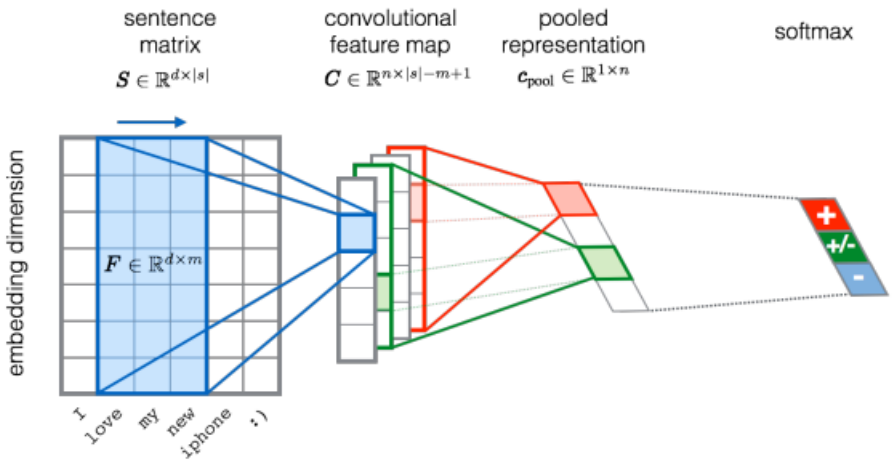


Figure 3.2: Example for a Convolutional Neural Network architecture for sentiment analysis. [44, p. 466]

word embeddings include GloVe[2] or fastText[3]. Pretrained word embeddings are avaiable from a variety of sources, including word2vec. They can bei used either as a default for retraining them during the training of the Convolutional Neural Network, or are used as a static lookup layer. In both use cases good improvements on network performance on language tasks have been reported, and the usage of word embeddings has become a standard procedure in natural language processing.

To use word embeddings as the input layer of a Neural Network, the word vectors are looked-up in the pretrained embeddings for any input sequence. From these vectors the input matrix (compare Figure 3.2) is constructed. The most important tunable parameter for word embeddings is their embedding size, i.e. the length of the word vectors. While shorter vectors are easier to train and limit input data complexity, longer vectors have superior capability in representing finer grained semantic and syntactic relationships.

3.1.2 Convolution and MaxPooling

Convolution applies a filter to a wandering window of the input data. For sake of simplicity, *2D-Convolution* is chosen as a starting point for the following considerations. The defining properties of Convolutional Neural Network layers are:

- **Kernel Size** The Kernel size defines the size of the moving filters. Let $S \in \mathbb{R}^{d \times |s|}$ be the size of the sentence input matrix, where s is the input sentence and d the embedding size. Then $F \in \mathbb{R}^{n \times m}$ is the filter matrix, where $n \leq d$ and $m \leq |s|$.

  When working on embeddings, it usually does not make sense to have the filter move along the axis of the embedding dimension, as the order of the embedding dimensions does not hold any usable information for most of the embedding techniques. Therefore, akin to Figure 3.2, when working on language data *1D-Convolution* is used, where $n = d$. Mathematically, 1D-Convolution still is 2D-Convolution, but the term has settled into every day use[4].

- **Strides** The strides value determines the way in wich the filters are advanced along the sentence input matrix. For regular 2D-Convolution, $(1, 1)$ strides are the most common setting, while bigger steps can be used especially when using large kernel sizes. By controlling the way the filter is advanced, the strides determine the output

[2] `https://nlp.stanford.edu/projects/glove/`
[3] `https://fasttext.cc`
[4] Many machine learning frameworks present pre-implemented 1D-Convolution layers

dimensions of the convolutional layer. For 1D-Convolution, the strides value in the embedding dimension is fixed at 0, effectively removing the second dimension of movement.

- **Filter Count** The third defining property of Convolutional Neural Network layers is the filter count. For each position of the kernel window in the input data, a seperate weight matrix is applied for each defined filter. As such, the filter count determines the amount of patterns which are searched at each sub-part of the input data. More filters lead to a wider range of trainable patterns, but make learning the patterns more difficult by greatly increasing the number of trainable parameters in the network.
- **Activation Function** Lastly, the output of the convolutional layer is moderated by a nonlinear activation function like softmax, tanh or ReLU variants [20].

The **max pooling** layer emloyed after each convolutional layer does two things simultanously: It creates invariance to spatial displacement in the input data and it reduces the resolution of the feature matrices [42]. While it is possible to define max pooling operations with arbitrary kernel size and strides, max pooling is generally applied to process non-overlapping areas of the input data. It then extracts the maximum values from these areas, as illustrated in Figure 3.3.

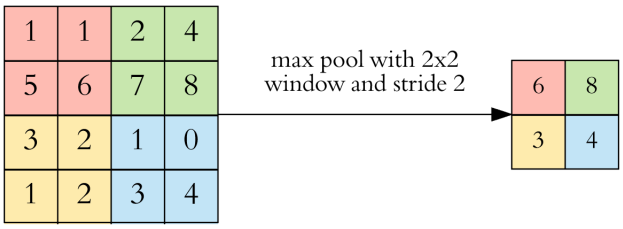


Figure 3.3: The basic max pooling operation. [14]

3.1.3 Densely Connected Layers and Dropout

Finally, after an arbitrary number of convolution-max pooling operations, one ore more densly connected layers are employed. These layers are used to produce the desired non-linear input-output mappings like in any conventional Neural Network.

Optionally, a *dropout* [46] layer can be fitted in after any hidden layer. Dropout can be employed to prevent the Neural Network from overfitting via randomly deactivating neurons during training with a probability of $p_{dropout}$, i.e. setting their value to 0. The basic idea behind dropout stems from a technique of preventing overfitting by combining many independently trained Neural Networks. By deactivating neurons during training, dropout effectively simulates training different Neural Networks, and by removing the dropout during deployment, a good approximation of the aforementioned approach is achieved [46, p. 1].

3.2 Example Use Cases

For their ability in pattern recognition on multi dimensional input data, Convolutional Neural Networks have been successfully employed in a wide variety of use cases. Perhaps the most notable is image processing [6, 15, 39], but successes have also been reported in speech recognition [2], sentiment analysis [16], and relation extraction slot filling [3].

4 Genetic Algorithms

This chapter will introduce the concept of Genetic Algorithms. The algorithms base upon the idea of emulating the very process that made complex life possible: Evolution managed to accomplish astonishing results where human systems often fail - in producing novel solutions to very high complexity problems. Genetic Algorithms therefore are a method of searching the solution space for an optimal solution.

The next sections will introduce the canonical Genetic Algorithm after giving a rough background in the evolutionary ideas that inspired it and related algorithmic techniques. The Genetic Algorithm's data representation and its component operations will be explained in more detail before finally giving a brief insight into some interesting use cases.

4.1 Evolution

"Living organisms are consummate problem solvers. They exhibit a versatility that puts the best computer programs to shame.", Holland begins his fundamental paper on Genetic Algorithms [27].

In fact, the mechanisms of evolution have created the very species *Homo Sapiens*, that has created those computer programs in turn. It is therefore easy to see for any researcher that there is potential for novel insight in the realm of problem solving in trying to understand and replicate these mechanisms. And indeed, one of the first breakthrough applications of such knowledge led to improvements in jet engine design, a very complex piece of machinery [27, p. 72]. Today, it is widely accepted that nature uses three main evolutionary mechanisms to accomplish design improvements: Selection, crossover and mutation.

The central role in all of this however falls to the genome, the DNA, a string of information that encodes the information necessary to build the actual life form[1]. This distinction between encoded genotype life form information and the resulting phenotype life form lends itself to the mechanisms named above: Crossover and mutation are easily imagined when looking at strings of information, and selection is a mechanism purely affecting the phenotype life form.

Selection, crossover and mutation will be looked at in more detail in the sections to follow.

4.2 Evolutionary Algorithms

Deriving from the evolutionary processes is the class of Evolutionary Algorithms. The term Evolutionary Algorithms is a common denominator bundling Genetic Algorithms, Evolution Strategies, Particle Swarm Optimization, and Evolutionary Programming. [28, 43]

Common to all of these approaches is the method of iteratively solving a problem by searching the solution space using a population of encoded candidate solutions, and then iteratively improving on them. Using the information gained from the population of candidate solutions to push the population towards the global maximum using combinations of candidate solutions and variations of individual solutions can be seen as emulating selection, crossover and mutation. [43, p. 1]

[1] Although newer research suggests that there are more sources of genetic information than the DNA alone, this simplification lends itself well for the purposes of Genetic Algorithms.

4.3 The Genetic Algorithm

The so called canonical Genetic Algorithm was first explored more deeply in the 1970s, mainly by John H. Holland [26], and recieved more widespread recognition when in 1993 the first issue of the international journal on Evolutionary Computation [13] appeared. It has since then found many real-world applications due to its rubust ability in locating global optima in arbitrary and multimodal search spaces, which are common in engineering challenges such as Neural Network design or training, structure design or flow optimisation problems [45].

