NTNU
Norwegian University of
Science and Technology

Deep Reinforcement Learning and Generative Adversarial Networks for Abstractive Text Summarization

Alf Niklas Håkonsen Kalmar
Borgar Rannem Lie

Master of Science in Computer Science
Submission date: June 2018
Supervisor: Massimiliano Ruocco, IDI
Co-supervisor: Erlend Aune, Exabel

Norwegian University of Science and Technology
Department of Computer Science

Abstract

News articles, papers and encyclopedias, among other texts can be time-consuming to digest. Often, you are not interested in reading all the material, but only some of it. Summaries can be useful to get a grasp of what they are about. The task of generating a summary is time-consuming because you need to read the text, and you need to understand which parts are important. This makes it very attractive to try to automatically generate summaries using a computer program.

Abstractive text summarization has gained a lot of attraction in recent years and the standard supervised learning approach have seen promising results when used to train abstractive text summarization models. However, they are limited by the fact that they assume the ground truth to be provided at each time-step during training. This is not the case at test time, where the previous generated word is provided instead. This creates a gap between training and testing, also known as ”exposure bias”.

In this thesis, we explore how to improve an abstractive text summarization model by employing reinforcement learning and generative adversarial networks, which do not assume the ground truth to be provided during training. As a base model to improve upon, we implement a variation of the Pointer-Generator Network [See et al., 2017].

There are a lot of implementation details and parameter choices that are important for training stability and convergence, which are mostly left out of research papers. In this regard, we conduct an extensive study on how different training strategies, parameters and objective functions affect training stability and convergence, as well as the generated summaries. Another problem with training of abstractive text summarization models is that it is generally very time-consuming. To cope with some of these problems, we propose code optimization techniques that will help speed up the training time.

We show improvements from the base model in terms of ROUGE scores, as well as differences in generated summaries, using the objective functions of ROUGE-1, ROUGE-2, discriminator (adversarial training), and a combined model of ROUGE-2 and the discriminator.

Sammendrag

Nyhetsartikler, forskningsartikler og oppslagsverk samt andre tekstformater kan være tidkrevende å fordøye. Ofte er du ikke interessert i å lese alt, men bare de viktigste delene. Sammendrag kan være nyttige for å forstå essensen i teksten. Det å generere et sammendrag er tidkrevende, fordi du må lese teksten, og du må forstå hvilke deler som er viktige. Dette gjør det veldig attraktivt å automatisk generere sammendrag ved hjelp av et dataprogram.

Generering av abstrakte tekstsammendrag har fått mye oppmerksomhet de siste årene, og den vanlige tilnærmingen for å trene abstrakte tekstsammendragsmodeller (supervised learning) har fått lovende resultater. Imidlertid er deres evne til å generalisere begrenset, fordi denne treningsmetoden forventer å motta det forrige korrekte ordet fra et referansesammendrag for hvert ord som skal genereres under trening. Dette er ikke tilfelle ved testing, hvor det forrige genererte ordet benyttes istedenfor. Dette skaper et stort skille mellom trening og testing, også kjent som ”exposure bias”.

I denne masteroppgaven undersøker vi hvordan en kan forbedre en abstrakt tekstsammendragsmodell ved å bruke reinforcement learning og generative adversarial networks, som ikke antar at de korrekte ordene benyttes under trening. Vi implementerer en variant av Pointer-Generator Network [See et al., 2017] som grunnmodell. Denne grunnmodellen skal forbedres ved hjelp av reinforcement learning og generative adversarial networks.

Det er mange implementeringsdetaljer og parametervalg som er viktige for stabilitet og konvergering under treningen, som ofte blir utelatt i forskningsartikler. I denne forbindelse gjennomfører vi et omfattende studie om hvordan ulike treningsstrategier, parametere og mål-funksjoner påvirker stabilitet og konvergering under trening, samt hvordan de påvirker genereringen av sammendrag. Et annet problem med trening av abstrakte tekstsammendragsmodeller er at prosessen er tidkrevende. For å håndtere noen av disse problemene, foreslår vi kodeoptimaliseringsteknikker som vil bidra til å forbedre treningstiden.

Vi viser ROUGE-score forbedringer sammenlignet med grunnmodellen, samt forskjeller i de genererte sammendragene. Dette blir oppnådd ved bruk av mål-funksjonene ROUGE-1, ROUGE-2, diskriminator (adversarial training) og en kombinert modell av ROUGE-2 og diskriminatoren.

Preface

This thesis was written by Alf Niklas Håkonsen Kalmár and Borgar Rannem Lie during the spring of 2018. The thesis is the final delivery of the Master of Science (MSc) degree at the Department of Computer Science (IDI) at the Norwegian University of Science and Technology (NTNU). We would like to thank Massimiliano Ruocco, Erlend Aune and Eliezer da Silva for the guidance during this thesis and for providing a good environment for exchanging ideas. We would also like to thank Linqing Liu for taking the time to answer our questions and discuss ideas.

Alf Niklas Håkonsen Kalmár & Borgar Rannem Lie

June 28, 2018

Contents

List of Code Snippets

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background and motivation

Summaries for articles or any kind of text is useful for many reasons. For example, there exist a wide amount of financial data, but most readers do not really need all the information provided in these texts, making short and concise summaries a lot more useful. News articles, papers and encyclopedias can be long and time-consuming to digest. Often, you are not interested in reading all the material you come by, but only some of it. Summaries can help to get a grasp of what they are about, and may include all the information you need, or help you decide on whether to read the entire text.

Generating summaries is time-consuming because you need to read the text, and you need to understand which parts are important. This makes it very attractive to try to automatically generate summaries using a computer program. This is also useful for journalists, as summarizing articles is a part of their work-flow. Automatic generation of summaries has been researched for quite a while, and there are generally two different ways of going about it. These are *Extractive text summarization* and *Abstractive text summarization*.

Extractive text summarization focuses on choosing how, e.g. paragraphs and important sentences, produces the original document in precise form. The implication of sentences is determined based on statistical and linguistic features [Moratanch and Chitrakala, 2017]. A summary that is produced using this method is basically a shortened version of the original document where the least important paragraphs or sentences are removed until a feasible size for a summary is achieved, and only the most important features are left.

Abstractive text summarization is the task of generating a short summary that captures the ideas of the original document. Abstractive, in this case, means that the summary is not a mere copy of a few sentences from the original document, but rather a compressed paraphrasing of the main contents, potentially using vocabulary not seen in the original document [Nallapati et al., 2016a].

Abstractive text summarization has gained a lot of attraction in recent years. In [Sutskever et al., 2014] and [Cho et al., 2014b] they show how LSTM networks can be used to map a sequence to another sequence (seq2seq), and [Bahdanau et al., 2014] improved this framework even further by incorporating what is known as an attention mechanism, which essentially allows the network to search for the most relevant information in the source text. [See et al., 2017] proposes a hybrid network between the seq2seq model and a pointer network, which they name Pointer-Generator Network. The advantage of this network is that it can copy

factual details from the input as well as generating words from a fixed vocabulary.

The standard supervised learning approach have seen promising results when used to train abstractive text summarization models. However, they are limited by the fact that they assume the ground truth to be provided at each time-step during training (teacher forcing). This is not the case at test time, where the previous generated word is provided instead. This creates a gap between training and testing, also known as ”exposure bias”. Two interesting directions that can be used for training abstractive text summarization models, which approach this issue in a flexible manner, is to apply reinforcement learning (RL) and generative adversarial networks (GAN).

Reinforcement learning has seen huge success in the domain of games, such as Go [Silver et al., 2016] and Atari [Oh et al., 2015]. Recently, reinforcement learning has also been used to improve sequence generation tasks, such as sentence simplification [Zhang and Lapata, 2017] and abstractive text summarization [Paulus et al., 2017]. Similarly, generative adversarial networks have seen success in the generation of real-valued data [Denton et al., 2015], and has recently been incorporated to generate sequences of discrete tokens [Yu et al., 2016].

In this thesis, we will be looking into abstractive text summarization, and how to improve upon the current state of the art models for generating abstract summaries by using reinforcement learning and generative adversarial networks. We explore what has been done so far in this regard and go into details about the different ways to improve the models using RL and GAN. There are a lot of details in how to train such models, and a significant portion of the information is not present in the papers describing them[1], making it hard to reproduce. We aim to make our work reproducible, making it easy for someone with an idea for using RL or GAN to test their theory. We explore how different training strategies and parameters affect the results, both quantitative and qualitative. We also include details about different ways to speed up the training process. Finally, we explore how different objective functions affect the results.

1.2 Research goals and questions

Goal Improving the generation of summaries by using reinforcement learning and generative adversarial networks.

The goal of this thesis is to improve an abstractive text summarization model by using reinforcement learning and generative adversarial networks. In this regard, we will explore different ways to implement the learning system as a whole, including sequence roll-out strategies, optimization objectives and parameters that affect the running-time, convergence and the generation of summaries. We aim to do an extensive study of the important details in the implementation, making the work easy to reproduce and extend.

To evaluate the results and measure improvement, we will use the ROUGE [Lin, 2004] evaluation metric, as well as studying the generated summaries. Convergence will be measured in terms of ROUGE scores, reward and loss.

Research question 1 How can reinforcement learning and generative adversarial networks for improving an abstractive text summarization model be implemented in an efficient manner, and what are the important factors that impact running-time?

[1] Results from papers not being easily reproducible is a common theme in the field of artificial intelligence [Kjensmo, 2017]

Research question 2 Which parameters, in the context of improving an abstractive text summarization model using reinforcement learning and generative adversarial networks, has a significant impact in regard to training stability and convergence?

Research question 3 How does different optimization objectives, as well as the combination of generative adversarial networks optimization and a static reward function impact the training convergence and the generation of summaries?

Research question 4 How does discriminator pretraining affect the convergence during adversarial training?

1.3 Research approach

As a starting point, we need an abstractive text summarization model that can be further improved by training with RL and GAN. In this regard, we implement a variation of the Pointer-Generator Network [See et al., 2017]. This model is pretrained using standard supervised learning. To train using GAN, we pretrain a CNN classifier [Kim, 2014] to be used as the discriminator.

To answer the research questions, we will explore multiple ways to implement the learning system as a whole, including pretraining, sequence roll-out strategies, optimization objectives and parameters that affect the running-time, convergence and the generation of summaries. We conduct extensive studies to find the important factors that makes training with RL and GAN for abstractive text summarization feasible.

1.4 Contributions

In this thesis, we explore how an abstractive text summarization model can be improved by training with RL and GAN. The experimental results demonstrate the importance of different aspects of the implementation. We also address some common time-consuming processes and suggest solutions on how to make them more efficient. Our complete codebase is available on GitHub[2]. The README file in the project folder explains how to set up the environment, how to preprocess the dataset and how to run the relevant parts. The repository also includes generated summaries from some of our trained models.

More specifically, in the context of abstractive text summarization, our contributions include, but are not limited to:

- An implementation of the Pointer-Generator Network [See et al., 2017] (with modifications) in PyTorch.
- An implementation of a general setup for both reinforcement learning and adversarial training, where the objective function and parameters can be easily interchanged.
- An extensive study of parameters and implementation details important for running-time, convergence and the generated summaries in the context of training with RL and GAN.
- Showing differences between optimization towards the objectives of ROUGE-1 and ROUGE-2, as well as adversarial training, where we optimize on the discriminator.

[2]GitHub repository: `https://github.com/borgarlie/TDT4900-Robo-Journalism`

- Studying the effect of optimizing towards a mixed objective of maximum likelihood estimation (MLE) and RL, as well as the effect of combining adversarial training with a static reward function (ROUGE).
- Exploring two different sequence roll-out strategies for reinforcement learning.
- Showing differences in parameter choices regarding beam search when generating summaries at test time.
- Proposing a batch-parallel sequence roll-out algorithm to speed up Monte-Carlo roll-out.
- Suggesting ways to optimize the learning system for reduced running-time without sacrificing convergence and training stability.

1.5 Thesis structure

The thesis is organized as follows. In Chapter 2, we start diving into some important background theory, explaining the general concepts and methods underlying the models used in the project. In Chapter 3, we briefly review the most relevant work in the current state of the art. Chapter 4 will introduce models, strategies, objectives and implementation details for our experiments. In Chapter 5, we report experimental results. Chapter 6 contains a general evaluation and discussion around the experiments and results, followed by answers to our research questions. Finally, in Chapter 7, we conclude our work, present possible improvements and discuss relevant future work.

Chapter 2

Background Theory

In this chapter, we will cover subjects that are relevant for the thesis. It serves as a gentle introduction to the different techniques and terminologies used.

The chapter starts off by introducing some basic building blocks. These include deep neural networks, long short-term memory, convolutional neural networks, reinforcement learning and generative adversarial networks. Following is a short introduction of regularization and optimization strategies. Finally, we present an overview of some of the evaluation metrics used in the thesis.

2.1 Artificial neural network

Artificial neural networks (ANN) is inspired by networks of biological neurons in the brain. A neuron, as illustrated in Figure 2.1, is a basic information-processing unit, and consist of a cell body, soma, a number of fibres called dendrites and a single long fibre called the axon. One single neuron has very little processing power, but an army of these can outperform todays supercomputers [Negnevitsky, 2005, p. 165-166].

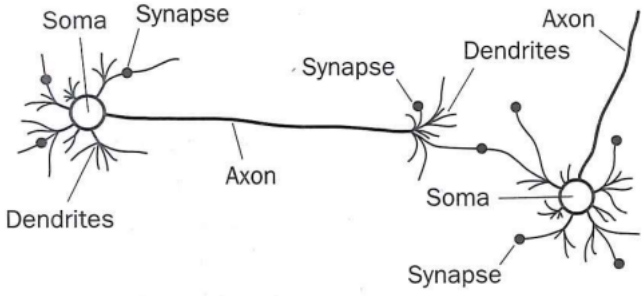


Figure 2.1: Illustrates a biological neuron. The figure is retrieved from [Negnevitsky, 2005, p. 166].

An artificial neural network is basically a number of simple processing units like neurons, and these units are connected with weighted links passing signals through the network. Each neuron can receive multiple input signals but will produce only one output signal which can split to other neurons or be the output for the network [Negnevitsky, 2005, p. 167-168]. Figure 2.2 is a graphic illustration of a typical ANN.

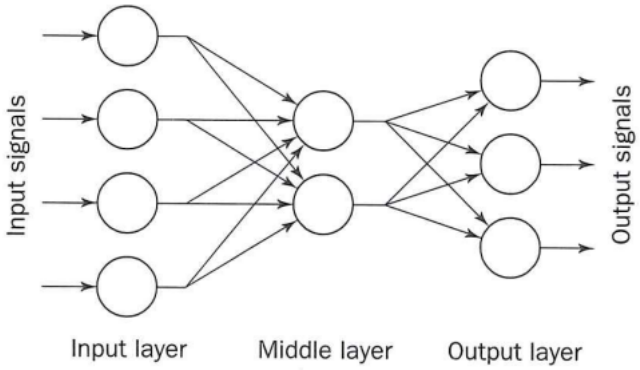


Figure 2.2: Architecture of a typical artificial neural network. The figure is retrieved from [Negnevitsky, 2005, p. 167].

A neuron can be viewed as a simple computing element. It receives several input signals, sums the signals and put the sum through an activation function. If the activation "fires", the neuron will send it as an output signal through the output links [Negnevitsky, 2005, p. 167-168].

2.1.1 Deep neural networks

As mentioned above, a single neuron is not very powerful, but many neurons connected in several layers can solve quite complex tasks. This type of network is called *deep neural networks (DNN)* or *deep feedforward networks*. Figure 2.3 shows an example off a typical DNN. It consists of an input layer followed by many layers with nonlinear hidden units and lastly an output layer. The layers between the input and output layer is called hidden layers because the data does not show the desired output for each of these layers [Negnevitsky, 2005, p. 175-176].

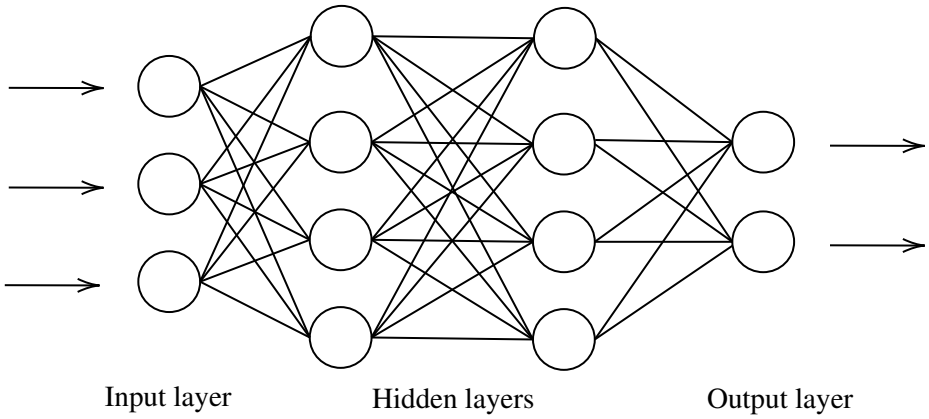


Figure 2.3: Typical architecture for deep neural networks[1]

The goal of a feedforward network is to approximate a function $f*$. The layers in the network can be viewed as a chain of functions that makes up the approximation of the entire function $f*$, such as $f* = f^3(f^2(f^1(x)))$ [Goodfellow et al., 2016, p. 164].

In a feedforward neural network, the input is propagated from the input layer to the hidden layers

[1]Deep neural network architecture inspired by: https://www.mathworks.com/content/mathworks/www/en/discovery/deep-learning/

and in the end produces the output, this procedure is called forward propagation.

For the network to learn the function, the weights that connects the neurons in the network have to be adjusted. This is a straight forward process when there is only one layer, because there is only one set of weights and they are directly responsible for the output error. When the network has multiple layers with nonlinear functions it is not so clear how to adjust the weights.

Luckily there is an easy way to do this. The idea is that every neuron in the network is responsible for the error, and thus the weights have to be adjusted accordingly. This idea is more commonly known as the back-propagation algorithm. The algorithm back-propagate information through the network and tries to minimize the error function in the weight space by using gradient descent. The information propagated backwards in the network is partial derivatives of the error or the cost function with respect to any weight w (or bias b) in the network [Chauvin and Rumelhart, 1995, p. 1-2][Goodfellow et al., 2016, p. 204-207][Rojas, 2013, p. 161]. Two examples of cost functions, also known as loss functions, are negative log loss (NLL), as seen in Equation 2.1 and binary cross entropy (BCE), as seen in Equation 2.2. In both equations, t is the target value, o is the predicted output value and N is number of examples in the minibatch. In Equation 2.1, M is the set of classes.

$$loss(o,t) = -\frac{1}{N}\sum_{i\in N}\sum_{j\in M} t_{ij} * log(o_{ij}) \tag{2.1}$$

$$loss(o,t) = -\frac{1}{N}\sum_{i\in N}(t(i) * log(o(i)) + (1-t(i)) * log(1-o(i)) \tag{2.2}$$

2.2 Recurrent neural network

Recurrent neural networks (RNN) is a specialized neural network that operates on time series and sequences of data. Unlike a feedforward neural network, an RNN have connections to itself as shown to the left in Figure 2.4. To the right, the same network is illustrated, only unfolded. It is called recurrent because it performs the same task on every element in the input sequence, and the output is depending on the previous computations [Martens and Sutskever, 2011][Goodfellow et al., 2016, p. 367].

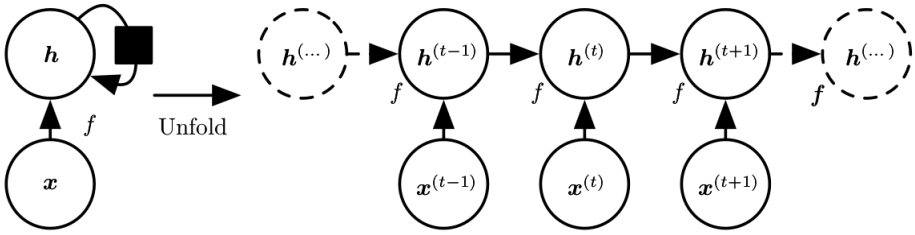


Figure 2.4: The architecture for RNN. The figure is retrieved from [Goodfellow et al., 2016, p. 370].

Since RNN's are doing the same task over and over again, they share the same parameters across time-steps. This reduces the number of parameters the network has to learn, and thus more easily converges on the training data [Sutskever, 2013, p. 9-10] [Goodfellow et al., 2016, p. 367].

One advantage a RNN have over a normal feedforward network is that it can operate on different input lengths. This makes it more flexible and a good choice for tasks where the data varies in length, such as natural language processing (NLP) [Goodfellow et al., 2016, p. 367-369].

Another advantage is that the network can retain information from previous steps. Retaining information gives opportunity to exhibit dynamic temporal behavior. This can be thought of as if the network has a memory which captures what have been done so far. In theory a RNN can use this memory for arbitrarily length of sequences, but in practice, this is not the case [Goodfellow et al., 2016, p. 367-369].

Long-term dependencies happen often in RNN's because steps that are far apart can strongly depend on each other. This gives rise to a big problem for RNN's, which is called vanishing/exploding gradient problem. The reason this happens is because as the network receives long input sequences, the gradients propagated backwards often becomes very large or very small, meaning that either the gradients explodes or vanishes [Sutskever, 2013, p. 10-11] [Goodfellow et al., 2016, p. 396-398]. There are some common approaches to solve this issue, and a popular solution is to use long short-term memory (LSTM). This will be covered more in depth in Section 2.2.1.

Back-propagation can also be used for training RNNs, but since the network uses the same set of parameters for each time-step some adjustments have to be made. The gradient for each output is not only dependent of the current calculations, but also the previous steps. This means that to find the gradients for the parameters, the algorithm has to back-propagate gradients through time, and therefore it is called back-propagation through time (BPTT) [Sutskever, 2013, p. 23] [Goodfellow et al., 2016, p. 374-376].

2.2.1 Long short-term memory

As mentioned above, RNNs struggle with long-term dependencies, and LSTMs are a special kind of RNNs that are capable of dealing with these dependencies. Hochreiter & Schmidhuber [Hochreiter and Schmidhuber, 1997] were the first to introduce this idea in 1997.

LSTMs are designed to deal with these long-term dependencies which gives them the natural behavior of remembering information for long periods of time. LSTMs are similar to normal RNNs in that they both have a chain of repeating modules. The difference is that normal RNNs have only one single layer which gives rise to a very simple structure, but LSTMs have four layers interacting in a complex way[2]. Figure 2.5 illustrates a typical LSTM architecture, and compared to the RNN architecture in Figure 2.4, the differences are quite clear.

[2]Taken with permission from Colah's blog about LSTMs: http://colah.github.io/posts/2015-08-Understanding-LSTMs/

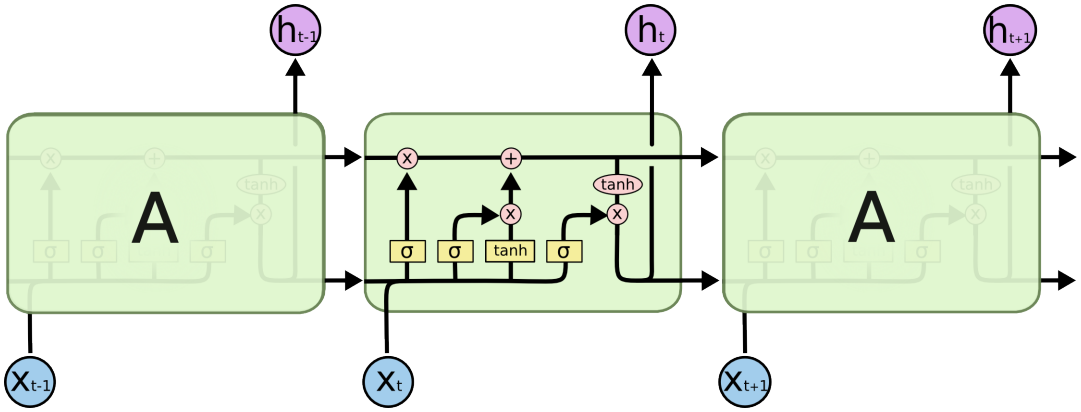


Figure 2.5: The architecture for a LSTM network[3]

The core idea of LSTMs is that they have a cell state. This state allows information to flow easy along the chain unchanged (the top line in Figure 2.5). The module uses gates to control how much information that should be allowed to get through to the next time-step. Figure 2.6 illustrate the structure of a gate in a LSTM module. This is typically a sigmoid layer which outputs a number between 0 and 1. This number essentially tell the module how much information to let through, where 0 means nothing at all and 1 means everything [Yu et al., 2015][Gers et al., 1999].

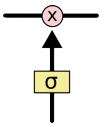


Figure 2.6: A gate in the LSTM architecture[4]

LSTMs have 3 different gates, where all of them are for controlling and protecting the information. The first gate, forget gate layer, is a simple sigmoid layer to decide what information to keep and what to throw away. The next gate, the input gate layer, is the layer which decide what new information that are going to be stored in the cell state. This gate has two parts, first a sigmoid layer that decides which values are going to be updated, and second, a tanh[5] layer that generates a candidate vector that could be added to the cell state. These two parts are then combined before a potential update to the state. Lastly is the output layer, which is based on the cell state. First, a sigmoid layer is used to decide what parts of the cell state is going to be output. Then, a tanh layer is used on the cell state to force the values in the range -1 to 1, and combines it with the sigmoid so the output gets filtered to what is desired [Yu et al., 2015][Gers et al., 1999].

One limitation of regular LSTM is that they are only using the previous context. Processing the data in both directions with two separate hidden layers by two different networks allows the model to access long-term dependencies in both directions. These hidden layers are provided to the same output layer. This approach is called Bidirectional LSTM (BLSTM). In situations where the whole input sequence is available from the beginning there is no reason to not exploit both future context and history context together

[3]LSTM architecture taken with permission from: http://colah.github.io/posts/2015-08-Understanding-LSTMs/img/LSTM3-chain.png

[4]A LSTM gate taken with permission from: http://colah.github.io/posts/2015-08-Understanding-LSTMs/img/LSTM3-gate.png

[5]Hyperbolic tangent [Goodfellow et al., 2016, p. 191] is a trigonometric function, with output values between -1 and 1.

[Graves and Schmidhuber, 2005] [Graves et al., 2013] [Yu et al., 2015]. Figure 2.7 gives a clear view of the BLSTM architecture and what information that is available at any given step.

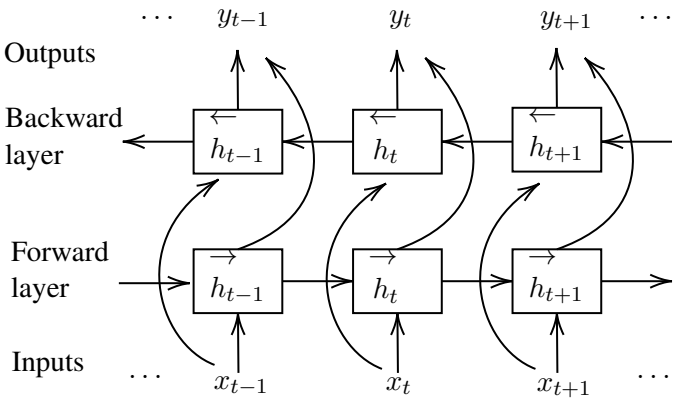


Figure 2.7: Typical architecture for a BLSTM. The figure is inspired by [Mousa and Schuller, 2016, p. 2837].

2.2.1.1 Gated recurrent unit

Gated recurrent unit (GRU) was introduced by [Cho et al., 2014a], and is a variation of LSTM. The difference can be seen in Figure 2.8. The forget gate and input gate is combined into a single "update gate". The cell state and the hidden state is also merged together. This gives rise to a simpler model, and therefore have become a very popular variation to use [Goodfellow et al., 2016, p. 407-408].

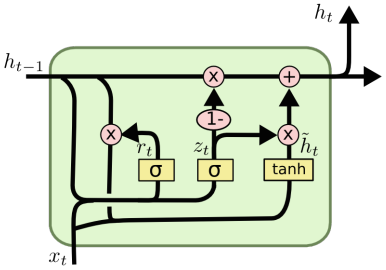


Figure 2.8: An illustration of the GRU architecture[6]

2.3 Convolutional neural network

Convolutional neural network (CNN) [LeCun et al., 1998] as the name suggest gets their name from convolution. A formal definition of convolutional neural networks is given as [Goodfellow et al., 2016, p. 326]:

> convolutional networks are simply neural networks that use convolution in place of general matrix multiplication in at least one of their layers.

[6]The GRU architecture taken with permission from: http://colah.github.io/posts/2015-08-Understanding-LSTMs/img/LSTM3-var-GRU.png

CNN have led to many breakthroughs in DNN's in the last few years. CNN's exceed in domains with pattern recognition, such as time-series data, images and voice recognition. A simplistic view of CNN, is that it uses many identical copies of the same neurons. One advantage of having many copies of the same neuron is that large models can be trained while keeping the actual number of parameters small.

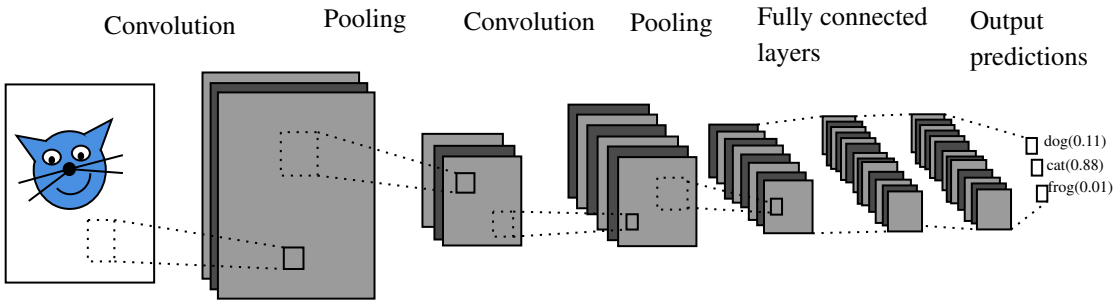


Figure 2.9: A typical convolutional neural network architecture[7]

The explanation above is very simple and leave out many details about the CNN. Figure 2.9 is a more correct view of a typical CNN. A CNN have one or more of the following four operators:

- Convolution
- Non Linearity function (Typically ReLU)
- Pooling or Sub Sampling
- Classification (A fully connected layer)

These four operators are the building blocks of most CNN, thus a good understanding of these is important to truly understand CNN. The primary focus of the convolution step is to extract features from the input. It can be thought of as a fixed grid moving over the input and doing the same operation everywhere. This operation is typically given as shown in Equation 2.3.

$$s(t) = (x * w)(t) \tag{2.3}$$

The first argument, x, is often referred to as the input, and the second, w, is called the kernel. The output of this operation is often called a feature map [Goodfellow et al., 2016, p. 327-328]. Figure 2.10 gives a clear illustration of the convolution step. The kernel, upper right in the figure, can be viewed as performing a dot product over the input data, upper left in the figure. The kernel moves over the input with stride steps, until it have covered the whole input data. The reason why the output is called "a feature map" is because it will only give output values if the kernel detects the features it is trained to detect. Thus, it is common to use multiple kernels on the same input to detect different features. Figure 2.11 gives a clear understanding of this effect.

[7]The figure is inspired by: http://www.wildml.com/2015/11/understanding-convolutional-neural-networks-for-nlp/

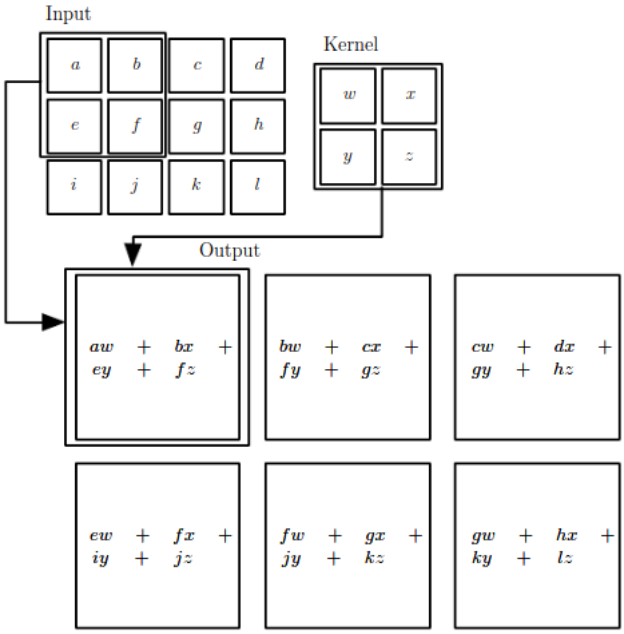


Figure 2.10: Illustrating a 2D convolution. The figure is retrieved from [Goodfellow et al., 2016, p. 330].

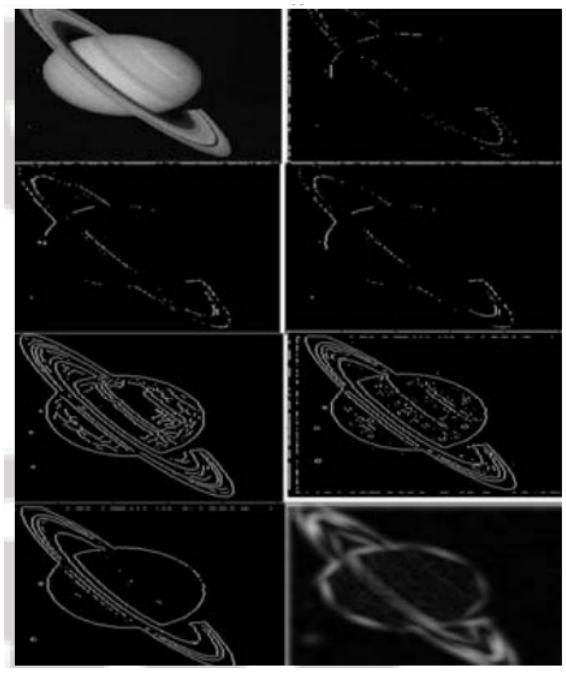


Figure 2.11: Features detected by different filters. The figure is retrieved from [Kaur and Kaur, 2014].

The next step in a CNN is to perform a nonlinear function on the feature maps. The purpose of this function is to introduce non-linearity in the network. This is necessary because without a nonlinear function it is impossible to learn something that is not linear. Most things in the world are not linear, and therefore non-linearity have to be introduced in the network. ReLU have been

found to give good results because of it's nice derivative properties, which always is one for non-negative numbers. Other activation functions could be used as well, such as sigmoid or tanh.

The third step is the pooling step. Spatial pooling reduces the dimensionality of each feature map while retaining the most important information. This can be done with many different functions, such as max, average, sum and so on [Goodfellow et al., 2016, p. 335-336].

