FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Deep Learning of Humor from Gary Larson's Cartoons

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Artificial Intelligence and Game Engineering

eingereicht von

Robert Fischer, BSc.
Matrikelnummer 01425684

an der Fakultät für Informatik
der Technischen Universität Wien
Betreuung: Dr. Horst Eidenberger, Ao.Univ.Prof.

Wien, 7. April 2020

Robert Fischer	Horst Eidenberger

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Deep Learning of Humor from Gary Larson's Cartoons

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Artificial Intelligence and Game Engineering

by

Robert Fischer, BSc.
Registration Number 01425684

to the Faculty of Informatics

at the TU Wien

Advisor: Dr. Horst Eidenberger, Ao.Univ.Prof.

Vienna, 7th April, 2020

Robert Fischer	Horst Eidenberger

Erklärung zur Verfassung der Arbeit

Robert Fischer, BSc.
Stauraczgasse 8/13
1050 Wien
Österreich

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 7. April 2020

Robert Fischer

Danksagung

Ich wünsche meinem Betreuer Professor Horst Eidenberger ein aufrichtiges Dankeschön. Nach meiner Bachelorarbeit habe ich bereits gelernt, was für ein hilfreicher, konstruktiver und vorwärts-denkender Betreuer er ist. Dennoch wurde diese Erwartung übertroffen.

Mein Danke geht an die TU Wien und dem gesamten Lehrpersonal für die beste Informatikausbildung in ganz Österreich und noch weiter. Es war der perfekte Ort um nicht nur den technischen Aspekt, sondern auch um die sozialen Fähigkeiten und Mentalität die notwendig ist um erfolgreich zu sein zu erlernen. Die TU Wien hat mich außerdem durch Stipendien, sowie durch meine Tutorenanstellung für einige Lehrveranstaltungen sehr unterstützt.

Ich hatte viel Glück, weil ich mit vielen Schulfreunden das Abenteuer auf die TU Wien mit mir fortsetzen konnten. Wir haben uns viel geholfen und uns gegenseitig motiviert wo es uns möglich war. Besonders möchte ich mich bei Simon Fraiss und Alfred Soare für die gemeinsame Zeit bedanken.

Schlussendlich, möchte ich mich bei meiner Familie bedanken. Sie haben mich immer unterstützt und geholfen wo sie konnten. Ich möchte mich bei meiner Mutter Doina, meinem Vater Maximilian und meinen vier Geschwistern Verena, Kathrin, Maximilian und Alexander bedanken.

Danke!

Kurzfassung

Das Ziel dieser Diplomarbeit ist es Humor durch Deep Learning basierend auf Gary Larsons Cartoons zu modellieren. Der jüngste Erfolg von Deep Learning in den Bereichen Computer Vision und Natural Language Processing zeigt, dass ähnliche Techniken im Bereich Computational Humor eingesetzt werden können. Das Training von Deep-Learning-Modellen benötigt Datensets mit vielen Trainingsbeispielen, weshalb ich ein neuartiges Datenset mit einigen tausend Cartoons mit Pointe und einer zugehörigen Lustigkeits-Annotation erstellt habe. Das Datenset wurde mit einem eigenen Labelling Tool erstellt, durch eine einzelne Person. Dadurch enthält das Datenset den Humor dieser einen Person. Damit ist es möglich den Humor quantitativ mit den Ergebnissen der Deep-Learning-Modelle oder anderen Personen zu vergleichen.

Nach einer extensiven Datensetanalyse, habe ich mehrere Deep Learning Architekturen entworfen und trainiert. Zuerst habe ich mich auf die visuelle Domäne (Cartoons) mit Convolutional Neural Networks, Transfer Learning und Objekterkennung konzentriert. Danach war der Fokus bei der Text-Domäne (Pointe) mit Long Short-Term Memory Netzwerken, Word Embeddings (deep-learning-basierte und klassische) und automatischem Machine Learning. Abschließend, habe ich alle Erkenntnisse in einer Zwei-Phasen-Architektur vereint.

Unglücklicherweise hat die Evaluation ergeben, dass die Aufgabe noch nicht mit den von mir angewandten Deep Learning Techniken fassbar ist. Ich habe zwei Performance Metriken ausgewählt (Durchschnittlicher Absoluter Fehler und Genauigkeit), sowie einige Baseline-Modelle (Häufigste Klasse, Durchschnittliche Klasse, etc.), aber kein Modell hatte signifikant bessere Ergebnisse als die Baselines. Auf dem Test-Set hatte ein Transfer-Learning-Ansatz die beste Genauigkeit mit 26.10%, während die häufigste Klasse eine Genauigkeit von 24.50% erreicht hat. Sowohl ein Deep-Learning Ansatz, als auch die mittlere Klasse konnten einen durchschnittlichen absoluten Fehler von 1.57 erreichen. Dies zeigt, dass die *semantische Lücke* zwischen Computer und Menschen noch zu groß ist und es mit aktuellen Deep-Learning-Techniken nicht möglich ist Humor einer einzelnen Person (subjektiv) zu modellieren. Es scheint, als sei für diese Aufgabe ein weiterer Durchbruch neben Deep-Learning notwendig.

Abstract

The aim of this thesis is to model humor using deep learning based on Gary Larson's cartoons. The recent success of deep learning in computer vision and natural language processing shows that similar techniques can be applied in the field of computational humor. The training of deep learning models requires a dataset with many training samples, which is why I created a novel dataset containing several thousands of Gary Larson's cartoons, punchlines and corresponding funniness annotations. The dataset was annotated using a custom labelling tool, by the single person. Therefore, the dataset entails the humor of a single person. With this dataset it is possible to quantitatively compare humor with the results of the deep learning models or with other people.

After an extensive dataset analysis, I designed and trained several deep neural architectures. First, focusing on the visual domain (cartoons) using convolutional neural networks, transfer learning and object detection techniques. Afterwards, I focused on the text domain (punchlines) using Long Short-Term Memory networks, several word embeddings (deep learning based and classical) and an automated machine learning approach. Finally, I tried to combine all the findings into a unified two stage architecture.

Unfortunately, the evaluation revealed that this task is not yet tractable by the deep learning techniques applied. I chose two performance metrics (Mean absolute error and accuracy) and several baseline models (most frequent class, mean class, etc.) and no model improved on the baselines significantly. On the test set a transfer learning based approach scored the best accuracy of 26.10%, while the most frequent class scored 24.50%. Both a deep learning approach and the mean class reached a mean absolute error of 1.57. These results show, that the *semantic gap* between computers and humans is too large for current deep learning based approaches to successfully model the humor of a single person. It seems another breakthrough besides deep learning is required for this task.

Contents

As witnesses later recalled,
two small dogs just waltzed into the place,
grabbed the cat, and waltzed out.

Cartoon by Gary Larson

CHAPTER 1

Introduction

1.1 Aim

This diploma thesis aims to research whether computers are capable of understanding humor by learning Gary Larson's cartoons. This was done by training several deep neural networks. Their task was to predict the funniness of a cartoon better than simple baselines. Even though humor is intrinsicly subjective, the available data allowed quantitative comparisons using the chosen metrics (accuracy and mean absolute error). The goal was to find the best neural network architecture for predicting the funniness of a cartoon.

Furthermore, based on Gary Larson's cartoons a new dataset for computational humor was created, which could help further research in this area.

The questions this work tries to answer, are the following:

1. Which neural network architecture is best suited for predicting the associated funniness of each cartoon?
2. Do the trained models outperform previously defined baselines? These baselines include most frequent funniness, average funniness, uniform random funniness and stratified random funniness.
3. If a model outperforms the baselines, does it generalize and understand humor? The model could instead exploit spurious statistical cues in the dataset similar to Niven et al. [74]
4. Which feature domain is best suited for predicting the funniness: Is using both visual and text input better than they perform individually?

On a higher level this thesis tries to bring subjectivity considerations into computational humor. Previous work in this field did not model the humor of a single person, but instead focused on what humans collectively consider humorous. The novel dataset and deep learning enable this new perspective.

1.2 Motivation

Based on the recent success of deep learning in fields such as self-driving cars [45], speech recognition [116], machine translation [115], etc., one could expect similar success in computational humor. Especially natural language processing and image-related tasks have seen significant improvements of the state-of-the-art through deep neural architectures.

Initially the idea of this thesis was, that the trained model might have been able to extract certain patterns of humor. Gary Larson's cartoons can be grouped into several themes. For example stone age, scientist or cats versus dog are common themes he uses. A competent model could distinguish these themes and might find patterns of humor among these subjects. Someone might find stone age themed cartoons funnier than cats versus dogs cartoons.

Important to note is that preference is very subjective. The model must be trained by data of the same person, because otherwise the personal preference might be lost. If trained successfully it would open a whole set of new research possibilities: The study of humor of people through the lense of deep neural networks. This is especially interesting, since it has been shown that there are key similarities between how deep neural networks work and how the human brain works [19].

For example if such a model were possible, it could possibly allow a more formal comparison of humor between different authors or different people. The same model might be able to be transferred to the work of other authors. Depending on how well this transfer works, one could argue whether the humor of these authors are similar or not, given the annotator's sense of humor. Also the trained model themselves could contain interesting features. Possibly a functioning model could develop a cat-detecting neuron, which could reveal insights into human humor.

Another key motivation was in the creation of the underlying dataset: This thesis uses a novel dataset, where each cartoon was paired with a funniness annotation of the same person. This allows to quantitatively evaluate the humor of a single person, as the dataset is comparatively large. The scope of the dataset is also unique, as there is no other dataset with similar cartoons available. In future the same dataset could be used for other tasks. A very hard, but very interesting task would be using a generative architecture, such as generative adversarial network (GAN) to generate a funny cartoon image from a punchline. Or vice versa: Generating a funny punchline from a cartoon image. And finally generating completely new funny cartoons with fitting punchlines [38][98].

In recent years more and more deep learning techniques have been applied in the field of computational humor. One problem of these approaches is that they do not take the subjectivity of this problem into account. Humor differs from person to person and also changes over time significantly. For example in Chiruzzo et al [18] the corpus was extracted from twitter where each tweet was classified as humorous or not humorous. The ground truth classification was annotated using crowd sourcing. The problem with such datasets is that the funniness is determined by some form of average. Some individuals might find the same text very humorous, while other individuals might find it not humorous at all. The crowd sourcing hides this fact by averaging. Therefore, these datasets and models usually do not learn the humor of a single person, but instead what most people generally find funny (or not funny). This is very important and also reveals interesting insights into humor, but this thesis' motivation is to focus more on the subjectivity of this topic.

This work finally shows that humor is still a hard problem and current deep learning architectures are not yet capable of generalizing humor in the chosen context of this thesis. Further research is still needed, especially since, if successful, it would enable many new research possibilities.

1.3 Methodology

Firstly I performed a detailed literature survey. The goal was to get a grasp of the state-of-the-art of deep learning in the field of computational humor, image classification, natural language processing and humor in general.

Next I did the data acquisition, in which the ground truth was created: A dataset of Gary Larson's cartoons with funniness annotations was created. These cartoons were prepared such that they could be used for training the neural networks effectively. This included cropping, resizing and filtering the cartoons. Extracting the punchlines, as well as annotating the funniness.

I performed the following steps iteratively. This allowed to adapt to the challenges which occured during the course of this research:

1. I picked an architecture candidate and designed it accordingly. This was based on prior knowledge and previous literature research.
2. Then I implemented and trained the architecture.
3. Afterwards I evaluated the resulting model. Problems were determined and possible solutions analyzed.

At first the focus was on visual-only models. Then the focus shifted towards text-only models. Finally I evaluated several approaches that combine both text- and visual

features into a single model. Also the architectures went from simple to more complex per iteration.

Finally I performed a detailed evaluation. The most interesting models were picked and analyzed thoroughly. Goal of this phase was to find answers to the questions listed in section 1.1.

1.4 Overview over thesis

This section gives a short overview over the contents of this thesis.

Chapter 2 (Background) contains the background necessary for this thesis. Topics covered are computational humor, machine learning, artificial intelligence, neural networks, natural language processing, automated machine learning and an overview over related work.

Chapter 3 (Design) covers the iterative design process of this thesis. This process began with the visual component of Gary Larson's cartoons using convolutional neural networks. Next, design shifted towards the development of deep neural networks for punchlines using LSTMs, AutoML, feed forward neural networks and word embeddings. And finally this chapter outlines the combination of both domains in a single architecture.

Chapter 4 (Implementation) focuses on the implementation aspects of the previously described neural network architectures. The reader gets an overview over the technology chosen and I implemented the the architectures.

Chapter 5 (Evaluation) gives the reader an insight into the final results obtained by selected interesting architectures, compared to the baselines. I examine the inner working of the models, outlining their mistakes using confusion matrices and other techniques.

Chapter 6 (Conclusion) finalizes this thesis with lessons learned during the course of the development of this work.

CHAPTER 2

Background

As this work combines the field of computational humor with deep learning, an understanding of both fields is required. This chapter contains the required background for this thesis. First an overview over artificial intelligence and machine learning is outlined. Section 2.3 lays out techniques of natural language processing. Section 2.4 gives an overview over a technique called transfer learning. The next section (2.5) outlines automated machine learning. Based on the previous sections I summarize the research area of computational humor in section 2.6. This chapter ends with related work of traditional computational humor and deep learning based computational humor in section 2.7.

2.1 Machine Learning and Artificial Intelligence

The field of Artificial Intelligence (AI) tries to reproduce (human) intelligence using computers [73]. It is yet impossible to develop a truly intelligent computer program, also known as strong AI [64]. AI researchers typically pick certain narrow subtasks of what is considered to require intelligence. Such a system is also known as weak AI [64]. These tasks usually entail some utility for other application domains and range from image classification, self driving cars to machine translation.

2.1.1 History of Artificial Intelligence

In the year 1950 Alan Turing proposed the idea of a test which could determine if machines have human-level intelligence. In this test a human chats with a human and a computer. Both try to convince the human that they are a real human being. If the human is not reliably able to tell them apart, then the computer passes the Turing test [111].

In the 1960s and 70s AI researchers overestimated the possibilities of their systems greatly. For example, Marvin Minsky (a key researcher back then) said in 1970: "In from three to eight years we will have a machine with the general intelligence of an average human being." The goal of artificial intelligence initially seemed tractable: The computational power of computers increased exponentially and new algorithms and methods were developed [100][73]. The first chatbot (ELIZA) by Joseph Weizenbaum which showed how communication between humans and machines could be implemented using relatively simple rules seemed promising [114].

There were many schools of thoughts, among others: The symbolic-based approach and the connectionism-based approach [63]. Symbolic-based systems reached early success, by applying logic reasoning to AI [63]. The idea of connectionism was to model AI systems using the human brain as a blueprint [63]. In the end of the 60s many AI researchers lost interest in connectionism, as the limitations of the perceptron were revealed [68]. This insight halted the research in this area for many years [73].

The symbolic-based approach initially showed much success in several tasks (for example: theorem proving [97]) but also reached a limit. The general approach was to model reasoning as a form of search [97]: Proceed step-by-step towards the goal and backtrack if the steps lead to a dead end. The problem with this approach is the sheer size of the search space. The combinatorial explosion is intractable and can only be handled by heuristics, which limit the power of the system [97]. The limiting capabilities of perceptrons and the lack of progress in key areas resulted in the first AI winter in which funding was cut to a minimum [22].

Expert systems dominated the 1980s [63]. These were systems which were crafted by human experts and entailed their knowledge of the specific domain it tried to model using logical rules [73]. One important contribution of expert systems was their usefulness. Even early experiments managed to diagnose infectious blood diseases [73].

Research in the field of connectionism continued as well. Backpropagation solved issues of previous attempts [73][97]. Training larger neural networks was now tractable. One of the most popular architectures of the time is the Hopfield network which models an associative memory [40]. Still, the expectations were not met and the second AI winter emerged during the late 1980s [63]. This time it not only hit the scientific community, but also many AI companies [73].

During the 1990s AI had a slightly negative connotation [61]: AI-based systems were working in the background and many problems were largely solved (for example speech recognition and information retrieval [61][99]). A big milestone of AI research was achieved, when Deep Blue managed to win against the best chess player of the time, Garry Kasparov [73]. This was largely due to the increased computational power and clever engineering, instead of some new paradigm [73].

During this period the collaboration with other research fields began [97]. For example, knowledge from mathematics and economics was applied in AI. Especially decision theory

and probability theory brought in very important techniques, such as hidden Markov models and Bayesian networks [82][97].

The next boom in the field of AI was fueled by Deep Learning and the success it had in many problems of artificial intelligence [45][115][116], beginning with the convolutional neural network AlexNet which was the first approach that reached a top-5 error of 15.3% on the ImageNet Large Scale Visual Recognition Challenge in 2012 [52].

The boom continued in 2016 with AlphaGo winning against one of the best human player of the board game Go [106]. Compared to chess Go is a much more difficult game to play for computers, as the combinatorial explosion is much larger and the heuristics available are much weaker in Go compared to chess [106].

The next frontier of artificial intelligence is not clearly established yet, as the progress of deep learning is very fast. For example the team behind AlphaGo applied similar techniques to real time strategy games successfully [5]. It seems that especially tasks requiring long time planning or deep understanding of the underlying structure are a problem [69]. For example this thesis shows that understanding humor is still very challenging.

2.1.2 Machine Learning

Machine Learning (ML) is a domain of AI [97]. Machine learning tries to build a model which generalizes as good as possible to previously unseen data by learning automatically from a training dataset [11]. Learning means extracting patterns from the dataset by applying a machine learning algorithm [11]. Models can be parametric or non-parametric. A parametric model means that the general function is fixed and can only be adjusted by adapting the parameters [97]. For example neural networks are parametric, as gradient descent adjusts the weights and biases for each neuron during training, while the general structure of the neural network is not. In contrast, for a non-parametric model the machine learning algorithm learns the function itself [97]. An example for this type of model would be k-Nearest Neighbor. Since the focus of this work are neural networks we will discuss parametric models in more detail.

More formally, a model is a function $\boldsymbol{y}(\boldsymbol{X}, \boldsymbol{\theta})$ where $\boldsymbol{X}$ is the training data and $\boldsymbol{\theta}$ are the parameters of the model [11]. Through some optimization algorithm (for example gradient descent) the model is adjusted such that it generalizes ("learns") the underlying structure [40]. This model should make good predictions for previously unseen samples. Important is the distinction between model parameters ($\boldsymbol{\theta}$) and model hyperparameters [97]. Model hyperparameters are parameters which cannot be estimated from the available data and are typically set by a human heuristically [21]. A parameter on the other hand, can be estimated from the data, given the previously set hyperparameters [40].

The data $\boldsymbol{X}$ is a matrix where each row is a sample and each column a feature. A feature is an individual measurable property or characteristic of a phenomenon being observed [11]. For example when trying to apply a machine learning task on an image one could

Regression Value	< 50	$51 - 100$	$101 - 200$	$200 <$
Classification Class	CLASS 50	CLASS 75	CLASS 150	CLASS 250

Table 2.1: Example for the conversion between regression and classification tasks

use the red, green and blue channel values of each pixel as features. Choosing appropriate features is a difficult task and can make or break the performance of the trained model.

Machine learning tries to approximate a function from the selected dataset. Depending on the available information, different strategies must be applied. Typically, the most convenient and preferred mode is supervised machine learning [97]. Supervised means that not only the data $\boldsymbol{X}$ is available, but also the target values $\boldsymbol{t}$. This means that the problem can be relatively easily solved using an optimization algorithm. More formally we try to find a $\boldsymbol{\theta}$ such that $\boldsymbol{y}(\boldsymbol{X}, \boldsymbol{\theta}) = \boldsymbol{t}$. If the $\boldsymbol{t}$ labels are not available, then unsupervised machine learning is typically applied [97]. For example clustering methods allow unsupervised ML. If a mixture of labelled and unlabelled data is available, machine learning practitioners apply semi-supervised machine learning [120].

Additionally, there is a distinction between classification and regression tasks [11]. If $\boldsymbol{t}$ is a continuous variable, machine learning practitioners call it a regression task and the model is a regressor. Otherwise, if $\boldsymbol{t}$ is categorical and can only take finite values, then machine learning practitioners call it a classification task and the model is a classifier. An example for a regression task is stockmarket prediction, where the model predicts the price of some stock. An example for a classification task is spam detection. Such a model predicts whether an e-mail is spam or not (binary classification). Often a regression task can be modelled as a classification task by converting the features using discretization and vice versa [30]. Instead of predicting the stock market directly, one can instead predict classes and use the average value of each bucket as the predicted stock price. Refer to table 2.1 for an example. The right approach is highly task dependent [97].

2.1.3 Training of Machine Learning models

The most important property of a model $\boldsymbol{y}$ is the generalization power for unseen data [11]. Training a model $\boldsymbol{y}$ which predicts the training data $\boldsymbol{X}$ perfectly is trivial: The machine learning algorithm could always build a lookup table which memorizes a mapping from the training samples to the target labels $\boldsymbol{t}$ [25]. If this happens, it is called *overfitting* [25].

The *holdout method* facilitates the approximation of the performance of a model $\boldsymbol{y}$ for previously unseen data [94]. There is often some bias towards the original dataset [110]. The better the dataset characterizes the underlying problem, the better the final model may perform practically [93]. For example, if one trains a vehicle detection model, but the dataset does not include any images of motorcycles, then the model will perform badly in the real world. Regardless of the method applied, the model would not be able to detect motorcycles - despite the performance metrics possibly indicating otherwise.

The holdout method first splits the data into three distinct subsets: Training, validation and test set [94]. The practitioner incrementally adjusts the hyperparameters and trains on the training set, based on the performance of the model on the validation set [94]. Finally, to gather the true performance the practitioner has to evaluate on the test set [94]. The following algorithm illustrates the holdout method using pseudo code:

Algorithm 2.1: Holdout method [94]

```
Take the dataset X and split randomly into three distinct subsets: Training,
  validation and test set.
while ¬some halting criterion do
    Train the model on the train split.
    Check performance of the trained model on the validation set.
    Manually adjust hyperparameters such that the model generalizes better
end
Check final performance on the test set of the best trained model
```

This variant is also commonly referred to as the three-way holdout method [94]. The validation/test split is only needed if there are hyperparameters which are being optimized iteratively [94]. Due to the lack of standardization, sometimes the validation set is actually called the test set and the test set the validation set [95] [40].

Commonly used halting criteria include:

- Maximum number of iterations [49]
- Plateauing performance: No improvement after a previously defined number of iterations [89]
- Maximum performance: The model reaches a certain performance threshold [49]
- A combination of multiple halting criteria [49]

If there is no clear separation between the test and training set, we call it a data leakage [103]. The actual performance of the trained model $\boldsymbol{y}$ could be potentially much worse for unseen data than the metrics suggest. Selecting the best performing model, instead of the last model is called early stopping or the pocket algorithm.

Another method for evaluating the performance of a machine learning model is called *cross-validation* [94]: This method randomly splits the data into n partitions. Then n models are trained for each partition, where the partition is the test set and the remaining data is the training set. In general cross validation approximates the performance better than the holdout method [97]. However, as many models are trained (depending on n) it can be computationally very expensive [11].

The following pseudo code illustrates cross-validation, which first splits the data in a $\boldsymbol{TRAIN}$ and a $\boldsymbol{TEST}$ set [94]. Then incrementally the $\boldsymbol{TRAIN}$ set is split into

k partitions $\boldsymbol{P}$. For each $(\boldsymbol{TRAIN} \setminus \boldsymbol{P}, \boldsymbol{P})$ pair a model is trained and evaluated [11]. Based on the overall performance the practitioner adjusts the hyperparameters accordingly. Finally the $\boldsymbol{TEST}$ set evaluation returns the performance on unseen data [94].

Algorithm 2.2: Cross-validation method [94]

```
1  Take the dataset X and split randomly into two distinct subsets: Training
     (TRAIN) and test set (TEST).
2  while ¬some halting criterion do
3  |  Take the train set TRAIN and split into k partitions
4  |  foreach partition P do
5  |  |  Train the model on TRAIN \ P
6  |  |  Calculate performance on P
7  |  end
8  |  Calculate average performance
9  |  Manually adjust hyperparameters such that the model generalizes better.
10 end
11 Check final performance on the test set TEST of the best trained model
```

Depending on the machine learning task and applied methodology, there is no distinct test set [94]. Figure 2.1 compares the splits of holdout method and cross validation.