While the details on the individual operations will be covered in the following sections, the basic Genetic Algorithm can be postulated as follows:

```
Population ← generate random population;
Fitnesses ← evaluate(P);
while not max(Fitnesses) > k do
    MatingPool ← SelectionOperator(Population);
    Population ← CrossoverOperator(MatingPool);
    for Individual ∈ Population do
        Individual ← MutationOperator(Individual);
    end
    Fitnesses ← evaluate(P);
end
```

Algorithm 1: The Genetic Algorithm

After creating a randomized initial population of genomes, their fitness values are evaluated by applying the fitness function on each of them. From there on, the actual Genetic Algorithm can start: While the solution is not yet found, a follow-up generation is computed by means of the selection operator, crossover operator and mutation operator. The selection operator builds a mating pool of genomes while considering their fitness values. Then the actual crossover happens, where a new population of genomes is built from the mating pool. Lastly, mutation is applied to each genome in the new population.

The end of the algorithm is reached when either a known optimal solution is reached $k = \max_x(F(x))$, or a sufficient solution is found. Alternatively if may end once convergence is reached within the population [23, p. 28].

The Genetic Algorithm has some distinct advantages over more traditional methods:

- Probably the biggest asset of the Genetic Algorithm is its capablity of escaping local minima. It achieves this by conducting a random, but directed, search which is not aimed at the nearest local minimum like gradient descend would be [45]. To do so, a *population* of candidate solutions (called *genomes* or *individuals*) are kept, and follow up generations of candidate solutions are iteratively computed through means of selection, crossover and mutation.

- Candidate solutions are evaluated by their so called *fitness* value, the result of an arbitrary function that measures their ability in solving the problem at hand. This function can provide absolute measures if the optimal solution is known, making it a search for the parameters producing this optimal solution. Or it can provide relative fitnesses by comparing the candidate solutions in a given population.

  The concept of the fitness function allows for a goal-oriented selection of candidate solutions to further process via selection and crossover while providing the flexibility of employing any kind of fitness function.

- The Genetic Algorithm in its most basic form does not rely on derivatives of target functions, like gradient descend does. Therefore it has a good capability of handling search spaces where derivatives are hard to compute or even impossible to obtain.
- The genetic operations mostly focus on single candidate solutions: The fitness function only ever needs to know one, and even crossover only ever works on a very limited number of candidate solutions, typically 2. Thus, the operations can be evaluated with parallel computing to drasticly reduce computation time for Genetic Algorithms.

4.3.1 Genome Encoding

Perhaps the most important component of a Genetic Algorithm is the so called *genome*. A genome is an encoded solution out of the solution space. Genomes are built out of *alleles*, parts of the genome encoding one aspect of the encoded solution. While real-valued allele strings are possible and actively being used [51, e.g.], the most basic representation would be a fixed-size binary genome, that is, a string of bits.

$$1011011010011001$$

Each bit or group of bits then encodes some characteristic of the encoded solution. A good example might be a geographic search for the lowest point in a confined area. Then, the above genome might encode x and y coordinates in the area, and the genome encodes one proposed solution by mapping directly onto one discrete point in the (solution) space.

$$\underbrace{10110110}_{x=182}\underbrace{10011001}_{y=153}$$

From there, the genome's fitness value can be computed by getting the altitude of Point $(182, 153)$.

There are two ways to expand on this simple idea on genome encoding: One could allow real valued variables instead of bits, and/or one could allow for variable length genomes. For the sake of simplicity, all further examples will be using binary genome encoding.

Variable Length Encoding

Variable length encoding has one major concern, especially with respect to eventual crossover operations: How to decide which part of the genome is encoding which aspect of the phenotype? There have been various proposals for this [30, 32, e.g.], three of which will be talked about now.

Using index strings Index strings, as shown in [30], can be used in conjunction with the actual genome to aid in decoding the genome information. The simplest version is a straightforward indexing of the genome positions:

Genome: 1 0 1 1 0 0 0 1 0 0
Index: 0 1 2 3 4 5 6 7 8 9

But these are unsuitable for variable length encoding. To allow for insertion or deletion of bits on arbitrary positions, the index string must be able to encode more fine grained information. A solution to this problem is the usage of real valued indexing values. This way, it is always possible to insert index values between existing values. This will prove very useful for variable length genome crossover.

Genome:	1	0	1	1	1	1	0	0	0	1	0	0
Index:	.0	.1	.2	.21	.22	.3	.4	.5	.6	.7	.8	.9

Crossover can then use the indexing information to avoid splitting up associated parts of the genome. In the above example, genome parts starting with the same decimal integer are seen as indivisable.

Using identifier genes Another approach that has been explored is the insertion of regulatory sequences into genomes [32]. Here, a predefined string of bits marks the starting (and optionally ending) point of a sequence of information on a longer genome. This is modeled very much after the human DNA, where special sequences are used to determine which parts of the DNA to read and decode.

Genomes containing identifier genes can be split into regions of continuous information via iteration over the genome string:

$$\overbrace{1\,1\,0\,0}^{identifier\ gene,\ (1^{st}\ attribute)} \quad \underbrace{0\,1\,1\,0\,0\,1\,1\,1}_{value,\ (1^{st}\ attr.)} \quad \overbrace{1\,1\,0\,1}^{id.\ gene,\ (2^{nd}\ attr.)} \quad \underbrace{0\,0\,1\,1}_{value,\ (2^{nd}\ attr.)}$$

For crossover the individual regions for each attribute can the be aligned to allow for controlled crossover operations without destroying the encoded structure.

Using complex data A third approach towards variable length genomes tries to circumvene the above issues by taking the easy route: Make the genome a complex object containing meta information alongside the actual genomic string. This is perhaps a less mathematically elegant solution towards genome design, but in the light of modern programming standards where object orientation is a fundamental aspect of many programming languages, this approach offers ease of implementation along with great flexibility.

Genome Class
attribute lengths : 4, 8, 4
genome string : $\underbrace{1011}_{4}\underbrace{00011011}_{8}\underbrace{0110}_{4}$

4.3.2 Crossover

With genome design established, the next thing to look at in Genetic Algorithms is the crossover operation, as it is closely tied to genome design. The goal of crossover is interpolating new locations in the solution space. To do so, crossover uses two or more proposed solutions of the current mating pool as anchors [23].

Single Point Crossover The most simple and most straightforward method for crossover is single point crossover [26, p. 65]. As both parent genomes might contain parts of a better or optimal solution, it is desired to combine the information of these two to try and create an even better performing offspring. This is done by picking a random crossover point in the parent genomes on which to split the parent genome information.

The next illustration demonstrates the principle and uses a and b on the second parent to provide a better visual cue:

Parent 1:	1 1 1 0 1 1 | 1 0 1 0 0 0 0
Parent 2:	$a\ a\ b\ a\ b\ a$ | $a\ b\ b\ a\ a\ a\ b$
Offspring 1:	1 1 1 0 1 1 $a\ b\ b\ a\ a\ a\ b$
Offspring 2:	$a\ a\ b\ a\ b\ a$ 1 0 1 0 0 0 0

Multiple Point Crossover Multiple Point Crossover takes the idea of single point crossover one step further by allowing an arbitrary number k of crossover points. Akin to single point crossover, upon reaching any crossover point, the origin of the subsequent genome section switches between the parents.

$k = 3$
Parent 1: 1 1 | 1 0 1 | 1 1 0 1 0 0 0 | 0
Parent 2: *a a* | *b a b* | *a a b b a a a* | *b*

Offspring 1: 1 1 b a b 1 1 0 1 0 0 0 b
Offspring 2: a a 1 0 1 a a b b a a a 0

Uniform Crossover Uniform crossover [17] is a random recombination method that takes multiple point crossover to its extreme and lets k equal the length of the genome, while copying the value of either parent with a probability of $p_{parent_1} = 0.5$, or with a probability that is proportional to the relative fitness values of the parents:

$$p_{parent_1} = \frac{f_{parent_1}}{f_{parent_1} + f_{parent_2}} \tag{4.1}$$

A second child can be created by inversing the decisions made for the first child.

$k = 3$
Parent 1: 1 1 1 0 1 1 1 0 1 0 0 0 0
Parent 2: *a a b a b a a b b a a a b*

Offspring 1: 1 a b 0 b 1 1 b 1 0 a 0 b
Offspring 1: a 1 1 a 1 a a 0 b a 0 a 0

Uniform Crossover has advantages and disadvantages: It is able to reach any recombination of the parent genome's bits with a single crossover operation, thus it is very versatile in searching the solution space. This however comes at the cost of destroying structures in the genome that might have already proven useful. Therefore random recombination is most useful when the individual values in the genome are largely independent from each other in regards to producing solutions.

Multi Parent Recombination Multi parent recombination, as discussed [17], takes an arbitrary number of parents into account when computing an offspring.

While uniform crossover can be extended to allow multi parent recombination, two different approaches prove to be more interesting: occurence based scanning [17, p. 2] and fitness based scanning [17, p. 3].

Occurence based scanning scans the parent strings for the majority class at each position in the genome. The thought process behind this approach is that index-value combinations that become frequent in genomes over the course of the Genetic Algorithm are probably useful for the end result. There are various ways to break ties, such as always picking the value of the first parent, or randomly choosing a parent to inherit from.

