Technische Universiteit
Eindhoven
University of Technology

Department of Mathematics and Computer Science

Automatically responding to customers

Master Thesis

Rik Huijzer

Supervisors:
dr. N. Yakovets
dr. G.H.L. Fletcher
dr. J. Vanschoren

Eindhoven, January 2019

Abstract

Artificial intelligence caused a lot of progress in the natural language processing (NLP) field. It is interesting to see whether this can be used to automate customer support. NLP has evolved to contain many tasks. The intent classification tasks is the most interesting due to the following practical constraints. Intents are often used by conversational agents (advanced chatbots) and therefore should classify in real-time. Also, the training data typically contains only a few dozen examples.

Accuracy of classification differs per system. Higher accuracy means responding correctly to customer utterances more often. Each system claims to have the best accuracy scores when comparing their system to others. This could be due to the fact that the authors cherry pick datasets for their evaluation. To solve this a benchmarking tool is created. The tool is aimed on creating comparisons in such a way that users can easily run new or re-run existing evaluations. The code can easily be extended to allow comparison of more datasets and systems.

To improve the accuracy of intent classification systems deep learning architectures for NLP have been investigated. New accuracy records are set every few months for various NLP tasks. One of the most promising systems at the time of writing is considered. The system, Google BERT, uses context from both sides of some word to predict the meaning of the word. The main difference with other recent systems is that the context is used in all hidden layers of the network. The model has shown state-of-the-art results for eleven NLP tasks. An attempt is made to increase that number by applying the model to intent classification. This obtained significant increases in running time, but not in accuracy. A second attempt trained the system jointly on intent classification and named-entity recognition. Here information from the former task is used while prediction the latter and vice versa. It is shown that joint training with BERT is feasible, and can be used to lower training time when compared to separate training of BERT. Future work is needed to see whether accuracy improvements are significant.

Preface

Please write all your preface text here. If you do so, don't forget to thank your supervisor, other committee members, your family, colleagues etc.

14 January 2019

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

This master thesis is the final report of the graduation project for the Computer Science and Engineering master at Eindhoven University of Techonology (TU/e). The project is carried out at Devhouse Spindle in Groningen.

1.1 Thesis context

Spindle is interested in automatically responding to customer questions. It is expected that automated responses to text are feasible by recent examples in artificial intelligence. Smartphones include speech recognition, allowing users to tell the phone what they want. Obtaining information about the weather or the age of a specific president by talking to a device is possible. Self-driving cars are constantly in the news. They are able to drive fully autonomous in certain regions. Regions are typically in America, since the country has wider roads and technology companies have offices there. Google AlphaGo has been able to beat the best Go players in the world. At the same time another Google team is working on Duplex. The goal of Duplex is to fill in missing information from their sites by calling people and asking for the information. Demonstrations by Google show that it is difficult for unsuspecting humans to tell the difference between Duplex and a real human.

Remarkable about the context is the speed in which the field evolves. Learning about some scientific topic means picking up a recent book or review paper. For example, Stanford provides a 'Deep Learning for Natural Language Processing' course. This course is given bij Christopher Manning who has been active in natural language processing for many years and is highly cited in the field. When comparing the material of the 'Winter 2018' version with the 'Winter 2017' we see many changes. Attention and transformer models [40] now get three lectures instead of one. This could be a coincidence where last year has been a particularly interesting one. Looking at the years before it does not seem to be so. A recent review paper for the same field [47] still see lots of potential. They expect better models for unlabelled data, reinforcement learning, zero-shot learning and network memory enrichment via knowledge base.

The fast pace implies that this thesis is in certain aspects unconventional. An atypical high amount of references are to arXiv publications, blogposts and websites. These sources are regarded with more than usual skepticism and it is tried to extract only empirically validated information. This would not seem to work for fields like theoretical physics. Arguably the benefit of deep learning is that everyone can reproduce and improve results given some time and a (cloud) computer.

1.2 Problem description

The field working on interpreting text is natural language processing. In this field many tasks seem interesting. For example, semantic text similarity could be used to find sentences having the same meaning. One could apply this in an application to classify user input as being a duplicate.

Duplicates can automatically be answered by using some template. Another interesting task is question answering. This field is still maturing. State-of-the-art systems focus on specifying the answer for some question given a paragraph of text or given a large pre-structured knowledge base. Note the word 'paragraph'. IBM Watson is an example where the latter is used for its responses in the Jeopardy quiz [14].

The intent classification tasks is chosen due to the following practical constraints. Intents are often used by conversational agents (chatbots) and therefore able to classify in real-time. Also, the training data typically contains only a few dozen examples. Another benefit of these systems is that they are being applied in industry (empirical evidence). The IBM sales department (`https://www.ibm.com`) claims that companies like Autodesk are able to respond to 60% of their customer chats automatically.

Various organisations claim to have obtained the best accuracy numbers for text classification tasks. Snips show that they outperform the competition by a large margin (`https://medium.com/snips-ai/2b8ddcf9fb19`) on 2 June 2017. The competition consists of api.ai, Wit.ai, Luis.ai and Amazon Alexa. Their small benchmark tests the systems against 70 queries per intent on their own dataset. Snips claim to score 79% accuracy, while the second best scores 73%. Also, via sentence examples Snips show that some named-entities are classified incorrectly by systems other than Snips. A comparison by Braun et al. [3] looks at LUIS, Watson Conversation, API.ai, wit.ai, Amazon Lex and Rasa. Three datasets are created and published by the authors. Code to re-run or extend the benchmarks has not been included. DeepPavlov [4] reports another high score for intent classification. It is based on the Snips dataset and compared against api.ai, Watson Conversation, Microsoft LUIS, Wit.ai, Snips.ai, Recast.ai and Amazon Lex. Botfuel show they are 'on par' with the competition (`https://medium.com/botfuel/eb8684a1e55f`). This is based on runs on the same datasets as Braun et al. [3]. Using micro averaging the Botfuel is one procent lower than Watson, equals LUIS and outperforms DialogFlow, Rasa, Snips and Recast. The results for Rasa match the results provided in the paper. This means that Botfuel has compared their system against an old version of Rasa.

The main problem with these claims is that the numbers appear convincing to a machine learning layman. It could be that some organisation has cherry picked a dataset for their system or omitted systems which outperformed theirs. The only way for users to determine what system is the most accurate for their data is to trust these claims or write their own benchmark. This gives rise to the following research question.

RQ1. Can an open-source benchmarking tool for natural language understanding systems and services be created?

Another interesting problem from an academic point of view is increasing accuracy. Natural language understanding is far from solved. Due to the probabilistic nature of machine learning one should not expect perfect results. However, accuracies one some computer vision tasks are above 99%, which cannot be said for natural language understanding. The second research question is as follows.

RQ2. Can the classification accuracy for natural language understanding be increased?

The focus in this question lies on English datasets. For Spindle Dutch datasets would be more useful, however baselines for Dutch datasets do not exist. Also, knowledge obtained from answering RQ2 is expected to generalise to any language. The first and second research question are discussed respectively in chapter 3 and 4.

1.3 Project goal

Answering RQ1 means writing code. Even if the answer is negative, it will have provided a baseline to use when answering RQ2. The aim of the software is to be used by others, so it should be easy

to run. Software extensions should also be possible. The goal related to the first research question is as follows.

RG1. Develop an open-source reproducible tool for benchmarking of natural language understanding systems.

To not let our guard down the goal related to the second research question is stated ambitiously.

RG2. Improve the accuracy of natural language understanding systems.

Chapter 2

Preliminaries

Both research questions are related to natural language understanding, which is a subfield of natural language processing. Natural language processing is introduced in section 2.1. Deep learning forms the basis for state-of-the-art systems in the field and is introduced in section 2.2.

2.1 Natural language processing

Natural language processing (NLP) aims to read text and extract meaningful information from it. One deviation from this definition is natural language generation, where text is generated. Note that this differs from interpretation of text and then returning some pre-defined piece of text. The difficulty in NLP is that humans have evolved to use a concise way of communication which makes use of knowledge from the world and context. For example, the response to "I want more money." depends on whether the sentence is uttered by a child or employee. A perfect NLP system would need artificial general intelligence or is 'AI-complete'. Current artificial intelligence systems "demonstrate intelligence in only one or another specialised area" [30], so called 'narrow AI'. This makes it an field which is far from solved.

The field is divided in tasks. Some well-known tasks are:

- Translating texts or **machine translation**.
- **Question answering** which is NLP on question sentences only.
- Classifying words or parts of sentences is done by **part-of-speech tagging** and **named-entity recognition**.
- **Optical character recognition** can use NLP knowledge to improve accuracy.
- Finding co-references like "The house is white, and it is located on a hill." is done using **coreference resolution**.
- NLP is not limited to text, because it includes **speech recognition** which transforms speech to text.

2.1.1 Language model

To be able to explain more complex neural network architectures we first take a look at a simple language model. Language models try to capture the grammar of a language. Capturing grammar can be done by using probabilities for sentences or probabilities of upcoming words given a part of a sentence. It is based on the assumption that grammatically correct sentences occur more often than incorrect sentences. The following three tasks demonstrate the usefulness of probabilities.

Spell correction	$P(\text{my car broke}) > P(\text{my car boke})$
Machine translation	$P(\text{the green house}) > P(\text{the house green})$
Speech recognition	$P(\text{the red car}) > P(\text{she read ar})$

A simple approach is to use counters to calculate the sentence probabilities. This makes use of the chain rule. Let W denote a sentence, or equivalently, sequence of words. For the probability of the sentence we have $P(W) = P(w_1, w_2, \ldots, w_n)$. The probability of the upcoming word w_i is $P(w_i) = P(w_i \mid w_1, w_2, \ldots, w_{i-1})$. Using the chain rule we can, for three variables, state that $P(w_3, w_2, w_1) = P(w_3 \mid w_2, w_1) \cdot P(w_2 \mid w_1) \cdot P(w_1)$. This pattern scales to any number of variables. To get the probability for the sentence 'the car broke' we rewrite it as follows.

$$P(\text{the car broke}) = P(\text{the}) \cdot P(\text{car} \mid \text{the}) \cdot P(\text{broke} \mid \text{the car})$$

Each term can be rewritten to a combination of counters, for example: $P(\text{broke} \mid \text{the car}) =$ COUNT(the car) / COUNT(broke). This does not scale well. The counters for each word and each pair of words are feasible. However, when doing this for long sentences the number of counters to keep track of becomes too large.

To reduce the number of counters an approximation defined by Markov is used. Markov states that only looking at a fixed number of previous words gives an approximation. Considering only one previous word for our example we get the following.

$$P(\text{broke} \mid \text{the car}) \approx P(\text{broke} \mid \text{car})$$

This is called the bigram model. When using this to generate sentences it becomes clear that bigrams do not have enough information. Take the generated sentence "I cannot betray a trust of them." [21]. Each pair of sequential words is correct, while the sentence as a whole is not. This simplification of looking at a fixed number of previous words is called n-grams. Although n-grams offer good performance for certain cases they are in practise not able to capture long-distance dependencies in texts.

2.1.2 Machine translation

Machine translation became known to the public by introduction of Google Translate in 2006. The system was statistical (as explained in section 2.1.1) and not rule-based. Rule-based means formulating linguistic rules which is "a difficult job and requires a linguistically trained staff" [36]. In an attempt to visualise the progress made in the field we consider the 'one sentence benchmark' by Manning and Socher [25]. This benchmark contains one Chinese to English example which is compared with Google Translate output. The correct translation for the example is:

In 1519, six hundred Spaniards landed in Mexico to conquer the Aztec Empire with a population of a few million. They lost two thirds of their soldiers in the first clash.

Google Translate returned the following translations.

2009 1519 600 Spaniards landed in Mexico, millions of people to conquer the Aztec empire, the first two-thirds of soldiers against their loss.

2011 1519 600 Spaniards landed in Mexico, millions of people to conquer the Aztec empire, the initial loss of soldiers, two thirds of their encounters.