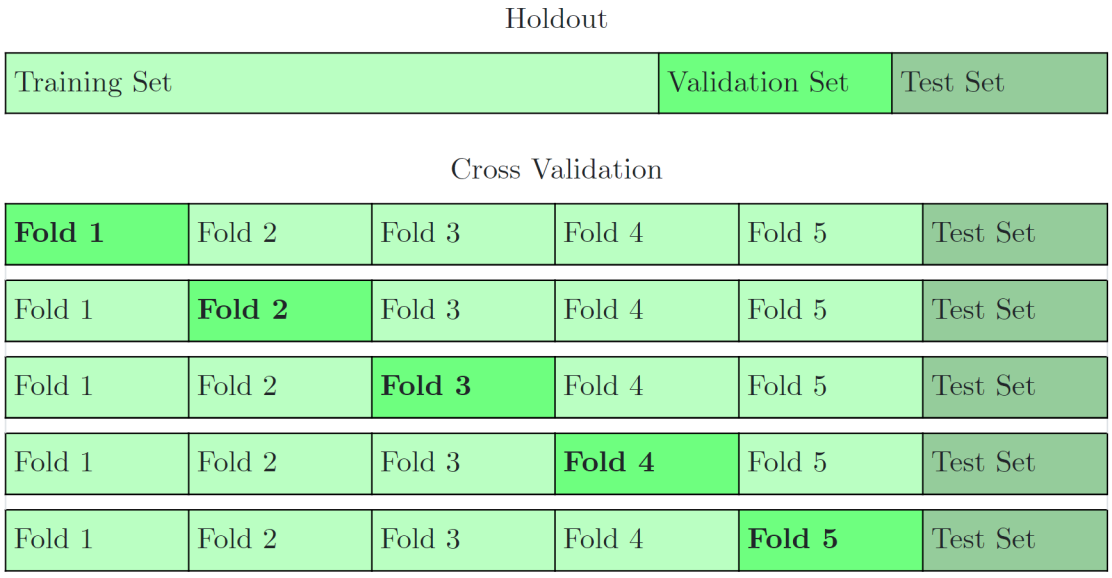


Figure 2.1: Comparison of the splits between holdout and 5-fold cross validation

2.1.4 Evaluation of Machine Learning models

Measuring the performance of a model $\boldsymbol{y}$ is an important aspect of proper machine learning [32]. Choosing the performance measure is highly task dependent [32]. For this thesis the mean absolute error (MAE) and the accuracy were selected [34]. Other metrics considered were F1 and mean squared error (MSE) [34] [87].

Mean absolute error and mean squared error

$$\text{MAE} = \frac{1}{n}\sum_{i=1}^{n} \mid \hat{X}_i - X_i \mid \tag{2.1}$$

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n} \left(\hat{X}_i - X_i\right)^2 \tag{2.2}$$

where $\hat{X}$ is the predicted sample
X is the ground truth

MAE is interesting because a near miss is less penalized [34]. For example, if a model predicts funniness of 5, while the real funniness is 6, the term $\mid 5-6 \mid = 1$. On the other hand if the predicted funniness were a 2, the term $\mid 2-6 \mid = 4$. The lower the MAE the better.

MSE works similarly to MAE, but big errors are penalized more [34]. The calculations for the above examples are $(5-6)^2 = 1$ and $(2-6)^2 = 16$, which is a significant difference. For a very noisy dataset MAE is usually better suited than MSE.

Accuracy and F1

$$\text{Accuracy} = \frac{|\text{True Positives}| + |\text{True Negatives}|}{|N|} \tag{2.3}$$

where	|True Positives|	is the number of correctly classified samples of the positive class
	|True Negatives|	is the number of correctly classified samples of the negative class
	|N|	is the total number of samples

Accuracy is easy to understand and is commonly applied in classification tasks [34]. The higher the accuracy the better [87]. When applied on an imbalanced dataset, the accuracy metric might not sufficiently describe the performance of the model, as it is trivial to increase the accuracy by returning the most frequent class of the dataset [34]. Consider the following example: The task is evaluating a model which predicts whether a patient has cancer or not. The test dataset is highly imbalanced, and contains 99 samples of patients without cancer and 1 sample of a patient with cancer. If the model always predicts that a patient has no cancer, the resulting accuracy is 99%.

F1 scores handles such cases better, by emphasizing false negatives and false positives more [87]. It is the harmonic mean of the precision and recall score [87]:

$$\text{Precision} = \frac{|\text{True Positives}|}{|\text{True Positives}| + |\text{False Positives}|} \tag{2.4}$$

$$\text{Recall} = \frac{|\text{True Positives}|}{|\text{True Positives}| + |\text{False Negatives}|} \tag{2.5}$$

$$\text{F1} = 2 * \frac{\text{Precision} * \text{Recall}}{\text{Precision} + \text{Recall}} \tag{2.6}$$

where	|True Positives|	is the number of correctly classified samples of the positive class
	|True Negatives|	is the number of correctly classified samples of the negative class
	|False Negatives|	is the number of incorrectly classified samples of the negative class
	|False Positives|	is the number of incorrectly classified samples of the positive class

For multi class classification (more than two possible classes) a confusion matrix helps to visualize the predictions of a model [34]. The rows of a confusion matrix are usually predicted classes and the columns the ground truth. A perfect model would show 100% along the diagonal of the matrix, as there would be no wrong classifications.

A confusion matrix usually reveals common mistakes of a model [34]. For example if a model never predicts a class, it can be easily detected using this visualization. For an example of a confusion matrix please refer to figure 5.3.

2.2 Artificial Neural Networks

One of the first artificial neural network, was the perceptron [73]. It is a binary classifier and uses a linear function to perform classification [68]. *Binary* refers to the fact that a simple perceptron can only distinguish between two states. The learning process starts with a random decision boundary, which is iteratively adapted such that the number of misclassifications is minimized [97]. Due to the linear decision boundary the perceptron is only able to learn decision boundaries that are linearly separable [73]. For example, a simple XOR cannot be learned by a basic perceptron. This motivated the introduction of multi layer perceptrons (MLPs), since they are in theory able to approximate any function, given a nonlinear activation function [24]. An MLP consists of multiple perceptrons grouped in layers and each layer connects with the previous layer [11]. MLPs are feed forward neural network [11].

Part of the reason why artificial neural networks are so powerful, especially deep neural networks, stems from their flexibility [45][116][115]. They can adapt to many different domains very easily. Illustrated at figure 2.2 is a fully connected feed forward neural network with 3 hidden layers. Neural networks can have many more layers (refer to section 2.2.5) or even contain loops (section 2.2.7).

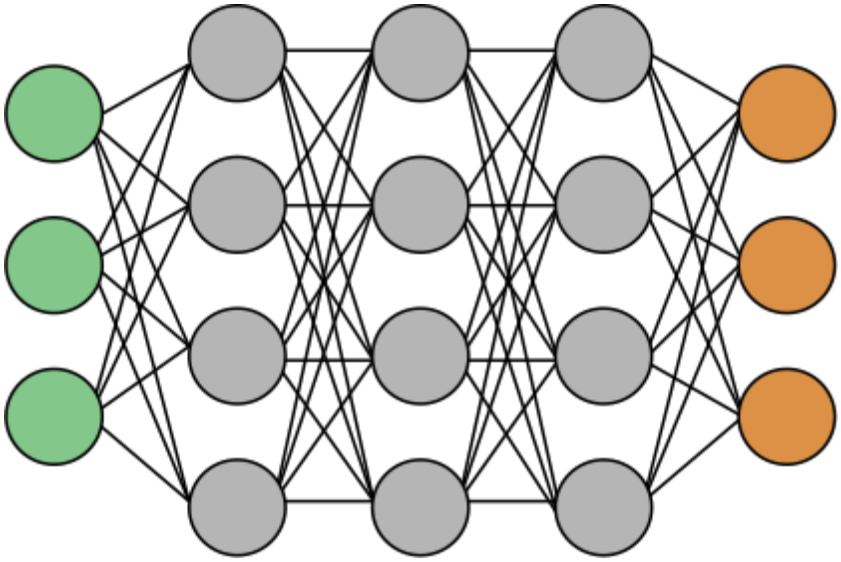


Figure 2.2: Fully Connected Feed Forward Neural Network

A neural network generally consists of multiple neurons, layers and connections [97]. The general inspiration for them is the human brain which also consists of similar building blocks [40].

In the human brain neurons receive their signals through dendrites (connections in neural networks) and are then sent through the axon to other neurons [40]. The signal strength is adjusted (analogous to activation functions in neural networks) such that other neurons in the brain are more stimulated [40]. Note that key parts of the human brain are not simulated in a neural network: During training the weights and biases are adjusted using gradient descent (for more information refer to section 2.2.2) [23]. Nevertheless artificial neural networks (especially convolutional neural networks) produce similar structures also observed in the human brain [19].

2.2.1 Basics of artificial neural networks

From a different perspective, neural networks can also be seen as a series of functional transformations of some input vector into an output vector [78]. Given the input $x_1, ..., x_D$ and M connections the input layer can be described as:

$$a_j = \sum_{i=1}^{D} w_{ji}^{(1)} x_i + w_{j0}^{(1)} \tag{2.7}$$

where $j = 1, ..., M$. $w_{ji}^{(n)}$ is usually referred to as weights and $w_{j0}^{(n)}$ as biases [11]. a_j is

called an activation or sometimes potential, which is then transformed using a nonlinear, differentiable activation function $h(\cdot)$ [11]:

$$z_j = h(a_j) \tag{2.8}$$

Depending on the problem different activation functions are applied. Common functions include:

- Tanh function [76]: zero-centered function between -1 and 1 [76]. Suffers from the vanishing gradient problem [76].

$$h(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{2.9}$$

- Sigmoid Function [40]: "Squashes" the input value between 0 and 1 [40].

$$h(x) = \frac{1}{1 + e^{-x}} \tag{2.10}$$

- ReLU (Rectified Linear Unit) [37]: Very commonly used in deep learning, as it performs very well on a wide range of different tasks [76]. Can be calculated very quickly, as it does not contain any division or exponential [76].

$$h(x) = \max(0, x) \tag{2.11}$$

- SoftMax [37]: Used to transform any vector into a probability distribution [76]. This is commonly used for the final layer of classification tasks with multiple classes (multi class classification) [76].

$$h(\boldsymbol{x}) = \frac{e^{x_i}}{\sum_{k=1}^{M} e^{x_k}} \tag{2.12}$$

- Logistic Sigmoid [37]: Similar to SoftMax, but used for binary classification [37].

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{2.13}$$

For the first hidden layer of a neural network the following values are linearly combined [11]:

$$a_k = \sum_{i=1}^{M} w_{kj}^{(2)} z_j + w_{k0}^{(2)} \tag{2.14}$$

where $k = 1, ..., K$ is the total number of outputs of the first hidden layer. The same procedure is repeated for the remaining hidden layers [11]. The final result for classification tasks is usually obtained by applying a softmax activation function of the last layer's output [11].

Combining these transformations together for a relatively simple two layer network results in the following equation:

$$y_k(\boldsymbol{x}, \boldsymbol{w}) = \sigma\left(\sum_{i=1}^{M} w_{kj}^{(2)} h\left(\sum_{i=1}^{D} w_{ji}^{(1)} x_i + w_{j0}^{(1)}\right) + w_{k0}^{(2)}\right) \tag{2.15}$$

This term can be simplified by assuming that the input vector x has always a constant term with the value 1 [11], as this removes the additional addition for the bias for each layer.

$$y_k(\boldsymbol{x}, \boldsymbol{w}) = \sigma\left(\sum_{i=1}^{M} w_{kj}^{(2)} h\left(\sum_{i=1}^{D} w_{ji}^{(1)} x_i\right)\right) \tag{2.16}$$

Evaluating 2.16 is called forward propagation [78].

2.2.2 Training of neural networks

For training a loss function $L(\boldsymbol{w})$ is defined, where $\boldsymbol{w}$ are the trained weights [37].

The loss can have many different forms, but in the following we will discuss primarily the cross entropy loss, since it is commonly used for classification tasks [88].

$$L(\boldsymbol{w}) = -\sum_{n=1}^{N} \left(t_n \ln y_n + (1 - t_n) \ln(1 - y_n)\right) \text{ where } y_n = y(\boldsymbol{x_n}, \boldsymbol{w}) \tag{2.17}$$

Often we want to calculate the loss for multiclass classification problems. Here we encode the labels as K mutually exclusive one hot-encoded-classes [88]. One-hot-encoded means that a classification $c \in \{1, ..., K\}$ is represented using a vector of size K where the c-th component is set to 1 and every other component is set to 0. Suppose we have 3 classes and the classification for a sample is 2, then the one-hot-encoding of this class is $(0, 1, 0)$. We apply the following loss function for multiclass classification:

$$L(\boldsymbol{w}) = -\sum_{n=1}^{N} \left(\sum_{k=1}^{K} \left(t_{kn} \ln y_k(\boldsymbol{x_n}, \boldsymbol{w})\right)\right) \text{ where } y_k(\boldsymbol{x}, \boldsymbol{w}) = \text{SoftMax}(\boldsymbol{y}(\boldsymbol{x_k}, \boldsymbol{w})) \tag{2.18}$$

For regression tasks the L1 loss can be used [34]:

$$L(\boldsymbol{w}) = \frac{1}{N} \sum_{n=1}^{N} |x_n - y_n| \text{ where } y_n = y(\boldsymbol{x_n}, \boldsymbol{w}) \tag{2.19}$$

The goal of training neural network is to minimize the loss of the model [88]:

$$\arg\min \quad L(\boldsymbol{w}) \tag{2.20}$$

Training takes advantage of the fact that it is a function composition of differentiable functions [37]. This allows the use of a technique called gradient descent [40]. The idea is intuitive: Walk along the path with the steepest descent until some halting criterion is met [40]. Because the neural network is differentiable the steepest descent is relatively easily obtained, by taking the negative of the first derivative of the loss function [37]. A halting criterion could be one of those described in section 2.1.3. Algorithm 2.3 illustrates the pseudo code of gradient descent.

Algorithm 2.3: Gradient descent pseudo code [40]

```
while ¬some halting criterion do
    Compute gradient: w' = ▽L(w)
    Update weights: w := w − αw'
end
```

$\alpha > 0$ is a hyperparameter and called the learning rate [37]. α is usually set empirically [37]. A too high learning rate (for example $\alpha = 1$) means that the loss of the neural network could oscillate violently, where the loss function often increases significantly [37]. The lower α the more likely it is that the gradient descent converges in a local optima [37]. Figure 2.3 illustrates the process of gradient descent.

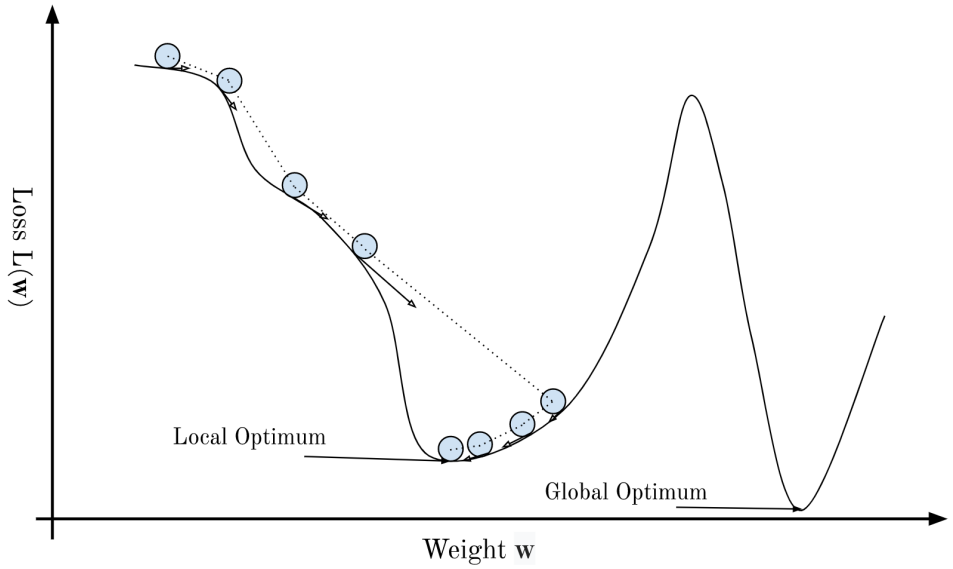


Figure 2.3: Visual representation of gradient descent with momentum. Longer arrows mean steeper gradient. [37]

2.2.3 Applying artificial neural networks

The naive implementation of gradient descent has problems, which can be mitigated by different measures. Usually a technique called momentum is applied [37]. A real world analogy would be a ball rolling down a steep hill. Over time, the steeper the gradient, the faster the ball rolls towards the local minimum. This makes the gradient descent algorithm to be less prone for local minima and usually causes faster converging to the desired optimum [37].

Another technique is called minibatching [37]: Here instead of calculating the gradient of the entire dataset $\boldsymbol{X}$, a small subset of $\boldsymbol{X}$ is taken [37]. This not only reduces the amount of computation needed, but also makes it such that local minima are less of a problem, as the gradient is slightly different depending on the minibatch [37].

Obtaining the gradient is also a computationally complicated step. The usual forward derivation method commonly learned in high school does not scale well for many parameters [78]. Therefore a technique called backpropagation is commonly used [78]. For more information, refer to the next section.

In general, overfitting is a problem of neural networks (and other machine learning algorithms) [97]. To counteract this phenomena we can apply a different loss function with *regularization*, synthetically increase the dataset size using *data augmentation* or randomly dropping certain neurons with *dropout layers* [37]. Note that these techniques are not exclusive and are usually applied in combination [37].

The idea of data augmentation is to synthetically increase the size of the dataset by applying transformations on the original data [105]. No transformation must alter the original label of the sample. Image data usually allows for many different kind of filters and transformations to be applied [105]. Common operations are random crop, random rotation and adding synthetic noise [105]. In general, this technique allows for better generalization of the trained model [105]. For example, a model which is trained on a cats versus dogs image dataset with a rotation transformation applied might distinguish rotated cats and dogs better, compared to a model trained on a dataset without this augmentation. On the other hand the transformations could unnecessarily force a model to generalize [105]: Applying a skew transformation on the cats versus dogs dataset might make the performance even worse than the trained model.

Regularization tries to avoid overfitting by penalizing high weights of neurons [97]. This avoids certain inputs from dominating the outputs and nudges the network to make use of all inputs [88]. This can easily be achieved by adapting the loss function slightly [11][88]:

$$L_{reg}(\boldsymbol{w}) = \frac{\delta}{2} \|\boldsymbol{w}\|^2 + L(\boldsymbol{w}) \tag{2.21}$$

where δ is called weight decay.

A dropout layers can be used as well [37]. The idea is similar to regularization: The network should not overly rely on certain inputs [37]. A dropout layer turns off neurons with a certain probability [88]. This forces the network to generalize better [37]. Often in CNNs dropout layers are placed at the end of the network before the last fully connected layer [37].

Due to the sheer amount of hyperparameter and possible approaches, successfully designing and training of a neural network can be a challenging task [40]. There are many rules of thumb for many hyperparameters but in the end it is in the hand of the developer to use a good mixture of clever tricks, as well as trial and error to find a good neural network model [37]. Automated machine learning is also a very promising research field, more information is available in the next section.

The architecture of the neural network also influences how much it tends to overfit: As a rule of thumb, the more neurons a neural network has, the bigger the learning capacity of the neural network and therefore it overfits easier if the dataset is not sufficiently large [37]. Empirically, deep networks (many layers) have shown to generalize better compared to less deep networks with a similar number of neurons [37].

2.2.4 Backpropagation

Backpropagation is an alternative way of determining the gradient of a function [37]. Backpropagation makes training big neural networks tractable for modern computers [11]. Due to its useful properties backpropagation has been reinvented independently in several fields [39]. It is also known as reverse-mode differentiation [78].

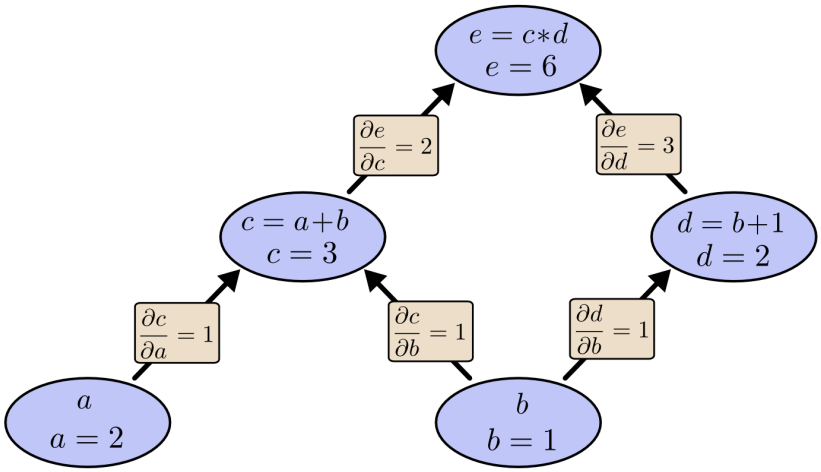


Figure 2.4: A computational graph with derivatives on the edges. [78]

Another view of neural networks is the view of computational graphs [37]. Neural networks essentially define a graph of simple computations consisting of differentiable operations [37]. In a computational graph derivatives are placed on the edges between nodes. Figure 2.4 illustrates a computational graph.

Calculating the derivative of e with respect to b in the graph illustrated in figure 2.4 can be calculated by summing up over all paths in the computational graph [78]:

$$\frac{\partial e}{\partial b} = 1 * 2 + 1 * 3 \tag{2.22}$$

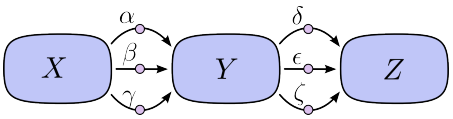


Figure 2.5: A simple computational graph [78]

Summing up over all paths quickly explodes combinatorially, as the number of paths may increase exponentially with the size of the graph [78]. For example, applying traditional forward-mode differentiation ($\frac{\partial}{\partial X}$) the computational graph illustrated in figure 2.5 would result in the following derivative [78]:

$$\frac{\partial Z}{\partial X} = \alpha\delta + \alpha\epsilon + \alpha\zeta + \beta\delta + \beta\epsilon + \beta\zeta + \gamma\delta + \gamma\epsilon + \gamma\zeta \tag{2.23}$$

which could be greatly simplified instead by using backpropagation ($\frac{\partial Z}{\partial X}$) instead:

$$\frac{\partial Z}{\partial X} = (\alpha + \beta + \gamma)(\delta + \epsilon + \zeta) \tag{2.24}$$

Backpropagation starts at the output node and works towards the root nodes by calculating the derivative of $\frac{\partial Z}{\partial}$ [78]. Figure 2.6 outlines backpropagation on a computational graph. Additionally this process also returns a derivative with respect to the output of every node in the graph [37]. This is a very useful property for training neural networks [37].

Backpropagation can be summarized using the pseudo code described in algorithm 2.4.

Algorithm 2.4: Backpropagation algorithm [88]

```
foreach node n do
    Compute local gradient l_c = ∂n/∂c for each child c
    Compute m_c = l_c ∂e/∂n for every child c
    Compute ∂e/∂c by summing up over all m_c
end
```

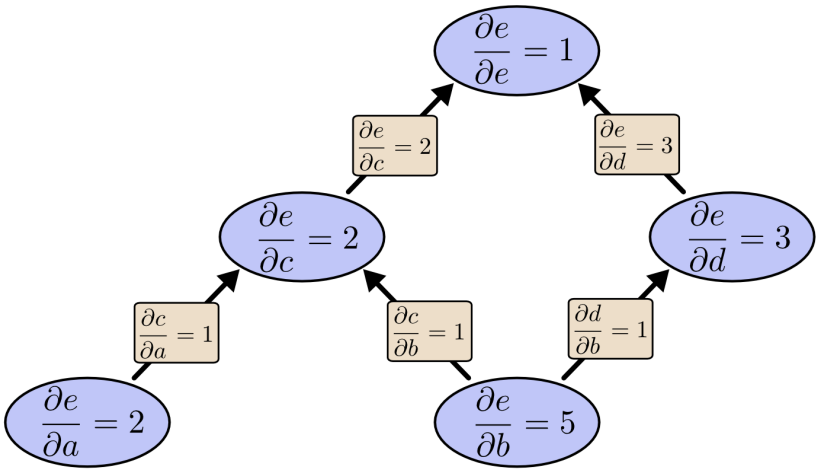


Figure 2.6: Backpropagation applied on computation graph 2.4 [78]

2.2.5 Deep Learning

A neural network with multiple layers, where each layer applies increasingly higher level feature extraction is a deep neural network [101]. The research field is called deep learning [101]. An advantage of deep neural networks is that they scale very well with the size of the dataset and with the amount of computation power available [52]. As the computational resources increase and the size of the datasets as well, the performance of deep learning models increases as well.

The technique of using multiple layers in neural networks has been around for many years [101]. Relatively recently (2012), the field of deep learning has gained a lot of attention, since the introduction of AlexNet [52] as this was one of the first practical deep neural network outperforming more traditional methods [101]. As the basic techniques of deep learning have been around for many years, the exorbitant computational requirements were intractable for many years [101]. Deep learning on a conventional central processing unit is infeasible for large models [101]. Emerging general purpose programming of graphics cards solved this problem: The training of deep neural networks can be very effectively parallelized and graphics cards excel at such tasks compared to CPUs [37]. Training neural networks requires to compute the gradient of a large function [37]. The gradient can be efficiently calculated using backpropagation [97].

Despite backpropagation and large datasets, controlling the number of weights and biases is still an important factor of successful deep learning models [101]. There are many types of layers, which essentially all serve to reduce the number of parameters [37]. For example, convolutional layers are especially suited for image related tasks, as they have a relatively small number of parameters, but perform significantly better than a comparable fully connected layer [101].

