UiO : Department of Mathematics
University of Oslo

FgFlex

A flexible, multitasking sequence-labeler for fine-grained sentiment analysis

Per M. C. Halvorsen

Master's Thesis, Spring 2022

Abstract

The task of fine-grained sentiment analysis aims to extract detailed opinions from text. In this context, opinions consist of four main elements: where the opinion is targeted, who holds the opinion, the scope of words that expresses the opinion, and a sentiment polarity classification, i.e. if the opinion is positive or negative. Machine learning models built to solve the task of fine-grained sentiment analysis must learn to represent the nuanced relational patterns between these elements.

In our thesis, we apply previous state-of-the-art sequence-labelers, built to solve fine-grained sentiment analysis, to Norwegian text. Originally built using English data sets, these baseline architectures split the main task of fine-grained sentiment analysis into subtasks, each focused on predicting their own respective opinion element. From these architectures, we synthesize a flexible variant of such a sequence-labeler, to study the impact of different relations between the subtasks of the model.

Each of the five models we built to conduct this study were hyperparameter tuned, giving us their respective optimized configurations. Using a hypothesis test formulated from our research questions, we compared final performances of our optimized models. A confidence interval drawn about an approximated population distribution of the expected results from our best baseline assists in the model comparisons.

Our findings suggest that simpler models often perform relatively well compared against more complex models. We discuss the most important components for high model performance, along with the computational requirements of each model. Before rounding off the thesis, we point out some areas for improvement of our project and offer some potential research directions future work could start in.

Acknowledgements

This experiment is a part if the SANT (Sentiment Analysis of Norwegian Texts) research project. The goal of SANT is to create open resources for sentiment analysis in Norwegian. The project represents a collaboration between the Language Technology Group (LTG) at the Department of Informatics at the University of Oslo, and three of Norway's largest media groups; the public Norwegian Broadcasting Corporation NRK/P3 and the privately held Schibsted Media Group and Aller Media. These media partners graciously provided data in the form of reviews, collected across a range of different domains; music, literature, restaurants, home electronics, and more. These reviews build the foundation of our data set.

The annotation efforts to create the finely curated data set used in our experiment were vital to this project. Without this immense effort, the experiments run here would not have been possible. NoReC$_{fine}$ (Øvrelid et al. 2020), the paper presenting these data, discusses the annotation process in detail, and explicitly acknowledges all of the seven kind souls that created these intricate annotations. We are incredibly thankful for their efforts!

We would also like to thank our advisors, Erik Velldal and Lilja Øvrelid for the guidance and motivation they provided along the way. They contributed with amazing detailed feedback throughout development, along with critical, yet objective, discussions that helped lead to debugging solutions and documentation clarity. Your help was very much appreciated, thank you!

Contents

CHAPTER 1

Introduction

Sentiment analysis (SA) refers to the act of extracting opinions from text in hopes to quantify the meaning of the input. Opinion extraction can occur at different levels compared to what the goal of an experiment is. In some cases, a data set contains large bodies of text associated with a single polarity classification. Experiments on such data sets are often referred to as document level analysis. Other times, smaller sequences of text are used as inputs along with more detailed annotations like the object an opinion is targeting, who holds the opinion, or the expression scope containing the opinion itself. Analyses on data sets like these are sometimes called targeted or fine-grained sentiment analysis.

Document level annotations have been readily available for SA tasks since the birth of the internet, because humans have a knack for attaching ratings to public comments (Dua and Graff 2011). Thanks, Yelp! On the other hand, finely annotated data sets have previously been hard to come by, and even harder to produce. Recently, however, the field of sentiment analysis has grown in tact with it's commercial demand, and the collateral increase in resources have helped produce fine-grained annotations for a plethora of languages (German - Klinger and Cimiano 2014; English - Pontiki, Galanis, Pavlopoulos et al. 2014; Basque and Catalan - Jeremy Barnes, Badia and Lambert 2018; Norwegian -Øvrelid et al. 2020). Among others, these detailed annotations unlock the possibility for deeper analysis on inter-sentence dependencies between targets and opinions discussed in a text, paving the way for research on neural architectures built to predict these types of data.

1.1 Project goal

This project will explore multitask-learning sequence-labelers for a fine-grained sentiment data set in Norwegian. Specifically, a blend of current state-of-the-art contextual embeddings (Kutuzov et al. 2021) together with interactive messaging mechanisms throughout the network (He et al. 2019, Chen and Qian 2020) are used to extract the target, holder, and expression scope of opinions in an input, along with each opinion's polarity classification. Contextual embeddings combine the idea of traditional word embeddings as vectors with self-attention mechanisms of a transformer (Devlin et al. 2019). An interactive, multitask learning network shares information between the subtasks of a model through stacking and attention blocks (Chen and Qian 2020).

With help from a hypothesis test, our project aims to find which components are useful for solving the task of fine-grained sentiment analysis. In the following chapters, we present the steps of building flexible fine-grained opinion sequence-labelers. Inspired by previous state-of-the-art models, our novel architecture, named `FgFlex`, makes for easy task-wise relation comparisons. Instead of being restricted to predetermined interactions between subtasks, our flexible model can apply attention relations between any of the subtasks our model is trained for. Our thesis presents how and why this flexible architecture was made.

1.2 Outline

Building fine-grained opinion sequence-labelling architectures requires many steps. In this section, we outline the evolution of our project, from background knowledge presentation through our experimental set-up, over to our development and implementation methodology, and finally ending with a summarizing discussion around the results we found.

Specifically, Chapter 2 will present and discuss some preceding experiments that introduced and inspired the topics explored in this thesis. The aim of the chapter is to provide a solid foundation of background knowledge important to our experiment. This includes detailed explanations of annotation granularity and terminology, introduction of attention based test-representations, along with the interactive multitask learners. The main data set that is used in our project is also introduced as part of this background knowledge presentation.

An outline of our project framework will be presented in Chapter 3. The platforms our code was built and run on are presented here, along with the metrics used to evaluate the code and our hypothesis used to guide our experimentation process. Also, the different architectural components will be briefly introduced, along with how each interacts with the others. The chapter ends with outlining the full models we test in this project.

The implementation process, including development and hyperparameter searching, will be the main topic of Chapter 4. We guide the reader through how our experiment unfolded, revealing some initial findings as motivations for each next step. The mistakes made and where they were caught will also be mentioned at their respective points of discovery throughout the experiment, together with a bug summary rounding off this methodology chapter.

Chapter 5 will present and discuss the results from experimentation. The components most important for model performance are summarized and reasoned for. Computational efficiency between the models tested is discussed, as well. The chapter ends with the final comparison between architectures, crowning a best final model, and discussing what statistically significance findings our model revealed.

Chapter 6 will summarize and conclude the experiments executed throughout our project. Here, the key insights are gathered and reiterated, providing a simplified summary of our biggest findings. Some areas for eventual improvements on project development are surveyed, as well. We finish the chapter by presenting a few potential directions future work can start in.

Finally, before the Bibliography section at the very end of our project, an appendix containing supplementary plots to our development and implementation model is provided.

1.3 Research Questions

Throughout Chapter 2, a handful of the most important experiments leading up to the current state of fine-grained sentiment analysis will be briefly introduced. As mentioned intermittently during these introductions, some of the experiments serve as the starting point for the architecture developed for this thesis. The overall goal of this thesis is to find an even more optimal multitasking configuration of a sequence-labeler for fine-grained sentiment analysis, through the introduction of novel interactions between subtasks.

Precisely put, this thesis asks (**RQ1**) if we can improve upon the current state-of-the-art systems for extracting opinions from input sentences. An opinion in the context of this thesis includes the opinion's target, holder (if present), polar expression scope, and polarity classification. Improvements are measured against the same metrics used in the original baseline projects (He et al. 2019, Chen and Qian 2020), testing all models on the same hold-out evaluation data, a pre-split partition of the NoReC$_{fine}$ data set (Øvrelid et al. 2020).[1] Both model performance along with training times and resource demands are taken into consideration during the comparison.

Additionally, this experiment asks (**RQ2**) what components are necessary for performance enhancement of fine-grained sequence-labelers on Norwegian data? Through comparing baselines against simpler and more complex models, we try to isolate the most important pieces of our studied architectures.

These questions are mentioned again when defining our hypothesis in Chapter 3, then again in our final model comparisons 5. Before that, however, we look into some important background knowledge.

[1]More on metrics in Section 3.2.

CHAPTER 2

Background

Natural language processing (NLP) can be generally explained as the use of digital technologies to enhance research on human language. NLP projects take on many shapes and forms, from generative language-to-text models (Ramesh et al. 2022) to analyses on how much linguistics a pre-trained model actually understands (Ettinger 2020). A key challenge that often arises in many experiments is how to correctly and precisely capture the main content of an input.

Fine-grained sentiment analysis (FGSA) is a branch of NLP that tackles this problem by searching for the individual opinions of an input, each with their own estimated sentiment polarity classification. While not directly applicable to non-judgemental input sequences (like the two example projects cited in the previous paragraph), FGSA models can be beneficial for analyzing comments, debates, speeches, and other areas of language usually doused with heavy rhetoric.

Engineering and researching fine-grained models are only part of the broader field within NLP called sentiment analysis (SA). SA models generally classify a sentiment polarity for a given input, but on varying scales. Often, individual opinions included in the input are looked at together, not one-by-one as done in FGSA models.

Different insights can be found through the different granularity levels a SA model can choose to focus on. For example, teaching a model to correctly predict document level sentiment on product reviews that express either a positive or negative tone could help a company find potential areas for product improvement. On the contrast, fine-grained opinion extraction applied to the sentences of a campaign speech could help map out a candidate's political stances on certain issues, allowing voters to make much more educated judgments in the polls. This thesis aims to solve the FGSA task, closely related to the latter, where we aim to correctly extract sentimental opinions from a sequence of individual input sentences.

Before jumping into architecture development for FGSA, though, we need to clearly define our terminology and understanding of the current state of research in this field. This chapter aims to provide some background knowledge on FGSA, including a clear definition of the elements that make up an opinion along the annotation scheme of the data set used in this experiment (2.1, 2.2), historical versus modern text-representation techniques (2.3, 2.4), Finally, interactive networks built specifically for fine-grained extraction are introduced to round off this chapter (2.5, 2.6).

The information presented in this chapter serves as the first stepping stone for understanding the motivation behind our experiment. With it, we ensure our audience syncs their interpretations of the concepts discussed in this thesis with our own, at least for the scope of the experiment.

2.1 Levels of Sentiment Analysis

The information obtained from a sentiment analysis experiment depends on the granularity of annotations in the data being analyzed. More detailed annotations naturally provide more context, opening up for the possibility to learn more about a given input. Of course, this increased level of detail requires more resources during training. Depending on the overall goal of the experiment, intricate information extraction may not always be desired or necessary.

The following sections present the three most common levels of sentiment analysis, occurring at document, sentence, and target level. In addition to their mere distinctions, we also discuss how one level naturally lead into the other, and how the data sets for each level were iteratively developed.

2.1.1 Document level

It's intuitive to begin with defining the largest, most generalized level of sentiment granularity, since it is here the simplest data sets are built. **Document-level** sentiment analyses produce a *single* polarity classification for an entire input. These inputs can range in size from single sentences to full documents with multiple paragraphs. While these models often focus on an opinion's polarity, other singular classifications can be learned depending on the data set, including toxicity, formality, or hostility (Georgakopoulos et al. 2018, Abu Sheikha and Inkpen 2010, Gupta et al. 2021). Due to variations in data collection methods, there is no standard limit to how large or small an input document must be. In this project, document-level analysis assumes inputs usually contain more than a single sentence.

Luckily for NLP scientists, there exists multiple sources of publicly available documents with polarity scores in the wild world of unlabeled data. From Yelp customers (Pontiki, Galanis, Pavlopoulos et al. 2014) through international movie critics (Pang and Lee 2004) to Amazon product reviews (Dua and Graff 2011), generous internet users who both rated some arbitrary object (out of 5 stars, 6 pip dice, or the like) and explained their rating through a comment naturally provide exactly this. These user-generated **rating reviews** paired with their quantified rating are perfect for document-level polarity classification, since little to no extra manual annotations are needed when creating the data sets.[1]

Originally, inputs were mapped to count vectors, sometimes referred to as bag-of-words embeddings (Harris 1954). Here, word order is disregarded, and only counts of co-occurrences between words in a given input are measured.

[1]Granted, not all comments are equally balanced and objective, a problem limiting complete unsupervised data set construction and encouraging inter-annotator agreements during development. See section 2.2 for more on this.

Experiments eventually began to replace representing individual words with representing n-grams in order to incorporate positional information [2] (Bahl, Jelinek and Mercer 1983). A bag-of-n-grams setup provided a lot more context for a given input, but was only practical for small sizes of n due to the *curse of dimensionality*[3].

Derived from bag-of-n-gram representations, static word embeddings mapped words to a smaller dimensional space than the previously used total vocabulary size of a dataset. In this abstract representation, similar concepts were given vectors pointing in more or less the same direction, while dissimilar concepts received vectors less correlated.

Neural approaches applied to downstream document representations were introduced shortly after static embeddings, thus further increasing model performance (Bergem 2018). Individually, both convolutional and recurrent neural networks improved state-of-the-art results for sentiment analysis at document level by capturing detailed dependencies previously disregarded in count vectors approaches (Santos and Gatti 2014, Tang, Qin and T. Liu 2015). Some experiments combined the two to form a hybrid system, and achieved even greater improvements (Xiang Zhang, Zhao and LeCun 2016, Lai et al. 2015, Bergem 2018). A more thorough presentation of these internal text representations will be explored in section 2.3 on static word embeddings.

Proper representations of an input document and internal model representations can only improve a system's performance up to a certain point. Documents containing balanced arguments (both for and against a particular topic) can confuse a model, providing a noisy signal. In order to convert some of this noise to meaningful information, some experiments dove even deeper into the details of their input data.

2.1.2 Sentence level

The next step of increased granularity is *sentence-wise* sentiment classifications. In natural language, not every sentence can be assumed to carry some opinion or evaluation. This means, sentence level models often find if a sentence is *evaluative* or *objective* first, before classifying the sentence's polarity. Some experiments also consider if a sentence is a *fact-implied non-personal*, which denotes that a sentence has some polarity expressed, even though the opinion expressed is not personally subjective (Mæhlum et al. 2019[4]).

A handful of experiments at this level used sentence classification as an auxiliary task. These passed the insights from sentence level classifications back to modules focused on document classification, in hopes to further increase overall performance at document level (Xiaoqian Zhang et al. 2011). Others compared the performance of sentence based approaches to document level analysis to argue for or against if even more detailed annotations are necessary (Yu and Hatzivassiloglou 2003). Even others focus only on sentence level labelling in hopes to get the most out datasets for low resource languages by finding the most content-bearing sentences of a document (Qu 2013). Independent

[2]individual words being consider a unigram, or 1-gram

[3]*Curse of dimensionality* refers to exponential increases in computation requirements due to increased dimensions of representations

[4]This was actually an experiment leading up to the one that produced our data set. More on this experiment in 2.2.4.2

of motivation, sentence level sentiment analysis extracts meaningful sentences from an input document to be analyzed further, making it an important step towards fine-grained analysis.

Early research applied traditional techniques, like Naive Bayes on n-gram occurrences, where sentence level predictions were generated from conditional probabilities of positive or negative features in a sentence, like Yu and Hatzivassiloglou 2003. In this particular experiment, features are considered to be everything from individual words, to bi- or trigrams, or even parts-of-speech tags.

As Toprak, Jakob and Gurevych 2010 points out, high quality sentence level classifications require reliable finely annotated data sets. These meticulously curated details help sentence level models filter important sentences from non-important, allowing for learning more relevant information more efficiently.

Natural developments on SA data sets evidently pushed toward fine-grained data sets annotated on top of these sentence level classifications.

2.1.3 Target level

The NLP task typically referred to as *fine-grained sentiment analysis* focuses on extracting and classifying each opinion found in a sentence, rather than producing a single classification for the sentence. Predictions at this level are directed toward *all of the targets* which have some sentiment expressed about them in the input. This allows models to capture multiple sentiment expressions in a single sentence, giving them a flexibility otherwise impossible to achieve at coarser annotation levels. Multiple expressions in a single sentence provide more information to a model trying to classify polarity in one orientation or the other.

In previous experiments, targets are sometimes referred to as *entities* or *aspects*[5]. While there does exist some differences between authors in the exact definitions of these terms, there is quite a bit of overlap between the definitions as well. All refer to the *main subject* of an input. In sentiment bearing sentences, this is the object in which a sentiment expression is directed toward. This thesis will use the term *target* to describe this concept. Section 2.2.3 clarifies some other varying terminology commonly used in literature about these projects.

2.1.3.1 Historical fine-grained experiments

Target level experiments were being explored already in the early 2000s. Minqing Hu and Bing Liu 2004 use fine-grained annotations to mine customer opinions from reviews, in hopes to summarize the customer's attitude toward a product. Wiebe, Wilson and Cardie 2005 presented another early project, this time using a fine-grained annotation schema to extract *private states* from an input. Wiebe defines a private state as an "internal states that cannot be directly observed by others." In other words, a private state can be thought of as an opinion.

[5] *Aspect*-based experiments often try to further classify the aspects into aspect categories (Pontiki, Galanis, Papageorgiou et al. 2016). For example, if "food" was a target, then the category it belongs to might be restaurant. However, this aspect category abstraction is irrelevant for our experiment since these categories are not included in the data set we chose to focus on.

Even though researchers had been edging toward target extraction for at least a decade prior, it was the three *Aspect Based Sentiment Analysis* tasks at the SemEval 2014, 2015, and 2016 conferences that forced fine-grained sentiment analysis into the NLP limelight. These tasks inspired solution proposals using a range of popular architectures, from convolutional filters to dependency parsing using hierarchical models (CNN: Santos and Gatti 2014; Hierarchical: Ruder, Ghaffari and Breslin 2016).

2.1.3.2 Useful details

Often, both opinion holders and targets are mined at the fine-grained level, along with their respective polarities (B. Liu 2012). As in most detailed NLP tasks, fine-grained experiments are usually haunted by the curse of dimensionality, meaning scalability is often an issue that needs to be considered. Models built to make these predictions need to be much more complex than for the previous levels since many more elements per input need to be predicted.

As model complexity increases, so too grows the required computational resources needed to generate new predictions. Further, extra annotations efforts are needed to manually label opinion holders and targets in developmental data sets, which require both additional time and resources. This raises the question of what degree of detail is necessary to run a meaningful sentiment analysis experiment in the most efficient manner possible.