Lastly is the fully connected part. This typically consists of multiple layers with a softmax[8] activation on the output layer. As the name suggest, this layer have connections to every neuron in the previous layer. The neurons from the previous layers include high level features of the input, and the fully connected layer use this to classify the input. It can also be used to learn nonlinear combinations of the features from the previous layer.

The main motivation to use CNN is sparse interactions, parameter sharing and equivariant representation. It also provides a mean for working with variable sized input.

A traditional neural network will use matrix multiplication on a matrix of parameters, with a separate parameter describing the interaction between each unit. This means that every output unit interact with every input unit. A CNN typically have sparse connections. This is achieved by having a kernel size that is smaller than the input. One advantage of having sparse connections is that fewer parameters are required, reducing the memory requirements and improving the statistical efficiency for the model. The number of operations for computing the output is also reduced [Goodfellow et al., 2016, p. 330-331].

The meaning of parameter sharing is that parameters are used more than once in a model. As mentioned above, the kernel is used over the entire input. This way, the CNN use the same parameters several times in the model. CNN networks only learn one set of parameters instead of a separate set of parameters for every location, like typical neural networks do. This will also reduce the memory requirements for the model. Therefore, CNN is several times more efficient than dense matrix multiplications, with respect to memory requirements and statistical efficiency [Goodfellow et al., 2016, p. 331-334].

This form of parameter sharing give the layers a property called *equivariance* to translation. This means that if the input changes, the output have to change in the same way. Thus, the model can produce the same representation of the input, independent of when it receives it in time series or if objects are moved in the input. However, this does not mean it can detect other transformations, such as rotation of objects or a change in scales for a picture [Goodfellow et al., 2016, p. 334-335].

2.3.1 Multilabel classification using CNN

Multilabel classification is a generalization of multiclass classification. The problem is to map the inputs X into a fixed-length vector Y, where every element in Y is either 0 or 1 depending on whether X fits that particular label.

Traditional approaches to this problem includes boosting, k-nearest neighbors, decision trees, kernel methods and neural networks (BP-MLL[9]). Regarding sentence level multilabel classification, [Kim, 2014] has achieved state of the art results using pretrained word vectors and CNNs. The model is especially attractive because of its simplicity[10].

[8] Softmax is a function that squashes the values into a vector with values between zero and one, that sums up to one.
[9] BP-MLL is a back-propagation algorithm for multilabel learning [Zhang and Zhou, 2006].
[10] There are already several implementations of the architecture in the paper.

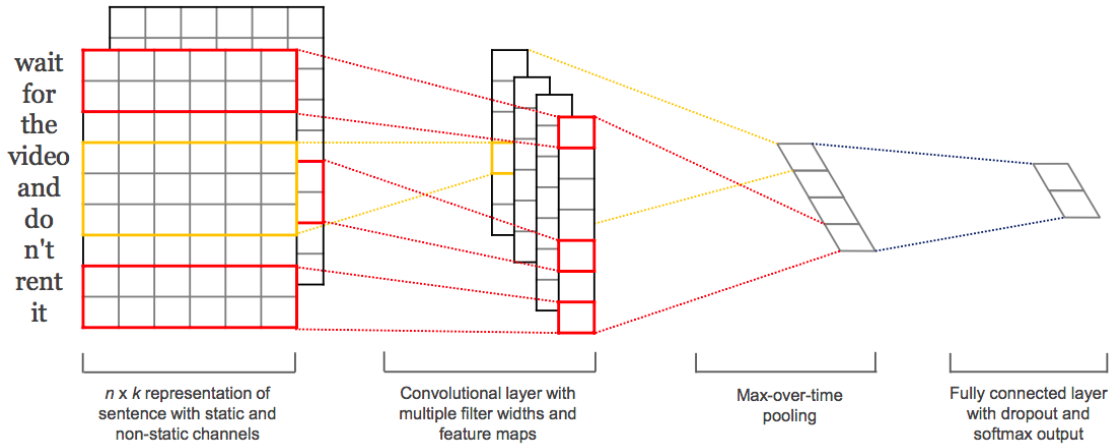


Figure 2.12: CNN classification with two channels for an example sentence. The figure is retrieved from [Kim, 2014].

The model, as shown in Figure 2.12, takes an input X that is a concatenation of the words in the sentence. These words are embedded using some word-embedding[11] scheme[12], and put through a convolution layer with multiple filter widths, producing multiple feature maps. A max-pooling layer is used to locate the most important features. Finally, a fully connected softmax layer is used to output the probability distribution over the possible labels. Binary classification can be achieved by simply letting the last fully connected layer output one single value[13].

2.4 Reinforcement learning

Reinforcement learning (RL) learns how to map from a situation to an action so it maximizes a reward signal. The learner has to discover by itself which actions gives the highest reward, and do so by trial and error. Sometimes the actions do not only impact the current reward, but also the next one, and all subsequent ones. These two features, trial-and-error and delayed rewards, are the most characteristic features of RL [Sutton and Barto, 1998, p. 1].

Compared to other machine learning strategies, such as supervised and unsupervised learning, RL is quite different. In supervised learning the objective is to generalize, so that it acts correctly in new situations. This type of learning is not suitable for learning from interaction with the environment. It is often difficult and impractical to obtain examples that is representative of all the situations the agent has to act in. In situations like this, one would expect that an agent learning from its own experiences would achieve much better performances. This should not be confused with unsupervised learning, that tries to find a hidden structure in the data. RL is trying to optimize toward a reward signal it receives from the environment, and are not trying to find a hidden structure [Sutton and Barto, 1998, p. 2].

There is a lot of challenges with machine learning problems, but the trade-off between exploration and exploitation is unique for RL. In order to obtain a high reward, the agent must prefer actions

[11] word-embedding is a technique to map words or phrases from a vocabulary to vectors of real numbers [Levy and Goldberg, 2014].

[12] In the paper, they use Word2Vec: `https://code.google.com/archive/p/word2vec/`

[13] Using one value as output is essentially the same as using two output values since the network will learn that $output_2 = 1 - output_1$ as long as the true labels are one or the other and not both.

that it has tried before and that it knows for sure produces good rewards. The only way the agent can learn this is by exploring new actions. In other words, the agent has to exploit actions that it knows to give high rewards, and it also has to explore other actions to get better action selections in the future. The dilemma is that it cannot only follow one of the directions without failing at the task. It will have to explore several actions and progressively move forward by favoring the actions that appears to be the best ones [Russell and Norvig, 2016, p. 839-842].

2.4.1 Reinforcement learning structure

In a RL system there can be found four important sub elements, a *policy*, a *reward signal*, a *value* function and optionally a *model* of the environment [Sutton and Barto, 1998, p. 5].

Policy describes what the agent should do in a given situation. Roughly, one can say it maps perceived states to actions that the agent performs in the perceived states. The policy for an agent can be a simple lookup-table, or it can be an extensive computational process, such as state-space searches. Policy could alone be enough to determine behavior, and is therefore at the core for a RL agent [Sutton and Barto, 1998, p. 5].

The *reward signal* defines the goal for the RL agent. For every time-step the environment will give the agent a single number, called a *reward*. The RL agent's objective is to maximize the total reward in the long run. Therefore, this reward signal will directly tell the agent which states are good and bad. Reward signals are the primary source for changes to the policy. If the agent receives a low reward after following its policy in one state it may change its policy and try something else next time [Sutton and Barto, 1998, p. 5].

The reward signal gives the agent the immediate reward for one action, but a *value function* tells the agent what the expected future rewards will be when choosing that action. The value function takes into account the states that follows when giving a value for the action, and therefore represents the long-term desirability of the actions [Sutton and Barto, 1998, p. 5].

The last element for some RL systems is a *model* of the environment. This model is used to make inferences about how the environment behave and is often used in planning. The model considers actions and their outcomes before actually experiencing them, and by doing so comes up with the best course of action. RL methods that uses models and planning are called model-based methods, and simpler methods that uses trial and error are called model-free methods [Sutton and Barto, 1998, p. 5-6].

2.4.2 Reinforcement learning goal

Markov decision processes (MDP) is a classical formalization of sequential decision making where actions influence not only the next reward, but also subsequent states and rewards. MDPs is the formal definition of a RL problem and is the idealized form where precise theoretical statements can be made. In the MDP setting the learner and decision maker is called the agent. The environment is defined as the "thing" the agent interacts with, and consist of everything outside of the agent. The agent interacts with the environment continually through actions, and the environment responding to the actions. The environment also gives rise to rewards that the agent seeks to maximize over time through its choices of actions [Sutton and Barto, 1998, p. 37]. The relationship between the agent and the environment is illustrated in Figure 2.13.

The notion of "how good" is often used in context with RL problems. It is defined as the future

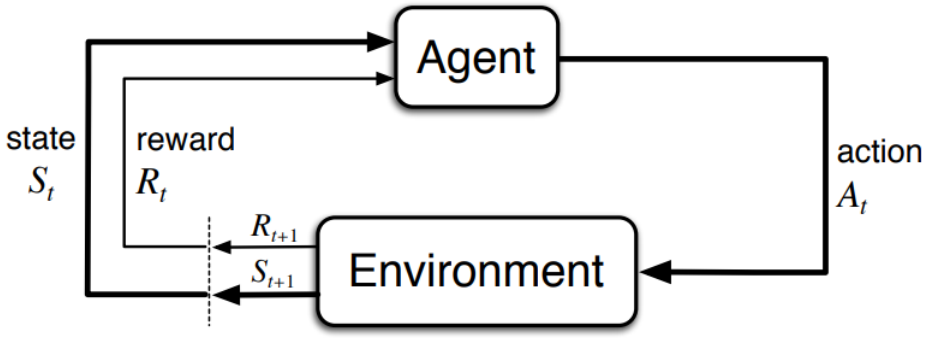


Figure 2.13: The agent-environment relationship in a MDP. The figure is retrieved from [Sutton and Barto, 1998, p. 38].

rewards that can be expected by being in one particular state. This is found by estimating the value function for a sequence of possible future states from that state. The sequence of future states is of course dependent on the agent's actions, thus, this sequence is predicted with respect to particular ways of acting, called policies. Policy is a mapping from states to probabilities of selecting each possible action. Policy is given as π, and if the agent follows the policy π at time t, then $\pi(a|s)$ is the probability that $A_t = a$ if $S_t = s$. The value of state s under a policy π, expressed as $v_\pi(s)$ is the expected future reward when starting in s and following the policy π thereafter [Sutton and Barto, 1998, p. 46-47]. This is formally defined in Equation 2.4:

$$v_\pi(s) \doteq \mathbb{E}_\pi[G_t|S_t = s] = \mathbb{E}_\pi\left[\sum_{K=0}^{\infty} \gamma^k R_{t+k+1}|S_t = s\right], \forall s \in S, \tag{2.4}$$

Here, $E_\pi[\cdot]$ denotes the expected value given that the agent follows policy π, and t is any time-step. G_t is an expression of the discounted future rewards, and γ is a discount rate with a value between 0 and 1.

A policy is better than other policies if its expected return is greater than the others for all states. An optimal policy is a policy that always is better than or equal to all other policies. There might be more than one optimal policy for a reinforcement task, and all of them are considered as a valid solution [Sutton and Barto, 1998, p. 50].

2.4.3 Policy iteration

The key idea of RL is the use of value functions to search for good policies. By finding a value function v_* that satisfies the Bellman optimality equation, it is easy to obtain an optimal policy. The optimal value function can be found by solving Equation 2.5 [Russell and Norvig, 2016, p. 656-658].

$$v_*(s) = \max_a \sum_{s',r} p(s', r|s, a)[r + \gamma v_*(s')] \tag{2.5}$$

If one action is clearly better than all the others, then the expected value for the states do not have to be exact. This insight leads to an algorithm for finding the optimal policy, called *policy*

iteration. This algorithm alternates between the following two steps, starting with a start policy π_0 [Russell and Norvig, 2016, p. 656-658].

1. Policy evaluation: which finds the expected value, U_i for each state if policy π_i is used.
2. Policy improvement: calculates a new policy π_{i+1} by one step lookahead (Equation 2.6) and using the expected value U_i from the policy evaluation step.

$$\pi^*(s) = \arg\max_{a \in A(s)} \sum_{s'} P(s'|s,a)U(s') \tag{2.6}$$

Each policy update is guaranteed to be a strict improvement over the previous policy. Since a finite MDP has only a finite number of policies, thus the policy and value function must converge to an optimal solution [Russell and Norvig, 2016, p. 658].

2.4.4 Value iteration

Another algorithm that can be used to find the optimal policy is called *value iteration*. It takes advantage of the fact that it is possible to truncate the policy evaluation step. Value iteration truncate the policy evaluation after one update of each state without losing the convergence guarantees. The Bellman equation is another way of viewing the value iteration algorithm. It has turned the Bellman equation to an update rule called *Bellman update*, given in Equation 2.7. The algorithm iteratively updates the expected value for each state with the value from all its neighboring states [Russell and Norvig, 2016, p. 652-656].

$$U_{i+1}(s) \leftarrow R(s) + \gamma \max_{a \in A(s)} \sum_{s'} P(s'|s,a)U_i(s') \tag{2.7}$$

2.4.5 Policy gradient methods

Another approach to RL problems is to learn a parameterized policy that can choose actions without using a value function. Value functions can be used to learn the parameterized policy, but it is not required during the action selection. One way to learn the policy parameter is to use the gradient of some performance measure $J(\theta)$ with respect to the policy parameter. These methods try to maximize their performance, thus their updates are approximate gradient ascent in J, as shown in Equation 2.8 [Sutton and Barto, 1998, p. 265].

$$\theta_{t+1} = \theta_t + \alpha\widehat{\nabla J(\theta_t)} \tag{2.8}$$

Where $\widehat{\nabla J(\theta_t)}$ is a stochastic estimate whose expectation approximates the gradients of the performance measure with respect to its argument θ_t. Methods that follows this schema are called *policy gradient methods*, and methods that learns approximations to both policy and value functions are called *actor-critic methods*. Where the "actor" is a reference to the learned policy and the "critic" refers to the learned value function [Sutton and Barto, 1998, p. 265].

2.4.6 REINFORCE: Monte Carlo policy gradient

The stochastic gradient ascent (Equation 2.8) requires a way to obtain samples such that the expectation of the sample gradient is proportional to the actual gradient of the performance measure as a function of the parameter. It only needs to be proportional to the gradient because any constant of proportionality can be absorbed into the step size α (the learning rate) [Sutton and Barto, 1998, p. 270-271]. The expectation of the gradient is expressed as:

$$\nabla J(\theta) = \mathbb{E}_{\pi}\left[G_t \frac{\nabla_\theta \pi(A_t|S_t,\theta)}{\pi(A_t|S_t,\theta)}\right] \tag{2.9}$$

where G_t is the discounted future reward, A_t is the sampled action and π is the target policy. This expectation of the gradient expression can be used to instantiate the generic stochastic gradient ascent, which yields the update:

$$\theta_{t+1} \doteq \theta_t + \alpha G_t \frac{\nabla_\theta \pi(A_t|S_t,\theta_t)}{\pi(A_t|S_t,\theta_t)} \tag{2.10}$$

The update algorithm (Equation 2.10) is called REINFORCE [Williams, 1992]. Each update is proportional to the product of a reward G_t and the gradient of the probability of taking the action divided by the probability of taking that action. The gradient vector is the direction in parameter space that most increases the probability of repeating the action A_t on future visits to state S_t [Sutton and Barto, 1998, p. 270-271].

The REINFORCE algorithm uses all the future rewards up until the end of the episode[14]. This makes the algorithm a Monte Carlo algorithm and only works for the episodic case with all updates made in retrospect after the episode is completed [Sutton and Barto, 1998, p. 270-271].

2.5 Generative adversarial networks

Generative adversarial networks (GANs) were first introduced by [Goodfellow et al., 2014] in 2014. In the paper they propose GANs as an adversarial framework, where the generation process is improved by having an adversary, called the discriminator. The discriminator learns to distinguish real samples from fake samples generated by the generator. This framework is shown in Figure 2.14, where the generator generates fake data G(z) given some noise z, and the discriminator tries to distinguish between the generated data and the real data x. The dotted line shows that the prediction is back propagated to both the discriminator and the generator. GANs have shown some promising results lately for generation of images, 3D-models and music, among other things.

[14] "Episode" is used as a generic term, and can for instance be an entire game sequence.

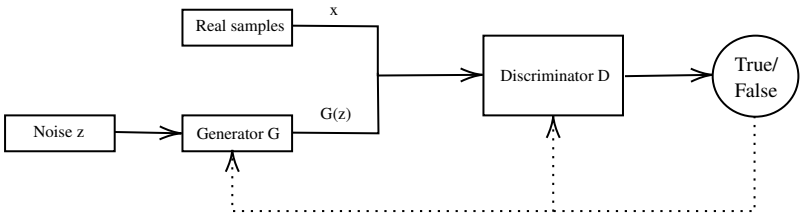


Figure 2.14: Illustration of a general GAN framework. The figure is inspired by [Sharmilan and Chaminda,].

One common analogy to the real world is the competition between counterfeiters and the police. The counterfeiters try to forge currencies, and the police tries to detect counterfeits currency, here the competition makes both parts improve their skills until the counterfeits currencies is indistinguishable from the real ones.

This framework is really beneficial for generative models. To better understand why this framework is so good for generative models it is essential to understand how generative algorithms work. Discriminative algorithms try to predict a label given the input data. Which can be expressed as: $P(y|x)$, where y is the label and x is the features for the input data. Generative algorithms can be viewed as the opposite, they try to predict features given a label, expressed as: $P(x|y)$. It is clear that this is a hard task, and in this adversarial framework the generator gets help from the discriminator to generate more realistic samples.

2.5.1 Adversarial training

GAN training is done in a minimax two-player game setting. Where the generator, G, tries to learn a data distribution, and the discriminator, D, estimates the probability that the sample comes from the real data distribution or not. The generator tries to maximize the probability that the discriminator makes mistakes, by following the value function in Equation 2.11. The optimal solution is a Nash equilibrium [Nash et al., 1950], and that happens when the discriminator is no better than a random guess, which means that the discriminator cannot discriminate between real and fake samples [Goodfellow et al., 2014].

$$\min_G \max_D V(D,G) = E_{x\sim p_{data(x)}}[logD(x)] + E_{z\sim p_z(z)}[log(1-D(G(z)))] \tag{2.11}$$

Both models are usually trained by backpropagating the loss for each model. The training alternates between training the generator and the discriminator. For each training epoch, the generator generates fake samples with its current weights. These fake samples are then combined with the training data and used for training the discriminator, as a binary classifier. By training the discriminator like this it should be able to classify training data as real samples and the generated data as fake. The training of the generator is done by creating new sets of samples, which is propagated through the whole network. The goal for the generator is to minimize $log(1-D(G(z))$, so it fools the discriminator. To accomplish this, the discriminator weights have to be frozen, so the generator can adjust its weights such that the current classifier would classify the generated samples as real ones [Goodfellow et al., 2014].

2.6 Regularization

Machine learning problems usually train on a dataset that is a subset of the whole domain. The goal is to learn the representation of this dataset. Overfitting is a term that says a network learns a representation too well, and it exist another representation that performs better on the entire distribution of data [Mitchell, 1997, p. 67]. I.e. the network does not generalize well. Generalization is a term that says how well a network performs on unseen data.

To overcome the issue of overfitting, many strategies have been developed to improve the performance on unseen data. Strategies that aims to achieve this behavior is called regularization.

Goodfellow et al defines regularization in their *Deep Learning* book [Goodfellow et al., 2016, p. 117] as:

> Regularization is any modification we make to a learning algorithm that is intended to reduce its generalization error but not its training error.

Some regularization strategies put extra constraints and penalties on the machine learning model. These constraints and penalties might be designed to incorporate prior knowledge, or they can express a preference for simpler models. Another example of regularization is the use of parameter norm penalization [Goodfellow et al., 2016, p. 226]. This regularization approach aims at limiting the capacity for the model, adding a penalty $\Omega(\Theta)$ to the loss function, which leads to the new function displayed in Equation 2.12:

$$\hat{L}(\Omega, C) = L(\Omega, C) + \alpha\Omega(\Theta)^{[15]} \tag{2.12}$$

L^1 and L^2 regularization are two of the most well-known methods of parameter norm penalization. They are very much alike, the difference is that L^2 is the sum of squared weights across the whole network while L^1 is the sum of absolute values [Goodfellow et al., 2016, p. 227-233].

Even though they seem very similar, they can have different effects. L^1 can have a sparsity effect, because it drives many of the weights to zero. This happens if most of the weights have small absolute values. L^2 can have a more stable update scheme, because the penalty scales linearly with the weights [Goodfellow et al., 2016, p. 227-233].

2.6.1 Dropout

Dropout is also a well-known regularization strategy. It provides a computationally efficient, yet a powerful method to regularize a model. One way to look at it, is that it works as a bagging method. Bagging is an ensemble method that combines multiple hypotheses to achieve better performance on new inputs. Bagging does this by training multiple models and combine these models (usually by majority vote) to get a prediction on each test example. Dropout effectively removes some units of the network by multiplying the output with zero [Goodfellow et al., 2016, p. 255]. This will in a way result in training multiple networks, as illustrated in Figure 2.15.

The term *dropout* refers to dropping out some units in the network. A common approach is to multiply the output with zero, but any operation that drops the unit can be used. For every

[15] Slightly altered equation from [Goodfellow et al., 2016, p. 226] for easier understanding. Regularized loss function ($\hat{L}$), L = loss function, C = cases, Θ = parameters of the estimator, Ω = penalty function and α = penalty scaling factor.

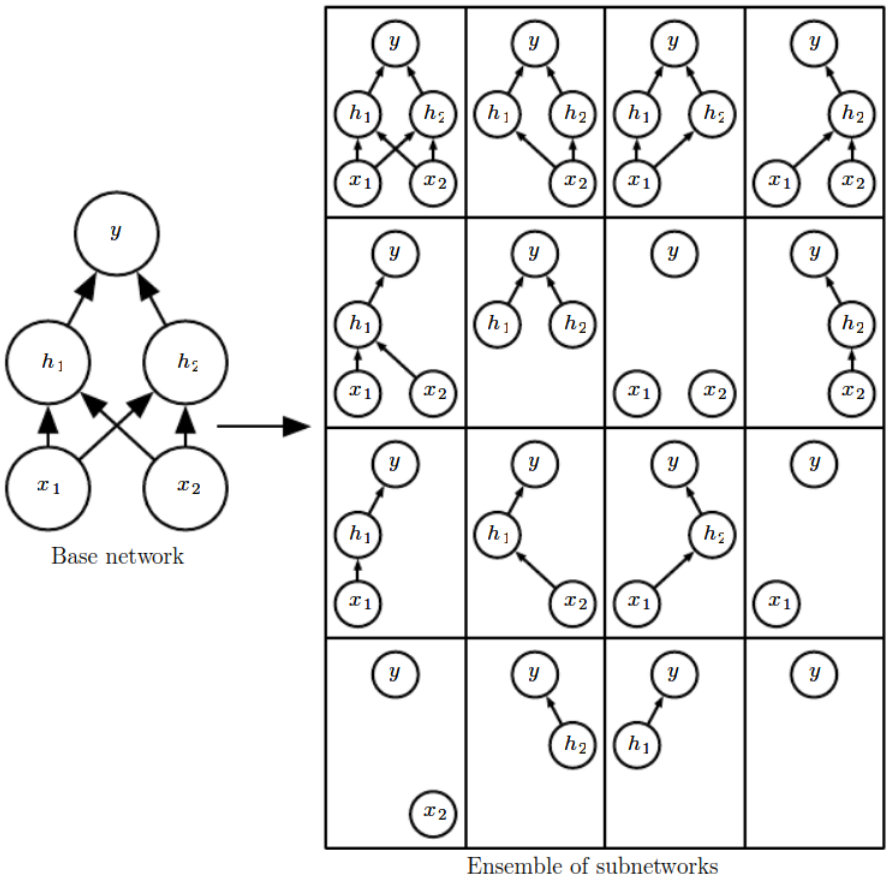


Figure 2.15: Visualization of the dropout functionality. The figure is retrieved from [Goodfellow et al., 2016, p. 256].

run, which units that get dropped is random, and is usually determined by a probability, p. Dropout is not exactly like bagging, because in bagging you train several independent models, but dropout shares the same parameters. Parameter sharing makes it possible to represent a large number of models with reasonable amount of memory [Goodfellow et al., 2016, p. 255-257] [Srivastava et al.,].

There exist multiple other regularization techniques, such as data augmentation, multitask learning and sparse representations, but for this thesis, only a general understanding of the regularization concept and the methods mentioned in this chapter will suffice.

2.7 Optimization techniques

Neural network training is the most difficult optimization problem involved in deep learning. It is quite common to spend several days to weeks and even months on solving a single instance of the neural network training problem. Luckily for us a specialized set of optimization techniques have been developed for solving this. This section will cover some of the most important optimization techniques for this thesis.

2.7.1 Stochastic gradient descent

Stochastic gradient descent (SGD) is probably the most used optimization algorithm for machine learning today. SGD is a stochastic optimization algorithm, which means that each update is not only based on one example but a mini-batch of examples. It finds gradients for each examples in the mini-batch and uses the average gradient to get an unbiased estimate [Goodfellow et al., 2016, p. 290].

The very basic gradient descent algorithm have a fixed learning rate, ϵ, but for SGD it is necessary to gradually decrease the learning rate over time. This is because the SGD gradient estimator introduces a source of noise that does not vanish. This introduces much variance in the update, so the learning rate also have to be small. Choosing the right learning rate is an art form and very difficult. There exist many strategies and approaches to achieve good result, such as, evaluating a holdout set after each epoch and anneal the learning rate when the change in objective between epochs is below a small threshold. Another approach is to decay the learning rate linearly until iteration τ, as shown in Equation 2.13, where $\alpha = \frac{k}{\tau}$. All the different approaches yield good results for some cases, but there is no single approach or strategy that works for all cases (no silver bullet) [Goodfellow et al., 2016, p. 290-292].

$$\epsilon_k = (1 - \alpha)\epsilon_0 + \alpha\epsilon_\tau \tag{2.13}$$

2.7.2 Algorithms with adaptive learning rates

As discussed above, the learning rate is the most crucial hyperparameter for the optimization algorithm. This is because it has a huge impact on the performance of the algorithm. The cost function is often highly sensitive to some directions in parameter space and not so much to others. It is possible to address this issue by using separate learning rates for each parameter and automatically adapt these learning rates throughout the course of learning.

2.7.2.1 AdaGrad

The AdaGrad algorithm is an incremental method for adapting the learning rate of all model parameters. It does so by scaling them inversely proportional to the square root of the sum of all the historical squared values of the gradient [Duchi et al., 2011]. Parameters with the largest partial derivative of the loss have a correspondingly rapid decrease in their learning rate, while parameters with small partial derivatives have a relatively small decrease in their learning rate [Goodfellow et al., 2016, p. 303].

2.7.2.2 Adam

Adam [Kingma and Ba, 2014] is another adaptive learning rate optimization algorithm. It has gotten its name from ”adaptive moments”. Momentum in Adam is incorporated directly as an estimate of the first-order moment (with exponential weighting) of the gradient. Adam includes bias correction to the estimates of both the first-order moments and second-order moments to account for their initialization at the origin. Adam is regarded as a robust optimization algorithm [Goodfellow et al., 2016, p. 305-306].

2.7.3 Gradient clipping

Figure 2.16 illustrates a common problem for neural networks with many layers; They can have regions in parameter space that are extremely steep and resembles cliffs. In regions like this the gradient update can move the parameters great distances and jumping off the cliff altogether.

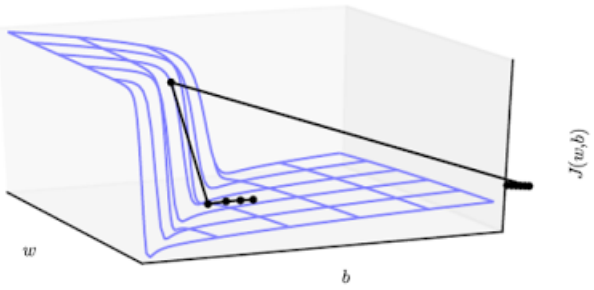


Figure 2.16: Illustration of a cliff in the parameter space. The long line shows how normal gradient descent suggest an update in such regions. The shorter line shows the updates when using gradient clipping. The figure is retrieved from [Goodfellow et al., 2016, p. 285].

Approaching the cliff from above or below does not matter, both are potentially dangerous and should be avoided. Luckily the most serious consequences can be avoided by using a simple trick called *gradient clipping*. The main idea is that the gradients does not specify the optimal step size, but only the optimal direction within an infinitesimal region. When the gradient descent algorithm proposes a large update step, the gradient clipping heuristic intervenes and reduces the step size. The interruption by the gradient clipping is making it less likely that the parameters end up outside the region of approximately steepest descent [Goodfellow et al., 2016, p. 285-286]. Gradient clipping is especially useful for RNNs, because they suffer from exploding and vanishing gradients (described in Section 2.2).

2.8 Evaluation metrics

There exist a wide range of evaluation metrics for different problems. In this thesis, there are two models that we need to evaluate; the classifier (discriminator), and the sequence generating model (generator). Some of the evaluation metrics we use to evaluate these models include accuracy, precision, recall, F1 scores and ROUGE.

Accuracy measures how close the predicted set is to the true set of labels. Precision and recall are standard information retrieval measures[16]. Precision says how many of the retrieved instances are relevant. It is given as: $Precision = \frac{|(Relevant set) \cap (Retrieved set)|}{|(Relevant set)|}$. Recall says how many of the relevant instances are successfully retrieved. It is given as: $Recall = \frac{|(Relevant set) \cap (Retrieved set)|}{|(Retrieved set)|}$. F1 score is commonly used as test accuracy when dealing with binary classification [Sasaki et al., 2007]. It uses both precision and recall to calculate an average harmonic score, formally given as: $F1 = 2\frac{Precision * Recall}{Precision + Recall}$. Accuracy, precision, recall and F1 results presented in this thesis is acquired with the *sklearn metric* API[17].

[16] Precision and recall formulas: https://developers.google.com/machine-learning/crash-course/classification/precision-and-recall

[17] Sklearn metric documentations: http://scikit-learn.org/stable/modules/classes.html#module-sklearn.metrics

Recall-Oriented Understudy for Gisting Evaluation (ROUGE) [Lin, 2004] is a summarization evaluation package that includes measures to automatically determine the quality of a summary by comparing it to other (ideal) summaries created by humans.

The different ROUGE measures count the number of overlapping units such as n-gram, word sequences and word pairs between the generated summary and the ideal summary. The package includes four main measures: ROUGE-N, ROUGE-L, ROUGE-W and ROUGE-S. ROUGE-N and ROUGE-L are the most common ROUGE measures reported in summarization papers.

ROUGE-N is a measure that counts N-gram co-occurrence statistics (I.e., overlap of N-grams between the generated summary and the reference summary). The general formula is seen in Equation 2.14, where n stands for the length of the n-gram, and $\text{Count}_{\text{match}}(\text{gram}_n)$ is the maximum number of n-grams co-occurring in a candidate summary and a set of reference summaries. The most widely used ROUGE-N variants are ROUGE-1 and ROUGE-2.

$$\text{ROUGE-N} = \frac{\sum_{S \in ReferenceSummaries} \sum_{gram_n \in S} Count_{match}(gram_n)}{\sum_{S \in ReferenceSummaries} \sum_{gram_n \in S} Count(gram_n)} \tag{2.14}$$

ROUGE-L is a Longest Common Subsequence (LCS) measure. The formula is given in Equation 2.17, where LCS(X,Y) is the length of the longest common subsequence between the reference summary X of length m and the generated summary Y of length n, and $\beta = \frac{P_{lcs}}{R_{lcs}}$ when $\frac{\partial F_{lcs}}{\partial R_{lcs}} = \frac{\partial F_{lcs}}{\partial P_{lcs}}$.

$$R_{lcs} = \frac{LCS(X,Y)}{m} \tag{2.15}$$

$$P_{lcs} = \frac{LCS(X,Y)}{n} \tag{2.16}$$

$$F_{lcs} = \frac{(1+\beta^2) R_{lcs} P_{lcs}}{R_{lcs} + \beta^2 P_{lcs}} \tag{2.17}$$

The ROUGE package also provides the option to use Porter stemming. Stemming is the process of reducing inflected words to their word stem, base or root form. In this case, it is mainly used to give an equal score to different endings to the same words, e.g. agreed and agree.

Chapter 3

State of the Art

In this chapter, we will be looking into the state of the art regarding sequence generation for NLP, and especially what has been done to improve abstractive text summarization. Both reinforcement learning and GANs have recently shown some promising results and will be the main focus of this chapter.

3.1 Text summarization using seq2seq

Deep neural networks have traditionally done very well on both regression tasks as well as classification tasks, but could not be used in the traditional way to map one sequence to another sequence. To cope with this problem [Sutskever et al., 2014] proposed[1] a model which usually is called *Sequence 2 Sequence* or *seq2seq*. This model uses a multilayered LSTM to map a input sequence to a vector of fixed dimensionality (referred to as the *Encoder*), and then another multilayered LSTM to decode the target sequence from this vector (referred to as the *Decoder*). [Sutskever et al., 2014] achieved state of the art performance on the machine translation task using this model. The standard seq2seq model is shown in Figure 3.1. The model reads an input sentence "ABC" and produces the output sentence "WXYZ". The model stops predicting new words after outputting the <EOS> token.

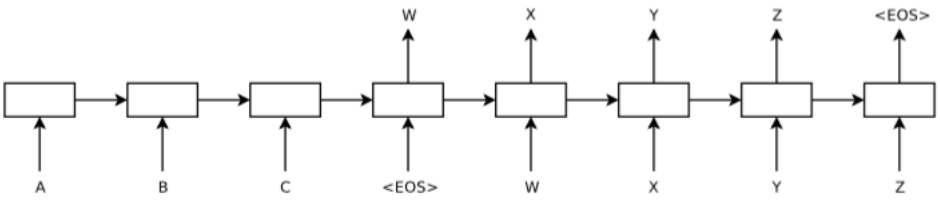


Figure 3.1: The seq2seq model. The figure is retrieved from [Sutskever et al., 2014].

The seq2seq model was further improved by [Bahdanau et al., 2014] using what is known as an *attention mechanism*.

The most important difference of this approach compared to the basic seq2seq model is that it does not attempt to encode the entire input sequence into a single fixed-length vector. Instead,

[1] It seems multiple research groups had the same idea about the same time, [Cho et al., 2014b], among others, also used a similar architecture.

it encodes the input sequence into a sequence of vectors and chooses a subset of these vectors to focus on while decoding the output. The attention mechanism basically allows the decoder to search for a set of positions in the source sequence where the most relevant information is located.