A general problem of neural networks and deep neural networks in particular is the lack

of interpretability [36]. For humans it is very difficult to make founded assertions of what the neural network actually uses for its prediction [15]. Neural networks are essentially black boxes with input on one side of the network and some prediction returned at the other side of the network [36]. This is especially a problem of safety critical systems, such as self-driving cars [45]. In this domain it is critical for humans to know on which basis the system has decided and whether it is sensible. Natural language processing is also prone to such "clever hans" behaviour, where the neural network exploits statistical cues of the underlying data, instead of really understanding the task [74]. Interpretable deep learning is an active field of research [36].

2.2.6 Convolutional Neural Networks

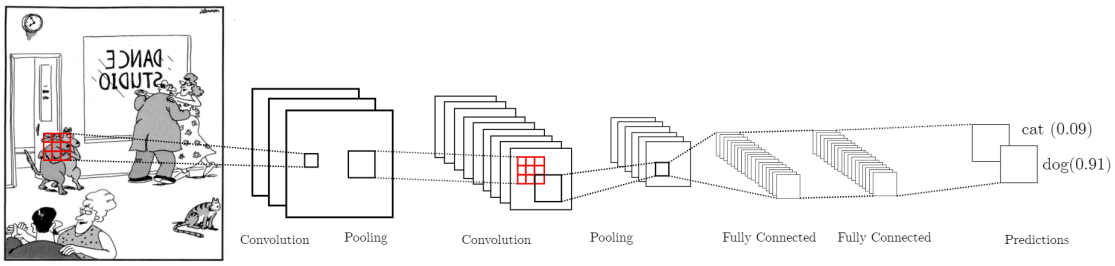


Figure 2.7: Visual representation of a convolutional neural network

Convolutional neural networks were developed based on feed forward neural networks [11]. A convolutional neural network (CNN) is a deep neural network [52]. The basic layer types of CNNs are fully connected hidden layers, convolutional layers and max-pooling layers [37]. Figure 2.7 illustrates an overview over the architecture of a convolutional neural network.

Convolutional layers apply a convolution on each pixel [11]. A convolution is a weighted sum between two functions [37]. The first function is the input signal and the second function a filter kernel [11]. Instead of each neuron being connected to all other neurons, they are connected to a relatively small local subset of the neurons in the previous layer, which allows the layer to recognize certain local patterns (for example: edges) [37]. The kernel is shared among the input neurons (parameter sharing) [11]. This has two main advantages: It introduces translation invariance, which means that it does not matter if a feature occurs at the center or near the edges of an image [37]. Also it reduces the number of learned parameters significantly [11]. Additionally to increase the capacity, multiple convolutions are applied per layer [37]. This is called the depth of a convolutional layer [37]. Due to their design, in contrast to fully connected layers, convolutional layers do not require a fixed input size, which means that they can be applied on images of different sizes [88]. The kernel is a quadratic matrix of size n [52]. To avoid the size reduction of standard convolutions, one can set the padding parameter of a convolutional layer [37]. This increases the input data synthetically (for example by introducing a border with constant value). There is also a stride, which defines how the filter moves along the

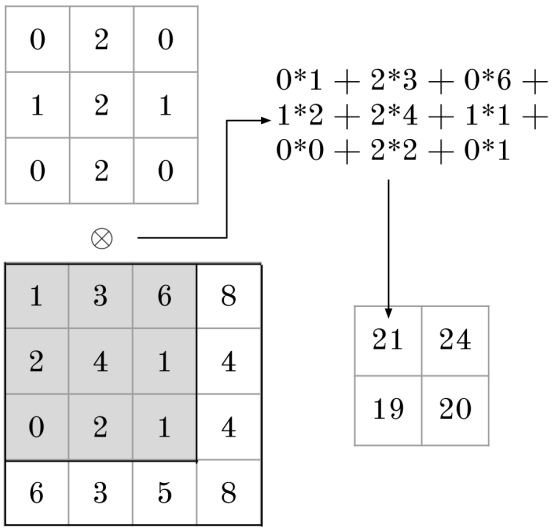


Figure 2.8: 3x3 convolution with stride=1 and no padding.

input [37]. If the stride is fractional ($1/s$), it is also called a deconvolution or transposed convolution [88]. This operation increases the output size [37].

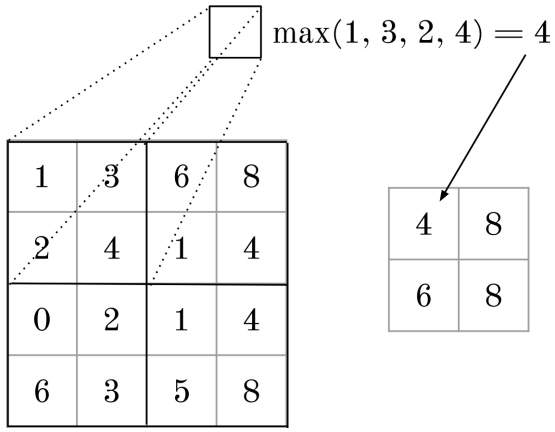


Figure 2.9: 2x2 max-pooling of a simple 4x4 input layer into a 2x2 output layer

Pooling layers are applied after convolutional layers [52]. Their goal is to reduce the dimensionality of the data (as well as sometimes translational invariance) [88]. Local pooling layers are connected to a local subset of the neurons in the previous layer [37]. The input is transformed by some function [88]. A typical pooling layer is max-pooling, where the result is the maximum value of all incoming connections of the current local subset (for example 2x2) [52]. Figure 2.9 illustrates a max-pooling operation. Another common type of pooling is average pooling, which is often combined with global pooling [56]. Global pooling means, that the entire data is pooled at once. This operation is

usually applied at the end of the convolutional part of a CNN [56]. Pooling does not involve learning weights, as the function applied is fixed before training [37]. There are also problems associated with pooling, as it essentially makes the model forget information [41]. This is one reason, why many modern convolutional neural network architectures mostly avoid max-pooling [41].

Fully connected layers are usually the last layers of a CNN, their task is to perform classification based on the extracted features of the previous layers [52]. Interestingly the convolutional part of a CNN can be used for feature extraction for different problem domains (see section 2.4) [101].

2.2.7 Recurrent Neural Networks

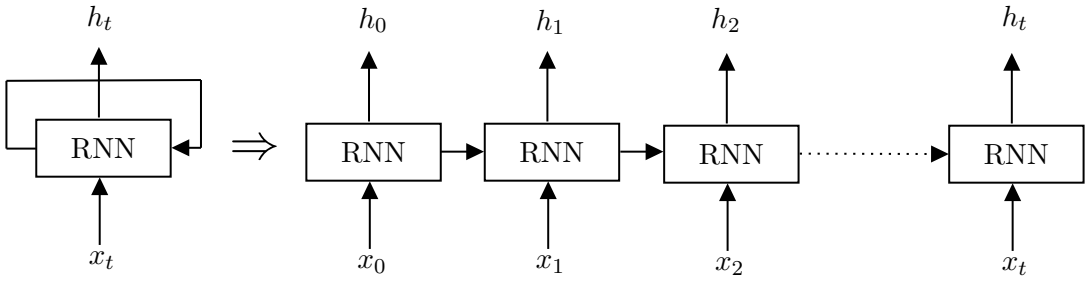


Figure 2.10: Unrolling of a simple RNN cell over time [79]

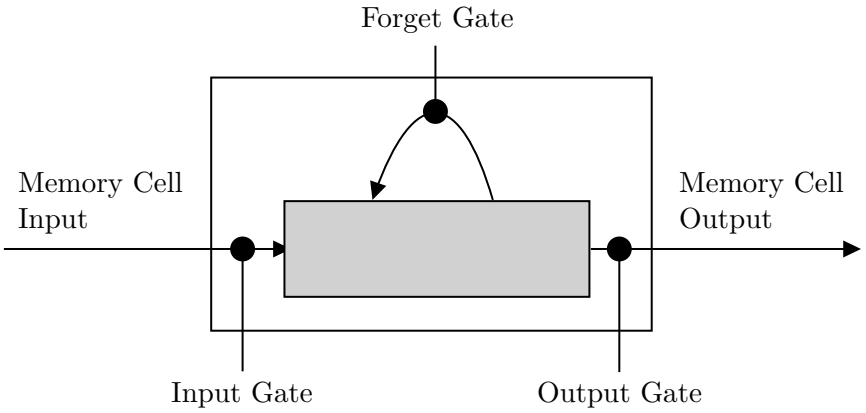


Figure 2.11: A schematic overview of an LSTM [44]

Recurrent neural networks (RNNs) are suited for sequences of arbitrary length, in contrast to normal fixed length input vectors for conventional feed forward neural networks [44]. Recurrent neural networks are recurrent in the sense that they contain cycles in the topology of the network [101]. In theory RNNs should be capable of understanding long-term dependencies [44]. In practice naive RNNs with simple loops are not suited for longer sequences, because of the vanishing/exploding gradient problem [44]. To

train a recurrent neural network it must be converted into an equivalent feed forward network [37]. Figure 2.10 shows a visualization of such unrolling. This process converts an RNN for a sequence of length n into a feed forward networks with n layers. During backpropagation we need to calculate the partial derivative of each weight w with respect to the loss function [44]. This partial derivative gets exponentially smaller with increasing number of layers, which causes the training to slow down significantly or even stop entirely [44].

Replacing the naive approach with Long Short-Term Memory (LSTM) cells has shown to be successful for many tasks [115][44][101]. As the architecture of LSTMs keep the vanishing gradient locked inside the cell, instead of propagating through the entire network [44]. LSTM networks are capable of learning long-term dependencies, which makes them very suited for problems in the domain of text and audio [115][107]. An LSTM cell is unrolled over time similar to an RNN cell, where for a sequence the LSTM cell is transformed into a multi layer feed forward network [79].

There are many different variations of LSTMs, but all share common building blocks [79]: Memory cell input & output, output gate and a forget gate [79]. The memory of an LSTM cell enables the network to remember information over time [44]. We will now examine a specific implementation of an LSTM cell [79].

The forget gate f_t removes information from the memory [79]:

$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] + b_f\right) \tag{2.25}$$

where W_f are the trained weights for the forget gate
$[.,.]$ is a concatenation of two vectors
h_{t-1} is the last LSTMs unit output
x_t is the current input
b_f is the learned bias for the forget gate

The σ ensures, that the result is between 0 and 1, where 0 means forget the state and 1 means keep the state.

The input gate i_t has a similar function to the forget gate, but instead it decides which information of the cell's memory needs to be updated [44]. Then the LSTM calculates a new memory vector $\widetilde{C_t}$ [79]:

$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] + b_i\right) \tag{2.26}$$

$$\widetilde{C_t} = \tanh\left(W_C \cdot [h_{t-1}, x_t] + b_C\right) \tag{2.27}$$

where W_i are the trained weights for the input gate
W_C are the trained weights for calculating the new memory
b_i is the learned bias for the input gate
b_C is the learned bias for calculating the new memory

Next, the forget gate and input gate are combined to calculate the new cell memory C_t [37].

$$C_t = f_t * C_{t-1} + i_t * \widetilde{C_t} \tag{2.28}$$

where C_t is the memory for the current LSTM unit
$*$ is element-wise multiplication

For the output gate o_t, we want to calculate the LSTMs output h_t, by combining the input and memory [79]:

$$o_t = \sigma\left(W_o \cdot [h_{t-1}, x_t] + b_o\right) \tag{2.29}$$

$$h_t = o_t * \tanh\left(C_t\right) \tag{2.30}$$

where W_o are the trained weights for the forget gate
b_o is the learned bias for the output gate

To increase the learning capacity of LSTMs we can apply multiple LSTM cells sequentially, where the output from the first LSTM is the input for the second LSTM, similar to how convolutional layers are stacked together [37]. Alternatively we can increase the learning capacity by increasing the size of the LSTM's memory C [37]. Empirically LSTMs have shown to have a tendency to overfit to the training data easily, which makes it even more important to find right hyperparameters [118].

A newer alternative to LSTMs are gated recurrent units (GRUs) [28]. Empirically GRUs have shown to train faster and are faster during inference, but in general no significant difference regarding performance has been observed [28]. Even more recently transformer architectures have shown to be also very successful at learning long-term dependencies [112]. Transformer architectures lack recurrent connections entirely [112].

2.2.8 Autoencoder

An autoencoder is a deep neural network which learns a compact representation of data without supervision [37]. This is similar to principal component analysis of traditional machine learning [11]. Application areas of autoencoders include feature extraction, image processing and anomaly detection [16][12][90][65]. An autoencoder consists of two parts:

- Encoder: Performs a dimensionality reduction from high dimensional data to a low dimensional representation [37]
- Decoder: Tries to reconstruct the original data from the low dimensional representation [37]

The intermediate low dimensional representation is also called *code* [37]. The intuition being, that this forces the neural network to extract more general features, which the autoencoder uses to reconstruct the original signal. Still, a big problem of autoencoders is their tendency to overfit and learn a trivial identity mapping [37]. There are several advanced autoencoder architectures, which try to mitigate this issue by applying some kind of regularization during training [37]. Denoising autoencoders get corrupted input data and their loss is still measured by how well it reconstructs the uncorrupted data [113]. The regularization of sparse autoencoders only activates a certain number of neurons per layer, which causes the network to especially react to unique statistical features of the dataset [72].

For example, an autoencoder architecture can be used to reduce the dimensionality of images while still preserving important semantic information [119]. Figure 2.12 shows a simple fully connected autoencoder. As CNNs perform very well in the image domain, they are a natural fit for autoencoders for images [52]. The encoders consists of standard convolutional blocks (convolution + max-pooling). While the decoder has to restore the original image, therefore it usually consists of layers performing transposed convolutions [37]. Transposed convolutions are used for up-sampling from a low resolution input to a high resolution output (refer to section 2.2.6) [88].

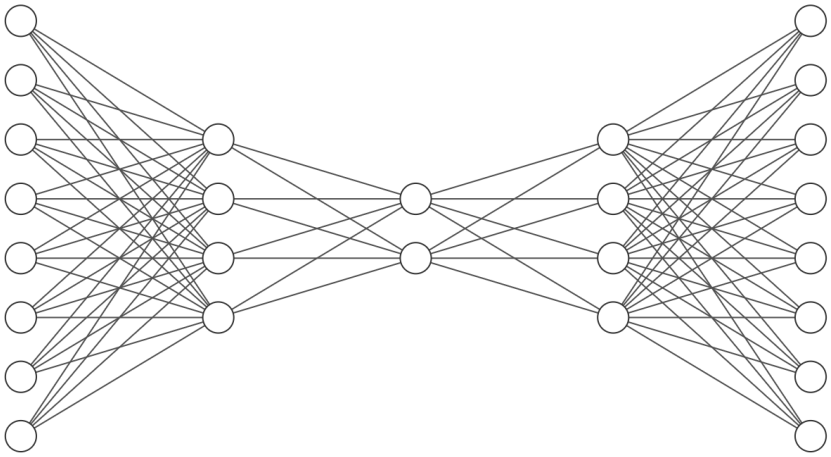


Figure 2.12: A fully connected autoencoder with 2 encoding and 2 decoding layers

2.3 Natural Language Processing

Natural Language Processing (NLP) combines the disciplines of artificial intelligence and linguistics [60][48]. NLP researchers try to build computer programs capable of understanding human language [48]. Common tasks of NLP include part-of-speech tagging, paraphrase identification, machine translation and speech recognition [86][115][107][116][54].

Early NLP tried to achieve its goal by building traditional parsers, similar to what compilers do for programming languages [48]. Unfortunately, these techniques are inadequate for human languages, due to the ambiguity of human language [60].

Another common approach has been to apply traditional machine learning for NLP [60]. The general approach is to extract certain features of a text corpus and use these features to train a classical machine learning model [55]. Empirically, the best performing machine learning algorithm for this context are support vector machines [55].

Deep learning currently achieves many state-of-the-art results in NLP [115][54]. An advantage of many text problems, is the abundance of large corpora, partly due to the rise of the internet [27]. This is an important advantage especially for unsupervised methods, since labelling of large corpora may be very expensive [85][27].

For both traditional machine learning and deep learning based natural language processing it is important to extract the right features from the dataset [60]. High level features (sentence length) are usually not sufficient, therefore the sentences must be converted

into more useful features [117].

A trivial approach would be to give each word a unique identification number [33]. This approach does not grasp important semantic information about the words [33]. For example, the words *king* and *queen* are more similar than *king* and *house*. If the featurization assigns the ID=2 for *king* and the ID=42 for *queen*, then the machine learning model has no additional information, that these words are similar concepts. Keeping this information is crucial for successful natural language processing [115][85]. Vector space models avoid this problem by representing each word in a high dimensional vector space, where a certain similarity measure holds [83].

There are different methods for applying natural language processing successfully. A great challenge is to entail semantic context into the trained models [60]. Because text is a sequence of words, we can apply recurrent neural networks in this context [57]. Especially recurrent neural networks which handle long-term dependencies reach state-of-the-art results in NLP (for more information, refer to section 2.2.7) [44]. Alternatively, a weaker form of context, is achieved by using a technique called n-grams [59]. Here we split the sentences into a set of tuples with size n, where each tuple contains the word itself and the next $n-1$ words of the sentence [59]. The intuition being, that the context of words can be represented by their surrounding words [59]. For example the 2-grams (also known as bigrams), for the sentence *I like machine learning* would be $\{(I, like), (like, machine), (machine, learning)\}$. This method allows applying bag-of-words techniques to also entail context [4].

Another way we can build a context-aware model is by applying hidden markov models (HMM) [11]. This is a statistical approach where we assume that sentences can be described by a markov process with latent (or hidden) variables [11]. HMMs are much simpler, compared to RNNs, as HMMs rely on common statistical techniques [60]. Given that the underlying process is actually a markov process, then they perform very well, otherwise other techniques may perform better [60].

Often the corpus available does not contain sufficient labelling. In such cases we can apply clustering methods, which search for patterns in the underlying data, without any additional labelling [97]. Gaussian mixture models are a very powerful clustering method, which assumes that the data is a mixture of multiple gaussians [97]. Through an algorithm called *expectation-maximization* we get a set of gaussian distributions, which allows us to get a probability for each sample that it originated from a specific gaussian, effectively a classification for this sample [11].

In the following subsections I will outline several techniques for extracting word embeddings from a sentence. Beginning with TFIDF, a technique which ranks words by the term frequency and inverse document frequency [53]. Then I continue with GloVe, an unsupervised statistical approach [83]. And finally I conclude this section with a state-of-the-art deep learning based word embedding called ELMo [85].

2.3.1 TFIDF

TFIDF stands for term frequency inverse document frequency and is a simple, but powerful vector space model [53]. The idea originated in information retrieval, where the goal is to weight words by their importance for a document and for a query [99]. The TFIDF consists of two terms multiplied: Term frequency and inverse document frequency. The term frequency (TF) describes the total number of times a word occurs in a document [77]. The idea here, is that the more often a word occurs in a given document, the more important it is in describing that document [77].

The problem with this is that certain words occur almost ever very frequently (stop words) [53]. For example, the word *the* has in almost every text corpus a very high term frequency, but is rarely important in describing the document. This is the reason why the inverse document frequency (IDF) is also considered [53]. This weight describes the importance of a word compared to all other documents [99]. For example, a document about the *Enigma machine* would have frequent mentions of the word *Enigma*, therefore the IDF would be relatively high compared to other words. In contrast, the word *the* occurs usually in all documents, therefore the IDF would be very low.

There exist many different formulas which entail this idea [53]. In this thesis the following TFIDF equations were used [53]:

$$\mathrm{idf}(t) = \log\left(\frac{n}{\mathrm{df}(t)}\right) + 1 \tag{2.31}$$

$$\mathrm{tfidf}(t, D) = \mathrm{tf}(t, D)idf(t) \tag{2.32}$$

where t is some term, D some document, $\mathrm{tf}(t, D)$ the number of occurrences of term t in document D and $\mathrm{df}(t)$ the number of documents containing term t.

The advantage of TFIDF is, that is relatively easy to understand and does not need any time consuming training, as it only counts word occurrences [77]. This makes it feasible to be trained locally, without the need of any pretrained model. The main disadvantage is that TFIDF does not capture the context of the word well. It is a bag-of-word approach which ignores the word order completely [77].

2.3.2 GloVe

The basis for training Global Vectors for Word Representation (GloVe) is a word to word matrix which denotes the frequency of words occurring together in the training data [84]. The intuition of this word embedding is that the ratios of word-word co-occurrence may encode the underlying meaning of the words [83]. During training of the model the objective is to learn word vectors such that the dot product equals to the logarithm of the probability of co-occurrence with another word [83]. GloVe is a bag-of-word approach, as it uses a word to word matrix without any information about sequence [83].

Probability and Ratio	$k = solid$	$k = gas$	$k = water$	$k = fashion$
$P(k\|ice)$	1.9×10^{-4}	6.6×10^{-5}	3.0×10^{-3}	1.7×10^{-5}
$P(k\|steam)$	2.2×10^{-5}	7.8×10^{-4}	2.2×10^{-3}	1.8×10^{-5}
$P(k\|ice)/P(k\|steam)$	8.9	8.5×10^{-2}	1.36	0.96

Table 2.2: Example for co-occurrence probabilities for words *ice* and *steam* given *solid*, *gas*, *water* and *fashion* [83]

Table 2.2 shows the calculated conditional co-occurrence probabilities for the words *ice* and *steam* given the words *solid*, *gas*, *water* and *fashion*. Expectedly *steam* co-occurs more frequently with *gas*, while *ice* co-occurs more frequently with *solid*. The unrelated word *fashion* co-occurs rarely with both *steam* and *ice*. The ratio of $P(k|ice)/P(k|steam)$ entails some basic semantic meaning: If the ratio is much greater than 1 than it correlates well with properties specific to *steam*. On the other hand, if it is significantly below 1 then the word has a high correlation with *gas*.

2.3.3 ELMo

The previously mentioned models have very limited sense of context, especially each word had the same word vector associated regardless of the context [83][53]. ELMo (Embeddings from Languages Models) assigns word embeddings depending on context [85].

The basis for ELMo is a language model trained on a large corpus of text [85]. A language model predicts the next word given a sequence of prior words [60]. For example given the sequence of words *Yesterday I was in New York. I visited the statue of ...*, a good language model could predict that the next word would be *Liberty*.

ELMo models language using a 2-layer bidirectional LSTM with residual connections, which allows training of deep models more efficiently. However the words still must be converted into a vector representation [85]. This is achieved by first converting the words into a character embedding and then a convolutional layer with max-pooling is applied [85]. The character embedding has the advantage that the model handles words better which are not in the vocabulary [85].

The novel idea of ELMo is not how it is trained, but how the trained language model is applied [85]: ELMo calculates the final word embedding for a word by combining the different hidden states of the LSTMs and the output of the convolutional part into a unified vector representation [85]. The combination is done by multiplying a weight to each state, which is also learned [85].

2.4 Transfer Learning

Usually, the more complex a given task is, the more labelled data deep learning needs to achieve good results [37]. This is not only expensive, but sometimes not possible. For example, Gary Larson has only drawn so many cartoons in his lifetime. Labelling a big dataset is also a big challenge and not always feasible.

Transfer learning tries to solve this problem: Instead of training a completely new model for each problem task, the idea is to reuse models and only slightly adapt them for new tasks [37]. The original model is trained for an already existing big dataset and the dataset of the fine tuning can be orders of magnitudes smaller [88]. This tuning in the context of deep neural networks is usually done by freezing the weights of neurons in certain higher layers during training [37]. Usually lower layers perform more general feature extraction, which can be transferred to other domains more easily [88].

For example there are already very big datasets for image classification tasks [52]. Training a good performing classifier on these datasets has already been done numerous times [41][52]. Especially CNNs have an architecture well suited for transfer learning: As mentioned in section 2.2.6, there is the convolutional part and the fully connected part. Usually the convolutional part is acting as a feature extractor and the fully connected part makes the final prediction. Empirically it has been shown that the feature extractor generalizes often very well to new domains. Depending on the similarity of the domains, less training is needed for the new domain. For example, this thesis uses a pretrained ResNet8 model which has very good performance on traditional image classification tasks and applies it on Gary Larson's cartoons [41].

Natural language processing works similarly by transferring word embeddings between different corpora [27]. Importantly, these word embeddings transfer well for a broad range of tasks and still achieving state-of-the-art performance [85].

2.5 AutoML

In machine learning models have increasingly more hyperparameters [52][41]. Traditionally these have been manually set by the machine learning practitioner, which does not scale very well for many hyperparameters [97]. Trivially grid search or random search can be applied. Grid search samples all possible combinations of hyperparameters [42]. Consider the following example: If a machine learning algorithm has four hyperparameters and each hyperparameter has six possible values, then 6^4 models have to be trained and evaluated. There are two significant problems with this approach: It scales exponentially with the number of hyperparameters and for continuous hyperparameters only a predefined subset of possible values can be tested [7]. Random search assigns random hyperparameters until some halting criterion is reached [7]. This technique performs very well for many hyperparameter optimization tasks, as it does not waste time on optimizing irrelevant hyperparameters [7].