In their paper titled *If You've Got It, Flaunt It!*, Jeremy Barnes, Øvrelid and E. Velldal 2021 explored research questions regarding exactly which annotations were most helpful for target extraction on English datasets, and whether or not holder and expression information is useful for polarity classification.

The findings here suggest architectures focused on target extraction benefit from jointly predicting the target of the expressed opinion as a combined label together with it's polarity, but not from including the holder and expression in this label. This means using joint target-BIO tags with each token's polarity instead of separating them as two individual tasks could help extract targets.

Additionally, when classifying an opinion's polarity, including the additional information about holders and expression scopes in the model help increase performance. A system trying to tackle both of these tasks, either as auxiliary[6] tasks next to the main goal or as subtasks[7] of the main goal, should in some way incorporate these annotations to increase performance further.

This thesis will attempt to build off the findings of the *Flaunt It!* experiment, this time using Norwegian data.

Limited availability of finely annotated data sets served as a bottleneck for fine-grained experiments in lower resource languages around the turn of the century. The boom of the internet following 1999 collaterally built many document level sentiment analysis data sets naturally. Contrarily, fine-grained data sets are still today mainly annotated by hand. Only recently have they become relatively widespread for smaller languages thanks to, among others:

[6]auxiliary task: a task that can provide useful information to overall task, but not directly a part of the main goal

[7]subtask: a part of the main task desired to be predicted

SemEval 2104 (Pontiki, Galanis, Pavlopoulos et al. 2014), OpeNER (Agerri et al. 2013), and the Stanford Treebank (Socher et al. 2013).

The next section will discuss the annotation schema we expect when looking at fine-grained data sets, as well as introduce $\mathbf{NoReC}_{fine}$, the fine-grained data set for Norwegian that will be used for this experiment (Øvrelid et al. 2020).

2.2 Annotations

Data sets for sentiment analysis come in many shapes and forms. Naturally, which of the three granularity levels explained above an experiment focuses on will affect the annotations needed in the data. However, even within these three granularity categories, there still exists a plethora of variations possible in the data sets.

Often, fine-grained or sentence level data sets are derived from previous document level sets. The history of sentiment analysis shows this as well; document level analyses were very popular in the early 2000s when Web 2.0 first became widespread and online review forums became available to the general public, while fine-grained and sentence level experiments became main-stream popular almost a decade later as annotation efforts started reaching larger scales.

Around the same time that the first SemEval task published it's finely annotated English data, many similar fine-grained data sets for both large and small languages were published (German: Klinger and Cimiano 2014; Spanish, English, French, German, Dutch and Italian: Agerri et al. 2013).

2.2.1 Multilevel analyses

The iterative process of producing even finer-detailed sets directly from larger, coarser ones opens the possibility of multilevel sentiment analysis. A multilevel analysis combines fine-grained data with document level annotations in hopes to provide even more meaningful information to the system through auxiliary tasks.

An example of this is the *Interactive Multitask Network* (IMN) introduced by He et al. 2019. Here, document level annotations are learned as an auxiliary task, which slightly improved model performance on a variant of the fine-grained analysis task. The IMN experiment is one of the leading inspirations for this thesis, and will be discussed more in depth in Section 3.5.3.

Another example of a multilevel system could batch sentences from the same root document together, learning from all sentences at once, comparing these aggregate extraction to the overall document classification. This can be useful both for checking data set consistency, but also for more comprehensive linguistic research around how sentiment arises.

Document level modules were originally planned to be incorporated in the initial architectures of the our thesis. However, a multi-level analysis was down-prioritized early on in during experimentation, in order to focus more on models reliant only on fine-grained data.

2.2.2 Inter-annotator agreements

Producing fine-grained sets is usually expensive because of the naturally intricate data, not to mention the consensus required between annotators. Some projects that introduce such data sets additionally include clear guidelines the annotators were expected to follow, like in the *Fine-Grained Norwegian Review Corpus* (Øvrelid et al. 2020). These guidelines help bring forth an inter-annotator agreement on which expressions should be given which labels.

Many projects use multiple annotators to label the same portions of the set to ensure objectivity. In NoReC$_{fine}$, discrepancies between annotated labels were settled democratically through votes during inter-annotator meetings (Øvrelid et al. 2020). These measures help to normalize annotations and maintain consistency throughout the entire data set.

An important part these annotation guidelines, along with any fine-grained analysis task, is a clearly defined vocabulary for the components of the data set. The terminology used in this project may vary slightly from other projects, as many authors adapt their own interpretations of the fine-grained sentiment analysis task. The next sections cover this terminology according to how our data set is defined, along with the data set's annotation scheme.

2.2.3 Terminology

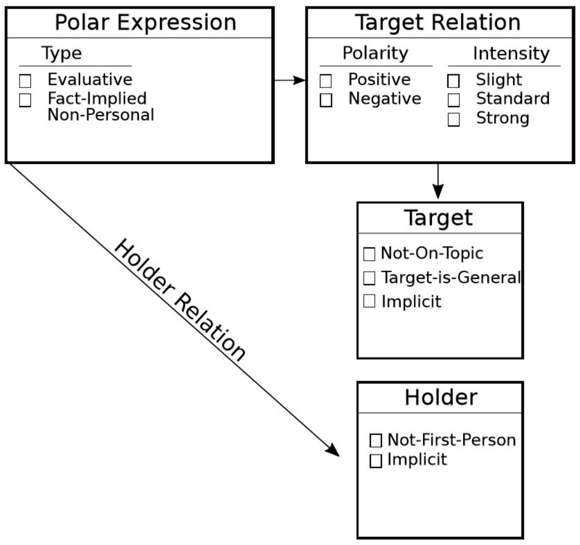


Figure 2.1: Annotation scheme for NoReC$_{fine}$ data set (Øvrelid et al. 2020)

A problem that often arises when combing through articles on "fine-grained sentiment analysis" is the variations in terminology used to describe the details of the data sets used for experimentation. In the scope of this project, **fine-grained sentiment analysis** refers to experiments focusing specifically on extracting individual opinions (including targets, holders, polarity, and intensity) from a given text.

Building off work done for the development of the NoReC$_{fnie}$ data set (Øvrelid et al. 2020), the core elements of an opinion are defined here as:

target	where the opinion is directed
holder	source or entity holding the opinion
polar expression	set of words carrying polarity of the opinion
polarity	positive or negative classification of the opinion
intensity	strength of the polarity

The **targets** can appear in various forms. In our data set, boolean flags labelling whether a target is *implicit*, *off-topic*, or *general* are provided. Each of these labels are independent of each other, meaning multiple flags can be true for a target. For example, a target can be off-topic and general, while another is implicit and general, and so on.

Polar **expressions** are defined as being either evaluative or fact-implied non-personal. Here, *evaluative* means the sentence is sentiment bearing in the form of a personal opinion, like *"I like the color of this model."* Alternatively, *fact-implied non-personal* expressions still carry sentiment, but are rather completely objective, as in, "*The screen broke on the second day of use.*"

Polarity along with **intensity** provide information on the target relation, i.e. how the polar expression relates to the target. While polarity and intensity is included in the NoReC$_{fine}$ data set, only polarity was used in this project. This decision was merely for development simplicity. Intensity classifications can be added at a later time for more research around this particular annotation.

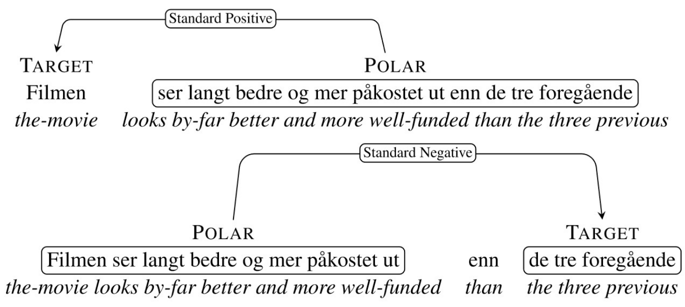
Holders also have a few boolean tags that could provide more information to the model. Whether a holder is in *first person* or not could be useful for expression scope extraction. Additionally, if a holder is *implicitly* assumed or *explicitly* stated in the sentence is labelled, especially helpful when polarity classification is either a sub- or auxiliary task, according to Jeremy Barnes, Øvrelid and E. Velldal 2021.

The extreme detail of these annotations help assign the correct polarity to the correct target, proving especially helpful in the cases where multiple opinions occur in the same sentence. For example, the following translated sample from NoReC$_{fine}$

"Some of this is computer graphics, but most of it looks like stunts."

is annotated as having a slightly negative opinion (*Some of this is computer graphics*), while also having a slightly positive opinion (*most of it looks like stunts*). The target is not explicitly mentioned in the sentence, and is therefore left blank in the annotations. Lastly, the holder is assumed to be the author themselves, which is labeled in the annotations as mentioned above.

Figure 2.1 visualizes an abstraction of the annotation scheme used in the NoReC$_{fine}$ data set and how these concepts are related to each-other, while Figure 2.2 shows another example from the paper with both positive and negative semantic polarities.



Standard Positive
TARGET
Filmen
the-movie
POLAR
ser langt bedre og mer påkostet ut enn de tre foregående
looks by-far better and more well-funded than the three previous
Standard Negative
POLAR
Filmen ser langt bedre og mer påkostet ut
the-movie looks by-far better and more well-funded
enn
than
TARGET
de tre foregående
the three previous

Figure 2.2: An example of a comparative sentence, with both negative and positive sentiment, from NoReC$_{fine}$

2.2.4 Datasets

As previously mentioned, experimentation in this thesis makes use of the NoReC$_{fine}$ data set, produced iteratively by the Language Technology Group at the University of Oslo (Full: E. Velldal et al. 2018; Eval: Mæhlum et al. 2019; Fine: Øvrelid et al. 2020).

These data originated from the Sentiment Analysis of Norwegian Texts (SANT) project, a joint effort of the Norwegian Broadcast Corporation (NRK)[8], Schibsted Media [9], AllerMedia[10], and the University of Oslo. The goal of SANT was to elevate the standard of Norwegian NLP tasks (specifically sentiment analysis) by gathering and annotating roughly 35,000 full-text published reviews, covering a range of different domains, including literature, movies, video games, restaurants, music and theater, among others.

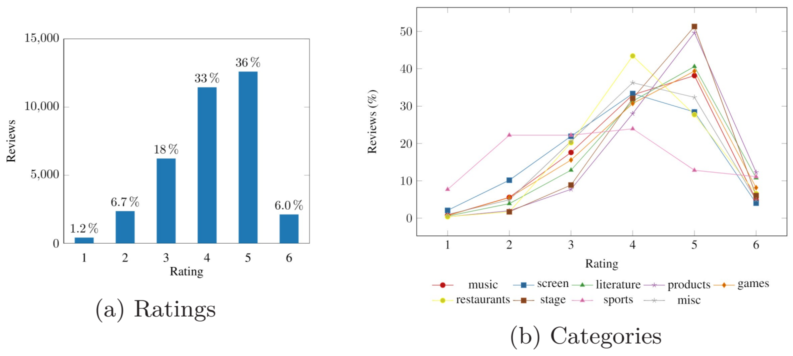


Figure 2.3: Distributions of ratings for the 35,000 reviews making up the NoReC data set. The left side shows the total proportions between the 6 possible ratings. The right shows how each category of reviews are distributed among the 6 possible ratings.

[8] https://nrk.no
[9] https://schibsted.com
[10] https://aller.no

2.2.4.1 Document level

The original reviews in the NoReC dataset came annotated with numerical scores on a scale of 1-6. This metric resembled the 6-pip dice ratings usually presented reviews found in Norwegian media. The distribution between scores seen in Figure 2.3 shows that reviews typically lean toward mid-high scores, but with a sharp decline at the best score 6. Some detailed analysis of scores per category are presented in the paper (E. Velldal et al. 2018).

Although the raw states of these data consisted of multiple different formats, the final data set is provided in CoNLL-U format, with relevant metadata for each document in JSON format. This standardization promotes accessibility of the data for later experiments, with a flexibility that encourages further detailing of annotations.

2.2.4.2 Sentence level

The first iteration of annotations on this corpora produced NoReC$_{eval}$ (Mæhlum et al. 2019). This data set introduced sentence level labels of various classifications.

As previously mentioned, annotations at sentence level typically first filter the objective sentences from the subjective before classifying polarity. NoReC$_{eval}$ also introduces the third label category, *fact-implied non-personal*, to distinguish between polar opinions versus polarity carrying facts.

The paper also provided baseline experimentation on previous state-of-the-art architectures (both neural and traditional). In this out-of-the-box experimenting, the three neural models showed the most promising results, with a single-layer convolutional network giving the top performance.

2.2.4.3 Fine-grained level

The next iteration of NoReC annotations produced NoReC$_{fine}$, the data set to be used in this project (Øvrelid et al. 2020). Here, opinions in both the evaluative and fact-implied sentences from NoReC$_{eval}$ were further annotated, specifically labeling their respective targets, holder, expression scope, polarity, and intensity.

	Train	Dev.	Test	Total	Avg. len
Sents.	5915	1151	895	7961	16.9
Holders	584	76	75	735	1.1
Targets	4458	832	709	5999	2.0
Polar exp.	5659	1050	872	7581	4.6

Table 2.1: Count statistics of NoReC$_{fine}$ data set (Øvrelid et al. 2020)

In total, over 7,500 polar expressions are labelled in the 8,000 sentences included in the set. Some count statistics for the annotated categories are presented in Table 2.1. Here we see the set contains about 10 times more expressions than holders, yet only about a fifth more than targets. However, looking at the average span of target versus expressions, it can be deduced that there are more sentences with targets in them than expressions. In total, we could expect slightly less then half our training data to contain targets, less than a quarter of them to contain expression, and about a tenth of them to contain holder scopes. This means our models will be taught both how to identify tokens expected to be labelled, but also how to distinguish whether a label is expected to be present in a given sentence or not.

As will be discussed later in Section 4.1, our preprocessing labels polarities across the same scope as targets, which is why polarity was excluded from Table 2.1.

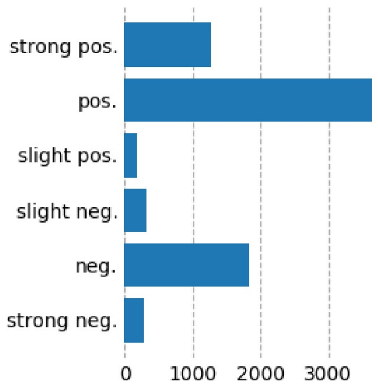


Figure 2.4: Distribution of polar intensities in NoReC$_{fine}$

The distribution of polar intensity shown in Figure 2.4 highlight the slight imbalanced ratio between positive and negative reviews in the data, favoring positivity. This class imbalance should be taken into consideration when defining a loss function, as to mitigate a model from favoring the more represented label. A commonality between both polarities is that neutral intensities seem preferable over strong or weak ones. This major imbalance also contributed to the initial exclusion of intensities from this experiment.

The data is split between three sets, train, dev, and test, each stored as their respective JSON files on the NoReC$_{fine}$ project repository[11]. An entry is a dictionary containing a `sent_id` string, a `text` string, and an `opinions` list. The `sent_id` contains a unique label, denoting both sentence number and document id. Under `text`, you find the actual sentence as a string. In the `opinions` list is where the fine-grained annotations are labeled.

Here, source, target, expression, polarity, intensity, sentence type, negation, and other boolean attributes are labelled. Sources, targets, and expressions are lists of tuples.

The tuples pairs contain lists of string(s) representing the tokens this attribute applies for. The list at the first index represents the actual tokens written out word by word.from the sentence given this label. The second list contains the character-wise index scope of the attribute in the original sentence, written in the format `"<start>:<stop>"` in the same way a Python list is indexed[12].

[11]https://github.com/ltgoslo/norec_fine

[12]Start at index 0 with the last index excluded from the scope

The tuples had to pairs of lists and not just strings, since any attribute can be split by non- or other-attribute tokens in a given input. For an example take the following sentence:

> *"Rollen som Wulff kan meget vel bli Jakob Oftebros internasjonale gjennombrudd."*

The expression of this sentence, *"Rollen som Wulff kan meget vel bli"* and *"internasjonale gjennombrudd"*, is split by the target, *"Jakob Oftebros"*.

2.3 Text representation

A necessary component of any language processing task is representing the input text in a way that makes mathematical calculations possible. Numerical representations of text allow computational models to find relationships and patterns within the input language. These representations often take the form of vectors, varying in size depending on mapping technique. This section will provides a brief history on these representation techniques, eventually leading in to the more modern representation techniques used in this project.

2.3.1 Bag-of-words

One of the simplest representation techniques often used as a baseline for semantic analysis at document level is the *bag-of-words* method (Harris 1954). As Harris points out, while positional information between words can be important for determining a document's structure, there still lies valuable information in comparing the distributions of co-occurrences between tokens.

"Bag-of-words" vectors are word count mappings of a tokenized input sentence to some N-dimensional vector space; here N is the total number of unique words, w_i, found in the entire training set, (w_i for $i \in [1, N]$). A new input sequence of length l is then represented by a one-hot encoded vector of size N, where at most l elements will have a non-zero value[13]. After training, the full vocabulary representation can be thought of as a matrix of count vectors of size $N \times k$, where k is the number of input sentences the model is trained on.

Since the full vocabulary of a large data set can be quite big, a model's vocabulary size is often limited to the top n most occurring words observed in the data, to save from unnecessary resource usage. Unfortunately, count vectors alone can only capture so much information. Their sparse representations make them victim of the "curse of dimensionality," meaning as vocabulary size increases, the amount of resources needed to compute predictions skyrocket. Limitations on the vocabulary size can help speed up otherwise slow experiments, but can potentially also throw away useful information while holding on to useless information.

2.3.2 Term frequency - inverse document frequency

Certain statistics were eventually developed to combat the dimensionality curse. These helped capture how much a particular word uniquely contributed to a

[13]The specification *at most* is used here in case a word is repeated in a sentence.

document more efficiently than pure count totals. TF-IDF, which stands for *term frequency - inverse document frequency*, was one of such metrics (Salton 1991).

As the name suggests, TF-IDF is the ratio between the occurrence count of a token in a single input and the total number of documents that term has been used in. Words often used in very different contexts, commonly referred to as *stop words*, are naturally filtered out of TF-IDF representations. This allows for more concentrated analysis on the words unique to a document, which presumably contain more context for that individual document.

2.3.3 Point-wise mutual information

Another statistic that focuses more on similarity between two words is the point-wise mutual information (PMI) metric (Church and Hanks 1989).