The calculations behind the attention mechanism are quite straight forward. We start by calculating a set of attention weights. This is done with a standard feed forward layer, using the decoder's input as well as the hidden state of the decoder as inputs, followed by a softmax. These weights are then multiplied by the encoder output vectors to create a weighted combination. The attention mechanism is illustrated in Figure 3.2.

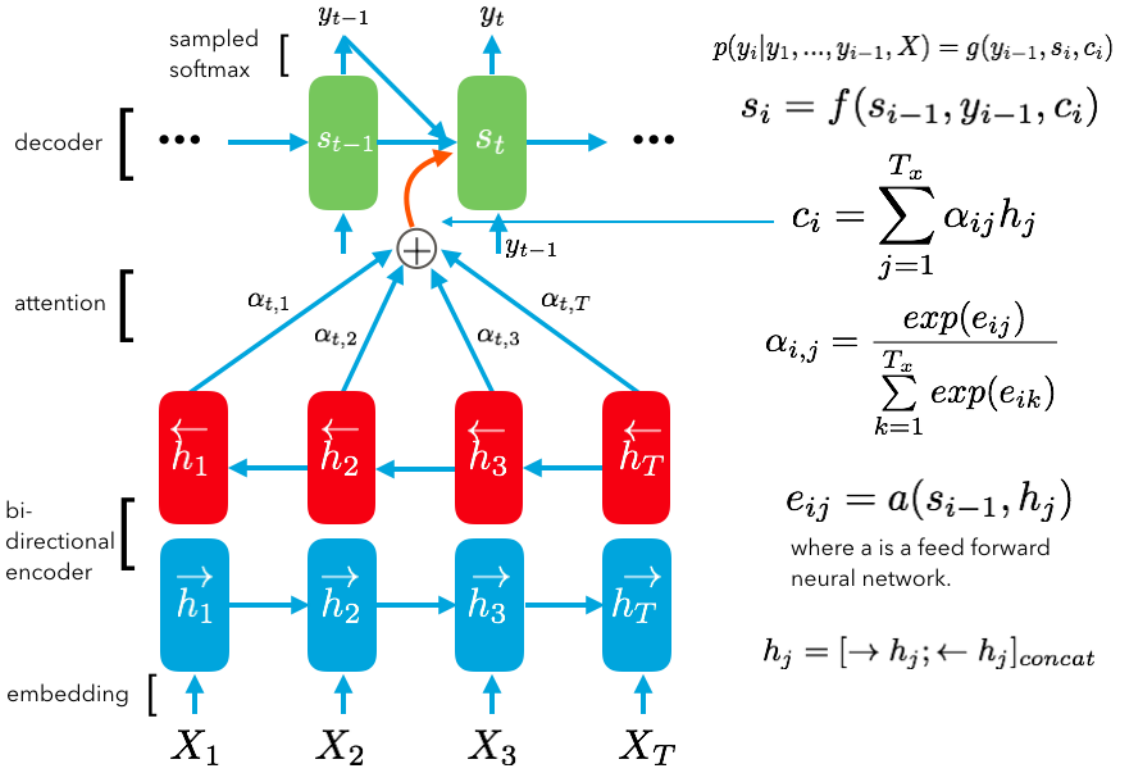


Figure 3.2: The attention mechanism[2]

The seq2seq model was mainly used for machine translation, and the same yields for the attention mechanism. However, [Lopyrev, 2015] used the seq2seq model to generate headlines, achieving very promising results on the GIGAWORD[3] dataset when only keeping articles below 50 words.

[Paulus et al., 2017] showed how this method could be applied to longer sequences by combining standard supervised word prediction and reinforcement learning (RL), producing state of the art results on abstractive summarization. [Paulus et al., 2017] also uses intra-temporal attention, preventing the model to attend to the same parts of the input on different steps during decoding. This is shown by [Nallapati et al., 2016a] to reduce the amount of repetitions when using attention to generate summaries on long documents.

One criticism to the standard seq2seq models are the fact that they are liable to reproduce factual details inaccurately. [See et al., 2017] proposes a hybrid between the standard seq2seq model

[2]Attentional interfaces taken with permission from: https://theneuralperspective.com/2016/11/20/recurrent-neural-network-rnn-part-4-attentional-interfaces/

[3]GIGAWORD: https://catalog.ldc.upenn.edu/ldc2012t21

and a pointer network [Vinyals et al., 2015], which they name *Pointer-Generator Network*, as it allows both copying words via pointing as well as generating words from a fixed vocabulary. This network is shown in Figure 3.3.

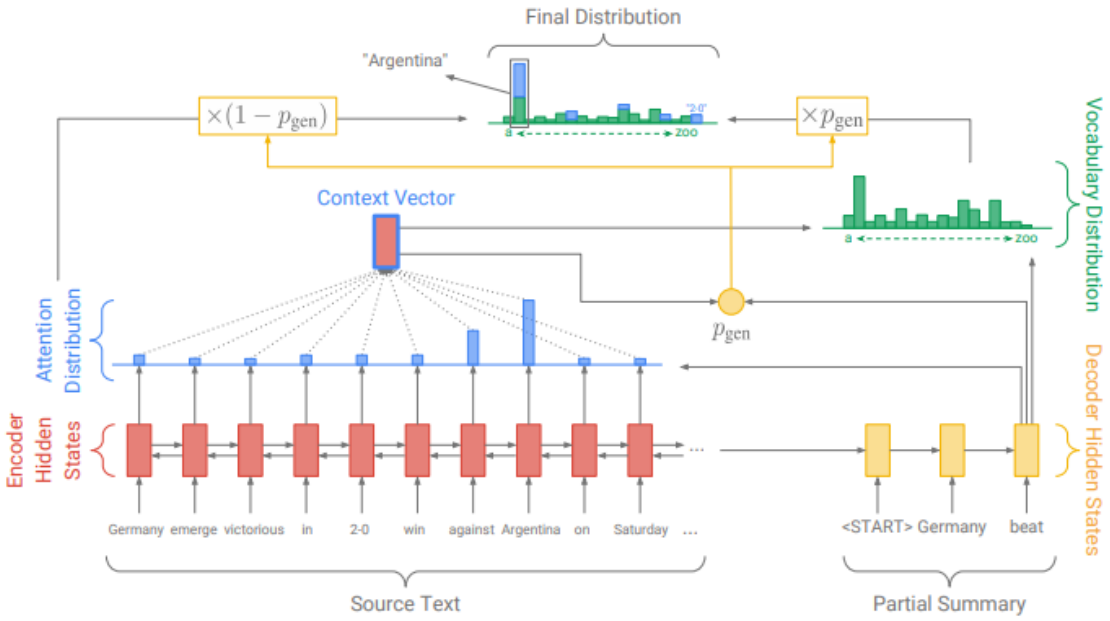


Figure 3.3: The Pointer-Generator Network. The figure is retrieved from [See et al., 2017].

They define the final word distribution from the vocabulary P_{vocab} as shown in Equation 3.1, where s_t and c_t are as defined in Figure 3.2, and V, V', b and b' are learnable parameters.

$$P_{vocab} = softmax(V'(V[s_t, c_t] + b) + b') \tag{3.1}$$

[See et al., 2017] defines p_{gen} as shown in Equation 3.2, where w_c, w_s, w_x and b_{ptr} are learnable parameters and σ is the sigmoid function. p_{gen} functions as a soft switch to choose between generating a word from the vocabulary by sampling from P_{vocab}, or copying a word from the input sequence by sampling from the attention distribution α.

$$p_{gen} = \sigma(w_c^T c_t + w_s^T s_t + w_x^T x_t + b_{ptr}) \tag{3.2}$$

The union between the vocabulary and all the words in the source document is defined by [See et al., 2017] as the extended vocabulary for a particular document. The final probability distribution over the extended vocabulary is shown in Equation 3.3. Sampling from this distribution makes it possible to produce out-of-vocabulary (OOV) words.

$$P(w) = p_{gen} P_{vocab}(w) + (1 - p_{gen}) \sum_{i:w_i=w} \alpha_i^t \tag{3.3}$$

Being able to produce out-of-vocabulary words is a huge advantage, removing the need for a special token to handle unknown words, making the produced summaries more readable. Being

able to copy words from the input text also makes the model less likely to reproduce factual details inaccurately.

Another shortcoming of the standard seq2seq models are the fact that they tend to repeat themselves. [See et al., 2017] proposes a coverage mechanism to deal with this problem. Their coverage mechanism works by maintaining a coverage vector C^t, which is the sum of attention distributions over all previous decoding steps. This vector is used as extra input to the attention mechanism seen in Figure 3.2, making the attention mechanism also condition on its previous decisions. They also introduce a coverage loss to penalize repeatedly attending to the same locations, as seen in Equation 3.4. This additional loss is added to the standard NLL loss function making the final loss function a composition between the NLL loss and the coverage loss.

$$covloss_t = \sum_i min(\alpha_i^t, C_i^t) \tag{3.4}$$

[Lopyrev, 2015], [See et al., 2017] and [Paulus et al., 2017] use what is known as *teacher forcing* to speed up the training process. Instead of generating a word and using that word as input when generating the next word, they use the word from the target text as input during training. This process is known to speed up convergence, but can also lead to a disconnect between training and testing. To overcome this disconnect, we can choose a ratio as to how often we want to use the target word vs. the generated word. This approach generally seems to work well. [Lopyrev, 2015] use a 0.9 ratio (using the target word 90% of the time, and the generated word 10% of the time). [Paulus et al., 2017] use a 0.75 ratio, and [See et al., 2017] use a 1.0 ratio.

3.2 Reinforcement learning for sequence generation

While attentional seq2seq models have achieved good performance on short input and output sequences, they still have some problems when it comes to longer sequences, where they tend to generate repetitive and incoherent phrases. Exposure bias caused by training with pure maximum likelihood estimation (MLE[4]) is also a problem caused by the fact that there exist more plausible summaries than the one single gold truth summary we try to optimize on.

[Paulus et al., 2017] uses reinforcement learning (RL) to try overcoming these issues. Their model is built on a neural intra-attention model with a pointer mechanism. The intra-attention mechanism should help cope with the repetition problem, the same way as the coverage mechanism of [See et al., 2017] aims to reduce repetition. Using reinforcement learning, they hope to reduce exposure bias, and by effect create more diverse summaries.

Their model is first pretrained using the MLE training objective, selecting the best performing model. (Usually, the REINFORCE [Williams, 1992] algorithm starts with a random policy, but in the case of large action-spaces, such as in language modeling (large vocabularies), this becomes intractable, and requires something better than a random policy, making the pretraining step very important for the convergence of the model). The model is then trained using a mixed MLE + RL objective, shown in Equation 3.7, where $y^* = y_1^*, y_2^*, ..., y_{n'}^*$ is the ground-truth output sequence for a given input sequence x, y^s is the sampled output from the decoder, and $\hat{y}$ is the baseline output. In their case the baseline is obtained by maximizing the output probability distribution at each time step (greedy search). $r(y)$ is defined as the reward function for an output sequence y,

[4] In the literature, the term MLE is sometimes used as an alternative to NLL (Equation 2.1). In fact, NLL is equal to L_{mle} (Equation 3.5) averaged over a minibatch. Hereafter, we will use the MLE term.

compared with the ground truth sequence y^*. In their work they optimize using ROUGE-L as the reward evaluation metric.

$$L_{mle} = -\sum_{t=1}^{n'} \log p(y_t^*|y_1^*, ..., y_{t-1}^*, x) \tag{3.5}$$

$$L_{rl} = (r(\hat{y}) - r(y^s)) \sum_{t=1}^{n'} \log p(y_t^s|y_1^s, ..., y_{t-1}^s, x) \tag{3.6}$$

$$L_{mixed} = \beta L_{rl} + (1 - \beta) L_{mle} \tag{3.7}$$

[Paulus et al., 2017] also incorporate a repetition avoidance trick (trigram avoidance) during beam search, where they force the decoder to never output the same trigram more than once. The way they do this is to set the probability of outputting a token to 0 if this token would create a trigram that already exists in the current sequence.

In their work, they show results for pure MLE, pure RL, as well as mixed MLE + RL. Both the pure RL model and the MLE + RL model outperforms the pure MLE model when comparing ROUGE scores. MLE + RL scored the highest on human evaluation tests, but pure RL scored lower than pure MLE, suggesting that the language model provided by MLE is necessary as a balancing factor for the model to continue creating readable sentences.

Another example of reinforcement learning for seq2seq models is [Zhang and Lapata, 2017], who use reinforcement learning for training a sentence simplification model. The main purpose of sentence simplification is to reduce the linguistic complexity of text, while still retaining its original information and meaning.

They propose a model which is based on the encoder-decoder seq2seq architecture. They use the seq2seq architecture as their backbone, and the model is trained in a reinforcement learning framework, allowing it to explore the space of possible simplifications while learning to maximize an expected reward function that encourages outputs which meet simplification specific constraints.

The author addresses a problem with generic encoder-decoder architectures, where copying becomes a very dominating factor, mostly excluding other types of rewrite operations. To encourage a wider variety of rewrite operations while remaining fluent and faithful to the meaning of the source, they employ a reinforcement learning framework.

Similar to [Paulus et al., 2017], they also pretrain their model using the MLE objective. Further, they adopt a curriculum learning strategy, similar to [Ranzato et al., 2015]. In the beginning of RL training, they give little freedom to their RL agent, allowing it to predict the last few words for each target sequence. For every target sequence, they use negative log-likelihood to train on the first L (initially, L = 24) tokens and apply RL to the remaining tokens. Every two epochs, they reduce L by 3 and continue training until L is 0.

Their RL loss function is similar to Equation 3.6, except that their baseline is a linear regression model used to estimate the expected future reward at time t. Their baseline model takes the concatenation of the encoder's hidden state and the decoder's context vector as input and outputs a real value as the expected reward. The parameters of the regressor are trained by minimizing mean squared error.

Their reward function is the weighted sum of three components aimed at capturing key aspects of the output, namely simplicity, relevance and fluency. The reward function is shown in Equation 3.8, where λ^S, λ^R, $\lambda^F \in [0, 1]$ and $r(\hat{Y})$ is a shorthand for $r(X, Y, \hat{Y})$ where X is the source, Y is the target and $\hat{Y}$ is the model output. r^S, r^R and r^F are respectively rewards for simplicity, relevance and fluency. The details of these individual rewards are left out as the implementation is not relevant. However, the details can be found in [Zhang and Lapata, 2017].

$$r(\hat{Y}) = \lambda^S r^s + \lambda^R r^R + \lambda^F r^F \tag{3.8}$$

Overall they find that reinforcement learning offers a great means to inject prior knowledge to the simplification task, achieving good results across the datasets WikiSmall [Zhu et al., 2010], WikiLarge [Zhang and Lapata, 2017] and Newsela [Xu et al., 2015]. They also suggest that the reinforcement learning framework presented is potentially applicable to generation tasks such as sentence compression, generation of programming code or poems.

Both [Paulus et al., 2017] and [Zhang and Lapata, 2017] uses the REINFORCE [Williams, 1992] algorithm to perform the policy gradient updates.

3.3 GANs for sequence generation

GANs that use a discriminative model to guide the training of a generative model has achieved large success in generating real-valued data, especially in computer vision tasks [Denton et al., 2015]. However, it has limited success with generating sequences of discrete tokens, due to the difficulty of passing gradient updates from the discriminator to the generator when the outputs are discrete. Another problem is that the discriminator can only evaluate a complete sequence, since a partial sequence is not always comparable to a full sequence.

[Yu et al., 2016] tries to solve these issues by proposing a sequence generation framework, which they name SeqGAN. They solve the differentiation problem by modeling the generator as a stochastic policy in RL, performing gradient policy updates. The reward is generated from the discriminator which evaluates a complete sequence. They are able to give rewards for each intermediate step by using Monte Carlo (MC) search to create complete sequences from partial sequences.

As in the usual GAN framework, they alternate the training between two models. The θ-parameterized generator G_θ, and the ϕ-parameterized discriminator D_ϕ. The objective of the generator $G_\theta(y_t|Y_{1:t-1})$ is to generate a complete sequence from the start state s_0 to maximize its expected end reward, as shown in Equation 3.9. Here, θ is the policy, R_T is the reward for a complete sequence and $Q^{G_\theta}_{D_\phi}(s_0, y_1)$ is the action-value function, i.e. the expected accumulative reward starting from state s, taking action a, and then following policy G_θ.

$$J(\theta) = \mathbb{E}[R_T|s_0, \theta] = \sum_{y_1 \in Y} G_\theta(y_1|s_0) \cdot Q^{G_\theta}_{D_\phi}(s_0, y_1) \tag{3.9}$$

The gradients are calculated using the REINFORCE algorithm, where the reward is given as the estimated probability of being real assigned by the discriminator D_ϕ, as shown in Equation 3.10. Since the discriminator only can evaluate a complete sequence, MC search is used to sample the unknown last $T-t$ tokens when evaluating an intermediate reward at timestep t. The MC search is done using a roll-out policy G_β, which is simply the same policy as the generator (Essentially, the

remaining tokens are sampled with a categorical distribution using the same decoding procedure as done in the previous timesteps). The roll-out is performed N times and averaged to reduce the variance of the reward.

$$Q_{D_\phi}^{G_\theta}(a = y_T, s = Y_{1:T-1}) = D_\phi(Y_{1:T}) \tag{3.10}$$

The generator used in their experiments is a RNN consisting of an embedding layer, a LSTM, a linear layer and a softmax output layer. The discriminator is a CNN classifier similar to [Kim, 2014]. The generator is pretrained using MLE, and the discriminator is pretrained via minimizing the cross entropy where the positive examples comes from the training set and the negative examples are generated by the pretrained generator. During adversarial training, the discriminator is re-trained using positive examples from the training data and negative examples produced by the updated generator. The objective function of the discriminator is shown in Equation 3.11, where p_{data} is the training data, i.e. the positive examples.

$$\min_\phi -\mathbb{E}_{Y\sim p_{data}}[logD_\phi(Y)] - \mathbb{E}_{Y\sim G_\theta}[log(1 - D_\phi(Y))] \tag{3.11}$$

The generator is trained using the objective function shown in Equation 3.9. The gradient of the objective function is shown in Equation 3.12 (detailed derivation can be seen in [Yu et al., 2016]), where $Y_{1:t-1}$ is the intermediate state sampled from G_θ, and $\mathbb{E}[\cdot]$ is approximated using MC search.

$$\nabla_\theta J(\theta) \simeq \sum_{t=1}^{T} \mathbb{E}_{y_t\sim G_\theta(y_t|Y_{1:t-1})}[\nabla_\theta log G_\theta(y_t|Y_{1:t-1}) \cdot Q_{D_\phi}^{G_\theta}(Y_{1:t-1}, y_t)] \tag{3.12}$$

The generator's parameters (θ) are updated using gradient ascent as shown in Equation 3.13, where $\alpha_h \in \mathbb{R}^+$ denotes the learning rate at step h.

$$\theta \leftarrow \theta + \alpha_h \nabla_\theta J(\theta) \tag{3.13}$$

Using synthetic data experiments they showed that SeqGAN outperformed other baselines such as MLE, Scheduled Sampling [Bengio et al., 2015] and PG-BLEU [Yu et al., 2016]. On the real-world scenarios of creating poems, speech language and music, SeqGAN showed great performance, generating creative sequences. Another key study included in the paper was the convergence of SeqGAN with regards to how much pretraining was done for the generator before adversarial training. Here they showed that the adversarial training process improves the generator slowly and unstably with insufficient pretraining (20 epochs for their synthetic data experiments), while the improvements would be quick and quite large when the generator is properly pretrained (150 epochs for their synthetic data experiments). This phenomenon is also discussed in Section 3.2.

The first, and only (to the best of our knowledge) experiments done using GANs for abstractive text summarization was conducted by [Liu et al., 2017]. They use GAN in the standard fashion way, alternating between the training of a generator and a discriminator. Their setup regarding policy gradients and MC search to get intermediate rewards are similar to [Yu et al., 2016]. The only difference from the equations used in [Yu et al., 2016] is their use of a baseline to reduce the variance of the rewards, similar to what is seen in Equation 3.6, except that their baseline is the running average reward instead of a greedy search. They also take some inspiration from

[Paulus et al., 2017], combining their loss function as a linear combination between policy loss and MLE loss, as seen in Equation 3.7.

Their generative model is a simplified version of [See et al., 2017], i.e. the same model, just without coverage. Their discriminative model is the same as used in [Yu et al., 2016], namely, the CNN classifier ([Kim, 2014]). Pretraining is done until convergence for both the generator and the discriminator before adversarial training. At test time, they also incorporate the same repetition avoidance trick as [Paulus et al., 2017].

Compared to the previous state-of-the-art results, which was [See et al., 2017] and [Paulus et al., 2017], their model showed improvements on ROUGE scores as well as human evaluation. They show that using GAN to train the model could make the generated summaries more abstractive, readable and diverse.

Other usages of GANs that have proved promising results for sequence generation includes [Fedus et al., 2018] (an actor-critic conditional GAN that fills in missing text conditioned on the surrounding context), [Yang et al., 2017] (improving neural machine translation by combining naive GAN with static BLEU RL objective) and [Wang et al., 2017] (combining GAN with VAE to generate realistic text).

Chapter 4

Models and Implementation Details

In this chapter, we start by explaining our base model, which is based on the *Pointer-Generator Network* [See et al., 2017]. We further elaborate on different training strategies and objectives regarding reinforcement learning and generative adversarial networks for abstractive text summarization. Finally, we present an overview of code optimization strategies to speed up the training of such models, which usually is very time-consuming.

4.1 Base model

The base model described in this section is a generative model used to generate abstractive summaries of text. Having a strong base model is important for further research, otherwise it becomes very hard, if not impossible to achieve good results, especially in the context of policy gradient methods, which requires a policy that performs better than a random policy (as described in Section 3.2). This base model is used as our pretrained model in the RL and GAN frameworks, which we will elaborate further in Sections 4.2 and 4.3. Our base model is based on the *Pointer-Generator Network* [See et al., 2017], which is a hybrid architecture between the seq2seq model with attention and a pointer network, as explained in Section 3.1. Essentially, we use a simplified model of [See et al., 2017], where we use trigram avoidance (Section 3.2) instead of coverage (Section 3.1) to speed up the training time[1]. This model is especially attractive because it is not limited by a vocabulary size, because of the capability of copying words from the input text. This way of doing abstractive text summarization seems to be the state of the art [See et al., 2017], thus it is only natural to use it as our base model.

To calculate the loss, we use the MLE objective function (Equation 3.5) averaged over a minibatch. We use AdaGrad (Section 2.7.2.1) as our chosen optimizer. [See et al., 2017] found AdaGrad to work best between Stochastic Gradient Descent, Adadelta, Momentum, Adam and RMSProp.

Instead of using LSTMs we chose to use GRUs, as they generally have the same behavior, but are faster to compute because of the shared gates and states as described in Section 2.2.1.1. Similar to [See et al., 2017] and [Liu et al., 2017], we use a bidirectional encoder, providing even more information to the decoder and the attention mechanism. We also learn the word-embeddings during training instead of using pretrained word-embeddings, similar to [See et al., 2017] and [Liu et al., 2017].

[1] [Liu et al., 2017] report about the same relative increase in ROUGE score ($\sim 3\%$) for both trigram avoidance and coverage.

During training, the top-1 scoring word from the decoder output (Equation 3.3) is appended to the sequence at each time-step. To hopefully generate better sequences, we implement beam-search when decoding the summaries during evaluation. This method has been used, to the best of our knowledge, in all recent research regarding abstractive text summarization. Examples include [Sutskever, 2013], [Paulus et al., 2017] and [See et al., 2017].

Our beam-search implementation is quite standard. We choose two parameters: $k1$ and $k2$. ($k1$) How many beams should be created from the previous beams. ($k2$) How many beams should be kept for each iteration. The beams are extended with the $k1$ top scoring words from the decoder output at every expansion. We then prune these beams, keeping only the $k2$ top scoring beams, where the score is computed as the average sum of the logarithms of the output probability for each word in the sequence thus far (as shown in Equation 4.1). This gives us an average score per beam. We finally output the top scoring beam when all beams have either reached an <EOS>[2] token or maximum length. In our experiments we choose $k1 = 3$ and $k2 = 6$. We choose these parameters based on computational efficiency, while providing a sufficient search space for the generation (further elaboration is included in Section 5.4.3). [See et al., 2017] use $k1 = 2$ and $k2 = 4$. [Liu et al., 2017] use $k1 = 2$ and $k2 = 5$. During beam-search, we incorporate the trigram avoidance trick, as described in Section 3.2 (this is not done during training).

$$\frac{1}{N}\sum_{word_i}^{N} log(P(word_i)) \tag{4.1}$$

The qualitative results shown in Chapter 5 are presented with sequences generated by beam-search. In the following sections, we will use the term *generator* to denote the encoder and decoder as a single unit.

4.2 Reinforcement learning

There are, to the best of our knowledge, two cases ([Paulus et al., 2017] [Liu et al., 2017]) of research that reports improvements on the *Pointer-Generator Network* for abstractive summarization, achieving state of the art results. As described in Section 3.2, [Paulus et al., 2017] use reinforcement learning to optimize on the ROUGE-L metric. [Liu et al., 2017] on the other hand, use a GAN framework that tries to optimize towards creating more human-like summarizes. One of the similarities between these two approaches is that they both use policy gradient methods to optimize the generator, which makes for a quite general framework, mostly reliant on the reward signal.

The REINFORCE algorithm (Equation 2.10), which is employed in both [Paulus et al., 2017] and [Liu et al., 2017], requires the calculation of a reward. In Section 3.3, we mention that the discriminator requires a complete sequence as input, as the reward will not be accurate in the case of a partial sequence. This is also true for the reward acquired from the ROUGE metrics. To be able to calculate an accurate reward, we need a strategy to generate a complete sequence from a partial sequence, which in turn will be used in the REINFORCE algorithm. There are generally two ways to go about this. The first option is to generate the entire sequence, accumulating the log probability of producing each token, then finally giving one single reward to the entire sequence, which will be multiplied to each log probability when doing gradient ascent. This strategy is used

[2] <EOS> is a token that represents the End of Sequence.

by [Paulus et al., 2017], and is described in Code 4.1, where $generator$ includes an encoder and a decoder, <SOS> is a token for "Start Of Sequence" and $gradient\_ascent(*)$ with the given loss as input essentially yields the REINFORCE algorithm. Figure 4.1 illustrates this strategy, which we name the *naive roll-out strategy*. The code snippet use a simple Python/pseudocode syntax.

```
def naive_rl_training(generator, discriminator, input, timesteps):
    hidden_state = generator.encode(input)
    log_probabilities = []
    sequence = []
    next_token = "<SOS>"
    for i in timesteps:
        next_token, probability, hidden_state
                = generator.decode(next_token, hidden_state)
        sequence.append(next_token)
        log_probabilties.append(log(probability))
    reward = discriminator.evaluate(sequence)
    # multiply every single log probability by the same reward
    loss = log_probabilities * reward
    # average the loss over time-steps
    avg_loss = avg(loss)
    generator.gradient_ascent(avg_loss)
```

Code 4.1: Naive roll-out strategy

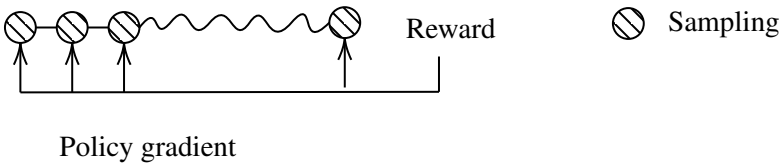


Figure 4.1: Illustration of the naive roll-out strategy. The entire sequence is accumulated, calculating the reward when the last token is sampled. Finally, the reward is passed back to every intermediate time-step.

The other strategy is the one used by [Yu et al., 2016]. Instead of evaluating the entire sequence, giving the same reward to each time-step, this strategy employs a N-time Monte-Carlo roll-out for each intermediate time-step, providing fine-grained rewards for each token (action) sampled by the generator. The strategy is described in Code 4.2, where N denotes the number of roll-outs. Figure 4.2 illustrates this strategy, which we call the *Monte-Carlo roll-out strategy*. Here, $N = 3$ and "state" is the previously accumulated sequence ($sequence$ in the code snippet).

```
def monte_carlo_rl_training(generator, discriminator, input, timesteps, N):
    hidden_state = generator.encode(input)
    total_loss = 0
    sequence = []
    next_token = "<SOS>"
    for i in timesteps:
        next_token, probability, hidden_state
                = generator.decode(next_token, hidden_state)
        sequence.append(next_token)
        # perform an N-time monte carlo roll-out
        rolled_out_sequence = monte_carlo_rollout(sequence, next_token, N)
        # evaluate rolled_out_sequence and average
        reward = discriminator.evaluate(rolled_out_sequence) / N
        # add scaled log probability to total_loss
        total_loss += log(probability) * reward
    # average the loss over time-steps
    avg_loss = total_loss / timesteps
    generator.gradient_ascent(avg_loss)
```

Code 4.2: Monte-Carlo roll-out strategy

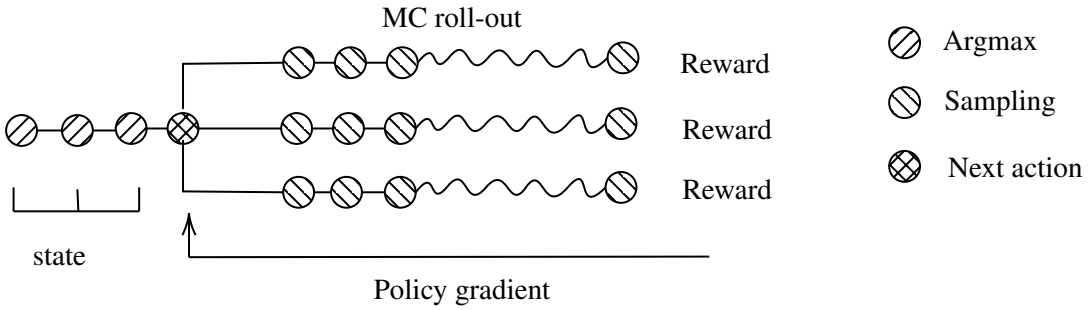


Figure 4.2: Illustration of the Monte-Carlo roll-out strategy. At every time-step, a MC roll-out is employed and a reward for that time-step (Next action) is calculated and averaged over N roll-outs.

In the naive roll-out strategy (Code 4.1), the tokens at each timestep needs to be sampled by some distribution. In our case, the distribution is the probabilities provided by the decoder's output (Equation 3.3), I.e. the final decoding layer, where the probability of sampling token X is $P(X)$. This type of action sampling is also employed in the Monte-Carlo roll-out strategy (Code 4.2). However, in this case, we can choose between accumulating the sampled token as the "previous generated sequence" ($sequence$ in the code snippets), or accumulating the top-1 (argmax) scoring tokens from the decoder output. In Figure 4.2, argmax is illustrated, but it could just as well have been replaced with "Sampling".

Another choice that is quite important for the stability during training is the choice of baseline used to reduce the variance of the reward signal (Equation 3.6). Generally, this baseline can be any number. It can be a varying number, or it can be a constant. We report results using two different baselines. The first one is a running average, which is basically just an average of all the rewards that has been seen so far. The second one is a greedy search, similar to [Paulus et al., 2017], where they maximize the output probability at each time step (argmax from the decoder output).

In our work we explore the objectives of ROUGE-1 and ROUGE-2, as there are, to the best of our knowledge, no reported results on this yet ([Paulus et al., 2017] mention that they tried ROUGE-2, but found that it created summaries that almost always reached maximum length and often ended sentences abruptly). As mentioned in Section 3.3, pretraining is important to be able to utilize the

reward-signal when training with REINFORCE. Similar to [Paulus et al., 2017], we pretrain the generator (Section 4.1) using MLE before training with REINFORCE.

4.3 GAN

In the following section, we will present an abstractive text generator model with elements from the sections above. The model is based on a GAN framework (Section 2.5), where we use the discriminator to guide the generation. As mentioned in Section 3.3, this is a challenging process because it is difficult to pass gradient updates from the discriminator to the generator when the outputs are discrete. We base our GAN setup on the work of [Yu et al., 2016], solving the differentiation problem by using policy gradients. We also take some inspiration from [Liu et al., 2017], which is the first, to the best of our knowledge, to use GAN for abstractive text summarization.

The generator in our model can be viewed as an agent that generates sequences of text. Each word it generates is consider as an action. The goal for the generator is to maximize the expected reward, and as mentioned in Section 4.2, we will use the REINFORCE algorithm to achieve this goal. We will use the discriminator (explained in more details below) to get an estimate of the reward for the generated sequence.

The generator is a pretrained version of the base model explained in Section 4.1. In a normal GAN framework, it is not common to pretrain the models, but in [Yu et al., 2016] they show the importance of pretraining. Without a good start policy, the REINFORCE training becomes quite challenging (essentially a random search in a high dimensional space).

The objective of the discriminator is to discriminate sequences as real or fake, where the sequences generated by our models are considered fake, and the ground truth sequences in the dataset is considered real. In our GAN framework we use the CNN classifier from [Kim, 2014], as described in Section 2.3.1. This classifier has proven to yield good results for both [Yu et al., 2016] and [Liu et al., 2017].

Since we pretrain our generator we have to pretrain the classifier as well, even though it is not normal to do this in a normal GAN framework. The problem if we do not pretrain the classifier is that the training becomes very unstable and the generator will never improve until the discriminator "catches" up to the generator. To pretrain the classifier we use the Adam optimizer (Section 2.7.2.2). The training data is a combination of real data from the dataset and fake data generated by the pretrained generator.

In a GAN framework one alternates between training the generator and the discriminator. If the performance between the generator and the discriminator is unbalanced, one might not get the desired training development. For instance, the discriminator could be so good that the generator never learns anything. To allow learning for both the discriminator and the generator, there needs to be a balance between the training of the two. We can attain this balance by training each part for a given number[3] of iterations before we switch the training to the other part. This can be seen in Code 4.3, where *n_generator* is the number of iterations we train the generator, and *n_discriminator* is the number of iterations we train the discriminator. We continue this alternation of training for n epochs[4]. For each iteration we train the generator, we use one of the strategies

[3]The number of iterations does not necessarily need to be preset. It may vary on performance, or some other measure.