Automated machine learning (AutoML) is the process of finding hyperparameters or even the entire machine learning pipeline automatically, without (or with much less) human intervention using machine learning itself [50]. An ideal AutoML system could perform the following tasks automatically [42]:

- Data pre-processing
- Feature engineering
- Feature selection
- Algorithm selection
- Hyperparameter selection

In practice completely autonomous AutoML it is not yet possible, as the possible search space is intractable by currently available computer hardware [42]. Nevertheless, AutoML still may save time and result in possibly better models, as we can focus on more important questions than hyperparameter tuning [42].

This thesis uses the automated machine learning algorithm implemented in the works of Komer et al. [50]. Their philosophy is that the selection of the entire machine learning pipeline (including data pre-processing) can be seen as a single large hyperparameter optimization problem [50].

Using Hyperopt-sklearn requires the machine learning practitioner to define the following parameters, also known as *hyper* hyperparameters (because they are parameters for training the hyperparameters) [50]:

- Objective function: Mapping of a configuration into a scalar value, which the algorithm tries to minimize [50]
- Search domain: Random variables whose distribution should match the most likely best configuration [50]
- Optimization algorithm: An algorithm which finds the best configuration in the search domain given the objective function [50]

Now that I covered the necessary foundations of machine learning, artificial neural networks, natural language processing, transfer learning and automated machine learning, next I will outline the context in which I applied them.

2.6 Computational Humor

According to the Oxford dictionary humor is the "the quality in something that makes it funny or amusing; the ability to laugh at things that are amusing" [3]. The exact reasons for why humans developed humor are still under debate. It seems that humor contributes to physical and psychological well-being [62][71]. Evolutionary psychologists have proposed that humor may have been a result of sexual selection: Women may have preferred men with humor, as it may have been a sign of intelligence, adaptability, and desire to please others [70].

Over the centuries, many theories of humor have been proposed:

- Superiority theory: One of the first theories of humor, which proposes that laughter expresses feelings of superiority over other people (or a former self) [71]. This theory was popular until the 20th century.
- Relief theory: Another early theory of humor, which proposes that humor can be seen as a valve which releases pressure [71]. This pressure was initially thought to be some kind of gas or energy, but later it was considered to be some form of *psychic energy* [70][71].
- Incongruity theory: This theory proposes that humor is the result of the perception of something incongruous [14]. In particular if the mind observes something which violates its mental model and expectations [14]. This is in line with many of Gary Larson's cartoons: The cartoon sets up some expectation, which is usually in contrast to what the reader would have expected. For example scientists behaving like toddlers in usually serious situations.
- A more modern view of humor is seeing it as a form of play [70]. It has been observed, that many animals learn important skills during play [70]. Humor seems to serve a similar purpose for human social behaviour [58]. Usually humor shows what should be avoided during communication or in other (social) situations [70]. Laughing and smiling enables these situations and therefore can be considered as a signal to play [58].

Computational humor combines the field of artificial intelligence with the research of humor [96]. As the interface between humans and computers becomes more natural, it is also important that computers can also understand humor [6]. The rise of chatbots and virtual assistants shows that humor is still a very difficult task for these applications [6]. An important challenge of computational humor is that humor is very subjective: There is no general humor, as every person finds different things funny [71]. This may be the reason why there is currently no unified computational model of humor. Instead models for certain subtasks of computational humor have been constructed [117][8].

Common tasks of computational humor involve generative or discriminative tasks [6][117]: An example for generative task would be joke generation or pun generation. In these

types of tasks the computer has to synthesize a joke or a pun. A discriminative task would be for example joke recognition. In this setting the computer should be able to detect whether some text is a joke or not. Computational humor and language understanding go hand in hand [117]: To really understand a joke, the interpreter (be it human or computer) has to understand the language. Proper language understanding is still an important research topic [85][27].

Many early approaches of computational humor have focused on the incongruity theory, where humor is an unexpected difference between the expected and the unexpected [10]. The general idea was to find or produce contradictions between what a human would expect and what has been communicated. These approaches had only limited success, most likely because the underlying language understanding was not sophisticated enough [10]. The computer did not have a proper model of what the human would understand and therefore could not apply the incongruity theory successfully.

Another problem of early computational humor models, was their heavy reliance on ontologies, such as WordNet and VerbNet [10][67]. These are human crafted databases with concepts and their relationships among each other. A challenge for these databases are the ambiguity of many concepts and also their limited size. For example VerbNet contains only around 4500 verbs, which results in a very limited understanding of the sentences [102].

2.7 Related Work

In this subsection I outline several works of the field of computational humor related to our work. I start discussing several traditional approaches, which do not use deep learning. Then I will continue with deep learning based works.

2.7.1 Computational Humor using Traditional Methods

In the following sections we will discuss several works in the field of computational humor that rely on traditional machine learning methods.

Humor Recognition and Humor Anchor Extraction

Yang et al. formalize humor as a binary classification task between humorous and non-humorous [117]. The authors perform their experiments on two datasets: Pun of the Day and One-liner dataset for positive examples. Samples for the negative class come from various other sources (Yahoo! Answers, New York Times, AP News and Proverb). The negative class samples are filtered, such that the sentences are approximately of the same length and contain the same words as the positive class.

Additionally their work also performs humor anchor extraction. This technique extracts the key words that enable the humor of a sentence. Removing these words would make the sentence not humorous anymore. The paper lists the example of the sentence *I used*

to be a watchmaker; it is a great job and I made my own hours with a sensible selection of anchors *watchmaker*, *made* and *hours*.

The authors extract several features from the input sentences that describe some aspect of humor.

- Incongruity structure: disconnection (maximum meaning distance of word pairs) and repetition (minimum meaning distance of word pairs)
- Ambiguity theory: sense combination (the higher the more possible interpretations a sentence has), sense farmost (largest path similarity of any sense of a word) and sense closest (smallest path similarity of any sense of a word)
- Interpersonal effect: negative and positive polarity (number of occurrences of negative or positive words) and weak and strong Subjectivity (number of occurrences of all weak or strong subjectivity oriented words)
- Phonetic style: alliteration (number and maximum length of alliteration chains) and rhyme (number and maximum length of rhyme chains)

The work describes the training of a classifier based on these features. Important for the humor anchor extraction is the fact that the features are word order independent and function also on partial sentences. Additionally to these human centric features the best performing model also uses Word2Vec features [66].

The procedure of determining humor anchors is surprisingly simple. As the feature can be computed on any (partial) sentence the algorithm can freely remove words and still get a reasonable prediction. Also, only certain word types are considered (verbs, nouns, adverbs and adjectives). Algorithm 2.5 describes this idea on a high level.

Algorithm 2.5: Humor anchor extraction

```
1 Take only verbs, nouns and adverbs of original sentence
2 Calculate humor score
3 foreach possible anchor subset of sentence do
4 |   Calculate humor score of subset
5 end
6 Return subset with maximum decrement
```

From a performance point of view the humor recognition seems to work well. On the Pun of the Day dataset the best model reaches an F1 score of 0.859. The One Liners dataset reaches an F1 score of 0.805.

Humor anchor extraction also performs quite well. The authors perform a quantitative evaluation by comparing 200 random samples that were annotated by three annotators. The strict variant, where the anchors must be exact reached an F1 score of 0.438 and

an F1 score of 0.756 for the relaxed version where at least one word had to match on the Pun of the Day dataset. For the One Liners dataset the performance was worse: The strict version reaches an F1 score of 0.288 and an F1 score of 0.616 for the relaxed version.

Humorist Bot: Bringing Computational Humour in a Chat-Bot System

Augello et al. implement humor detection and joke generation in the context of a chatbot [6]. A chatbot is a software agent able to hold a conversation with a human. Humorist bot is able to generate humorous anecdotes and react to the users humorous messages by adjusting its avatar. A funny message not only affects the answer of the bot, but also changes its avatar to a laughing avatar.

This chatbot uses a rather traditional knowledge base with different answer sets:

- Set for general conversations
- Set for humorous sentence generation
- Set for humorous sentence recognition

AIML knowledge base is an XML format which describes a set of possible answers that are matched using pattern matching.

```
<category>
    <pattern>How are you?</pattern>
    <template>I am fine. How about you?</template>
</category>
```

The authors implemented humor detection by focusing on the following aspects of humor:

- Alliteration: based on phonetic transcription
- Antinomy: based on the antinomy relationships of WordNet
- Adult slang: based on a set of words classified as adult slang

They do not apply any machine learning techniques, but instead implement handcrafted rules. The authors implemented joke generation by combining several AIML rules.

Evaluation shows that recognition of humor achieves an accuracy of 66% for 100 randomly selected humorous samples, while a sample of 100 non-humorous sentences contained 19 false-positives. The authors did not perform an evaluation of the joke generation.

2.7.2 Computational Humor using Deep Learning

In the following subsections we will discuss the application of deep learning in the domain of computational humor.

Deep Learning of Audio and Language Features for Humor Prediction

The work of Bertero et al. compares several machine learning techniques for predicting and detecting humor in dialogues [8]. They use a dialogue dataset from the series *Big Bang Theory*, with annotations of canned laughter. Their goal is to predict these punchlines and where the canned laughter would appear. This work compares several techniques against each other: Convolutional neural networks, recurrent neural networks and conditional random fields. The convolutional neural network use low level representations of words and acoustic frames, while the other two methods use a set of acoustic and language features:

- Acoustic features: extraction of pitch, line spectral frequencies, intensity, loudness, probability of voicing, F_0 envelope, zero-crossing rate, MFCC and speaking rate features
- Language features: extraction of lexical features (unigrams, bigrams and trigrams appearing at least five times), structural and syntactic features (proportions of verbs, nouns, adjectives and adverbs, sentence length and average word length), sentiment (average of positive/negative sentiment scores of SentiWordNet), antonyms (presence of verb, noun, adjective and adverbs antonyms from WordNet compared to the previous utterance), speaker turns (speaker identity) and word2vec features [67][66]

Interestingly, their work does not use an LSTM/GRU cell, but instead a simple RNN cell, which often have the problem of vanishing/exploding gradients [44]. Their RNN architecture applies an embedding layer on the n-gram data and a separate embedding layer on the acoustic and language data for dimensionality reduction. Afterwards they combine both vectors and use this as an input for the RNN cell.

The CNN architecture uses two separate CNN branches for the sentence inputs and for the audio inputs. They use one convolution layer, with a sliding window of five tokens for the sentence input and a sliding window of three frames for the audio input. Both branches have an embedding layer applied that reduces the dimensionality. After the convolutional layer, the authors apply a standard max-pooling layer. Finally their implementation combines both CNN branches and global features into a single feature vector using a softmax layer.

Their corpus is split into a train/validation/test split with 35865 utterances in the train set, 3904 utterances in the validation set and 3903 utterances in the test set. The corpus has in total 42.8% punchlines.

The CNN shows to be the best performing approach to this task, as it reaches an F1-score of 68.5%. RNNs performance is relatively low, which may be due to overfitting. Applying an LSTM instead of a simple RNN could potentially learn much more long-term dependencies. Next, I will discuss the authors follow-up work, which uses LSTMs on the same Big Bang Theory dataset.

A long short-term memory framework for predicting humor in dialogues

Bertero et al. apply LSTMs on the same Big Bang Theory dialogue dataset as in their previous work [8][9]. Instead of a simple RNN they use a more sophisticated LSTM architecture. Additionally they combine the LSTM with a CNN.

Their feature vector is a combination of the following:

- One-hot encoded word tokens: Models the likeliness of a word of being humorous
- Character trigrams: Reduces impact of word stems
- Word2Vec: Models similarity of words and their semantic meaning [66]

First their architecture converts the high dimensional vectors into a lower dimensional representation using an embedding layer. Then they apply a convolutional layer and perform max-pooling afterwards on each feature. Furthermore they combine these features into a single feature vector and add a final sentence encoding layer, which combines the different features. Additionally, similarly to their previous work, the authors also extract high level features (structural features, POS proportions, etc.).

The LSTM in combination with high level features achieves the best performance. The F1-score of this approach is 62.9%. This seems to be a worse result than the best F1-score of 68.5% in the previous work, but the key difference is, that the best result of their previous work relied on acoustic features and this LSTM approach is only text-based.

Humor recognition using deep learning

The work of Chen et al use several corpora from various sources [17]: Pun of the Day, Short Jokes dataset, 16000 One-Liners and PTT Jokes each with a binary annotation, where "1" means the sample is humorous and "0" means it is not humorous. The authors take negative samples from news headlines.

The basis for their architecture is a convolutional neural network. First they convert the words using pre-trained GloVe embeddings. The authors use GloVe trained on Wikipedia 2014 + Gigaword 5 with a vocabulary containing 400,000 words [83]. Then they apply a convolutional layer and try filters of size 3 to 20 for the convolution. Finally they apply a regular max-pooling after the convolutional layer.

In order to improve performance, the authors apply *highway layers*. Highway layers introduce shortcut connections with gate functions, which allows the information to

flow more easily through the deeper parts of the network. Additionally they also apply dropout layers.

The authors perform a 10 fold cross validation. A CNN with highway layers and a large filter size of 100 performs the best on 16000 One Liners and Pun of the Day with an F1-score of 90% for both datasets. Short jokes even reaches 92% and PTT Jokes 94%.

Interesting about this approach is especially that the authors achieved these result without feature selection. The deep learning model can generalize humor in the given context. The chosen experiment could probably even be improved by using new state-of-the-art word embeddings such as ELMo or BERT [85][27].

Given the presented papers we have a broad overview over the state-of-the-art in computational humor using deep learning and traditional machine learning. No work tried to model the humor of a single person, and instead opted for a generalized approach. Mostly using crowd annotated datasets or using humorous content which many people find funny (for example the series Big Bang Theory). This shows that the approach this thesis takes, where I try to model the humor of a single person, is novel and could reveal new insights into computational humor.

CHAPTER 3

Design

In this chapter I outline the design process of this thesis. The goal was to construct deep neural network architectures, which understand Gary Larson's cartoons. This process had four main phases: Dataset design, visual model design, text model design and combined model design. The section dataset design describes how the dataset was created and provides a detailed look into the dataset itself. Visual model design focuses on the cartoons, while text model design focuses on the punchlines. Finally, I discuss the design of a combination of both models in subsection 3.4.1

3.1 Dataset design

Due to the novel requirements of this approach I created a new dataset from Gary Larson's cartoons. Then I performed an extensive data analysis, which motivated the remaining design process.

3.1.1 Dataset acquisition

The dataset consists of 2487 cartoons. Each sample consists of a cartoon image, punchline text and a funniness from 1 to 7. The funniness is ordinal, where 1 means not funny at all and 7 is hilarious.

Annotation was performed by me (Robert Fischer) over the course of half a year. The annotation was split in two steps. The first step was preparing the cartoons. In this step the cartoons were cropped and rotated accordingly. The punchline was transcribed from the image into text format, using an OCR system. Nearly every cartoon's punchline needed additional manual corrections. Furthermore, cartoons with bad quality, as well as duplicates were removed.

The second step was to annotate the funniness of the cartoons. I, as the annotator, experienced a problem I referred to as *humor fatigue*: After annotating cartoons for too

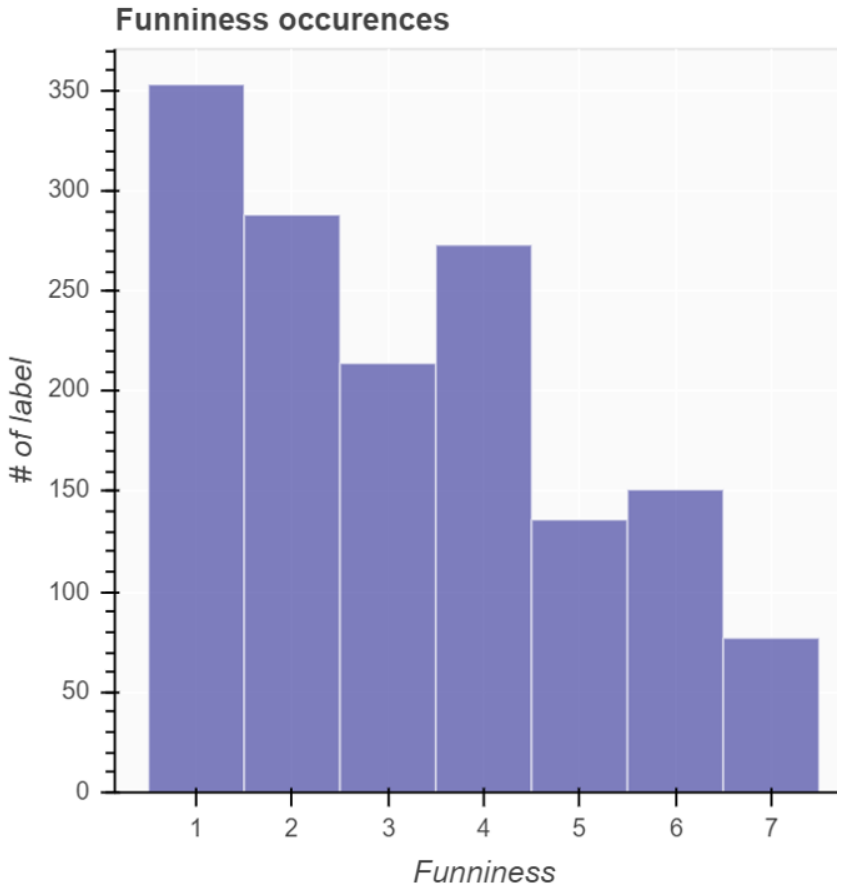


Figure 3.1: The label distribution of the training set

long, the annotations would get unreliable and could potentially lead to wrong labels. This issue was mitigated by limiting the duration of the annotation sessions, where each annotation session lasted 30 minutes at maximum, but shorter sessions were preferred.

3.1.2 Dataset analysis

The dataset analysis revealed several interesting facts about the dataset. Figure 3.1 shows a bar plot of the label distribution. Unexpectedly the funniness is not uniformly distributed. Over 14% of the cartoons were deemed to be not funny at all, while only 2% of cartoons were deemed to be very hilarious.

For verification that the distribution of the annotations has not changed significantly over the course of annotation, please refer to figure 3.2. Each cartoon has an ascending identification number. Cartoons were combined into buckets by this number. Finally, for each of this bucket a box plot is created. This plot shows no significant change of label distribution over the course of annotation.

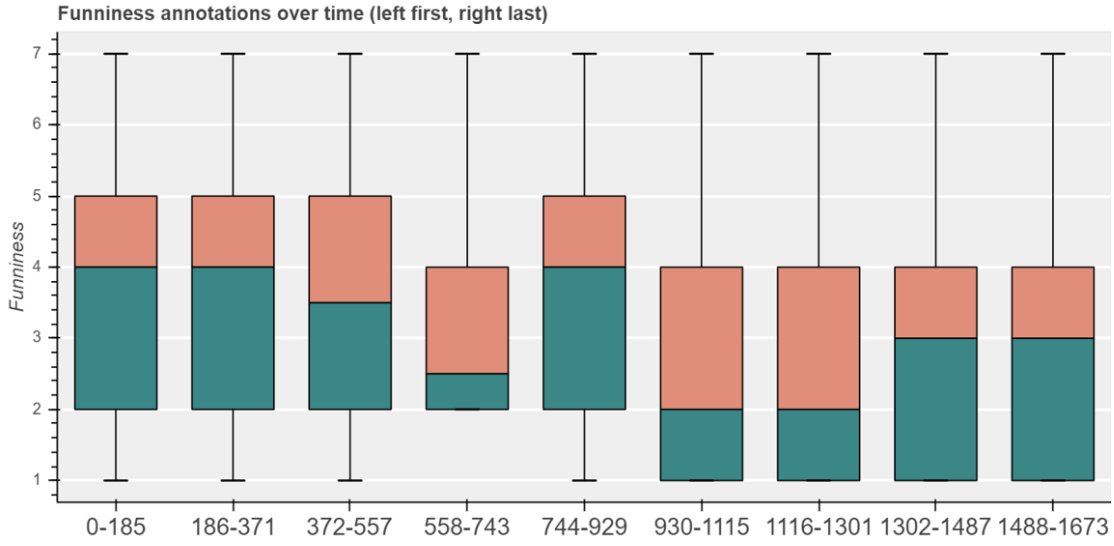


Figure 3.2: Box plots of the average funniness over time in the training split.

A word count analysis showed that there are significant differences in the frequency of certain words. Figures 3.3 and 3.4 illustrate bar plots of the top 10 most occurring words per funniness class. These words seem to contain certain connections to a cartoon theme. For example, one of the most frequent words for funniness class 7 is *Thag* which is a common name for cartoons set in the stone age. This could motivate a machine learning model which detects such words in a punchline or such themes in the cartoon to predict a potentially higher funniness.

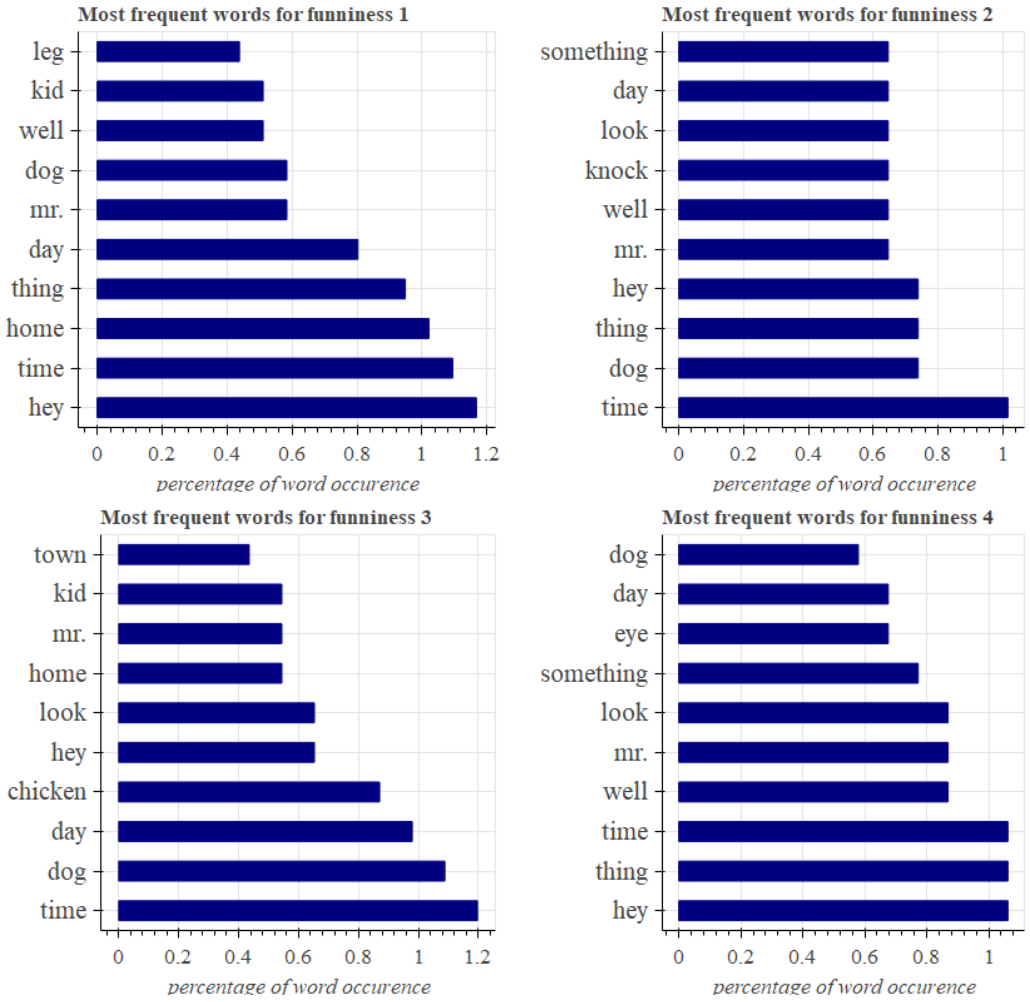


Figure 3.3: Most frequent nouns per funniness class 1 (not funny), 2, 3 and 4.

3.2 Visual domain model design

To reduce the problem space I decided first to try the visual domain of Gary Larson's cartoons. This has the advantage that there are well defined neural network architectures which knowingly perform well on various image related tasks [41][52].