Parent 1: 1 1 1 0 1 1 1 0 1 0 0 0 0
Parent 2: 1 0 0 0 1 0 0 1 1 1 1 0 1
Parent 3: 1 1 0 0 0 1 0 1 0 1 0 0 1

Offspring: 1 1 0 0 1 1 0 1 1 1 0 0 1

Fitness based scanning picks the values according to the relative fitness values of the parent. So the probability of choosing parent i is

$$p(i) = \frac{F(parent_i)}{\sum_{k=1}^{n} F(parent_k)} \tag{4.2}$$

For order based genome representations, methods of adjacency based crossover exist and can be employed [17, p.3f.].

4.3.3 Selection

There are several ways to select which individuals to add to the mating pool, as long as the basic heuristic of favouring better performing individuals is being kept. [22] compare a number of basic approaches to crossover selection and evaluate their performance tradeoffs regarding convergence time and other metrics. Generally, "[t]he convergence rate of a GA is largely determined by the selection pressure, with higher selection pressures resulting in higher convergence rates." [37, p. 195]

The single most important denominator for selection decisions is the *fitness* function $F(x)$, which evaluates a given individual (as encoded by its genome) regarding the problem it should solve. $F(x)$ is therefore the function that returns the surface altitude of the solution space at the point the individual represents. Sometimes the fitness function is called *objective function*, or, inversedly, *cost function*. As the value of $F(x)$ is highest at the optimal solution, is is preferable to mate individuals with high fitness values to produce offsprings closer to the optimal solution.

Elitism One way to favor well performing individuals in the population is a concept called *elitism*. Here, the k best individuals are added to the next generation's population without altering them in any way. This way it is possible to prevent crossover and mutation from destorying the currently best candidate solution(s) [49].

Proportional Selection Proportional Selection, often nicknamed 'Roulette Selection' [12, 21], is a basic selection strategy that models the probability for any one individual x_k to be chosen for reproduction proportionally to its fitness value $F(x_k)$. The basic formula would then look like this, where n is the number of individuals in the population:

$$p_k = \frac{F(x_k)}{\sum_{j=1}^{n} F(x_j)} \tag{4.3}$$

From the pool of individuals are then selected two individuals to mate, where the probability of choosing any one individual is weighted by the above formula. It is unusual to allow replacement in the sampling process, but it may be advantageous in certain cases.

Ranking Selection The Idea of ranking based selection was introduced by [4]. Before doing a weighted selection, a rank based assignment function is used to determine the genome weights. The mathematical explanation to follow is a simplified version from the one presented in [22].

Let $x_1, ..., x_n$ be the sorted population of individuals, where $f_1, ..., f_n$ is their fitness value $f_i = F(x_i)$ and $f_i >= f_{i+1}$ for all $i \in [1, n]$.

A probability distribution α is introduced, where for each x_i, $\alpha(i)$ is the probability for x_i to be picked. Using the probability distribution, ranking selection can then be done just like proportional selection via simple probabilistic selection.

To be suitable as a probability distribution, α however needs to comply with a set of rules:

1. $\alpha(i) \geq 0$

2. $\alpha(i)$ is non-increasing

3. $\sum_{i=1}^{n} \alpha(x_i) = 1$

It is easy to see that these constrains force α to work as a probability distribution, where a total probability of 1 is distributed among the individuals in the population.

The main advantage of ranked selection is a discoupling of the proportions of individuals in the population from the selection process, slowing the takeover of very fit individuals during subsequent generations [22, p. 77].

Tournament Selection Tournament selection is a popular method of selecting individuals for mating, as it can effectively contain the selection process, preventing very fit individuals from taking over the entire population too easily. In fact, the selection pressure is proportional to the tournament size t [37, p. 194], and can thus be controlled via adjusting the parameters of the tournament selection [37]. Were it too high, the chance of a premature convergence on a local optimum rises. In the other extreme, if the selection pressure is too low, the algorithm takes unneccessarily long to complete.

For tournament selection a set of t individuals is randomly chosen from the population, and the 'tournament winning' individual gets added to a mating pool. This is repeated until the mating pool is filled.

Winning a tournament can be deterministic. Then the winner is decided by

$$\arg\max_x F(x)$$

Alternatively, the tournament can be probabilistic. Then a probability distribution is applied to the ordered list of individuals before choosing the winner. The most simple case for this is when $t = 2$. Then the individual with the higher fitness score wins with a probability p, where

$$0.5 < p \leq 1.0$$

Larger tournaments can be simulated by a bracketed knock-out system. In this case, successive rounds of binary ($t = 2$) tournaments are held, whereby the winners advance into the next round.

Alternatively, akin to ranking selection, a probability distribution α is applied to all participating individuals after they have been sorted for their fitness values. Let $x_1 ... x_t$ be the descendingly ordered participants in the tournament. Then the probability of winning the tournament for the i^{th} individual is

$$p_i = \alpha(i)$$

To achieve a probability distribution, α needs to meet the requirements postulated in the section on ranking selection:

1. $\alpha(i) >= 0$

2. $\alpha(i)$ is non-increasing

3. $\sum_{i=1}^{t} \alpha(x_i) = 1$

4.3.4 Mutation

The last and final operation of the Genetic Algorithm is mutation. Mutation is an operation that has a low probability of modifying parts of a given genome. "[It] alone does not generally advance the search for a solution, but it does provide insurance against the development of a uniform population incapable of further evolution" [26, p. 68]. Viewed in terms of the solution space surface, mutation is a random local area search operation that helps escaping local maxima [30].

Mutation achieves this by iteratively walking the genome, and changing each value with a very small probability p_{mut}. Two interpretations of the mutation operator exist on binary genome strings: Setting the value to random or switching the bit value. As long as the user does mind the implications in the probability of a change in value happening, both approaches are equivalent.

Before Mutation: 1 1 1 0 1 1 1 0 1 0 0 0 0
After Mutation: 1 1 $\underline{0}$ 0 1 1 1 0 1 0 $\underline{1}$ 0 0

Real valued genome strings, or genome strings comprised of complex data need a more sophisticated mutation operator, where newly generated values can be taken from uniform or stochastic distributions over the space of possible values.

Mutation in variable length genomes also has a chance to change the length of the genome via insertion or deletion. For variable length genomes it might however be desired to not increase the expected amount of mutations with the size of the genome. Then, a global mutation probability $p_{globalmut}$ can be used to determine whether a mutation should happen, and in case a mutation occurs a random index is chosen on which to apply the mutation.

4.4 Example Use Cases

4.4.1 Feature Selection and Weighting

Feature selection for classification systems is a task where it is easy to see how Genetic Algorithms may perform well: The avaiable features can be encoded with a binary genome, where each 1 stands for 'use the feature' and each 0 stands for 'ignore the feature'. The Genetic algorithm has been successfully applied to search this space for the optimal combination of features to use [52, e.g.]. As features are are generally rather independent from each other in aiding the classification process, the solution space is likely to have lots of local optima - a landscape, where Genetic Algorithms perform well due to their inherent design.

Feature selection can be seen as a special case of feature weighting: The features are to be weighted with binary weights of 1 and 0 [52, p. 2]. As such, the genetic approach to feature selection can be applied to feature weighting as well.

4.4.2 Evolvable Hardware

Genetic Algorithms have been employed on the task of evolving hardware layouts [24, e.g.], both theoretical and via real time adaptable hardware (also called Evolvable Hardware). Evolvable Hardware is built from programmable logic devices, whose logic gates can be reconfigured via a software bit string. This string can be learned using Genetic Algorithms, where the device is tested to produce a fitness value for each configuration that has been encoded in a Genetic Algorithm Genome [25]. Research in the area of self-reprogramming logic devices could prove very useful for the development of autonomous robotics.

4.4.3 Jazz Music

The range of use cases for Genetic Algorithms doesn't stop in the confines of pure technology however. Computer generated music has lifted the approach into the domain of musical art. One example is the JazzJam system by Prof. Al Biles from the Rochester Institute of Technology. He devised a Genetic Algorithm to generate jazz music bits from recorded sound progressions, effectively creating a virtual improvising jazz player[2] [7]. The system is able to listen to music played and generate fitting musical responses or accompaniments in real time, creating a musical conversation between the algorithm and a human musician, just like in a real Jam session.

[2]Visit `https://www.youtube.com/watch?v=rFBhwQUZGxg/` for a demonstration

5 System Description

Within the framework of this paper a set of experiments regarding Genetic Algorithms for the optimization of Convolutional Neural Networks for the ATIS Slot Filling Task has been devised. One of them has been carried out and is evaluated in Chapter 6.3. In this chapter, an overview over the envisioned system is given. A practically oriented consideration of the used data and algorithmic complexity is carried out, as well as a theoretical consideration of the experimental algorithms.

5.1 System Data

The ATIS data set used in the experiments consists of 5872 sentences containing $65,788$ word tokens, making it a comparatively small dataset to work with. The tokens are annotated with 126 classes and the outisde class. Classes are given in IOB notation and may represent subclasses. Reference Table 2.2 for an example sentence from the ATIS data set. The data set is split into 4978 sentences of training data and 893 sentences of test data.