2013 1519 600 Spaniards landed in Mexico to conquer the Aztec empire, hundreds of millions of people, the initial confrontation loss of soldiers two-thirds.

2014-2016 1519 600 Spaniards landed in Mexico, millions of people to conquer the Aztec empire, the first two-thirds of the loss of soldiers they clash.

2017 In 1519, 600 Spaniards landed in Mexico, to conquer the millions of people of the Aztec empire, the first confrontation they killed two-thirds.

One important concept in machine translation is alignment. Alignment refers to the fact that words in different languages tend to be aligned. Consider the sentences

"well i think if we can make it at eight on both days"

and

"ja ich denke wenn wir das hinkriegen an beiden tagen acht uhr".

The first five words of the sentences are perfectly aligned. The last five words of the sentences are not. Alignment is visualised in figure 2.1. Non-perfect alignment can also be observed from the fact that the number of words in English sentence is higher.

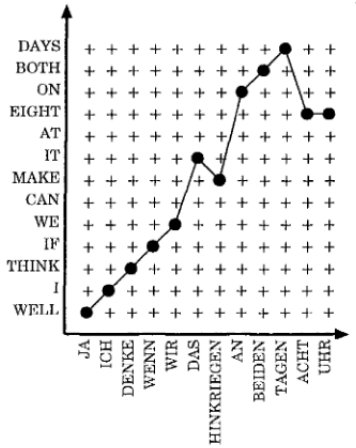

Figure 2.1: Word alignment for a German-English sentence pair [41, Figure 1].

2.2 Deep learning

As with many computer science subfields deep learning has outperformed manual feature engineering on many NLP tasks. For the last few years the best performing NLP systems have been neural networks. This section will provide a basic overview of the most important model architectures. Importance is decided by seeing what is mentioned often when reading about NLP and looking at the state-of-the-art at the time of writing. So, completeness of this discussion is not guaranteed.

2.2.1 word2vec

2.2.2 glove

2.2.3 Vanishing gradient problem

The vanishing gradient problem has for a long time halted the progress of neural networks with many layers, also known as deep neural networks. The problem relates to vanishing and exploding updates to weights in the earlier layers. These extreme updates are caused by the backpropagation. Consider some weight w which is at the front of some network. Lets say this weight is updated by the following partial derivative $w = d_1 \cdot d_2 \cdot \ldots \cdot d_i$. Here i denotes the number of layers in the network. These partial derivatives d_x can become small ($0 \leq d_x < 1$) or large ($1 \ll d_x$). Typically the learning rate has an order of magnitude of 1e-3. When one derivative is small the weight update when multiplied by learning rate might become very small. Conversely when one derivative is big the update might become very big. Since the weight is at the front of a deep

network i is big. Thus, the chance that weights in the front vanish or explode increase by increment of i. Effectively the problem causes layers which are have a long distance from the prediction layer to stop learning.

2.2.4 Recurrent neural networks

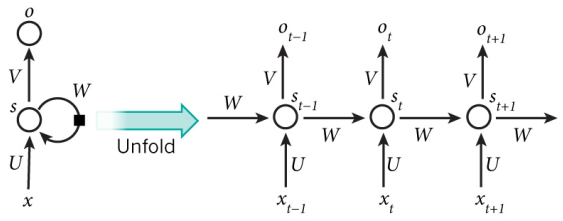

Figure 2.2: Recurrent neural network [22, Figure 5].

A recurrent neural network (RNN) is an extension on neural networks which allows the network to have memory. Simply put the network remembers a summary of information from all previous states. A common way to visualize this is depicted in figure 2.2. On the right side the model is unfolded in time. The unfolded representation shows the state of the network and the flow of information for three consecutive points in time. To access this history the neural network in current state s_t obtains a 'summary' from the previous state s_{t-1}. Suppose we are training a RNN in an unsupervised way. The model trains on real-world texts. For each word w_i it is asked to predict w_i based only on previous words $w_1, w_2, \ldots, w_{i-1}$. In the image w_i is denoted as input x_i. After predicting the prediction is compared to the correct word, if these are not equal the loss is backpropagated. The backpropagation is then able to 'change history' to improve prediction accuracy.

The benefit of this architecture over n-grams is that the information is compressed inside the neural network. Also, there is a certain sense of importance since the weights are not uniformly updated for all previous states (words). Take, for example, 'the car, which is green, broke'. For this sentence the word 'broke' can more easily be predicted based on 'the car' than on 'which is green'.

In practice RNNs are not using the complete history. The cause for this are vanishing and exploding gradients. Solving explosions can be done by putting a threshold on the updates [26]. The vanishing gradients are harder to solve because these updates do not 'jump out'. In practise gradients still vanish and simple RNNs become like 7-grams for most words [25]. This is clearly not enough.

2.2.5 Gated recurrent units

Basic RNNs do not yet provide a way to translate languages. Machine translation requires to convert some sentence from a source language to a target language. To this end a RNN encoder-decoder has been proposed by Cho et al. [7].

Similar to the basic RNN the decoder at some time has access to only the previous state, see figure 2.3. The encoder takes the source sentence of variable length T and maps it to a fixed length vector. The decoder in turn maps the fixed length vector to the target sentence of length T'. To do this the network reads all words $x_1, x_2, \ldots, x_T$ until an end of line token is received. At that point the entire sentence is captured in the hidden state C. The decoder starts generating words $y_1, y_2, \ldots, y_{T'}$ based on the previous decoder states and C. This is visualised by the arrows. The authors of the paper recognized that this approach has the same limitations as the basic RNN. Typically words in translation sentence pairs are somewhat aligned, as described in subsection 2.1.2. For example, when generating the first word y_1 the algorithm mainly needs

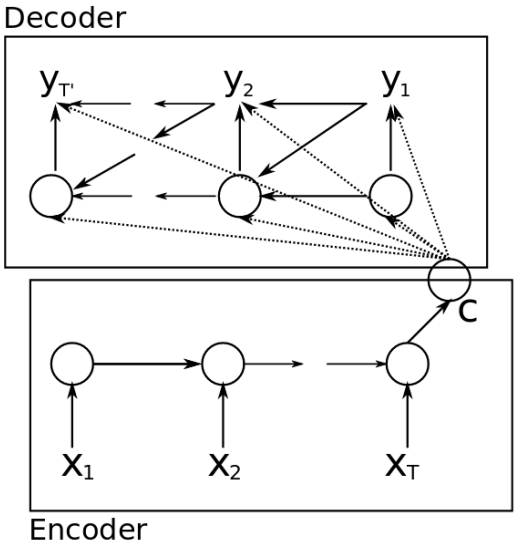

Figure 2.3: RNN Encoder-decoder [7, Figure 1].

info from x_1, but has more recently seen the next words in the sequence $x_2, x_3, \cdots, x_{T'}$. Vanishing gradients will cause the network to forget its far history, so this methods does not work well for long sentences. To solve this the authors have also introduced gates to RNNs.

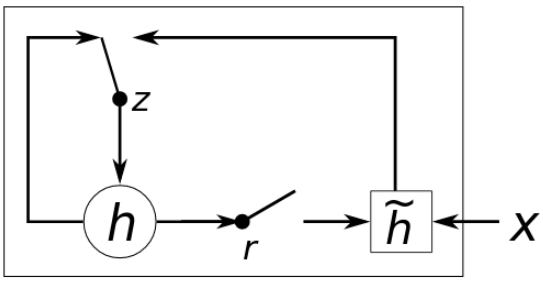

Figure 2.4: RNN Encoder-decoder [7, Figure 2].

Gated recurrent units have gates which automatically learn to open and close for some hidden state h. The update gate z whether backpropagation should be applied to h. The reset gate r decides whether the p

TODO: Really draw this out. GRU intuition slide is nice. It explains that if reset is close to 0 the previous hidden state is ignored.

2.2.6 Long short term memory networks

The encoder goes through different states until the end of line is reached. After this the decoder uses the final state to generate the next sentence (for example, a translated sentence). Seq2Seq? See also CS224n.

2.2.7 Attention

See also CS224n including 'advanced attention' slides.

One drawback of recurrent neural networks is that the decoder has all information from the final state of the encoder. A problem with this can be explained by looking at machine translation. When translating one will find that similar words often appear at similar places in sentences. TODO: Insert alignment image and paper from slides. This is called alignment. When generating the n-th word during decoding information we generally need mainly information from the n-th word which occurred during encoding. It makes more sense to focus the neural network *attention* on the part of the sentence we actually need the most. To this end the neural net learns to choose what hidden states are important. TODO: Really add image here just like slides. The important states get a higher weight which yield a context vector. TODO: Way too short, need to explain better.

2.2.8 Bidirectional recurrent neural networks

All recurrent architectures described above use only the information on the left of some word to predict it. Take for example the following sentences containing the polyseme 'drinking'.

Peter was drinking after a workout.

Peter was drinking excessively.

The meaning of the word 'drinking' changes after reading the next words in the sentence. To take this into account bidirectional recurrent neural networks (BRNN) have been developed by Schuster and Paliwal [34]. A BRNN contains two separate RNNs as depicted in figure 2.5. The paper only considers RNNs, but the method can be applied to gated recurrent models as well. One RNN goes through elements of the sequence from left to right and the other in the reverse direction. Training can be done by showing the network all words except for one. Both networks learn to predict the next word given a sequence of words. Calculating the loss is done by taking the average of the predictions of both RNNs. To reduce required computational power one simplification is used. Suppose we want to learn from the word at location k, w_k. A solution would be to get all the way to states s_{k-1} and s'_{k+1} to predict w_k. Then we update the weights and, assuming we go forward, now want to learn from w_{k+1}. The RNN in state s_{k-1} takes one step forward, but the RNN in state s'_{k+1} has to restart from the last word in the sequence. To solve this both RNNs make one prediction for each word and the answers of both for the entire sequence are used to update the weights.

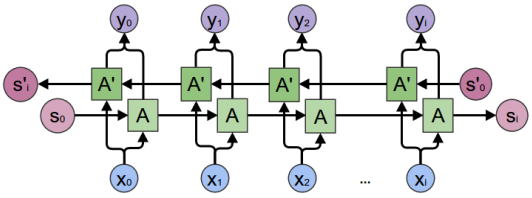

Figure 2.5: Bidirectional RNN [29].

2.2.9 Convolutional neural networks

A review by Young et al. [47] argues that CNNs are suited for specific NLP tasks. However, when data is scarce they are not effective. Foundational problems with CNNs are that they are unable to model long-distance contextual information and preserving sequential order. The former means that CNNs are not well suited for question answering tasks. Interestingly, this paper mentions "BERT surpass ELMo to establish state-of-the-art in multiple tasks".

2.2.10 Transformers

The main issue in the recurrent approaches is that distant information needs to pass through all the intermediate states. In the basic RNN for each state all the information is updated, causing less recent information to gradually disappear. The GRU and LSTM reduce this problem by being more selective when viewing or changing information. In *attention is all you need* [40] a new approach is presented. Instead of making a prediction on the previous state the network is allowed to make a prediction based on a large number of previous inputs. For example, suppose we are translating 'impounded' in the following sentence pair from the WMT'14 English-German dataset.

The motorcycle was seized and impounded for three months.

Das Motorrad wurde sichergestellt und für drei Monate beschlagnahmt.

Suppose the system has correctly predicted the German sentence up to and including 'Monate'. The next step is to predict 'beschlagnahmt'. To do this the system needs mainly information about the word 'impounded'. Gated recurrent architectures learn to look at the previous state in such a way that the attention is focused on 'impounded'. This requires the information of the word to not been overwritten during execution. Transformers evade this overwriting problem by allowing the system to see all d previous words, where d is 1024 for the biggest model. The only thing the transformer then needs to learn is where to focus its attention. The information of all the previous words is stored in an array of word vectors (a tensor). To apply focus to parts of this tensor the model learns to put a mask over the tensor. In the mask zeroes correspond to hidden items. One drawback is the required computational power. Suppose we only need one word from the d previous words. The mask will hide $d-1$ words. This still requires to multiply the masked word vectors by zero. Google argues that this is not really an issue since matrix multiplication code is highly optimized and graphic processing units (GPUs) exist. So, the model can relate any dependency in constant time when the range is lower than d. This in contrast to recurrent layers which are in linear time. When the sequence length is greater than d computations will require more than linear time.