3.2.1 Simple CNN

The goal of this approach was to establish a performance baseline, by training a relatively simple convolutional neural network. Figure 3.5 illustrates the best performing architecture of this approach. It consists of two main CNN blocks (with a 3x3 kernel and a depth of 100). The second CNN block also contains a batch normalization layer. The classification is performed by a single fully connected layer. This is the result of training

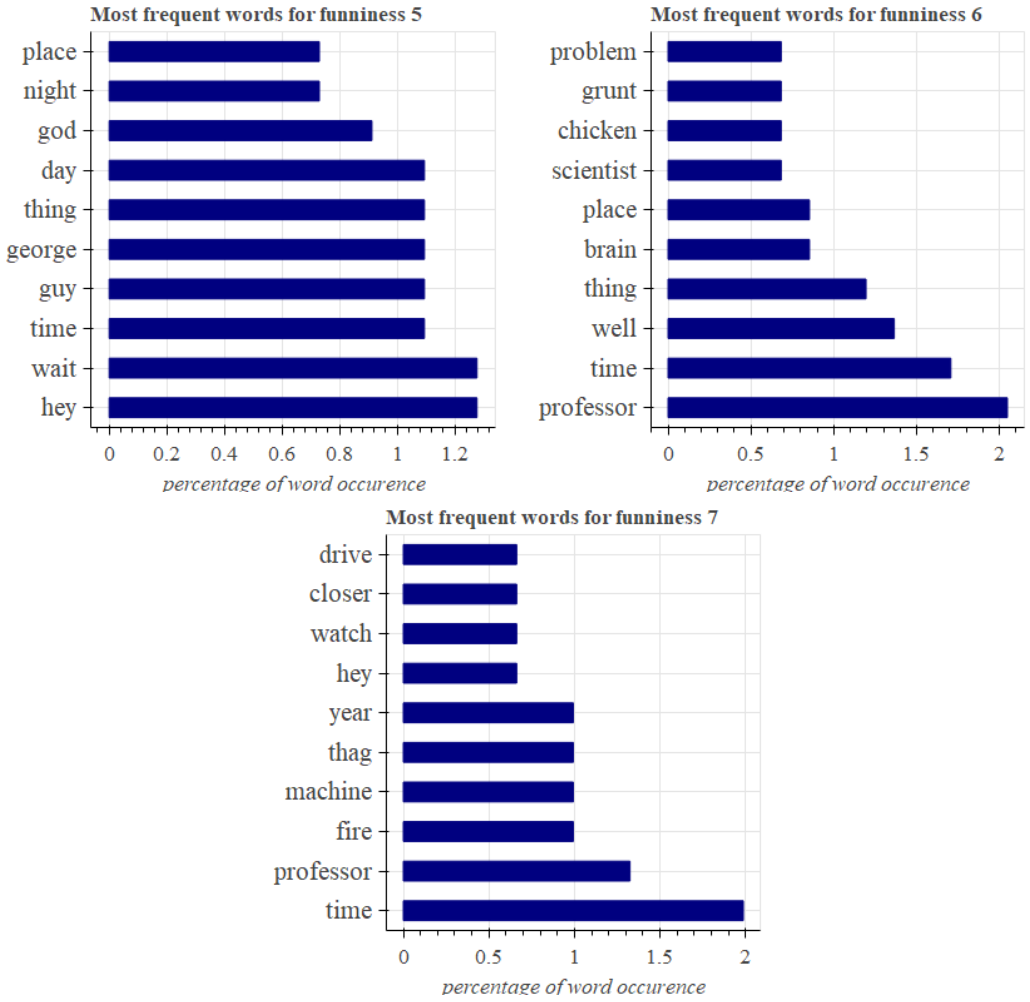


Figure 3.4: Most frequent nouns per funniness class 5, 6 and 7 (hilarious).

several configurations. The number of CNN blocks I trained ranged from one to four, with kernels ranging from 3x3, 5x5 and 7x7. The depth hyperparameter ranged from 10, 50, 100 and 200. Important to note, as no architecture had significantly better performance than the baselines, the differences between different hyperparameter configurations was minor. In total roughly 50 architectures were trained and evaluated using the validation set.

An important design decision a machine learning practitioner has to make is the input data format and layout. For images the number of input channels and the dimensions of the image are most relevant. Because the cartoons are grayscale the number of channels can be set to 1 without any loss of information. The input size is more difficult to define: If the input size is too large, the neural network has too much information to process and cannot find discriminative patterns. A fitting analogy would be to play the game *Where*

is Waldo on an image of the size of a football field. On such image, it would for humans be very difficult to find the right pattern (in this case Waldo). Otherwise, if the input size is too small, then crucial information may be unintentionally removed. Resizing an image can be seen as a crude form of dimensionality reduction, as the dimensionality of the data is reduced. Another important consideration is the orientation of images. Data analysis showed that 90% of the images are in portrait mode, while the remaining are in landscape. Therefore, predicting the images in portrait well has priority and the input orientation is in portrait. Different configurations for the input size were empirically probed and validated using the validation set. This includes the average size of cartoons, average size of landscape cartoons, several downscaled image sizes. The final best input size was 141x174 px^2.

Modelling the output of a neural network is equally important. I performed two experiments: One where the network was modelled in a classification setting where the final neurons returned a one hot encoding of the funniness. For classification I used cross entropy loss [26]. Another experiment was modelled as a regression task, where the final neuron returned the funniness directly. I tested both L1 loss (mean absolute error) and L2 loss (mean squared error) for regression [46]. The regression neural networks did not even approximate the baselines on the validation set, which is the reason why I did not pursue this approach any further.

Eventually, another important hyperparameter of CNNs is the number of conv layers and number of neurons per layer. As the dataset is relatively small in deep learning terms, it was not possible to train too many layers. Conversely, too few layers would mean that the network does not have sufficient capacity to learn the necessary concepts of humor. The final network consists of two convolutional blocks and one fully connected dense layer at the end. Their depth is set to 100.

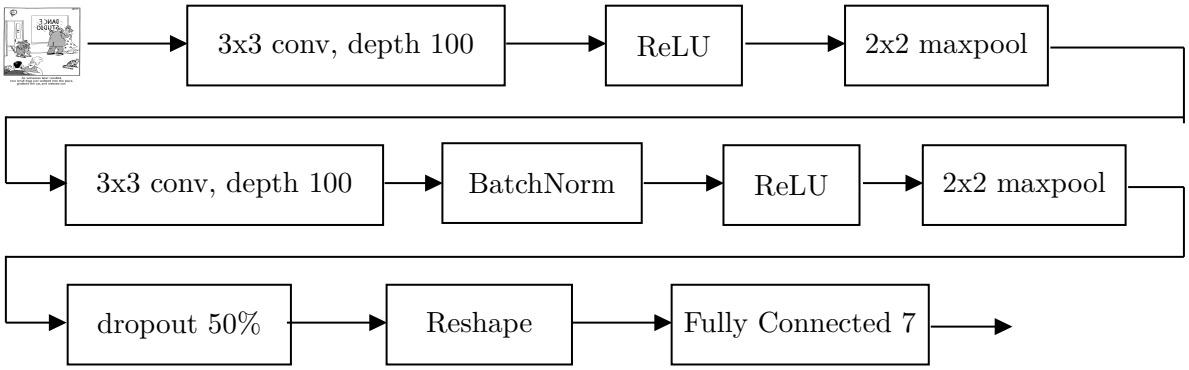


Figure 3.5: The simple CNN architecture

3.2.2 Transfer Learning CNN

As described in section 2.4 transfer learning is a common approach to deal with a small dataset in deep learning. This experiment uses a pretrained ResNet18 model [41]. ResNet

consists of multiple residual blocks, which allows the network to be significantly deeper compared to a naive CNN and still improve performance. Earlier blocks perform simple feature extraction and later blocks perform high level feature extraction [41].

I probed several configurations of this transfer learning approach:

1. Reuse entire convolutional feature extractor and retrain final classifier
2. Retrain last block of ResNet18 and the final classifier
3. Reuse entire model and add a new final fully connected dense layer

The second approach was the best performing on the validation set. Additionally applying data augmentation was crucial. Applying random horizontal flip, random resize and crop and slight random rotations also improved performance on the validation set.

3.2.3 Transfer Learned Object Detection

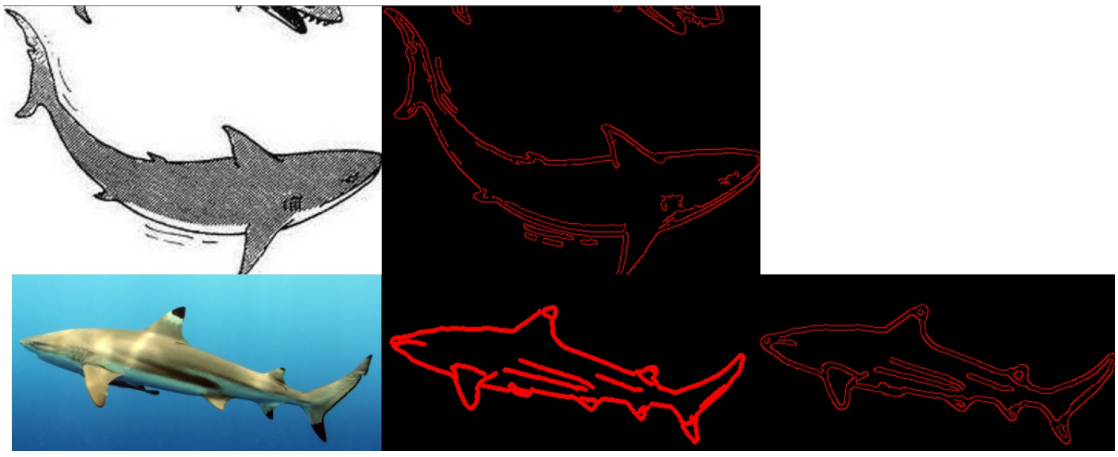


Figure 3.6: First row is a shark drawn by Gary Larson. Second row is a real world shark. The images of the second and third column were obtained by applying Canny edge detection.

There were two suspected issues with the previous transfer learning approaches:

1. The domain of real world images (which ResNet18 was trained on) and cartoons may be too different for a naive transfer learning approach. I addressed this issue by implementing a novel data augmentation technique.
2. The semantic level of humor may be too high for CNNs to understand. I planned to address this issue by extracting a high-level feature vector based on object detection using CNNs.

As witnesses later recalled,
two small dogs just waltzed into the place,
grabbed the cat, and waltzed out.

Has Human?	Has Dogs?	Has Cats?	Has Alien?	Has Shark?	...
1	1	1	0	0	...

Table 3.1: Hypothetical feature vector

The data augmentation (point 1) works by applying the Canny edge detection filter multiple times (figure 3.6) [13]. The cartoon image is transformed by Canny (using default parameters of OpenCV [13]) into the target domain. The real world image is transformed by applying Canny twice. The first application has a stroke size similar to the average size of Gary Larson strokes. The second application of Canny, with the same parameters, should bring the real world images into the same target domain.

To improve the problem of high-level semantics (point 2) I planned to train an object detection model based on the CIFAR-100 dataset augmented using the double edge detection data augmentation technique [51]. This would allow to compute a high-level description of a cartoon based on the objects detected by the object detection algorithm. To validate this approach I first trained a simpler image classification model based on ResNet18. If image classification does not perform sufficiently, then object detection will most likely also fail. Such a feature vector is outlined in table 3.1.

Unfortunately, the image classification did not achieve significant results, as there were no consistent classifications. Therefore, I did not train the final object detection. If, for example, a person was consistently assigned the same class, then the feature vector could have entailed enough semantic information for cartoon theme-based predictions. The data analysis revealed that there are certain cartoon themes which have higher funniness scores compared to others.

Another approach I tried was training an object detection model based on other datasets: The *TU Berlin sketch* dataset and *Googles Quick, draw!* dataset initially seemed to be just similar enough for Gary Larson's drawing style for transfer learning to succeed [31][1]. These approaches did not need the edge detection preprocessing, as they were already in the sketch domain. Still, the images were too distinct, as the image classification performance never reached significant performance.

3.3 Text domain model design

The following architectures concentrated on the text domain without any additional visual information. I began with an LSTM and a sentence embedding-based approach and then continued with probing an AutoML-based machine learning model.

3.3.1 ELMo

First I tried to model the punchlines using an LSTM/GRU neural network architecture. This approach quickly overfitted. I probed several configurations with multiple LSTM or GRU cells sequentially (one or two), an additional trained embedding layer and different memory vector sizes. Figure 3.7 shows the final architecture in detail.

Recurrent neural networks require big datasets, which could explain the bad performance of the LSTM based approach. Therefore removing the LSTM cell had the potential to improve performance. I achieved this by generating a *sentence embedding*. A sentence embedding is a vector representation of a sentence in a vector space. There are many different ways to obtain such an embedding. I chose the simple approach to take the mean of all noun word embeddings of a punchline. The intuition is that nouns entail most context and information in a sentence. This resulted in a fixed sized vector representation of a punchline, which allowed the use of a regular fully connected neural network.

I chose ELMo as the main context-aware word embedding for this thesis [85]. Mainly due to the state-of-the-art results it achieves while still being relatively easy to use in the chosen NLP framework AllenNLP [35]. An issue of such pretrained word embeddings is their lack of domain specific words. For example, the name *Thag* is not included in the vocabulary of ELMo, but seems to have an important contextual meaning, as the data analysis revealed.

Figure 3.8 illustrates the architecture which uses sentence embeddings. It consists of three convolutional blocks with increasing depth and batch normalization, followed by an average pooling and a dropout to counteract overfitting. Finally the prediction is performed by a fully connected layer. Because this approach performed better on the validation set I chose this approach for a more detailed evaluation in chapter 5.

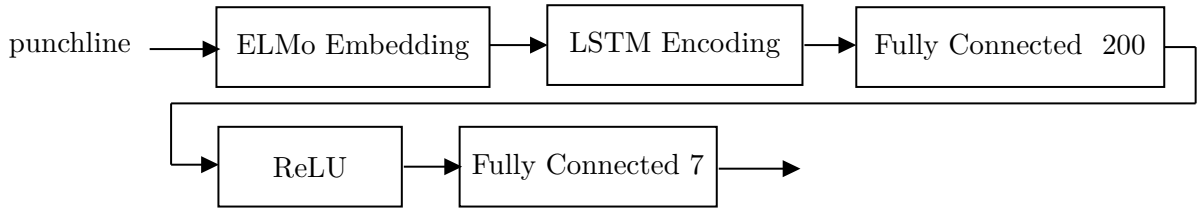


Figure 3.7: The LSTM architecture with ELMo word embeddings

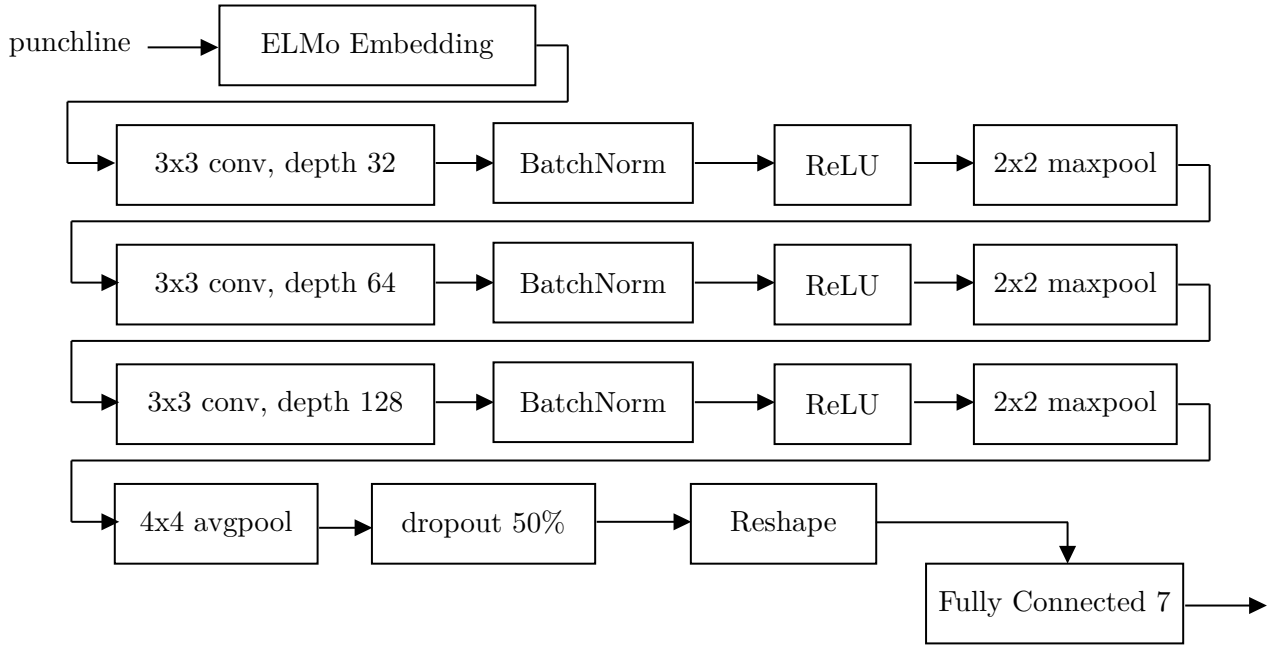


Figure 3.8: The architecture of the ELMo based model without LSTM

3.3.2 AutoML

Using AutoML my goal was to validate whether an automated approach might find a good model. Because the computer can optimize many more hyperparameters than the human experimenter, I hypothesized that it might perform better than the manually trained models. For example, it could be that some other classifier (e.g. SVM) might be better suited for the task at hand.

The selected AutoML library has only limited support for text input feature vectors [2]. So I had to convert the words to fixed length numerical feature vectors. I used TFIDF, ELMo and GloVe, as well as combinations of different word embeddings. This was applied in order to counteract the weaknesses of GloVe/ELMo. TFIDF does not have the same problem of GloVe/ELMo of missing words in the vocabulary, because TFIDF was entirely trained on the available dataset. I probed the following feature vectors:

- TFIDF
- ELMo sentence embeddings
- GloVe sentence embeddings
- TFIDF + GloVe mean
- TFIDF + ELMo mean

The best performing approach on the validation set was the model which only uses the ELMo sentence embeddings.

3.4 Visual and text domain combined model design

Finally I combined the findings of the previous approaches in the final two stage model, which combines the visual and the text domain into a unified neural network architecture.

3.4.1 Two Stage Model

With the two stage models I tried to solve two issues: The issue of class imbalance in the dataset and the small dataset size. Figure 3.1 illustrates this problem. Analysis revealed that previous models (simple CNN, transfer learning CNN) learn to predict the most frequent class (figure 5.1 and 5.2).

The two-stage approach tries to mitigate these two problems by applying a cascade of multiple individually trained classifiers and finally combining them in an ensemble model. I trained a binary classifier for each funniness class which detects whether a given sample belongs to its class or not. This *first stage* results in seven independent classifiers. The *second stage* consists of a regressor which predicts the final funniness based on the last hidden layer of each binary classifier. A similar second stage, but returning a classification (one hot encoding) was also probed, but did not perform well on any metric.

Figure 3.9 shows an illustration of the first stage. It handles both the visual and text domain: For the cartoons, first the autoencoder is applied, then two convolutional blocks (5x5 and then 3x3 kernel size) with batch normalization and a depth of 8 and 16 respectively. Afterwards the input is flattened and reshaped into a vector and further processed using a simple fully connected layer. For the punchline, first two sentence embeddings are calculated (ELMo and GloVe), combined into a single vector, and finally reduced using a fully connected layer, ReLU function and batch normalization. The resulting feature vectors of both the punchline and cartoon data are combined using again a fully connected layer, ReLU and batch normalization. Eventually, since I am training an independent classifier for each level of funniness we need to make a prediction, which returns whether the given sample belongs to the class of the classifier or not. Therefore a final fully connected layer, ReLU and batch normalization returns the binary classification.

For the second stage (see figure 3.10) we need to combine the predictions of each trained model. Additionally the network performs some additional transformations (dropout, fully connected, batch normalization and ReLU) per feature extractor. Afterwards the network combines the results into a single vector and applies the same sequence of operations two more times.

I tested three different configurations of this approach:

1. Only punchlines: I trained the binary classifiers on the punchline sentence embedding (ELMo and GloVe sentence embeddings).
2. Punchlines and cartoons: This approach uses both the punchlines and the cartoon images.
3. Punchlines and autoencoded cartoons: Same as the previously mentioned approach, but preceded by applying the encoder of a deep convolutional autoencoder.

This modular two stage design has the advantage of requiring less training data. This is due to the individual neural networks being quite shallow (relatively small number of layers). Therefore the gradient does not vanish and the signal propagates through the entire network easily [81]. Also, the approach allows to combine both the visual and the text domain, as only the first stage needs to be adjusted accordingly.

To further reduce the impact of class imbalance I applied oversampling. This means that more infrequent classes are sampled more frequently, so that for the binary classifier positive classes are as frequent as the negative classes. Additionally, I tried penalizing false positives in the neural network's loss function, but this strategy did not improve the results.

In order to further reduce the number of parameters of the neural network, I chose to reduce the size of the cartoon images. However, resizing loses crucial semantic information. To solve this problem I trained a deep convolutional autoencoder separately. The expectation was that the encoder results in more semantic meaningful representations of the cartoons. Unfortunately this was not the case, as validation revealed no significant increase in performance.

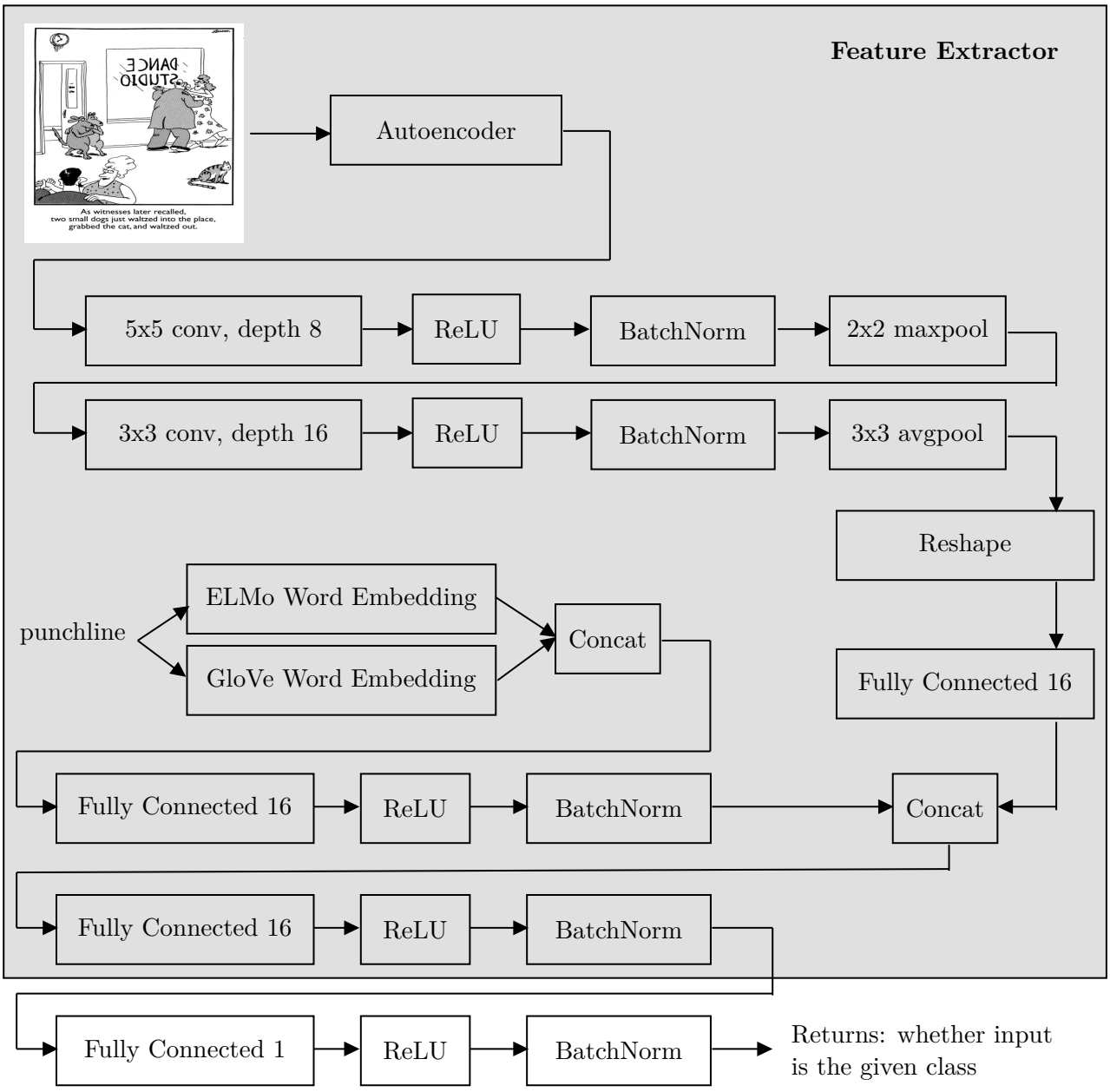


Figure 3.9: The first stage's architecture. The gray part is used for feature extraction in the second stage.