There are two ways a Convolutional Neural Network labeling the ATIS data could be structured: Either as a sequence-to-sequence model where each sentence gets mapped to a corresponding sequence of labels, or as a sequence classification task, where word n-grams have to be assigned the correct label. Both however impose practical upper bounds to the input length of the convolutional network, as larger input lengths would require excessive use of padding.

5.2 System Goal

The goal of the experiments is to explore the usefulness of the Genetic Algorithm in regards to optimizing the structural aspects of a Convolutional Neural Network. There are several aspects open for optimization:

- **Neural connectivity** It is possible to learn the connectivity pattern between neurons in the network, as well as the count of neurons itself. While research has been done for relatively small networks [29, 36], and other aproaches have tried to devise genome encoding methods for bigger networks [32], the convolutional architecture is relatively rigid in its structure and does not allow for much freedom in the patterns to be learned. This is why this approach has been discarded in favor of other learnable aspects.

- **Network weights** While it is very possible to learn network weights via Genetic Algorithms [38], this approach is not it the scope of this bachelors thesis.

- **External hyper-parameters** Another area of learnable variables are the external hyper-parameters of a Neural Networks. In this context, 'external' is used to denominate parameters which are not describing the network itself, but rather the training process. These parameters include: Number of training epochs, batch size for training, learning rate, learning weights, optimizer function, loss function and word embedding size. All of the aforementioned are values that could be learned, but as the focus of this paper lies on structural optimization, the next area is chosen.

- **Constituent hyper-parameters** The remaining area of learnable parameters are the constituent parameters of the Convolutional Neural Network itself. 'Constituent' means those parameters that define the structure, shape, and count of the layers in the Neural Network. As the structure of one such network has many ramifications on its learning behaviour, this approach was chosen for the experimental approach in this paper.

5.3 Encoding

A genome has been devised to encode the constituent hyper-parameters of a Convolutional Neural Network. For the encoding method a complex data genome (henceforth called *CNNGenome*) has been chosen for two reasons:

- Convolutional Neural Networks have special constraints to their structure. Convolution and max pooling downsample the input data and are thus limited in the possible configurations one can devise. Also, convolutinal Neural Networks need different constutuent information, depending on the type of layer being defined (convolutional or dense).

- Modern programming languages like Python 3 offer extensive support for object oriented programming and thus do not pose a barrier in trying to implement complex genome data classes or operations on them. In the conrary: Object oriented programming techniques lend themselves well in recreating processes that resemble real world mechanisms like crossover and mutation.

Structure The devised genome has a layered structure, as shown in Figure 5.1: Alleles of kernel size k, strides s and filter count f are repeated a times, followed by alleles of hidden unit size u and dropout rate d. Convolutional layer count a and dense layer count b are stored seperately to allow for a structured decoding of the string on real valued information pieces.

CNN Genome Class
Convolutinal Layer Count: a
Dense Layer Count: b
Genome string: $(k, s, f) \times a + (u, d) \times b$

(5.1)

CNN Genome Example
Convolutinal Layer Count: 2
Dense Layer Count: 3
Genome string:
$(5, 2, 32, 2, 1, 64, 500, 0.30, 150, 0.45, 750, 0.50)$

To disallow impossible or entirely useless structures, some constraints are imposed on the different parts of the genome, and may also be dependent on the location in the genome string: Convolutional layer kernel size and strides depend on the input dimensions to each layer, which in turn depends on the kernel sizes and strides of earlier layers. These constraints reduce the search space by disallowing solutions that can be deemed

undesirable a priori.

1. $a \in [1, 2]$[1]
2. $b \in [1, 3]$
3. $2 \leq k \leq |\text{layer input}|$
4. $f \in \{2^x\}$ for $x \in [2, 10]$
5. $u \in \mathbb{N} \wedge 1 \leq u \leq 1000$
6. $d \in \mathbb{R} \wedge 0.0 \leq d \leq 0.75$

5.4 The Algorithm

The Genetic Algorithm that builds upon the genome introduced in the last section is purposefully left very close to the canonical standard Genetic Algorithm as envisioned by Holland [26]. Algorithm 2 presents a schematic view of the proposed algorithm: After generating a random population of CNNGenome instances, a global variable genCount controls the amount of generations to iteratively compute. To do so, each generation's population gets evaluated to get the fitness value of each individual. Then, crossover and mutation is used to compute the next generation. Elitism is employed to let the fittest genome survive the generation transition. The details on the selection, mutation, and crossover operations will be given in the following parts.

```
Data: CNNGenome // Genome Class
      genCount // Amount of generations to compute
      popSize // Number of Genomes in a population
      mode // Crossover mode: Either roulette or tournament mode
P ← generateRandomPopulation(CNNGenome, popSize);
for i ← i to genCount do
    foreach g ∈ P do
        g.f ← evaluateFitness(g);
    end
    best ← arg max_g(g.f);
    P_new ← {best} ∪ crossover(P, mode);
    P ← mutate(P_new);
end
```

Algorithm 2: The employed Genetic Algorithm

5.4.1 Genetic Operations

While fitness and selection are invariant to genome encoding, selection and mutation need to be defined with respect to the genome encoding method.

Mutation 'Bit-by-bit' mutation is used on the genome string of length $3 \cdot A + 2 \cdot B$, where A is the number of alleles describing convolutional layers and B is the number of alleles describing densely connected layers with dropout. A pointer walks the genome string and inserts a new, random, valid value on each position with a independent probability of p_{mut}.

Additionally, with a probability of p_{mut} a size changing operation is attempted, wherein the probability of an attempt to change either the number of convolutional layers A or the number of dense layers B is equal. Shrinking operations on minimum sized layer types are discarded, as well as growing operations on maximum sized layer types.

[1] This constraint stems from the input size of the ATIS data used: Even the least downsampling leaves only 2 nodes after two layers of convolution and max pooling

When changing the size of the genome, again, only structurally possible CNN configurations may be generated.

Crossover The approach of only allowing sound genome configurations is kept for the crossover operation, when one offspring g_{new} is produced from two parent genomes g_1 and g_2. First, the convolutional genome information is handled via a brute force method:

1. Pick the resulting convolutional layer count by letting A_{new} equal the value of a random parent.

2. Jointly walk the parent's genomes and pick the value of either parent with equal probability[2].

3. Test the resulting genome for structural impossibility.

4. If a impossible structure has been built, discard the genome and go back to step (2).

This approach is guaranteed to terminate, as the edge cases of $g_{new} = g_1$ and $g_{new} = g2$ only from one parent are always structurally sound.

The dense layer information for the offspring genome is computed in a very similar way: Pick the densly connected layer count of either parent with equal probability. Then walk the genomes of the parents and pick either value with a probability of 0.5^2. Here, no feasibility checks are needed, as the densly connected layers are independent of the surrounding layers and can only take on values generated with the original constraints in mind.

Selection Two algorithms for selection were chosen to be usable for the experiments (see 4.3.3).

1. Probabilistic 'roulette' selection 2. Binary tournament selection ($t = 2$)

Both employ elitism and let the fittest individual join the next generation without undergoing mutation. It can however still father offspring genomes in the normal crossover process.

While the probabilistic roulette selection is chosen as the standard selection mechanism, tournament selection has been added to the pool of possible selection modes because of its lower selection pressure when compared to the former.

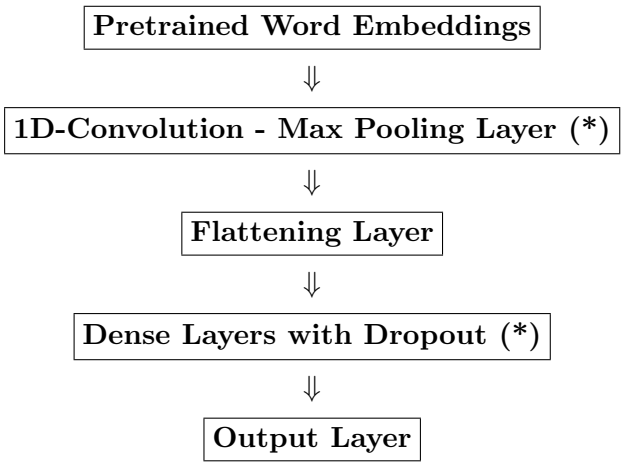
Evaluation Genome evaluation is done by the fitness function employed in the system. To assess the properties of the encoded Neural Network, it has do be decoded and built before it can be trained and evaluated on the data at hand. The evaluation operation of the system does exactly that: Build the Convolutional Neural Network, train it and measure the metrics required to determine its fitness. As the measures themselves are implementation details and may be subject to variation in different experimental settings, those will be discussed in the next chapter.

5.5 Network Architecture

With the CNNGenome defined, it is possible to create an automated decoding algorithm translating genome instances to a Convolutional Neural Network. It's structure is defined

[2]If the parent genomes are of unequal length, non-aligning parts are always chosen from the longer parent.

as follows, where layers with an asterisk (*) are repeated as often as defined in the genome:



In the first layer, the n-gram input is translated into its representation of pretrained wod embeddings. Then, an arbitrary number of convolution and max pooling operations searches for partterns in the input data. Hidden dense layers are employed to learn the nonlinear mappings from the feature representations to the output classes, which are generated by a 127 node wide output layer representing the 127 label classes present in the ATIS data set.