This statistic is the probability two tokens occurring in the same input over the combined probability of the two tokens occurring individually. In other words, PMI in a measurement of the independence between 2 tokens. The closer this ratio is to 1, the more independent are the words being compared, thus the more context bearing each word is assumed to be.

Turney 2002 was one the earliest experiments to incorporate this statistic in a document level sentiment analysis setting. The findings from this experiment show that PMI statistics are most useful for correctly classifying negative reviews. This sparked research questions about the importance of negation in internal representations, a topic that was explored for the NoReC data set in (Barnes, Velldal and Øvrelid 2020), and therefore will not be explored in major detail in this project. While this the negation analysis on the Norwegian data might not be considered an exhaustive search, meaning more research can still be done on the topic, our thesis chose to focus on other areas of interest than negation.

2.3.4 Word embeddings

As data set sizes grew in tandem with model complexity, even smart statistics like TF-IDF and PMI couldn't save a system from the curse of dimensionality. A more efficient representation of the count vectors needed to be developed.

For a very large vocabulary, count vectors usually were extremely sparse, meaning most words in the vocabulary didn't show up in more than a few inputs. *Dense count vectors* reduced the size of a vocabulary's count matrix to lower dimensional spaces.

Simply explained, these reductions preserved the (general) direction a count vector was pointing in within a smaller dimensional matrix. Some examples of dense word embeddings were word2vec (Mikolov et al. 2013), GloVe (Shi and Z. Liu 2014), and fastText (Bojanowski et al. 2017).

Researchers quickly found many benefits of such techniques. They proved better at generalizing than sparse vectors, since vectors of one word often pointed in a direction close to that where other words used in similar contexts were pointing. Also semantic similarities were easier to capture, allowing for relational inference through basic vector calculations, as shown in Figure 2.5.

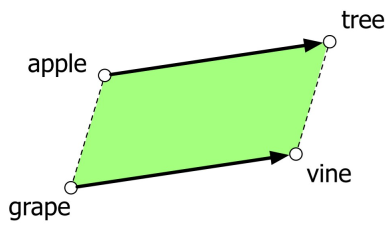


Figure 2.5: Semantic inference through vector calculations produced from word embeddings (Jurafsky and J. Martin 2020)

These early techniques served as rudimentary building blocks for representing text in a prediction system. While rich with information on the given data, it can be argued that such determinant statistics only start to generalize well when the size of the training are very large, meaning slow training. Additionally, words that were not seen during training will be completely disregarded in downstream models, because there is no way to add these to the vocabularies after the fact. To get more out of less data, researchers in the last decade have worked to replace fixed dense representations with dynamic ones, often incorporating neural networks into their text representation mechanisms.

2.4 Embeddings from Transformers

Until about 2018, most fields of NLP still used different variations of static embeddings to internally represent text as vectors, like word2vec (Mikolov et al. 2013), GloVe (Shi and Z. Liu 2014), and fastText (Bojanowski et al. 2017). While their algorithms became more and more complex, such count-based static embeddings limit model vocabularies to tokens observed during training [14].

The mapping of each token to a single value completely disregards homonyms, or words that share same spelling, but have completely different meanings like *bat*[15] or *bank*[16]. Additionally, little to no structural information of the *context* the word was used in is included in a bag-of-words embedding, meaning the embedding would not change if the token occurs in different settings. As can be assumed, context is important when determining the sentiment polarity of an input.

2.4.1 Contextual embeddings

The rigidity of static embeddings often disregard useful information in a token's surroundings, or the input sequence the token is used in. *Context embeddings* do the opposite by adding contextual information to the static embeddings. Language models like ELMo (Peters et al. 2018) and BERT (Devlin et al. 2019) do this with help from modern machine learning concepts like recurrent neural networks and transformers.

[14]The exception here is fastText, which builds tokens from word parts (prefix, root, suffix), not full words. This opened up for some flexibility toward new words, as long as the new word has roots or sub-words similar to other known tokens in the vocabulary.

[15]The flying *bat*... (A: was thrown by a baseball player; B: usually hunts at night).

[16]I'll go to the *bank* ...(A: to test the water levels in the river; B: to deposit my check.)

ELMo was the first of such language models that introduced a novel way of representing both "complex characteristics of word use (e.g. syntax and semantics), and how their uses vary across linguistic contexts (e.g. to model polysemy)" (Peters et al. 2018). In this setup, a bidirectional long short-term recurrent neural network helps encode every token with information from both it's preceding and succeeding neighbor. The success of this architecture inspired more research in contextual representations.

An even newer example here is BERT, introduced by Devlin et al. 2019. In this architecture, every token is represented by the underlying static word vector together with information from the *all* of the other embeddings in the rest of the sentence. This coupling is done through a process called *attention*, the cornerstone of the Transformer architecture.

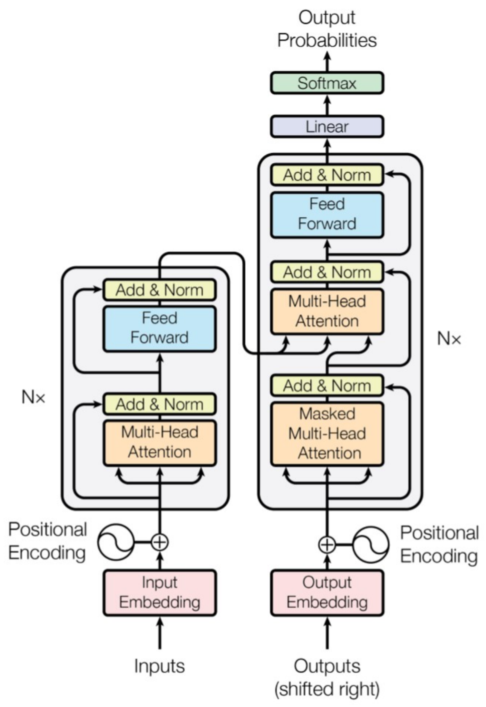


Figure 2.6: Transformer architecture as presented in Vaswani et al. 2017

2.4.2 Transformers

In the paper *Attention is all you need*, Vaswani et al. 2017 introduced the Transformer architecture which revolutionized natural language processing research. A Transformer is an encoder-decoder system that applies self-attention to an encoder embedding and a decoder embedding.

The decoder input is usually positionally shifted to the right to ensure the model learns to predict the next word, instead of just copying the input to the decoder. The outputs of both self-attention layer are then concatenated and fed through a cross-attention transformation, to compare the context observed from both parts.

In general, the difference between cross attention and self attention is the former compares two different inputs (of same length) through an attention matrix, while the latter uses a single input for both variables to produce the attention matrix[17]. Figure 2.6 visualizes the components of this architecture.

Transformers introduced not only a new way of representing input with information from it's neighbors, but also provided the field of NLP with a robust, new component that naturally allowed models to learn even more from given inputs.[18]. This thesis will make use of Transformers in the form of BERT, which will be explained in further detail in 2.4.4. Before we can dive into BERT, we first need to present the inner workings of an attention block.

2.4.3 Attention

Attention refers to a cross-multiplication technique, first introduced by Bahdanau, Cho and Bengio 2016. It was originally aimed to aid in the sequence-to-sequence task of translation, a task machines often struggle with due to variations in sentence structure, like length and word-type order. As anyone who speaks two languages will tell, grammar rules often jumble word order between languages and homonyms only perfectly translate sometimes.

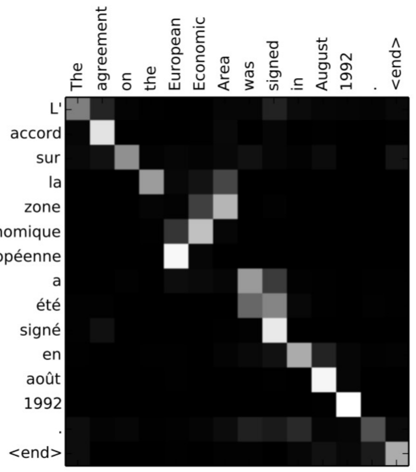


Figure 2.7: Example of attention to solve a translation task.

By comparing one and the same sentence in two different languages through a matrix, a machine could learn correct associative mappings between the corresponding translated words. This potentially helps translation systems better learn the expected output order for a given input sequence.

Researchers quickly found out that the contextual information obtained from attention could be just as beneficial for other tasks when only focusing on a single language. For example, if the work "bank" is used in the same sentence as the word "river", there's is a greater chance the intended interpretation of "bank" should be the natural bend in flowing water, and not the place people keep their money.

[17]More on this in Section 2.4.3.

[18]*Attention is all you need* was cited more than 25,000 times the first 3 years after being published!

2.4.3.1 The attention formula

The math behind an attention layer consists of three linear transformations performed on an input of size:

$$[\text{sequence length}, \text{embedding dimensions}]$$

These three transformations are often called *queries*, *keys*, and *values*, depicted below as $Q, K, \& \ V$ respectively:

$$\text{Output} = softmax(Q(\text{Input}) \times K(\text{Input})^T) \times V(\text{Input}) \tag{2.1}$$

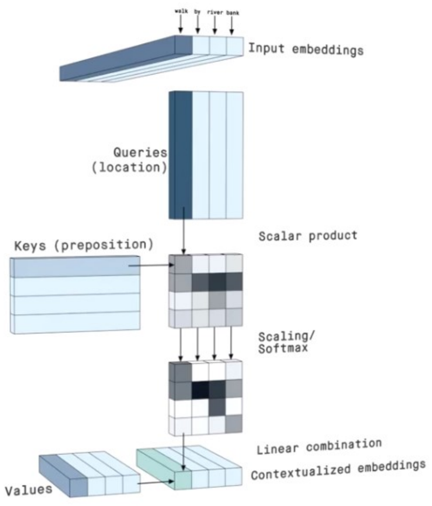


Figure 2.8: 3D illustration of a transformer (Peltarion 2020)

As Equation 2.1 and Figure 2.8 show, inputs are linearly transformed into the queries and keys before being cross multiplied. The resulting matrix is then fed through a softmax activation, the result of which then goes on to be matrix multiplied with the last linear transformation of the inputs, otherwise called the values. The final output (for a single sequence) is the embeddings for each token in the sequence of the same size the input was:

$$[\text{sequence length}, \text{embedding dimensions}]$$

The two main types of attention often discussed in NLP literature are cross-attention and self-attention. While both follow a structure similar to Equation 2.1, the inputs vary slightly between the two. Cross-attention compares two different sentences to each other, while self-attention compares a sentence to itself, picking up on the heavily correlated words in the sentence[19].

The difference between attention based representations and static embeddings is that each contextual embedding now contains not only information for that singular token, but also a blend of the other embeddings in the sequence. This opens for the possibility of ambiguous words to be disambiguated as a direct affect of the context the word is used in.

[19]An example here of heavily correlated words are "river" and "bank" in the sentence "The bank grew deeper as the water-level in the river increased".

2.4.4 BERT

Just as Vaswani et al. did with their introduction of Transformers, Devlin et al. 2019 made waves over the entire field of NLP by introducing BERT, the **B**idirectional **E**ncoder **R**epresentations from **T**ransformers. BERT is in short an expansion of the Transformer architecture that has proven to generalize well across multiple NLP tasks (Word sense disambiguation:Wiedemann et al. 2019; Sentence SA:Merchant et al. 2020).

The 12 Transformer-like encoder layers making up the first layers of a BERT head encode tokens of an input sequence according to their respective embedding representation as well as incorporating contextual information of the neighboring tokens. The decoder component of a BERT model changes depending on the downstream task in which the contextual embedder is applied to.

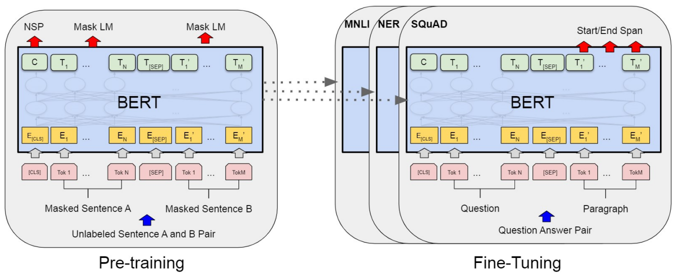


Figure 2.9: BERT training schema

During pre-training, a BERT setup applies a bidirectional RNN to the output of the final attention layer looking to solve two main pre-training tasks: *next sentence prediction* and *masked language modeling*.

Next sentence prediction is a single classification of whether the second sentence of a two-sentence input logically follows the first. Here, input sentences are paired with original natural predecessors from their data set for roughly half the inputs. The others are paired with some random other sentence in the set. This is expected to teach a BERT more a greater context scope, which can make it more likely to correctly classify context of future inputs.

Masked language modeling is the act of reconstructing a corrupted input, where some tokens have been purposely removed. In other words, BERT learns to replace missing words from a sentence at the time as it's learning if sentence B follows sentence A. While starting off with only a few masked tokens, this task grows more and more difficult as training continues, masking more and more tokens from new input sequences.

These two tasks can't be assumed to cover all nuances of natural language, even though BERT-based models still managed to generalize better than any previous language models. The original BERT architecture was presented with state-of-the-art performance on 11 subtasks within NLP at the time of introduction! Specifically, important for this thesis, BERT proved beneficial for sentiment analysis on both sentence level and token level tasks.

This suddenly jump in performance helped establish BERT as a necessity of most modern NLP experiments, and even created a new subdomain of NLP, BERTology (Rogers, Kovaleva and Rumshisky 2020), looking into exactly the details around what BERT models can and can't do. Much research in this field is however still being developed, as BERT models are popping up left and right for languages other than English (German: Scheible et al. 2020, French: L. Martin et al. 2020). NorBERT, the Norwegian flavor of BERT (Kutuzov et al. 2021), will be used in this project, making this in-depth sentiment analysis experiment using context embeddings the first of it's kind for Norwegian.

2.4.5 NorBERT

Developed and maintained by the Language Technology Group at the University of Oslo, NorBERT is one of the leading BERT architectures built for Norwegian (Kutuzov et al. 2021).

The first version of NorBERT, released in spring 2021, was relatively small compared to other languages, being only trained on roughly 1.9 billion word tokens. However, it was ground-breaking in that it allowed for preliminary experiments using contextual embedders for the Norwegian language.

A massive size increase came with the second version, released in the spring of 2022, and trained on roughly 15 billion word tokens from close to 1 billion sentences. The size increase was largely due to the incorporation of the Norwegian Colossal Corpus and the Norwegian part of the C4 web-crawled corpus (NCC: Kummervold et al. 2021; C4 web:Xue et al. 2021).

Both NorBERTs were trained according to the pre-training and fine-tuning setup, as well as the same model configurations, as defined in the original BERT paper[20] (Devlin et al. 2019). The vocabularies of both NorBERT 1 and NorBERT 2 consist of 30,000 and 50,000 BERT WordPiece formatted tokens, respectively. The former took approximately 3 weeks to train, while the latter took roughly 4 weeks[21].

The NorBERT models can be easily integrated in new experiments via the HuggingFace API[22]. Also referred to as the `transformers`-library, this API introduced a standard interface for building new architectures off pre-trained models for a variety of machine learning tasks. Both PyTorch and Tensorflow frameworks are supported, with good documentation provided for each.

Our project makes use of NorBERT 2 loaded into our models via the HuggingFace interface.

2.4.5.1 NorBERT's rival

The close competitor to NorBERT is the NB-BERT, developed by the National Library of Norway (Kummervold et al. 2021). Built around the same time as NorBERT 1, this model is trained on close to 7 billion words spread across roughly 21 million documents. The larger size made the NB-BERT model slightly more attractive during the first year of experimentation after BERT models for Norwegian first were introduced. It also was the leading inspiration to build an even bigger and better NorBERT version by the Language Technology Group in Oslo (who doesn't love some friendly competition?)

[20] 12 layers and hidden size 768

[21] More details about their respective training workflows can be found at: http://wiki.nlpl.eu/Vectors/norlm/norbert

[22] https://huggingface.co/

NB-BERT is also loadable via the HuggingFace API, meaning that the models and architectures developed in our experiment can easily be tested on this rival, if desired. The GitHub repository with the source code used to train NB-BERT shows recent updates, showing that this model is currently still being maintained. Maybe a new version of NB-BERT will be released in the later half of 2022? Only time will tell.

2.4.6 Is BERT necessary?

The introduction of BERT sparked a frenzy of new experiments in many fields within NLP. However, these smarter models increase internal parameter count, thus requiring much larger compute resources to be run. This raises the question of exactly how necessary BERT architectures are for particular tasks.

As mentioned before, experiments on BERTology (Rogers, Kovaleva and Rumshisky 2020) look into how BERT works, and what it's limitations are. Another interesting experiment, presented in the Third IN5550 Teaching Workshop on Neural Natural Language Processing, looked specifically into the affects of BERT on fine-grained analysis (Workshop: J. Barnes et al. 2021; Paper: Walker 2021).

In their paper cleverly titled *To BERT or not to BERT*, Walker 2021 compares the performances of static embeddings against contextual ones applied to a variant of a FGSA task. They also test different downstream architectural components, from simple feed-forward layers to more complex recurrent structures like LSTMs[23].

The findings show that BERT embeddings boost performance on the target-polarity classification task greatly. We would argue that some of the downstream architectural variants would only making learning more difficult, so these are be reevaluated in our experimentation. However, the major increase in performance obtained by a simple BERT head with a single linear output layer tells us that BERT embeddings in FGSA models are greatly beneficial.

Our project will use these finding to further evaluate downstream architectures on an FGSA task. We assume the results presented in *To BERT or not to BERT* about Norwegian contextual embeddings outperforming static as true, and disregard static embeddings from our experimentation. It is important to note this assumption is not necessarily a hard fact. Future experiments may uncover than static embeddings actually can be tuned or set up to compete more closely with contextual embeddings. In such a case, our experiment should be rerun, but including the new configurations for the static embeddings.

2.5 Simple multitask-learners

A natural assumption that can be made after seeing the benefits from BERT is that added model complexity provides more space for models to learn weaker nuances in a task's signal, thus increasing performance. There are a number of ways complexity can be increased for any given architecture, including increasing number of layers or size of specific components, adding multiple components with varying sizes for a single task, providing increased number of attributes in the input data, et cetera.

[23]Recurrent components will be discussed in further detail in Section 3.4

However, as previously mentioned, the curse of dimensionality will always need to be considered when increasing complexity. Irrelevant increases in complexity will slow down training, yet provide little to no increases in performance. So complexity increases alone can not be the general goal of new experimentation, smarter complexity is needed.