[4]Optimally, we would continue this training until reaching a nash equilibrium, but this is not easily obtainable.

mentioned in Section 4.2 to get the reward for the generated sequences and apply the REINFORCE algorithm (Equation 2.10). For each discriminator iteration, we generate negative (fake) examples with the current policy (G_θ), and combine these with positive (real) examples from the training data. We train on this new combination of training data for *n_epochs_discriminator* to allow more discriminator training without the cost of generating new samples.

```
pretrain G_θ
generate negative examples with current policy G_θ
pretrain D_ϕ
for epoch in n:
    for i in n_generator:
        generate sequences and get rewards with chosen strategy
        update G_θ by REINFORCE
    for j in n_discriminator:
        use G_θ to generate negative examples
        training_data = negative_examples + positive_examples
        for k in range(n_epochs_discriminator):
                train D_ϕ using the new training data
```

Code 4.3: GAN setup

4.4 Combined objectives

In the preceding sections, we talk about guiding the generation towards some objective. [Zhang and Lapata, 2017] show the possibility to combine multiple objectives to achieve even greater results. This inspiration makes us believe that this also can be done for abstractive text summarization. To the best of our knowledge, combining multiple reward objectives when training models for abstractive text summarization has not been explored yet. We therefore want to explore combinations of objectives and how they impact the generation.

One of the objectives we have mentioned already is the real or fake objective (GAN discriminator). It makes a lot of sense to try imitating real abstract text summaries. By doing this, the generator can hypothetically learn to fool a human to believe that the summaries are human made (thus in a way pass the Turing test). We will use the term ”discriminator objective” or ”discriminator reward” when talking about optimizing towards the objective of fooling the discriminator.

In Section 3.2, we talked about how [Paulus et al., 2017] used the ROUGE-L objective to achieve better results. One common problem with abstractive text summarization is repetition and the ROUGE objectives can potentially help to cope with this problem. Lastly, in [Paulus et al., 2017] and [Liu et al., 2017] they have combined their policy gradient loss with normal MLE. They do this to stabilize their training, which makes sense since MLE is a language model, and can be viewed as a balancing factor when training with REINFORCE.

To illustrate how each one of these objectives (discriminator, ROUGE and MLE) impact the generation, we will first study them independently. Thereafter combine them to see if, and how they work together. Combining reward objectives (policy gradient loss) with MLE is done by employing Equation 3.7 in our generator update rule. To combine the reward objectives, we will use Equation 4.2, where we use a scaling factor, λ, to scale the magnitude between the objectives while keeping the reward in the range between zero to one.

$$combined\_reward = \lambda * ROUGE\_reward + (1 - \lambda) * discriminator\_reward \tag{4.2}$$

4.5 Code optimization

Training of the seq2seq models previously mentioned is generally very time consuming. For instance, [See et al., 2017] reports 4 days and 14 hours training time for their base model (attention based seq2seq) when it is trained for 33 epochs with a vocabulary size of 50k. The training time increased to 8 days and 21 hours when the vocabulary size was set to 150k. However, their pointer-generator model with a 50k vocabulary size, even though it has a lot more parameters, only required about 12.8 epochs of training, which took 3 days and 4 hours to train, with an additional 2 hours to fine-tune their model by training with coverage-loss (Equation 3.4) for 3000 iterations.

Some of the reasons why training of such models become slow boils down to the following elements:

- Large networks[5][6]
- Data transfer between CPU and GPU
- A lot of special case handling[7]

One way to keep the network from becoming too large is to keep the vocabulary at a lower size (which is possible because of the pointer-generators ability to handle OOV words).

There is no way around the special case handling, but they can however be optimized quite well. For instance, the naive implementation of a code snippet used to replace OOV words with the <UNK>[8] (mask) token is shown in Code 4.4. An optimized version is shown in Code 4.5.

```
def mask_input(input):
    # copy the data from GPU to CPU
    cpu_copy = input.cpu()
    for i in range(0, len(cpu_copy)):
        if cpu_copy[i] >= vocabulary_size:
            cpu_copy[i] = mask_token
    # copy the modified data from CPU back to GPU
    cuda_copy = cpu_copy.cuda()
    return cuda_copy
```

Code 4.4: Naive masking

[5] Adding an additional layer to a network increases the time used to calculate the forward pass as well as the backward pass for backpropagation.

[6] [See et al., 2017] reports 21,499,600 parameters in their base model when the vocabulary size is 50k.

[7] When it comes to the Pointer-Generator Network; when the network outputs an out-of-vocabulary (OOV) token by pointing, the token needs to be replaced by another placeholder token before being used as input to the next decoder step (since the embedding layer can only handle in-vocabulary tokens).

[8] <UNK> is a token that represents an OOV word.

```
MASK = ([mask_token] * batch_size).cuda()
UPPER_BOUND = ([vocabulary_size] * batch_size).cuda()
def mask_input(input):
    # create a matrix of 1 and 0,
    # where cond[n] = 1 if input[n] < vocabulary_size
    cond = input < UPPER_BOUND
    # combine the input with the mask by vectorization
    masked_input = (cond * input) + ((1 - cond) * MASK)
    return masked_input
```

Code 4.5: Vectorized masking

The optimized code with vectorization not only decreases the time taken to go through the whole batch of elements, but also removes the need to transfer the elements to the CPU and back to the GPU. These kind of vectorization solutions can help speed up the training time drastically.

Another time-consuming process when it comes to reinforcement learning is Monte-Carlo roll-out. In the reinforcement learning strategy where N-time Monte-Carlo roll-out is employed (Code 4.2), the total training time of the model generally increases linearly with N. However, since this operation is only done to calculate a reward R for the rolled-out sequence and averaged over the N roll-outs, the operation becomes quite easily parallelizable. Pseudocode for the naive implementation is shown in Code 4.6. [Clemente et al., 2017] shows one example of how this can be parallelized, distributing the workload across multiple workers, as shown in Figure 4.3. Another way to parallelize this is to increase the width of the batch, multiplying the batch input by N. Pseudo code for this is shown in Code 4.7.

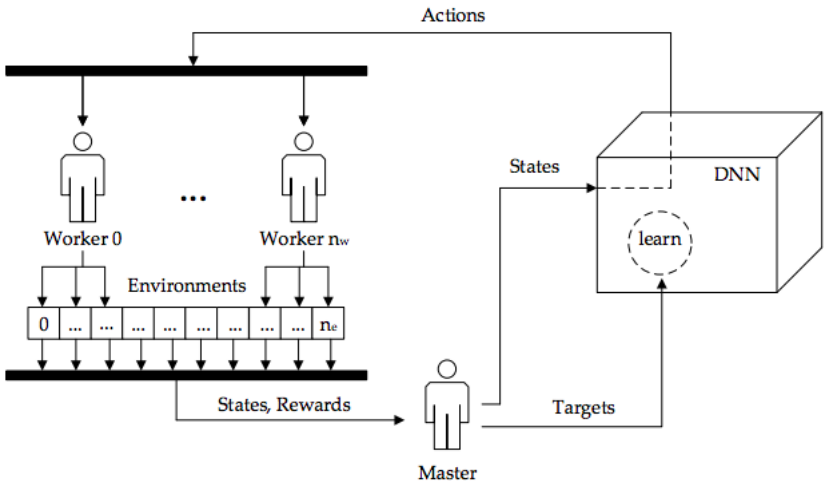


Figure 4.3: Parallel framework for deep RL. Retrieved from [Clemente et al., 2017].

```
def naive_rollout_reward(batch_size, current_sequence, action, N):
    total_reward = [0] * batch_size
    for i in N:
        # sample remaining tokens for one batch
        sample = monte_carlo_rollout(current_sequence, action)
        # get a batch of rewards
        total_reward += get_reward(sample)
    # average the reward
    avg_reward = total_reward / N
    return avg_reward
```

Code 4.6: Naive roll-out

```
def batch_parallel_rollout_reward(batch_size, current_sequence, action, N):
    # duplicate action batch N times
    action = [action] * N
    # duplicate current_sequence N times
    current_sequence = [current_sequence] * N
    # sample remaining tokens on the extended batch
    sample = monte_carlo_rollout(current_sequence, action)
    # get a extended batch of rewards
    reward = get_reward(sample)
    # split the reward in N
    rewards = reward.split(N)
    # sum up rewards for all N original batches
    total_reward = [0] * batch_size
    for i in rewards:
        total_reward += rewards[i]
    # average the reward
    avg_reward = total_reward / N
    return avg_reward
```

Code 4.7: Batch-parallel roll-out

A few other options to reduce the training time when doing reinforcement learning with Monte-Carlo roll-out includes:

1. Stop sampling at a certain length. Given that the average length of a generated sample is a lot lower than the maximum, there will be very minor differences[9] to the training if we stop sampling somewhere between the average and the maximum. Since most of the training time comes from sampling, reducing the number of times we have to sample by X% would also decrease the total training time by close to X%.

2. Reduce the sample rate. Doing a Monte-Carlo roll-out for every time-step would yield a 1.0 sample rate, but the network does not necessarily need to train on every time-step for every example. We could lower this rate, e.g. only sampling every 5th time-step. This would increase the diversity of the learner for a specific amount of time[10], possibly reducing the time needed for convergence.

3. Stop sampling when the entire batch has reached <EOS>. This one is quite straight forward. If the network stops learning anything new after outputting <EOS>, then there is no reason to keep sampling at that point. However, this check can be quite expensive[11].

[9] If the network has already output an <EOS> token, then the remaining tokens will always be sampled as <PAD> (a token used for zero padding) and would not provide any useful information to the network.

[10] Or in other words; increase the difference in explored states in a specific amount of time.

[11] At some point during this check, there will be a need to transfer data from GPU to CPU.

Chapter 5

Experiments and Results

In this chapter, we will present our experiments and results. We start by explaining the details behind the dataset that we use, and what kind of preprocessing we do with this dataset. Following is a brief overview of the experimental setup. Afterwards, we go through the hyperparameters used in our experiments, followed by results obtained when pretraining the generator and the discriminator. Next, we show extensive results when exploring parameters and strategies used in the reinforcement learning setup and the adversarial training setup, as explained in Sections 4.2 and 4.3. We also show quantitative results and qualitative results for different objective functions and strategies. Finally, we show runtime differences between the algorithms and ideas presented in Section 4.5, as well as differences in runtime between the two roll-out strategies presented in Section 4.2.

5.1 Dataset

The dataset we use in our experiments is the CNN/Daily Mail dataset first used for abstractive text summarization by [Nallapati et al., 2016a][1]. This dataset has gained popularity because of its unique characteristics; long documents and ordered multi-sentence summaries. The dataset has been used by e.g. [See et al., 2017], [Paulus et al., 2017] and [Liu et al., 2017].

When preprocessing the dataset, we start with the tokenized version[2], which contains a total of 312 085 documents (article-summary pairs). All words, numbers, special characters and combinations of these are regarded as single tokens. Some of the articles and summaries are missing periods, which we fix by inserting a full stop punctuation at the end of the mentioned sentences. We generate multi-sentence summaries by combining all the human-created summaries marked with *@highlight* for each article. Finally, all the letters are lower-cased.

The average number of tokens of the articles is 790, varying from just a few tokens to over 2000 tokens. We remove all the article-summary pairs where either the article contains less than 100 tokens, or the summary contains less than 20 tokens. This gives us a total of 307 525 article-summary pairs.

For computational efficiency, we truncate articles at 400 tokens. This does not seem to be a

[1] Originally, the dataset was used for the task of passage-based question answering [Hermann et al., 2015]. The original dataset can be downloaded at: `https://cs.nyu.edu/~kcho/DMQA/`

[2] Tokenized version of the CNN/Daily Mail dataset: `https://github.com/JafferWilson/Process-Data-of-CNN-DailyMail`

problem, as most of the important information is located in the first few sentences [Chen et al.,]. Figure 5.1 shows the number of summaries at different token length bins (intervals). As we can see in this figure, most of the summaries are in the range between 30 and 70 tokens. To further speed up training time, and to keep mini-batches at more similar lengths, we truncate summaries at 100 tokens (both [See et al., 2017] and [Liu et al., 2017] truncate articles and summaries at 400 tokens and 100 tokens, respectively). At test time, when doing beam-search to generate summaries, we increase the upper limit to 120 tokens, similar to [See et al., 2017].

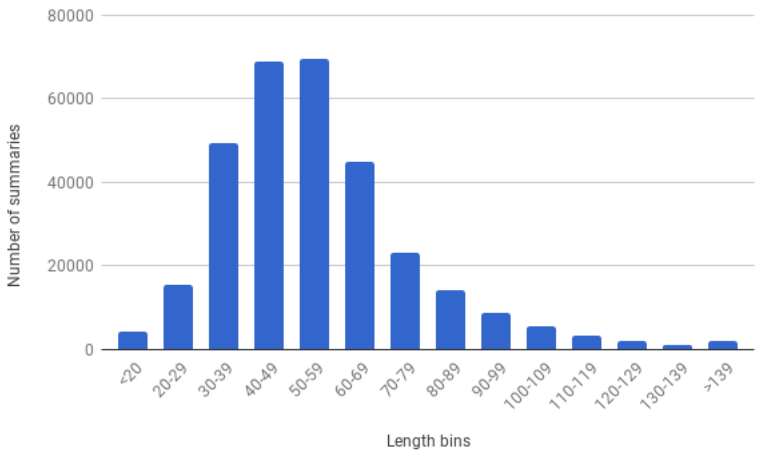


Figure 5.1: Summaries per token length bin (intervals)

The vocabulary size (number of distinct tokens) of the entire set of preprocessed article-summary pairs was 532 449. This includes regular words, numbers, special-characters, miss-spelling, and more. The number of parameters in the network increases exponentially with the vocabulary size, making a huge impact on training time, as mentioned in Section 4.5. To cope with a huge vocabulary size, we use a common approach of replacing rare words with the special token <UNK>. We restrict the size of the vocabulary to 50 000, similar to [See et al., 2017]. We also keep a separate set of the ground truth summaries that is not restricted by the vocabulary to be able to use the pointing mechanism explained in Section 3.1.

To make our model able to conclude when to not generate more words, we append an end of sentence token (<EOS>) to the articles and the summaries. To make it possible to train in batches we also append padding tokens (<PAD>)[3] to the articles as well as the summaries to make all the articles have the same length and all the summaries have the same length. This would not seem to be a problem for the loss function, as it would generally not add extra loss, but rather just decrease the average loss across the sequence, which can be compensated for by using a higher learning rate or training for more iterations.

When running our experiments, we split the dataset in a training set, validation set and a test set, where we have 294 525 training pairs, 2000 validation pairs and 11 000 ($\sim 3.6\%$) test pairs. The size of the test set is similar to [See et al., 2017] ($\sim 4.2\%$). The size of the validation set is quite a bit lower than [See et al., 2017] to be able to quickly validate models, as beam-search is quite time-consuming. The validation set is used to evaluate models during training for early-stopping, as well as to evaluate different hyperparameters.

Figure 5.2 show an example of a preprocessed article-summary pair, where out-of-vocabulary

[3]This is also known as ”zero padding”.

tokens are kept, and <PAD> and <EOS> tokens are left out.

Article
serie a club parma have been deducted four points for breaching financial regulations . the sanction adds to the three they were penalised earlier in the season , the italian football federation -lrb- figc -rrb- said on thursday . parma 's players celebrate beating juventus at the weekend but the club are rooted to the bottom of serie a . the decision means that parma , who despite being last in serie a beat leaders juventus 1-0 on saturday , have been left with 12 points from 30 games . the club were declared bankrupt last month and have not paid their players all season .

Summary
parma declared bankrupt last month and have n't paid players all season . they were deducted three points earlier in the serie a season . the four-point sanction leaves them with 12 points from 30 games .

Figure 5.2: Example of a preprocessed article-summary pair

5.2 Infrastructure

We run our experiments on a single Tesla P100-PCIE-16GB that is provided by Telenor AI Lab NTNU or from the EPIC cluster at NTNU. All the different models presented in the thesis are written in PyTorch[4]. PyTorch is a deep learning framework for Python. We also used Tensorboard[5] to log all our results. Tensorboard is a visualization tool provided by Tensorflow.

5.3 Proceedings

We will start by presenting results from the pretrained generator and the pretrained discriminator, followed by results obtained with adversarial training and the different RL objectives. We also show intermediate results when varying different parameters relevant to the RL training setup. Finally, we show differences in training time when optimizing the code.

Results on the generative model are mainly presented as ROUGE scores and qualitative examples. In some cases, we use training loss (MLE), policy log sum, reward and the percentage of novel n-grams to show differences between parameters and objectives. *Policy log sum* is calculated as the sum of the log probability of outputting every word in the generated sequence (S) (Equation 5.1)[6]. *Reward* is used as a shorthand for the calculation; $reward - baseline$, averaged over the number of time-steps. Both the reward (given by the objective function) and the baseline is given in the context.

[4]PyTorch website: http://pytorch.org/

[5]TensorBoard website: https://www.tensorflow.org/get_started/summaries_and_tensorboard

[6]Policy log sum is essentially a measure of the agent's current belief.

$$\text{policy log sum} = \sum_{w_t \in S} log(P(w_t)) \tag{5.1}$$

For the discriminator we will present training loss (BCE), accuracy, precision, recall and F1 score.

Measures done during training and for parameter search are calculated on the validation dataset. The final results for the different models are calculated on the test dataset.

ROUGE scores are calculated using Porter stemming and full-length F1 scores are presented. The scores are averaged over the whole corpus of examples. ROUGE scores are calculated after performing beam-search.

The percentage of novel n-grams are calculated as the number of overlapping n-grams divided by the number of total n-grams, where an overlapping n-gram means that the n-gram in the summary (S) also exists in the article (A), as shown in Equation 5.2.

$$\text{novel n-grams} = \frac{count(S \cap A)}{count(A)} \tag{5.2}$$

To be able to run all the experiments and to validate results efficiently, we need to set restrictions on how long we want to continue training and how often we want to do validation. We train with the naive roll-out strategy (Code 4.1) for a total of 10 epochs, where we validate results at every 0.5 epoch. We train with the Monte-Carlo roll-out strategy (Code 4.2) for a total of 5 epochs[7], where we validate results at every 0.25 epoch[8]. For some experiments, the number of epochs and validation-intervals may vary, as we do not always require as many data-points.

All graphs and charts presented in this chapter are created using Google Sheets[9].

5.4 Hyperparameters

In this section, we will present hyperparameters and code optimization techniques that are relevant for the generative model during pretraining and after pretraining, as well as hyperparameters for the discriminative model. Finally, we will present results from different beam search parameters.

5.4.1 Generator parameters

For our generative model, we have the following hyperparameters: *hidden size* (The dimensionality of the GRU), *embedding size*, *batch size* and *learning rate*.

We use an embedding size of 100, similar to [Liu et al., 2017]. [See et al., 2017] use an embedding size of 128, but we did not see much difference in convergence. Similar to [See et al., 2017] and [Liu et al., 2017], we use a batch size of 16 and do not use dropout for the generator.

The hidden size did matter quite a bit. With a very large hidden size (e.g. larger than 512), the model starts to fit the data all too well, and the results on the validation set are a lot worse compared to results with a smaller hidden size. Generally, everything between 64 and 512 seemed

[7] We train using the Monte-Carlo roll-out strategy (Code 4.2) for only half the number of epochs compared to the naive roll-out strategy (Code 4.1), since the time requirement is a lot higher.

[8] We double the validation-intervals to get an equal amount of data points.

[9] Google Sheets: `https://www.google.com/sheets/about/`

to work pretty well with the size of our training set, where 128 and 256 seemed to give the best results. Both [Liu et al., 2017] and [See et al., 2017] report a 128 hidden size for their bidirectional encoder and 256 hidden size for their decoder, which we also use.

During MLE training, *teacher forcing ratio* is also a relevant parameter, which we keep as 1.0 for faster convergence, similar to [See et al., 2017].

We use AdaGrad (Section 2.7.2.1) as our chosen optimizer with 10^{-5} L2 penalty. [See et al., 2017] report using AdaGrad with a learning rate of 0.15 and an initial accumulator value of 0.1. The PyTorch version we use (0.3.1) seems to have a different implementation of AdaGrad and do not have an accumulator value parameter[10]. We did some experiments and found a learning rate of 0.015 to yield good results. In Section 2.7.3 we talk about how gradient clipping can be beneficial for a recurrent neural network. We also see that [Nallapati et al., 2016b] and [See et al., 2017] make use of this optimization technique with good results. We apply this to our network and use gradient clipping with a maximum gradient norm of 2, as done in [Nallapati et al., 2016b] and [See et al., 2017].

After pretraining we have to do some changes to our generative model to stabilize the training. Similar to [Liu et al., 2017] and [Paulus et al., 2017], we lower the learning rate as well as increasing the batch size to 50. We found 0.0001 learning rate to work best, as can be seen in Figure 5.3. The figure shows the policy log sum averaged for every 200 iterations for different learning rates.

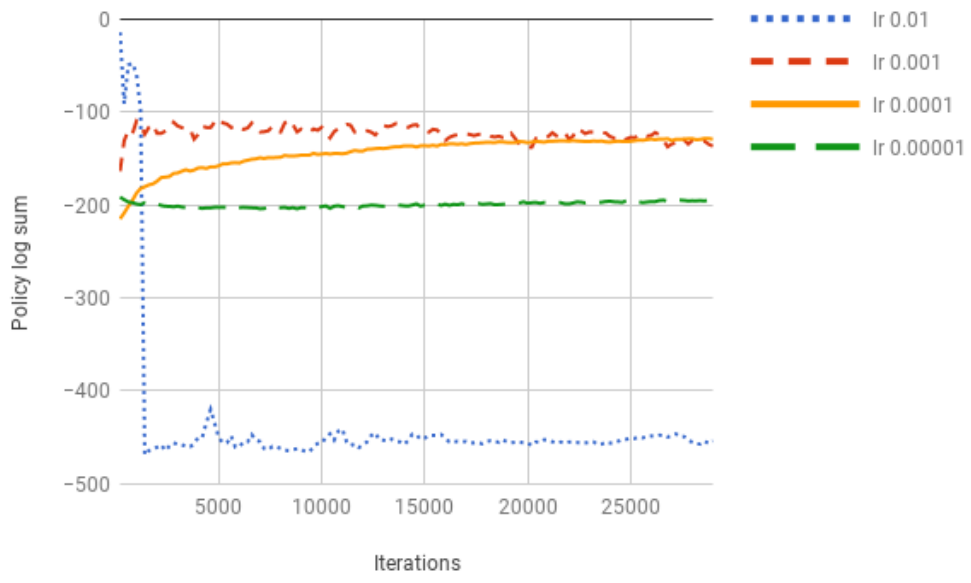


Figure 5.3: Policy log sum for different learning rates (lr)

In Section 5.10, we conduct experiments on code optimization techniques and parameters, as presented in Section 4.5. For the experimental results in Section 5.7, 5.8 and 5.9, we employ the vectorized masking operation (Code 4.5), the batch-parallel roll-out algorithm (Code 4.7), and a sample rate of 0.2. We also set the maximum number of decoding steps to the maximum length of a single summary in the current reference batch, and start checking for EOS at time-step = 80.

[10] PyTorch AdaGrad documentation: `https://pytorch.org/docs/0.3.1/optim.html`

The code optimization techniques and parameters are further discussed in Section 6.8.

5.4.2 Discriminator parameters

The classifier used as a discriminator in our thesis is based on [Kim, 2014]. As mentioned in Section 3.3, this model has shown good performance on text classification as well as in a GAN setting. Most of the parameters we use are taken directly from [Kim, 2014]. This includes using filter windows of 3, 4 and 5 with 100 feature maps each, embedding size of 100 (trained from scratch), and dropout set to 0.5. Adam is our chosen optimizer and we use it with 0.0001 learning rate, a 10^{-5} L2 penalty and a batch size of 64. In [Liu et al., 2017] and [Yu et al., 2016] they use a learning rate of 0.001, but in our case, this was not optimal. This is due to the fact that the fake data created by our generator has so large variance, that with a large learning rate our classifier was not capable to learn the features of the fake data. This can be seen in Figure 5.4, where we show the training loss of the classifier with different learning rates. The figure shows a steady decrease in loss for learning rate = 0.0001 that reaches a lower point compared to the other learning rates. We can also note that learning rate = 0.001 seems to decrease drastically in the beginning, but after about 10 000 iterations, it seems to stagnate and become quite unstable. Moreover, we do not alter the discriminator parameters when doing adversarial training.

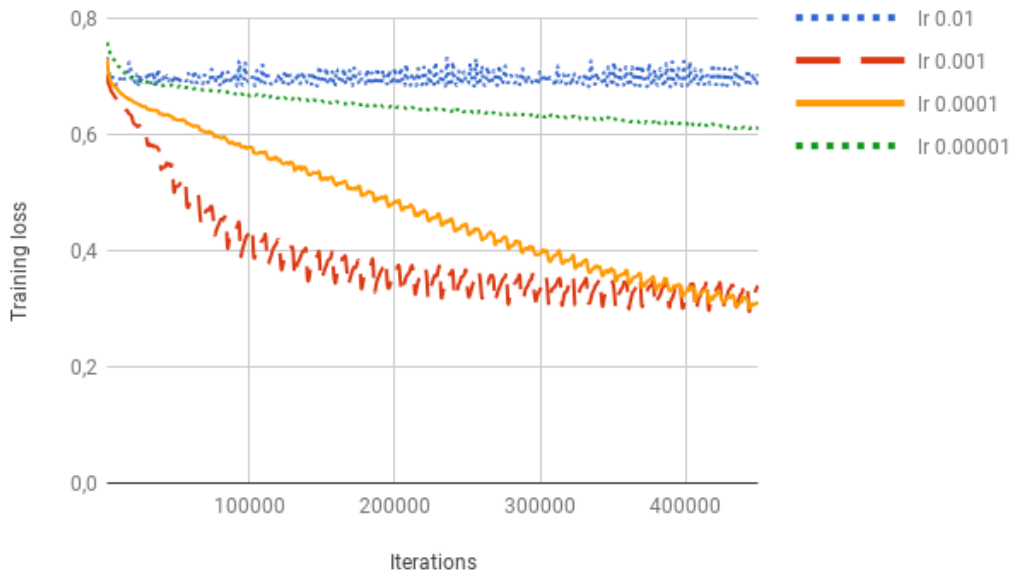


Figure 5.4: Training loss for the discriminator using different learning rates (lr)

5.4.3 Beam search parameters

To be able to choose the $k1$ and $k2$ parameters for the beam search approach explained in Section 4.1, we conduct tests using the validation dataset where we vary the parameters $k1$ and $k2$ for a given trained model. Figure 5.5 shows the ROUGE-1 score, Figure 5.6 shows the ROUGE-2 score, Figure 5.7 shows the ROUGE-L score and Figure 5.8 shows the time in minutes to run a beam-search on the entire validation dataset (2000 summaries). These experiments were conducted on

the pretrained model at epoch 13[11]. In further experiments, we use $k1 = 3$ and $k2 = 6$.

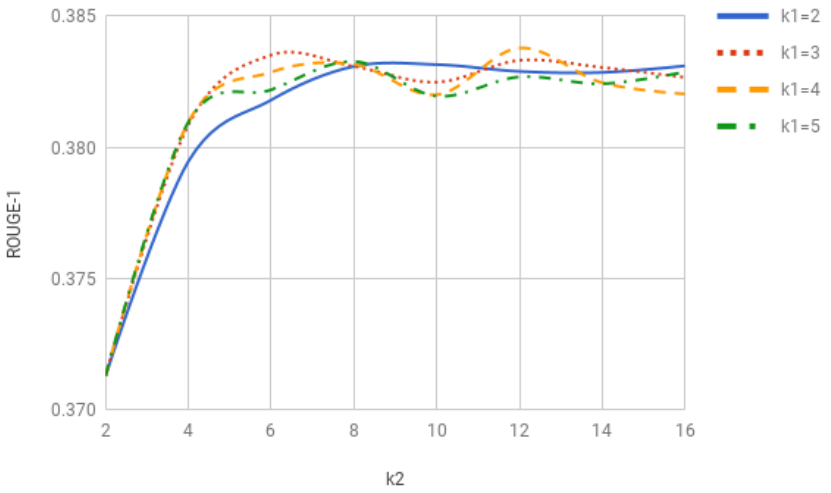


Figure 5.5: ROUGE-1 scores when varying the beam search parameters k1 and k2

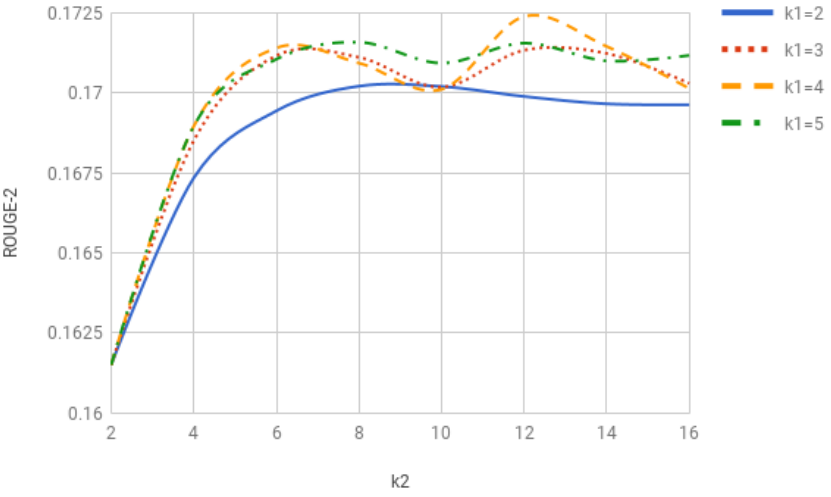


Figure 5.6: ROUGE-2 scores when varying the beam search parameters k1 and k2

[11]Epoch 13 was chosen as it is very close to the number of epochs used to train the best model reported by [See et al., 2017].

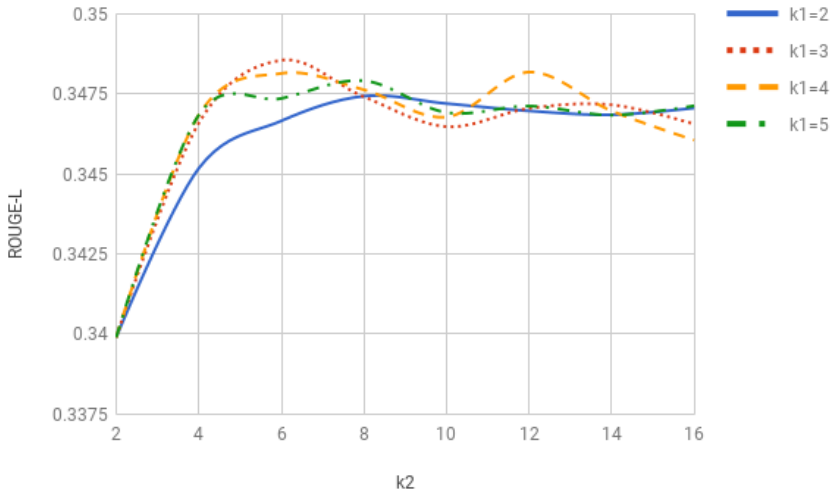


Figure 5.7: ROUGE-L scores when varying the beam search parameters k1 and k2

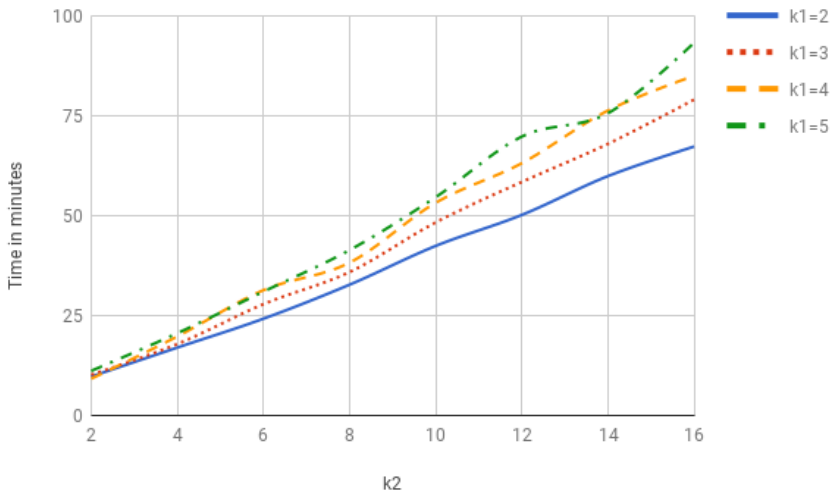


Figure 5.8: Time in minutes to run beam search on the entire validation dataset (2000 summaries), when varying the parameters k1 and k2

5.5 Generator pretraining

In this section, we will present the results for our pretrained model. We will present the training loss and ROUGE scores. The results presented in this section uses the hyperparameters described in Section 5.4.1, but for convenience purposes we will summarize them in a table here, as seen in Table 5.1. The model is trained on the infrastructure mentioned in Section 5.2 for 20 epochs, which takes about 2 days and 6 hours.

Hyperparameter	value
Batch size	16
Learning rate	0.015
L2 penalty	10^{-5}
Embedding size	100
Encoder hidden size	128
Decoder hidden size	256
Teacher forcing ratio	1.0

Table 5.1: Pretraining hyperparameters

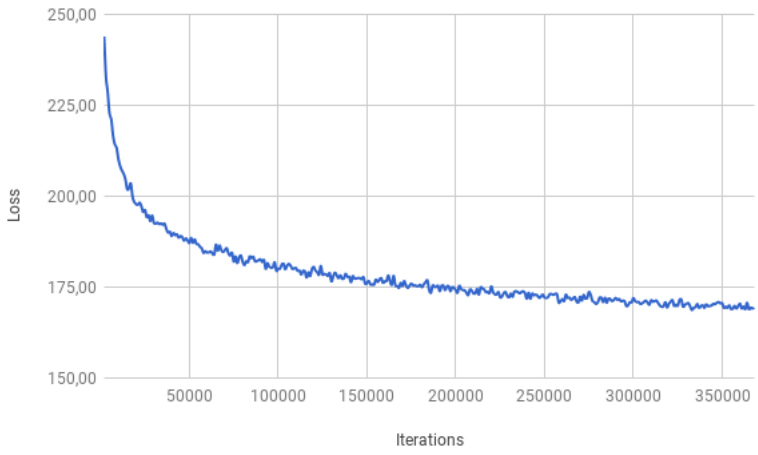
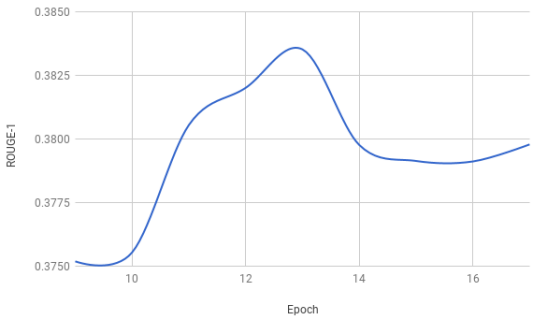


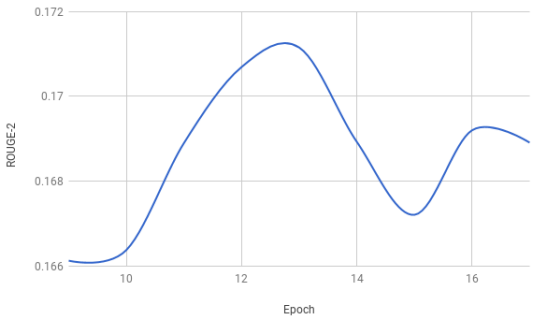
Figure 5.9: The training loss during pretraining

Figure 5.9 shows the training loss during pretraining from 0 to 368 000 iterations, where 18 400 iterations is equal to one epoch. We have chosen not to include the evaluation loss here since it does not give a correct picture of how well the model is generalizing[12]. The loss presented here is averaged over the batch size. Figure 5.10 shows ROUGE-1, ROUGE-2 and ROUGE-L scores for the pretrained model from epoch 9 to 17. Table 5.2 shows the ROUGE scores on the test dataset for the best pretrained model (epoch 13).