5.6 Complexity Considerations

With the general system architecture laid out, but before the conduction of practical experiments with Genetic Algorithms, a short consideration of complexity is needed. Because the procedure at hand combines training Neural Networks and the Genetic Algorithm, computation times can quickly grow unfeasible.

5.6.1 Network Training

Considering constant network input size, network training time depends largely on four factors, and training times of many hours or even days are not uncommon for a large Convolutional Neural Networks operating on big data sets.

1. Amount of training data. Training time grows linearly with the size of training samples.

2. Amount of trainable parameters. Training time increases proportionally to the trainable parameters.

3. Number of training epochs. Training time grows proportional to the number of epochs it is trained.

4. Type of hardware used. Dedicated GPU(s) or TPU(s) can speed up the training process considerably by using dedicated parallel processing units for matrix operations.

For the experiment conducted within the limits of this bachelors thesis, the ATIS dataset proved to be a very small data set, helping in making Neural Network training times feasible on standard hardware. Table 5.1 compares training times[3] of different system configurations on the hardware used during the development of this bachelors thesis[4].

[3] Note: The training time encompasses accompanying computations such as callback functions during the training process

[4] Hardware used: Intel CPU i5-6600, nvidia GeForce GTX 970

Learnable parameter count	Batch size	Training epochs	GPU	Time
min size	16	20	no	5:50m
min size	32	40	no	6:00m
min size	16	20	yes	28s
avg size	16	20	yes	1:10m
max size	16	20	no	42m
max size	16	20	yes	3:50m
max size	16	40	yes	7:40m

Table 5.1: CNN training times

5.6.2 Genetic Algorithm

The Genetic Algorithm itself does not have a big overhead in computation time. The combinatoric operations operate on relatively small sets of genomes and even the brute force method employed for crossover terminates in fractions of a second in a worst case scenario of entirely incompatible max size genomes.

While the Genetic Algorithm itself does not add much computation time, it does govern the amount of network training procedures to be had, and thus the overall algorithmic complexity. Let P be the population size, G the generation count, then $P * G$ the number of times the fitness function is imploring Neural Network training. Let furthermore t be the single pass network training time and e be the epoch count for Neural Network training.

Table 5.2 compares different configurations of network training times and Genetic Algorithm parameters, and their effect on completion time of the algorithm, by giving a lower bound. The lower bound is reached assuming only network training takes up computation time, and is located in the same order of magnitude as the real value, because network training time ideed takes up the vast majority of computation time. The lower bound can be expressed as

$$t_{GA} \geq P \cdot G \cdot t_{train} \cdot e \tag{5.2}$$

P	G	$P \cdot G$	t_{train}	e	Completition time lower bound t_{GA}
10	10	100	30s	20	16.6h
10	10	100	2m	40	5.5d
20	20	400	5m	20	27.7d
50	20	1000	30s	20	6.9d
50	20	1000	2m	40	55.5d
50	20	1000	5m	20	69.4d

Table 5.2: Algorithm completion times with serial network training

To deal with this exploding amount of computation time, the parallel nature of the Genetic Algorithm needs to be employed: The computation of the fitness values can be done fully concurrent, removing P from the equation in an ideal environment:

$$t_{GA} = G \cdot t_{train} \cdot e \tag{5.3}$$

Table 5.3 shows the same setups seen in Table 5.2, but with perfect parallel network training employed.

P	G	$P \cdot G$	t_{train}	e	Completition time lower bound t_{GA}
10	10	100	30s	20	1.66h
10	10	100	2m	40	0.55d
20	20	400	5m	20	1.39d
50	20	1000	30s	20	3.3h
50	20	1000	2m	40	1.1d
50	20	1000	5m	20	1.4d

Table 5.3: Algorithm completion times with fully parallel network training

6 Experiments and Results

Due to the discussed time complexity behaviour of the experimental system, the development of the system couln't be completed in time to conduct an extensive series of experiments. Only one complete test run could be done in the timeframe of this bachelors thesis. This run will be evaluated in the current chapter, while proposed further evaluations and experiments will be outlined in chapter 7.2 Future Work. In this chapter, first of all the software used to implement the system is described. Then, the experimental setup is explained and its results discussed.

6.1 Used Software

6.1.1 TensorFlow

TensorFlow[1] is a open source software library for high performance numerical computation, like Neural Networks. It uses data flow graphs, where graph nodes represent mathematical operations, while the edges represent tensors flowing between them. It allows for distributed computation on CPUs, GPUs and TPUs [1]. TensorFlow was originally developed by researchers and engineers of the Google Brain team within Google's AI organization, but later made open source[2]. At Google, it was built upon the DistBelief framework [11], and subsequently replaced it [1, p. 266]. TensorFlow is mostly used with a Python API[3], other avaiable APIs include C++, Swift and Java.

6.1.2 Keras

Keras[4] is a API for high-level Neural Network implementation. It is written in Python and can be used as a layer above either TensorFlow, CNTK[5], Theano[6], or MXNet[7]. Keras wraps lower level functionality into convenient predefined classes, allowing for fast experimentation and prototyping, being able to be run on the CPU as well as on the GPU. The user can however extend Keras classes to implement any desired type of functionality, as well as mix and match Keras code with code from the backend used.

6.1.3 CUDA

CUDA[8] is a "parallel computing platform and programming model developed by NVIDIA for general computing on graphical processing units" [10]. CUDA allows programs to harness the parallel computing power of GPUs, and is used by both TensorFlow as well as Keras to provide the option of running their models on the GPU.

[1] https://www.tensorflow.org/
[2] https://github.com/tensorflow/tensorflow/
[3] https://www.tensorflow.org/api_docs/
[4] https://keras.io/
[5] https://github.com/Microsoft/cntk/
[6] https://github.com/Theano/Theano/
[7] https://mxnet.incubator.apache.org/
[8] https://developer.nvidia.com/cuda-downloads/

6.1.4 GloVe

GloVe is a learning algorithm for obtaining high quality vector representations for words. Pretrained embeddings in various embedding sizes are avaiable from the GloVe website[9]. For this experiment, 200-dimensional embeddings from the GloVe 6B set have been used, which were trained on 6 billion tokens from wikipedia and newswire data.

6.2 Experiment Setup

The experiment that could be conducted had to be subject to a number of decisions for the hyper-parameters of the Genetic Algorithm and the Convolutional Neural Network, being the target of the optimization attempt. In these decisions some tradeoffs towards algorithm completion time had to be made: With an average Neural Network training time of 1:20 minutes per epoch, reasonable settings like 30 genomes, 30 generations, 40 training epochs became unfeasible with a total expected algorithm completion time of over 30 days.

6.2.1 Parameters

Selection Method Standard roulette selection has been chosen for this experimental setup. Two reasons led to this decision: Firstly, it is of great interest to establish a baseline performance with conventional methods first, before expanding on the gathered results. Second, this decision removes one variable from the system by eliminating the need of setting a tournament size and deciding the tournament resolvement method.

Population Size and Generation Count With population size P and generation count G being the biggest factor in time complexity of the experimental system, reasonable values had to be chosen. Empirical probing led to the belief that values in the range of $P = 30$ and $G = 30$ were at least needed to generate expressive results. Time constraints though led to the decision to run the algorithm with values of $P = 20$ and $G = 10$, effectively quartering the completion time of the algorithm.

Neural Network Training A standard value of 16 has been chosen for the Neural Network training data batch size. *RMSprop* was chosen as the optimizer function, which has proven to be a very reliable optimization method [41]. As for training epochs, again, a sacrifice had to be made to trim the algorithm down to size: Each Neural Network is being trained for 20 epochs, sacrificing peak network performance for computation time. The repercussions of this decision will be discussed in the evaluation.

Additionally to the above parameters, a weighting function has been employed to combat the imballanced frequency of the training labels. Let n be the amount of different classes. With most of the labels being O-tags, a function $w(l)$ has been devised which maps class labels l of the ATIS data set to a inverse measure of their frequency f_l, boosting backpropagation runs of rare labels and dampening those of frequent labels. The weighting function for the i^{th} label is shown in equation 6.1, with k denoting the total number of class occurences $\sum f_l$.

$$w(l_i) = \frac{k \cdot n}{f_{li} \cdot \sum_{m=1}^{n} (\frac{k}{f_{lm}})} \tag{6.1}$$

[9] `https://nlp.stanford.edu/projects/glove/`

6.2.2 Fitness Function

Yet perhaps the most interesting aspect of building a Genetic Algorithm is the decision on the fitness function. With the goal of optimizing constituent hyper-parameters for a Convolutional Neural Network, the measure to evaluate the performance of a resulting network is not clear cut. Neural networks have more than one metric of performance: Naturally, their raw prediction performance is of importance, but also their tendency to overfit. In fact, combatting overfitting is a constant struggle in Neural Network design. Lastly, the network size is a discriminating factor. A small network is preferable, as it is faster to train, but it needs to be sufficiently big to be able to learn complex nonlinear mappings. A combined measure for multi-objective evaluation has been devised, averaging three properties of Neural Networks.