Another benefit of the transformers are that self-attention visualisations can more easily be done than in recurrent architectures. By self-attention the authors refer to attention that is used to generate context aware word representations. An example of a tranformer model correctly applying coreference resolution is presented shown in figure 2.6.

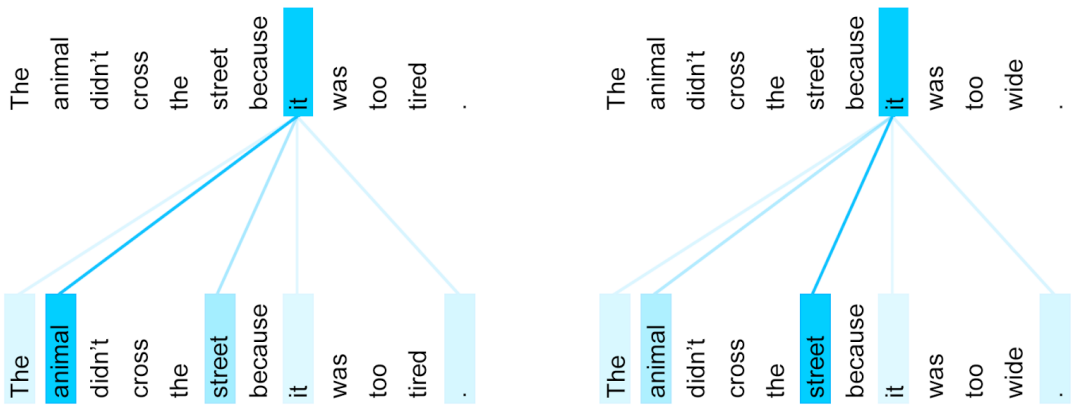

Figure 2.6: Encoder self-attention distribution [38].

2.2.11 ELMo

Word embeddings generated by systems such as Word2vec and GloVe do not take context into account when determining word representations. For ELMo "each token is assigned a representation that is a function of the entire input sentence" [32]. This is done by using a bidirectional LSTM. To improve accuracy further the authors advise to use the deep internals of the network. These internals can be used for downstream models. For example, the authors show that word sense disambiguation tasks are captured by higher level LSTM states. Part-of-speech tagging or named entity recognition are captured by lower level states. ELMo is not the first system to use context, but was obtaining state-of-the-art empirical results on multiple non-trivial tasks at the time of publication. One reason for this is that the system is character based. Word based systems cannot generate an embedding for a word they have not seen during training (out-of-vocabulary tokens). In character based systems morphological clues can be used to guess the meaning of the out-of-vocabulary words. Other reasons seem to be that the system is more general, having a better set-up, and trained on more data. The model is integrated into the AllenNLP open-source NLP library created by Gardner et al. [13].

2.2.12 BERT

WordPiece tokenization, as published by Wu et al. [46], is used. They argue that the tokenization "provides a good balance between the flexibility of 'character'-delimited models and the efficiency of 'word'-delimited models".

Chapter 3

Natural language understanding

This chapter aims to answer the first research question. The goal is to create a reproducible benchmark tool, simply called BENCH. Datasets used by the benchmark tool are described in section 3.1. The benchmarked systems are described in section 3.2. Running the benchmarks on various datasets and systems resulted in scores, see section 3.3. It has been found that BENCH is impractical, described in section 3.4. Notes on using the tool and a high-level overview is presented in appendix A.

3.1 Datasets

Trained models are considered black boxes at the time of writing. To verify performance of a model data is required. This is fed to the model and results are measured.

3.1.1 Format

The dataset format needs to be able to specify a sentence annotation and subsentence or token annotations. One often used dataset for token classification is CoNLL-2003 [37]. It uses the NER task definition as described by Chinchor et al. [6]. The definition uses tags to specify entities, for example:

```
<B_ENAMEX TYPE="PERSON">bill<E_ENAMEX> and <B_ENAMEX TYPE="PERSON">
susan jones<E_ENAMEX>
```

Note that this limits the text since angled brackets ('<', '>') cannot be used without escaping the brackets. A less verbose and non text constraining annotation standard is the BIO2 annotation standard. The origin is unclear, but adaptation is done by at least Stanford as seen in Glove [31]. Here sentences are annotated as follows.

```
I B-Person
and O
John B-Person
Doe I-Person
worked O
yesterday B-Date
. O
```

where B, I and O respectively mean begin, intermediate and 'empty' annotation. Note that other annotations, like part-of-speech tagging, is possible by adding another column of tokens. A benefit of this annotation is that measuring performance can be done by looking at each token

annotation separately. In a tag example like above it is unclear how cases where the classifier is partly correct should be solved. Suppose only 'susan' is classified as person and not her last name 'jones'. Metrics now have to decide how to handle this partially correct situation. In the BIO2 annotation standard token classifications can only be correct or incorrect. One drawback is that the standard is not easy to read for humans. A more readable format is the Rasa Markdown format. Here it constraints the text by using square ('[', ']') and round bracket ('(', ')') symbols to denote annotations. For example:

```
[I]
    (person) and [John Doe](person) worked [yesterday](date).
```

Unlike BIO2 this standard does not easily allow to also specify other annotations, for example part of speech tagging. One could argue that by the readability of a Markdown format and the versatility of the BIO standard show that there is no single 'best' approach.

A combination of sentence annotations and token annotations is not supported by standard described above. For BIO one could track the sentence annotations in a separate file or put it before or after the sentence. The former adds duplicate information while the latter is not compatible with the standard. For Rasa one could change it to a table and put the Markdown and sentence annotation in separate columns. This is very readable and compact, but means transforming the token annotation for easier validation as described above. Datasets which combine sentence annotations with token annotations seem to take yet another approach. They use json to store any information they have. These formats should allow for easier parsing, but can not easily be read by humans. For example one dataset annotates entities as as follows.

```
"entities": [{
"entity": "StationDest",
"start": 4,
"stop": 4,
"text": "marienplatz"
}]
```

This entity belongs to the sentence "i want to go marienplatz". 'start' and 'stop' here assume the sentence to be tokenized using a WordPunctTokenizer having regexp `\w+|[^\w\s]+`. Drawbacks are that the entity text is duplicated and human readers need to tokenize the sentence manually for verification.

3.1.2 Available datasets

Three datasets for intent classification and entity recognition are created by Braun et al. [3] and made publicly available. The paper and its Github version use different names, this document will stick to WebApplications, AskUbuntu and Chatbot. WebApplications and AskUbuntu are obtained by pulling questions from StackExchange (https://stackexchange.com). For example, they respectively contain "How can I delete my [Hunch](WebService) account?" and "How to install a [Brother MFC-5890CN](Printer) network printer?". Intents for these examples are 'Delete Account' and 'Setup Printer'. StackExchange datasets are labeled using Amazon Mechanical Turk. The Chatbot dataset is based on a Telegram chatbot in production use. This dataset contains sentences like "when is the [next](Criterion) [train](Vehicle) in [muncher freiheit](StationStart)?" having intent 'DepartureTime'. Labeling is done by the authors of the paper.

Snips (https://snips.ai) is a company which provides software to locally run a voice assistant. They have shared some of the data generated by their users as well as results for their benchmarks on Github (https://github.com/snipsco/nlu-benchmark). The incentive for sharing these datasets seems to be showing that their system performs better than other systems. Two datasets have been published by Snips. The thesis has only used the 2017 version and not the

dataset	train	test	intents	entities
WebApplications	30	54	7	1
AskUbuntu	53	109	4	3
Chatbot	100	106	2	5
Snips2017	2100	700	7	unknown

Table 3.1: Number of labeled train and test sentences and unique intents and entities per dataset

2016 version. The 2017 version will from now on be referred to as Snips2017. Sentences in this dataset are typically short. They utter some command to the system, for example for the intent 'PlayMusic': "i want to listen to [Say It Again](track) by [Blackstratblues](artist)".

More information about both corpora can be found in table 3.1. In this table 'None' is not counted as an intent. The reason for specifying this is that falling back to *null* or some intent during unsure predictions result in different scores for most metrics. F1 score calculations, for example, do not ignore *nulls* or 'None', but instead consider them as a separate group. Information about the unique number of entities for Snips2017 is not specified by the dataset authors.

3.2 Systems

3.2.1 Rasa

Rasa (https://rasa.com/) is an open-source system allowing users to build conversational agents. The systems consists of two parts, namely RASA_NLU and RASA_CORE. The former classifies sentences and sub-sentences. To train the system users can specify (hierarchical) intents, synonyms and regexes. Hierarchical intents is a recent addition which allows the system to extract multiple intents from a sentence. For example, it can extract 'hi+book_ticket' from "Good morning. Can I order a ticket to London please?". The system is actively used in production. As a result the code is well documented and stable.

RASA_CORE aims to handle dialogue management. This is an extension on the classifiers of RASA_NLU which aims to understand text in context. Also, it can be used to specify conversation flow. This part remains one of the most difficult problems for conversational agents. Humans tend to switch rapidly between topics in conversations. For example, suppose one ticket order conversation flow contains six questions to be answered by the customer. Customers expect to be able to switch topic during each one of these questions and then return to the flow. Enabling this behaviour via state machines or flowcharts is cumbersome, because the number of transitions tends to grow quickly. One of the Rasa solutions is applying machine learning to let developers train dialog flows interactively.

Rasa allows the system to be used as API and 'Python API'. The Python API is the most efficient. Here users install RASA_NLU in their programming environment and call functions directly. Depending on the used back-end a selection of dependencies have to be installed. Since Rasa is called from Python it is not able to maintain a state. The regular API advises to spin up a Docker container. This is less efficient, but more modular. Containers are published to Docker Hub by Rasa. Users can pull these for free and use the newest stable configuration of choice.

Configurations are defined as a pipeline. Pipelines specify what components should be applied to sentences and in what order. Typical pipelines contain at least a tokenizer followed by some classifier. Pipelines are meant to be modified easily and specified in yaml. In practise default pipelines often suffice for end-users. A back-end refers to the used intent classifier in some pipeline, for example 'tensorflow'. At the time of writing three back-ends are offered by Rasa, namely RASA-MITIE, RASA-SPACY and RASA-TENSORFLOW. RASA-MITIE is the oldest and depreciated. Training MITIE (https://github.com/mit-nlp/MITIE) takes at least a few minutes for small datasets. This is caused by the fact that it is tuning hyperparameters during training. On two computers used for this thesis the Docker Hub image occasionally hangs on various datasets. RASA-SPACY is the successor of MITIE and, unsurprisingly, based on spaCy (https://spacy.io). In 2015

spaCy was in the top 1% for accuracy and the fastest syntactic parser [8]. spaCy (and by that RASA-SPACY) requires a language model to parse text. It includes seven language models for which English, Spanish and French include vectors. The multilingual model supports only named entities. Unlike the other two back-ends RASA-TENSORFLOW is not based on a pre-trained language model. This is like classifying sentences in an unfamiliar language (say Chinese) after only seeing some examples. Rasa advises to use this back-end when training data contains more than 1000 training examples. The benefit is that this back-end is language independent and can handle domain specific data and hierarchical intents.

3.2.2 DeepPavlov

DeepPavlov [4] is similar to Rasa. Unlike Rasa, DeepPavlov aims to aid researchers in development of new techniques for conversational agents. Being a newer system than Rasa and aimed at researchers the system is not yet production ready. The system does only provide a Python API, requiring Python 3.6. One claimed benefit of the system is that they do not export machine learning components from other systems. Another reason why the system is not well suited for production is that pipelines can download information. This means that a generic Docker needs to download many megabytes of data for each time the Docker is started. Manually defining new Dockers holding this information is possible, but does require some knowledge about Docker and some time to set it up. For users who want to use few training examples pre-trained models are necessary. DeepPavlov by default includes DSTC 2, Wikipedia, Reddit, RuWiki+Lenta and ELMo embeddings. Its hard to tell what model should be chosen for some use-case.