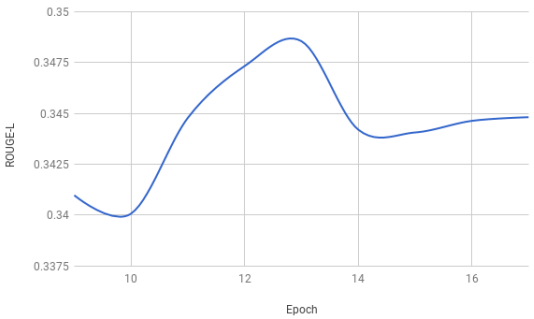
[12]MLE without teacher forcing will yield a very high loss, which does not give any insight into generalization.



(a) ROUGE-1 scores for the pretrained model



(b) ROUGE-2 scores for the pretrained model



(c) ROUGE-L scores for the pretrained model

Figure 5.10: ROUGE scores for the pretrained model

	ROUGE-1	ROUGE-2	ROUGE-L
Pretrained model (epoch 13)	0.3763	0.1642	0.3410

Table 5.2: ROUGE scores for the best performing pretrained model, calculated on the test dataset

5.6 Discriminator pretraining

As explained in Section 4.3, we need to pretrain the discriminator used in our GAN experiments. The dataset used to pretrain the discriminator is a combination of fake examples created by the pretrained generator and real examples from the training dataset mentioned in Section 5.1.

To create the fake examples, we use the articles from the training dataset as input, sampling every token until the model outputs <EOS>, or maximum length (100) is reached. We repeat this process 50 times (similar to [Yu et al., 2016]) and combine all the fake examples with the real data. To balance the complete discriminator training dataset, we duplicate the real data 50 times. This yields a total of 29 452 500 discriminator training examples. We train the discriminator for a total of 5 epochs on the complete discriminator training dataset. This process of creating discriminator training examples and training for multiple epochs on those examples is similar to how we train the discriminator during adversarial training, as seen in Code 4.3.

We generate a discriminator test dataset by sampling from the articles in the test dataset mentioned in Section 5.1. This gives us a total of 11 000 fake test examples, which we combine with the real test examples, yielding a total of 22 000 discriminator test examples.

Table 5.3 shows the precision, recall, F1 score and accuracy for the pretrained discriminator on the combined test examples.

Metric	Score
Precision	0.8212
Recall	0.8039
F1 score	0.8013
Accuracy	0.8040

Table 5.3: Metric scores for the pretrained discriminator

5.7 Reinforcement learning parameters and strategies

In this section, we will cover the parameters and strategies explained in Section 4.2, 4.3 and 4.4 regarding reinforcement learning and adversarial training. ROUGE scores, reward and running time is presented to show the differences between the different parameter settings and strategies.

5.7.1 Reward with baseline

In Section 4.2, we mention the importance of a baseline to reduce the variance of the reward signal during training. Figure 5.11 and Figure 5.12 shows the ROUGE-1 and ROUGE-L scores, respectively, for epochs 1 to 5 when using two different baselines; running average and argmax (greedy search), with and without truncating negative rewards (only allowing positive rewards). The figures also include results when no baseline is used. Training is done using the naive roll-out strategy (Code 4.1), optimizing on ROUGE-1. In further experiments, we use the running average baseline if not explicitly said otherwise.

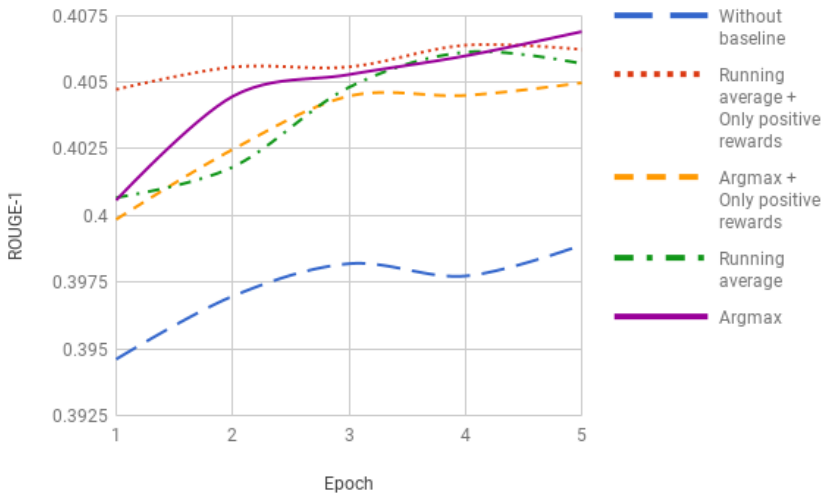


Figure 5.11: ROUGE-1 scores using different baselines

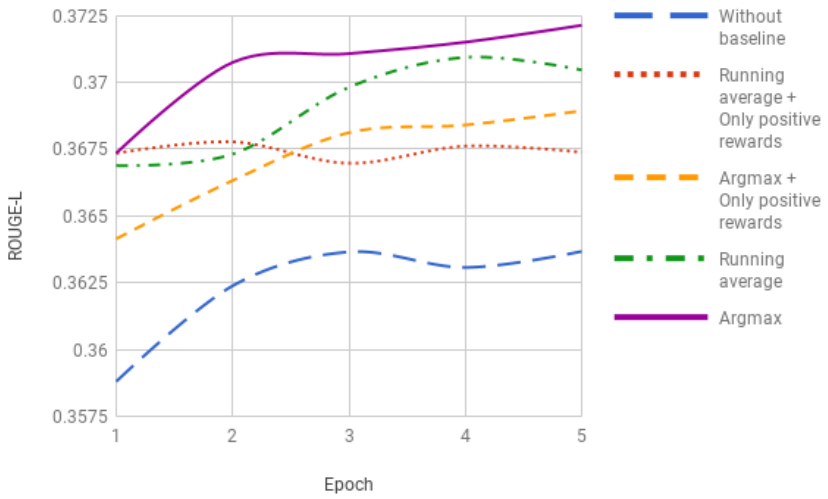


Figure 5.12: ROUGE-L scores using different baselines

5.7.2 Number of Monte-Carlo roll-outs

The number of roll-outs (N) has a huge impact on time when training with the Monte-Carlo roll-out strategy (Code 4.2), as can be seen in Figure 5.13. However, this parameter is not obvious as to how much it matters for training stability. Figure 5.14 shows the ROUGE-1 score up to 2 epochs when optimizing on ROUGE-1 using this strategy with different number of roll-outs. In these experiments, we used the running-average baseline where we truncated negative rewards. In further experiments, we use $N = 8$, if not explicitly said otherwise.

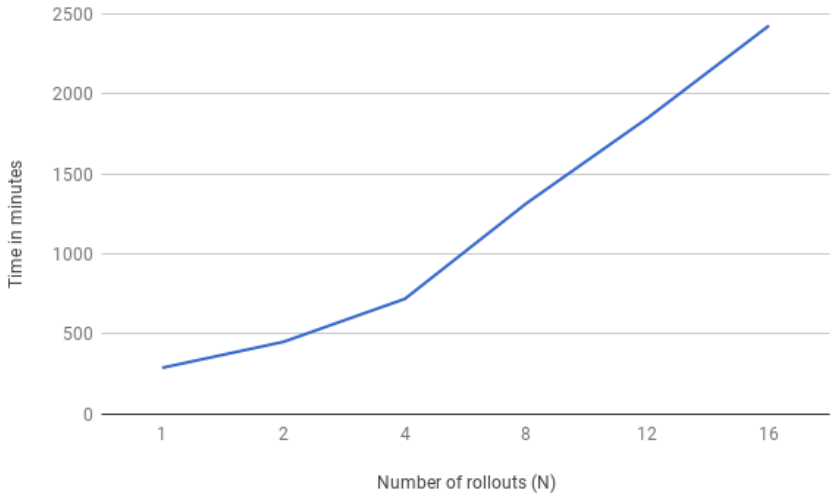


Figure 5.13: Time in minutes to run one epoch when varying the number of roll-outs (N)

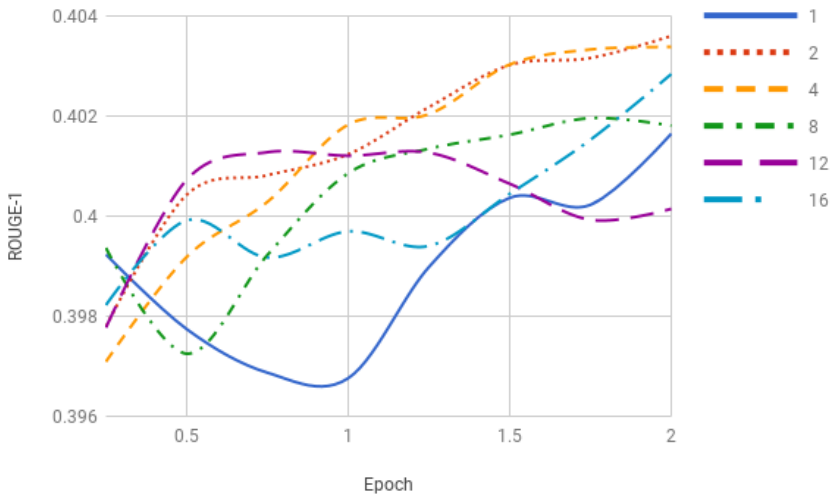


Figure 5.14: ROUGE-1 scores when varying the number of roll-outs (N)

5.7.3 Sequence accumulation

As mentioned in Section 4.2, when employing the Monte-Carlo roll-out strategy (Code 4.2), we have to choose between accumulating a sampled token (sampling) and the top-1 scoring tokens (argmax) from the decoder output for each time-step. Figure 5.15 and 5.16 shows ROUGE-1 and ROUGE-L scores, respectively, for a total of 2 epochs when optimizing on ROUGE-1 using the two different ways to accumulate the previous generated sequence. We show results both with and without MLE loss (Equation 3.7), where $\beta = 0.9984$ and the running average baseline is used. We use sampling for all time-steps in all further experiments.

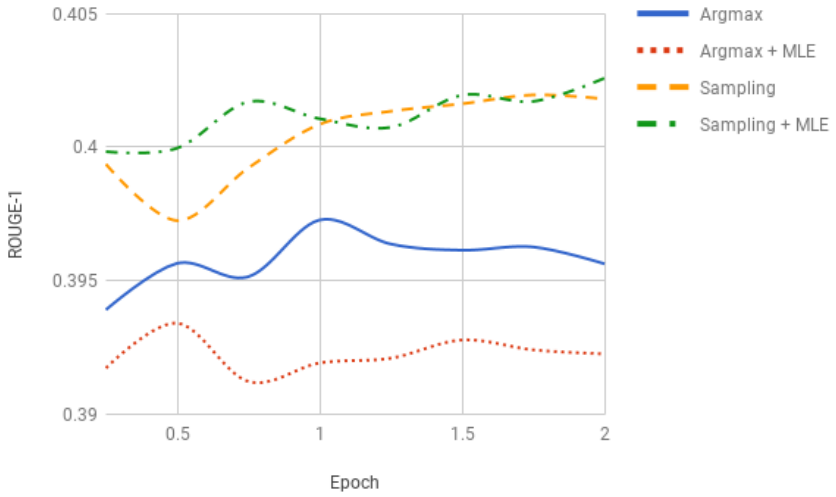


Figure 5.15: ROUGE-1 scores when using sampling or argmax sequence accumulation, both with and without MLE

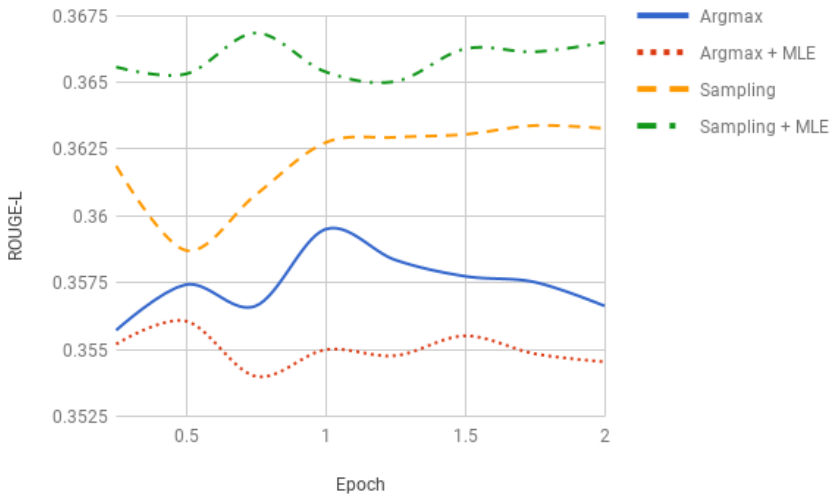


Figure 5.16: ROUGE-L scores when using sampling or argmax sequence accumulation, both with and without MLE

5.7.4 Adversarial training parameters

In a GAN setting, there is a trade-off between training the generator and the discriminator. Properly balancing the training of the two parts should hopefully stabilize the training and make the generator better, as mentioned in Section 4.3. Code 4.3 defines three parameters: *n_generator*: The number of iterations we train the generator before training the discriminator. *n_discriminator*: The number of iterations we train the discriminator before training the generator. *n_epochs_discriminator*: The number of epochs we train the discriminator on the data created for this iteration. Figure 5.17 shows the ROUGE-1 scores for epochs 1 to 5, varying the parameters *n_generator* and *n_discriminator*. The parameters are presented as: [n_generator, n_discriminator, n_epochs_discriminator]. Figure 5.18 shows the ROUGE-1 scores when varying the parameters *n_discriminator* and *n_epochs_discriminator* for epochs 1 to 5. Training is done using the naive roll-out strategy (Code 4.1), using the pretrained discriminator explained in Section 5.6.

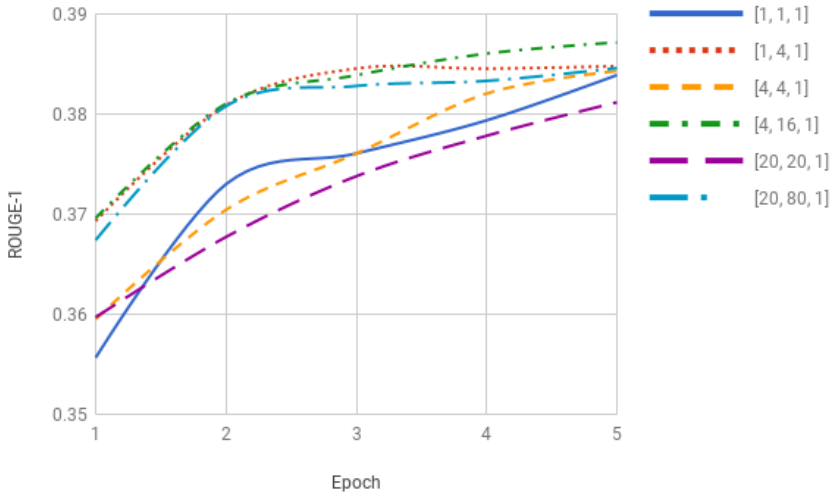


Figure 5.17: ROUGE-1 scores when varying n_generator and n_discriminator

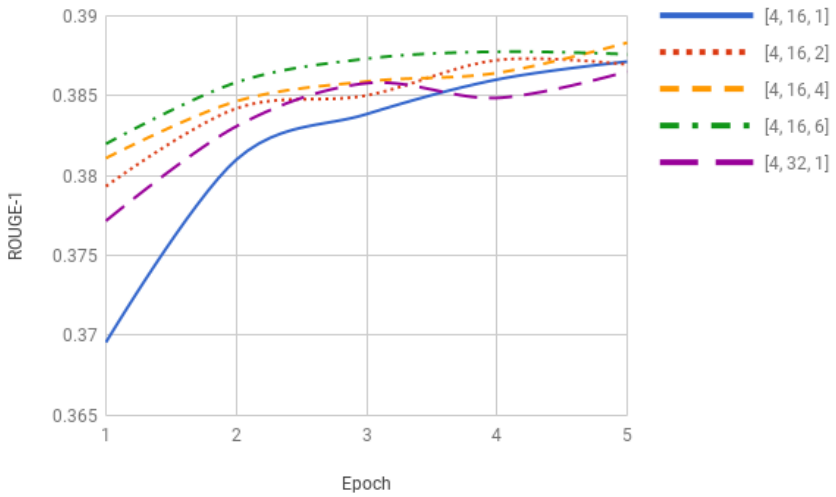


Figure 5.18: ROUGE-1 scores when varying n_discriminator and n_epochs_discriminator

Figure 5.19 shows the time in minutes for the same parameters seen in Figure 5.18 after running for one epoch.

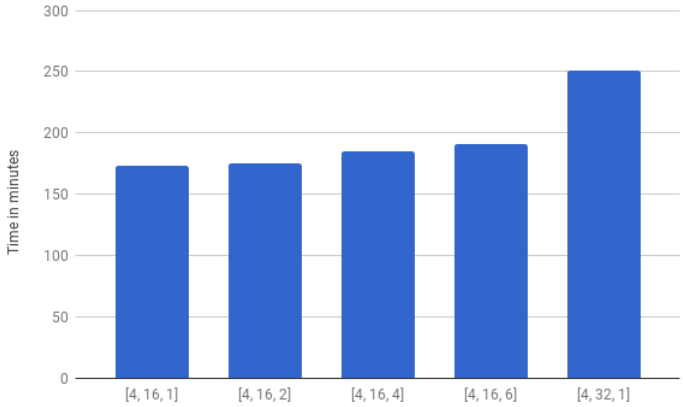


Figure 5.19: Time in minutes when varying n_discriminator and n_epochs_discriminator

Figure 5.20 shows the ROUGE-1 scores up to epoch 4 when we set the parameter *n_discriminator* = 0.

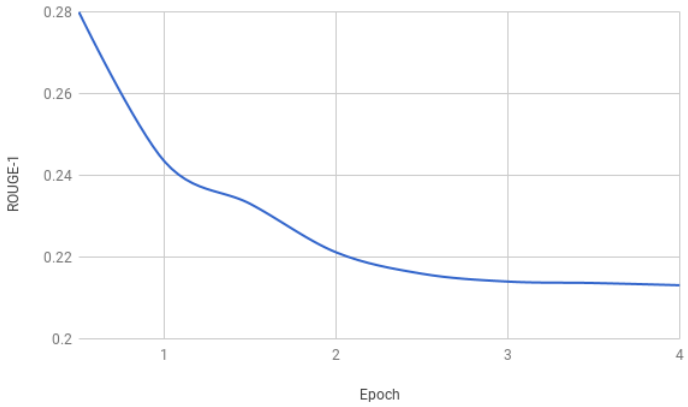


Figure 5.20: ROUGE-1 scores when setting n_discriminator = 0

5.7.5 Discriminator without sufficient pretraining

Table 5.4 shows the precision, recall, F1 score and accuracy on the discriminator test dataset for a discriminator trained on one single epoch of fake generated data combined with real data (a total of 589 050 examples). Here, we only train on the combined dataset for one epoch.

Metric	Score
Precision	0.6046
Recall	0.6079
F1 score	0.6063
Accuracy	0.5926

Table 5.4: Metric scores for a discriminator without sufficient pretraining

Figure 5.21 and 5.22 shows the ROUGE-1 and ROUGE-L scores, respectively, when training

with the discriminator without sufficient pretraining (Table 5.4) as well as the ROUGE scores when training with the sufficiently pretrained discriminator explained in Section 5.6 (Table 5.3). Training is done using the naive roll-out strategy (Code 4.1), using the parameters *n_generator* = 4, *n_discriminator* = 16 and *n_epochs_discriminator* = 4. Results are shown up to 5 epochs. Both figures include a line showing the ROUGE scores for the best pretrained model (epoch 13), for comparison. Figure 5.23 shows the reward for the same two discriminators trained with the same strategy and parameters as in Figure 5.21 and 5.22.

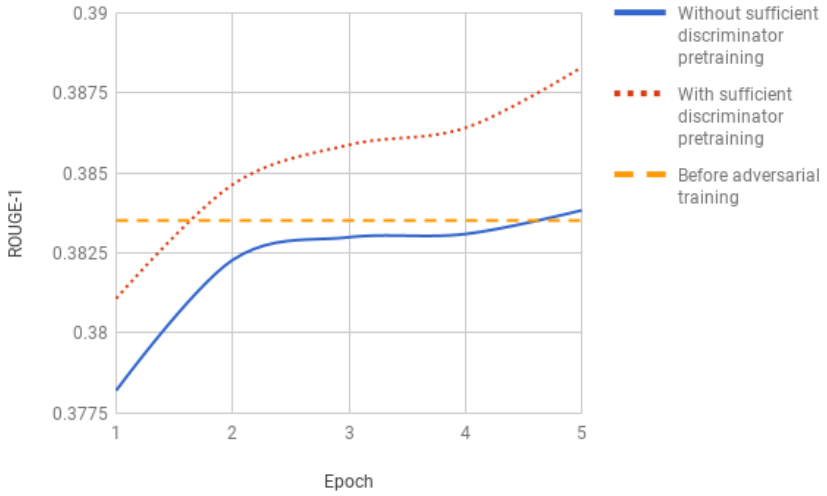


Figure 5.21: ROUGE-1 scores when training with a discriminator both with and without sufficient pretraining. A straight line showing the best performing pretrained model (before adversarial training) is also included.

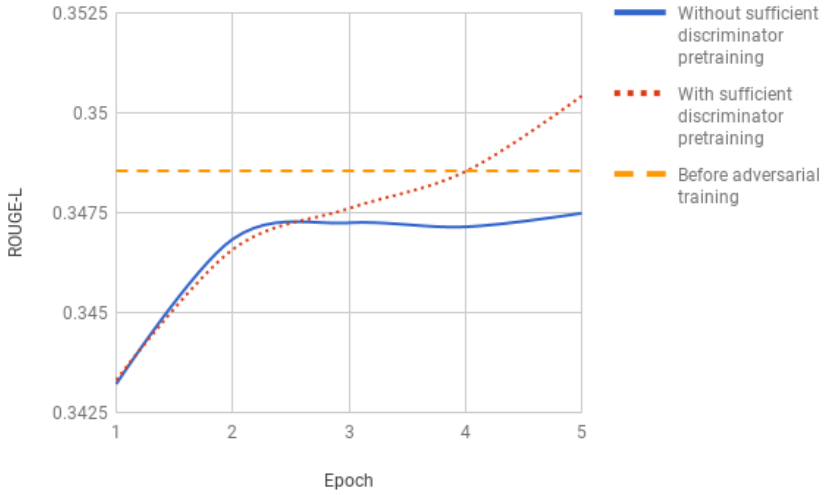


Figure 5.22: ROUGE-L scores when training with a discriminator both with and without sufficient pretraining. A straight line showing the best performing pretrained model (before adversarial training) is also included.

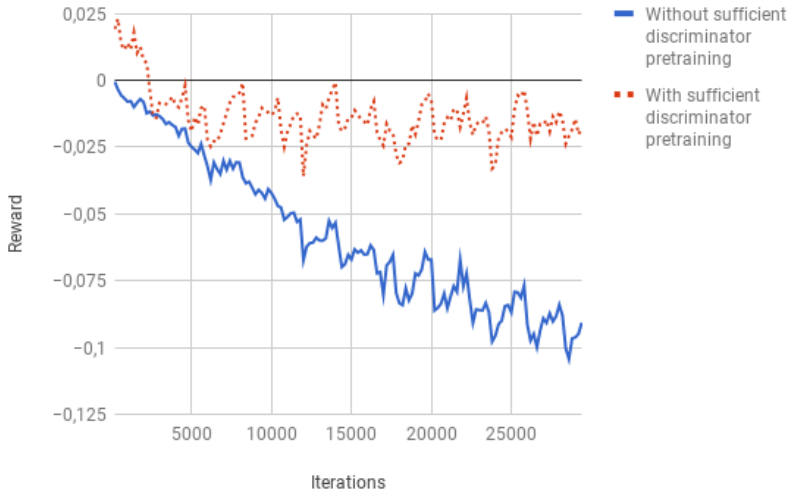


Figure 5.23: Average reward at every 200 iterations, with and without a sufficient pretrained discriminator

5.7.6 Mixed RL and MLE

[Paulus et al., 2017] show that a mixed objective of MLE and RL (Equation 3.7) can outperform both pure MLE and pure RL objectives (Section 3.2), where MLE works as a balancing factor for the training. In Figure 5.24 we can see the ROUGE-1 score development for different β values over 5 epochs. Here, we used the running average baseline where we truncated negative rewards. Figure 5.25 shows the ROUGE-1 score up to 5 epochs when we set $\beta = 0$ (only MLE)[13]. In both figures, training is done using the naive roll-out strategy (Code 4.1), optimizing on ROUGE-1.

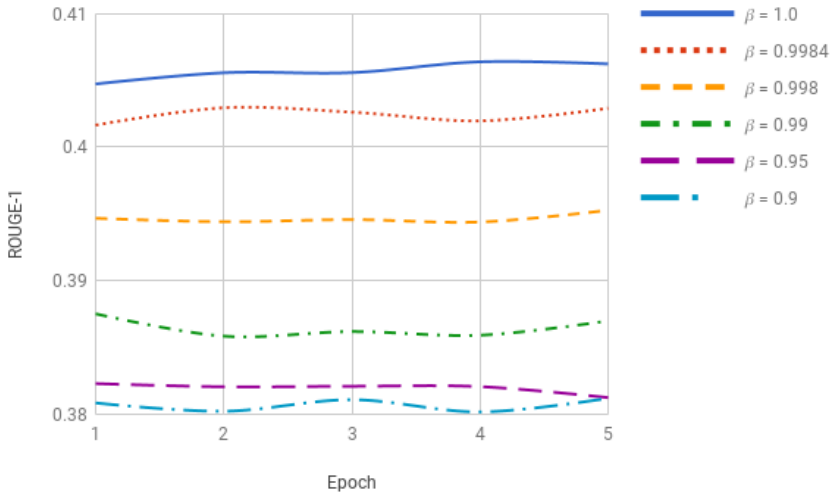


Figure 5.24: ROUGE-1 scores for different β values

[13]This is different from the pretraining after epoch 13 seen in Figure 5.10, as we have reduced the learning rate to 0.0001 and increased the batch size to 50.

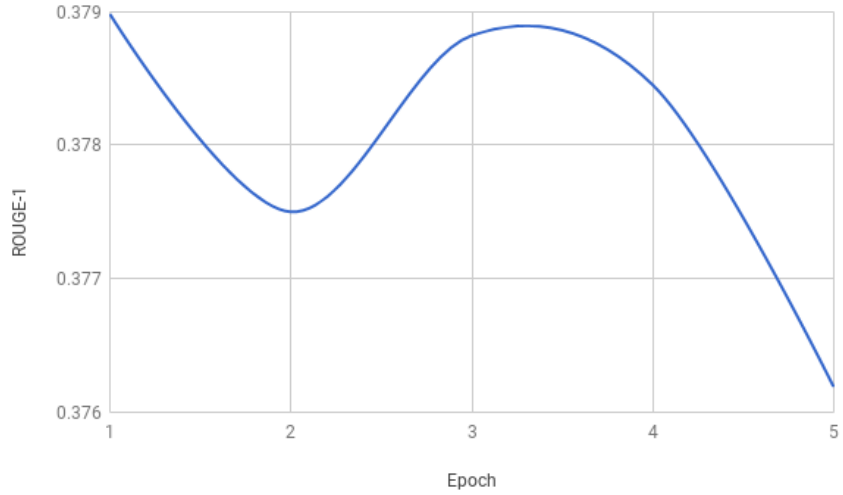


Figure 5.25: ROUGE-1 scores for $\beta = 0$

5.7.7 Combined objective

In this section, we will present results for combination of objectives, by using Equation 4.2 as the reward signal. Figure 5.26 shows ROUGE-1 scores when varying the λ parameter. Optimization is towards the combined objective (Equation 4.2) of ROUGE-1 and the discriminator (Section 5.6). Training is done using the naive roll-out strategy (Code 4.1), with the parameters *n_generator* = 4, *n_discriminator* = 16 and *n_epochs_discriminator* = 4. Results are shown up to 5 epochs. Figure 5.27 shows the same ROUGE scores, trained with the same parameters as the results in Figure 5.26, but the objective is changed to a combination of ROUGE-2 and the discriminator.

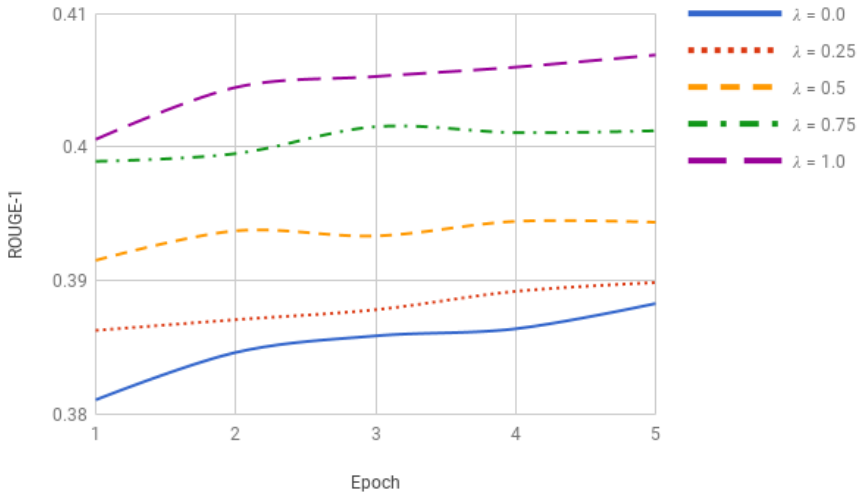


Figure 5.26: ROUGE-1 scores when varying the λ parameter. Optimizing towards the combined objective of ROUGE-1 and the discriminator.

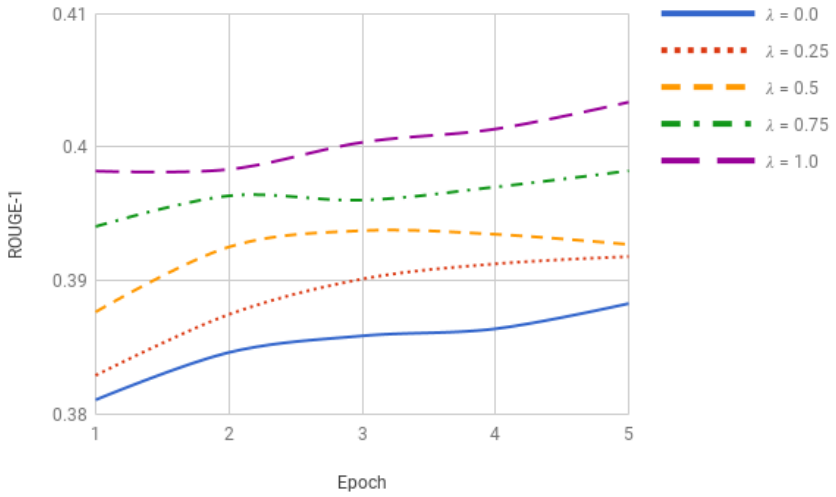


Figure 5.27: ROUGE-1 scores when varying the λ parameter. Optimizing towards the combined objective of ROUGE-2 and the discriminator.

5.8 Quantitative results on different objectives and strategies

In this section, we will present results on the 4 different objectives: ROUGE-1, ROUGE-2, discriminator and combined (ROUGE-2/discriminator). We show results using both the naive roll-out strategy (Code 4.1) and the Monte-Carlo roll-out strategy (Code 4.2). All the results are calculated using the test dataset, where we pick the best performing model (number of training epochs) based on results from the validation set[14]. We also show average length of summaries created from the models trained with the different objectives, and the number of summaries that reached maximum length. Finally, we include the percentage of novel n-grams and the percentage of novel sentences in the summaries.

Table 5.5 and 5.6 shows the ROUGE scores of the best performing models when optimizing on the ROUGE-1 and ROUGE-2 objectives, respectively. In these experiments, the argmax baseline is used.

	ROUGE-1	ROUGE-2	ROUGE-L
Naive roll-out strategy. $\beta = 1.0$ (epoch 5)	0.4039	0.1789	0.3680
Naive roll-out strategy. $\beta = 0.9984$ (epoch 4)	0.4009	0.1773	0.3636
Monte-Carlo roll-out strategy. $\beta = 1.0$ (epoch 3.5)	0.3982	0.1774	0.3643
Monte-Carlo roll-out strategy. $\beta = 0.9984$ (epoch 2.5)	0.3983	0.1755	0.3614

Table 5.5: Results when optimizing on ROUGE-1 with different roll-out strategies

[14]Validation set results for every epoch are left out because they are irrelevant to any discussion. They are merely used to find the best performing model (early stopping).

	ROUGE-1	ROUGE-2	ROUGE-L
Naive roll-out strategy. β = 1.0 (epoch 10)	0.4056	0.1822	0.3673
Naive roll-out strategy. β = 0.9984 (epoch 3)	0.4030	0.1799	0.3642
Monte-Carlo roll-out strategy. β = 1.0 (epoch 1.5)	0.3980	0.1764	0.3606
Monte-Carlo roll-out strategy. β = 0.9984 (epoch 3)	0.4001	0.1776	0.3616

Table 5.6: Results when optimizing on ROUGE-2 with different roll-out strategies

Table 5.7 shows the ROUGE scores of the best performing models when optimizing on the discriminator objective. In these experiments, the running average baseline is used.