Prediction Performance The prevalent way to measure the raw performance of a Neural Network is to compute metrics on their performance when predicting seen and unseen data. The two standard metrics to use are *Accuracy* and *F-Score*. Because of the very imbalanced ATIS data set, where more than 70% of all words are labeled with O-tags and other labels are very sparse, evaluation attempts with Accuracy quickly run into the Accuracy paradox [48], where the score is heavily inflated by the one prevalent class.

The decision was made to omit the *O*-tag completely from the performance measurements. A moderated version of the classic F_1-Score is used on all the other classes. The F_1-Score value is mapped to a region of $[0, 100]$, and lower scores are additionally punished by taking the square of the F_1-Score (Equation 6.2). This leads to a bigger fitness gap between very highly performing networks and mediocre ones.

$$P(x) = 100 \cdot x_{F1}^2 \tag{6.2}$$

$$F_1 = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}} \tag{6.3}$$

$$\text{Precision} = \frac{\text{true positives}}{\text{all positives}} \tag{6.4}$$

$$\text{Recall} = \frac{\text{true positives}}{\text{true positives} + \text{false negatives}} \tag{6.5}$$

Overfitting Neural networks can only learn from the train data that is provided, but are supposed to achieve good results on unseen data. Specialising too much on the training data may result in worse ability to generalize to unseen data, making networks with less tendency to overfit preferable. To measure overfitting, again a function has been devised that maps the network trait into the numerical space of $[0, 100]$. The difference Δ in F_1-Score between the training data and the evaluation data is a good indicator for overfitting. Equation 6.6 has been devised to map overfitting deltas between 0 and 30 F_1 points into $[0, 100]$, as probing runs have shown that overfitting deltas generally stay in these magnitudes.

$$\begin{aligned} \Delta(x) &= x_{\text{F1-eval}} - x_{\text{F1-train}} \\ O(x) &= \begin{cases} \max(1, 100 \cdot (5 \cdot \delta(x) + 1)) & \text{if } \Delta(x) \leq 0 \\ 100 & \text{if } \Delta(x) > 0 \end{cases} \end{aligned} \tag{6.6}$$

Network Size This criterion is straightforward: If two networks exhibit comparable properties in performance and their tendency to overfit, the smaller one is to be prefered, as it can be trained more quickly. To measure network size and normalize the value into a numeric area of $[0, 100]$, the minimum possible network size and maximum possible network size is derived from the genome encoding method. Network size is measured directly in the amount of trainable parameters in the network, as it directly corresponds to node count and training time.

$$S(x) = 100 - 100 \cdot \frac{s_x - s_{min}}{s_{max} - s_{min}} \tag{6.7}$$

Multi-Objective Function To allow for easy weighting of these three performance measures, the multi-objective fitness function of the system has been implemented as the arithmetic mean of the individual performance measures. The weighting factors α, β, and γ can be set independently, to allow for fine grained adjustments in the desired focus of the genetic optimization process.

$$F(x) = \frac{\alpha \cdot P(x) + \beta \cdot S(x) + \gamma \cdot O(x)}{3} \tag{6.8}$$

However, other methods of calculating the mean might be even more useful for multi-objective performance measures: The geometric mean, for example, has properties punishing uneven distributions of single performance values in the combined measure, as visualized in Figure 6.1 for combinations of two independent values. In reality, the decision on the averaging function should be made with respect to the desired system output: Should the individuals be forced into evolving into versatile problem solvers, ultimately achieving comparable results in all performance measures, or should more extreme configurations be rewarded, where individuals reach very high scores in few of the metrics, but can fall off in other areas in turn?

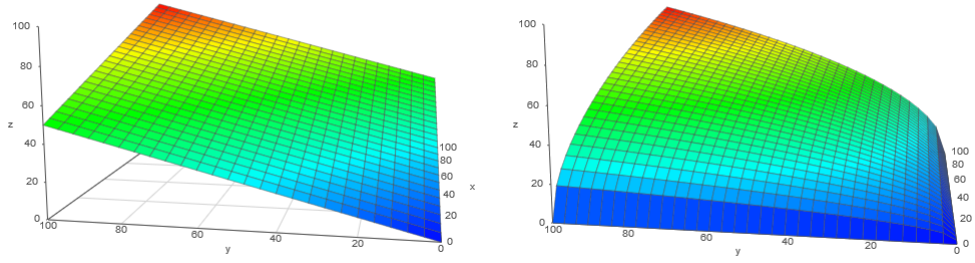


Figure 6.1: Arithmetic mean (left) and geometric mean (right) of two variables x and y. Created with [5]

6.2.3 Goal

The goal of the conducted experiment was to see whether a reasonable and ballanced Convolutional Neural Network architecture could be generated from random individuals. The weighting factors (performance, size, overfitting) have been set to

$$\alpha = 3$$
$$\beta = 2$$
$$\gamma = 1$$

in an attempt to favor well performing networks of small size. The results of the experiment will be presented in the next section.

6.3 Evaluation

6.3.1 Genetic Algorithm Performance

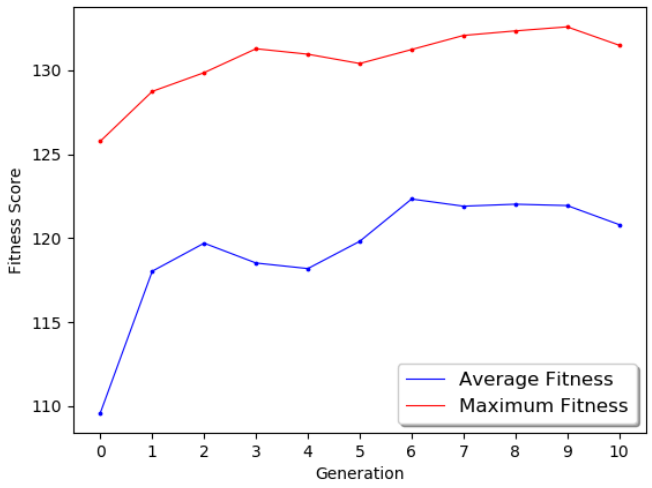


Figure 6.2: Maximum and average fitness values

The Genetic Algorithm managed to consistently improve the average fitness values of its population over the course of 10 generations. Figure 6.2 draws the development of the average population fitness, as well as the development of the fitness value of the fittest individual in each generation. Note that even though elitism is employed, the maximum fitness may lower due to the non-deterministic nature of the Convolutional Neural Network training process. It has been observed that the randomized batch order of the training process leads to variations in the resulting performance and overfitting values.

6.3.2 Network Size

Interesting observations could be made on the development of the resulting network sizes (Figure 6.3). Note that the size measure in the Figure represents node and filter counts, and is thus compressed towards the top. This was chosen to increase the readability of the plot. Already in the first couple of generation changes a sharp decrease from the initial average can be observed before the line shows largely constant behaviour in its oscillations. It seems that smaller Neural Networks are sufficient for the ATIS task, which is not surprising given the small size of the task itself.

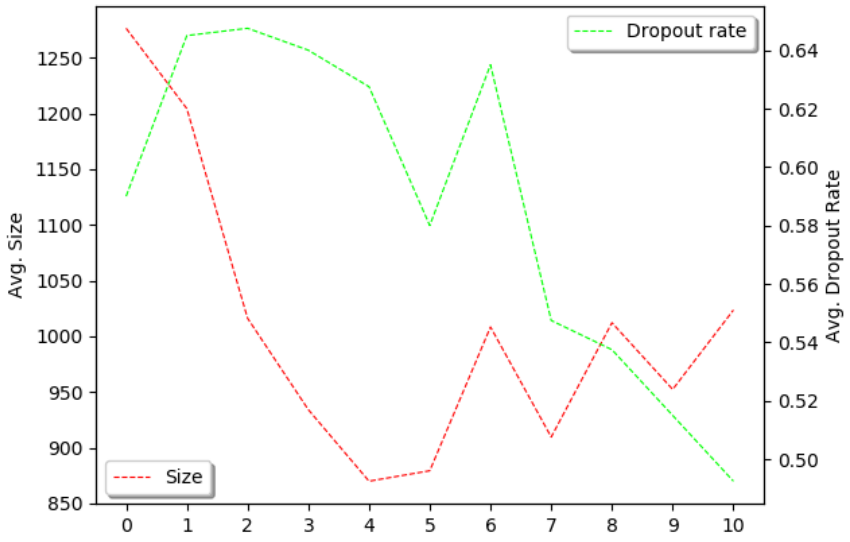


Figure 6.3: Size and dropout rate development

6.3.3 Dropout

Another very interesting observation can be made upon the development of the prevalent dropout rates: Their average converges at around 0.5, which is in line with the empirical observations on good dropout rates by [46]. In fact, closer examination of the encoded network architectures shows that almost all dense layers in generation 10 contain dropout rates either in the range of $p = 0.35$ to $p = 0.55$ or lower than $p = 0.15$[10].