3.2.3 Cloud services

Cloud service providers and various small companies (start-ups) provide APIs for conversational agents. Functionality differs per provider, but in the basis they all over the same features. Naming conventions do not seem to exists. For example, Rasa calls intent classification training examples *utterances*, while Google Dialogflow calls them *training phrases*. Via the web interface or API examples can be sent to a server and the system can be configured. Configurations specify the utterances, dialog flows, how to classify entities (used for slot filling) and input language. At some point the server will use the provided examples to train the model. This takes a few seconds. This makes sense, since users do not want to wait and computations cost money. Settings for the dialogs and APIs are located somewhere in the complete cloud offerings. An extension on the intent classification and slot filling described above is using knowledge bases. When looking at IBM Watson a document needs to be uploaded and annotated by humans. This is an example of a structured knowledge base.

In the period that IBM Watson won the Jeopardy quiz a lot of math and reasoning was required to create NLP systems. Nowadays, training a competitive neural network for natural language processing is relatively easy. It takes a PhD candidate a few months [25]. This results in a lot of companies providing natural language processing services. An in-depth analysis of all services is left out. The following is a non-exhaustive list. Some offer full conversational agent capabilities, while others focus on natural language understanding.

Watson Assistant (`https://www.ibm.com`) is a conversational agent by IBM.

Dialogflow (`https://dialogflow.com`) is a conversational agent by Google.

Lex (`https://aws.amazon.com/lex`) is a variant created by Amazon.

wit.ai (`https://wit.ai`) can be used for chatbots and is acquired by Facebook. The system is free to use, but Wit is allowed to use the data sent to their servers.

Deep Text (`https://deeptext.ir`) provides sentiment analysis, text classification, named-entity recognition and more.

Lexalytics (https://www.lexalytics.com) provides categorization, named entity recognition, sentiment analysis and more.

Pat (https://pat.ai) has as goal to humanize AI and provides some conversational agent services.

kore.ai (https://kore.ai) focus lies on intent classification and entity extraction with as goal to replace graphical user interfaces with chatbots.

Sixteen more are listed by Dale [9].

3.3 Benchmark results

This section will present the benchmark results for intent classification using micro f1 score. An explanation for the differences in averages for the f1 score is presented in section 3.4.2. Here micro f1 scores are used to allow comparing results with Braun et al. [3]. The results are listed in table 3.2.

System	Source	AskUbuntu	Chatbot	WebApplications
Rasa:0.5-mitie	see section D.1	0.862	0.981	0.746
Microsoft LUIS	see section D.3	0.899	0.981	0.814
Watson Conversation (2017)	see section D.2	0.917	0.972	0.831
Rasa:0.13.7-mitie	BENCH	0.881		0.763
Rasa:0.13.8-spacy	BENCH	0.853	0.981	0.627
Watson Conversation (2018)	BENCH	0.881	0.934	0.831

Table 3.2: Micro f1 scores for intent classification. One score is missing due to a bug in BENCH.

The paper remarks that "For our two corpora, LUIS showed the best results, however, the open source alternative RASA could achieve similar results." When considering only intents this does not hold. Watson Conversation has very similar results, and in fact slightly higher scores on two out of three datasets. The MITIE back-end outperforms the spaCy back-end in terms of accuracy. This would not support the choice of Rasa to depreciate MITIE. It is expected to be caused by the facts that training MITIE takes more time than spaCy and MITIE tends to freeze during training. Interesting to see is that the accuracy for Watson Conversation has dropped. The cause can only be guessed since IBM does not provide information about the Watson back-end. It could be that the calculations for BENCH and the paper differ. Alternatively it could be that the back-end for Watson has changed. The datasets under consideration are small, so it might be that Watson has chosen a back-end better suited for large datasets. Note that IBM is aimed at large companies. These companies have the resources for creating lots of training examples.

3.4 Issues

3.4.1 Benchmarking system

Larry Page (Google founder) advises people to "fail fast". By this he means that one has to try things and if some plan does not work one has to quickly realise that and move on. This appears to be sound advise for business and research alike. With that in mind it is time to concede defeat for the benchmarking system. It is found to be highly impractical.

Datasets, for example, are often not publicly available. This is probably caused by the sensitive nature of natural language. Dependencies increase the maintenance costs. The dependencies are in the form of application programming interfaces (APIs). APIs are used by software and will therefore not change often. Eventually they will do, ergo eventually the benchmarking software needs to be updated. Another problem is that the services which offer APIs are not open. When

the benchmark is extended to include all systems then all API keys need to be added as well. The owner of the benchmarking tool can decide to offer paid keys or let users set keys manually. The latter requires users to have an account for each service. A closed-source solution is Intento (https://inten.to). One can send data to the site via their API and they will run a benchmark on various services for a given task. Their 'catalog' contains machine translation, intent detection, sentiment analysis, text classification, dictionaries, image tagging, optical character recognition and speech-to-text. The final issue with BENCH is as follows. When choosing a system not only the performance matters. One reason for this is that accuracies are volatile. If companies would base their decision solely on 'the highest accuracy' then they would need to change system each month. Since companies cannot spend their time constantly switching systems they should spend their time on other factors. These factors can include pricing, privacy (whether open-source) and in-house API preferences.

3.4.2 Methodology

Creating a benchmarking tool has resulted in more insight into intent classification. This helped in identifying issues in the methodology proposed by Braun et al. [3]. The created datasets and many presented ideas are useful, however some improvements are possible. As discussed in section 3.1.2 falling back to 'None' or a random intent changes f1 score. In the paper Chatbot does not have a 'None' intent, while WebApplications and AskUbuntu do. Furthermore, drawn conclusions about some system being more accurate than others seems insubstantial. The conclusion that accuracy of some system depends on the domain seems convincing, but is poorly grounded. Reason for this is that both conclusions are based on the f1 score.

In this paper the f1 score is calculated using micro f1 score. Such a score does not take classes of different size, so called class imbalances, into account. This is combined with a situation where intents and entities are given the same weight. For WebApplications there are in total 74 labeled intents and 151 labeled entities. AskUbuntu contains 128 labeled intents and 123 labeled entities. So, when using micro f1 on AskUbuntu the score is based somewhat equally on intents and entities. For WebApplications the score is based for about one thirds on intents and two thirds on entities. This could mean that some system has scored significantly better that others simply because it labels entities in WebApplications particularly well. Another reason which makes this score odd is that users interested in either intent or entity classification are not well informed. Better seems to be using weighted f1. Here the f1 score for each class is multiplied (weighted) by the number of elements in that particular class. Imbalances can also be handled by calculating a macro f1 score, but this is more computational expensive. Rasa, for example, uses the weighted average in their evaluation according to (https://github.com/RasaHQ/rasa_nlu/blob/master/rasa_nlu/evaluate.py).

Another reason to distrust presented f1 scores is the probabilistic nature of neural networks. Although inference (classification) is deterministic, training is not. During training models often start with random weights. Random initializations can move into different local minima for the same training data. This could change the inference results. During benchmarking this effect has been observed for Rasa using the Spacy back-end. According to a mail from the main author Rasa 0.5 with the MITIE back-end is used for the results described in the paper. The MITIE back-end has not shown to change accuracy after re-training the model.

3.4.3 State of field

Chapter 4

Improving accuracy

TODO: Write intro.

4.1 Search

The field of NLP is rapidly evolving due to the introduction of deep learning. Systems which obtain state of the art (SOTA) accuracy are often surpassed within a few months. A search is conducted to improve the accuracy of the existing systems. The research for this part has not been systematic. This is caused by the goal to improve accuracy. When aiming for such a goal completeness is not important, as long as the improvement is realised. The method for finding an improvement is based on coming up with 'novel' approaches to improve accuracy. After having such an 'novel' approach the literature is consulted. This search method relies on the assumption that papers have done their research and will provide proper related work. This section will explain the considered ideas and related literature.

4.1.1 Duplicate finding

A large part of communications with customers consist of answering questions. Some questions will be duplicates, or in other words, some questions will have been asked and answered before. Finding duplicate questions is the same as finding clusters in the data. Another approach could be based on template responses used by customer support teams. A classifier could be trained to come up with these template responses.

Another approach to find duplicates is using semantic text similarity (STS). STS is a NLP tasks focusing on finding sentences (or texts) having the same meaning. It is a recurring task in the SemEval workshop. Systems in this field obtain impressive results, however it would not help with the chosen task of intent classification. Also, intent classification is more useful for the thesis than only knowing whether two sentences are the same.

4.1.2 Using data

According to Warden [42] it is more effective to get more training data than to apply better models and algorithms. For conversational agents in company settings it is easy to get raw data. This shifts the problem to applying to automatic data wrangling. This is not further investigated since the data obtained from customers cannot be part of a public thesis. Learning automatically from users is applied by some Microsoft chatbots. Microsoft Tay famously started to learn offensive language from users and has been shut down as a result. In China Microsoft has had a more successful release of XiaoIce. XiaoIce is optimized for "long-term user engagement" [50]. Engagement is achieved by establishing an emotional connection with the user. This system which is created by a research lab and being used by 660 million users does not automatically use the data to learn. The authors manually optimize the engagement of the system. From this it is concluded that

reinforcement learning and automatic data wrangling are not yet feasible approaches to increase accuracy.

4.1.3 Kaggle

Kaggle (https://www.kaggle.com) is a well-known site in the machine learning domain. On this site a framework exists where datasets can be published. The site, along other things, shows statistics, a comments section and a scoreboard. It is famous for hosting 'competitions', where the person or team obtaining the highest accuracy for some task gets price money from the dataset hoster. Kaggle provides a way for machine learning enthusiasts to communicate. People who obtain top 20 high scores on difficult tasks tend to explain their pipeline in a blogpost. Research papers tend to focus on designing the best deep learning architectures. The Kaggle explanations are valuable sources for learning how to get the most out of the architectures. One such post [20] uses three embeddings, namely Glove, FastText and Paragram. The author argues that "there is a good chance that they (the embeddings) capture different type of information from the data". This method is called boosting. Predictions from the embeddings are combined by taking the average score. A threshold is set to remove answers where the model is unsure. This method could be used to improve performance for natural language understanding. Running three systems in parallel does increase the training time, but the difference is not too large. It would be interesting to test whether averaging can be replaced by a more involved calculation. Meta-algorithms such as boosting, bagging and stacking are not investigated further since the improvement is expected to be insignificant.

4.1.4 Meta-learning

Meta-learning is "is the science of systematically observing how different machine learning approaches perform on a wide range of learning tasks, and then learning from this experience, or meta-data, to learn new tasks much faster than otherwise possible" Vanschoren [39]. Few-shot learning aims to learn useful representations from a few examples. In practise most intent classification systems use few examples, so few-shot learning is interesting to the research question. This was also concluded by the IBM T. J. Watson Research Center [48]. The authors show that their system outperforms other few-shot learning approaches. They do not compare their system against natural language understanding solutions and conclude that their research should be applied to other few-shot learning tasks. This implies that natural language understanding specific systems obtain higher accuracies. Automatically tuning hyperparameters as done in TensorFlow's AutoML is based on the work by Andrychowicz et al. [1]. Industry claim that AutoML obtains 95% of the accuracy of hand-tuning hyperparameters. Another problem is that it does not scale well [15]. Transfer learning approaches like MAML [12] and Reptile [27] could be useful for intent classification as well. Different domains require different models. Reptile seems interesting to be used to train a model on one domain and then be able to easily switch the model to other domains. This would introduce a lot of complexity in the code. More convenient would be using a model which works on all domains.