2.5.1 Interacting subtasks

One common way of implementing smarter complexity in language processing models is by breaking complex label predictions in simpler subtask estimations. Such models are often referred to as multitask-learners. By building unique components for each subtask in the expected label output, multitask-learning models can learning more details of a particular subtask with less noise from the other subtask's labels.

The smartness of such models can be increased by allowing some information learning in one of the tasks to be available to the other subtasks. Minghao Hu et al. 2019 present some of the most common methods of information sharing between similar subtask architectures: namely *collapsed*, *pipeline*, and *joint* models.

Collapsed models join the subtask labels into a single label, just as was done for the previous experiments mentioned in the sections above. This type of setup was included in the experiment done by Minghao Hu et al. 2019 in order as a baseline to compare how the other multitask set ups performed in the same setting as previous set ups.

Pipeline model solve one subtask at a time, passing the information learned from the first subtask to the next. While each subtask produced their own output predictions for their respective label, sending this information to the next subtask saves the model from having to relearn some of the same information over again in this next component.

Joint models assume similar architectures between each of the subtasks trying to be solved. Here, models solve each subtask in parallel, but restrict learning of a subtask's specific component parameters based on the updates made from the other subtasks. Basically, each subtask has it's own architectural components, but of the exact same shape and form as the other subtasks' components. Applying a loss between each subtask component restricts learning between all the components If a certain subtask tries updating it's parameters too much compared the other subtasks, or in a completely different direction, the model will be penalized more.

There are three types of sharing that can be done for these setups: *hard*, *soft*, and *no-sharing*. *Hard sharing* means this share-loss is applies to all layers of the subtask components, allowing only the final output layer to learn freely. *Soft sharing* only applies this share-loss to the upstream components, like the BERT-head and maybe a few more "shared" layers, allowing a handful of layers before the final output layers to also learn freely. Finally, *no-share* setups do not share any information between subtask architectures, but allows the model to learn each subtask on its own, only concatenating the final outputs for true label comparison.

2.5.2 Multitask-learning in Norwegian

In the same workshop where *To BERT or not to BERT* was presented (WNNLP: J. Barnes et al. 2021), another team tackled the FGSA task variant, taking inspiration from the three multitask-learning setups presented in 2.5.1. The team of Pereira, Halvorsen and Guren 2021 presented their paper *Applying Multitask Learning to Targeted Sentiment Analysis using Transformer-Based Models*, which was awarded the Best Paper for the WNNLP 2012 workshop.

Using a simplified variant of the NoReC$_{fine}$ data set, the team compared hard, soft, and no-share joint techniques, in addition to collapsed and pipeline methods mentioned previously. In their experiment, found that the pipeline approach to a multitask-learner gave the best results on this target-polarity specific FGSA task. However, they mentioned that time constraints lead to little fine-tuning of their joint models, which could have been the reason behind limited performance of such set ups. Additionally, none of the multitask-learners built in that project were able to beat the simple BERT baseline with linear output layers.

Our experimentation will take the findings from *Applying Multitask Learning to Targeted Sentiment Analysis* one step further, digging deeper into the most promising configurations of these joint setup models, in an attempt to find a novel multitask set up that out performs the simple BERT to linear model. Namely, a soft sharing model where the BERT embedder along with some shared neural components are forced to learn from all will be tested in our experiment. Additionally, some variants of pipeline set ups are tested, where information from one subtask is passed directly into the next. More on specific model outlines in Section 3.5

2.6 Interactive networks

Originally presented as a learning exercise for NLP students, the FGSA task of the WNNLP workshop (J. Barnes et al. 2021) was reduced in complexity compared to the ramifications of how we defined a fine-grained sentiment analysis study. The findings of the experiments accepted in the proceedings of the workshop can serve as a initial motivation for directions of further exploration.

However, in order to compare our novel architectures fairly against other models, we need to find baselines built for solving an FGSA task closer to how we've defined it. The two architectures chosen for this comparison make use of interactive multitasking, in hopes to capture these more abstract relations between subtasks. This section presents those two networks as our baseline models.

These networks can be seen as more complex multitask-learners, where specific information passing is allowed, based on the benefits their information has on the other subtasks. Keeping the three multitask-learning set ups presented in Section 2.5.1 in mind, the reader will see how both these baselines make use of smarter subtask interactions.

2.6.1 IMN

An important modern experiment on interactive networks relative to fine-grained sentiment analysis was presented by He et al. 2019. In their paper *Interactive Multi- task Learning Network* (IMN), He et al. solve the two main tasks of targeted sentiment analysis, (i) target extraction and (ii) sentiment classification, as individual subsequent tasks with help from a message passing mechanism.

Simply explained, the IMN network maps inputs to a shared feature space. The two subtasks (i) and (ii) learn to solve their respective tasks using the shared latent vectors from the feature space, as most multitask setups do. A pipeline like method is used to directly share the predicted targets with the polarity classifications, so (ii) is actually learned from a concatenation of the shared vectors and the output from (i).

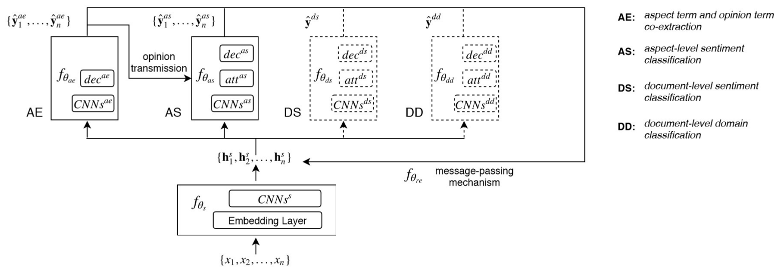

$\{\hat{\mathbf{y}}_1^{ae},\ldots,\hat{\mathbf{y}}_n^{ae}\}$
$\{\hat{\mathbf{y}}_1^{as},\ldots,\hat{\mathbf{y}}_n^{as}\}$
$\hat{\mathbf{y}}^{ds}$
$\hat{\mathbf{y}}^{dd}$
opinion transmission
$f_{\theta_{ae}}$ dec^{ae} $CNNs^{ae}$
$f_{\theta_{as}}$ dec^{as} att^{as} $CNNs^{as}$
$f_{\theta_{ds}}$ dec^{ds} att^{ds} $CNNs^{ds}$
$f_{\theta_{dd}}$ dec^{dd} att^{dd} $CNNs^{dd}$
AE
AS
DS
DD
$\{\mathbf{h}_1^s,\mathbf{h}_2^s,\ldots,\mathbf{h}_n^s\}$
$f_{\theta_{re}}$ message-passing mechanism
f_{θ_s} $CNNs^s$
Embedding Layer
$\{x_1,x_2,\ldots,x_n\}$
AE: aspect term and opinion term co-extraction
AS: aspect-level sentiment classification
DS: document-level sentiment classification
DD: document-level domain classification

Figure 2.10: The IMN network as presented by He et al. 2019

Novelty of IMN comes from the updating of these shared latent vectors with information learned in each subtask, only to be fed back into the subtasks for a predetermined number of iterations, n. This looped-learning architecture can potentially reveal discreet patterns that a low-level analysis would otherwise fail to pick up on.

As an auxiliary task, the IMN architecture also learns from document level annotations. The subtasks are trained separately from the auxiliary tasks, excluding interaction-iterations for the auxiliary document classification, to cut back on compute time. Unfortunately, in the standard setup, the multilevel IMN variants performed worse than those excluding this data. Lower performance could signal that these data are not beneficial for fine-grained analysis, but could also be due to suboptimal learning for this auxiliary task.

Multilevel analysis is just outside the scope of this thesis, however future work on our models could explore derivatives of this feature, to see if multilevel analysis could prove beneficial in an interactive multitask network for Norwegian.

2.6.2 RACL

Another interactive network important to this thesis is the *Relation-Aware Collaborative Learning Network* (RACL) presented by Chen and Qian 2020. Like the IMN architecture, RACL attempts to share information between subtask components. However, instead of iterations, RACL makes use of stacked RACL layers, similar to BERT. Deeper layers can learn to pick up deeper relations, while still preserving and optimizing for shallow patterns further upstream.

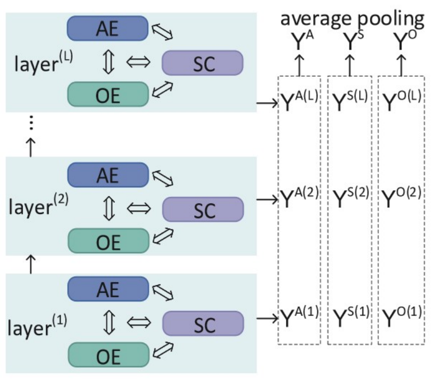

average pooling
Y^A Y^S Y^O
layer$^{(L)}$
layer$^{(2)}$
layer$^{(1)}$
AE
OE
SC
$Y^{A(L)}$ $Y^{S(L)}$ $Y^{O(L)}$
$Y^{A(2)}$ $Y^{S(2)}$ $Y^{O(2)}$
$Y^{A(1)}$ $Y^{S(1)}$ $Y^{O(1)}$

Figure 2.11: RACL relations as presented by Chen and Qian 2020

RACL was built on the argument that not all relationship types between subtasks were taken into consideration in previous experiments on interactive multitask networks for fine-grained sentiment analysis. In particular, no experiments had yet proposed an online method to share information from target extraction with polar expression extraction. Figure 2.11 shows an abstraction of how relations between target extraction (AE), expression extraction (OE), and polarity classification (SC) occurs for multiple stacked layers of RACL.

While some papers argued different specific interactions between subtasks as being most relevant for fine-grained sentiment analysis, RACL attempts to shared all the available information between the three subtasks: target extraction, polar expression extraction, and sentiment classification.

The NoReC$_{fine}$ data set comes with even more detailed annotations than used in the RACL experiment, including scope of the opinion holder (if present), intensity of polarity, generality of opinion, to name a few[24]. Taking inspiration from RACL, along with the findings from *Flaunt It!* (Jeremy Barnes, Øvrelid and E. Velldal 2021), this thesis will also test if the holder annotations can improve a multitask-learning model, by providing yet even more known information to the model as the data set allows.

2.6.3 Comparing IMN and RACL

As a baseline comparison, the RACL paper considers a derivate of the IMN architecture where a BERT head is used instead of the standard domain-specific vectors. This was done to present a fair comparison between architectures, since the best standard RACL setup already included these performance enhancing contextual embeddings. Improvements obtained from the RACL-BERT model outscored the IMN-BERT setup across metrics, as shown in Table 2.2. While IMN-BERT performed better than many of the other baseline models included in the RACL experiment, the RACL models proved superior on all three data sets tested, both for task-specific and full F_1 scores.

It should be noted that some experiments aren't always fair when comparing against baselines. Most likely a true best winner probably does exist in most

[24]Not to mention the presence of coarser annotations on the exact same data.

cases, models that perform close to others in a given metric could often be considered more or less equal. The RACL paper found their scores using fixed epochs, over 5 runs with random initialization. This hyper parameter limitation is expected, since the authors were interested in presenting a novel architecture, and less focused on statistically robust results. So, while there is nothing directly erroneous with their conclusions that their models outperformed the IMN-BERT set up, further fine-tuning along with more randomly initialized runs could be interesting to double check these results.

Dataset	Model	AE-F_1	OE-F_1	SC-F_1	ABSA-F_1
Lap14	RACL-GLoVe	**81.99**	79.76	71.09	60.63
	RACL-BERT	81.79	79.72	73.91	**63.40**
	IMN-BERT	77.55	**81.00**	**75.56**	61.73
Res14	RACL-GLoVe	85.37	85.32	74.46	70.67
	RACL-BERT	**86.38**	**87.18**	**81.61**	**75.42**
	IMN-BERT	84.06	85.10	75.67	70.72
Res15	RACL-GLoVe	72.82	**78.06**	68.69	60.31
	RACL-BERT	**73.99**	76.00	**74.91**	**66.05**
	IMN-BERT	69.90	73.29	70.10	60.22

Table 2.2: Results presented in RACL paper (Chen and Qian 2020)
AE: target, OE: polar exp., SC: polarity, ABSA: overall
Best scores per metric per data set in bold

Some vital differences between these two architectures could partially explain their variation in performance. As previously pointed out, the stacked setup implemented in RACL allows for separated extraction of patterns from different abstraction levels. In the iterative setup (IMN), the same components have to learn both shallow patterns together with deeper ones, which could cause some confusion for high numbers of iterations.

Another vital difference is the separation of target and polar expression extraction. In IMN, the first subtask labels tokens as either targets, expressions, or neither. This assumes the two are completely independent, making each task slightly harder than it has to be. RACL separates these as two individual subtasks, and shares information between them, allowing for contextual dependencies between extracted elements, potentially improving performance for both individual classification, but also polarity classification.

The final major difference between models is the possibility to learn from more abstract, document annotations in the IMN setup, which the RACL setup lacks. Even though no major performance boost came from these extra data in the original paper, updates to the architecture could possibly incorporate these data in a meaningful way and show further improvements to these models.

Notice how the results obtained by these baselines vary according to data sets. This variation is due to different distributions of opinions across the different domains tested. We should expect to see even greater differences between the results obtained by the English and Norwegian data sets, since opinion distribution would not be assumed to equal across languages.

A deeper breakdown on the model outlines of these two architectures will be detailed in Chapter 3.5, along with the specific adaptations that will be necessary for the experimentation of this thesis.

2.7 Chapter 2: Summary

Up to this point, background knowledge about fine-grained sentiment analysis has been presented, along with clear definitions of terminology and the current status of research for solving the task at hand.

Specifically, the three main levels sentiment analysis experiments often focus on has been thoroughly defined. Datasets with their annotations and terminology were also discussed, in hopes to dissolve ambiguity among flexible terminology often used throughout the field (at least for the sake of this experiment). A brief history of text representations along with their more modern counterparts was explored as well. Some architectural specifics about transformers, attention, and BERT were also introduced. More details involving these components are discussed in Chapter 3, including how they are incorporated into our optimal setup. Finally, the important interactive networks that are used as baseline models for our experiments were presented, along with their results from their respective original papers.

With this solid foundation of previous work of this field, the reader should now be ready to dive even deeper into the framework of our optimal setup, along with the metrics used to compare architectures. Chapter 3 covers these topics, in addition to model outlines and how data flows through these architecture. It also extracts the most useful pieces of the baselines to eventually create our flexible synthesized architecture for fine-grained sentiment analysis.

CHAPTER 3

Framework

Preparation is vital for the efficacy, credibility, and reproducibility of an analysis on fine-grained sentiment. Decisions made during the early stages of an investigation mold the potential insights to be discovered through experimentation. Development environments, performance metrics, individual components, together with full model outlines make up the fundamental framework of our experiment. This chapter presents and discusses these four topics, providing a scaffolding of understanding around the foundation laid in the previous chapters.

Robust hardware setups are needed when investigating complex language processing tasks, such as fine-grained sentiment analysis. Computation time and storage space budgets can drastically affect the results obtainable from a single project. Additionally, the use of systems presented in previous works often require unique alterations to development environments, since most experiments are developed at different times, using different standards. Problems related to platform and resource requirements are discussed in the first section of this chapter, Section 3.1.

The next section (3.2) explores the metrics used to compare the models tested. Metrics analogous to those used in the baseline experiments are also necessary to use here, since the baseline models were specifically optimized for those metrics. Only comparing against new metrics in future experiments would be unfair to those setups, and too easy too manipulate in favor of the desired solution for a given experiment. However, new metrics are introduced in this section to allow for even more sound comparisons against future projects that want to reproduce this experiment's setup.

After metrics are presented, we go on to describe our hypothesis in Section 3.3. This gave the project an overall goal, and a direction for development. Here, our research questions are used to define a hypothesis test that is later used in Section 5.3 to compare our best models.

Next, before surveying the full model blueprints used in this experiment, specific model components are briefly discussed in Section 3.4. While a detailed introduction to these components is outside the scope of this thesis, some vital differences about input/output sizes, internal states, and other conveniences between components are discussed in relation to their relevance for this experiment.

From there, full model outlines are presented in Section 3.5. Our novel architectures, some simpler than the baselines and others more complex, are introduced in detail. The original implementation of the baseline architectures are also laid out, together with the alterations we applied to these to fit our experiment. Some notes on each of the model's tunable hyperparameters are also presented. Results of the hyperparameter tuning are discussed later in Chapters 4 and 5.

3.1 Platform

Two big questions that arise when building complex machine learning models are (i) What environment is needed to build the models? and (ii) Where will the models run? Depending on the goal of a particular project, the answers to these questions can vary greatly.

For example, models built for industrial purposes will probably be lighter and less complex, so that they can be run more often with general but reliable results. On the other hand, models built for academic purposes may focus less on restraining complexity, and rather allow for more freedom of exploration across complex model spaces. Another scenario entirely could be machine learning competitions, where model complexity is rarely treated as a deficiency, and is often that extra little edge needed to steal first place.

Following the typical development style for academia, this thesis explores setups that potentially improve upon current state-of-the-art models, while maintaining reasonable but not necessarily restricted model complexity. Future researchers wanting to build off the findings of this thesis will need access rights to no more than a single GPU node.

3.1.1 Version-control

The code developed for this experiment can be found on GitHub[1]. The use of a source code management system like git allows for easier sandbox development through use of branches. A new idea can be created on a new branch, then merged into master when proven stable. Also, future researchers wanting to use parts of this code can easily clone the source code and recreate the project directory for testing on their own machines.

3.1.2 Development Environment

Continuing the theme of reproducibility, a well-documented development environment is always appreciated when researching new architectures. This includes both the runtime version of the programming language used in the project, along with version-pinned dependency packages the project requires. In some experiments, operating system is important to mention as well. However, our project can be run independent of operating system, as long as Python can be installed.

3.1.2.1 Requirements

This thesis was built and run using Python version 3.7.4. A list of version-pinned requirements can be found in the project repository. Similar lists of dependency

[1]GitHub repository: https://github.com/pmhalvor/fgsa/

packages were provided for both baselines used, although through different documentation techniques.

For clarification, version-pinned dependency packages in this context refer to explicitly providing a list of pip-modules used in the experiment, along with each module's specific version labels. This is vital for reproducibility since most Python libraries are open-sourced, with new contributions and updates released often. Older libraries like `numpy` and `scikit-learn` have fewer fundamental changes in new releases. However, newer libraries, like `transformers` and `pytorch` (both used in this project) have previously been known for releasing large changes that have lead to dependency-error nightmares. Practicing good version-pinning saves from these unnecessary, and time-consuming, bugs.