	ROUGE-1	ROUGE-2	ROUGE-L
Naive roll-out strategy. β = 1.0 (epoch 6)	0.3827	0.1648	0.3439
Naive roll-out strategy. β = 0.9984 (epoch 7)	0.3794	0.1648	0.3411
Monte-Carlo roll-out strategy. β = 1.0 (epoch 4)	0.3817	0.1651	0.3426
Monte-Carlo roll-out strategy. β = 0.9984 (epoch 5)	0.3798	0.1659	0.3412

Table 5.7: Results when optimizing on the discriminator objective with different roll-out strategies

Table 5.8 shows the ROUGE scores of the best performing models when optimizing on the combined (ROUGE-2/discriminator) objective. In these experiments, the running average baseline is used.

	ROUGE-1	ROUGE-2	ROUGE-L
Naive roll-out strategy. β = 1.0 (epoch 7)	0.3892	0.1697	0.3499
Naive roll-out strategy. β = 0.9984 (epoch 7)	0.3816	0.1671	0.3427
Monte-Carlo roll-out strategy. β = 1.0 (epoch 5)	0.3869	0.1694	0.3473
Monte-Carlo roll-out strategy. β = 0.9984 (epoch 3)	0.3815	0.1669	0.3427

Table 5.8: Results when optimizing on the combined (ROUGE-2/discriminator) objective with different roll-out strategies

Table 5.9 shows a summary of ROUGE scores from the best obtained results on the different objectives (with MLE). The table also includes ROUGE scores from the pretrained model, as well as results from [See et al., 2017] (Pointer-Generator Network), [Paulus et al., 2017] (Deep RL)[15] and [Liu et al., 2017] (Abstractive GAN). The results on the objectives of ROUGE-1, ROUGE-2, discriminator and combined are taken from Tables 5.5, 5.6, 5.7 and 5.8, respectively.

[15]For Deep RL [Paulus et al., 2017], we select the results obtained with RL + MLE training, even though it obtains only the second highest ROUGE scores. We select this model because the summaries with this model obtained highest readability and is thus more relevant for the summarization task.

5.8. Quantitative results on different objectives and strategies

	ROUGE-1	ROUGE-2	ROUGE-L
Pretrained model (epoch 13)	0.3763	0.1642	0.3410
ROUGE-1. $\beta = 0.9984$ (epoch 4)	0.4009	0.1773	0.3636
ROUGE-2. $\beta = 0.9984$ (epoch 3)	0.4030	0.1799	0.3642
Discriminator. $\beta = 0.9984$ (epoch 5)	0.3798	0.1659	0.3412
Combined. $\beta = 0.9984$ (epoch 7)	0.3816	0.1671	0.3427
Pointer-Generator Network	0.3953	0.1728	0.3638
Deep RL	0.3987	0.1582	0.3690
Abstractive GAN	0.3992	0.1765	0.3671[16]

Table 5.9: Summary of the best results on the different objectives. Also includes results from the pretrained model, as well as results from [See et al., 2017] (Pointer-Generator Network), [Paulus et al., 2017] (Deep RL) and [Liu et al., 2017] (Abstractive GAN).

Table 5.10 shows the average length of summaries for the pretrained model and the best models trained with the different objectives. We show results both with and without MLE. The table also includes the average summary length of the ground truth (real test data).

	Ground truth	Pretrained	ROUGE-1	ROUGE-2	Discriminator	Combined
Without MLE	57.6	49.6	76.4	85.1	54.7	55.9
With MLE	57.6	49.6	66.4	66.1	49.5	49.0

Table 5.10: Average length of summaries, with and without MLE

Table 5.11 shows the number of summaries that reached maximum length for the pretrained model and the best models trained with the different objectives. These are summaries that did not produce an <EOS> token before reaching maximum length, which means that the last sentence may have ended abruptly.

	Pretrained	ROUGE-1	ROUGE-2	Discriminator	Combined
Without MLE	1	168	862	0	0
With MLE	1	29	37	0	0

Table 5.11: Number of summaries that reached maximum length

Figures 5.28, 5.29, 5.30 and 5.31 shows the percentage of novel 2-grams, 3-grams, 4-grams and sentences, respectively. The figures show results from the best models optimized towards the different objectives (ROUGE-1, ROUGE-2, the discriminator and combined (ROUGE-2/discriminator)), as well as results from the pretrained model and the ground truth summaries.

[16]When downloading the generated summaries from the test dataset from [Liu et al., 2017] and running ROUGE evaluation, we got a ROUGE-L score of 0.364, but the values from ROUGE-1 and ROUGE-2 seemed correct. We are not sure what causes the discrepancy between our test and their reports.

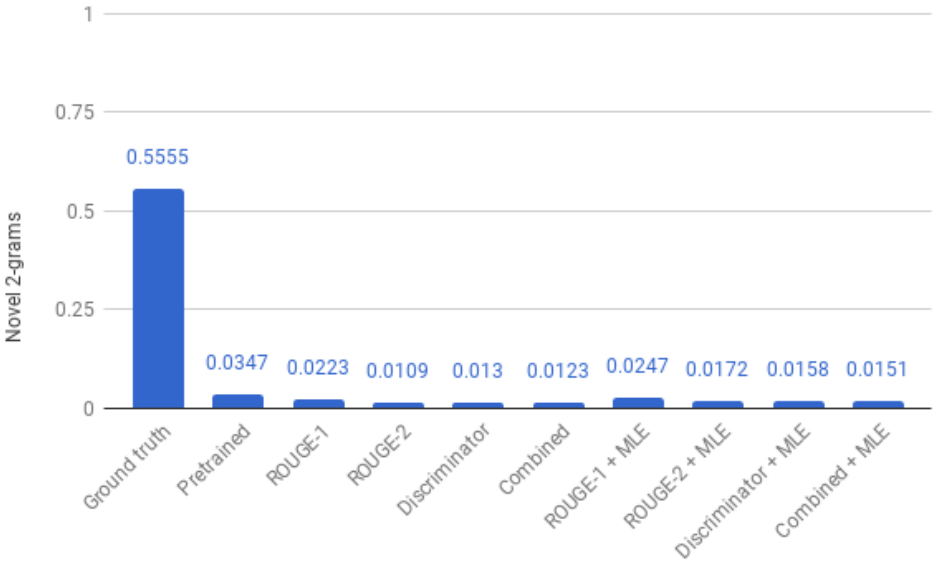


Figure 5.28: Percentage of novel 2-grams

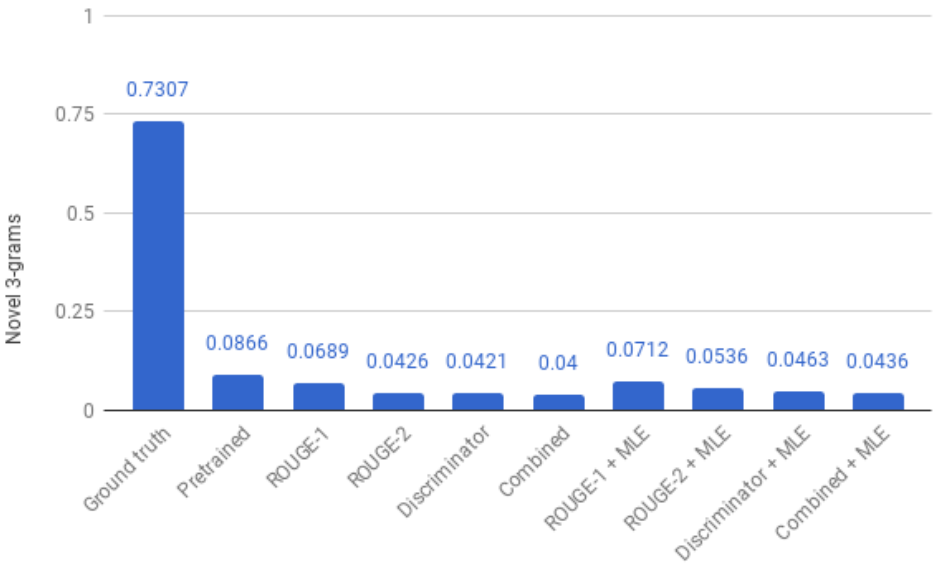


Figure 5.29: Percentage of novel 3-grams

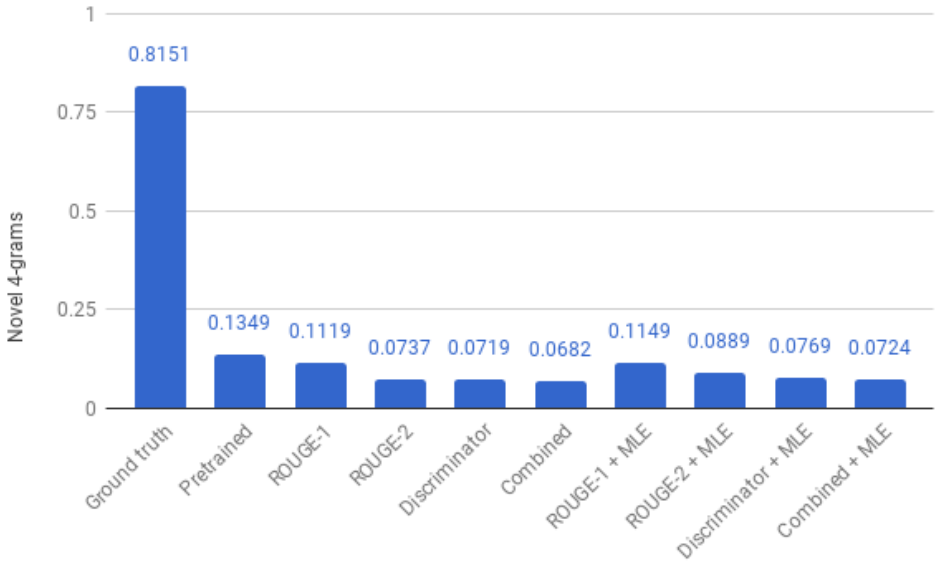


Figure 5.30: Percentage of novel 4-grams

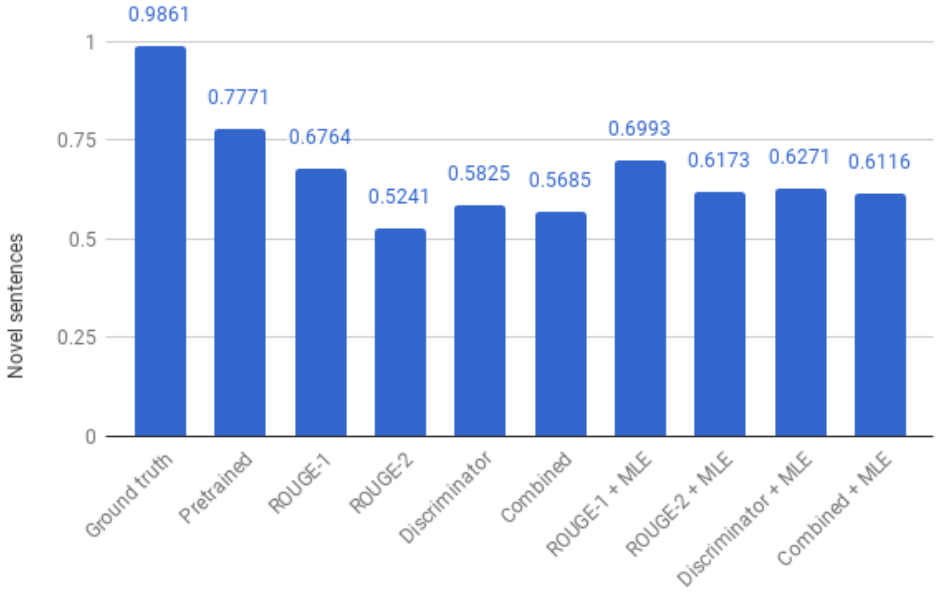


Figure 5.31: Percentage of novel sentences

5.9 Qualitative results

In this section, we will present qualitative results for the pretrained model, and the models trained with the different objectives. We show results on the best performing models, as summarized in Table 5.9. We also show an example where we compare training with and without MLE. More qualitative examples from the best performing models trained with the different objectives are

included in Appendix A. The GitHub repository[17] includes results on the entire test dataset.

Figure 5.32 shows an example article and the summaries created from the models trained with the ROUGE-2 objective, both with and without MLE. Figure 5.33 shows an example article, the ground truth summary, the summary generated by the pretrained model, and the summaries created from the models trained with the different objectives as summarized in Table 5.9 (results from other papers are not included).

Article
by . alex sharp . chev walker clearly is n't afraid of getting stuck in when it matters . the bradford bulls star posted a gruesome image on his official twitter account of his hand sporting a huge gash . along with the x-rated picture , walker posted the message ' well after doing this in the game on sunday , i thought i was gon na be having an op but just been told does n't need it . ' gruesome : chev walker had his hand slashed open attempting a tap tackle against huddersfield on sunday . walker sustained the injury whilst attempting a tap tackle in the bulls ' 52-26 thrashing at the hands of huddersfield giants . a sixth straight defeat for bradford confirmed their relegation from super league will start next season outside rugby league 's elite for the first time in over four decades . double pain : walker suffered horrific hand injury and hurt of relegation with bradford bulls on sunday .

Summary from model trained without MLE
chev walker suffered horrific hand injury and hurt of relegation with bradford bulls on sunday . the bradford bulls star posted a gruesome image on his official twitter account of his hand sporting a huge gash . walker sustained the injury whilst attempting a tap tackle in the bulls ' 52-26 thrashing at the hands of huddersfield giants . chev walker had his hand slashed open from super league will start next season outside rugby league 's elite for the first time in over four decades .

Summary from model trained with MLE
chev walker suffered horrific hand injury and hurt of relegation with bradford bulls on sunday . the bradford bulls star posted a gruesome image on his official twitter account of his hand sporting a huge gash . walker sustained the injury whilst attempting a tap tackle in the bulls ' 52-26 thrashing at the hands of huddersfield giants .

Figure 5.32: Summaries created from models trained on the objective of ROUGE-2, with and without MLE

[17] GitHub repository: https://github.com/borgarlie/tdt4900-robo-journalism

Article
bayern munich star david alaba suffered a knee ligament injury in their 2-0 victory over as roma in the champions league on wednesday . the gifted 22-year-old austria international was in outstanding form , setting up the first goal for franck ribery . david alaba had to be taken off in the second half of bayern 's 2-0 win over roma on wednesday night . the austrian international had set up the opening goal for franck ribery at the allianz arena . he was taken off injured in the 81st minute as bayern cruised to the champions league knockout stage after their win over the italians secured them top spot in group e. ' there is a problem with david . he was injured , ' bayern coach pep guardiola told reporters . ' he will get checked out tomorrow when we will know more . ' the club later said on twitter the player sustained ' a medial collateral ligament injury ' . bayern were already without a string of key injured midfielders including bastian schweinsteiger , javi martinez and thiago alcantara . alaba joins a growing midfield injury list for the german champions .

Ground truth summary
david alaba taken off in bayern 's champions league win against roma . alaba joins bastian schweinsteiger , javi martinez and thiago on injury list . austrain international will be assessed on thursday .

Summary from pretrained model
david alaba had to be taken off in the second half of bayern 's 2-0 win over roma in the champions league on wednesday night . the austrian international had set up the opening goal for franck ribery at the allianz arena . bayern coach pep guardiola told reporters on twitter the player sustained ' a medial collateral ligament injury '

Summary from model trained with the ROUGE-1 objective
david alaba had to be taken off in the second half of bayern 's 2-0 win over roma in the champions league on wednesday night . the austrian international had set up the opening goal for franck ribery at the allianz arena . david alaba suffered a knee ligament injury in their 2-0 victory over roma . bayern coach pep guardiola said on twitter the player sustained ' a medial collateral ligament injury '

Summary from model trained with the ROUGE-2 objective
david alaba had to be taken off in the second half of bayern 's 2-0 win over roma in the champions league on wednesday night . the austrian international had set up the opening goal for franck ribery at the allianz arena . bayern munich star david alaba suffered a knee ligament injury in their 2-0 victory over roma .

Summary from model trained with the discriminator objective
david alaba had to be taken off in the second half of bayern 's 2-0 win over roma in the champions league on wednesday night . the austrian international had set up the opening goal for franck ribery . the club later said on twitter the player sustained ' a medial collateral ligament injury ' bayern were already without a string of key injured midfielders including bastian schweinsteiger , javi martinez and thiago alcantara .

Summary from model trained with the combined objective
david alaba had to be taken off in the second half of bayern 's 2-0 win over roma in the champions league on wednesday night . the austrian international had set up the opening goal for franck ribery . the club later said on twitter the player sustained ' a medial collateral ligament injury '

Figure 5.33: Summaries created from all the best performing models

5.10 Code optimization

In this section, we will show runtime differences between the algorithms and ideas presented in Section 4.5. The experiments are conducted using the Monte-Carlo roll-out strategy (Code 4.2). We also show ROUGE scores on the test dataset when necessary.

Table 5.12 shows the time in minutes spent for the single operation of naive masking (Code 4.4) and vectorized masking (Code 4.5) for a total of 1000 iterations. The experiments are conducted with $N = 8$, using the discriminator objective and the running average baseline.

Naive masking	200 min
Vectorized masking	15 min

Table 5.12: Time difference between naive masking and vectorized masking

Figure 5.34 shows the total time in minutes to run 1000 iterations using naive roll-out (Code 4.6) and batch-parallel roll-out (Code 4.7). The experiments are conducted with $N = 8$ for different batch sizes. Here, the discriminator objective is used with the running average baseline.

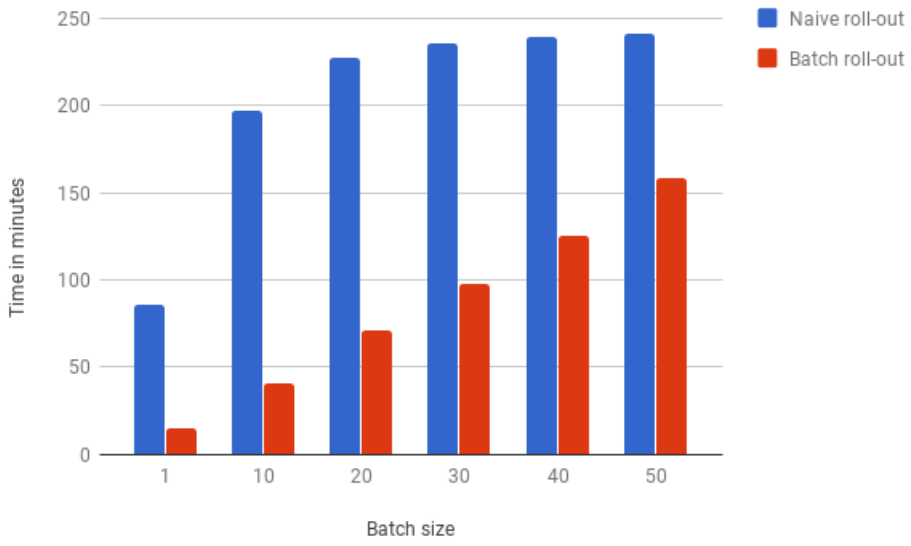


Figure 5.34: Time in minutes for naive roll-out and batch-parallel roll-out when varying batch size

Figure 5.35 shows the time in minutes for 1000 iterations when varying the maximum number of decoding time-steps. The experiments are conducted with $N = 8$, the discriminator objective and the running average baseline.

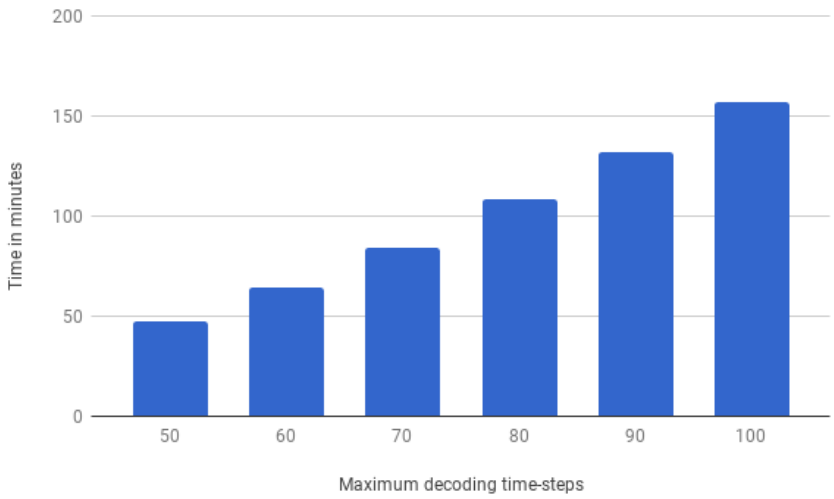


Figure 5.35: Time in minutes when varying the maximum number of decoding time-steps

Figure 5.36 shows the time in minutes for one epoch when varying the sample rate. Figure 5.37 shows the ROUGE-1 scores up to one epoch when varying the sample rate. In these experiments, $N = 4$ and the ROUGE-1 objective is used with the argmax baseline.

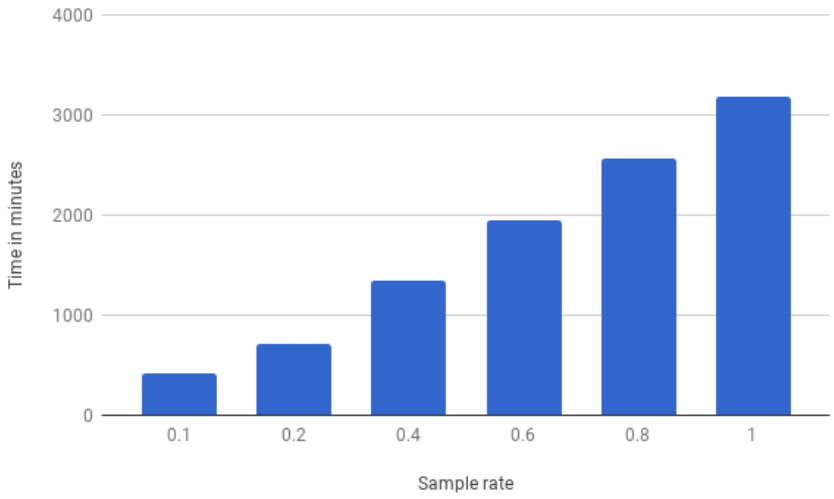


Figure 5.36: Time in minutes when varying the sample rate

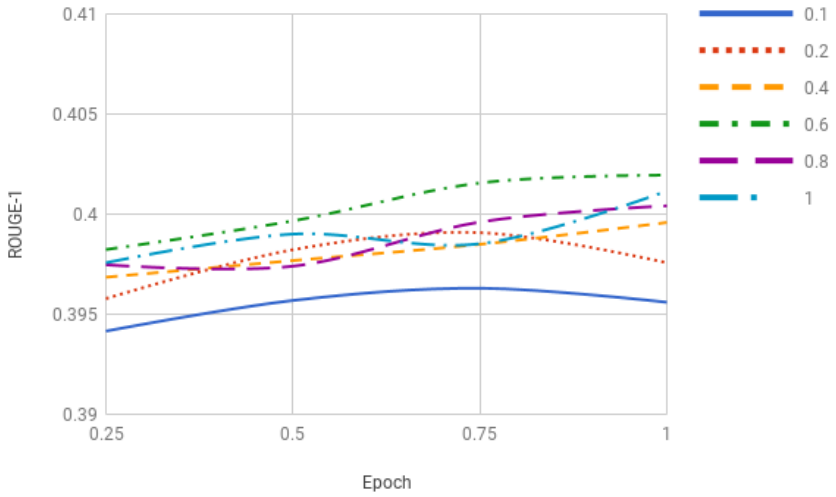


Figure 5.37: ROUGE-1 scores when varying the sample rate.

Figure 5.38 shows the time in minutes for 1000 iterations when varying the time-step to start checking for EOS. We experiment with both sampling and argmax when accumulating the previous sequence. The experiments are conducted with $N = 8$, the discriminator objective and the running average baseline.

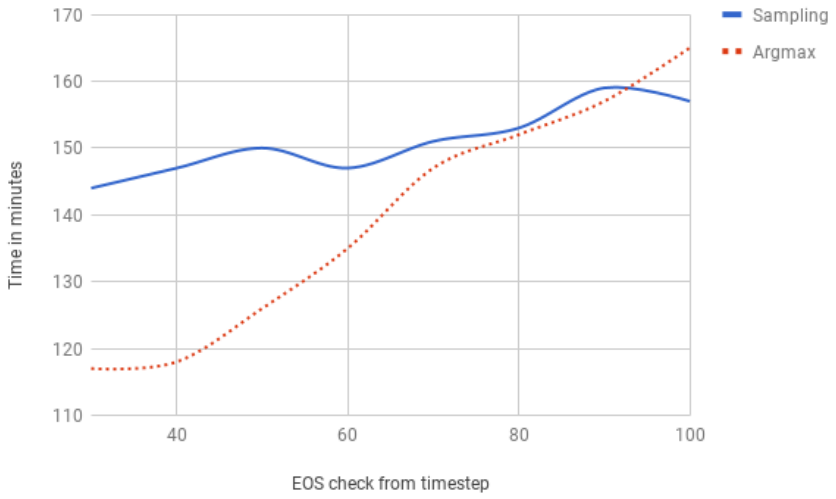


Figure 5.38: Time in minutes when varying the time-step to start checking for EOS

5.11 Running time for the roll-out strategies

In this section, we will present the running time for the two roll-out strategies. Table 5.13 shows the time in minutes to run one epoch. The results are acquired by running with the naive roll-out strategy and the Monte-Carlo roll-out strategy, with and without MLE, while optimizing on ROUGE-2 with the argmax baseline. Table 5.14 also shows the time in minutes for the two

strategies, but in this case, the results are for adversarial training (optimizing on the discriminator objective), using the running average baseline and the parameters *n_generator* = 4, *n_discriminator* = 16 and *n_epochs_discriminator* = 4.

Naive roll-out strategy	119 min
Naive roll-out strategy with MLE	176 min
Monte-Carlo roll-out strategy	1359 min
Monte-Carlo roll-out strategy with MLE	1396 min

Table 5.13: Running time for one epoch for the different roll-out strategies, with and without MLE, while optimizing on ROUGE-2

Naive roll-out strategy	184 min
Naive roll-out strategy with MLE	242 min
Monte-Carlo roll-out strategy	1062 min
Monte-Carlo roll-out strategy with MLE	1118 min

Table 5.14: Running time for one epoch for the different roll-out strategies, with and without MLE, while optimizing on the discriminator objective

Chapter 6

Evaluation and Discussion

In this chapter, we will evaluate and discuss the results and observations made in Chapter 5. The chapter will provide insight into thoughts behind dataset preprocessing, evaluation methods and possible shortcomings in our experiments. We will cover topics related to pretraining, different parameters, the two sequence roll-out strategies, different optimization objectives and code optimization, discussing their strengths and weaknesses. The chapter will also provide a discussion on the qualitative differences between the summaries created with the different optimization objectives. Finally, we will answer our research questions, based on our experimental results and what we have discussed.

6.1 Dataset preprocessing

During early experiments, we were doing quite a lot more preprocessing than what is mentioned in Section 5.1. This includes removing unnecessary text, such as *"written by Author ."*, replacing numbers with a special token (#) and removing other special tokens, such as "-" and ":". However, the Pointer-Generator Network [See et al., 2017] seems to generalize better and converge faster when we remove all these "unnecessary" procedures when preprocessing the dataset. The reason seems to be that the Pointer-Generator Network does not really care all too much about the vocabulary and which tokens that are included in it, so trying to optimize the vocabulary by removing words before creating it does not help. It also seems that generalization is better when all the information is kept intact, which means that the model is quite robust to noise. The feature of being general enough to handle a lot of noise in the data is a great property of the Pointer-Generator Network.

At test time, when doing beam-search to generate summaries, we increase the upper limit of maximum tokens from 100 to 120. Table 5.11 show that the number of summaries that exceeds the upper limit of 120 tokens is still 168 and 862 (of 11 000 total) when optimizing on the objective function of ROUGE-1 and ROUGE-2, respectively, when not mixing RL and MLE loss. This leads us to conclude that the increase in upper limit from 100 to 120 is reasonable. Regardless of whether it would have an actual impact on the ROUGE scores, it would have an impact on readability, as even more summaries would have ended abruptly if we did not increase the upper limit.

6.2 ROUGE evaluation metric

The ROUGE calculations we have performed is a possible error source. As seen in Table 5.9, when performing ROUGE calculation on the results of [Liu et al., 2017], we get a slightly different ROUGE-L score, which we cannot explain. However, it should not affect the results when it comes to differences between our own results, as they are calculated using the exact same setup.

Further, we can add that the ROUGE scores from other papers, e.g. as seen in Table 5.9, are not directly comparable to the ROUGE scores reported in our experiments, as the dataset preprocessing and splitting is quite different, given that we preprocess it from scratch using the tokenized version of the dataset. However, we do not consider this a problem, as we are not interested in whether we outperform the current state of the art ROUGE scores, but rather care about the differences when using the wide range of possible parameters, strategies and objectives reported in Chapter 5.

The ROUGE metric by itself is not an optimal measure of how good a summary is. One reason why it is not optimal is the fact that we only compare the generated summary with a single ground truth summary, which does not reflect all the possible *good* summaries that could be created for a particular article. This is not only a problem with the metric itself, as the metric could potentially be used with multiple reference summaries. It does however not currently exist a large enough open and available dataset with multiple reference summaries. Another reason why the metric is not optimal is the fact that it only considers overlapping units such as n-gram, word sequences and word pairs between the reference summary and the generated summary, and does not measure any form of *quality*, which is not easy to define in itself.

6.3 Hyperparameters

We will not go into much details about the hyperparameters for either the generator base model or the discriminator, as they are briefly discussed already in Section 5.4.1 and 5.4.2.

For the generator learning rate, we show the policy log sum because it was the best way to show differences in stability when training with REINFORCE. As can be seen in Figure 5.3, both the learning rates of 0.001 and 0.0001 seems to converge to about the same area. In this case, a higher value is better, as the policy log sum can be thought of as the networks belief state. However, we can see that the learning rate of 0.0001 seems most stable and is the only one that constantly moves in the right direction, hence is the one we use in our experiments.

When comparing the beam search parameters $k1$ and $k2$, there are negligible differences between ROUGE scores when $k2 \geq 6$, however, there is a large difference in running time, which scales linearly with $k2$. The only spike we can note is for ROUGE-2, where $k2 = 12$ is quite a bit higher when $k1 = 4$. We do however not consider it as an option, as the other ROUGE scores are so similar that it is not worth doubling the running time. For the parameter $k1$, there is a huge gap from $k1 = 2$ to $k1 = 3$, but the scores do not improve for $k1 > 3$. Based on these observations, to keep running time low while still getting near optimal ROUGE scores, we use $k1 = 3$ and $k2 = 6$ in our experiments.

6.4 Pretraining

As mentioned in Section 3.2, pretraining the generator is important to allow the reinforcement learning agent to learn. If pretraining is left out, then there are too many states to try, and the problem becomes impossible to solve in a reasonable amount of time. Therefore, we first try to optimize the generator by training with MLE. As seen in Figure 5.9, the training loss keeps decreasing steadily, which may suggest that we should keep training the generator. However, as seen in Figure 5.10, the ROUGE scores spike at epoch 13, and gets quite a bit lower afterwards, suggesting that the model starts overfitting at that point. Overfitting is usually spotted by calculating loss on a validation set, but we did not include this in these experiments as we train using teacher-forcing, which does not give any insight into generalization, given the fact that at test-time, teacher-forcing is not used. Without teacher-forcing, the loss becomes very high, as the MLE loss is calculated as the probability to output the correct tokens (the tokens from the reference summary) at the correct time-steps. Even with a very high loss, the summaries can still be good. It is merely very unlikely to output an exact equal summary as the reference. We can also note that, when reducing the learning rate and increasing the batch size, while only training with MLE, as seen in Figure 5.25, we still do not get higher ROUGE scores than at epoch 13 during pretraining, suggesting that we need some alternative way of training, which is where reinforcement learning, and adversarial training comes in.

It is similarly important to pretrain the discriminator, which becomes clear when looking at Figures 5.21 and 5.22. Here, we can see huge differences when doing adversarial training with a sufficiently pretrained discriminator, and training with a discriminator without sufficient pretraining. For the discriminator without sufficient pretraining, the generator will not even learn to get better than the original pretrained model, whereas when training with the discriminator with sufficient pretraining, the generator seems to keep learning, even after 5 epochs. We can also look at the average reward in Figure 5.23. Here, we can see that the average reward is quite stable (around 0) for the sufficiently pretrained discriminator, while the average reward keeps decreasing when training without a sufficiently pretrained discriminator.

For the sufficiently pretrained discriminator we got precision, recall, F1 score and accuracy at $\sim 80\%$, compared to the discriminator without sufficient pretraining, where precision, recall, F1 score and accuracy is measured at $\sim 60\%$. The number of training examples are 50 times higher for the sufficiently pretrained discriminator, and it is trained for 5 epochs instead of 1. At this point, it seems hard to generalize across the sampled data very well, which is why, even with such high amounts of data, we still do not get closer to 100%. One interesting direction here is to try increasing the log-probability of the model (the belief state), before pretraining the discriminator. By doing this, we would decrease the variance in the sampled data, making it easier for the discriminator to discriminate between the real and the fake data, and thus, give better feedback to the generator. One way to achieve this could be to train for a few thousand iterations using the ROUGE objective before pretraining the discriminator and doing adversarial training.

We should also consider the fact that a convolution based classifier may not be the optimal discriminator architecture. One of the reasons why seq2seq models and RNNs in general perform well in the context of abstractive text summarization and other sequential tasks is the fact that they take both short-term and long-term dependencies into account. It would be interesting to see how a LSTM based or a hybrid RNN/CNN architecture would perform on the task of discriminating between real and fake summaries. If the performance is better than the purely CNN based model, then we can expect even further improvements to the generative model when training with the GAN framework.

6.5 Reinforcement learning parameters and strategies

In Section 4.2, we mention the importance of a baseline to reduce the variance of the reward signal during training. Figures 5.11 and 5.12 shows a clear difference in ROUGE scores when training with and without a baseline. After epoch 5, training without a baseline has reached almost 0.40 ROUGE-1 score, while training with the argmax baseline is closer to 0.41, which is a significant difference. When looking at ROUGE-L (Figure 5.12), We can note that the scores are generally worse when truncating negative rewards (only keeping rewards > 0). The difference is not as large for ROUGE-1. We can also note that the argmax baseline performs slightly better than the running average baseline when optimizing on ROUGE.