4.1.5 Embeddings

Embeddings capture knowledge about language and use that for downstream tasks. There appears to be a consensus about the timeline of embeddings evolution. Glove [31] was superseded by FastText [16]. The Facebook FastText embedding is aimed to be quick, allowing it to be used as a baseline. With 157 languages (https://fasttext.cc/) it is a mutli-lingual model. Another state-of-the-art and easy to implement embedding is the universal sentence encoder [5]. The word 'universal' denotes that the system has used a supervised training training task which has been chosen such that the embeddings generalize to downstream tasks. Not only Google, but also Microsoft research is working on multi-task learning [35]. These embeddings are not enough to improve on existing systems, since Rasa is using the universal sentence encoder [43]. One step

further would be to let the model decide what embedding it wants to use [19]. A caveat is the fact that one then needs to implement multiple embeddings (even when the model decides that one embedding is the best).

4.2 BERT

At the start of October 2018 Google published their NLP system called BERT [11]. During publication it was able to score state-of-the-art (SOTA) results for eleven NLP tasks.

todo: write section intro

4.2.1 Model description

Looking into the tasks we see that the results are obtained for a wide range of tasks, including:

- Entailment classification. For example, "People formed a line at the end of Pennsylvania Avenue." is contained in (logically implied by) "At the other end of Pennsylvania Avenue, people began to line up for a White House tour." [44]
- Semantic text similarity. For two sentences determine whether they mean the same thing.
- Sentence classification. Tasks involve classifying linguistic acceptability (correctness).
- Question answering. The Stanford Question answering dataset (SQuAD) [33] contains questions and a segment of text. The answer should be extracted from the segment if it answers the question.

The paper describes various reasons for the good results on the GLUE, MultiNLI and SQuAD datasets.

4.2.2 Training

From now on training is used to denote fine-tuning of the model. Training the general language model on some downstream task is presented as being inexpensive. Relative to the pre-training it is. Running training with the advised batch size of 32 will run out of RAM on a 16 GB RAM machine. The RAM usage on some machine will stay around 15 GB with batch size 16.

To train the model on some tasks it is advised to run 'a few epochs'. Let a be 'a few', and b the number of training examples. Based on the example code provided by Google researchers $a = 3$ and $b = 1000$. So, it is advised to show the system 3000 examples. For our smaller datasets of around 50 examples this means running $3000/50 = 60$ epochs. When measuring the training time in steps it means running $3000/16 \approx 188$ steps for a batch size of 16. Preliminary experiments on the AskUbuntu dataset (having 53 training examples) with a batch size of 32 confirm this estimate, see figure 4.1 and 4.3. The images show that the system takes does not easily find a local minimum, and can even have a sudden drop in performance, see figure 4.2. The results are interesting because it shows that the model is able to learn something even for a dataset with only tens of training examples. Training the model for 5 steps or 80 examples takes at least a few hours. Interpolation suggests that training 188 steps will take at least 36 hours. This is impractical when doing experiments.

According to the paper the benefit of the Transformer models is that they are highly parallelizable. Training BERT consist mainly of matrix multiplications [10]. These can be done quickly and efficiently on graphic processing units (GPUs) and tensor processing units (TPUs). The latter are ASICs created by Google specifically to do machine learning inference [17]. When using the TensorFlow implementation of BERT GPUs with 16 GB of RAM are required. GPU optimizations are available in the PyTorch implementation provided by Wolf et al. [45], but this does not support TPUs at the time of writing. Prices for these GPUs are at least a few thousand euros, which means most users and companies resort to cloud services. At the time of writing Google

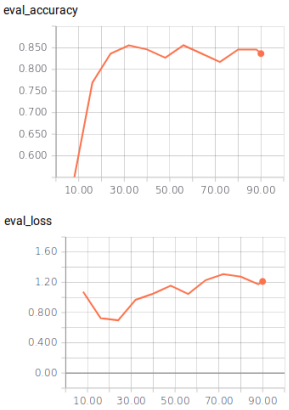

Figure 4.1: BERT-Base

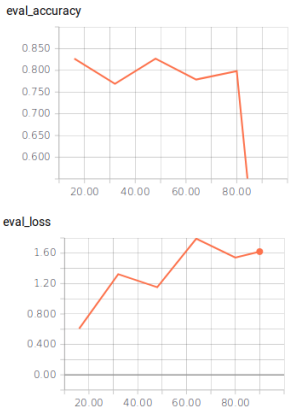

Figure 4.2: BERT-Large

Figure 4.3: BERT-Large, second run

Colab (`https://colab.research.google.com`) provides free access to a GPU and TPU instance. One example implementation is provided [2].

Using Colab is a compromise between usability and costs. The costs are limited to the use of some storage in a Google Cloud Bucket. Usability is hindered by the usual constraints of Jupyter Notebooks, for example no unittests, no autocomplete and poor Git integration. To overcome these issues most of the code is written and tested locally and pushed to a Github repository. In the Colab the code is then pulled from the repository and main functions are called. Using Colab has benefits as well. Hyperparameters and output are visible and can easily be modified in the Notebook, this eases verification. Reproducibility is possible by opening the Notebook and running all cells. The first cell will ask to link the Colab to a Google account, make sure this account has access to a Google Cloud Bucket.

The plots in figure 4.1, 4.2 and 4.3 are created using the default TensorFlow visualisation tool TensorBoard. Generating these plots can be done by specifying a model and metrics using the TensorFlow Estimator API.

4.2.3 Sentence classification

4.2.4 Joint training

One reason why neural networks are obtaining the best results for many fields is because networks are now deep. Deep networks have more layers and can therefore learn more complex tasks. One application of this is adding a larger portion of some pipeline to the model. For example, the code by Keller and Bocklisch [18] for the default pipeline for Rasa Spacy contains the following steps:

1. tokenizer
2. intent entity featurizer regex
3. intent featurizer spacy
4. ner crf
5. ner synonyms
6. intent classifier sklearn

Training occurs in the ner crf (Stanford Named Entity Recognizer) and intent classifier sklearn component. For Rasa these two components work separately. Preferably one would have one model which could learn to do the entire pipeline, also known as an end-to-end model. It seems

unfeasible to train the entire pipeline in a reasonable amount of time. Using BERT it is possible to train named entity recognition (NER) and intent classification in the same model.

That the combination improves independent models has been shown by Ma et al. [24] and Zhang et al. [49]. The results for the former are obtained by using a LSTM network. The latter introduces an algorithm to combine hidden states from an LSTM. They show this for the more general problem of sequential labeling and classification. Intuitively the improvement was to be expected for the following reason. Suppose we are trying to classify a dataset which contains the sentence:

"I would like to book a ticket to London tomorrow."

The sentence has intent 'BookFlight'. Training the model could be simplified by providing the sentence classifier with:

"I would like to book a ticket to <location> <date>."

Now the model does not have to learn to classify sentences while also learning that London is a location and that tomorrow is a date.

Note that an end-to-end model is preferred over two separate models. At the time of writing NER classifiers do not obtain perfect accuracy. This means that some classifications will be incorrect. The example from above could instead be converted to:

"I would like to book a <date> to <location> tomorrow."

This could make the intent classifier drop in accuracy. In an ideal end-to-end model incorrect NER classifications would be less of an issue. The model would learn to ignore the named entity recognition if it would not increase accuracy.

4.2.5 BERT joint training

That joint training BERT is possible can be observed from figure 4.4 and figure 4.5.

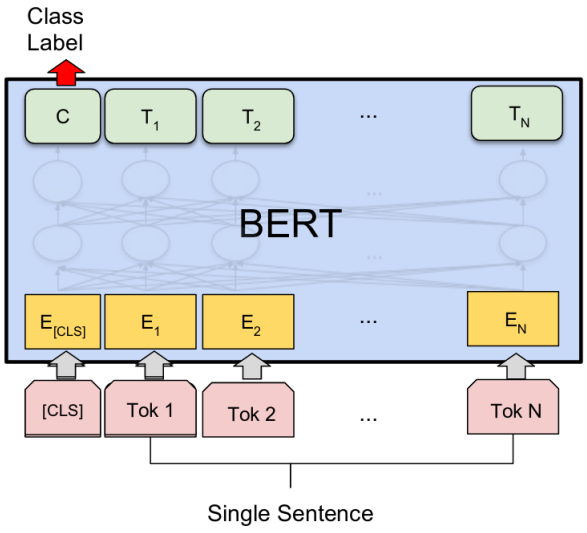

Figure 4.4: Single sentence classification task [11, Figure 3].

Let $A = A_1, A_2, \cdots, A_n$ denote the layer which is depicted below $C, T_1, \cdots, T_n$, and let $B = B_1, B_2, \cdots, B_n$ denote the layer below A. Let s denote the number of tokens for some input sentence. By default the max sequence length for the model is set to 128. For each sentence the sequence length is padded to this max sequence length. When predicting a 'class label' C will only

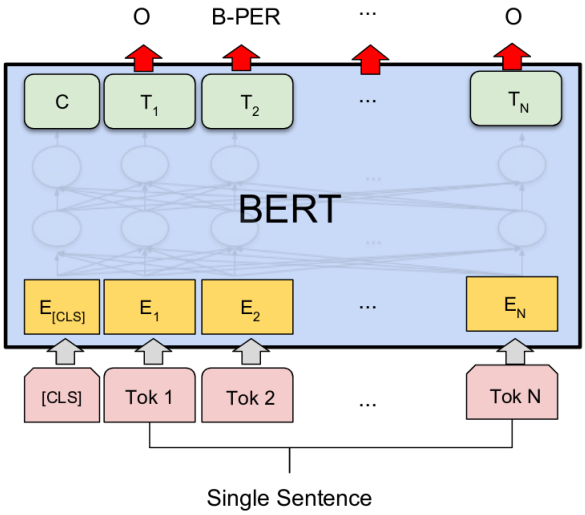

Figure 4.5: Single sentence tagging task [11, Figure 3].

be based on A_1 which is based on $B_1, B_2, \cdots, B_s$. $A_2, A_3, \cdots, A_n$ are not used. When predicting entities only $A_2, A_3, \cdots, A_s$ are used. It seems that a joint model is possible by providing the model with a combination of these two. Consider the NER as a base model and suppose we add some input to C. Now when predicting C the model is expected to learn to look at input from A. For this it can use entity information from $A_2, A_3, \cdots, A_s$. To also learn non-trivial patterns in non-entity words in the sentence it can use $A_2, A_3, \cdots, A_n$. Typically sentences are much shorter than 128 tokens so enough space should be available in $A_2, A_3, \cdots, A_n$. To allow for more space the max sequence length can be increased, this will increase training and inference time.

To do this the input for the model has been changed from:

```
text: ['how', 'do', 'i', 'disable', 'the', 'spam', 'filter', 'in', 'gmail', '?']
true: ['O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'B-WebService', 'O']
```

to

```
text: ['INTENT', 'how', 'do', 'i', 'disable', 'the', 'spam', 'filter', 'in',
                                                              'gmail', '?']
true: ['FilterSpam', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'B-WebService', 'O']
```

where 'text' is passed to $C, T_1, \cdots, T_s$ and 'true' to $E_{[CLS]}, E_1, \cdots, E_n$ during training. The BERT tokenizer splits words which are not listed in the vocabulary corresponding to a pre-trained model. INTENT is capitalized to force it to be out-of-vocabulary, it is converted by BERT to [UNK]. The goal of this is to avoid overriding the default interpretation BERT has for 'intent' (or any other uncased token we choose).

The NER loss function can be applied to the joint training without change. This is counter-intuitive because input examples have one position for the class C and s positions for the entities $T_1, T_2, \cdots, T_s$. Typically sentences have around 10 tokens after tokenization. So, the loss is based on around one intent position and ten entities. This means that the model will learn entity recognition much quicker than intent classification. For situations where the model is able to learn the entities quickly this is not expected to affect the performance of the intent classification significantly. It is expected that the only significant difference of this change is in the difficulty of the loss function.