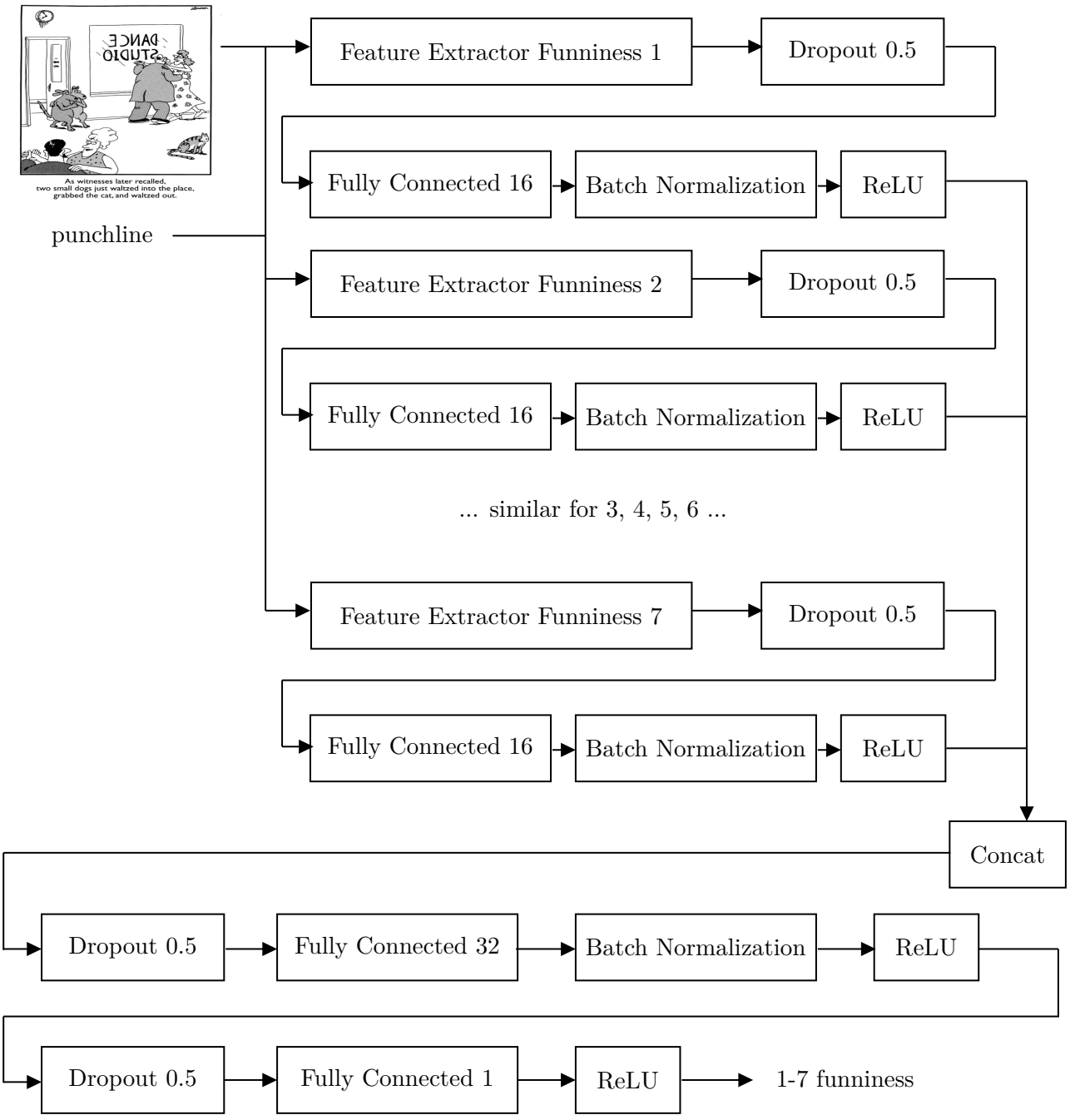


Figure 3.10: The second stage combines the previously trained models and returns a funniness prediction.

CHAPTER 4

Implementation

In the following subsections I describe the implementation procedure of this thesis, beginning with the technical foundation, which outlines the libraries and frameworks in use, then continuing with the special labelling tool implemented for annotating the dataset and then continuing with the debugging of deep learning models. This chapter finishes with important aspects of the architecture.

4.1 Technical Foundation

For this thesis I applied several frameworks and libraries for deep learning, data handling and other machine learning tasks. Python is one of the most widely used programming languages for deep learning, as most major deep learning frameworks (PyTorch, Tensorflow [92][109]) are available in Python [91]. In this thesis I chose to use *PyTorch*, which is a very common deep learning framework [92]. For data handling I applied *pandas* and *NumPy* which both allow efficient handling of large datasets in Python [80][75]. For more general machine learning tasks (e.g. oversampling and error calculation) *scikit-learn* proved to be very useful [104]. I also trained automated machine learning models using *hyperopt-sklearn* [50]. Hyperopt-sklearn plays very well with scikit-learn and covers many different machine learning algorithms.

For the punchline-based models *AllenNLP* was the best suited framework, as it is based on PyTorch and features easy to use word embeddings (such as ELMo [85]) [35]. It also has a very flexible JSON configuration system which makes it easy to try out different hyperparameters without changing the source code.

An important aspect of deep learning is reproducibility. It is generally difficult to make sure that experiments can be reproduced, due to the non-deterministic nature of mini-batching and some optimizations performed by deep learning frameworks. Fortunately, PyTorch has a deterministic mode, which disables these optimizations. Minibatching can

be deterministic by setting the initial random seed explicitly. Additionally, all relevant development milestones are recorded in a source code control system, which contains all necessary files to redo this experiment. Remark: Many machine learning practitioners use a notebook interface for machine learning. For example, Jupyter Notebook is currently widely used [47]. Early on, I decided against using this methodology, because of issues with reproducibility and less powerful debugging functionality.

4.2 Labelling tool

Creating a good Gary Larson cartoon dataset with reliable funniness labels was very important to train the ML models successfully. A great challenge was the fact the the entire labelling had to be performed by a single person, as the goal was to model the humor of this person. These requirements meant that a dedicated labelling tool was the best way to ensure the quality of the dataset, and also save time. Figure 4.1 shows the interface of the application.

It was possible to perform the following tasks using the labelling tool:

- Filter out cartoons with low visual quality.
- Automated cartoon processing and punchline extraction.
- Manual cartoon processing allowed cropping the images and manual punchline correction in cases where the automated system failed (which happened with almost all cartoons to some degree).
- Labelling the funniness of a cartoon.
- Cartoon object bounding box annotation allowed to annotate the bounding boxes and classes of objects seen in cartoons.
- Annotating the theme of a cartoon (this feature was not used).
- Automated duplicate detection, based on image and punchline data.
- Extended search and ordering allowed finding cartoons by punchline and ID.

Whether the tool has actually saved time is difficult to say for sure in hindsight, but considering how helpful the tool was during labelling, it is very likely that it saved a considerable amount of time. Assuming that using the tool annotating a single cartoon took 5 minutes on average, with initially not filtered 3634 cartoons, meant that annotation required roughly 302 hours of work. The time spent developing the labelling tool must also be considered. Development of the tool took about 40 hours, therefore the total time spent annotating was around 342 hours. Conversely, assuming that annotating without a special tool would double the amount of time needed on average to prepare a single cartoon, 604 hours of work were to be expected, if no labelling tool was used. Additionally,

Figure 4.1: Labelling the funniness of a cartoon using the labelling tool

the potential for errors would have been much greater, as the tasks performed were very repetitive and could have lead to more human errors. Such errors could be due to selecting a wrong cell in the spreadsheet application or losing all data due to the lack of a proper database.

The tool was implemented using Python, the same programming language as the machine learning part uses. The interface was a mixture of command line interface and web interface. Tasks which were only done once (such as filtering duplicates or applying the automated cartoon pre-processing) were triggered using a command line interface. The more repetitive tasks, such as manual preparation of a cartoon or funniness labelling was done in a web interface. For the web interface I chose the Django framework with an SQLite database [29][108].

4.3 Debugging of models

Debugging of deep learning models is a particularly difficult task, because interpreting neural networks is very difficult generally. It was necessary to detect whether the neural network did not train properly or if the problem itself was not tractable for the chosen methodology. It is not possible to fully detect this issue, but there are a few techniques a machine learning practitioner can apply to identify this problem.

The first and easiest technique is to check the network for exploding/vanishing gradients [81]. If either the gradients vanish towards 0 or explode towards infinity, then it is clear, that something is wrong with the network architecture. For example, the network may be too deep and the signal vanishes. Similarly, it is also very important to check the output values of the network. If the network always returns the same prediction, then there might be an issue with the target labels. During development I experienced such an issue, which was caused by incorrectly loaded labels.

One very useful approach to verify the neural networks was by making the task much simpler. For this debugging technique I replaced the cartoons with images containing the target label. For example, a cartoon with funniness 1 was replaced by an image of the number 1. If the trained model is not able to predict the target labels properly, we know that the implementation is faulty and not the task too difficult, as it is already established that neural networks are capable of detecting digits in images with very high accuracy.

Another problem that occured during implementation was when the performance was way too good. The accuracy reached over 80% after just a few epochs, which was very suspicious. After intense debugging I recognized that the train/validation/test split was not disjunct and the training data leaked into the validation set. This problem is known as data leakage.

4.4 Architecture

Figure 4.2 shows a simplified class diagram of the dataset system. The abstract *getItem* method allows for a unified access to all datasets, which makes it possible to use the same training code for different kinds of datasets. The *experiment* reference allows the dataset to apply experiment-specific transformations on the training data. For example, some experiments required different types of data augmentation.

The class diagram of figure 4.3 shows the structure of the experiments. An experiment always has one network associated with it, which defines the structure of the neural network (in PyTorch this is called *nn.Module*). Based on this network it defines all other required information: Which dataset to use (*getDataset*), which parameters should be optimized (*getOptimizationParameters*) and how to get the predictions of the network (*getPredictions*). Similarly the required evaluation is defined by *getEvaluations*, which returns multiple evaluations. Figure 4.4 shows the class diagram for the evaluation system.

By decoupling the experiment from the actual training algorithm it was very easy to reuse it for different types of experiments. The alternative would have been writing a similar gradient descent and evaluation logic for each experiment, which would have meant significantly more duplicated code. Running an experiment was done in a command line interface, where the parameter `-model` defined the experiment, `-train_cnn` trained a CNN-like architecture and the hyperparameters were also defined by such command line parameters. For example, by setting the parameter `-learning_rate` the learning rate during training is defined. This meant that it was very easy to try different configurations and to iterate quickly.

Originally I intended to use the same logic for the NLP part as well, but as the AllenNLP already provides a very similar functionality in the form of a JSON configuration, it would have been unreasonable to fit AllenNLP into the structure previously defined.

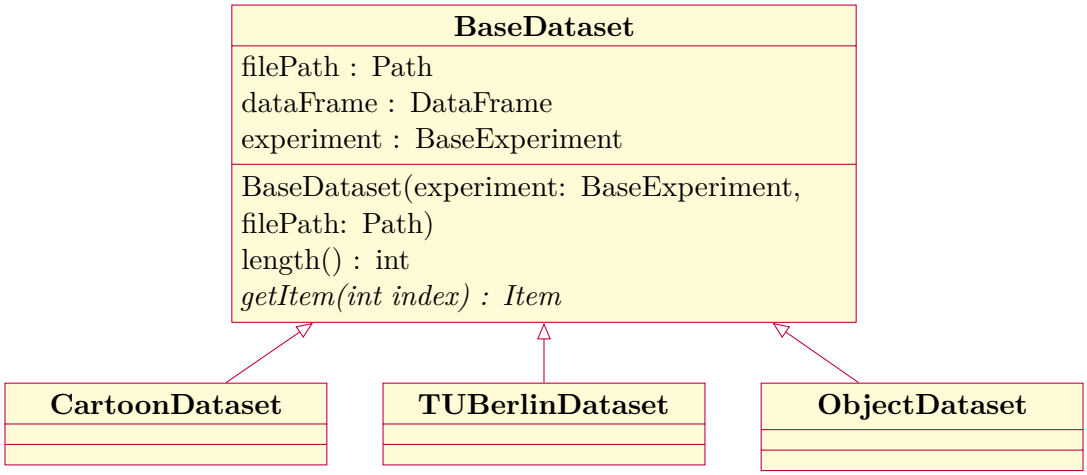


Figure 4.2: A UML class diagram illustrating a simplified structure of the dataset class hierarchy

I conclude the chapters Design and Implementation with the following summary: After the dataset was created using the custom labelling tool, the dataset analysis revealed possible patterns a classifier could exploit to potentially achieve better performance than the baseline. Several different architectures have been designed and implemented. The architectures for the visual domain included relatively simple convolutional neural networks, to more complicated transfer learning approaches and an object detection based approach. The text domain architectures includes an LSTM based approach using ELMo word embeddings, an ELMo sentence embeddings architecture and models learned by automated machine learning. Finally, all these finding cumulated in a two stage model using a cascade of multiple classifiers. During the implementation phase I have used a Python machine learning stack using PyTorch and AllenNLP. Based on the findings I gathered during the design and implementation process, I performed an evaluation of the trained models, which is outlined in the next chapter *Evaluation*.

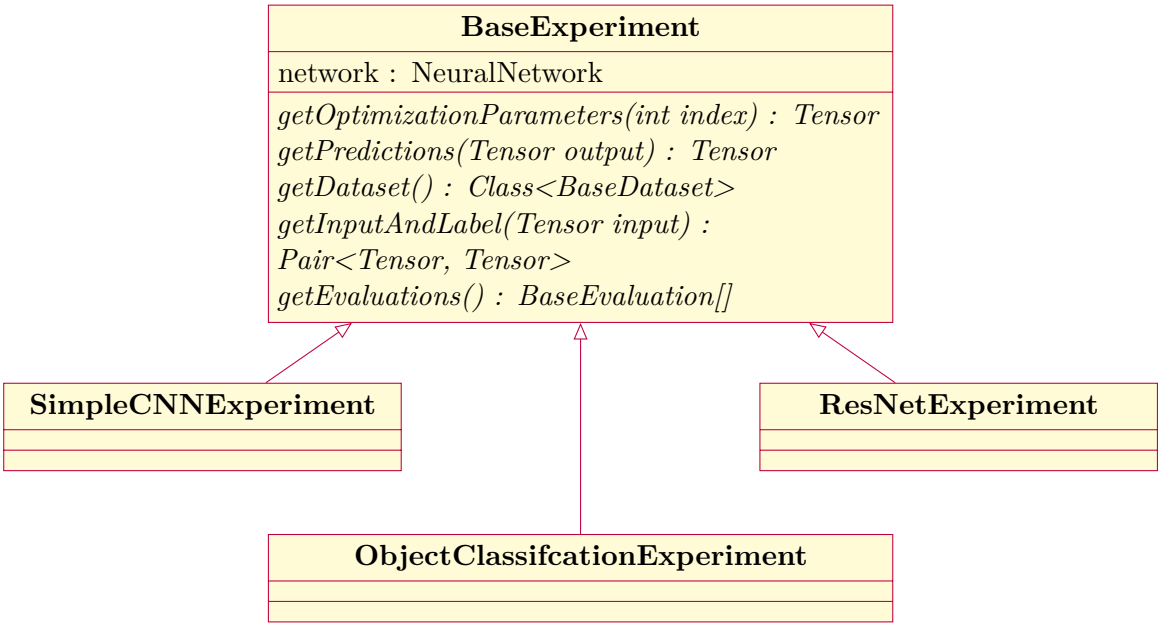


Figure 4.3: A UML class diagram detailing the experiment class hiararchy

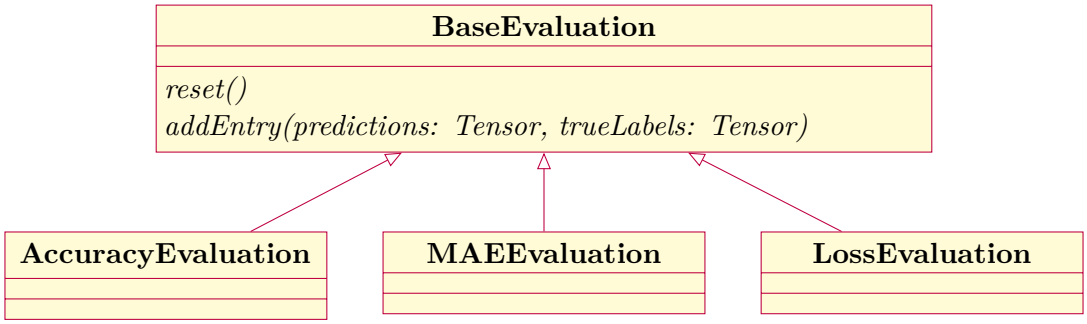


Figure 4.4: A UML class diagram illustrating the evaluation architecture

CHAPTER 5

Evaluation

This chapter evaluates the performance of the implemented architectures. I will outline the strengths and weaknesses of each implemented architecture quantitatively and qualitatively using confusion matrices, qualitative analysis of selected cartoons and metrics (mean absolute error and accuracy). I will also perform a runtime evaluation. Finally, I outline an analysis regarding the reproducibility of the funniness labels.

The cartoon dataset is split into three subsets:

- Training Set: 1492 samples (60%)
- Validation Data: 746 samples (30%)
- Test Data: 249 samples (10%)

The experiments were run on a PC running Ubuntu 18.04 LTS using PyTorch 1.0 and Python 3.6. The hardware is a GTX 1070 Max-Q with an Intel Core i7-8750H.

5.1 Architectures

In the following I will describe the results of the evaluation for the architectures presented in chapter 3. For comparison I evaluate several simple baseline strategies. Then I continue with the visual-only architectures and the punchline-only architectures. Finally, I outline the two stage model evaluation results.

5.1.1 Baseline

Four baseline strategies have been selected:

- Most Frequent Baseline: Picks the most frequent label in the dataset.
- Average Baseline: Returns the average funniness of the dataset. For the accuracy metric the average is rounded to the nearest integer.
- Random Baseline: Returns a random funniness picked uniformly from one to seven.
- Stratified Baseline: Returns a funniness sampled from the distribution of the training set.

For the mean absolute error metric the average performed best, while for the accuracy score the most frequent class has the best score.

5.1.2 Simple CNN

The simple CNN architecture overfits very quickly. The accuracy and MAE scores are very similar to the most frequent baseline, which indicates that the model most likely learns to predict the most frequent class.

5.1.3 Transfer Learning of Pretrained ResNet18

Noteworthy about this approach is that enabling the data augmentation in the testing/-validation phase makes it better than the baseline. The exact reason is not obvious, but indicates that the model expects the cartoons to be augmented. The data augmented version of this experiment unexpectedly achieved the best results during testing phase.

Sampling for each class one cartoon and examining the output layer of the trained neural network reveals some interesting insight about how the model works. In general the results show that the classifier never assigns high confidence into the predictions. No classification assigns a probability higher than 40%.

When examining the distribution across the different predictions over the different samples it seems that the general distribution does not change significantly. Some deviations are present, for example when comparing the probabilities for funniness level 2. Additionally, it seems that the model often ignores the input data and instead learns the label distribution. The similarities between the predictions and the histogram of funniness occurrences are high. For comparison, please refer to figures 5.1, 5.2 and 3.1.

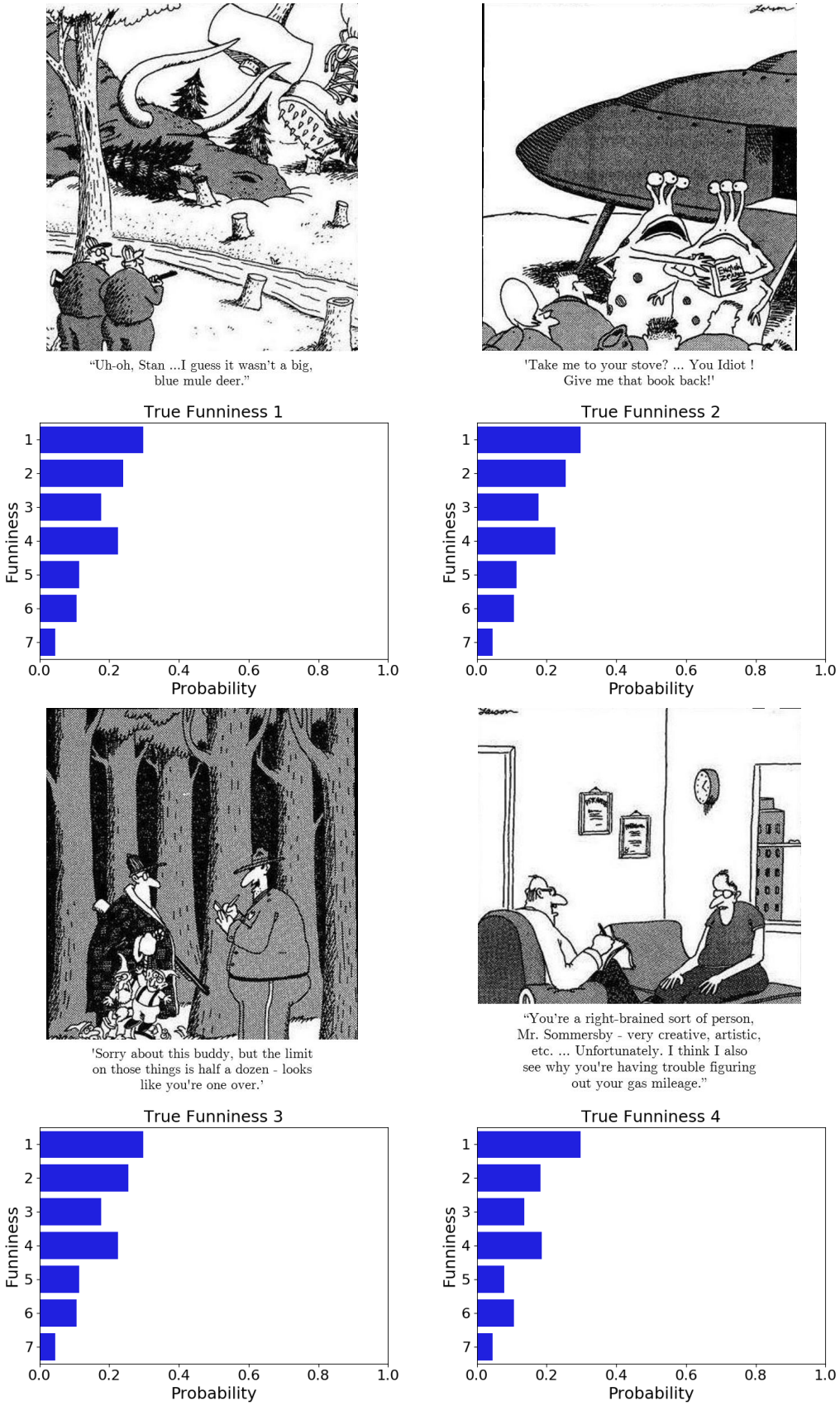


Figure 5.1: For the funniness classes 1, 2, 3 and 4: The bar plots represent the probability the classifier assigned to each category for the cartoons above.

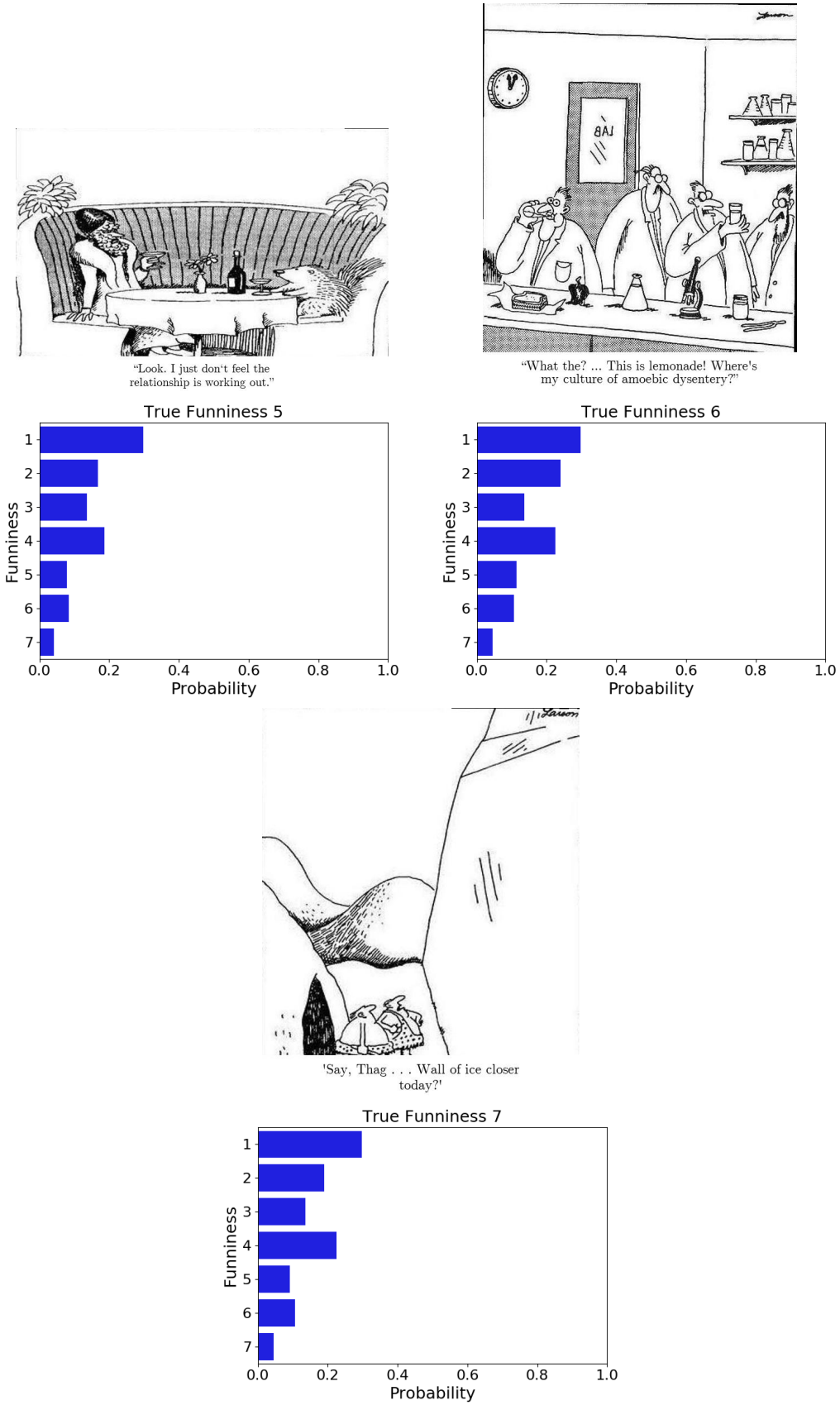


Figure 5.2: Continued for the funniness classes 5, 6 and 7.