Both the runtime environment and dependencies for our project differ from those used in the baseline projects. Tools such as `conda` exist to assist in environment management between projects. Nonetheless, development around baseline architectures is often still time consuming. When testing different architectures in different environments on Slurm-based servers, project-specific modules can be loaded according the requirements to ensure proper configuration on the machine. The next section discusses the ups and downs of such development environments.

3.1.3 Servers

The hardware requirements needed to run modern multi-million parameter models are often too expensive for individual researchers or institutions to train locally. Instead, academic research tends to take place on HPC-clusters, often maintained through collaboration between multiple research institutions. These servers often have integrated job scheduling systems, like Slurm, that queue and deploy jobs according to their expected resource demands.

While different environmental requirements of experiments run locally are easily managed through `conda`, a more scalable management system is needed on Slurm servers, with up to a couple hundred users logged in at a time. Dependency switching is sometimes exacerbated by these systems because libraries are often bundled into larger project-specific modules. New bundles can always be created for each new system to be tested, however, this access is not always provided to all users, creating potential delays and other hassles around environment setup. This often leads to a limited choice between library versions, which makes debugging dependency errors laborious. Learning how to navigate around these hurdles also took more time than initially planned when setting this project up.[2]

Experimentation done for this thesis was split between two Slurm based HPC-clusters, Saga and Fox. A small amount of baseline testing and initial model development was done on the Saga HPC-cluster managed by UNINETT Sigma2. This server is quite popular, causing some delays in job queues up to multiple hours during peak usage. This motivated the transition over to University of Oslo's Fox server.

Fox is a relatively new cluster, with less than 50 active projects currently running tests regularly at the time of writing this thesis. There are 12 NVIDIA

[2]We later learned that the use of `conda` on the Fox server was possible, but only after we had found proper module configurations for the systems we wanted to check there.

A100 GPUs are available for queueing jobs, meaning we had access to more than enough space to run heavy computations quite fast. Our experience with Fox was much better than with Saga, solely due to the smaller wait times for larger jobs, allowing for more effective testing and experimenting.

More details on the technical specifications of the hardware for both Saga and Fox can be found in their respective documentation.[3,4]

3.2 Metrics

Baseline comparisons are necessary to reliably evaluate the performance of a model. These often compare current setups against previous experiments. Good baseline experiments usually introduced novel architectures or configurations that achieved promising results. The main topic of this section if to introduce the metrics to be used during experimentation.

Ideally, the same metrics presented with the baseline systems should also be used when comparing them against new architectures, since it was those models were optimized for those metrics. However, new metrics can supplement novel architectures in addition to those used in baseline tests, especially when a model is trained on more intricately detailed data than has previously been seen, like in our experiment.

When measuring performance of a sequence-labeler, one is often interested in observing both how many true labels a model misses, as well has how prone the model is to predicting erroneous labels. F1-scores are often used to cover both of these scenarios that can occur when producing label estimations. With help from a confusion matrix, concise introductions of robust metrics for measuring performance across varying label outputs are presented in Section 3.2.1.

The baseline architectures provide a good starting point for model complexity we explore in our thesis. These novel, baseline models focused only the on extracting targets, expressions and polarities, as holders were not included in the data sets used in these projects. This naturally limited the evaluation metrics these architectures could use to measure performance, covering only these three labels. The metrics used by these baselines, along with our slight alterations to better fit our experiment, are discussed in Section 3.2.2.

Often, extra annotations add to the necessary complexity a model must have in order to use the newly provided data. A single metric does a good job at standardizing comparisons between models, but only from a single perspective. Alternate metrics can supplement training processes with new perspectives on how the model is learning. Section 3.2.3 helps the reader understand how these other relevant metrics included in our evaluation schema gave us an expanded vantage point of model performance during training.

External characteristics of a model, like train time and memory usage, can also be used to compare model efficacy. Such metrics stand independent of any reports of similar metrics in the original papers presenting the baseline models, since we run our systems on entirely different platforms. These were not heavily prioritized in our experiment, but some discussion around their importance is mentioned in section in 3.2.3.5.

[3]Saga: https://documentation.sigma2.no/hpc_machines/saga.html#saga

[4]Fox: https://www.uio.no/english/services/it/research/platforms/edu-research/help/hpc/docs/fox/system-overview.md

3.2.1 Metric Fundamentals

A system's ability to estimate true label values for a given input is often represented through four main metrics: accuracy, precision, recall, and F1-score. Each represents a unique perspective of the system's performance, and can be expressed through values in a confusion matrix.

		Predicted	
		Present	***Absent***
True	***Present***	True Pos.	False Neg.
	Absent	False Pos.	True Neg.

Table 3.1: Confusion matrix for single binary label.

$$\text{Accuracy} = \frac{\text{True Pos.} + \text{True Neg.}}{\text{Total \# data points}} \tag{3.1}$$

Accuracy is the measurement of the number of correctly predicted labels (both present and absent in the binary example) over the total number of data points estimated over. It is the simplest of the four main metrics, but fails to highlight imbalance between the labels in the evaluation data.

$$\text{Precision} = \frac{\text{True Pos.}}{\text{True Pos.} + \text{False Pos.}} \tag{3.2}$$

Precision is the number of correctly classified data points for a given label over the total number of data points estimated as having that label. The inclusion of the falsely predicted labels in the denominator helps show how biased the model is toward predicting the label of focus. Precision can be found for any of the labels estimated over, making precision a label-wise performance metric.

$$\text{Recall} = \frac{\text{True Pos.}}{\text{True Pos.} + \text{False Neg.}} \tag{3.3}$$

Recall is the number of labels correctly estimated over the total number of true labels in the training data for a given label. In other words, it gives an experimenter a measurement of how many true labels the model missed. Similar to precision, unique recalls can be found for a each of the labels estimated over, making recall also a label-wise metric.

$$\text{F1-score} = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}} \tag{3.4}$$

F1-score is represented as two times the ratio of the product of precision and recall over the sum of the two. It is often the preferred performance metric for machine learning tasks where label classes not expected to be balanced. This is because it takes into account both a measurement on how biased a model is to excessively estimating a label's presence, as well as a measurement on the model's tendency to neglect that same label.

F1-scores, precision, and recall are all three calculated per subtask. A macro-average of these task-wise F1-scores can be found to aggregate the performances of all subtasks into a single metric.

3.2.2 Baseline Metrics

Both IMN and RACL evaluate against *exact-match aggregated target and polarity F1-score* as their main performance metric, labelled in our experiment as `absa`.

The *exact match* part of the name denotes that the scope of the attributes being predicted must perfectly match the expected label in order for the prediction to count as correct. This means the predictions with some correct overlapping tokens, but missing at least one token's label, count as both false negative and false positive, giving a double penalty. Compared to softer metrics, an exact-match metric can be considered a very strict performance measurement, likely to score very low for most architectural set ups.

An example of a softer metric here could be to consider an estimation "correct" as long as part of the scope overlaps with the true, expected value, called *binary overlap*. Another more lenient metric than exact match could be to count all tokens correctly classified with their respective true labels. More on why we also included some of these softer metrics in in Section 3.2.3.

The *aggregation* of the main baseline F1 score comes from the fact that precision and recall of both target and polarities are found together, instead of individually. Here, we take the sums of correctly predicted labels for both target and polarity over each respective metric's similarly summed denominator. This is sometimes referred to as a *micro-averaged F1 score.*[5]

Equations 3.5, 3.6, and 3.7 show how each precision, recall, and F1 score for the `absa` metric is calculated.

$$P_{absa} = \frac{TP_{target} + TP_{polarity}}{TP_{target} + TP_{polarity} + FP_{target} + FP_{polarity}} \tag{3.5}$$

$$R_{absa} = \frac{TP_{target} + TP_{polarity}}{TP_{target} + TP_{polarity} + FN_{target} + FN_{polarity}} \tag{3.6}$$

$$F1_{absa} = 2 \cdot \left(\frac{P_{absa} \cdot R_{absa}}{P_{absa} + R_{absa}} \right) \tag{3.7}$$

While a majority of the code written to calculate this metric was reused from the baseline source code, some slight alterations were made to fit the metric into our experiment. Framework adaptations, including type-casting argmaxed output vectors from PyTorch tensor defaults of floats to integers, were added to the evaluation script, along with comments denoting where these changes were added. Additionally, we chose to adapt the metric to disregard tokens that the model was also told to ignore. This feature was added to ensure we were not being overly strict on tokens excluded from the loss function, i.e. tokens the model never learned to properly estimate anyway.

In Chapter 4, we show that this metric proved to be the hardest to train for. Even though it was also hard for the baseline models, run on an English data set, it could be argued the metric was extra difficult on our Norwegian

[5]Note the difference between micro- and macro-averaging. Micro-averaged scores are found by grouping all true positives together before dividing by total true positive and false positives for precision, then repeats a similar process for recall. Macro-averages are found from the average of the individual F1 scores.

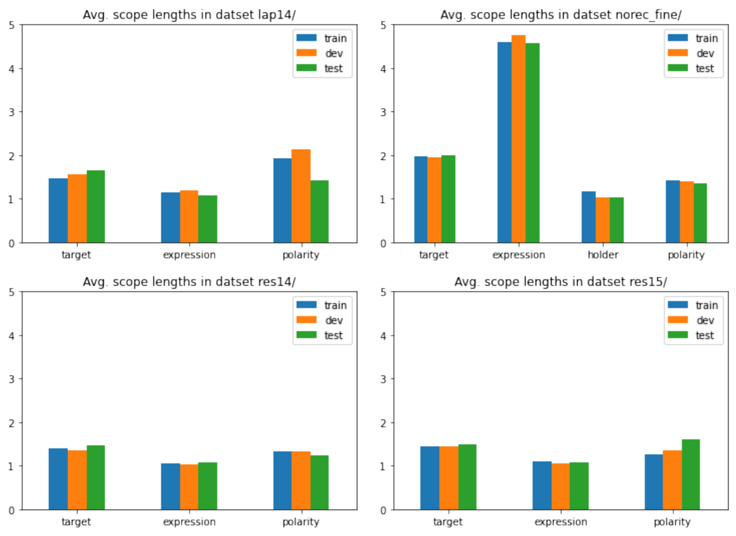


Figure 3.1: Expression scopes for the Norwegian dataset were much higher than any of the other sets, making evaluation extra strict for these label predictions

data. The full scope is needed to get counted as correct, so longer label scopes means that model must correctly predict more tokens to avoid penalty. The more elements a model must get right, the more difficult a task is.

Figure 3.1 shows how average scope length varies between the English data sets (`lap14`, `res14`, `res15`) versus the Norwegian data set (`norec_fine`). As can be seen, the average size of target scopes in the Norwegian data is slightly higher than those from the English sets. We also observe an average expression scope size up to 4 times larger than the other sets. This tells us that an exact match metric may be slightly unfair when looking at target and expression predictions on Norwegian data. Because target is a part of our main metric, we would expect the Norwegian data set to score lower on this metric than any score obtained from the English data.

3.2.3 Alternate Metrics

In addition to the very strict *exact match aggregated F1* score, a few other metrics are also used to show different perspectives of how our models were learning. The alternate metrics we will cover here are a binary overlap F1 score, an averaged individual exact match F1, a proportional F1, and a span F1 score. While some of these metrics were based off the baseline evaluation method, others were calculated using the SciKit-Learn library.

3.2.3.1 Averaged individual exact match F1

The metric closest to our main baseline metric mentioned above was our *averaged individual exact match F1*, labeled `hard` in our experiment. This metric consists of equal parts of an exact match F1-score for each subtask the model was told to predict. The difference between this metric and the main `absa` metric shown in

Equation 3.7 is the inclusion of holder and expression evaluations. In Chapter 4, we see that these extra annotations seem to help performance slightly, with the `hard` metric scoring slightly higher than `absa` throughout most studies.

$$F1_{hard} = \frac{1}{4}\Big(F1_{expression} + F1_{holder} + F1_{polarity} + F1_{target}\Big) \tag{3.8}$$

For this particular metric, the task-wise F1-scores were found using the code from the baseline models. Again, slight changes needed to be made to adapt this evaluation-method to our code.

Our data set contained only negative and positive polarities, yet the baseline architectures also looked for a neutral polarity. We adapted the evaluation source code from the baseline to take into account whether three possible polarities exists, or only two.

Originally, these individual F1-scores for each annotation scope were also considered as secondary metrics to provide insights on a model's sub-task dexterity. However, plotting task-wise metrics with overall metrics gave cluttered plots difficult to decipher. We decided that an average of the individual scores would suffice for capturing the eventual information found by this metric.

3.2.3.2 Binary overlap F1

Working on the opposite end of the strictness spectrum, the *binary overlap F1* metric, labelled `binary` in our experiment, counts predictions as true positives if as much as a single token within the label scope is correctly predicted. Such an easy metric can show that a model is doing something right, even if it struggles to perform well on the more stricter metrics.

Given a model's random initialization, and the fact that each attribute can be classified as one of three possible states, it can be expected to see that this metric will achieve high results already from the first epoch. As long as the model does not learn to only predict zeros (as if all attributes should have no labels), this metric should plateau very early, even as other metrics still slowly increase. Such a situation would tell us that the model needs more time to learn clear scope limits than it needed to find the general neighborhood a particular attribute is expected to be in.

3.2.3.3 Proportional F1

To get an idea of how well our model was doing at token level predictions (not entire scopes), we also included two variants of SciKit-Learn's `f1_score()`, namely with *micro* and *macro* averaging, labelled `proportional` and `macro` respectively in our experiment.

A variant of a *micro* averaged F1 score was shown above in Equation 3.7. Since every task's token-wise annotations can exist in one out of three states (0, 1, or 2), precision and recall must be calculated for each of the states where a label is present, in other words, has a value greater than zero. The true positives for token-wise labels of 1 or 2 are summed to represent the numerator of each precision and recall. For precision, each states' true positive and false positive predictions are summed to represent the denominator. For recall, each states' true positives and false negatives are summed to represent the denominator.

Macro averaging takes the F1 scores for each of the states individually, then averages the two to produce the attributes token-wise macro F1. This resembles the formula for our `hard` metric, denoted in Equation 3.8.

We should expect to see these metrics also increase much faster than the stricter `absa` and `hard` metrics, since token-wise classification will not be penalized for single missing token-label predictions. Although, every new token that gets correctly labelled will result in a slight increase in this metric. In other words, this metric will not plateau if the stricter metrics are still slowly increasing. However, flatter slopes of this metric over the span of a few epochs means the model is getting very close to reaching its utmost potential.

3.2.3.4 Span F1

This last metric considered in our experimentation is the *span F1*, originally presented as a robust metric for fine-grained tasks by Jeremy Barnes, Kurtz et al. 2021. Similar to our two token-wise F1 scorers, this metric counts all tokens predicted to match their respective true labels as true positives. A prediction thats not zero, but that does not match the true value is counted as a false positive. Additionally, when a true label is expected to be non-zero, but the predicted label does not match counts as a false negative. This means that a single token prediction can count as both a false positive and a false negative, if both predicted label and true label are non-zero, but not equal to each other.

This metric gives us a measuring stick of the spans of our model predictions. Different from our simpler token-wise micro- and macro-F1 scores, the span metric does not consider each label state individually. Instead, it puts an emphasis on label presence in general, providing a measure of a perspective between token-wise and scope-wise evaluations.

3.2.3.5 Hardware Metrics

Mere resource consumption of our models was not heavily prioritized during development, but we kept a conscious eye these metrics to maintain a reasonable computation budget. In our evaluation chapter (5), our top models will be presented along with their respective processing footprints. The metrics presented there are single-epoch train times and maximum memory requested during training. This gives future researchers reproducing this experiment an estimation of what resources they will need before starting development.

3.2.4 Final Evaluation Metrics

The evaluation metrics presented above each show their own unique perspective on how our models are training. While only one will be used for crowning a best model, each of the alternative metrics are meant to help development of architectures, showing what aspects work well, and which need some tweaking.

In summary, Table 3.2 shows each F1 variant presented above, along with their unique differences and advantages.

Name	Label	Type	Level	Strict?	Other advantage
E.M. aggr.	`absa`	main	scope	yes	used in baselines
Avg. ind. E.M.	`hard`	alt.	scope	yes	shows task dexterity
Binary overlap	`binary`	alt.	scope	no	easiest metric
Prop. (micro)	`prop.`	alt.	token	no	flatten tokens eval.
Prop. (macro)	`macro`	alt.	token	no	avg. label eval.
Span	`span`	alt.	token	yes	label presence focus

Table 3.2: Metrics used in our experiment summarized. *E.M.* stands for exact-match. *Prop.* stands for proportion.

3.3 Hypothesis

Recall from Section 1.3 that our main research questions ask if we can improve upon some of the current state-of-the-art architectures focused on solving fine-grained sentiment analysis as a sequence-labelling task (**RQ1**) and which eventual components prove most important (**RQ2**). In order to conduct a rigorous comparison of models, we needed to formulate these research questions as our hypotheses.

We define an α-level hypothesis test to measure any potential performance differences between our flexible architecture, the simpler setups, and the baselines. Definitions of a hypothesis and hypothesis testing from Berger and Casella 2001 and used here are as followed:

> A *hypothesis* is a statement about a population parameter θ.
>
> The goal of a *hypothesis test* is to decide which of two complementary hypotheses are true, based on a sample of data points from the population.
>
> These two complementary hypotheses are often called the *null hypothesis* and the *alternative hypothesis*. They are denoted by H_0 and H_1, respectively.

3.3.1 Defining our statistics

Sample parameter θ, the parameter we use for hypothesis comparisons, will be based on our main aggregated F1 score. When running our studies, we noticed exact same model configurations gave varying F1 scores, often in a range spanning between $[0.05, 0.10]$ percentage points. The varying performance was due to random initialization of component parameters and random splits of the data set. In consequence, we use the average score over five random runs for an optimal configuration as the sample parameter, θ_s.

Population parameter For the known population parameter, θ_0, we use an approximated expected value of our main metric on the best baseline model we test. The Central Limit Theorem (5.5.15 in Berger and Casella 2001) tells us that when assuming a population follows a normal distribution and sample size increases to infinity, the average and standard deviation of the sample converges to true values of those parameters describing the distribution of the population where the sample is drawn from. We decided that the average of scores from

100 unique training runs of our best baseline would suffice as an approximate expected value for our known population parameter[6].

Distribution of population parameter Just as we used the average of the scores from 100 of the best baseline runs as an approximation for the true expected value of the population, we can find the sample variance of the scores from 100 runs to formulate an approximate standard deviation, σ_0. Following our assumption that these scores come from a normally distributed random variable made above, we can use our approximate expected value and standard deviation to construct a distribution, $N(\theta_0, \sigma_0)$, describing the probability of a particular sample θ_s.