4.3 Results

Experiments are conducted on the AskUbuntu, Webapplications, Chatbot and SNIPS2017 dataset as introduced in section 3.1.2. Comparisons are made for a fixed number of steps (or equivalently epochs). The reason for this is that intermediate results are not easily reported for the BERT model as explained in section 4.2.2. The number of steps to be used for training is basically a guess on what should be enough. For each dataset various runs for BERT are executed. During one run only intents are shown to the system and accuracy is measured for intents. Another run only shows entities and measures intents. A third run shows the system intents and entities and measures both. These methods are denoted as separate or joint. It is expected that the joint training increases accuracy for both intent and entity classifications. The reason for this is that the model sees more varied data and hence should be able to more easily find a good internal representation of the data. Results are listed in table 4.1.

Note that separate training consists of two runs, and hence ran twice as many epochs. This seems fair, since intent or entity improvements which require twice as many training steps are not interesting for expensive models such as BERT. As a baseline Rasa 0.13.8 with the Spacy pipeline is used. Only intent classification is possible using the benchmark code, so entity scores are missing. Omitting scores for other systems has been deliberate. The table is merely meant to support that joint training is feasible. A final remark is that the scores have been rounded to two decimals. The reason for this is that results vary between runs and for changes to hyperparameters. To avoid falsely reporting perfect scores (having accuracy of 1.0), all intent and entity numbers are rounded down. The number of epochs is calculated by taking number of training steps times training batch size and dividing by number of training examples. The training time for each model is around 10 minutes. Running time differences between runs are small, since most time is spent on training preparations and transferring model checkpoints between TPU and cloud storage.

Dataset	**Steps**	**Batch size**	**Epochs**	**Method**	**Intent**	**Entity**
AskUbuntu				Rasa	0.83	
	250 (twice)	32	151 (twice)	separate	0.74	0.99
	250	32	151	joint	0.98	0.79
WebApplications				Rasa	0.67	
	250 (twice)	32	267 (twice)	separate	0.00	0.79
	250	32	267	joint	0.65	0.81
	1000 (twice)	8	267 (twice)	separate	0.72	0.79
	1000	8	267	joint	0.53	0.80
Chatbot				Rasa	0.98	
	250 (twice)	16	40 (twice)	separate	0.99	0.74
	250	16	40	joint	0.98	0.79
Snips2017				Rasa	0.99	
	1500 (twice)	32	22.9 (twice)	separate	0.03	0.83
	1500	32	22.9	joint	0.97	0.85

Table 4.1: Weighted f1 accuracy scores for separate and joint training on four datasets.

The code to reproduce the results the logs (including predictions) can be found at the links provided in table 4.2.

Dataset	**Url**
AskUbuntu	https://github.com/rikhuijzer/improv/tree/master/runs/askubuntu
WebApps	https://github.com/rikhuijzer/improv/tree/master/runs/webapplications
Chatbot	https://github.com/rikhuijzer/improv/tree/master/runs/chatbot
Snips2017	https://github.com/rikhuijzer/improv/tree/master/runs/snips2017

Table 4.2: Hyperlinks for Rasa and BERT run information.

From the results it can be concluded that joint training is feasible. The model will not learn anything when training on intents separately for SNIPS2017 and WebApplications. For WebApplications it has been found that lowering the step size can solve this problem. This suggests that the gradient descent does not converge because the step size is too big. A bigger batch size means a more stable error gradient. It might be that this caused the model to get stuck in a local minimum. Specifically, the difference with joint training is that the joint training data is much more varied. Say a typical sentence contains 12 tokens. Then a joint training batch of size 32 will contain about 3 tokens related to intents and 29 related to entities. For an intent training batch of size 32 it will contain 32 tokens related to intents. Hence, the data for the joint training is much more complex. This seems to indicate that the joint training forces the model to learn a more complex representation. An alternative explanation could be related to the fact that training data is not shuffled. The default BERT code hints to shuffle data, but when comparing the Colab log with the training data, the implementation in IMPROV does not.

An important thing to note about the results is that the datasets are very small. One would expect that the large BERT model is better suited for datasets which contain more training examples. Furthermore, the experiments are based on a naive implementation. Not only the batch size but also other hyperparameters can be tuned for better results. Training has used a fixed number of steps or epochs. It might be that more epochs give higher accuracies. On the other hand it might also be that less epochs correspond to similar accuracies in less training time. Other interesting hyperparameters are *max_seq_length* and *learning_rate*. Lowering the former to the expected maximum number of tokens in sentences reduces training and inference time.

Chapter 5

Future work

Write intro.

5.1 Improved datasets

5.2 BERT implementation improvements

Chapter 6

Conclusions

Media suggests that difficult natural language processing (NLP) tasks can be solved by using artificial intelligence. It is interesting to see whether this can be used to automate customer support. To this end various NLP tasks have been considered. Eventually it is decided that intent classification is the most interesting task. Intent classification attempts to classify the intention of some user when he utters some sentence. For example, "what is the weather tomorrow?" could be labeled as 'get_weather'. A conversational agent (advanced chatbot) could then use this intent to decide how to respond. While reading about this task it was found that various systems claim to have obtained the highest accuracy scores for certain datasets. This is highly suspicious and gives rise to the following research question and goal.

RQ1. Can an open-source benchmarking tool for natural language understanding systems and services be created?

RG1. Develop an open-source reproducible tool for benchmarking of natural language understanding systems.

The answer for the first research question is that it is possible, but impractical. Main issue is that to test a service one has to make API calls. This requires the user of a benchmark tool to have an user account for each service and it forces the tool to update for each changing service. Another issue is that evaluation of results is done on pre-defined datasets. This does not guarantee that some system is indeed the best choice for some problem at hand. It could be that the pre-defined dataset contains more training data, is in another domain or uses another language. Even when using the benchmarking tool with some use-case specific dataset knowing the best system is not very useful. Services push new models into production without letting users know, so benchmark results can become invalid at any moment.

Next, the tool (and knowledge obtained by creating the tool) is used to work on the following research question and goal.

RQ2. Can the classification accuracy for natural language understanding be increased?

RG2. Improve the accuracy of natural language understanding systems.

The search for improvement has considered increasing the amount of training data and using new meta-learning algorithms and embeddings. A recent model called Google BERT [11] is expected to be the most likely candidate for increasing accuracy. The model uses pre-training to learn about language and can be fine-tuned to some specific task. It is a big model, meaning that fine-tuning takes around 1,5 days on a modern computer and a few minutes on a high-end GPU. Experiments on intent classification datasets showed non-significant improvements in accuracy. To improve accuracy further the model has been jointly trained on intent classification and

named-entity recognition. The benefit is that named-entity information can be used to determine the intent and vice versa. The Google model is a good candidate for jointly training because it uses left and right context in all layers (deep bidirectionality). BERT has obtained state-of-the-art results in a wide range of tasks including named-entity recognition. This implies that jointly training BERT should obtain state-of-the-art results on the natural language understanding task. Specifically, on tasks where data consists of intents and named-entities which is typical for conversational agents. Basic experiments are conducted in which training BERT separately is compared to training it jointly. The experiments show that jointly training is possible and in some cases obtains higher accuracies than separate training. It is expected that the results can be improved by taking a second look at the implementation and improving it. One issue for the current implementation is that the implementation does not shuffle the training data.

In general the NLP field is in an interesting state. Technology companies have a lot of incentive to push the field forward. Leaps in the last few years have come from those companies. For example, FastText by Facebook and transformer models by Google. A second observation is that state-of-the-art scores are increased every few months. This results in papers which are quickly pushed to arXiv and cited before any scholarly peer review.

Bibliography

[1] Marcin Andrychowicz, Misha Denil, Sergio Gomez, Matthew W Hoffman, David Pfau, Tom Schaul, Brendan Shillingford, and Nando De Freitas. Learning to learn by gradient descent by gradient descent. In *Advances in Neural Information Processing Systems*, pages 3981–3989, 2016.

[2] Sourabh Bajaj. Bert finetuning tasks in 5 minutes with cloud tpu. https://colab.research.google.com/github/tensorflow/tpu/blob/master/tools/colab/bert_finetuning_with_cloud_tpus.ipynb, 2018. Accessed: 2018-12-27.

[3] Daniel Braun, Adrian Hernandez-Mendez, Florian Matthes, and Manfred Langen. Evaluating natural language understanding services for conversational question answering systems. In *Proceedings of the 18th Annual SIGdial Meeting on Discourse and Dialogue*, pages 174–185, 2017.

[4] Mikhail Burtsev, Alexander Seliverstov, Rafael Airapetyan, Mikhail Arkhipov, Dilyara Baymurzina, Nickolay Bushkov, Olga Gureenkova, Taras Khakhulin, Yuri Kuratov, Denis Kuznetsov, et al. Deeppavlov: Open-source library for dialogue systems. *Proceedings of ACL 2018, System Demonstrations*, pages 122–127, 2018.

[5] Daniel Cer, Yinfei Yang, Sheng-yi Kong, Nan Hua, Nicole Limtiaco, Rhomni St John, Noah Constant, Mario Guajardo-Cespedes, Steve Yuan, Chris Tar, et al. Universal sentence encoder. *arXiv preprint arXiv:1803.11175*, 2018.

[6] Nancy Chinchor, Erica Brown, Lisa Ferro, and Patty Robinson. 1999 named entity recognition task definition. *MITRE and SAIC*, 1999.

[7] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

[8] Jinho D Choi, Joel Tetreault, and Amanda Stent. It depends: Dependency parser comparison using a web-based evaluation tool. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, volume 1, pages 387–396, 2015.

[9] Robert Dale. Text analytics apis, part 2: The smaller players. *Natural Language Engineering*, 24(5):797–803, 2018.

[10] Tim Dettmers. Tpus vs gpus for transformers (bert). http://timdettmers.com/2018/10/17/tpus-vs-gpus-for-transformers-bert, 2018. Accessed: 2018-12-27.

[11] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[12] Chelsea Finn, Pieter Abbeel, and Sergey Levine. Model-agnostic meta-learning for fast adaptation of deep networks. *arXiv preprint arXiv:1703.03400*, 2017.

[13] Matt Gardner, Joel Grus, Mark Neumann, Oyvind Tafjord, Pradeep Dasigi, Nelson F. Liu, Matthew Peters, Michael Schmitz, and Luke S. Zettlemoyer. AllenNLP : A deep semantic natural language processing platform. In *ACL workshop for NLP Open Source Software*, 2018.

[14] Rob High. The era of cognitive systems: An inside look at ibm watson and how it works. *IBM Corporation, Redbooks*, 2012.

[15] Llion Jones. Learning to learn by gradient descent by gradient descent. https://hackernoon.com/learning-to-learn-by-gradient-descent-by-gradient-descent-4da2273d64f2, 2017. Accessed: 2019-01-14.

[16] Armand Joulin, Edouard Grave, Piotr Bojanowski, and Tomas Mikolov. Bag of tricks for efficient text classification. *arXiv preprint arXiv:1607.01759*, 2016.

[17] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-datacenter performance analysis of a tensor processing unit. In *Computer Architecture (ISCA), 2017 ACM/IEEE 44th Annual International Symposium on*, pages 1–12. IEEE, 2017.

[18] Caleb M Keller and Tom Bocklisch. config_spacy_duckling.yml. https://github.com/RasaHQ/rasa_nlu/blob/master/sample_configs/config_spacy_duckling.yml, 2018. Accessed: 2018-12-24.

[19] Douwe Kiela, Changhan Wang, and Kyunghyun Cho. Dynamic meta-embeddings for improved sentence representations. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 1466–1477, 2018.

[20] Sudalairaj Kumar. A look at different embeddings.! https://www.kaggle.com/sudalairajkumar/a-look-at-different-embeddings/notebook, 2018. Accessed: 2018-11-11.

[21] Irene Langkilde and Kevin Knight. The practical value of n-grams is in generation. *Natural Language Generation*, 1998.

[22] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553):436, 2015.

[23] Steven Lott. *Functional Python Programming*. Packt Publishing Ltd, 2015.

[24] Mingbo Ma, Kai Zhao, Liang Huang, Bing Xiang, and Bowen Zhou. Jointly trained sequential labeling and classification by sparse attention neural networks. *arXiv preprint arXiv:1709.10191*, 2017.