As observed in Figure 5.12, truncating negative rewards after subtracting the baseline from the reward gives a slightly worse ROUGE-L score. However, it might be beneficial to do this in some cases. We believe that if the reward signal is unstable, e.g. where a lot of rewards for seemingly good sequences become negative with large magnitude, the model will try to not output these sequences anymore. What we rather would want in this case, is to only optimize towards what we know is better (i.e. high rewards), while not necessarily learning to optimize away from the sequences with low rewards. This is similar to saying that we are uncertain in the quality of the reward evaluation, where we trust the high rewards, but not necessarily the low rewards.

In our final experiments, where we evaluate the different objectives using optimal parameter settings, we use the argmax baseline when optimizing for ROUGE, while we use the running average baseline for adversarial training, optimizing towards the discriminator objective. The reason we do this is that the discriminator is pretrained using sampled data (sampling from the softmax distribution), which has a higher variance than data generated by greedy search (argmax). This creates a large gap between a reward given to a randomly sampled sequence and a reward given to a sequence created by a greedy search. Since we randomly sample the sequences during training, the argmax baseline would be quite unstable when optimizing towards the discriminator objective. The running average baseline does not have this problem and is thus a logical choice. We can also note that the running average baseline is faster to compute, since we only accumulate the value after each iteration, without extra computation, whereas the argmax baseline needs to compute the greedy search for every iteration, which can be quite expensive, depending on the implementation.

Training using the Monte-Carlo roll-out strategy is very time-consuming compared to the naive roll-out strategy, as can be seen in Tables 5.13 and 5.14. The most impactful parameter when it comes to training time is the number of roll-outs (N). From Figure 5.13, we can see that the training time is approximately linear in N. However, looking at Figure 5.14, it is not clear whether there is much improvement when increasing N. We can note that $N = 1$ seems to be slightly less stable compared to $N > 1$, but overall the difference seems to be quite low. Generally, by averaging the reward across multiple roll-outs, we expect much better estimates for a high N compared to a low N, but this is hard to verify with tests, since the nature of random sampling would require us to perform many similar experiments to get a better grasp of the effect, which becomes infeasible due to lack of resources (time and hardware) to perform such extensive testing.

Figure 5.15 and 5.16 shows the difference in ROUGE-1 and ROUGE-L scores, respectively, when sampling and using argmax to accumulate the previous generated sequence. In both figures there is a very noticeable difference between the two strategies. For ROUGE-L, sampling achieves a score of 0.363, while argmax peaks at 0.359. The trend is similar both with and without MLE. Experimentally, sampling seems to be the better option among the two.

During adversarial training, there is a trade-off between training the generator and training the discriminator. Balancing the training between the two parts is often tricky to do properly. We conduct a series of tests, as seen in Figure 5.17 and 5.18, where we vary the parameters $n\_generator$, $n\_discriminator$ and $n\_epochs\_discriminator$. We can note that the model converges faster when setting $n\_discriminator > n\_generator$. We can also note that there is not much difference between setting a higher value for $n\_discriminator$ and setting a higher value for $n\_epochs\_discriminator$. There is however a large difference in time when increasing $n\_discriminator$, as we need to create more training data for the discriminator, as seen in Figure 5.19. To keep the runtime low, while allowing the discriminator to keep up with the generator, we found the parameters $n\_generator = 4$, $n\_discriminator = 16$ and $n\_epochs\_discriminator = 4$ to perform well. The reason why we set $n\_generator > 1$ is to allow the generator time to fool the discriminator before we re-train the discriminator. Looking at Figure 5.20, we can see quite clearly why it is important to re-train the discriminator during adversarial training. When we only train the generator, the generator starts to fool the discriminator with bad (mostly random) data, significantly reducing the ROUGE score.

According to [Paulus et al., 2017], using a mixed objective of MLE and RL outperforms using a pure MLE or a pure RL objective. In Section 5.8 and 5.9, we can see that the summaries created by a model trained with MLE and the summaries created by a model trained without MLE are quite different, both in length and in language. Combining the objectives makes it possible to optimize towards an objective, such as ROUGE, while still maintaining a good language model. We explore different values for β, as seen in Figure 5.24. Here, we can note that a higher value converges towards the objective a lot quicker. A lower value of β, i.e. increasing the importance of MLE when calculating the gradient, will keep the model more in place. We found $\beta = 0.9984$ to work well, which is the same value used by [Paulus et al., 2017]. With a high value for β, the model is allowed enough room to learn the features of the objective function, while still maintaining important language features.

Figure 5.26 and 5.27 shows the ROUGE-1 scores when varying the λ parameter, while optimizing towards the combined objective of ROUGE-1/discriminator and ROUGE-2/discriminator, respectively. As expected, the ROUGE score becomes higher when more weight is used for the ROUGE objective (higher λ). We can also see that no matter the value of λ, the general trend is that the models keep on getting better (increasing ROUGE score) throughout the 5 epochs they are trained, which suggest that the model has no problem learning while combining the objectives. Moreover, there are only minor differences between using ROUGE-1 or ROUGE-2 together with the discriminator. In our final results, we use the ROUGE-2 objective together with the discriminator, using a λ value of 0.5. Through experiments, we found that the mean reward (before subtracting the baseline) at the first 200 iterations for the discriminator is about 0.15, regardless of parameter settings (as long as we use a sufficiently pretrained discriminator). This is much closer to the average reward we found for ROUGE-2 at the first 200 iterations (0.08), compared to the average reward for ROUGE-1 (0.29). In the discriminator case, the reward will be kept quite steady, as we keep training the discriminator, while the reward when training with the ROUGE objectives should be increasing. We found the average discriminator reward to be at about 0.14 after 1 epoch, while ROUGE-2 increases to 0.11, and ROUGE-1 increases to 0.31. After 10 epochs, the average discriminator reward is at about 0.12, similar to ROUGE-2 which also is at 0.12, while ROUGE-1 is at about 0.32. Combining ROUGE-2 and the discriminator is much simpler to scale, compared to ROUGE-1. The reward from the discriminator objective and the reward from the ROUGE-2 objective will become approximately equal in magnitude after a few epochs of training, which is a simpler case to study. In this regard, a λ value of 0.5 makes sense.

6.6 Evaluation of quantitative results

In Table 5.5, 5.6 and 5.7 we can see the ROUGE scores for the best models trained with three different objectives. The tables clearly show that the discriminator objective scores are quite low compared to the ROUGE objectives scores. This is not that strange since we measure their performance in ROUGE scores, thus one should assume that optimizing towards ROUGE objectives would achieve higher scores. We can also note that optimizing towards ROUGE-2 seems to yield the best overall ROUGE scores. It even gets a higher ROUGE-1 score than what optimizing towards ROUGE-1 does.

We also have scores for the different objectives with both naive roll-out strategy and the Monte-Carlo roll-out strategy. The scores are generally a little lower for the Monte-Carlo roll-out strategy. For instance, when optimizing on ROUGE-2, the naive roll-out strategy achieves a ROUGE-L score of 0.3673 and the Monte-Carlo roll-out strategy achieves a score of 0.3616. One thing to note here is that the model trained with the naive roll-out strategy has trained 3 times as much as the model trained with the Monte-Carlo roll-out strategy, which indicates that Monte-Carlo roll-out strategy requires fewer updates to the model. However, looking at Table 5.13 and 5.14 there is a large difference in running time between those two methods. The fact that the Monte-Carlo roll-out strategy is about 5 to 12 times slower than the naive roll-out strategy heavily outweighs that it require fewer updates to the model.

Table 5.5, 5.6 and 5.7 also include results with and without MLE. As mentioned in Section 3.2, the MLE loss provides a balancing factor that allows the model to continue to produce readable sentences. [Paulus et al., 2017] showed that their pure RL model got higher ROUGE scores than their model trained with the mixed RL and MLE objective, but the mixed objective outperformed the pure RL model on human evaluation. The results in Table 5.5, 5.6 and 5.7 backs up their findings. For all the different objectives, the models trained without MLE achieves higher scores, except for the ROUGE-2 objective trained with the Monte-Carlo roll-out Strategy. The quality of the produced summaries is discussed more in Section 6.7.

Table 5.8 shows ROUGE scores when optimized towards the combined objective of ROUGE-2 and the discriminator. Compared to the other objectives, we can see that it achieves better ROUGE scores than the pure discriminator objective, but worse scores than the models optimized on pure ROUGE-1 and pure ROUGE-2. This is as expected when setting $\lambda = 0.5$ in Equation 4.2.

Table 5.9 summarize all our best performing models, our pretrained model and the state of the art. All our models trained with RL and GAN get improvements over our pretrained model. The models optimizing on ROUGE-1 and ROUGE-2 gets about 0.02 increase in ROUGE-1 and ROUGE-L scores and about 0.01 increase in ROUGE-2, which is a significant increase. The discriminator only achieves a minor increase, but the combined model achieves about 0.005 increase in ROUGE-1, 0.003 increase in ROUGE-2 and 0.002 increase in ROUGE-L. All our models are achieving comparable results with the state of the art. We present models that achieves both higher and lower ROUGE scores than the state of the art. However, this does not mean that our models are better or worse than the current state of the art. It merely shows that the ROUGE scores are comparable.

As already mentioned, the MLE objective provides a balancing factor that allows the model to continue to produce readable sentences and outperform pure RL models. This is also clearly illustrated in Table 5.10 where we see the average length of summaries. All the models trained with different objectives have longer summaries without MLE. For instance, the model trained with the ROUGE-2 objective have an average summary length of 85.1 without MLE and 66.1 with

MLE. Table 5.11 also gives a good illustration of the effect of using MLE. Here, we can see the number of summaries that reach maximum length for our pretrained model and our best models trained with the different objectives. With the numbers from Table 5.10 in mind, we would expect that very few of the summaries generated by the pretrained model and the models trained with the discriminator objective and the combined objective would reach the maximum length. We expect this because of the fact that they are very close to the average length of ground truth and is more likely to be below the maximum length. As we can see, this is also the case. There is only one summary for the pretrained model that reaches the maximum length. The models optimizing on the ROUGE objectives have a lot more summaries that reaches the maximum length. This is also as expected, since they try to optimize on matching n-grams. Longer summaries are more likely to have more n-grams that are in the ground truth, and therefore get a higher reward. Without MLE, the ROUGE-1 model generates 168 summaries that reaches the maximum length, while only 29 of the summaries reach maximum length when training with MLE. The model optimized on the ROUGE-2 objective generate 862 summaries that reach maximum length without MLE, while only 37 of the summaries reach maximum length when training with MLE. This shows how the MLE objective allows the model to optimize towards a different objective, while keeping the language model intact.

Figure 5.28, 5.29, 5.30 and 5.31 show the percentage of novel 2-grams, 3-grams, 4-grams and sentences, respectively. From the figures, we can see that the 2-grams gets the lowest scores, even for the ground truth which is human made. The novelty of sentences is much higher. This makes sense, as you only have to change one word in the sentence for the sentence to be qualified as novel. We can see that our models are quite far away from the ground truth in novelty, but this is as expected since we use the Pointer-Generator Network [See et al., 2017]. The most significant property of the Pointer-Generator Network is that it can copy parts of the input, thus achieving less novelty (this can be considered a trade-off between novelty and creating non-factual sentences). Another thing to note with these figures is that our pretrained model achieves higher novelty than all our other models. This suggest that our reinforcement learning agents gets a higher reward when copying from the input, and therefore ends up generating less novel summaries. Once more we can see that using MLE is outperforming the pure RL models. Here, we can see that all our models get better novelty by using MLE, which means that there is not as much copying from the article when the MLE objective is included.

6.7 Evaluation of qualitative results

Figure 5.32 shows the difference between training with MLE and without MLE for a generated summary when optimized on the ROUGE-2 objective. In this case, both summaries are identical up until the last part. For the identical part they summarize the article quite well, leaving out uninteresting parts (e.g. who the author of the article is). The model trained without MLE adds an extra sentence in the end. This alone might not be a bad thing, but this extra sentence is repeating what has already been said, have bad grammar and is incorrect, given the content of the article. Despite the fact that the last sentence is non-factual with bad grammar, the reward will get higher when optimizing towards ROUGE-2 because of the extra sentence. I.e. a longer summary will be more likely to have more 2-gram matches with the ground truth. However, when including MLE, this behavior will get punished, and the generated summaries will be objectively better.

Generally, it is quite difficult to say something qualitatively about the differences between the objectives. The most noticeable thing is that the summaries in general are quite good, and quite frequently, the summaries created by the models optimized on the four different objective

functions include more useful information than the summaries created by the pretrained model. In most cases, the models generate long and meaningful summaries. Another noticeable thing is that the summaries created by the models trained with the four different objectives are quite similar, especially in the first couple of sentences. The largest difference is, in most cases, towards the end of the summaries. Here, the ROUGE-1 objective and the ROUGE-2 objective seems to exhibit similar traits. Correspondingly, the discriminator objective and the combined objective also exhibit a lot of similarities.

More specifically, we can look at the summaries in Figure 5.33. First thing we can note here is that none of these summaries are exactly equal. However, as seen in Appendix A, this is not always the case. There are a quite a few of the summaries that are exactly equal (e.g. summaries created by models trained with the ROUGE-1 objective and the ROUGE-2 objective). Another thing to note is that all the generated summaries are longer than the ground truth, which in this case is quite useful. The generated summaries often add extra content that is more informative and gives us a better picture of the whole article.

If we look at the content of the summaries generated by the pretrained model and the model trained with the ROUGE-1 objective, as seen in Figure 5.33, we can see that both summaries present non-factual information. For convenience we show the non-factual part, as well as the correct phrasing from the article in Figure 6.1. The sentence that is created by our models is syntactically good, but the first half of the sentence is incorrect. Here, the sentence have a reference to the wrong speaker and also have two audiences, where the ”reporters” are not supposed to be included. The model optimized on ROUGE-2 solves this issue by simply not including this particular piece of information. The summaries created by the model optimized on the discriminator objective and the combined objective includes the same information as displayed in the non-factual part, but they include it successfully with the correct content. This is a typical example where we see that the discriminator objectives focus on more realistic and factual content, while the ROUGE objective is more prone to error because of the nature of the ROUGE calculation.

Non-factual: *bayern coach pep guardiola told reporters on twitter the player sustained ' a medial collateral ligament injury '*

Factual: *the club later said on twitter the player sustained ' a medial collateral ligament injury '*

Figure 6.1: The non-factual part of the summaries generated by the pretrained model and the model optimized on the ROUGE-1 objective (taken from Figure 5.33). The correct phrasing (factual) from the article is also included.

Comparing the ROUGE objectives (ROUGE-1 and ROUGE-2) and the discriminator objectives (discriminator and combined discriminator/ROUGE2), we can consider the length of the generated summaries as the most significant difference, as seen in Table 5.10. When looking at the summaries in Figure 5.33 and Appendix A, we can note a couple of things. The extra length provided by the ROUGE objectives often includes non-factual content, while the discriminator objectives often manages to summarize the important content in a more concise form. Another difference to note is that the ROUGE objectives have a higher percentage of novel n-grams compared to the discriminator objectives, as seen in Figure 5.28, 5.29 and 5.30. This is consistent with the fact that the ROUGE objectives generate more non-factual content, which in turn will lead to more novel summaries. The downside of this is that it will lead to incorrect information in some cases.

6.8 Code optimization

Table 5.12 shows the difference in runtime between naive masking (Code 4.4) and vectorized masking (Code 4.5). We can see that the vectorized masking is an order of magnitude faster to compute compared to naive masking. The most important difference between the algorithms is that for vectorized masking, we remove the need to transfer data between GPU and CPU, which is very time-consuming. Using vectorization to solve such trivial computation problems also removes the need to write custom CUDA kernels, which is a lot more elaborate.

The most significant part of the time used to train using the Monte-Carlo roll-out strategy is spent doing roll-outs. In Section 4.5 we proposed a batch-parallel algorithm (Code 4.7) to speed up the time to run N roll-outs. Figure 5.34 shows the runtime difference between naive roll-out (Code 4.6) and batch-parallel roll-out when varying the batch size. Here, we can see that for batch sizes < 30, the batch-parallel roll-out is 3 to 5 times faster to compute compared to the naive roll-out. For batch sizes ≥ 30, the difference is not as large, but still significant (between 1.5 and 2.5 times faster to compute). The naive implementation has a large gap from batch size = 1 to batch size = 10, but the runtime does not increase so much after that point. The batch-parallel implementation, on the other hand, scales approximately linearly with the batch size.

Even though we do gain a large speedup by using the parallel implementation, it is still far from optimal. Ideally, the computation would not increase so much when increasing batch size. The reason why we do not see a perfect scaling solution is because the GPU usage is already maxed out at lower batch sizes in the parallel case. We do not see this problem in the naive case, because the GPU can handle a single batch of 50 elements without a problem. Our solution would most likely have better scaling on smaller problems (smaller matrices / simpler models). However, the speedup we gain is still significant, even on such a large problem.

We tried implementing the multi-process parallel setup suggested by [Clemente et al., 2017]. However, running on a single node (server) with limited resources proved problematic. We could only run a few processes in parallel, which gave us a minor speedup, but not as large speedup as the batch-parallel implementation. When launching processes which are to run code on GPU, the processes set aside a large amount of virtual memory (between 15 and 25 GB in our case). The server we run our code on have 128 GB disk space, restricting us to run only a few processes simultaneously. The GPU is also a bottleneck in this case, similar to the batch-parallel implementation. One of the upsides of the multi-process parallel setup is that it has potential for much higher speedups, since it can potentially be implemented across multiple nodes. This is however a non-trivial problem, since it requires communication across nodes (MPI[1] is a possible choice for implementation).

Figure 5.35 shows the runtime when varying the maximum number of decoding time-steps. Naturally, the runtime scales approximately linearly with the number of time-steps. When choosing a value for the maximum number of decoding time-steps, we need to consider the length of the summaries. We want to lower the runtime, without losing too much information. Cases where the model outputs <PAD> do not gain any information and can be skipped. If we expect a lot of the generated summaries to end (output <EOS>) before reaching max length, then we can reduce the maximum number of decoding steps without worrying too much about losing information. Looking at the length of the reference summaries (Figure 5.1) should give us a good indication, since this is essentially what we try to mimic. In our work, we set the maximum number of decoding steps to the maximum length of a single summary in the current reference

[1] Open MPI (`https://www.open-mpi.org/`) is a popular implementation of the Message Passing Interface.

batch. We did not conduct further experiments to see how much information would be lost if we lowered the maximum to a static value < 100. Our approach is a trade-off between saving time and keeping information. Our assumption is that, if the reference summary is short for a particular article, we can expect our generator to also output a short summary in most cases.

As mentioned in Section 4.5, reducing the sample rate (how often we perform Monte-Carlo roll-out) could be beneficial. The sample rate has a large impact on time, as the roll-out is the largest time-consuming process. As seen in Figure 5.36, the runtime scales approximately linearly with the sample rate. For instance, performing one epoch with a sample rate of 0.4 takes 1336 minutes, while a sample rate of 0.8 takes 2563 minutes. Figure 5.37 shows the difference in ROUGE-1 scores when varying the sample rate. Here, we can see a gap between sample rate $= 0.1$ and the other sample rates, but the difference is marginal when the sample rate is ≥ 0.2. Based on these observations, we chose to use a sample rate of 0.2 in our experiments, which still has good performance, while reducing the training time drastically. In general, a higher sample rate would give a better gradient estimation, and would always be preferable in the long run, but because of the long training time, we could not perform extensive experiments with it.

As previously mentioned, if the model outputs <EOS>, we will not gain any information for the following outputs. One way we can deal with this is to check if the entire batch has reached <EOS>, as proposed in Section 4.5. Figure 5.38 shows the runtime difference when varying the time-step to start checking for <EOS>, both when using sampling and argmax for accumulating the previous sequence. Here, we can notice that we will save time no matter when we start checking (except for a small discrepancy at time-step = 90 for sampling), and earlier is better than later. However, there is a large difference between sampling and argmax for time-step < 80. Because of the fact that this check will only stop the decoding procedure in the case where every element in the batch has reached <EOS>, it is only natural that sampling will not pass the check as often, as it is less likely to output <EOS>, compared to the top-1 (argmax) token. When the time-step to start checking is < 80, the runtime for argmax keeps decreasing, while the runtime for sampling is quite steady.

6.9 Answers to research questions

In this section, we will attempt to answer the research questions presented in Section 1.2.

6.9.1 Research question 1

> *How can reinforcement learning and generative adversarial networks for improving an abstractive text summarization model be implemented in an efficient manner, and what are the important factors that impact running-time?*

In our experiments, we looked at models trained with both the Monte-Carlo roll-out strategy and the naive roll-out strategy. From the tables presented in Section 5.8, we can see that generally, there are only minor differences in ROUGE scores between the two strategies. Running-time, on the other hand, is very different between the two strategies. Table 5.13 and 5.14 show that the Monte-Carlo roll-out strategy is about 5 to 12 times slower to compute for one epoch. From this, we can only conclude that ”simple” objective functions do not require a complicated approach, and the naive roll-out strategy should be sufficient. However, for more complex and harder-to-optimize objectives, we cannot say for sure that the naive roll-out strategy will be enough. For instance,

when optimizing on the discriminator objective with MLE, we got higher ROUGE scores for the Monte-Carlo roll-out strategy.

More specifically, we saw that there were large differences in running-time when it came to parameter choices. When training with adversarial training, we saw from Figure 5.18 and 5.19 that $n\_discriminator$ had a significant impact on running-time, and as an alternative to increasing this value, we could increase $n\_epochs\_discriminator$ to gain the same results without increasing running-time as much. When training with the Monte-Carlo roll-out strategy, the number of roll-outs (N), had the largest impact on running-time. As discussed in Section 6.5, our results are inconclusive in regard to this parameter. However, even though we do believe that a higher N will give better results in most cases, the significant increase in running-time might not be worth it.

When it comes to code optimization, the largest factor that impacts running-time is transfer of data between GPU and CPU. A remarkable example of this is seen in Table 5.12, where the vectorized masking operation is an order of magnitude faster to compute compared to the naive masking operation. In Section 4.5, we proposed an algorithm for batch-parallel roll-out (Code 4.7). As seen in Figure 5.34 and discussed in Section 6.8, the speedup gained from the parallelized roll-out is quite significant. We can conclude that a parallel roll-out implementation can help make the Monte-Carlo roll-out feasible, and it is certainly possible to gain even larger speedups than seen in Figure 5.34. One way to achieve this is to implement a parallel roll-out across multiple nodes. This idea is elaborated further in Section 7.2.1.

In Section 5.10, we looked at the running-time when varying the number of decoding time-steps (Figure 5.35). Naturally, this has a large impact on running-time, but as discussed in Section 6.8, we did not conduct further experiments to see how much information would be lost if we lowered the maximum to a static value < 100, so we cannot conclude how much information would be lost in this case, and whether it would make an impact on the generated summaries. We also looked at reducing the sample rate (Figure 5.36 and 5.37), which had a quite significant impact on running-time. As discussed in Section 6.8, the sample rates ≥ 0.2 seem to perform quite similarly, while the running-time difference is huge. This makes a lower sample rate attractive to train more efficiently. A final code optimization trick we used was to stop sampling if the entire batch reached <EOS>, but as seen in Figure 5.38, it only had a significant impact when accumulating the argmax token as the previous sequence, while we found sampling to be superior for convergence. However, the trick is simple and should be considered worth implementing, even for a small speedup.

In conclusion, the code optimizations presented in this thesis makes the Monte-Carlo roll-out strategy feasible. However, in most cases, the naive roll-out strategy should be sufficient, while being a lot more efficient.

6.9.2 Research question 2

> *Which parameters, in the context of improving an abstractive text summarization model using reinforcement learning and generative adversarial networks, has a significant impact in regard to training stability and convergence?*

Perhaps the most obvious parameter that has a significant impact in regard to training stability and convergence is the learning rate. As discussed in Section 6.3, we found both 0.001 and 0.0001 to work quite well, while a higher or a lower learning rate does not even start to converge. Another set of parameter choices that has a large impact on the stability of the training is the difference in number of iterations for the generator ($n\_generator$) and the number of iterations

for the discriminator ($n\_discriminator$ and $n\_epochs\_discriminator$). Here, we saw that the discriminator required quite a few more iterations than the generator to stabilize the training. For instance, there was a noticeable gap between a 1:1 ratio between the parameters and a 1:4 ratio. In our experiments, we found the parameters $n\_generator = 4$, $n\_discriminator = 16$ and $n\_epochs\_discriminator = 4$ to give good results, while not being too expensive in terms of running-time. These parameters yield a 1:16 ratio between the number of iterations for the generator and the discriminator.

While not strictly being a parameter, the choice of baseline had a large impact on convergence. As seen in Figure 5.11 and 5.12, there is a significant difference between using a baseline and not using a baseline. Further, as discussed in Section 6.5, we found the argmax baseline to work best with the ROUGE-1 and the ROUGE-2 objectives, while the running average baseline worked best with the discriminator objective and the combined objective, suggesting that the optimal baseline is problem dependent. Moreover, in our experiments, we found that allowing negative rewards outperformed truncating rewards at 0 (keeping only positive rewards).

Finally, the β parameter, which is used to scale the magnitude between the policy loss and the MLE loss is worth mentioning as an important factor. When optimizing on ROUGE-1, we saw that there was a large difference in the obtained ROUGE scores when varying the β parameter between 0.9 and 1.0. At 0.9, the ROUGE score gets almost no increase compared to the pretrained model, while at 1.0 and close to 1.0 values achieves a significant increase in ROUGE score. As discussed in Section 6.5, even though $\beta = 1.0$ achieves the best ROUGE scores, a lower value is necessary to keep the language model intact. In our experiments, a value of 0.9984 seemed to work well.

6.9.3 Research question 3

> *How does different optimization objectives, as well as the combination of generative adversarial networks optimization and a static reward function impact the training convergence and the generation of summaries?*

There is a large difference in training convergence between the different objectives. Both the objectives of ROUGE-1 and ROUGE-2 are quite easy to optimize, and we have seen that the model can learn the features of the objective with far from optimal parameter choices. The discriminator, on the other hand, is way more elaborate to deal with. Here, we have seen that discriminator pretraining, the choice of baseline and the ratio of iterations between the generator and the discriminator are all key factors to make convergence possible. The combined objective can be seen as a middle-ground between the ROUGE objectives and the discriminator, where a larger value for the λ parameter is easier to optimize.

Generally, the summaries created by the models trained with the four different objectives are quite similar, especially in the first couple of sentences. The largest difference is, in most cases, towards the end of the summaries, where the ROUGE-1 objective and the ROUGE-2 objective seems to exhibit similar traits. Correspondingly, the discriminator objective and the combined objective also show a lot of similarities.

The largest difference between the generated summaries is the difference in length. When including MLE in the objective function, we still get a large difference between the objectives. The average number of tokens in the summaries created by the models trained with the ROUGE-1 objective, the ROUGE-2 objective, the discriminator objective and the combined objective is 66.4, 66.1, 49.5 and 49.0, respectively, which are quite significant differences. We can also note

that the models trained with the ROUGE objectives are more abstract than the models trained by adversarial training. This makes sense considering that the ROUGE objectives only care about n-gram matches, leading to the generation of longer sequences of text that may include incorrect information.

Even though it is easier to optimize a model towards the ROUGE objectives, we still see a few problems with the generated summaries, such as repeating bulks of information and non-factual sentences. In contrast, the models trained with the discriminator objective and the combined objective often manage to summarize the important content in a more concise form.

6.9.4 Research question 4

> *How does discriminator pretraining affect the convergence during adversarial training?*

As discussed in Section 6.4, discriminator pretraining has a large impact on convergence during adversarial training. When training with a discriminator without sufficient pretraining ($\sim 60\%$ accuracy), we saw that the model dropped rapidly in performance in terms of ROUGE scores after one epoch, compared to the pretrained model. When training with a sufficiently pretrained discriminator ($\sim 80\%$ accuracy), the model seemed to possess much better learning capabilities, converging steadily towards higher ROUGE scores. This leads us to conclude that properly pretraining the discriminator is key for adversarial training to work. We also believe that, if we had been able to achieve even better results for the discriminator (better accuracy, precision, recall and F1 score after pretraining), we would achieve even better results from the models optimized on the discriminator. We elaborate this further in Section 7.2.2.

Chapter 7

Conclusion and Future Work

In this chapter, we will conclude our research. We do so by summarizing our work, elaborating on our main findings and revisiting our goal. Finally, we will discuss some ideas for future work.

7.1 Conclusion

In this thesis, we have explored how to improve an abstractive text summarization model by employing reinforcement learning and generative adversarial networks. As a base model, we implemented a variation of the Pointer-Generator Network [See et al., 2017]. We pretrained the base model using standard MLE, and utilized the REINFORCE algorithm as our choice of policy gradient method for RL and GAN training. We explore four different objective functions in total; ROUGE-1, ROUGE-2, discriminator (adversarial training), as well as a combined model of ROUGE-2 and the discriminator. To calculate a reward for these objective functions during training we require a way to roll-out a complete sequence from a partial one. In this regard we explore two different strategies, which we named the Monte-Carlo roll-out strategy (Code 4.2), and the naive roll-out strategy (Code 4.1).

There are a lot of implementation details and parameter choices that are important for training stability and convergence, which are mostly left out of research papers about the topic. We provide an extensive study of these details to make our work easily reproducible and welcome for further studies. Another problem with training of abstractive text summarization models is that it is generally very time-consuming. The problem with long running-times gets even worse when doing RL and GAN training. To cope with some of these problems, we propose code optimization techniques that will help speed up the runningtime during training.

In regard to running-time, we found that transfer of data between GPU and CPU have the largest impact. Here, vectorization techniques can be utilized to remove the need for data transfer. We found our proposed solution for vectorized masking (Code 4.5) to be an order of magnitude faster to compute compared to the standard masking operation computed on the CPU. Another time-consuming task is the roll-out when employing the Monte-Carlo roll-out strategy. Here, we propose a batch-parallel implementation (Code 4.7) that achieves quite significant speedup compared to a naive serial implementation.

Comparing the two roll-out strategies, we found that there were only minor differences in regard to convergence and generated summaries. However, the Monte-Carlo roll-out strategy is about 5 to 12 times slower to compute compared to the naive roll-out strategy for one single epoch. From

this we conclude that, in most cases, the naive roll-out strategy should be sufficient, while being a lot more efficient.

In regard to implementation details and parameter choices that matters for training stability and convergence, we found that the choice of baseline, the ratio between generator training and discriminator training, as well as the β parameter (scaling between mixed RL and MLE training) to have the most significant impact. More specifically, we found that the running-average baseline works well in most cases, while the optimal baseline is problem dependent. When it comes to the ratio between generator training and discriminator training, we found that the number of iterations for the discriminator needs to be quite a lot larger than the number of iterations for the generator. In our experiments we found a 1:16 ratio between generator and discriminator iterations to work well. Finally, we found that we required a high value for the β parameter to be able to optimize properly. Even when setting $\beta = 0.9984$, we still see significant difference in the generated summaries, showing the importance of the MLE objective.

We found a large difference in training convergence between the different objectives. Both the objectives of ROUGE-1 and ROUGE-2 were quite easy to optimize, while the discriminator, on the other hand, was more elaborate to deal with. Discriminator pretraining, the choice of baseline and the ratio of iterations between the generator and the discriminator were all key factors to make convergence possible for adversarial training. Generally, the summaries created by the models trained with the four different objectives were quite similar, especially in the first couple of sentences. The largest difference between the generated summaries was the difference in length, where the average number of tokens in the summaries created by the models trained with the ROUGE-1 objective, the ROUGE-2 objective, the discriminator objective and the combined objective was 66.4, 66.1, 49.5 and 49.0, respectively. Even though we found it easier to optimize a model towards the ROUGE objectives, we still see a few problems with the generated summaries, such as repeating information and non-factual sentences. Models trained with the discriminator objective and the combined objective, on the other hand, often manage to summarize the important content in a more concise form.

Regarding the goal of *improving the generation of summaries by using reinforcement learning and generative adversarial networks*, we found that this is certainly feasible. We show improvements from the base model in regard to ROUGE scores using all four objective functions, and show differences in the generated summaries. As already mentioned; there are multiple implementation details and parameter choices that play a large role to be able to actually improve the model. We believe that even better results can be achieved by further improving the discriminator and possibly looking at other objective functions.

7.2 Future work

In this section, we will provide insight into interesting aspects, directions and approaches to extend our work.

7.2.1 Parallelizing across multiple nodes

In Section 4.5, we mentioned the possibility to implement the Monte-Carlo roll-out strategy with multiple workers (processes), similar to how [Clemente et al., 2017] distribute the workload to multiple workers. We tried implementing this approach, but since we had limited resources on the node, this proved to be problematic. We managed to run a few workers simultaneously, but this

only gave us minor speedups compared to the batch-parallel approach. We believe that one could get a significant speedup by combining these two approaches, where one run multiple workers in parallel across nodes. This would effectively give each worker their own GPU, and if the workers also use the batch-parallel algorithm, we could in theory increase the number of roll-outs proportionally with the number of workers without noteworthy increase in running-time. The main problem with this approach is that it requires communication between the nodes, which requires a more elaborate implementation.

7.2.2 Improving belief-state before discriminator pretraining

As mentioned in Section 6.4, it would be interesting to increase the log-probability (the belief-state) of the model before pretraining the discriminator and doing adversarial training. By e.g. training for a few thousand iterations using the ROUGE objective (or any objective that is fairly simple to optimize on, and thus increase the log-probability), we expect the variance of the sampled data to decrease enough to make it possible to reach even higher accuracy ($> 80\%$) for the discriminator after pretraining, thus giving better feedback to the generator and most likely increase the speed of convergence. In Section 6.4 we discussed the huge difference between optimizing with a poorly pretrained discriminator ($\sim 60\%$ accuracy) and optimizing with an improved discriminator ($\sim 80\%$ accuracy), which leads us to believe that we would get even better results in terms of ROUGE scores and generated summaries by further increasing the performance of the discriminator.

7.2.3 Alternative dataset

The CNN/Daily Mail dataset has been used extensively in recent studies, but it is relatively small compared to a few other datasets. In most deep learning cases, including more data would increase the generalization. Therefore, it would be intriguing to see the differences between the performance on the CNN/Daily Mail dataset and another dataset with our RL and GAN setup.

The Cornell Newsroom Summarization dataset [Grusky et al., 2018] is a large dataset with 1.3 million articles and summaries written by authors and editors in the newsrooms of 38 major publications from 1998 to 2017. This dataset is 4 times as large as the preprocessed version of the CNN/Daily Mail dataset we use, and therefore it is a good suggestion to explore, both to see if the generalization gets better and to compare qualitative results. The authors also include state of the art result for the dataset posted on their website[1], which is convenient for comparison.