Significance level It should be noted that the statement *improvements on the baselines* from **RQ1** can be ambiguous since there are many variables at play in a fine-grained sentiment analysis experiment. Distributions of targets, expressions, and holders in a dataset can vary greatly from one language to another, giving slightly different performance results between languages. Also, cherry-picked initialization seeds or excessive resource consumption can induce subtle improvements on one set-up, but might not be generally applicable to different set-ups applied to similar, other datasets. Among others, these variable nuances have an effect on the criteria of an "improvement". In a competition setting, any slight increase in performance would be considered useful to grab that first place. However, for our experiments, we need to define an improvement in a more robust fashion, especially knowing that we have not fully fine-tuned any of our architectures, on purpose[7].

Instead of counting an improvement as any little difference in performance greater than our approximated θ_0, we define a common, yet strict, α-level significance threshold for our hypothesis test. We set $\alpha = 0.05$, giving us a 95% confidence interval about our approximated true population distribution. This means that any statistically significant differences in performance can only be assumed present if a sample parameter θ_s falls outside of 2 standard deviations from of our known population's mean. If the smoothed performance from 5 runs of a model falls under 2 standard deviations below θ_0, then we consider this model as worse than our baseline. Performance over 2 standard deviations above θ_0 is evidence that the measured model is better than our baseline. We define the upper boundary to this confidence interval as θ_α and the lower boundary as θ_β. These areas are colored in red (below = worse) and green (above = better) in Figure 3.2.

Run length For consistency, we train all of our final models used in this statical comparison on a fixed number of 30 epochs. The limit of 30 was found by testing various epoch configurations on our different architectures. We observed our models often approach their respective metric asymptotes early on during training, then remained close for more than 80 epochs, neither increasing or decreasing in performance. This meant our models were not learning anything

[6]We needed to set this number quite high, but also wanted to maintain a reasonable computation budget, for better reproducibility purposes.

[7]We are looking for general architectural changes that apply everywhere and lead to performance increases, not for a single random seed or data set

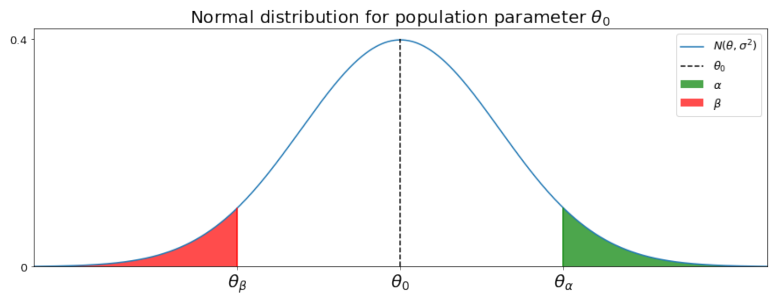


Figure 3.2: Our hypothesis test visualized as a Gaussian curve.

more, but also not over-fitting (at least not by very much). By 30 epochs, all models reached their highest performance, giving us our limit.[8]

Additionally, we decided that 30 epochs is a reasonable number of training iterations when using a warm-up constant of 3.5. This constant allows some of the true labels to influence model predictions in the first few epochs to help guide initial training in the correct direction. A value of 3.5 for this constant gives a influence percentage of 0 by 25 epochs, thus ensuring that models are fully independent of such gold label transmissions by the time training cuts off at 30 epochs. We will explain this feature in more detail in Section 3.5.3, as well as mention it throughout our development and implementation chapter, 4.

3.3.2 Null hypothesis

In our experiment, our null hypothesis, H_0, will be that all of our models performance similarly to our best baseline. Namely, if a sample parameter falls within our confidence interval on our a known population distribution, the null hypothesis can be accepted. We write this mathematically as:

$$H_0 : \theta_\beta \leq \theta_s \leq \theta_\alpha \tag{3.9}$$

With a two-tailed confidence interval using a threshold of $\alpha = 0.05$, the null hypothesis states that the expected value of any particular sample parameter will fall within 2 standard deviations of the estimated expected value of our population parameters. According to Figure 3.2, this means any configuration that produces a sample parameter less than θ_α but greater that θ_β is evidence that we should *accept* our null hypothesis. A sample parameter that is greater than θ_α or less than θ_β is evidence that we should *reject* our null hypothesis.

3.3.3 Alternative hypothesis

Complementarily, the alternative hypothesis represents the outcome where the null hypothesis does not hold, meaning we accept H_1 when H_0 is rejected. For our experiment, this occurs when the observed sample parameter is outside the confidence interval. Because our hypothesis test is two-tailed, we have

[8]An alternative here could have been to implement an early-stopping mechanism that stops training when over-fitting starts occurring. However, training until an early-stopping kicked in would have resulted in 80 or more unnecessary epochs for every model tested. This means many unnecessary hours of training, with little to no extra benefit.

two versions of this alternate hypothesis; $H_{1,L}$, where aa sample parameter is less than the lower boundary, and $H_{1,U}$, where the sample parameter is higher than the upper boundary. The null hypothesis is rejected if either of these are accepted. $H_{1,L}$ provides evidence that the model that produced the sample is worse than the best baseline. $H_{1,U}$ provides evidence of the opposite, that the model producing the sample is better than our baseline. If we find evidence to accept the latter version of our alternate hypothesis, then we have affirmed **RQ1**.

We write our alternative hypothesis mathematically as:

$$H_{1,L} : \theta_s < \theta_\beta \qquad H_{1,U} : \theta_s > \theta_\alpha \tag{3.10}$$

Due random initialization of model parameters and data set splits, observed variability between performances measured on the exact same configurations should be expected. We therefore reiterate that a sample parameter representing a particular configuration's performance is an average of performances from five runs. A run in this context is the full 30 epoch training session we use on all our models. This smoothed average of performance provides a more stable estimate of how a particular configuration actually performs, making our comparisons later on more robust.

As can be assumed from both our research questions and hypothesis test, we hope to find enough evidence to reject our null hypothesis. Doing so would also provide strong evidence that our architecture increases the current state-of-the-art for a fine-grained sentiment analysis sequence-labeling task.

Now that the formalities around our hypothesis are understood, we can present the engineering steps of our experiment.

3.4 Components

Multitask-learning networks are often comprised of individual partitions for each subtask, each containing their own number of layers and components. This section will present and discuss some of the typical components of modern neural networks, mainly to get an understanding of how they affect the flow of data through the model.

Recall transformers introduced in Chapter 2, and shown in Figure 2.6. Even the simplest form of these revolutionary models consist of an encoder, a decoder, and a final output layer. Each of these parts contain multiple components like convolutional filters, attention mechanisms, and densely connected feedforward components. All of these pieces work together, allowing the transformer to encode inputs into some abstract representation, then decode that representation into the desired output format.

3.4.1 Feedforward components

In a classic neural network structure, fully connected feedforward layers are often referred to as dense layers. A block of many such layers one after each-other can be denoted as a *feedforward component*.

A feedforward component consists of a predetermined number of internal nodes and sub-layers, as well as a desired number of output nodes. Each node

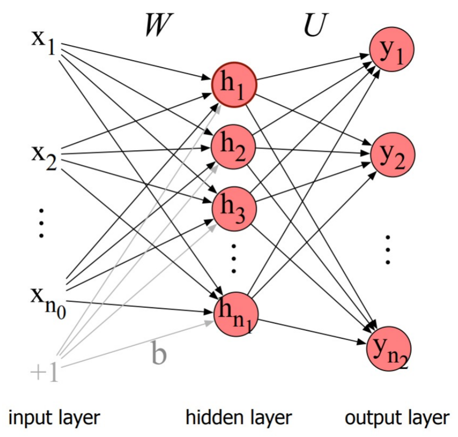


Figure 3.3: A dense feedforward component as presented by Jurafsky and J. Martin 2020

consists of a weight and bias element, each randomly initialized. Weights are multiplied with the inputs from the previous layer, and biases added to them. If the feed component is deemed "fully connected" then every node from one layer feeds directly into every node in the next layer. Dropout is often used to skip a random portion of these connections to help speed up computation time, as well as enforce some regularization to the component.

The final layer of a feedforward component can contain a different number of nodes than those used in the internal layers. This gives feedforward components a flexibility to manipulate output sizes as needed.

While both flexible and robust[9], feedforward components are notoriously slow. Dropout probabilities can help speed calculations up, and can sometimes help generalize learning. However, if too large of dropouts are needed for faster run-time, a drop in performance might arise. The curse of dimensionality throttles the optimal number of nodes and layers that can be used in a feedforward component, forcing the architect to find a balance between component complexity and desired insight depth to be obtained.

Usually, feedforward blocks are used to change input sizes for downstream components, leaving the information extraction to more complex architectural parts like convolutional or recurrent networks. An example of this is used in our baseline architectures, where concatenated task-specific outputs are re-encoded for new iterations of the same subtasks. More on this in Section 3.5.3.

3.4.2 Convolutions

One of the more specialized neural components apply a convolutional filter, or kernel, to the inputs over a sliding window. These are called *convolutional neural networks.*

Contrary to feedforward components, the number of nodes in a kernel is usually smaller than the input size. Instead of being applied to all input elements at once, the kernel captures only the number of input elements equal to the size of the filter.

[9]Robust used here to highlight the loss-less characteristics of feedforward components.

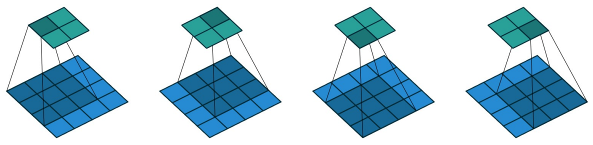


Figure 3.4: A 2D example of convolving a 3×3 kernel (dark blue) over a 4×4 input (lighter blue) producing a 2×2 output (teal) as presented by Dumoulin and Visin 2018

The benefit of applying a convolutional filter to an input is the ability to capture information about an input element's neighborhood. This process is sometimes referred to as n-gram extraction, where n depends on the size of the kernel. The downside here is that not every element is kept separate, but rather is only represented as an aggregate of the neighborhood the element exists in.

For image processing, convolutional filtering has proven helpful in reducing input size while preserving most of the information of the original input, and capturing localized patterns that can appear any where in an image[10].

For language processing, however, the benefits of convolutional filters have been less superior, although still beneficial. Localized patterns are not as common in natural language as in images. Often, dependencies between concepts of a input sentences can stretch over the entire sentence, which a small convolutional filter will fail to capture. This can be mitigated by applying many differently size convolutions to an input, an idea we implement in some of the models we outline in Section 3.5.

Further, variations in input lengths cause variable output sizes form convolutional filter. A pooling mechanism the averages or sums these filter outputs is usually used to find particularly interesting elements of a sequence, as depicted in Figure 3.5. In our project, we use the convolutional components native to PyTorch, which include an average pooling mechanism. Output size of our convolutions can then be set to any constant that fits downstream components. In theory, if a convolutional component's output size is set to 100, then the component will apply 100 randomly configured convolutional filters to the input, and each element of the output will be the averaged results of the applied convolution.

3.4.3 Recurrent networks

Another typical neural component often used in language analysis are recurrent networks. In general, these components consist of a fixed number of internal nodes and layers. Sequences of varying lengths are then fed into the recurrent component, one element at a time, updating the weights and biases relative to the new input element.

When an entire sequence has been fed in, the final hidden states of the nodes are considered the recurrent network's output. Since this is the same size as defined before training, these components are very helpful with dealing with sequences of varying lengths.

[10]This is a major improvement from feedforward networks, that would learn local patterns, and only ever expect to find similar patterns in future images in the same location

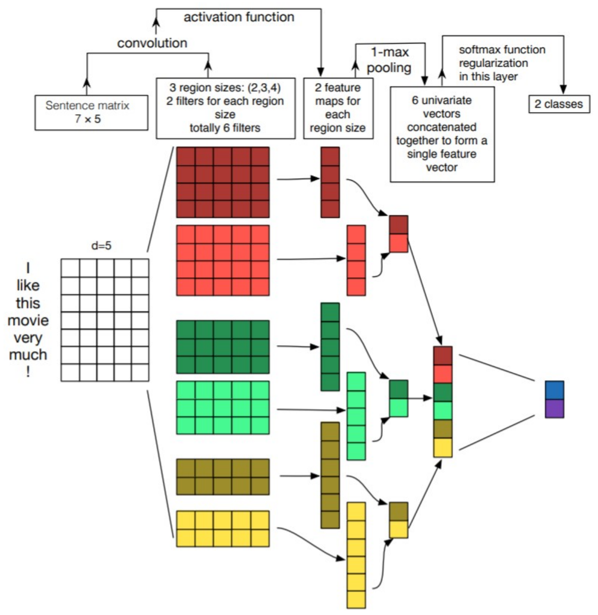

activation function
convolution
1-max pooling
softmax function regularization in this layer
Sentence matrix 7 × 5
3 region sizes: (2,3,4) 2 filters for each region size totally 6 filters
2 feature maps for each region size
6 univariate vectors concatenated together to form a single feature vector
2 classes
d=5
I
like
this
movie
very
much
!

Figure 3.5: Apply multiple convolutional filters of different sizes over text embeddings with pooling mechanism, presented by Y. Zhang and Wallace 2016. The downfall here is models are forced to condense all the learned information into a single pooled value, which can send a blurry signal further downstream.

Different flavors of recurrent networks can be useful depending on the task at hand. For example, enabling a *bidirectional* setup, instead of unidirectional, means that sequences are read from front to back, then back to front. This could be useful for sequence-labeling tasks where dependencies can point both forward and backward in the input sequences. Usually, the last hidden states after the forward pass are concatenated with those after the backward pass, thus doubling the otherwise fixed output size of the recurrent component.

A slightly more complex variant of a recurrent neural net is the LSTM network. *Long short term memory networks* use activation layers called "gates" to determine what information is useful to remember, and what can be considered irrelevant to be forgotten. This helps a model remember patterns spanning over both long and short scopes of input sequences. LSTMs are often beneficial when looking at temporal data, where both immediate neighbors, and long-term patterns, like seasons, arise.

The original decoders of the BERT architecture used bidirectional LSTMs to decipher what the 12 attention layers encoded in the model's head to assist in the initial pre-training. Taking inspiration from this setup, we also build a model with LSTM outputs, to compare against our of fine-grained set ups.

3.4.4 Concatenation

While not typically discussed as a "component" in most architectures, understanding how concatenation of outputs produced from several subtasks of a multitask model can be useful when attempting to combine information learned in each task. It is important to know which dimensions should be concatenated, and how the shapes of the next downstream components should be configured.

For our models, concatenation usually occurs on the embedding dimension. This maintains original sequence output (sentence size), by now representing each token with twice as many embedding dimensions. This architectural choice follows what was done in both IMN and RACL architectures (Sections 3.5.3, and 3.5.4 respectively).

3.5 Model Outlines

In the Sections 2.4.3 and 3.4, we presented the main building blocks used in the models built for our experiment. We saw how different components can transform inputs while learning from them, specifically which dimensions components act on and learn from. This section aims to put these pieces together, laying the blueprints for the models to be tested in our experiment.

The first two architectures outlined (BertHead: 3.5.1 and FgsaLSTM: 3.5.2) are considered to be simpler than our baseline architectures, but with varying complexities. These will tell us if the complexity of our baselines were really necessary, and how far we can get in a fine-grain sentiment analysis task just from use of BERT embeddings.

The next two architectures (IMN and RACL) detail the system set ups of our two baseline models. These outlines are presented here as close to their original source code as possible. However, when implemented in a different framework, some architectural tweaks were needed. These implementations and changes are thoroughly discussed in Sections 3.5.3 and 3.5.4.

The final architecture presented here is our novel iteration on the baseline models, called FgFlex. This model was built as a flexible combination of our baselines architectures, to allow researchers the ability to explore different component configurations easily. Section 3.5.5 presents how this model was built, along with the configurable parameters it allows for.

Even a simple model like our BertHead can span 700 lines of code, so doc-string documentation is vital for any chance of reproducibility. We made an effort to correctly explain all methods important to all our models, including short descriptions of the method, their expected input parameters, and returned outputs. This will help future developers know exactly what methods they should replace when trying to iterate on one of our architectures. See the GitHub repository[11] for more information on the models outlined in this section.

3.5.1 BertHead

As the simplest of models built for our experiment, BertHead reflects the simple BERT-based models presented in the WNNLP experiments mentioned in Section 2.5. Here, a shared, upstream NorBERT head connects to individual task-wise linear output layers.

[11] https:/www.github.com/pmhalvor/fgsa

This architecture relies mainly on the power of fine-tuning NorBERT to learn how each task interacts with the other.[12] The ability to fine-tune BERT according each individual task is configurable through hyperparameters, along with which subtasks should be included in the set up.

A standard cross-entropy loss is used, in order to protect against unbalanced label distribution. Optimizer and optimizer parameters are also configurable as hyperparameters, some of the few that this model actually allows. Other hyperparameters for this architecture include dropout, loss weights, and task-wise learning rates.

The hyperparameters can all be configured through JSON files saved in the `studies/` directory off the project's main root. More on these configuration files, and how they assisted in hyperparameter tuning in Chapter 4.

BertHead serves as the flexible base model for the rest of the models built in this experiment, allowing for quick component initialization for more complex models. Most new architectures need only overwrite the `init_component()` method to specify which components they need in their model, and the `forward()` method, to tell the model how to use the components initialized.

During component initialization, a `torch.nn.ModuleDict()` should be made containing module dictionaries for each of the subtasks desired to train for. Shared layers can be added to the components dictionary under the `shared` key. Following this recipe will allow the `init_optimizers()` method to properly sort which components should be optimized for which subtasks.

Components belonging to multiple subtasks, yet not all, need to be specified in a model specific attribute named `self.other_components`. This feature mainly handles the attention blocks between two subtasks, presented later in Sections 3.5.5. In some cases, it may be easier to overwrite the `init_optimizers()` method entirely, although we tried making this method as flexible as possible for most future architectures.

The forward pass will almost always need to be updated when new components are initialized. The only exception is if all components consist of single pipeline-like data flows (like `torch.nn.Sequential` objects), where no information is meant to be passed to the other subtasks (most likely not the case for interactive fine-grained models). It is important to note that only the current `batch` is needed as a parameter for a forward method of a BertHead-child module. However, the inputs and attention masks included in the batch must be cast to same device as model in forward pass, in order for PyTorch to correctly feed this data through all the components. Our models default to `cuda`-devices if available, since we mainly ran our experiments on the Fox GPUs.