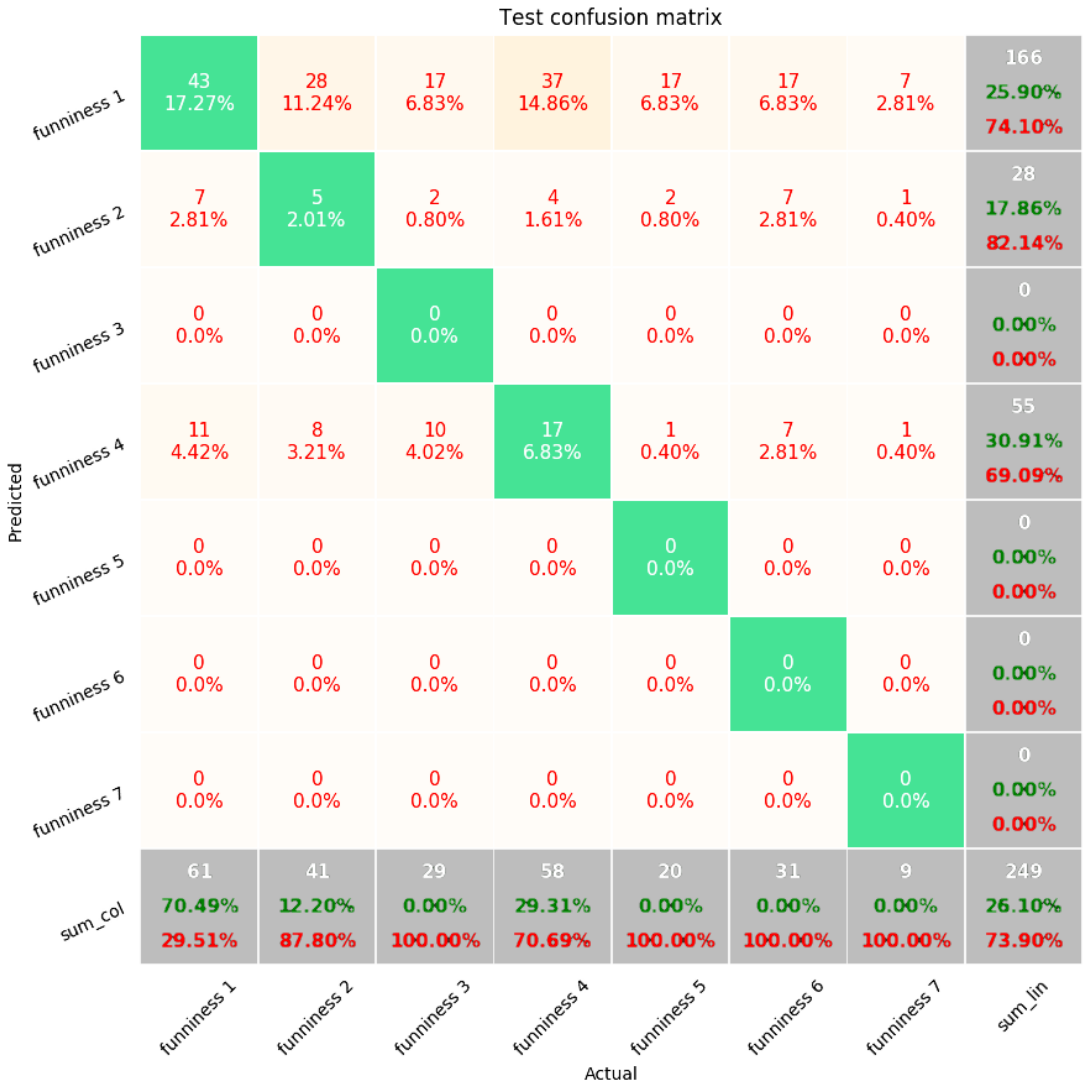


Figure 5.3: Confusion Matrix of the transfer learning CNN (with data augmentation) on the test split.

Confusion Matrices

One can clearly see at figure 5.3 and figure 5.4 that for both the validation and test split the model learned to approximate the label distribution, instead of learning to generalize the underlying structure. The three most frequent funniness classes (1, 4 and 2) are the only ones the model predicts and are therefore by chance correct. Other classes are not predicted once by the classifier.

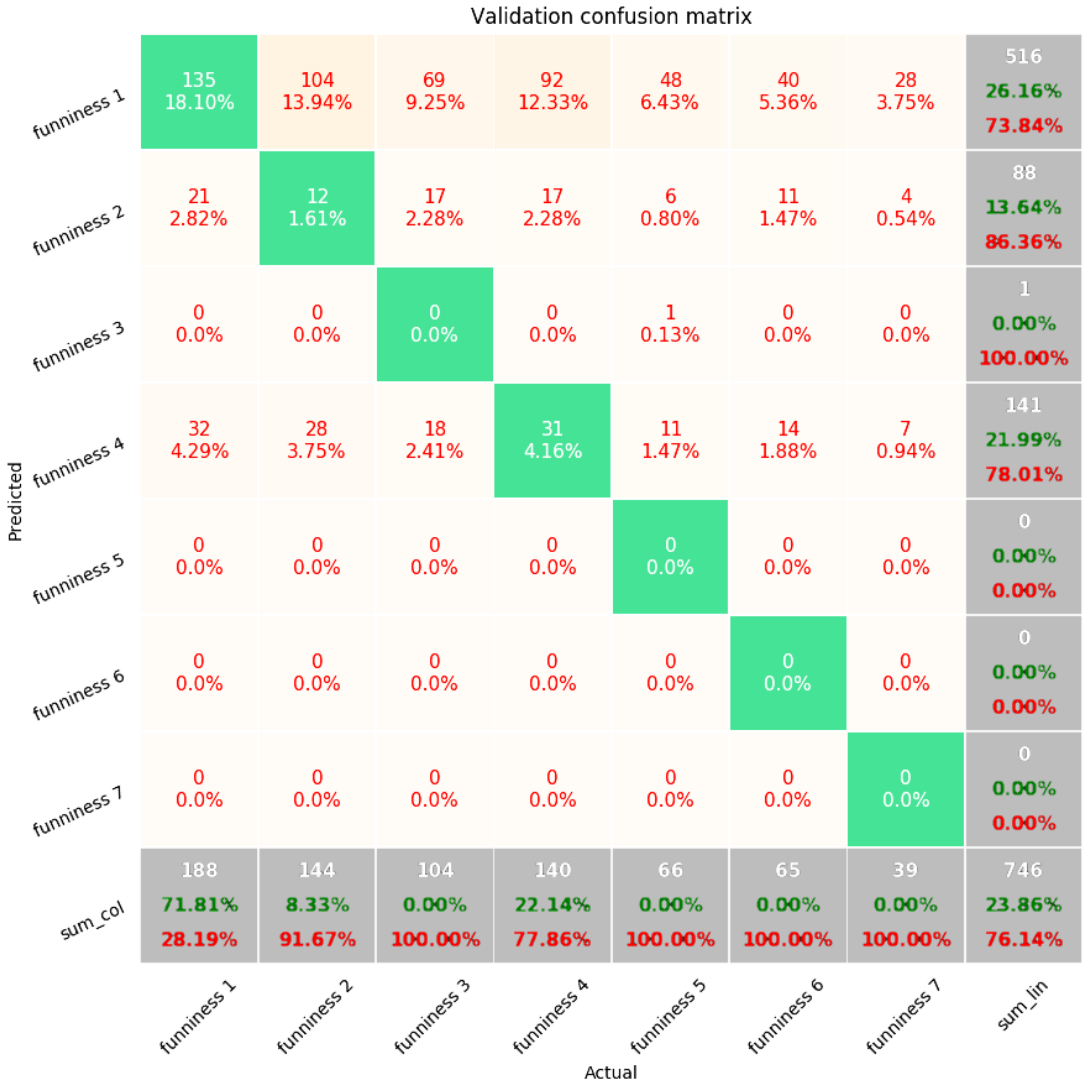


Figure 5.4: Confusion Matrix of the transfer learning CNN (with data augmentation) on the validation split.

5.1.4 Transfer Learned Object Detection

The result of the object detection approach are incomparable to the other results, as it was not implemented such that it predicted any funniness. Due to inconsistent detections of objects it seemed infeasible to continue with this approach.

5.1.5 ELMo Pretrained Model

Based on the pretrained ELMo model this approach initially seemed the most promising, as it achieved the highest validation accuracy. Unfortunately this was only due to overfitting, as the performance on the test set dropped below the baseline results.

Most likely a problem of this approach is that many insider jokes are lost in the word embeddings, due to the fact that the pretrained vocabulary of the ELMo model does not contain many of them, so they can not be used by this model. For example, the word *Thag* which is one of the most frequent terms for cartoons with funniness 7 is not in the ELMo vocabulary. ELMo's architecture considers the fact that not all words are in the vocabulary, but it seems that this is not sufficient for modelling and understanding the Gary Larson dataset.

5.1.6 AutoML Model

Initially, when performing the evaluation on the validation set, the TFIDF configuration looked very promising as well. Interestingly, it achieved top results with a very simple feature representation. Unfortunately, the test set evaluation revealed overfitting: The ELMo feature representation performed worse on the validation set and on the test set. Since the TFIDF feature representation would not have relied on a pre-trained vocabulary it would not have suffered from the problem of missing domain-specific words.

5.1.7 Two Stage Model

The two stage models are not better than the average baseline. The idea of trying to avoid overfitting by using multiple classifiers for each funniness class did not work as expected. Combining the visual and text information did improve the results, but not significantly. The MAE score is on par with the baseline, while still reaching a better accuracy. This improvement cannot be seen as a significant improvement.

Funniness	**1**	**2**	**3**	**4**	**5**	**6**	**7**
Two Stage Model	2.42	1.57	0.97	0.8	1.53	2.28	3.64
Baseline Average	2.21	1.21	0.21	0.79	1.79	2.79	3.79

Table 5.1: MAE per class

When comparing the MAE of the two stage model to the baseline (table 5.1) an interesting phenomenon can be observed: Even though the performance is very similar to the average baseline, the actual MAEs per class are different. This means that the model does not simply return the average funniness, but does something different. As neural networks are black-box models, it is very difficult to assert the exact reasons for why this happens.

The confusion matrices of all classifiers (figure 5.5) in the first stage reveal whether this approach could be effective. If all classifiers have a reasonable accuracy the second decision stage has a higher chance of predicting the correct class. If the first stage only returns noise the second stage would not be able to beat the average baseline. It seems that the first stage is not effective in doing so. For funniness 3, 4, 5, 6 and 7 there is no true negative, which means that the model for this class only learns to always return score "1".

5.1.8 Autoencoder

Against the expectation that the convolutional autoencoder would improve the results due to a significantly reduced input size, it did performed worse both on the validation and on the test set. Comparing the reconstructed cartoon with the original cartoon reveals that the autoencoder loses important semantic information which could be crucial for understanding humor. Importantly, the facial expressions are lost, as well as many other high frequency details. For example the native American lying on the floor is not recognizable anymore in figure 5.6.

These details seem to be very important for the funniness classification task and if missing cause the image data to be not more than additional noise, which is probably the reason why the two stage model with autoencoded cartoons performs worse compared to the other two stage models.

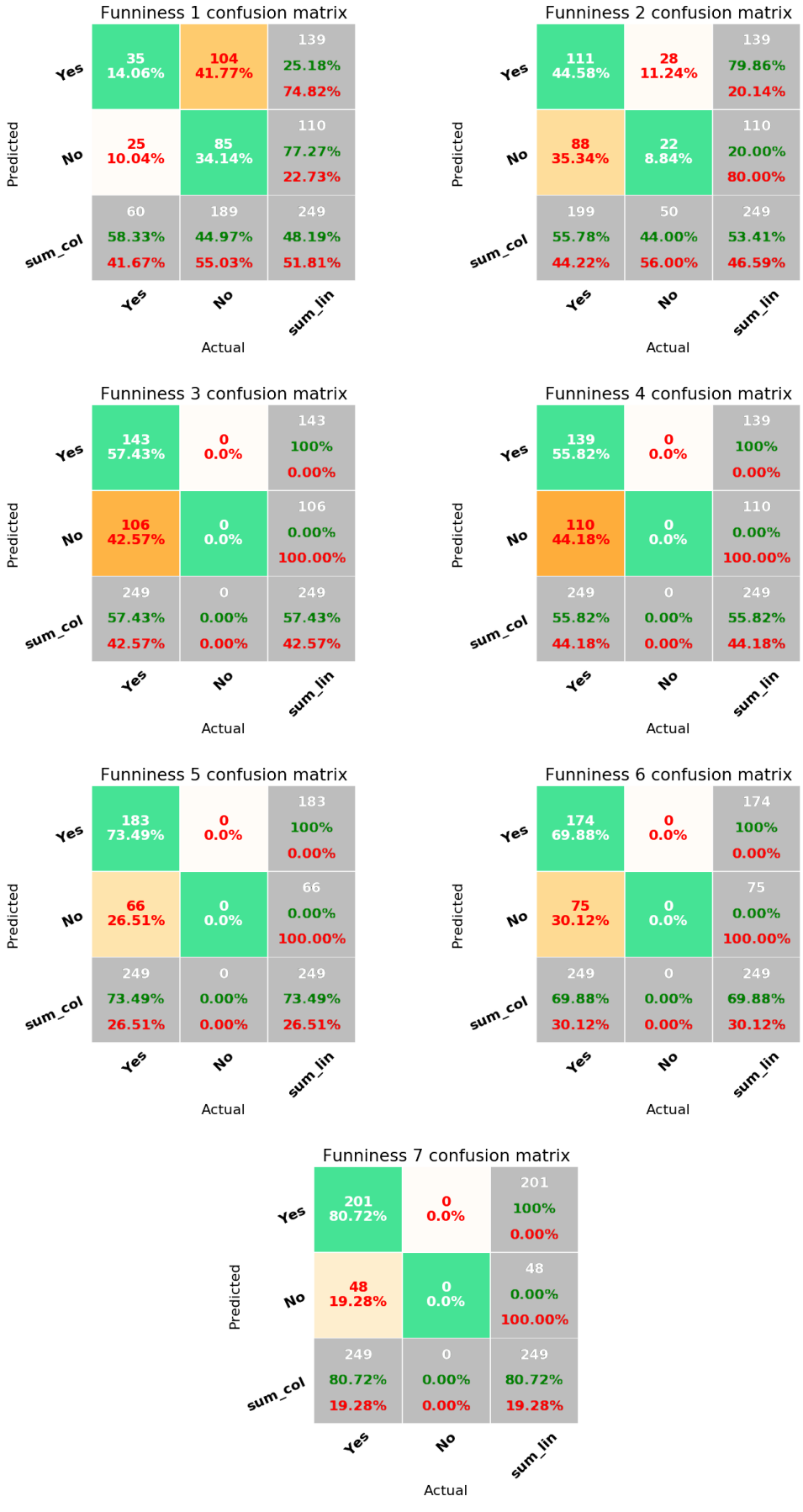


Figure 5.5: Confusion Matrices for first stage.

Figure 5.6: First row shows the original cartoons and the second row the results after applying encoder and decoder.

5.2 Results

The following subsections show the performance achieved in each experiment on the validation and test sets. Additionally the training runtime on the development machine is also measured.

5.2.1 Validation Results

For a detailed table of validation performance of selected models, please refer to figure 5.2. The ELMo based model achieved on this subset the highest accuracy, while the two stage model achieved the best Mean Absolute Error. Both models only use the text data and discard the visual information entirely.

5.2.2 Test Results

The results at figure 5.3 show the performance of the models for the test set. Compared to the validation set the best performing models change: For the test set, the model which maximizes the accuracy measure is using the visual data, namely the transfer learning CNN on the data augmented test set. Also the two stage model without cartoon data seems to generalize slightly better.

Experiment	**Validation MAE**	**Validation Accuracy**	**Text**	**Visual**
Baseline Most Frequent	2.14	25.20%		
Baseline Average	1.53	13.94%		
Baseline Random	2.33	13.81%		
Baseline Stratified	2.03	16.62%		
Simple CNN	2.14	25.20%		✓
Transfer Learning CNN	1.88	25.53%		✓
Transfer Learning CNN (with data augmentation)	1.96	23.86%		✓
ELMo	1.81	**26.81%**	✓	
AutoML Model with ELMo Vectors	2	25.87%	✓	
AutoML Model with TFIDF Vectors	2.05	26.76%	✓	
Two Stage Model	**1.52**	18.10%	✓	
Two Stage Model with Cartoons	1.55	15.68%	✓	✓
Two Stage Model with Autoencoded Cartoons	1.62	15.82%	✓	✓

Table 5.2: Model performances on the validation set.

Experiment	**Test MAE**	**Test Accuracy**	**Text**	**Visual**
Baseline Most Frequent	2.26	24.50%		
Baseline Average	**1.57**	11.65%		
Baseline Random	2.17	13.65%		
Baseline Stratified	2.22	14.46%		
Simple CNN	2.26	24.50%		✓
Transfer Learning CNN	1.96	24.90%		✓
Transfer Learning CNN (with data augmentation)	2.01	**26.10%**		✓
ELMo	1.84	25.70%	✓	
AutoML Model with ELMo Vectors	2.12	24.50%	✓	
AutoML Model with TFIDF Vectors	2.23	24.90%	✓	
Two Stage Model	1.6	18.80%	✓	
Two Stage Model with Cartoons	**1.57**	16.06%	✓	✓
Two Stage Model with Autoencoded Cartoons	1.58	14.46%	✓	✓

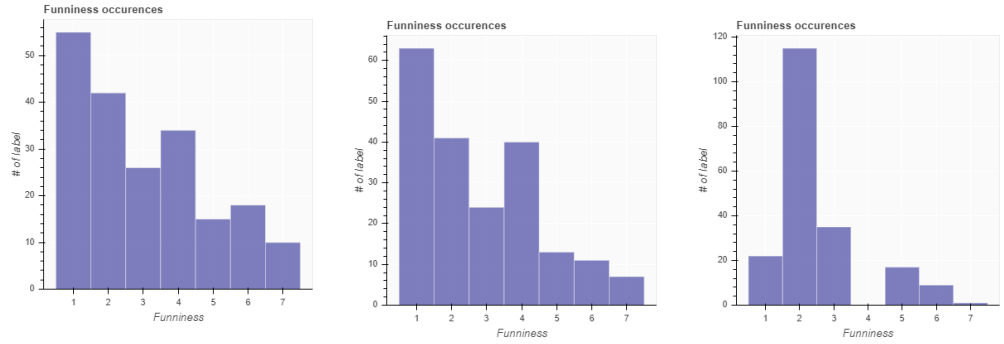
Table 5.3: Model performances on the test set.

Experiment	**Train Duration**	**Validation MAE**	**Validation Accuracy**	**Test MAE**	**Test Accuracy**
Baseline Most Frequent	<1s	2.14	25.20%	2.26	24.50%
Baseline Average	<1s	1.53	13.94%	**1.57**	11.65%
Baseline Random	<1s	2.33	13.81%	2.17	13.65%
Baseline Stratified	<1s	2.03	16.62%	2.22	14.46%
Simple CNN	7m 50s	2.14	25.20%	2.26	24.50%
Tranfer Learning CNN	15m 45s	1.88	25.53%	1.96	24.90%
Tranfer Learning CNN (with data augmentation)	9m 52s	1.96	23.86%	2.01	**26.10%**
ELMo	4m 39s	1.81	**26.81%**	1.84	25.70%
AutoML Model with ELMo Vectors	29m 7s	2	25.87%	2.12	24.50%
AutoML Model with TFIDF Vectors	24m 20s	2.05	26.76%	2.23	24.90%
Two Stage Model	8m 42s	**1.52**	18.10%	1.6	18.80%
Two Stage Model with Cartoons	39m 42s	1.55	15.68%	1.57	16.06%
Two Stage Model with Autoencoded Cartoons	15m 23s	1.62	15.82%	1.58	14.46%
Autoencoder	32m 34s				

Table 5.4: Training runtime of each experiment

5.2.3 Training Duration

Figure 5.4 shows the training durations of each experiment. A general trend is that the text based models (ELMo, Two Stage Model) train significantly faster than the visual based models (Simple CNN, Transfer Learning CNN, Two Stage Model with Cartoons). This can be explained by the size of the input vector for the neural network. The data for the text representation is denser compared to the representation of images.



(a) *Original* annotation set (200 samples)

(b) *Future* annotation set

(c) *Other* annotation set

Figure 5.7: This figure shows a comparison of the label distribution of the annotation sets *Original*, *Future* and *Other*. To improve comparability the *Original* annotation set contains the same number of samples as the other two annotation sets by randomly sampling 200 cartoons.

5.3 Label Reproducibility

An important question regarding the dataset is whether it is even possible for a human to reproduce the labels, as well as whether humor is actually subjective. Do different people find the same cartoons equally funny? To give a first indication on the answer to these questions, I randomly sampled a subset of the dataset (200 cartoons). This subset has been annotated using the same rules as the original annotation (rank each cartoon from 1 to 7). Then I applied the same performance metrics to the different annotation sets and evaluate them accordingly.

In table 5.5, the *Future* annotation set is by the same annotator (Robert Fischer) as the *Original* annotation with the only difference that this annotation was performed approximately one year later. This means that the original funniness annotations most likely have already been forgotten by the annotator. For example, if the timespan between the original and the future annotations was in the order of weeks or months, the annotator could have possibly remembered what he or she annotated before, which could have distorted the results. This dataset shows if it is even possible for the same person to reproduce his or her annotations. The *Other* annotation set was created by the supervisor of this thesis. If the performance metrics are significantly improved compared to the baseline, one could argue that the humor is similar between those annotators. *Most Frequent* is the comparison with the most frequent label of the annotation set it is compared to. *Mean* is similarly the mean of all annotated labels, for the accuracy score the mean is rounded half away from zero.

There might be a problem of bias, as both annotators might have some interest in the success of this thesis. This was mitigated as much as possible by being aware of it. Other solutions would also have significant problems. For example, the alternative of a paid

Annotation Set 1	Annotation Set 2	MAE	Accuracy
Original	Future	1.08	35.86%
Original	Other	1.67	18.70%
Future	Other	1.53	21.21 %
Future	Most Frequent	1.79	31.81%
Future	Mean	1.47	12.12%
Other	Most Frequent	0.61	57.57%
Other	Mean	0.76	57.57%

Table 5.5: This table shows the performance measures compared for different annotation sets. *Original* is the original annotation, *future* is the annotation from the same author a year later, *other* is the annotation from a different person.

annotator would have made it impossible to actually validate the results. Distinguishing randomly selected annotations from true funniness annotations seems to be impossible. Additionally, it would have been costly, as doing these annotations is rather labour intensive.

Figure 5.5 shows the metrics gained from the annotation sets. The annotation sets *Original* and *Future* are the only annotations significantly better than the baselines of the original test set (both the MAE and accuracy). All things considered, the reproducibility is low. This may be due to different factors, such as humor changing over time and with the annotators mood. Contrary, the annotations of the *Other* annotation set are not similar to the *Original* set. This may lead to the conclusion that humor is indeed subjective, as well as it is to a certain degree reproducible. Comparing the baselines for each annotation set reveals different distributions. The most frequent and mean baselines have relatively high accuracy and MAE, compared to the annotations of *Original* and *Future*. This could mean, that there is a high variance among different annotators. Figure 5.7 shows a significant difference in label distribution when comparing the *Other* with the *Original* and *Future* annotation sets. Contrary, there is a high similarity when comparing the *Original* and *Future* label distribution, supporting the assumption that annotators are able to reproduce their own funniness score.