[25] Christopher Manning and Richard Socher. Natural language processing with deep learning. Lecture Notes Stanford University School of Engineering, 2017.

[26] Tomas Mikolov, Armand Joulin, Sumit Chopra, Michael Mathieu, and Marc'Aurelio Ranzato. Learning longer memory in recurrent neural networks. *arXiv preprint arXiv:1412.7753*, 2014.

[27] Alex Nichol and John Schulman. Reptile: a scalable metalearning algorithm. *arXiv preprint arXiv:1803.02999*, 2018.

[28] Taiichi Ohno. *Toyota production system: beyond large-scale production*. crc Press, 1988.

[29] C Olah. colah's blog. http://colah.github.io/, 2005. Accessed: 2018-12-10.

[30] Cassio Pennachin and Ben Goertzel. Contemporary approaches to artificial general intelligence. In *Artificial general intelligence*, pages 1–30. Springer, 2007.

[31] Jeffrey Pennington, Richard Socher, and Christopher Manning. Glove: Global vectors for word representation. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, pages 1532–1543, 2014.

[32] Matthew E Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer. Deep contextualized word representations. *arXiv preprint arXiv:1802.05365*, 2018.

[33] Pranav Rajpurkar, Robin Jia, and Percy Liang. Know what you don't know: Unanswerable questions for squad. *arXiv preprint arXiv:1806.03822*, 2018.

[34] Mike Schuster and Kuldip K Paliwal. Bidirectional recurrent neural networks. *IEEE Transactions on Signal Processing*, 45(11):2673–2681, 1997.

[35] Sandeep Subramanian, Adam Trischler, Yoshua Bengio, and Christopher J Pal. Learning general purpose distributed sentence representations via large scale multi-task learning. *arXiv preprint arXiv:1804.00079*, 2018.

[36] Eiichiro Sumita and Hitoshi Iida. Experiments and prospects of example-based machine translation. In *Proceedings of the 29th annual meeting on Association for Computational Linguistics*, pages 185–192. Association for Computational Linguistics, 1991.

[37] Erik F Tjong Kim Sang and Fien De Meulder. Introduction to the conll-2003 shared task: Language-independent named entity recognition. In *Proceedings of the seventh conference on Natural language learning at HLT-NAACL 2003-Volume 4*, pages 142–147. Association for Computational Linguistics, 2003.

[38] J Uszkoreit. Google ai blog. https://ai.googleblog.com, 2017. Accessed: 2018-12-10.

[39] Joaquin Vanschoren. Meta-learning: A survey. *arXiv preprint arXiv:1810.03548*, 2018.

[40] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems*, pages 5998–6008, 2017.

[41] Stephan Vogel, Hermann Ney, and Christoph Tillmann. Hmm-based word alignment in statistical translation. In *Proceedings of the 16th conference on Computational linguistics-Volume 2*, pages 836–841. Association for Computational Linguistics, 1996.

[42] Peter Warden. Why you need to improve your training data, and how to do it. https://petewarden.com/2018/05/28/why-you-need-to-improve-your-training-data-and-how-to-do-it/, 2018. Accessed: 2019-01-12.

[43] Georg Wiese. Enhancing intent classification with the universal sentence encoder. https://scalableminds.com/blog/MachineLearning/2018/08/rasa-universal-sentence-encoder, 2018. Accessed: 2019-12-03.

[44] Adina Williams, Nikita Nangia, and Samuel R. Bowman. A broad-coverage challenge corpus for sentence understanding through inference. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. Association for Computational Linguistics, 2018.

[45] Thomas Wolf, Victor Sanh, Gregory Chatel, and Tim Rault. pytorch-pretrained-bert. https://github.com/huggingface/pytorch-pretrained-BERT, 2018. Accessed: 2018-12-27.

[46] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, et al. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*, 2016.

[47] Tom Young, Devamanyu Hazarika, Soujanya Poria, and Erik Cambria. Recent trends in deep learning based natural language processing. *ieee Computational intelligenCe magazine*, 13(3): 55–75, 2018.

[48] Mo Yu, Xiaoxiao Guo, Jinfeng Yi, Shiyu Chang, Saloni Potdar, Yu Cheng, Gerald Tesauro, Haoyu Wang, and Bowen Zhou. Diverse few-shot text classification with multiple metrics. *arXiv preprint arXiv:1805.07513*, 2018.

[49] Chenwei Zhang, Yaliang Li, Nan Du, Wei Fan, and Philip S Yu. Joint slot filling and intent detection via capsule neural networks. *arXiv preprint arXiv:1812.09471*, 2018.

[50] Li Zhou, Jianfeng Gao, Di Li, and Heung-Yeung Shum. The design and implementation of xiaoice, an empathetic social chatbot. *arXiv preprint arXiv:1812.08989*, 2018.

Appendix A

bench

The benchmarking tool is called BENCH and located on Github (https://github.com/rikhuijzer/bench). Its goal is to be a reproducible benchmarking tool for intent classification and named-entity recognition. Reproducible means that other people can run the code for their own use, or to reproduce results presented in this report. The code is written in Python, since it is the default choice for machine learning. Python is conceived as a object oriented language. Over time it has included more and more functional programming ideas. The code in this project will aim to be adhering to functional programming. Reasons are pedagogic value, improved modularity, expressiveness, ease of testing, and brevity. Some general notes on functional programming in Python are listed in appendix B.

The functional programming constraints for the project are that we do not define any new classes. Specifically, we do not use the class keyword. Exceptions being NamedTuples and Enums. Both do provide functional APIs, but these are not fully supported by the autocomplete in PyCharm. The code prefers returning iterators over collections, the reason for this is explained in appendix C. A final remark is about the imports. When importing an attempt is made to explicitly import using 'from <module> import <class>'. When more implicit imports are used 'import <module>' this is can have multiple causes. It is either caused by the appearance of circular imports, by the fact that some names are too common or to avoid reader confusion. An example for the latter are the types defined in SRC.TYP. The names are quite generic and could cause name clashing or confusion when imported explicitly.

A.1 Usage

Installation is similar to other Python projects. Pull the code and in a terminal set the current working directory to the project folder. Install the required pip packages by running the following command.

```
pip install -r requirements.txt
```

If one wants to check accuracy for an open-source system then run the following command.

```
docker-compose up
```

DOCKER-COMPOSE will read 'docker-compose.yml' and use that information to spin up various Docker containers. All dockers listed in the file are available from Docker Hub. This avoids having to build Dockers manually. DeepPavlov has been removed from the configuration file, since it was found to be unstable, see section 3.2.2.

After the set-up the program can be executed by running 'bench.py'. To change on which system the benchmarking occurs, replace the first parameter in the GET_SYSTEM_CORPUS call. The prefix is used to determine which system is being tested. Possible prefix options are 'mock', 'rasa', 'deeppavlov', 'lex' and 'dialogflow'. Rasa and DeepPavlov will use the complete string to

find a matching port from 'docker-compose.yml'. So, based on the Docker configuration one can also specify 'rasa-tensorflow', 'rasa-spacy' or 'rasa-mitie'. The corpus (dataset) to run the bench on is specified by an enumerable, see SRC.TYP.CORPUS for possible options. When running the script in a modern IDE autocomplete will suggest the possible corpora. Slightly more convenient would be to have the script take input arguments using SYS.ARGV. After setting the two parameters the script can be executed and will display all predictions as well as micro, macro and weighted f1 scores. The predictions and f1 scores will also be written to files, see the 'results' folder.

A.2 Overview

A high-level code overview will be presented. Since the code does not contain classes, the overview is simply a tree-like structure. This is analogous with a book, where subsections are contained in sections and sections are contained in chapters. In the code small functions are called by larger functions and these larger functions are called by even larger functions. For an overview this idea can be generalized to modules. An overview for the modules of BENCH is roughly as follows. The '.py' suffix is omitted for all elements in the tree. Tests are also omitted.

- BENCH
  - SRC.UTILS
  - SRC.TYP
  - SRC.DATASET
    - SRC.DATASETS.CORPORA
    - SRC.DATASETS.SNIPS
  - SRC.SYSTEM
    - SRC.SYSTEMS.AMAZON_LEX
    - SRC.SYSTEMS.DEEPPAVLOV
    - SRC.SYSTEMS.DIALOGFLOW
    - SRC.SYSTEMS.MOCK
    - SRC.SYSTEMS.RASA
    - SRC.SYSTEMS.WATSON
  - SRC.EVALUATE
    - SRC.RESULTS

Some generic functions are listed in SRC.UTILS and used through the entire project. The project makes use of type hints as introduced in Python 3 (and via comments in Python 2.7). Also, the project does not use classes and therefore tends to pass more data through functions. To define containers for these data NamedTuples are used. A more in-depth explanation of why these are needed can be found in section B.2. All NamedTuples or 'types' are defined in SRC.TYP. The module also contains enumerable's or 'Enums'. These are used in cases where function behaviour depends on some parameter from a fixed set of options. Alternatively one could use strings for these cases depending on user-preference.

The real work of the project is done by SRC.DATASET, SRC.SYSTEM and SRC.EVALUATE. 'Dataset' takes input files and converts them to an internal representation as defined by SRC.TYP. Input files here denote the original dataset files as created by the dataset publishers. For the internal representation a Rasa Message is used, specifically RASA_NLU.TRAINING_DATA.MESSAGE.MESSAGE. The benefit of this is that it avoids defining the same structure and that it can be used in combination with Rasa code. For example, SRC.DATASET.CONVERT_MESSAGE_TO_ANNOTATED_STR uses Rasa code to print the internal data representation as a sentence in Markdown format (section 3.1.1). Next, the data reaches SRC.SYSTEM. Here it is passed to the system under consideration, either in training or prediction mode. For the predictions this is done by finding out which

function can convert SRC.TYP.QUERY to SRC.TYP.RESPONSE. When, for example, Rasa is under consideration the function SRC.SYSTEMS.RASA.GET_RESPONSE is called. DeepPavlov would be handled by SRC.SYSTEMS.DEEPPAVLOV.GET_RESPONSE. PyCharm is known to have the best type inference for Python. The IDE is not yet able to infer function type for a function mapping, even when all functions have the same input and output type. A workaround is to manually define the type of the function returned by the mapping as `func: Callable[[tp.Query], tp.Response] = ...`. SRC.EVALUATE takes all responses TP.RESPONSE evaluates the performance of the system under consideration. Printing f1 score is a matter of three functions and about a dozen lines of code. At one point more advanced logging has been included which is responsible for the other 12 functions and 110 lines of code.

Appendix B

Notes on functional programming in Python

Python is not a pure functional language. However more and more constructs of functional programming are being added to the language each year. This appendix will explain some functional ideas used in the code, as presented by Lott [23]. Higher-order functions take or return functions, this is used to replace the factory design pattern as explained in section B.1. Keeping track of state without a class results in function signatures to contain many parameters, these can be handled by using NamedTuples, see section B.2. Another benefit of classes is that data can be stored, used for example in caching. A convenient solution for caching is described in section B.3. Collections of data are typically transformed via loops. Here each loop will transform the entire collection and move to the next transformation. Lazy evaluation, as described in section B.4, uses a more efficient way.

B.1 Mapping to functions

In code we often have a function which calls other functions depending on some conditionals. For example in 'system.py' the factory design pattern is replaced by a more functional design. In this design 'system.py' behaves like a super and delegates the work based on what system we currently interested in. We give an example for two systems. The delegation could be done via conditional statements.

```
if 'mock' in system.name:
  response = src.systems.mock.get_response(tp.Query(system, message.text))
elif 'rasa' in system.name:
  response = src.systems.rasa.get_response(tp.Query(system, message.text))
elif ...
```

This introduces a lot of code duplication. Therefore a dict is created.

```
get_intent_systems = {
  'mock': src.systems.mock.get_response,
  'rasa': src.systems.rasa.get_response,
  ...
}
```

Now we can just get the correct function from the dict and call it.

```
func: Callable[[tp.Query], tp.Response] =
  get_substring_match(get_intent_systems, system.name)
query = tp.Query(system, message.text)
response = func(query)
```

Note that GET_SUBSTRING_MATCH() implements the substring matching used in the conditional code (IF 'MOCK' IN SYSTEM.NAME:). Since the code can return any of the functions contained in the mapping they should all have the same signature and output. The used IDE (PyCharm 2018.2.4) is not able to check this. Therefore, functions from the mapping FUNC get a type hint. This allows the IDE to check types again and it allows developers to see what signature should be used for all the functions in the mapping.