The output of this model is a dictionary object containing predictions for each of the subtasks the model was trained for. The format for each task's individual output is a float-tensor of size `[batch size, sequence length, number of labels]`.

The last dimension consists of logits for the three possible labels a token can have, since each subtask can have three unique label states per token (0, 1, or 2). To get the model prediction, we needed to find the argmax of these logits, meaning the index with the highest value.

[12]Surprisingly enough (or unsurprising for you BERT-believers out there), this model scored very well out-of-the-box. More on it's results in Chapter 4

The output layers of this model make no use of activation functions, which were deemed unnecessary here since all subtask inputs came from the same embedding outputs. Feeding the linear outputs directly to the loss functions allowed for very confident optimization steps in the wrong direction to be penalized more than if an activation function were applied.

A final point to mention on this flexible base architecture is the built-in `self.find(param, default=None)` method. We added this functionality for easier class initialization using `**kwargs`. Our `self.find()` checks if the string input parameter exists as an attribute in the model, and returns the value. If the model does not have an attribute with the same name denoted in the parameter string, a configurable default is returned. This allows for model flexibility, and avoids crashing when a certain parameter is not provided in the configuration files.

3.5.2 FgsaLSTM

The next iteration of models swaps out the linear outputs of the BertHead module with recurrent LSTM layers. Again, the particular subtasks a model should train on and predict for can be configured via the model parameter `subtasks`.

In additional the hyperparameters configurable for the BertHead module, LSTM hidden size and hidden layer counts can also be added, although only globally. Other than individual learning rates, unique task-wise configurations were not included in the initial setup of this architecture. The ability to set one task's hidden shape to a larger value than other could have given performance boost, but can be engineered on future updates to the repository if felt necessary.

This model was developed to serve as a complexity step above the simple BertHead, but without the FGSA specific interactions present in the baselines architectures.

3.5.3 IMN

Introduced in Chapter 2, the *Interactive Multi-task Learning Network* presented by He et al. 2019 is one of the main projects our thesis is based on. It's iterative message passing setup allows for communication between subtasks. With an extendable document-level auxiliary task, this setup can make use of multi-level sentiment annotations, which make it an attractive architecture for FGSA on the NoReC$_{fine}$ data set.

First trained and tested with Norwegian data on it's original source code, an IMN template was eventually built in our framework, PyTorch, for closer analysis on the underlying architectural advantages of this model. More discussion around this framework adaption was necessary is presented in our experimentation chapter, Chapter 4.

To start off, inputs are fed through our BERT embedder, followed by a shared feature extraction component, f_{θ_s} in Figure 3.6. This shared component is series of convolutional blocks, with kernel size 5 for all but the first layer. The IMN architects chose to split the first shared layer between two convolution filters, one with kernel size 3 and the other with kernel size 5. Output sizes were halved, so they could be concatenated and fed through the next shared layers as the same shape the original shared embeddings were.

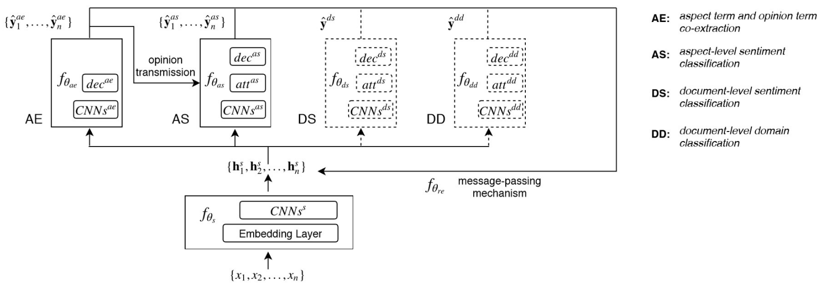

$\{\hat{\mathbf{y}}_1^{ae},...,\hat{\mathbf{y}}_n^{ae}\}$
$\{\hat{\mathbf{y}}_1^{as},...,\hat{\mathbf{y}}_n^{as}\}$
$\hat{\mathbf{y}}^{ds}$
$\hat{\mathbf{y}}^{dd}$
opinion transmission
$f_{\theta_{ae}}$ dec^{ae} $CNNs^{ae}$ AE
$f_{\theta_{as}}$ dec^{as} att^{as} $CNNs^{as}$ AS
$f_{\theta_{ds}}$ dec^{ds} att^{ds} $CNNs^{ds}$ DS
$f_{\theta_{dd}}$ dec^{dd} att^{dd} $CNNs^{dd}$ DD
$\{\mathbf{h}_1^s,\mathbf{h}_2^s,...,\mathbf{h}_n^s\}$
$f_{\theta_{re}}$ message-passing mechanism
f_{θ_s} $CNNs^s$ Embedding Layer
$\{x_1,x_2,...,x_n\}$
AE: aspect term and opinion term co-extraction
AS: aspect-level sentiment classification
DS: document-level sentiment classification
DD: document-level domain classification

Figure 3.6: Visualization of the Interactive Multi-task Learning Network (He et al. 2019)

The outputs from these shared layers, $\{\mathbf{h}_1^s, \mathbf{h}_2^s, ..., \mathbf{h}_n^s\}$ in 3.6, are then passed forward to each subtask specific components, in a pipeline-like manner.

The first task tackled is target extraction, a sequence-labeling task assisted by a convolutional block. The resulting outputs are duplicated, sending one copy concatenated with the initial shared latent vectors to the a downstream subtask, and the other through a dense feedforward component $f_{\theta_{ae}}$ producing this first task's output $\{\hat{\mathbf{y}}_1^{ae}, ..., \hat{\mathbf{y}}_n^{ae}\}$. In the output layer, tokens are assigned one of three labels, with the following mapping: $\{0 : 0, 1 : B, 2 : I\}$.

Expression extraction occurs in parallel to the first task of target extraction. Here, an identically shaped convolutional block takes the shared latent vectors as inputs, and outputs the predicted expressions. The same mapping from above is used in the feedforward output component. In figure 3.6, both target extraction and expression extraction are denoted as AE.

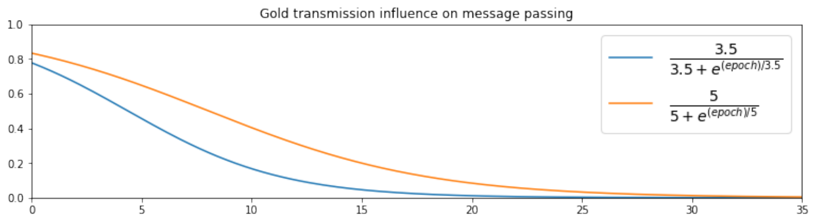


Figure 3.7: transmission proportion for constants 3.5 and 5

Before these outputs are sent to the final subtask, they are meshed with a *gold transmission* of their true labels. For the first few epochs a model is tested on, the gold transmission allows some of the signal from true label values for target and expressions into their predicted outputs from the task-wise components. The amount of affect gold transmission has on these outputs is specified by the following proportion p, based on current epoch number[13]:

$$p = \frac{5}{5 + e^{(\text{current epoch})/5}} \tag{3.11}$$

As the final subtask, sentiment classification aims to assign a polarity to the opinion directed at the target found above. Here, yet another task-specific

[13]Our implementation of this proportion allows researchers to configure the constant (set to 5 in Equation 3.11) as a hyperparameter.

CNN is applied to the shared latent vectors, this one optimized for predicting polarity outputs. Then, attention is applied to a concatenated vector of the target and expression outputs, along with the polarity label in hopes to pick up on some contextual relationships between the target and expression outputs and polarity outputs. The resulting attention outputs are again fed through a dense feedforward layer $f_{\theta_{as}}$, resulting in the polarity output, $\{\hat{\mathbf{y}}_1^{as}, ..., \hat{\mathbf{y}}_n^{as}\}$ in Figure 3.6. The labels assigned at this step are either *positive*, *negative*, or *none*.

If document level classifications are enabled, latent vectors will also be fed into two auxiliary tasks: document-level sentiment classification and document-level domain classification. Each of these apply their own CNNs to the latent inputs, then further apply attention to those outputs, $f_{\theta_{ds}}$ and $f_{\theta_{dd}}$ respectively. Finally, the results are fed through their respective dense layers, resulting in two more task outputs, $\hat{\mathbf{y}}^{ds}$ and $\hat{\mathbf{y}}^{dd}$. However, this feature was irrelevant for our initial experimenting, since document level annotations were excluded from this experiment. It was just mentioned here for model consistency.

The outputs generated from each of the subtasks are then concatenated, fed through a "re-encoding" dense layer, then used as inputs to each subtask or auxiliary task, again. Using a dense layer to re-encode the concatenated outputs from each subtask ensures that the latent vectors to be re-used as inputs maintain the same size as before, protecting against scalability issues with multiple iterations. He et al. (2019) refer to this process as an *iterative message passing mechanism*, also depicted in Figure 3.6 as $f_{\theta_{re}}$. The idea here is that information from the separate tasks individually can be passed over to the other subtask. With more iterations, the more of a chance the different tasks have to affect one another. The number of message passing iterations is controlled and fine-tuned by a hyperparameter.

After the total number of iterations is reached, the outputs from each of the subtasks are joined and evaluated against the input's gold standard. Back propagation then updates the weights of the CNNs based on the loss between the estimates and the true values.

The system lacks BERT-like contextual embeddings, but instead integrates the ability to use *domain specific embeddings*. These are static GloVe embeddings relevant to the distribution of input data[14] These generally increased performance, highlighting potential future gains from more context specific embeddings. In later experiments, like the one presented in the next section, an IMN setup incorporating a BERT head instead of domain specific embedding is used as a comparable baseline, giving even better results.

3.5.3.1 Changes between implementations

As mentioned, some component adaptations were needed to build an IMN-like template in PyTorch. The inclusion of BERT embeddings instead of static GloVe embedders has already been mentioned.

Another vital change found through hyperparameter tuning was the removal of activation functions on linear feedforward components. We mentioned our reasoning behind why this change was needed already when presenting our

[14]IMN used data sets containing either restaurant reviews or laptop reviews. So domain specific embeddings would have been curated to match these respective realms as a preprocessing step.

BertHead model. To summarize, we believe this makes our loss step more strict, by penalizing large changes directly proportional to how the model represents a given feature internally. This is just a theory developed after we realized the removal helped performance.

The last major change was the ability to configure which task output we wanted to use as our queries, keys, and expression for the attention step. We initially wanted to check what difference it would have to swap the queries and keys, but then allowed also for value flexibility to test all three tasks together. More on our findings for the optimal configurations in Section 4.4.

Some new hyperparameters that were made tuneable for our implementation of this were the number of layers each individual task should be allowed, together with layers for shared component and interaction count, the tasks to use for query, key, and value inputs to attention, and the activation of an auxiliary function to find scope boundaries. More on fine-tuning studies for these in Section 4.4.

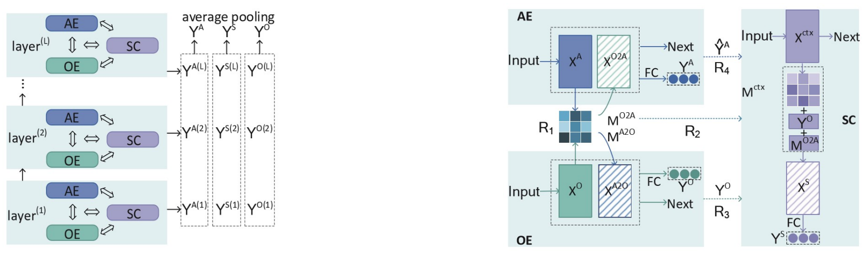
3.5.4 RACL

The other interactive network explored in this thesis is the *Relation-Aware Collaborative Learning Network* (RACL) presented by Chen and Qian 2020. Like the IMN architecture, RACL attempts to share information between subtask components. However, they argue that previous attempts of doing so (including IMN) have often disregarded potential relations of the elements predicted.

The experiment's setup aims to pass information between 4 possible relationship types:

1. target and polar expression
2. polarity with both target and polar expression together
3. polarity with polar expression alone
4. polarity with target alone

As depicted in Figure 3.8b, interactions between subtasks occur mainly through cross-multiplication with some concatenations.



(a) Stacks of RACL layers (b) Details of single RACL layer

Figure 3.8: RACL components as presented by Chen and Qian 2020, with relations mentioned above as R_i for $i \in [1, 2, 3, 4]$

Training starts by sending an input batch through a BERT object, creating the shared high-level contextual embeddings. These are then fed to each respective subtask as the first task inputs. Each subtask starts with their own one-dimensional convolution blocks, each with kernel size 5 and a configurable filter output dimension.

For relation type (1), the outputs of the target extraction and expression extraction components are cross-multiplied through an attention block, creating matrices M^{O2A} and M^{A2O} in Figure 3.8b, where $(M^{O2A})^T = M^{A2O}$.

M^{O2A} is then concatenated with the output from the target extraction CNN, and fed through a fully connected dense layer to produce BIO tag predictions for the targets given the current available information. Similarly, M^{A2O} is concatenated with the opinion-convolution output, then fed through its fully connected dense layer, producing BIO tag predictions for polar expression. These are stored in respective subtask lists, which are eventually averaged after all stacks have been run.

Polarity classification starts with sending it's task inputs (the shared embeddings for the first stack) through a polarity specific convolutional block. The outputs here are then fed through an attention mechanism as keys and values, against a concatenated tensor containing target and expression outputs and the relation matrix M^{O2A} as the query.

Again, the information from both target and expression extraction are provided some warm-up guidance, through a gold transmission mechanism, as explained for IMN.

Outputs from this grouped attention are then fed through the polarity specific feedforward component, producing tri-labelled polarity logits. These too are stored in a task-specific list, to later be average for the final model output.

Each subtask convolutional output, concatenated with their respective attention outputs, are fed through their own re-encoding feedforward component. This layer reshapes concatenated sequence sizes back to the same shape the original shared BERT embeddings were, preparing the data for a new run through the subtask components.

When the re-encoded task-wise inputs are fed back into the subtask components, they replace the previously used embeddings inputs. Every stack's re-encoded task-wise output becomes the next stack's task-wise inputs, as depicted through the `Next` label in Figure 3.8b.

3.5.4.1 Improvements from IMN

The RACL network incorporates two new relation types that the IMN previously was lacking: attention between target and expression and attention using all three subtasks. These, including the target-polarity and expression-polarity relations as a results of the concatenated attention input, make up the four novel relationships introduced by RACL.

Another major difference between the RACL network and the IMN network arises at the stacking step. When resending the learned information back through the system, RACL makes use of new components for every stack, while IMN iterated over the same components. This gives a RACL model more freedom to learn new features in each stack, similar to BERT's 12 encoding layers. New components for every stack allows for optimization steps to update completely different components.[15] Of course, completely new components for each stack increase the size of the model, along with run-time.

3.5.4.2 Changes between implmentations

Dropout used for convolutional blocks in the original RACL implementation followed the DropBlock technique (Ghiasi, Lin and Le 2018), implemented in TensorFlow. However, the potential gains presented in the DropBlock paper were less than we observed through different random initializations, so we deemed a full PyTorch implementation of a DropBlock to be low priority. Instead, we used a vanilla DropOut, native to PyTorch language.

While the relation-types were the cornerstone of the RACL architecture, we also explored alterations to query-key configurations for the different relation-based attention blocks during development. This was partially due to difficulties deciphering the hand-written RACL attention classes, but also due to conflicting interpretations of the proper query-key-value set up for an attention block.

3.5.5 FgFlex

Building off both the inspiration of the two previously presented baselines as well as the frustrations around their source code, we introduce our novel FgFlex model. This architecture combines the best from both IMN and RACL in the same architecture, adding an additional flexibility previously unseen in our interactive systems.

Like all of our models, an FgFlex instance starts training by sending inputs through our imported BERT model[16] to produce the shared contextual embeddings.

These embeddings are then fed through a series of shared convolutional layers, which allow for a configurable kernel size via our JSON configuration files. Taking inspiration from the first layer of shared convolutions in IMN, we looked into building a convolution block with the ability to set multiple kernel sizes, instead of a single size for all tasks. Activating such a split convolution for the shared layers, instead of a standard convolution block with a single fixed kernel, could let the model learn dependencies across neighborhoods that span $\{3, 5, 7, ...\}$ tokens long, potentially finding different patterns between the scopes.

When all shared layers have seen the data, a concatenation of the shared output and initial embeddings sequences is added to a dictionary holding the inputs to each of the subtasks, called `task_inputs`. Each of the `task_inputs` are fed forward to their respective task-wise convolution blocks.[17] Each subtask specified produces an output of the same shape as we got from our embeddings, allowing for easy relational applications in the next step.

[15] Although, components of each stack share the same shape as their predecessors. This opens the possibility for soft-parameter sharing on these components, as mentioned in Section 2.5.1.

[16] In all our models, NorBERT2 model is used: https://huggingface.co/ltgoslo/norbert2

[17] These task-wise blocks could potentially also be given split convolution blocks by changing only a few lines of code in `init_components()`. However, we felt this was unnecessary for our initial release, due to poor initial results and a lack of robust testing in support for the change.

By specifying the `attention_relations` parameter in the configuration files, developers are given the freedom to apply an attention block to any pair of subtasks desired. Take for example, the following configuration:

```
"attention_relations": [
    [
        ["target", "expression"],
        ["holder", "polarity"],
        ["expression", "expression"],
        ["shared", "all"],
    ]
],
```

This would result in four attention blocks. The first relation would have `expression` outputs as the query, and `targets` as keys and values. The second would use `polarity` as the query, and `holder` as keys and values. The third would apply self-attention to the `expression` outputs. Finally, the fourth would use a summation of all the present subtask outputs as the query, the outputs from the shared layer as key and values. [18]

All the attention inputs are applied a gold transmission, similar to both IMN and RACL, to let information from the true values into the attention blocks for the first few epochs, configurable through the `warm_up_constant` hyperparameter. The ability to configure this constant via a hyperparameter was necessary since it would directly affect the results obtained from a run. Too high of a constant, over too few runs would mean the final evaluations were actually leaking true labels into the model, thus invalidation any performance metrics. With a constant of 5, at least 40 epochs were needed before the transmission probability was down to 0. A constant of 3.5 shortened this required train time to 30 epochs.

After all the specified relationships are applied to our data flow, the information obtained for each subtask is concatenated, re-encoded, and overwritten to their respective `task_inputs` indexes, setting up for the next stack.

Like RACL, the FgFlex architecture makes uses of stacks, where every new stack feeds the task-wise inputs to new components with the same shapes as the previous stack.