6.3.4 Network Architecture

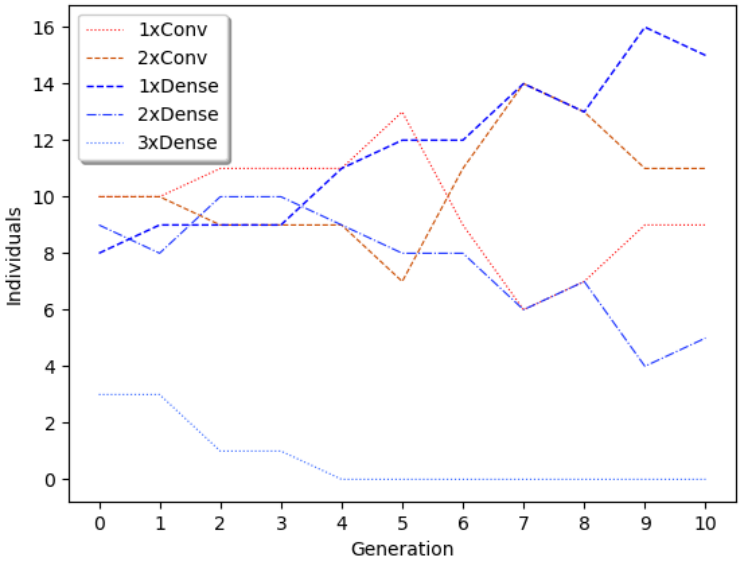


Figure 6.4: Network architecture development

Figure 6.4 plots the prevalent architecture types against each other over the course of all 10 generations. While genomes with one or two convolutional layers seem to be equally competitive, the amount of dense layers encoded in the genomes of later generations undergoes a significant shift towards simpler architectures with less hidden layers. Genomes with three hidden dense layers go extinct by the fourth generation, and the amount of genomes with two hidden dense layers is shrinking with almost every generational transition.

There are two possible explanations for this behaviour: It is very possible that one hidden dense layer with a sufficiently high count of nodes is satisfactory for the ATIS task at hand. On the other hand, this behaviour could be a repercussion of the limited amount of epochs of training each Neural Network architecture gets allocated. Deeper networks take longer to train, and thus the 20 epochs may be insufficient to fully develop the mappings in deeper network architectures.

When looking at the plot of convolutional layer configurations, another convergent process can be seen. Figure 6.5 draws the frequency of all encountered architectures for the first convolutional layer of the network over all 10 generations. Annotated are the accumulated appearance counts for the architectures over all generations. From the graph alone is is apparent that the configuration with kernel size 4, strides value 1 and filter count 32 is the most competitive and begins to sweep the population. In fact, 32 filters seem to be largely the most competitive configuration in this environment. This again poses the

[10] Note that p is defined as a retention probability in [46], while in this paper p is inversely defined as dropout probability.

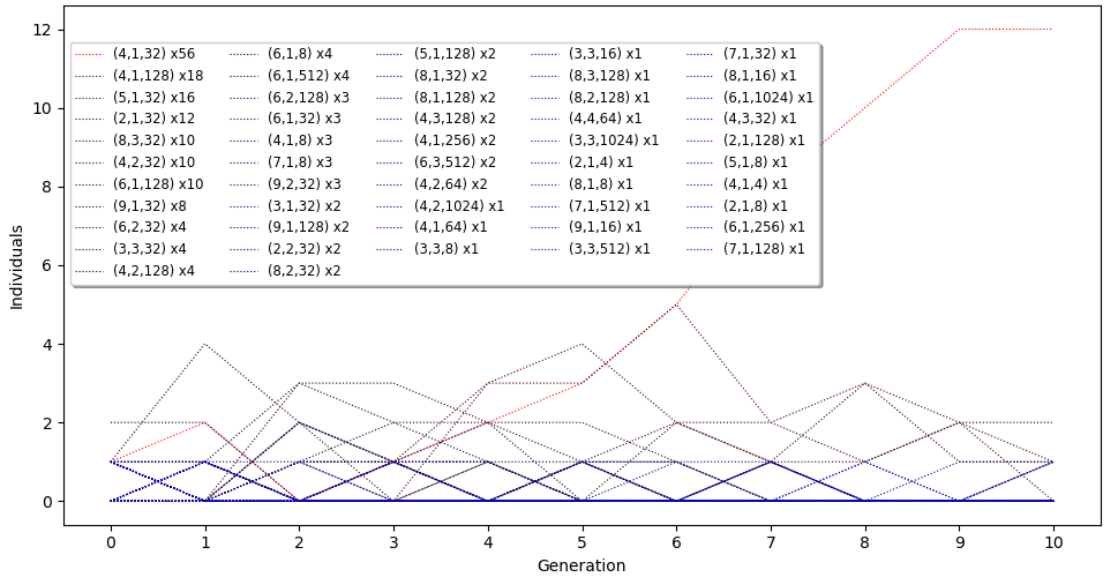


Figure 6.5: Development of the first convolutional layer architecture with aggregated counts. Encoding is (kernel size, strides, filters).

question, whether the results are influenced by the low amount of training epochs for the Neural Networks. It is entirely possible that longer training times would favor convolutional layers with a bigger filter size. However, in that case the results would only show the Genetic Algorithm's ability to adapt to the apparent circumstances.

Ultimately the network architecture that achieved the highest fitness score and started to take over the population was a comparatively small network comprised of one convolutional layer and one hidden dense layer (Figure 6.6).

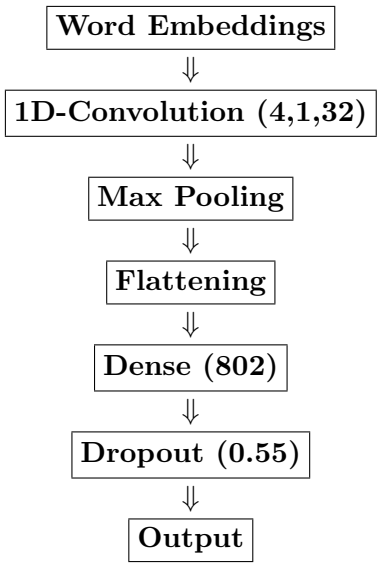


Figure 6.6: Best performing network architecture after 10 generations

7 Conclusion

7.1 Experiment Summary

The canonical Genetic Algorithm has been adapted to the task of learning Convolutional Neural Network structures with the goal of optimizing the performance aspects of the network. While extensive experiments couldn't be carried out, first results confirm the assumptions on the usefulness of the Genetic Algorithm and encourage further research in the area. The gathered data exhibits the expected evolutionary traits: Average genome performance rose with the number of generations, as well as useful genome configurations started to take over the population. Dropout rates converged to measures which were asserted to be most useful in previous works, thus confirming the theoretical considerations on the genetic operators of selection and crossover. But, as mutation did not play a big role in these early and still noisy generations, its influence on the overall performance of the system could not be evaluated.

7.2 Future Work

The big amount of parameters to tweak for the proposed system leaves space for many experiments to further the understanding of the behaviour and performance of the Genetic Algorithm in different settings.

Notably, three experiments are of great interest:

1. **Optimizing the Convolutional Neural Network for performance extremes** While the conducted experiment aimed at a ballanced consideration of raw performance, overfitting tendency and size, the effect of a reduction of the fitness function to any one of them should have interesting repercussions on the evolved network architectures. It is very possible that there are extreme architecture variants that perform well in certain areas of performance. The gained knowledge could help understand network architectures better and further our understanding in the limits of useful network structures.

2. **Evaluating the effects of population size and generation count** While the avaiable hardware limited the scope of the experiments in this paper, a thourough exploration of the effects of different configurations of the Genetic Algorithm in regards to population size, generation count, and even selection method and elitism could broaden our understanding of the Genetic Algorithm and bring the system closer to its real world model: Evolution works with a vast pool of individuals over thousands of generations.

3. **Finding architectures that learn quickly** Another extreme that could be worth exploring is purposefully limiting the training time of the encoded networks. This could lead to networks with a capability of quick adaptation to win out in the evolutionary process. The field of quickly trainable networks is interesting because real time applications like robotics can benefit greatly by employing retrainable networks, giving them the ability to adapt to changing circumstances. However, it is likely that interesting results in this area would require an expansion of the original scope of architecture encoding: Learning individual connections or nonlinear, branching layer configurations.

8 Materials

The materials accompanying this bachelors thesis are made avaiable under `https://github.com/LanyK/BachelorsThesisYKaiserLMU2018/`. These include:

- The original LaTeX version of this paper
- The PDF version of this paper
- The python scripts of the practical experiment
- PDF captures of the referenced web ressources

Bibliography

[1] Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., ... & Kudlur, M. (2016, November). TensorFlow: A System for Large-Scale Machine Learning. In *OSDI* (Vol. 16, pp. 265-283.)

[2] Abel-Hamid, O., Mohamed, A. R., Jiang, H., Deng, L., Penn, G., & Yu, D. (2014). Convolutional neural networks for speech recognition. *IEEE/ACM Transactions on audio, speech, and language processing, 22*(10), 1533-1545.