5.4 Discussion

In computer science there is the concept of the *semantic gap*: "A *semantic gap* is the difference in meaning between constructs formed within different representation systems. A representation system can be e.g. a natural language, a formal language or a *proper language* within an engineering domain" [43]. It may be that the humor operates in a completely different representation system than I chose for this thesis. The main representation system for the punchlines were essentially statistical word and sentence embeddings (ELMo, GloVe [84][85]), and the main representation system for the cartoons were the resulting vectors of the convolutional feature extractors. Both representation systems work for certain tasks of artificial intelligence very well (for example image classification) but failed for modelling humor in the context of this thesis.

Before deep learning has achieved its successes there was a similar issue of semantic gap for tasks like image classification [88]. It was not reliably possible to distinct a cat from a dog, before deep learning [52]. This was because the representation for images, which was obtained by traditional computer vision techniques, was too distinct from the human representation to perform a proper classification. The semantic gap was closed once convolutional feature extractors performed a transformation which made it possible to easily separate cats from dogs semantically. Interestingly, after further examination of these convolutional feature extractors researchers found striking similarities with the human vision system [19]. By introducing feature vectors from object detection I tried to bridge this semantic gap further, but unfortunately this was not possible with the available datasets.

The chosen machine learning algorithms were not able to extract useful patterns from the available data, which is supported by the fact that no model performed significantly better than the baselines. Sometimes a minor improvement was achieved, but either the results were just statistical outliers or if a model actually learned something, it was most likely a *clever hans* phenomenon. Especially, natural language processing is prone to such *clever hans* behaviour, where the neural network exploits statistical cues of the underlying data, instead of actually generalizing humor [74]. For example, when examining the actual output distribution (figures 5.1 and 5.2) we see that the output probabilities have a striking similarity with the label distribution (figure 3.1). This is an indication that the trained models do not extract useful features, but instead learn to mainly approximate the label distribution.

As the model's training split contains 1429 samples, it may very well be that the data does not capture enough information about the humor of the annotator. Humor is very context sensitive. It depends on many different outside factors, such as the culture, the education and the current mood of the person. Based on all these factors it may be very difficult for a model to generalize into a predictive model, as it does not know these influences beforehand, especially as they change over time and are not static.

Interestingly, also the models with access to both the punchline and cartoon information perform equal or worse than the baselines. One could expect that more information could

lead to a better generalization capability. As the results shown in table 5.3 indicate this is not the case. The problem with more information is that it increases the dimensionality of the problem as well. The more parameters a model has (more degrees of freedom), the more data are usually required to train such models. This is why most successful deep learning models are trained on huge datasets, as it is not uncommon for a deep learning model to have several million parameters [115][41]. This is called the *curse of dimensionality* [37] and may be a reason why combining both input domains does not improve the performance, as expected.

Through dimensionality reduction machine learning practitioners can decrease the number of parameters a model requires, but in the process of doing so the data might lose crucial information as well. For example, the autoencoder I have applied does not result in an improved performance. This may be due to the loss of such crucial information. Qualitatively, I tried to understand some of the cartoons shown in figure 5.6 and I was not able to understand these cartoons anymore. This may be a sign for such loss of semantic information. If a human is not able to reproduce the meaning properly, it seems to be even more difficult for a machine learning model to achieve essentially the same task.

Another issue of all the architectures was the class imbalance. All models essentially only classified the lower funniness (1 ,2, 3), but virtually never chose the higher classes (5, 6, 7). This phenomenon is illustrated in the confusion matrices of figures 5.3 and 5.4. The number of annotations for the higher funniness levels was orders of magnitude lower then for the lower funniness levels. The models generalized and never predicted these classes. Oversampling and custom loss functions did not improve this problem. This problem seems to be intrinsic to humor, as it is usually more common to consider something not funny, as to consider something to be actually funny.

Eventually, the label reproducibility reveals that humor is indeed subjective and it cannot be assumed that two people find the same cartoons funny - at least according to the data gathered for this thesis. Additionally, the same person seems to be able to reproduce the same annotations to a certain degree, but not very reliably. Nevertheless, a machine learning algorithm should in theory be sufficiently powerful to reproduce these results.

To summarize the discussion, no model learned to generalize the humor of the annotated funniness to a satisfactory degree. There has been no significant improvement compared to the baselines and the concept of humor seems to be intractable for the chosen architectures. Given these results, I will outline the conclusion of this thesis in the following chapter.

CHAPTER 6

Conclusion

This thesis aimed at modelling the humor of a person using deep learning. This was achieved by a custom dataset created from Gary Larson's cartoons. The dataset itself may - if copyright issues can be resolved - be used in future research and allows for novel analysis in the field of computational humor. The techniques developed allow for quantitative comparison of humor between different annotators. Afterwards I trained several deep learning based models for the visual domain (cartoons), text domain (punchlines) and a combined model which considers both domains simultaneously. Unfortunately, the main goal selected for this dataset was not achieved, namely humor classification.

Potential causes can not be attributed to exactly one issue. One reason is most certainly, that the problem space is not sufficiently covered by the dataset. As most deep learning datasets are many orders of magnitudes larger. Unfortunately, it is unrealistic that a human will annotate millions of cartoons. Therefore it would be required to apply transfer learning. Unfortunately, it seems that current techniques of transfer learning are not sufficiently sophisticated to solve this problem. Finding a way to transfer knowledge from other domains to the domain of cartoons is an important challenge to solve in the future.

One could also argue that the problem itself is flawed. Letting a human rate a cartoon is very subjective and likely depends on many factors. For example: Current mood, whether the person has already seen the cartoon, time of day and also the exact reason for performing the annotation. It could be that these factors outweigh the true humor. Evaluation shows, that there is certainly a high degree of noise in the annotations and it is even for the same annotator difficult to reproduce the annotations. Finding a set up where this noise can be reduced could be an important step for the modelling of humor. For example, future work could reduce the number of categories to *not funny*, *funny* and *very funny*.

Another reason may be the semantic gap between the funniness annotation and the actual experience of humor [43]. Funniness is difficult to project onto a scale from 1 to 7, it may possibly be that it is simply *lost in translation* when performing such annotation.

6.1 Future Work

An important topic for humor is scene understanding. Understanding the cartoons on a higher semantic level than currently seems to be important for humor classification. The approaches applied in this thesis (for example, a transfer learned object detector) did not prove successful. Available datasets for this domain are not in the domain of cartoons and therefore are not applicable in this setting. An interesting task would be cartoon theme classification. If a model is able to predict the theme of a cartoon it could theoretically use this information also for humor classification. Such a classification could be achieved by annotating the cartoon themes.

Additionally, training a successful model which performs robust object detection could be useful for both humor and cartoon classification. Solving this task could be achieved either by properly applying transfer learning from a similar enough domain or by annotating the objects occurring in the dataset itself.

To close the semantic gap further, future work could measure brain activity per cartoon. Instead of predicting the funniness score one could instead build a model which predicts the brain activity observed when the annotator sees the respective cartoon. If such model can predict the brain activity better than the models based on the funniness score, then it might be an indication for the source of the semantic gap (namely the process of scoring).

Eventually, the dataset should be expanded to other authors of cartoon series. Adding new authors could make it possible to quantitatively compare the funniness perception of the authors. Is one author more funny than the other? Is it possible to transfer a machine learning model from one author to the other? A successful model should in theory be able to be transferable and generalize humor.

List of Figures

Bibliography

[1] Google's quick, draw! dataset. https://opensource.google/projects/quickdrawdataset. [Online; accessed 28-10-2019].

[2] Hyperopt-sklearn github repository. https://github.com/hyperopt/hyperopt-sklearn/, 2019. [Online; accessed 09-08-2019].

[3] Oxford dictionary, 2019.

[4] R. Al-Shalabi and R. Obeidat. Improving knn arabic text classification with n-grams based document indexing. In *Proceedings of the Sixth International Conference on Informatics and Systems, Cairo, Egypt*, pages 108–112, 2008.

[5] K. Arulkumaran, A. Cully, and J. Togelius. Alphastar: An evolutionary computation perspective, 2019. cite arxiv:1902.01724.

[6] A. Augello, G. Saccone, S. Gaglio, and G. Pilato. Humorist bot: Bringing computational humour in a chat-bot system. In *2008 International Conference on Complex, Intelligent and Software Intensive Systems*, pages 703–708, March 2008.

[7] J. Bergstra and Y. Bengio. Random search for hyper-parameter optimization. *Journal of Machine Learning Research*, 13(Feb):281–305, 2012.

[8] D. Bertero and P. Fung. Deep learning of audio and language features for humor prediction. In *LREC*, 2016.

[9] D. Bertero and P. Fung. A long short-term memory framework for predicting humor in dialogues. In *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 130–135, 2016.

[10] K. Binsted, A. Nijholt, O. Stock, C. Strapparava, G. Ritchie, R. Manurung, H. Pain, A. Waller, and D. O'Mara. Computational humor. *IEEE Intelligent Systems*, 21(2):59–69, March 2006.

[11] C. M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag, Berlin, Heidelberg, 2006.

[12] A. Borghesi, A. Bartolini, M. Lombardi, M. Milano, and L. Benini. Anomaly detection using autoencoders in high performance computing systems. *CoRR*, abs/1811.05269, 2018.

[13] G. Bradski. The OpenCV Library. *Dr. Dobb's Journal of Software Tools*, 2000.

[14] M. Buijzen and P. Valkenburg. Developing a typology of humor in audiovisual media. *Media Psychology - MEDIA PSYCHOL*, 6:147–167, 05 2004.

[15] D. V. Carvalho, E. M. Pereira, and J. S. Cardoso. Machine learning interpretability: A survey on methods and metrics. *Electronics*, 8(8):832, Jul 2019.

[16] S. Chaidaroon and Y. Fang. Variational deep semantic hashing for text documents. *CoRR*, abs/1708.03436, 2017.

[17] P.-Y. Chen and V.-W. Soo. Humor recognition using deep learning. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 2 (Short Papers)*, pages 113–117, 2018.

[18] L. Chiruzzo, S. Castro, M. Etcheverry, D. Garat, J. J. Prada, and A. Rosá. Overview of haha at iberlef 2019: Humor analysis based on human annotation. In *Proceedings of the Iberian Languages Evaluation Forum (IberLEF 2019). CEUR Workshop Proceedings, CEUR-WS, Bilbao, Spain (9 2019)*, 2019.

[19] R. M. Cichy, A. Khosla, D. Pantazis, A. Torralba, and A. Oliva. Comparison of deep neural networks to spatio-temporal cortical dynamics of human visual object recognition reveals hierarchical correspondence. *Scientific Reports*, 6:27755 EP –, Jun 2016. Article.

[20] D. Ciregan, U. Meier, and J. Schmidhuber. Multi-column deep neural networks for image classification. In *2012 IEEE Conference on Computer Vision and Pattern Recognition*, pages 3642–3649, June 2012.

[21] M. Claesen and B. De Moor. Hyperparameter search in machine learning. 02 2015.

[22] D. Crevier. *AI: The Tumultuous History of the Search for Artificial Intelligence*. Basic Books, Inc., New York, NY, USA, 1993.

[23] F. Crick. The recent excitement about neural networks. *Nature*, 337:129–132, 1989.

[24] G. Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals, and Systems (MCSS)*, 2(4):303–314, Dec. 1989.

[25] E. Davis. Algorithms and everyday life. *Artificial Intelligence*, 239:1–6, 2016.

[26] P.-T. de Boer, D. Kroese, S. Mannor, and R. Rubinstein. A tutorial on the cross-entropy method. *Annals of operations research*, 134(1):19–67, 1 2005.

[27] J. Devlin, M. Chang, K. Lee, and K. Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. *CoRR*, abs/1810.04805, 2018.

[28] R. Dey and F. M. Salem. Gate-variants of gated recurrent unit (GRU) neural networks. *CoRR*, abs/1701.05923, 2017.

[29] Django Development Team. Django. `https://www.djangoproject.com/`, 2020 (accessed 09.01.2020).

[30] J. Dougherty, R. Kohavi, and M. Sahami. Supervised and unsupervised discretization of continuous features. In *Proceedings of the Twelfth International Conference on International Conference on Machine Learning*, ICML'95, pages 194–202, San Francisco, CA, USA, 1995. Morgan Kaufmann Publishers Inc.

[31] M. Eitz, J. Hays, and M. Alexa. How do humans sketch objects? *ACM Trans. Graph. (Proc. SIGGRAPH)*, 31(4):44:1–44:10, 2012.

[32] P. Flach. Performance evaluation in machine learning: The good, the bad, the ugly, and the way forward. In *AAAI*, 2019.

[33] R. S. Francois Chaubard, Rohit Mundra. Francois chaubard, rohit mundra, richard socher, 2015.

[34] P. Galdi and R. Tagliaferri. *Data Mining: Accuracy and Error Measures for Classification and Prediction*. 01 2018.

[35] M. Gardner, J. Grus, M. Neumann, O. Tafjord, P. Dasigi, N. F. Liu, M. E. Peters, M. Schmitz, and L. Zettlemoyer. Allennlp: A deep semantic natural language processing platform. *CoRR*, abs/1803.07640, 2018.

[36] L. H. Gilpin, D. Bau, B. Z. Yuan, A. Bajwa, M. Specter, and L. Kagal. Explaining explanations: An approach to evaluating interpretability of machine learning. *CoRR*, abs/1806.00069, 2018.

[37] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[38] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio. Generative adversarial nets. In *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'14, pages 2672–2680, Cambridge, MA, USA, 2014. MIT Press.

[39] A. Griewank. Who invented the reverse mode of differentiation. 2012.

[40] K. Gurney. *An Introduction to Neural Networks*. Taylor & Francis, Inc., Bristol, PA, USA, 1997.

[41] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015.

[42] X. He, K. Zhao, and X. Chu. Automl: A survey of the state-of-the-art, 2019.

[43] A. M. Hein. Identification and bridging of semantic gaps in the context of multi-domain engineering. 01 2010.

[44] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural Comput.*, 9(8):1735–1780, Nov. 1997.

[45] B. Huval, T. Wang, S. Tandon, J. Kiske, W. Song, J. Pazhayampallil, M. Andriluka, P. Rajpurkar, T. Migimatsu, R. Cheng-Yue, F. A. Mujica, A. Coates, and A. Y. Ng. An empirical evaluation of deep learning on highway driving. *CoRR*, abs/1504.01716, 2015.

[46] K. Janocha and W. M. Czarnecki. On loss functions for deep neural networks in classification. *CoRR*, abs/1702.05659, 2017.

[47] Jupyter Notebook Development Team. Jupyter notebook. `https://jupyter.org/`, 2020 (accessed 09.01.2020).

[48] D. Jurafsky and J. Martin. *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition*, volume 2. 02 2008.

[49] Keras Community. Keras documentation - early stopping. `https://keras.io/callbacks/#earlystoppin`, 2019. [Online; accessed 06-11-2019].

[50] B. Komer, J. Bergstra, and C. Eliasmith. Hyperopt-sklearn: Automatic hyperparameter configuration for scikit-learn. pages 32–37, 01 2014.

[51] A. Krizhevsky, V. Nair, and G. Hinton. Cifar-100 (canadian institute for advanced research).

[52] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012.

[53] S. Kumar, C. Columbus, and R. Scholar. Various improved tfidf schemes for term weighing in text categorization: A survey. *International Journal of Engineering Research*, 10:11905–11910, 04 2015.

[54] W. Lan and W. Xu. Neural network models for paraphrase identification, semantic textual similarity, natural language inference, and question answering. *CoRR*, abs/1806.04330, 2018.

[55] Y. Li, K. Bontcheva, and H. Cunningham. Adapting svm for natural language learning : A case study involving information extraction. 2006.

[56] M. Lin, Q. Chen, and S. Yan. Network in network, 2013.

[57] Z. C. Lipton. A critical review of recurrent neural networks for sequence learning. *CoRR*, abs/1506.00019, 2015.

[58] E. Loizou. Humour: A different kind of play. *European Early Childhood Education Research Journal - EUR EARLY CHILD EDUC RES J*, 13:97–109, 01 2005.

[59] P. Majumder, M. Mitra, and B. Chaudhuri. N-gram: a language independent approach to ir and nlp. In *International conference on universal knowledge and language*, 2002.

[60] C. D. Manning and H. Schütze. *Foundations of Statistical Natural Language Processing*. MIT Press, Cambridge, MA, USA, 1999.

[61] J. Markoff. `https://www.nytimes.com/2005/10/14/technology/behind-artificial-intelligence-a-squadron-of-bright-real-people.html`, 2005. Online; accessed 03-11-2019.

[62] R. Martin. Sense of humor and physical health: Theoretical issues, recent findings, and future directions. *Humor-international Journal of Humor Research - HUMOR*, 17:1–19, 01 2004.

[63] P. McCorduck. *Machines Who Think: A Personal Inquiry into the History and Prospects of Artificial Intelligence*. AK Peters Ltd, 2004.

[64] A. Melnyk. Searle's abstract argument against strong ai. *Synthese*, 108(3):391–419, 1996.

[65] Q. Meng, D. R. Catchpoole, D. B. Skillicorn, and P. J. Kennedy. Relational autoencoder for feature extraction. *CoRR*, abs/1802.03145, 2018.

[66] T. Mikolov, K. Chen, G. S. Corrado, and J. A. Dean. Computing numeric representations of words in a high-dimensional space, Mar. 26 2013. US Patent 10,241,997.

[67] G. A. Miller. Wordnet: A lexical database for english. *Commun. ACM*, 38(11):39–41, Nov. 1995.

[68] M. Minsky and S. Papert. Perceptrons - an introduction to computational geometry. 1969.

[69] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. Playing atari with deep reinforcement learning, 2013.

[70] J. Morreall. Philosophy of humor. In E. N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, winter 2016 edition, 2016.

[71] M. Mulder and A. Nijholt. Humour research: State of the art. 11 2002.

[72] A. Ng et al. Sparse autoencoder. *CS294A Lecture notes*, 72(2011):1–19, 2011.

[73] N. J. Nilsson. *The Quest for Artificial Intelligence: A History of Ideas and Achievements*. Cambridge University Press, Cambridge, UK, 2010.

[74] T. Niven and H.-Y. Kao. Probing neural network comprehension of natural language arguments. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 4658–4664, Florence, Italy, July 2019. Association for Computational Linguistics.

[75] NumPy Development Team. Numpy. `https://numpy.org/`, 2020 (accessed 09.01.2020).

[76] C. Nwankpa, W. Ijomah, A. Gachagan, and S. Marshall. Activation functions: Comparison of trends in practice and research for deep learning. *CoRR*, abs/1811.03378, 2018.

[77] M. Nyamisa. A survey of information retrieval techniques. *Advances in Networks*, 5:40, 01 2017.

[78] C. Olah. Calculus on computational graphs: Backpropagation, 2015. [Online; accessed 06-11-2019].

[79] C. Olah. Understanding lstm networks, 2015. [Online; accessed 06-11-2019].

[80] Pandas Development Team. pandas. `https://pandas.pydata.org/`, 2020 (accessed 09.01.2020).

[81] R. Pascanu, T. Mikolov, and Y. Bengio. Understanding the exploding gradient problem. *CoRR*, abs/1211.5063, 2012.

[82] J. Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1988.

[83] J. Pennington, R. Socher, and C. D. Manning. Glove: Global vectors for word representation. In *Empirical Methods in Natural Language Processing (EMNLP)*, pages 1532–1543, 2014.

[84] J. Pennington, R. Socher, and C. D. Manning. Glove: Global vectors for word representation. In *In EMNLP*, 2014.

[85] M. E. Peters, M. Neumann, M. Iyyer, M. Gardner, C. Clark, K. Lee, and L. Zettlemoyer. Deep contextualized word representations. *CoRR*, abs/1802.05365, 2018.

[86] R. Ponnusamy. A systematic survey of natural language processing (nlp) approaches in different systems. 4, 01 2018.

[87] D. M. W. Powers. Evaluation: From precision, recall and f-factor to roc, informedness, markedness and correlation. 2008.

[88] C. Pramerdorfer. Lecture slides of deep learning for visual computing, 2019.

[89] L. Prechelt. Early stopping - but when? 03 2000.

[90] Y. Pu, Z. Gan, R. Henao, X. Yuan, C. Li, A. Stevens, and L. Carin. Variational autoencoder for deep learning of images, labels and captions. In D. D. Lee, M. Sugiyama, U. V. Luxburg, I. Guyon, and R. Garnett, editors, *Advances in Neural Information Processing Systems 29*, pages 2352–2360. Curran Associates, Inc., 2016.

[91] Python Software Foundation. Python. `https://www.python.org/`, 2020 (accessed 09.01.2020).

[92] PyTorch Development Team. Pytorch. `https://pytorch.org/`, 2020 (accessed 09.01.2020).

[93] J. Quionero-Candela, M. Sugiyama, A. Schwaighofer, and N. D. Lawrence. *Dataset Shift in Machine Learning*. The MIT Press, 2009.

[94] S. Raschka. Model evaluation, model selection, and algorithm selection in machine learning. *CoRR*, abs/1811.12808, 2018.

[95] B. D. Ripley and N. L. Hjort. *Pattern Recognition and Neural Networks*. Cambridge University Press, New York, NY, USA, 1st edition, 1995.

[96] G. Ritchie. Current directions in computational humour. *Artificial Intelligence Review*, 16(2):119–135, Oct 2001.

[97] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edition, 2009.

[98] T. Salimans, I. J. Goodfellow, W. Zaremba, V. Cheung, A. Radford, and X. Chen. Improved techniques for training gans. *CoRR*, abs/1606.03498, 2016.

[99] M. Sanderson and W. Croft. The history of information retrieval research. *Proceedings of The IEEE - PIEEE*, 100:1444–1451, 05 2012.

[100] R. R. Schaller. Moore's law: Past, present, and future. *IEEE Spectr.*, 34(6):52–59, June 1997.

[101] J. Schmidhuber. Deep learning in neural networks: An overview. *CoRR*, abs/1404.7828, 2014.

[102] K. K. Schuler. *Verbnet: A Broad-coverage, Comprehensive Verb Lexicon*. PhD thesis, Philadelphia, PA, USA, 2005. AAI3179808.

[103] R. Schutt and C. O'Neil. *Doing Data Science: Straight Talk from the Frontline*. O'Reilly Media, Inc., 2013.

[104] Scikit-Learn Development Team. Scikit-learn. `https://scikit-learn.org/stable/`, 2020 (accessed 09.01.2020).

[105] C. Shorten and T. M. Khoshgoftaar. A survey on image data augmentation for deep learning. *Journal of Big Data*, 6(1):60, Jul 2019.

[106] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 529:484–503, 2016.

[107] H. Soltau, H. Liao, and H. Sak. Neural speech recognizer: Acoustic-to-word lstm model for large vocabulary speech recognition. *ArXiv e-prints*, 2016. https://arxiv.org/abs/1610.09975.

[108] SQLite Development Team. Sqlite. `https://www.sqlite.org/index.html`, 2020 (accessed 09.01.2020).

[109] Tenorflow Development Team. Tensorflow. `https://www.tensorflow.org/`, 2020 (accessed 09.01.2020).

[110] A. Torralba and A. A. Efros. Unbiased look at dataset bias. In *CVPR 2011*, pages 1521–1528, June 2011.

[111] A. M. TURING. I.—COMPUTING MACHINERY AND INTELLIGENCE. *Mind*, LIX(236):433–460, 10 1950.

[112] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin. Attention is all you need. *CoRR*, abs/1706.03762, 2017.

[113] P. Vincent, H. Larochelle, Y. Bengio, and P.-A. Manzagol. Extracting and composing robust features with denoising autoencoders. In *Proceedings of the 25th International Conference on Machine Learning*, ICML '08, pages 1096–1103, New York, NY, USA, 2008. ACM.

[114] J. Weizenbaum. Eliza a computer program for the study of natural language communication between man and machine. *Commun. ACM*, 9(1):36–45, Jan. 1966.

[115] Y. Wu, M. Schuster, Z. Chen, Q. V. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, K. Macherey, J. Klingner, A. Shah, M. Johnson, X. Liu, L. Kaiser, S. Gouws, Y. Kato, T. Kudo, H. Kazawa, K. Stevens, G. Kurian, N. Patil, W. Wang,

C. Young, J. Smith, J. Riesa, A. Rudnick, O. Vinyals, G. Corrado, M. Hughes, and J. Dean. Google's neural machine translation system: Bridging the gap between human and machine translation. *CoRR*, abs/1609.08144, 2016.

[116] W. Xiong, L. Wu, F. Alleva, J. Droppo, X. Huang, and A. Stolcke. The microsoft 2017 conversational speech recognition system. *CoRR*, abs/1708.06073, 2017.

[117] D. Yang, A. Lavie, C. Dyer, and E. H. Hovy. Humor recognition and humor anchor extraction. In *EMNLP*, 2015.

[118] W. Zaremba, I. Sutskever, and O. Vinyals. Recurrent neural network regularization. *CoRR*, abs/1409.2329, 2014.

[119] Y. Zhang. A better autoencoder for image: Convolutional autoencoder. In *ICONIP17-DCEC. Available online: http://users. cecs. anu. edu. au/Tom. Gedeon/-conf/ABCs2018/paper/ABCs2018_paper_58. pdf*, 2018.

[120] X. Zhu. Semi-supervised learning tutorial. 2007.