Another interesting idea is to perform a cross-validation test, where we look into how well our model trained on the CNN/Daily Mail dataset perform on test data from another corpus. This would give us more insight into how well our model is generalizing and how useful it is for a real-world scenario.

7.2.4 Headline generation

It would be quite interesting to see how well the Pointer-Generator Network trained with RL or GAN would do on the task of headline generation. Essentially, generating headlines is the same problem as generating summaries, except that the sequences are a lot shorter. Summaries are

[1] Cornell Newsroom: `https://summari.es/`

usually tightly-coupled with the content of the article, while headlines, on the other hand, can be very abstract and can seem disconnected from the article in a lot of cases. Using GANs, especially, would be an interesting approach to bring more variation to the generation.

7.2.5 Novel n-grams objective

As discussed in Section 6.6, all the objectives explored in this thesis seems to prefer copying to increase the reward, which yields a reduced percentage of novel n-grams in the generated summaries. An interesting way to try increase the abstractness of the generated summaries, could be to incorporate the percentage of novel n-grams as an objective function. A basic implementation of this is to give a reward as the percentage of novel n-grams (e.g. 4-grams) in the generated summary. This could either be used as a standalone objective or combined with other objectives such as ROUGE. This is a fairly simple idea, but we have already proved that simple metrics such as ROUGE-1 works. One possible problem with this approach is that it is very easy to learn to get a high percentage of novelty. If the model simply output random words, then the novelty would skyrocket, but it would yield very poor summaries. Combining this idea with e.g. ROUGE and MLE could be a reasonable middle-ground solution to keep the language model in-place, while still increasing abstractness.

7.2.6 Multiple objectives

We have shown that combining objectives work quite well, and we get the benefits of each one of the objectives we combine. In this context there is no reason why more than two objectives would not work. [Zhang and Lapata, 2017] show an example where they combine three objectives to simplify sentences and finds that reinforcement learning offers a great means to inject prior knowledge to the simplification task. In text summarization this could be knowledge about abstractness, fluency or relevance. In our case this could for instance be the combination of ROUGE-2, ROUGE-L and the discriminator objective. In fact, we could use any objective function, as long as it provides some sort of useful feedback.

7.2.7 Combined roll-out strategy

In Section 6.6, we discussed differences between the naive roll-out strategy and the Monte-Carlo roll-out strategy. We saw that the Monte-Carlo roll-out strategy was a lot slower to compute (5 to 12 times slower), while the naive roll-out strategy generally achieved higher ROUGE scores, given the fact that it was allowed to run for twice the amount of iterations. Another thing we noted was that the Monte-Carlo roll-out strategy converged faster in terms of number of updates to the network (required fewer iterations). This makes sense since the Monte-Carlo roll-out strategy gives more fine-grained rewards and thus gives a better gradient estimate. A possible trade-off to achieve even better results without sacrificing as much running-time could be to start with the naive roll-out strategy and switch to the Monte-Carlo roll-out strategy when the naive strategy has converged (stops learning). This way, we can speed up the initial learning phase, which seems trivial for both strategies, while being able to make use of the improved gradient estimates of the Monte-Carlo roll-out strategy at the later stages to allow the model to keep increasing the reward.

7.2.8 Alternative generator architectures

In this thesis, we have used a variation of the Pointer-Generator Network [See et al., 2017] as the architecture for the generator. One of the advantages of this architecture is the ability to copy factual details from the input article. However, as discussed in Section 6.6, the reinforcement learning agents learn to rely mostly on the pointer mechanism, which is not surprising, being an easy way to increase the reward for all the objective functions studied in this thesis. This leads to the question: How would other architectures that do not rely on a pointing mechanism learn to increase the reward in a RL or GAN setup? Currently, the state of the art which rely on a pointing mechanism seems to outperform other architectures in the task of abstractive text summarization [See et al., 2017] [Liu et al., 2017]. However, to the best of our knowledge, no architecture without a pointing mechanism has been used in the context of RL or GAN for improving abstractive text summarization. One example of an architecture which has seen promising results in the task of machine translation, that does not incorporate a pointing mechanism is the Transformer [Vaswani et al., 2017]. It would be interesting to see if such architectures gain the same improvements, and more specifically, how they learn to increase the reward.

7.2.9 Alternative discriminator architectures

In Section 6.4, we mention that a convolution based classifier may not be the optimal discriminator architecture. Trying out other types of architectures for the discriminator would be an interesting direction for future work. Examples of alternative architectures could be a LSTM based or a hybrid RNN/CNN architecture. Compared to a pure convolution based architecture, a RNN based architecture can potentially capture important short-term and long-term dependencies. If the discriminative performance of such a model performs better than the purely CNN based model, then we can expect even further improvements to the generative model when training with the GAN framework.

Bibliography

[Bahdanau et al., 2014] Bahdanau, D., Cho, K., and Bengio, Y. (2014). Neural machine translation by jointly learning to align and translate. *CoRR*, abs/1409.0473. Retrieved October 16, 2017.

[Bengio et al., 2015] Bengio, S., Vinyals, O., Jaitly, N., and Shazeer, N. (2015). Scheduled sampling for sequence prediction with recurrent neural networks. In *Advances in Neural Information Processing Systems*, pages 1171–1179. Retrieved March 24, 2018.

[Chauvin and Rumelhart, 1995] Chauvin, Y. and Rumelhart, D. E. (1995). *Backpropagation: theory, architectures, and applications*. Retrieved October 21, 2017.

[Chen et al.,] Chen, V., Montaño, E. T., and Puzon, L. An examination of the cnn/dailymail neural summarization task. Retrieved November 22, 2017.

[Cho et al., 2014a] Cho, K., Van Merriënboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., and Bengio, Y. (2014a). Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*. Retrieved November 17, 2017.

[Cho et al., 2014b] Cho, K., van Merrienboer, B., Gülçehre, Ç., Bougares, F., Schwenk, H., and Bengio, Y. (2014b). Learning phrase representations using RNN encoder-decoder for statistical machine translation. *CoRR*, abs/1406.1078. Retrieved October 15, 2017.

[Clemente et al., 2017] Clemente, A. V., Martínez, H. N. C., and Chandra, A. (2017). Efficient parallel methods for deep reinforcement learning. *CoRR*, abs/1705.04862. Retrieved April 30, 2018.

[Denton et al., 2015] Denton, E. L., Chintala, S., Szlam, A., and Fergus, R. (2015). Deep generative image models using a laplacian pyramid of adversarial networks. *CoRR*, abs/1506.05751. Retrieved January 26, 2018.

[Duchi et al., 2011] Duchi, J., Hazan, E., and Singer, Y. (2011). Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(Jul):2121–2159. Retrieved April 26, 2018.

[Fedus et al., 2018] Fedus, W., Goodfellow, I., and Dai, A. M. (2018). Maskgan: Better text generation via filling in the _. *arXiv preprint arXiv:1801.07736*. Retrieved March 23, 2018.

[Gers et al., 1999] Gers, F. A., Schmidhuber, J., and Cummins, F. (1999). Learning to forget: Continual prediction with lstm. Retrieved October 25, 2017.

[Goodfellow et al., 2016] Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep learning*. MIT press. Retrieved October 21, 2017.

[Goodfellow et al., 2014] Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., and Bengio, Y. (2014). Generative adversarial nets. In *Advances in neural information processing systems*, pages 2672–2680. Retrieved April 3, 2018.

[Graves et al., 2013] Graves, A., Jaitly, N., and Mohamed, A.-r. (2013). Hybrid speech recognition with deep bidirectional lstm. In *Automatic Speech Recognition and Understanding (ASRU), 2013 IEEE Workshop on*, pages 273–278. IEEE. Retrieved November 13, 2017.

[Graves and Schmidhuber, 2005] Graves, A. and Schmidhuber, J. (2005). Framewise phoneme classification with bidirectional lstm and other neural network architectures. *Neural Networks*, 18(5):602–610. Retrieved November 13, 2017.

[Grusky et al., 2018] Grusky, M., Naaman, M., and Artzi, Y. (2018). Newsroom: A dataset of 1.3 million summaries with diverse extractive strategies. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, New Orleans, Louisiana. Retrieved June 14, 2018.

[Hermann et al., 2015] Hermann, K. M., Kociský, T., Grefenstette, E., Espeholt, L., Kay, W., Suleyman, M., and Blunsom, P. (2015). Teaching machines to read and comprehend. *CoRR*, abs/1506.03340. Retrieved June 6, 2018.

[Hochreiter and Schmidhuber, 1997] Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. *Neural computation*, 9(8):1735–1780. Retrieved October 27, 2017.

[Kaur and Kaur, 2014] Kaur, H. and Kaur, L. (2014). Performance comparison of different feature detection methods with gabor filter. *International Journal of Science and Research (IJSR)*, 3:1880–1886. Retrieved May 30, 2018.

[Kim, 2014] Kim, Y. (2014). Convolutional neural networks for sentence classification. *CoRR*, abs/1408.5882. Retrieved October 16, 2017.

[Kingma and Ba, 2014] Kingma, D. and Ba, J. (2014). Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*. Retrieved December 12, 2017.

[Kjensmo, 2017] Kjensmo, S. (2017). Research method in ai-reproducibility of results. Master's thesis, NTNU. Retrieved June 7, 2018.

[LeCun et al., 1998] LeCun, Y., Bottou, L., Bengio, Y., and Haffner, P. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324. Retrieved November 3, 2017.

[Levy and Goldberg, 2014] Levy, O. and Goldberg, Y. (2014). Neural word embedding as implicit matrix factorization. In *Advances in neural information processing systems*, pages 2177–2185. Retrieved June 18, 2018.

[Lin, 2004] Lin, C.-Y. (2004). Rouge: A package for automatic evaluation of summaries. *Text Summarization Branches Out*. Retrieved March 19, 2018.

[Liu et al., 2017] Liu, L., Lu, Y., Yang, M., Qu, Q., Zhu, J., and Li, H. (2017). Generative adversarial network for abstractive text summarization. *CoRR*, abs/1711.09357. Retrieved January 19, 2018.

[Lopyrev, 2015] Lopyrev, K. (2015). Generating news headlines with recurrent neural networks. *CoRR*, abs/1512.01712. Retrieved October 16, 2017.

[Martens and Sutskever, 2011] Martens, J. and Sutskever, I. (2011). Learning recurrent neural networks with hessian-free optimization. In *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, pages 1033–1040. Retrieved October 23, 2017.

[Mitchell, 1997] Mitchell, T. M. (1997). Machine learning. 1997. *Burr Ridge, IL: McGraw Hill*, 45(37). Retrieved November 15, 2017.

[Moratanch and Chitrakala, 2017] Moratanch, N. and Chitrakala, S. (2017). A survey on extractive text summarization. In *Computer, Communication and Signal Processing (ICCCSP), 2017 International Conference on*, pages 1–6. IEEE. Retrieved November 14, 2017.

[Mousa and Schuller, 2016] Mousa, A. E.-D. and Schuller, B. W. (2016). Deep bidirectional long short-term memory recurrent neural networks for grapheme-to-phoneme conversion utilizing complex many-to-many alignments. In *INTERSPEECH*, pages 2836–2840. Retrieved June 17, 2018.

[Nallapati et al., 2016a] Nallapati, R., Xiang, B., and Zhou, B. (2016a). Sequence-to-sequence rnns for text summarization. *CoRR*, abs/1602.06023. Retrieved October 14, 2017.

[Nallapati et al., 2016b] Nallapati, R., Zhou, B., Gulcehre, C., Xiang, B., et al. (2016b). Abstractive text summarization using sequence-to-sequence rnns and beyond. *arXiv preprint arXiv:1602.06023*. Retrieved December 10, 2017.

[Nash et al., 1950] Nash, J. F. et al. (1950). Equilibrium points in n-person games. *Proceedings of the national academy of sciences*, 36(1):48–49. Retrieved April 3, 2018.

[Negnevitsky, 2005] Negnevitsky, M. (2005). *Artificial intelligence: a guide to intelligent systems*. Pearson Education. Retrieved October 16, 2017.

[Oh et al., 2015] Oh, J., Guo, X., Lee, H., Lewis, R. L., and Singh, S. (2015). Action-conditional video prediction using deep networks in atari games. In *Advances in Neural Information Processing Systems*, pages 2863–2871. Retrieved January 26, 2018.

[Paulus et al., 2017] Paulus, R., Xiong, C., and Socher, R. (2017). A deep reinforced model for abstractive summarization. *CoRR*, abs/1705.04304. Retrieved October 14, 2017.

[Ranzato et al., 2015] Ranzato, M., Chopra, S., Auli, M., and Zaremba, W. (2015). Sequence level training with recurrent neural networks. *arXiv preprint arXiv:1511.06732*. Retrieved February 15, 2018.

[Rojas, 2013] Rojas, R. (2013). *Neural networks: a systematic introduction*. Springer Science & Business Media. Retrieved October 21, 2017.

[Russell and Norvig, 2016] Russell, S. J. and Norvig, P. (2016). *Artificial intelligence: a modern approach*. Malaysia; Pearson Education Limited,. Retrieved May 14, 2018.

[Sasaki et al., 2007] Sasaki, Y. et al. (2007). The truth of the f-measure. *Teach Tutor mater*, 1(5). Retrieved February 15, 2018.

[See et al., 2017] See, A., Liu, P. J., and Manning, C. D. (2017). Get to the point: Summarization with pointer-generator networks. *CoRR*, abs/1704.04368. Retrieved April 4, 2018.

[Sharmilan and Chaminda,] Sharmilan, S. and Chaminda, H. T. Generate bioinformatics data using generative adversarial network: A review. Retrieved June 14, 2018.

[Silver et al., 2016] Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., Van Den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., et al. (2016). Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484–489. Retrieved January 26, 2018.

[Srivastava et al.,] Srivastava, N., Hinton, G. E., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. Dropout: a simple way to prevent neural networks from overfitting. *Journal of machine learning research*, 15(1):1929–1958. Retrieved November 15, 2017.

[Sutskever, 2013] Sutskever, I. (2013). Training recurrent neural networks. *University of Toronto, Toronto, Ont., Canada*. Retrieved October 23, 2017.

[Sutskever et al., 2014] Sutskever, I., Vinyals, O., and Le, Q. V. (2014). Sequence to sequence learning with neural networks. *CoRR*, abs/1409.3215. Retrieved October 15, 2017.

[Sutton and Barto, 1998] Sutton, R. S. and Barto, A. G. (1998). *Reinforcement learning: An introduction*, volume 1. MIT press Cambridge. Retrieved May 30, 2018.

[Vaswani et al., 2017] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. (2017). Attention is all you need. *CoRR*, abs/1706.03762. Retrieved June 21, 2018.

[Vinyals et al., 2015] Vinyals, O., Fortunato, M., and Jaitly, N. (2015). Pointer networks. In *Advances in Neural Information Processing Systems*, pages 2692–2700. Retrieved February 8, 2018.

[Wang et al., 2017] Wang, H., Qin, Z., and Wan, T. (2017). Text generation based on generative adversarial nets with latent variable. *CoRR*, abs/1712.00170. Retrieved January 25, 2018.

[Williams, 1992] Williams, R. J. (1992). Simple statistical gradient-following algorithms for connectionist reinforcement learning. In *Reinforcement Learning*, pages 5–32. Springer. Retrieved February 14, 2018.

[Xu et al., 2015] Xu, W., Callison-Burch, C., and Napoles, C. (2015). Problems in current text simplification research: New data can help. *Transactions of the Association of Computational Linguistics*, 3(1):283–297. Retrieved March 14, 2018.

[Yang et al., 2017] Yang, Z., Chen, W., Wang, F., and Xu, B. (2017). Improving neural machine translation with conditional sequence generative adversarial nets. *CoRR*, abs/1703.04887. Retrieved January 25, 2018.

[Yu et al., 2016] Yu, L., Zhang, W., Wang, J., and Yu, Y. (2016). Seqgan: Sequence generative adversarial nets with policy gradient. *CoRR*, abs/1609.05473. Retrieved January 26, 2018.

[Yu et al., 2015] Yu, Z., Ramanarayanan, V., Suendermann-Oeft, D., Wang, X., Zechner, K., Chen, L., Tao, J., Ivanou, A., and Qian, Y. (2015). Using bidirectional lstm recurrent neural networks to learn high-level abstractions of sequential features for automated scoring of non-native spontaneous speech. In *Automatic Speech Recognition and Understanding (ASRU), 2015 IEEE Workshop on*, pages 338–345. IEEE. Retrieved November 13, 2017.

[Zhang and Zhou, 2006] Zhang, M.-L. and Zhou, Z.-H. (2006). Multilabel neural networks with applications to functional genomics and text categorization. *IEEE transactions on Knowledge and Data Engineering*, 18(10):1338–1351. Retrieved October 16, 2017.

[Zhang and Lapata, 2017] Zhang, X. and Lapata, M. (2017). Sentence simplification with deep reinforcement learning. *CoRR*, abs/1703.10931. Retrieved January 24, 2018.

[Zhu et al., 2010] Zhu, Z., Bernhard, D., and Gurevych, I. (2010). A monolingual tree-based translation model for sentence simplification. In *Proceedings of the 23rd international conference on computational linguistics*, pages 1353–1361. Association for Computational Linguistics. Retrieved March 14, 2018.

BIBLIOGRAPHY

Appendix A

Here, we will present qualitative results for the pretrained model and the models trained with the different objectives. We show results on the best performing models, as summarized in Table 5.9. In the following figures, we show an article, the ground truth summary, the summary generated by the pretrained model, and the summaries created from the models trained with the different objective functions.

Article
by . simon jones . west ham are poised to win the battle for arsenal right-back carl jenkinson . the 22-year-old will be allowed out on loan following the arrival of calum chambers from southampton and mathieu debuchy from newcastle . hull city had made a concerted effort to sign the england international but denied making a formal # 3million proposal once it became clear the player wanted to stay in london . linked : west ham are set to sign carl jenkinson on loan from arsenal . signed : arsenal are willing to let him move following the arrival of calum chambers from southampton . video arsenal-bound chambers latest to exit saints . swansea and sunderland have also shown an interest but swansea are focusing on kyle naughton of tottenham . west ham are intensifying their search for a striker and want another centre back and a left-back . cardiff city 's fabio and declan john have been discussed as possibles for the left-back berth to give competition to aaron cresswell .

Ground truth summary
west ham poised to secure signature of arsenal right-back jenkinson . 22-year-old set to join hammers on loan following arrivals of calum chambers and mathieu debuchy . hull city , swansea and sunderland had all shown an interest .

Summary from pretrained model
west ham are set to sign carl jenkinson on loan from arsenal . the 22-year-old will be allowed out on loan following the arrival of calum chambers from southampton and mathieu debuchy from newcastle . west ham have also shown an interest but swansea are focusing on kyle naughton .

Summary from model trained with the ROUGE-1 objective
west ham are set to sign carl jenkinson on loan from arsenal . the 22-year-old will be allowed out on loan following the arrival of calum chambers from southampton and mathieu debuchy from newcastle . the england city 's fabio and declan john have been discussed as possibles for the left-back berth to give competition to aaron cresswell .

Summary from model trained with the ROUGE-2 objective
west ham are set to sign carl jenkinson on loan from arsenal . the 22-year-old will be allowed out on loan following the arrival of calum chambers from southampton and mathieu debuchy from newcastle . hull city 's fabio and declan john have been discussed as possibles for the left-back berth to give competition to aaron cresswell .

Summary from model trained with the discriminator objective
west ham are set to sign carl jenkinson on loan from arsenal . hull city 's fabio and declan john have been discussed as possibles for the left-back berth to give competition to aaron cresswell . the 22-year-old will be allowed out on loan following the arrival of calum chambers from southampton and mathieu debuchy from newcastle .

Summary from model trained with the combined objective
west ham are set to sign carl jenkinson on loan from arsenal . the 22-year-old will be allowed out on loan following the arrival of calum chambers from southampton and mathieu debuchy from newcastle .

Article
alan pardew claims that matt phillips ' 40-yard wonder strike for qpr against crystal palace was better than david beckham 's goal from inside his own half at selhurst park 19 years ago . phillips picked the ball up in the middle of the park and took a touch before belting an effort over palace keeper julian speroni and into the top left corner from 40 yards . but it proved to be too little , too late as qpr went down to a 3-1 defeat on saturday afternoon . palace manager pardew said : ' i remember beckham scoring the goal here and , from a technical point of view , phillips ' was better because it was an outstanding hit . ' matt can be very proud of it . in 25 or 30 years on a training ground , i ca n't remember seeing a better goal . ' matt phillips unleashed a ferocious strike from 40 yards that found the top corner . phillips ' effort dipped and swerved past julian speroni but it was too little too late for rangers . phillips grabs the ball in a hurry as he makes his way back to the centre spot .

Ground truth summary
matt phillips scored 40-yard wonder goal for qpr on saturday . alan pardew claimed strike was better than david beckham 's stunner .

Summary from pretrained model
matt phillips ' 40-yard wonder strike for qpr against crystal palace was better than david beckham 's goal from inside his own half at selhurst park 19 years ago . matt phillips unleashed a ferocious strike from 40 yards that found the top corner . palace manager pardew said : ' matt can be very proud of it in 25 or 30 years on a training ground , i ca n't remember seeing a better goal '

Summary from model trained with the ROUGE-1 objective
matt phillips ' 40-yard wonder strike for qpr against crystal palace was better than david beckham 's goal from inside his own half at selhurst park 19 years ago . matt phillips unleashed the ball in a hurry as he makes his way back to the centre spot in the middle of the park and took a touch before belting an effort over palace keeper julian speroni and into the top left corner from 40 yards . palace manager pardew said : ' i remember beckham scoring the goal here and i ca n't remember seeing a better goal '

Summary from model trained with the ROUGE-2 objective
matt phillips ' 40-yard wonder strike for qpr against crystal palace was better than david beckham 's goal from inside his own half at selhurst park 19 years ago . phillips picked the ball up in the middle of the park and took a touch before belting an effort over palace keeper julian speroni . matt phillips unleashed a ferocious strike from 40 yards that found the top corner .

Summary from model trained with the discriminator objective
matt phillips ' 40-yard wonder strike for qpr against crystal palace was better than david beckham 's goal from inside his own half at selhurst park 19 years ago . matt phillips unleashed the ball in a hurry as he makes his way back to the centre spot .

Summary from model trained with the combined objective
matt phillips ' 40-yard wonder strike for qpr against crystal palace was better than david beckham 's goal from inside his own half at selhurst park 19 years ago . phillips picked the ball up in the middle of the park and took a touch before belting an effort over palace keeper julian speroni .

Article
danny welbeck is facing a race against time to be fit for saturday 's crunch showdown against liverpool . the arsenal forward has returned to north london for treatment after sustaining a knee injury in england 's win over lithuania on friday . the news will come as a blow to arsene wenger ahead of the clash against liverpool , which could prove pivotal in the race for a top-four finish and qualification for next season 's champions league . danny welbeck sustained a knee injury in england 's win over lithuania at wembley stadium on friday night . the arsenal star will miss england 's game against italy on tuesday after he returned to london for treatment . welbeck is responding well to treatment at arsenal 's london colney hq and has not be ruled out of the visit of brendan rodgers side . gunners medical staff will make a late decision on his availability for the encounter but are confident he will prove his fitness . welbeck , in action against newcastle , faces a race to be fit for arsenal 's saturday clash against liverpool .

Ground truth summary
danny welbeck has returned to london with a knee injury which he sustained in england 's euro 2016 qualification win over lithuania . the arsenal forward is responding well to treatment at london colney . medical staff will make a late decision on welbeck 's availability . arsenal host liverpool in lunchtime kick-off at the emirates on saturday .

Summary from pretrained model
danny welbeck sustained a knee injury in england 's win over lithuania at wembley stadium on friday night . welbeck is responding well to treatment at arsenal 's london colney hq and has not be ruled out of the visit of brendan rodgers side . the arsenal star will miss england 's game against italy on tuesday .

Summary from model trained with the ROUGE-1 objective
danny welbeck sustained a knee injury in england 's win over lithuania on friday . the arsenal star will miss england 's game against italy on tuesday . danny welbeck is facing a race to be fit for saturday 's crunch showdown against liverpool .

Summary from model trained with the ROUGE-2 objective
danny welbeck sustained a knee injury in england 's win over lithuania at wembley stadium on friday night . the arsenal forward has returned to north london for saturday 's crunch showdown against liverpool . danny welbeck is responding well to treatment at arsenal 's london colney hq .

Summary from model trained with the discriminator objective
danny welbeck sustained a knee injury in england 's win over lithuania . the arsenal forward has returned to north london for treatment . welbeck is responding well to treatment at arsenal 's london colney hq .

Summary from model trained with the combined objective
danny welbeck sustained a knee injury in england 's win over lithuania on friday . the arsenal forward has returned to north london for treatment . arsenal star will miss england 's game against italy on tuesday .

Article
by . simon jones . steve bruce has made an offer to free agent defender joleon lescott and held talks with tom ince over a possible move to hull city . former blackpool winger ince , 22 , has turned down the chance to move to inter milan and is talking to clubs in england with stoke city also keen . although ince is a free agent , blackpool believe the # 1.5 m loan fee crystal palace paid for him in january means they can demand # 8m in training compensation at a tribunal . in demand : joleon lescott is wanted by west ham , aston villa , crystal palace , portland timbers and al ain . that would have a huge bearing on whether hull proceed . bruce has already opened talks with tottenham over an # 8m deal for michael dawson and jake livermore and has offered lescott a two year deal with option for a third . west ham , aston villa , crystal palace , portland timbers and al ain have also made approaches to the 31-year-old defender . money matters : blackpool could demand # 8m in training compensation for tom ince in a tribunal .

Ground truth summary
steve bruce wants former man city defender joleon lescott to join hull . lescott has also interested west ham , aston villa and crystal palace . tigers boss is also in talks to sign crystal palace winger tom ince . ince has turned down the chance to follow father paul at inter milan . blackpool may seek # 8m training compensation for ince at a tribunal .

Summary from pretrained model
joleon lescott has turned down the chance to move to inter milan and is talking to clubs in england with stoke city . blackpool could demand # 8m in training compensation for tom ince in a tribunal . bruce has already opened talks with tottenham over an # 8m deal for michael dawson and jake livermore .

Summary from model trained with the ROUGE-1 objective
blackpool believe the # 1.5 m loan fee crystal palace paid for him in january means they can demand # 8m in training compensation for tom ince in a tribunal . ince , 22 , has turned down the chance to move to inter milan and is talking to clubs in england with stoke city . bruce has already opened talks with tottenham over an # 8m deal for michael dawson and jake livermore .

Summary from model trained with the ROUGE-2 objective
former blackpool winger ince , 22 , has turned down the chance to move to inter milan and is talking to clubs in england with stoke city . bruce has already opened talks with tottenham over an # 8m deal for michael dawson and jake livermore . blackpool believe the # 1.5 m loan fee crystal palace paid for him in january means they can demand # 8m in training compensation for tom ince in a tribunal .

Summary from model trained with the discriminator objective
joleon lescott is wanted by west ham , aston villa , crystal palace , portland timbers and al ain . ince is a free agent , blackpool believe the # 1.5 m loan fee crystal palace paid for him in january means they can demand # 8m in training compensation for tom ince in a tribunal .

Summary from model trained with the combined objective
joleon lescott is a free agent , blackpool believe the # 1.5 m loan fee crystal palace paid for him in january means they can demand # 8m in training compensation for tom ince in a tribunal . bruce has already opened talks with tottenham over an # 8m deal for michael dawson and jake livermore .

Article
arsene wenger is set to receive a huge fitness boost ahead of arsenal 's crunch clash with liverpool a week on saturday . jack wilshere , mikel arteta and mathieu debuchy are all back in training and close to rejoining the first team . if all goes according to plan the trio should be available to face liverpool at the emirates . arsenal midfielder jack wilshere could face liverpool on april 4 after recovering from an ankle injury . captain mikel arteta -lrb- left -rrb- is also expected to be available for selection following his return to training . wilshere has missed four months following an ankle operation , and arteta has missed five months with ankle and calf issues . right back debuchy has been sidelined since january 11 with a shoulder injury . midfielder alex oxlade-chamberlain also hopes to be available for the game against brendan rodgers 's side on april 4 after a hamstring injury . but he may have to wait until the trip to burnley on april 11 for his return . arsenal right back mathieu debuchy has been sidelined since sustaining a shoulder injury against stoke city .

Ground truth summary
jack wilshere , mathieu debuchy and mikel arteta have returned to training . the arsenal trio should be in contention to face liverpool on april 4 . wilshere has been out for four months following an ankle operation . arteta 's last match for arsenal came against dortmund back in november . summer signing debuchy has been out of action since january 11 .

Summary from pretrained model
arsenal face liverpool at the emirates stadium on saturday . jack wilshere , mikel arteta and mathieu debuchy are all back in training . arsenal midfielder jack wilshere could face liverpool on april 4 . click here for all the latest arsenal news .

Summary from model trained with the ROUGE-1 objective
arsene wenger is set to receive a huge fitness boost ahead of arsenal 's crunch clash with liverpool . jack wilshere , mikel arteta and mathieu debuchy are all back in training and close to rejoining the first team . arsenal midfielder jack wilshere could face liverpool on april 4 .

Summary from model trained with the ROUGE-2 objective
arsene wenger is set to receive a huge fitness boost ahead of arsenal 's crunch clash with liverpool . jack wilshere , mikel arteta and mathieu debuchy are all back in training and close to rejoining the first team . arsenal midfielder jack wilshere could face liverpool on april 4 .

Summary from model trained with the discriminator objective
arsene wenger is set to receive a huge fitness boost ahead of arsenal 's crunch clash with liverpool . jack wilshere , mikel arteta and mathieu debuchy are all back in training and close to rejoining the first team . arsenal midfielder jack wilshere could face liverpool on april 4 . click here for all the latest arsenal news .

Summary from model trained with the combined objective
arsene wenger is set to receive a huge fitness boost ahead of arsenal 's crunch clash with liverpool . jack wilshere , mikel arteta and mathieu debuchy are all back in training . arsenal midfielder jack wilshere could face liverpool on april 4 . click here for all the latest arsenal news .

Article
a four-year-old girl has undergone surgery to her face after being attacked by a dog in south-east queensland . a statement from racq careflight rescue says the girl was bitten on the face by a dog on friday evening in the town of stanthorpe in the southern downs region . a four-year-old girl has undergone surgery to her face after being attacked by a dog in southeast queensland . she was rushed to brisbane 's mater children 's hospital in a stable condition . the girl was taken to a local hospital before being flown to brisbane 's mater children 's hospital in a stable condition later that evening . she underwent surgery for lacerations to her nose and cheek . the dog was not the family pet but had been spotted in the stanthorpe area by the girl 's family before the attack . the girl was bitten on the face by a dog on friday evening in the town of stanthorpe , southeast queensland .

Ground truth summary
the girl was bitten in the face in stanthorpe , southeast queensland . she was flown to brisbane 's mater children 's hospital in a stable condition . she underwent facial surgery for lacerations to her nose and cheek . the dog had been spotted in the area by the girl 's family before the attack .

Summary from pretrained model
the girl was bitten on the face by a dog on friday evening in the town of stanthorpe , southeast queensland . the dog was not the family pet but had been spotted in the stanthorpe area by the girl 's family .

Summary from model trained with the ROUGE-1 objective
the girl was bitten on the face by a dog on friday evening in the town of stanthorpe in the southern downs region . she was rushed to brisbane 's mater children 's hospital in a stable condition . the dog was not the family pet but had been spotted in the stanthorpe area by the girl 's family before the attack .

Summary from model trained with the ROUGE-2 objective
the girl was bitten on the face by a dog on friday evening in the town of stanthorpe in the southern downs region . the four-year-old girl was taken to brisbane 's mater children 's hospital in a stable condition . the dog was not the family pet but had been spotted in the stanthorpe area by the girl 's family before the attack .

Summary from model trained with the discriminator objective
the girl was bitten on the face by a dog on friday evening in the town of stanthorpe in the southern downs region . the dog was not the family pet but had been spotted in the stanthorpe area .

Summary from model trained with the combined objective
the girl was bitten on the face by a dog on friday evening in the town of stanthorpe in the southern downs region . she was rushed to brisbane 's mater children 's hospital in a stable condition . the dog was not the family pet but had been spotted in the stanthorpe area .

Appendix A

Article
by . simon jones . swansea are in talks to sign ecuador winger jefferson montero from monarcas morelia for # 4million . the 24-year-old appeared in all three of his country 's world cup matches as ecuador were knocked out at the group stage . the swans will also listen to offers for # 10m-rated michu . talks : swansea are keen to sign ecuador wideman jefferson montero from monarcas morelia for # 4million . proven : the 24-year-old montero -lrb- left -rrb- has scored eight goals in 41 appearances for ecuador . spanish frontman michu made a huge impact during his first year in the premier league but missed much of last season through injury . meanwhile , captain ashley williams , 29 , has signed a new four-year contract with the south wales club . staying put : swansea defender ashley williams has decided to stay at the liberty stadium .

Ground truth summary
swansea in talks to bring ecuador wideman jefferson montero to the premier league in # 4million deal . winger played in all three of his country 's world cup matches . swans will also listen to offers for frontman michu .

Summary from pretrained model
swansea defender ashley williams has signed a new four-year contract with the south wales club . swansea are keen to sign ecuador wideman jefferson montero from monarcas morelia for # 4million . ashley williams , 29 , has scored eight goals in 41 appearances for ecuador .

Summary from model trained with the ROUGE-1 objective
swansea are in talks to sign ecuador winger jefferson montero from monarcas morelia for # 4million . the 24-year-old appeared in all three of his country 's world cup matches as ecuador were knocked out at the group stage . ashley williams , 29 , has signed a new four-year contract with the south wales club .

Summary from model trained with the ROUGE-2 objective
swansea are in talks to sign ecuador winger jefferson montero from monarcas morelia for # 4million . the 24-year-old appeared in all three of his country 's world cup matches as ecuador were knocked out at the group stage . ashley williams , 29 , has signed a new four-year contract with the south wales club .

Summary from model trained with the discriminator objective
swansea are in talks to sign ecuador winger jefferson montero from monarcas morelia for # 4million . the 24-year-old appeared in all three of his country 's world cup matches as ecuador were knocked out at the group stage . swansea defender ashley williams has signed a new four-year contract with the south wales club .

Summary from model trained with the combined objective
swansea are in talks to sign ecuador winger jefferson montero from monarcas morelia for # 4million . the 24-year-old appeared in all three of his country 's world cup matches as ecuador were knocked out at the group stage . swansea defender ashley williams has signed a new four-year contract with the south wales club .