[3] Adel, H., Roth, B., & Schütze, H. (2016). Comparing convolutional neural networks to traditional models for slot filling. *arXiv preprint arXiv:1603.05157*.

[4] Baker, J. E. (1985, July). Adaptive selection methods for genetic algorithms. In *Proceedings of an International Conference on Genetic Algorithms and their applications* (pp. 101-111).

[5] Ball, M. (2018). 3D Surface Plotter. Retrieved from `https://academo.org/demos/3d-surface-plotter/`

[6] Bengio, Y., LeCun, Y., & Henderson, D. (1994). Globally trained handwritten word recognizer using spatial representation, convolutional neural networks, and hidden Markov models. In *Advances in neual information processing systems* (pp. 937-944).

[7] Biles, J. A. (1994, September). GenJam: A genetic algorithm for generating jazz solos. In *ICMC* (Vol. 94, pp. 131-137).

[8] Bonabeau, E., Dorigo, M., & Theraulaz, G. (2000). Inspiration for optimization from social insect behaviour. *Nature, 406*(6791), 39.

[9] Convolutional Neural Network. (2018, May 18). Retrieved from `www.worldlibrary.org/articles/eng/Convolutional_neural_network`

[10] CUDA Zone. (2018, May 14). Retrieved from `https://developer.nvidia.com/cuda-zone/`

[11] Dean, J, Corrado, G., Monga, R., Chen, K., Devin, M., Mao, M., ... & Ng, A. Y. (2012). Large scale distributed deep networks. In *Advances in neural information processing systems* (pp. 1223-1231).

[12] De Jong, K. A. (1975). An analysis of the behavior of a class of genetic adaptive systems. (Doctoral dissertation, University of Michigan). *Dissertation Abstracts International, 36*(10), 5140B. (University Microfilms No. 76-9381).

[13] De Jong, K. A. (Ed.) (1993). Evolutionary computation (journal). MIT press, Cambridge MA.

[14] Dertat, A. (2017, Nov 8). Applied Deep Learning - Part 4: Convolutional Neural Networks. Retrieved from `https://towardsdatascience.com/applied-deep-learning-part-4-convolutional-neural-networks-584bc134c1e2`

[15] Donahue, J., Anne Hendricks, L., Guadarrama, S., Rohrbach, M., Venugopalan, S., Saenko, K., & Darrell, T. (2015) Long-term recurrent convolutional networks for visual recognition and description. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 2625-2634).

[16] dos Santos, C., & Gatti, M. (2014). Deep convolutional neural networks for sentiment analysis of short texts. In *Proceedings of COLING 2014, the 25th International Conference on Computational Linguistics: Technical Papers* (pp. 69-78).

[17] Eiben, A. E., Raue, P. E., & Ruttkay, Z. (1994, October). Genetic algorithms with multi-parent recombination. In *International Conference on Parallel Problem Solving from Nature* (pp. 78-87). Springer, Berlin, Heidelberg.

[18] Fraser, A. (2017). Information Extraction, Introduction. Retrieved from `www.cis.uni-muenchen.de/~fraser/information_extraction_2017_lecture/01_introduction_motivation.pdf`

[19] Fukushima, K., & Miyake, S. (1982). Neocognitron: A self-organizing neural network model for a mechanism of visual pattern recognition. In *Competition and cooperation in neural nets* (pp. 267-285). Springer, Berlin, Heidelberg.

[20] Glorot, X., Bordes, A., & Bengio, Y. (2011, June). Deep sparse rectifier neural networks. In *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics* (pp. 315-323).

[21] Goldberg, D. E. Genetic algorithms in search, optimization, and machine learning (1989). *New York, Adison-Wesley.*

[22] Goldberg, D. E., & Deb, K. (1991). A comparative analysis of selection schemes used in genetic algorithms. In *Foundations of genetic algorithms* (Vol. 1, pp. 69-93). Elsevier.

[23] Haupt, R. L., & Haupt, S.E (1998). *Practical genetic algorithms* (Vol. 2). New York: Wiley.

[24] Higuchi, T., Iwata, M., Kajitani, I., Yamada, H., Manderick, B., Hirao, Y., ... & Furuya, T. (1996, May). Evolvable hardware with genetic learning. In *Circuits and Systems, 1996. ISCAS'96., Connecting the World., 1996 IEEE International Symposium on* (Vol. 4, pp. 29-32). IEEE.

[25] Higuchi, T., & Yao, X. (Eds.). (2006). *Evolvable hardware.* Springer Science & Business Media.

[26] Holland, J. H. (1975). Adaptation in natural and artificial systems: an introductory analysis with applicatins to biology, control, and artificial intelligence.

[27] Holland, J. H. (1992). Genetic algorithms. *Scientific american, 267*(1), 66-7.

[28] Kennedy, J. (2011). Particle swarm optimizsation. In *Encyclopedia of machine learning* (pp. 760-766). Springer US.

[29] Kitano, H. (1990). Designing neural networks using genetic algorithms with graph geeration system. *Complex systems, 4*(4), 461-476.

[30] Lee, C. Y., & Antonsson, E. K. (2000, July). Variable Length Genomes for Evolutionary Algorithms. In *GECCO* (Vol. 2000, p.806).

[31] Lee, L. P., & Szema, R. (2005). Inspirations from biological optics for advanced photonic systems. *Science, 310*(5751), 1148-1150.

[32] Mattiussi, C., & Floreano, D. (2007). Analog genetic encoding for the evolution of circuits and networks. *IEEE Transactions on evolutionary computation, 11*(5), 596-607.

[33] Mesnil, G., He, X., Deng, L., & Bengio, Y. (2013, August). Investigation of recurrent-neural-network architectures and learning methods for spoken language understanding. In *Interspeech* (pp. 3771-3775).

[34] Mesnil, G., Dauphin, Y., Yao, K., Bengio, Y., Deng, L., Hakkani-Tur, D., ... & Zweig, G. (2015). Using recurrent neural networks for slot filling in spoken language understanding. *IEEE/ACM Transactions on Audio, Speech, and Language Processing, 23*(3), 530-539.

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

[36] Miller, G. F., Todd, P. M., & Hedge, S. U. (1989, June). Designing Neural Networks using Genetic Algorithms. In *ICGA* (Vol. 89, pp. 379-384).

[37] Miller, B. L.,& Goldberg, D. E. (1995). Genetic algorithms, tournament selection, and the effects of noise. *Complex systems, 9*(3), 193-212.

[38] Montana, D. J., & Davis, L. (1989, August). Training Feedforward Neural Networks Using Genetic Algorithms. In *IJCAI* (Vol. 89, pp. 762-767).

[39] Nowlan, S. J., & Platt, J. C. (1995). A convolutional neural network hand tracker. *Advances in neural information processing systems*, 901-908.

[40] Pearce, P. (1990). *Structure in Nature is a Strategy for Design.* MIT press.

[41] Ruder, S. (2016, January). An overview of gradient descent optimization algorithms. Retrieved from `ruder.io/optimizing-gradient-descent/index.html`.

[42] Scherer, D., Müller, A., & Behnke, S. (2010, September). Evaluation of pooling operatons in convolutional architectures for object recognition. In *International conference on artificial neural networks* (pp. 92-101). Springer, Berlin, Heidelberg.

[43] Schwefel, H. P., & Rudolph, G. (1995, June). Contemporary evulution strategies. In *European conference on artificial life* (pp.891-907). Springer, Berlin, Heidelberg.

[44] Severyn, A., & Moschitti, A. (2015). Unitin: Training deep convolutional neural network for twitter sentiment classification. In *Proceedings of the 9th international workshop on semantic evaluation (SemEval 2015)* (pp. 464-469).

[45] Srinivas, M., & Patnaik, L. M. (1994). Adaptive probabilities of crossover and mutation in genetic algorithms. *IEEE Transactions on Systems, Man, and Cybernetics, 24*(4), 656-667.

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

[47] Tur, G.,Hakkani-Tür, D., &Heck, L. (2010, December). What is left to be understood in ATIS?. In *Spoken Language Technology Workshop (SLT), 2010 IEEE* (pp. 19-24). IEEE.

[48] Valverde-Albacete, F. J., & Peláez-Moreno, C. (2014). 100% classification accuracy considered harmful: The normalized information transfer factor explains the accuracy paradox. *PloS one, 9*(1), e84217.

[49] Vasconcelos, J. A., Ramirez, J. A., Takahashi, R. H. C., & Saldanha, R. R. (2001). Improvements in genetic algorithms. *IEEE Transactions of magnetics, 37*(5), 3414-3417.

[50] Wang, Y. Y., Deng, L., & Acero, A. (2005). Spoken language understanding. *IEEE Signal Processing Magazine, 22*(5), 16-31.

[51] Wright, A. H. (1991). Genetic algorithms for real parameter optimization. In *Foundations of genetic algorithms* (Vol. 1, pp. 205-218). Elsevier.

[52] Yang, J. & Honavar, V. (1998). Feature subset selection using a genetic algorithm. In *Feature extraction, construction and selection* (pp. 117-136). Springer, Boston, MA.

List of Figures

List of Tables