B.2 NamedTuple

Pure functions by definition cannot rely on information stored somewhere in the system. We provide one example from the code where this created a problem and how this can be solved using NamedTuples.

The benchmarking tools communicates with a system called Rasa. Rasa starts in a default, untrained, state. To measure its performance we train Rasa and then send many sentences to the system. In general one prefers to functions should be as generic as possible. It makes sense to have one function which takes some sentence, sends it to Rasa to be classified and returns all information from the response. To avoid re-training Rasa for each system we have to remember whether Rasa is already trained. Passing a flag 'retrain' to the system is insufficient, since the function does not know where Rasa should train on. To make it all work we need the following parameters:

- SENTENCE: The sentence text.
- SENTENCE_CORPUS: The corpus the sentence is taken from.
- SYSTEM_NAME: Used to call the function which can train the specific system we are interested in.
- SYSTEM_KNOWLEDGE: Used in combination with SENTENCE_CORPUS to determine whether we need to re-train.
- SYSTEM_DATA: In specific cases even more information is needed.

re-training the system to check whether its outputs differ.

When this function has decided to train the system the system_knowledge changes. So as output we need to return “

Since Python 3.5 a NamedTuple with type hints is available.

To allow for better type checking and reduce the number of function parameters use is made of TYPING.NAMEDTUPLES.

B.3 Function caching

Functions can be cached using FUNCTOOLS.LRU_CACHE. This is mainly used for reducing the number of filesystem operations. A typical example is as follows. Suppose we write some text to a file iteratively by calling WRITE multiple times. Since we try to avoid storing a state WRITE does not know whether the file already exists. To solve we can do two things. The first option is passing parameters telling the function whether the file already exists. This is cumbersome, since this state needs to be passed through all the functions to the function which is calling the loop over WRITE. This can be done directly by calling WRITE or indirectly by calling some other function. The second option is defining a function to create a file if it does not yet exists CREATE_FILE. We call this function every time WRITE is called. This does mean that the filesystem is accessed to check the folder each time WRITE is called. To avoid all those filesystem operations CREATE_FILE can be decorated using FUNCTOOLS.LRU_CACHE. Now on all but the first calls to CREATE_FILE just query memory.

There is one caveat with this using function caching. Make sure to not try to mimic state. In other words the program should not change behaviour if the cache is removed. Reason for this is that any state introduced via the cache is similar to creating functions with side-effects but without all the constructs from object-oriented programming.

B.4 Lazy evaluation

By default Python is not interested in performance and advises to use a list for every collection. However, lists are mutable and therefore not suitable for hashing. Since hashing is not possible any function taking lists as input is not suitable for function caching.

Also, in many cases the list might not be the final structure we need. Consider the following use cases where the output of type list is used:

- Only unique values are required, so the list is casted to a set.
- Only whether some value satisfies P is required.
- The x first elements are required.
- Only the values satisfying x are required.
- Only an output which is transformed is required.

Considering all these use cases it makes more sense to return an iterator by default instead of a collection. One practical example for the bench project which supports this notion is using an iterator on classification requests.

Suppose we want to measure the performance of some cloud service. Suppose we wrote some code which takes a sentence from some corpus and performs the following operations on this sentence:

1. Send the sentence to some cloud service.
2. Transforms the response to the pieces of information we need.
3. Store this information.

Suppose one of the last two operations contains a mistake causing the program to crash. When not using an iterator all sentences will have been sent to the cloud service after the first operation. Since the post-processing did not succeed we did not obtain results and need to redo this operation. In effect the programming error caused us to waste about as many API calls as there are sentences in the corpus we are testing. This is a problem since the API calls cost money and take time to execute.

To solve this use lazy evaluation. For example, functions supporting lazy evaluation in Python are MAP, FILTER, REDUCE and ANY. Another benefit for using iterators is that it improves modularity and, once used to the paradigm, readability. Take the following typical Python code.

```
my_list = []
for item in some_iterable:
  updated_item = g(f(item))
  my_list.append(updated_item)
```

In this code some iterable is read and the transformation F and G are applied to each item in the iterable. The same code can be rewritten to use MAP as follows.

```
def transform(item: SomeType) -> OtherType:
  return g(f(item))

my_iterable = map(transform, some_iterable)
```

Appendix C

Lazy evaluation demonstration

This appendix demonstrates the effect of using iterators instead of regular collections. The code demonstrates this by processing some fictional raw materials to a chair. The first is function called FORD is similar to a Ford factory around 1915. Here each part of the assembly line just keeps producing items as long as there is input coming in. After a while the other parts of the assembly line start processing the items and discover a fault in the items. One problem of this way of working is that the factory now has a pile of incorrect items in their stock.

The second function called TOYOTA is similar to a Toyota factory after 1960. Here just-in-time (JIT) manufacturing is used as developed by Toyota [28]. Each item is processed only when the next step in the process makes a request for this item.

C.1 Benefits

Using JIT makes sense in computer programs for the following reasons. It saves memory. In each step in the process we only store one intermediate result instead of all intermediate results.

It can detect bugs earlier. Suppose you got a combination of processing steps, lets call them F and G and you apply them to 100 items. In F we send some object to a system and get a response. In H we store the response of this API call. Suppose there is a bug in H, lets say the file name is incorrect. Suppose this is not covered in the tests and we decide to run our program to get all the results we want. Using an approach similar to FORD the program crashes after doing 100 executions of F and G. This means that the program executed 100 API calls. Using TOYOTA the program crashes after just one API call. Here FORD has in essence wasted 99 API calls.

It does not make assumptions for the caller. Suppose some function K returns an iterable and is called by L. The function L can now decide how it wants to use the iterable. For example it can be casted to unique values via SET or it can partly be evaluated by using ANY.

C.2 Code

```
from typing import List, NamedTuple
""" See README.md """

Wood = NamedTuple('Wood', [('id', int)])
Chair = NamedTuple('Chair', [('id', int)])


materials = [Wood(0), Wood(1), Wood(2)]


def ford():
```

```
    """ Processing all the items at once and going to the next step. """
    def remove_faulty(items: List[Wood]) -> List[Wood]:
        out = []
        for material in items:
            print('inspecting {}'.format(material))
            if material.id != 1:
                out.append(material)
        return out

    def process(items: List[Wood]) -> List[Chair]:
        out = []
        for material in items:
            print('processing {}'.format(material))
            out.append(Chair(material.id))
        return out

    filtered = remove_faulty(materials)
    processed = process(filtered)
    print('Result of ford(): {}'.format(processed))


def toyota():
    """ Processing all the items one by one. """
    def is_not_faulty(material: Wood) -> bool:
        print('inspecting {}'.format(material))
        return material.id != 1

    def process(material: Wood) -> Chair:
        print('processing {}'.format(material))
        return Chair(material.id)

    filtered = filter(is_not_faulty, materials)
    processed = list(map(process, filtered))
    print('Result of toyota(): {}'.format(processed))


if __name__ == '__main__':
    ford()
    print()
    toyota()
```

C.3 Output

The output for the program is as follows.

```
inspecting Wood(id=0)
inspecting Wood(id=1)
inspecting Wood(id=2)
processing Wood(id=0)
processing Wood(id=2)
Result of ford(): [Chair(id=0), Chair(id=2)]

inspecting Wood(id=0)
```

```
processing Wood(id=0)
inspecting Wood(id=1)
inspecting Wood(id=2)
processing Wood(id=2)
Result of toyota(): [Chair(id=0), Chair(id=2)]
```

This demonstrates that iterator elements are only executed when called.

Appendix D

Intent f1 score calculations

The f1 score calculation by [3] uses micro averaging. For two reasons this appendix focuses these calculations on intent only. The first reason is that these results are compared against benchmarks from the BENCH project in section 3.3. Secondly, micro averages could be skewed when the number of intents and entities differ, as described in section 3.4.2. It is interesting to compare the differences. This appendix lists calculations for Rasa in section D.1, Watson Conversation in section D.2 and Microsoft LUIS in section D.3.

D.1 Rasa

Corpus	**Intent**	**True +**	**False -**	**False +**	**Precision**	**Recall**	**F1 score**
Chatbot	DepartureTime	34	1	1	0.971		
	FindConnection	70	1	1	0.986	0.986	0.986
	Σ	104	2	2	0.981	0.981	**0.981**
WebApps	ChangePassword	4	2	0	1	0.667	0.8
	DeleteAccount	9	1	5	0.643	0.9	0.75
	DownloadVideo	0	0	1	0		
	ExportData	0	3	0		0	
	FilterSpam	13	1	0	1	0.929	0.963
	FindAlternative	15	1	8	0.652	0.938	0.769
	None	0	4	1	0	0	
	SyncAccounts	3	3	0	1	0.5	0.667
	Σ	44	15	15	0.746	0.746	**0.746**
AskUbuntu	MakeUpdate	34	3	2	0.944	0.919	0.931
	SetupPrinter	13	0	2	0.867	1	0.929
	ShutdownComputer	14	0	6	0.7	1	0.824
	SRecommendation	33	7	4	0.892	0.825	0.857
	None	0	5	1	0	0	
	Σ	94	15	15	0.862	0.862	**0.862**

D.2 Watson Conversation

Corpus	Intent	True +	False -	False +	Precision	Recall	F1 score
Chatbot	DepartureTime	33	2	1	0.971	0.943	0.957
	FindConnection	70	1	2	0.972	0.986	0.979
	Σ	103	3	3	0.972	0.972	**0.972**
WebApps	ChangePassword	5	1	0	1	0.833	0.909
	DeleteAccount	9	1	3	0.75	0.9	0.818
	DownloadVideo	0	0	1	0		
	ExportData	2	1	2	0.5	0.667	0.572
	FilterSpam	13	1	2	0.867	0.929	0.897
	FindAlternative	15	1	1	0.938	0.938	0.938
	None	0	4	1	0	0	
	SyncAccounts	5	1	0	1	0.833	0.909
	Σ	49	10	10	0.831	0.831	**0.831**
AskUbuntu	MakeUpdate	37	0	4	0.902	1	0.948
	SetupPrinter	13	0	1	0.929	1	0.963
	ShutdownComputer	14	0	1	0.929	1	0.963
	SRecommendation	35	5	3	0.921	0.875	0.897
	None	1	4	1	0.5	0.2	0.286
	Σ	100	9	9	0.917	0.917	**0.917**

D.3 Microsoft LUIS

Corpus	Intent	True +	False -	False +	Precision	Recall	F1 score
Chatbot	DepartureTime	34	1	1	0.971	0.971	0.971
	FindConnection	70	1	1	0.986	0.986	0.986
	Σ	104	2	2	0.981	0.981	**0.981**
WebApps	ChangePassword	3	3	0	1	0.5	0.667
	DeleteAccount	8	2	0	1	0.8	0.889
	DownloadVideo	0	0	0			
	ExportData	3	0	1	0.75	1	0.857
	FilterSpam	12	2	0	1	0.857	0.923
	FindAlternative	14	2	2	0.875	0.875	0.875
	None	3	1	8	0.273	0.75	0.4
	SyncAccounts	5	1	0	1	0.833	0.909
	Σ	48	11	11	0.814	0.814	**0.814**
AskUbuntu	MakeUpdate	36	1	4	0.9	0.973	0.935
	SetupPrinter	12	1	2	0.857	0.923	0.935
	ShutdownComputer	14	0	0	1	1	1
	SRecommendation	36	4	5	0.878	0.9	0.889
	None	0	5	0		0	
	Σ	98	11	11	0.899	0.899	**0.899**

Appendix E

improv

Appendix F

nlu_datasets