Along the way, the outputs for task-wise components are sent through their respective linear output layers, storing the resulting logits in task-specific output lists, like in RACL. Additionally, when a task is used as the key and values in an attention block, the output from the attention mechanism are also added that task's output list.

The final step of a FgFlex model's forward pass takes the average for each of these lists. The task-wise averaged tensors of size

```
[batch size, sequence size, number of labels]
```

are returned as final outputs in dictionary format, with one tensor for each subtask.

[18]Notice that this last relationship contains special elements `shared` and `all` not included as subtasks. This required specific updates to the optimizer initializer, which we will discuss further in Section 3.5.5.1.

In addition to the hyperparameters possible to tune over for all the previous models mentions, the FgFlex model could also tune `attention_relations`, `loss_weight`, `kernel_size`, and `split_cnn_kernels`.

3.5.5.1 The Good

Configurable subtasks The first major improvement our FgFlex model provided was the ability to configure the subtasks the model should train for. This was implemented merely because we had a data set with holder annotations, previously not available to the IMN and RACL developers. Subtask flexibility allows us to further study exactly which annotations helped in model performance.

Shared layers with multiple attention relations Another improvement was the combination of shared convolutional layers with multiple attention relations later downstream. The IMN architecture made use of shared layers, but only used a single attention mechanism during the task-wise interaction step. RACL had multiple attention relations, but no shared layers, making them fully reliant on BERT embeddings to be tuned for the FGSA task at hand. Our implementation gives the freedom for both, increasing model complexity slightly.

Split convolutional blocks Implementation of split convolutional blocks was inspired by IMN, who only used this feature for a single shared layer. We wanted to make this feature available for any part of the system, completely configurable via JSON files. This allows for more exploratory experimentation with less effort.

Flexible attention relations The ability to apply attention to any combination of subtask pairs is one of the main flexibility traits of our model. Instead of being restricted to previous architectural decisions, future developers using the FgFlex model can easier test new relations between subtasks through slight alterations of model hyperparameters. This implementation also opens the door for a grid-search over all possible subtask combinations, a study that was excluded from our experimenting due to required resource consumption.

Task-specific learning rates The last big upgrade from the baseline architecture that FgFlex provides is the task specific learning rates. While also a part of our implementations of the baselines in PyTorch, this was not a feature of their original models. Task-wise learning rates and optimizers allows for larger learning steps for tasks with clearer signals with smaller steps for the tasks with more noise.

3.5.5.2 The Bad

While the FgFlex model introduces flexibility to the baselines chosen for this experiment, there were some features that remain sub-optimal in this initial release. Some of the following points can be improved upon in further updates, which are mentioned again in Section 6.4, while others are vital parts of the FgFlex architecture. Any major flaws that we were forced to stick with are mentioned quickly below, but discussed in further detail in 6.3.

Re-encoder A major potential drawback of this architecture (along with the IMN and RACL architectures) is the use of a re-encoding layer after a stack or iteration is complete. This are necessary to get the right sizes back to the start of the subtask components, but we feel there may be much information learned for the subtask, yet lost during this distillation. For an interactive network, unfortunately, some type of re-encoding is almost always necessary. One idea that could be tested is expanded component sizes for each stack, according to concatenations in a given stack. We did not implement such an idea, but discuss it further in Section 6.4.

Task-wise split convolutions We mentioned the possibility to use split convolution blocks with multiple kernel sizes in task-wise components, but we did not implement this feature in our initial release of FgFlex. Also discussed further in Section 6.4, this inspiration came to us late in experimentation, with not enough time to correctly implement and test it.

Activation functions We also completely removed activation functions from our linear output layers due too performance jumps during experimentation. While the idea behind an activation function is to standardize and provide non-linear transformations of the outputs, we found our architectures learning better without these features. However, we know that activation functions are usually both helpful and necessary for model robustness, so it is possible we did not configure ours properly or hyperparameter search around them thoroughly enough.

Sub-par gold transmission We implemented a gold transmission just like RACL and IMN, but it did not give the same performance increases we saw in the baselines. We struggled to determined exactly which features of a task's output data this should have been applied to. We ended up applying the transmission to the embeddings dimension, with hope to strengthen the signal for these tokens in any attention block the outputs would be applied to. We acknowledge that this is often not recommended, however it gave the best results during our testing, so we kept it. Future work can potentially look into exactly why this occurs in greater detail as well.

3.6 Chapter 3: Summary

The aim of these four sections from above was to present a scaffolding for our experiment.

The servers where our experiment was run on, along with the required dependencies, were presented to ensure easy reproducibility of our models. We discussed metrics, both our main one but also many of the alternative metrics together with the unique perspective they provided during training and debugging. We gave a brief introduction to the goal of our project, formulated as a hypothesis. The major components we use in our architectures were laid out, including how data gets read in to, learned from, and outputted for each of them. Entire model outlines detailing how data is expected to flow through them and their respective hyperparameters were also discussed.

Now, the reader should have a proper understanding of the experiment's framework. With this knowledge, we are prepared to begin discussion around the experimentation process.

CHAPTER 4

Development and Implementation

With the template for the project set, we can walk through how the series of experiments conducted for this project unfolded. In this chapter, we discuss why certain decisions were made along the way, and the findings we discovered. A few major bugs initially built into our systems are documented and explained in detail according to when they were found during development. The project trajectory described in the following sections highlights some of the reproducibility problems that typically arise when building and comparing complex systems.

Development on our project started by preprocessing our data into the format used in the baselines in Section 4.1. The next natural step, detailed in Section 4.2, was to test the formatted data on the baselines, giving us the first results we would eventually need to beat. In Section 4.3, we discuss how we built the simplest architecture to solve the task of fine-grained sentiment analysis, as defined in the previous chapters. Iterating on this architecture, we then increased subtask complexity in our next model, to better understand each task's respective difficulty. Using the source code for the baselines as a reference point, we then reconstructed baseline architectures in the same framework our models are built in, providing us with baseline systems now overlapping with our own, i.e. built on top of our `BertHead` model. The reconstruction process is detailed in Section 4.4. Here, we hoped to eliminate any ambiguities between set-ups to find exactly which components give performance increases. Finally, we look at our flexible iteration of these baseline models, that take only the best components from the previous structures in Section 4.5. Along the way, we presented hyperparameter studies for each architecture, and brain-stormed some model alterations that could possibly increase performance; some of these were implemented, others saved for future work.

This chapter is meant to guide the reader through the methodology briefly laid out above. We describe how our baseline models were not ready to use right out-of-the-box, but rather required as much attention as our novel architectures. The bugs that followed new model implementations, but that were eventually caught, are described in hopes to provide the reader with the same information we had when making new development choices. While some initial results and conclusions can be drawn from the information presented here, our final conclusion with result tables and in-depth discussions around the most useful components will be saved until Chapter 5.

4.1 Preprocessing sync

One reason both the Interactive Multitask Network (He et al. 2019) and the Relational-Aware Collaborative Learning architecture (Chen and Qian 2020) were chosen as baselines is because they shared the same preprocessing format. In fact, the repository for the RACL system included the data used in both projects [1,2]. This allowed for little hassle around preprocessing errors or ambiguities when test running those architectures. The use of the same preprocessing format in our project as used by the baselines made it easy to check that the NoReC data was preprocessed correctly.

4.1.1 Converting NoReC

The main preprocessing efforts for this project focused solely on this conversion of the NoReC$_{fine}$ data to the IMN-format. The NoReC data is hosted on GitHub[3] as three pre-split subsets (train/test/dev), stored as `json` files. These splits help prevent from training and evaluating a model on the same data, sometimes referred to as *data leakage* in machine learning projects. Such a scenario is considered cheating, and would render any improvements on previous state-of-the-art results as invalid.

Here the train set is fed to the model and optimized over, providing the model with the loss in which it updates it's parameters with. Development data is then used for model evaluation once per epoch, but does not provided any loss feedback to the model. The development set should be used for hyperparameter searching, as it gauges how well the model predicts "new" data, which the model has not been optimized for. When model have been hyperparameter tuned, the held-out test set should be used to validate model performance. This step is necessary in case excessive hyperparameter searching caused overfitting on the development set.

The IMN-format also used pre-split train, test, and dev partitions, further breaking each into four text files: `target.txt`, `opinion.txt`, `sentence.txt`, `target-polarity.txt`. The files `target.txt` and `opinion.txt` are meant to contain BIO-labeled targets and polar expressions, respectively, where 1=Beginning, 2=Inside, and 0=Outside. The file `sentence.txt` contains the raw text of the input sentences, separated into tokens. Tokens are usually the individual words along with the different punctuation marks in the sentence. A model's embedder maps these tokens to their respective embeddings, representing the tokens in a manner interpretable for a machine.

In our thesis, we chose to redefine the previously used term "opinion" as "expression," as to remain consistent with Section 2.2 and the annotation schema used for the NoReC$_{fine}$ date set. While this could cause some confusion during preprocessing, the use of the term "expression" is consistent throughout the codebase focused on model development (which is the best that we can do given the discrepancy between projects). Relevant comments are also provided in the preprocessing file to help future developers catch this change.

[1]IMN: https://github.com/ruidan/IMN-E2E-ABSA
[2]RACL: https://github.com/NLPWM-WHU/RACL
[3]NoReC: https://github.com/ltgoslo/norec_fine

4.1.2 PyTorch `Dataset` and `DataLoader`

After the data was preprocessed and saved in IMN format, we needed to load the data into PyTorch-based tensors. This was done using our hand-written `Norec` class, a child of the `torch.utils.data.Dataset` module. An individual `Norec` object is created for each of the train, dev, and test splits in our data.

In the `Norec` module, an instance of the NorBERT2 model is downloaded from HuggingFace's API, and used as a tokenizer for the raw sentences, found in `sentence.txt`. An integer tensor containing the respecitve vocabulary ids for each token is generated for each sentence. The target, holder, and expression labels for each sentence are mapped directly to their own respective integer tensors, and stored as a list in the `Norec` object.

During a training process, the `Norec` dataset objects are transformed into `torch.utils.data.DataLoader` objects, with help from a specialized padding function. To increased training efficiency, we train with batched inputs. The padding function expands all the elements in a batch to the length of the largest input sentence, by appending 0's or a specified `ignore_id` to the end of each input, depending on the tensor type (labels or ids). A mask tensor is also generated for each input sentence in a batch, specifying where a NorBERT2 instance should focus on, and where it should ignore paddings.

When fitting a model, we iterate over one of these `DataLoader` objects to get the elements of our batches. A batch is a list of tensors, that consists of six elements. Table 4.1 shows the order in which a batch is indexed.

0	input ids	NorBERT2 token ids for each word in a sentence
1	mask	binary tensor for token present 1, otherwise padding 0
2	expression	BIO-labels for expression
3	holder	BIO-labels for holder
4	polarity	BIO-labels for polarity
5	target	BIO-labels for target

Table 4.1: Elements of a batch

4.2 Useful results from baselines (source)

We checked that the Norwegian data was properly preprocessed by downloading the source code for the baselines, specifying the directory of our data, and training the models. The papers detailing the baselines provided their best default values for most of the configurable hyperparameters. We left the remaining parameters as their default, avoiding any initial hyperparameter searching during this first check. This was probably a naive decision considering we were testing the models on completely new languages, with assumably different distributions between annotation relations. However, as the first models tested with our Norwegian data, these initial tests were mainly to check that the preprocessing was in the correct format, and the models were able to learn *something*.

As will be discussed in Sections 4.2.1 and 4.2.2, the initial results obtained from training the baselines on our NoReC data were much lower than those results presented in the original baseline papers. While our lack of fine-tuned hyperparameters could have caused this discrepancy, we deemed the scores as much lower than any gains fine-tuning should yield. Contrarily, the score increased enough from their otherwise random starting point, proving the models were in fact learning, and providing more evidence for our assumption of difference in distributions between languages.

4.2.1 IMN (source)

To get the IMN code to run on Norwegian data, we needed to download Norwegian GloVe embeddings in a similar format to the English based embeddings used in the IMN network. These were found at the *NLPL word embedding repository* hosted by the Language Technology Group at the University of Oslo[4].

The IMN architecture also allows to train on domain specific embeddings, which proved beneficial for performance in the original paper. However, we dropped this step, and did not activate the functionality via hyperparameter configurations. Based on the evidence presented in the RACL paper, we deduced that BERT based embeddings would lead to greater performance improvements than domain embeddings. We decided finding correctly formatted domain embeddings for Norwegian was not necessary for our experiment.

For us, the IMN architecture was surprisingly simple, because it was the only model tested capable of running on our local machines, without any GPU access. A single epoch took only about 5 minutes to run on a CPU with limited storage. This efficiency is likely a result of the pipeline-like structure the IMN model follows. Targets are extracted first, then expressions after. This information is then fed together into an attention mechanism focused on polarity classification. Since none of these subtasks require to be parallelized, a machine can distribute its limited resources to single tasks at a time, thus avoiding out-of-memory errors some larger models might give.

The choice to only run initial tests on the IMN setup locally limited how much fine-tuning that was done. The small hyperparameter experimenting we did was run through single executed jobs over the span of a few days, meaning no extensive grid-search variant was used here. Our main goal was to ensure we had a good-enough representation for what the IMN model could achieve. Once we saw that our preprocessed NoReC$_{fine}$ data was trainable, we wanted to move the project along to the other architectures to test. Therefore, these initial results are excluded from our final result comparisons[5].

After about 40 epochs, the IMN model reached a validation score of $F1_{agg} \approx 0.25$ on our main metric. The task with the highest individual score was polarity classification, scoring approximately 0.43 on it's task-wise F1 score. Target and expression extraction scored considerably lower than this, with validation scores of about 0.30 and 0.25 respectively. Considering the architecture of IMN, this was not too surprising, since it was precisely the polarity subtasks that received all the other subtask information found for a given interaction, in an attention mechanism. This tells us that attention components are likely very beneficial for subtask performance.

[4]http://vectors.nlpl.eu/repository/

[5]It could also be noted here that we reconstructed variants of the IMN architecture in our PyTorch framework, which eventually took the place of these initial results.

Recall, that our main metric, the aggregated F1 score, checks for perfect matches of both target and polarity. In the original IMN paper, this metric was described as the following:

> *To compute F1-I, an extracted aspect term is taken as correct only when both the span and the sentiment are correctly identified* (He et al. 2019).

The results obtained for the source-based IMN setup defined a definite lower limit our novel architectures should all aim to beat.

4.2.2 RACL (source)

Unlike the IMN network, the RACL architecture quickly exceeded memory limitations on our local machines, forcing the rest of our experimentation over to the cloud. This was not very surprising, however, since the number parameters a RACL model updates during each back-propagation increases significantly from the IMN setup for every new attention mechanism added (4 in RACL versus 1 in IMN). Additionally, the RACL model used a BERT embedder, which requires its own optimization updates, instead of the static GloVe embeddings used in the IMN setup.

To run this model, we still needed to download embeddings from the NLPL word embedding repository, since the RACL source code was not integrated with `transformers`, the user-friend HuggingFace API. NorBERT2 model checkpoints were downloaded and stored in the public directories of the Saga and Fox servers, to avoid exceeding our allowed project disk-space limitations.

The initial configurations of the BERT model used inside a RACL instance differed from those of the NorBERT2 model. The RACL architecture used the typical configurations for a BERT-large model (Devlin et al. 2019). This configuration contains 24 layers and hidden sizes of 1024, versus the 12 layers and hidden size of 768 of the original BERT base, the latter of which NorBERT2 followed. Our quick-and-dirty fix to these mismatches was to add a few `if/else` statements in the source code where important shape changes were defined.

Already on the first epoch, the RACL model reached F1 scores competitive with the best score from the IMN source code, although a single epoch took upwards of 3 to 4 hours to train on CPU[6]. This first epoch produced estimations scoring around 0.31 on our main metric. After many hours of training and only 5 epochs later, the score had slowly, but steadily, increased to about 0.33.

While these scores are very low considering the results presented with this architecture on English data[7], we considered this initial testing as a success. We could see the model was learning through major decreases in loss for all tasks for each iteration. The best source RACL model we trained was cut off after 7 epochs, not because it had started over-fitting, but because the Slurm job exceeded the specified time-limit we set for the job on the Saga server. We concluded these results were promising enough to move forward in developing our novel fine-grained sentiment analysis architectures, saving any eventual hyperparameter tuning until similar results were achieved for our models.

[6]This meant it was also necessary to configure all future jobs to be run on GPU.

[7]RACL scored around 0.70 on the main F1 score when trained on the 2014 restaurant data set from SemEval 2014 (Pontiki, Galanis, Pavlopoulos et al. 2014), versus the 0.33 found in the first 5 epochs.

4.3 Simple multitask-learners

With the preprocessing directories and baseline source codes checked, we started architecture development in PyTorch. The two models we experiment on and discuss in this section (4.3) are meant to be very simple implementations with BERT heads compared to our two baselines. This helped us gauge how effective the smart complexity of the baselines is, and provides a lower limit to how much BERT embeddings can learn when fine-tuned for a fine-grained sentiment task.

As will be presented, initial development of these simpler models was effected by a bug in our base model that was finally found when more complex models were built. For both architectures, we mention possible culprits of this bug, and discuss how this ultimately affected our project trajectory.

4.3.1 Flexible multitasking

A natural first starting point for these simpler architectures was to begin with a BERT embedder upstream, then feed these outputs onward to components focused on each subtask individually, without any interactions or communication between them. `BertHead`, the class built for this set-up, served as the foundation for the future, more complex architectures tested in this experiment, as mentioned in Section 3.5.1. Therefore, model complexity was limited as much as possible here, using only a single linear component for each task after the shared embeddings. Additionally, because the main classes of a neural network are often very similar, the `BertHead` class was written to limit as much rewriting boiler-plate code as possible, only requiring one or two method overwrites for entirely new architectures, `init_components()` and `forward()`[8].

When training on our NoReC data, some initial learning rate tuning was necessary in order to find a steadily decreasing loss. We found that learning rates between `1e-5` and `1e-6` gave the most stable decreases in loss, without too much relative variation. We included a learning rate scheduler in our model, that would pay attention to each optimizer, and decrease the learning rate when loss was measured to be increasing for a fixed number of epochs, called the scheduler's `patience`. This constant, along with the factor in which to decrease the learning rate with, were made tunable via our configuration files, in case we felt the default values we initially chose were suboptimal. Default values were set to `patience=5` and `factor=0.1`.

4.3.1.1 Initial development

Like for our baseline models trained from their source code, loss decreases showed that our `BertHead` models were learning. Figure 4.1 shows the task-wise loss for each batch trained over 9 epochs. Here, epochs are separated by the black dotted lines. While expression seems to vary the most, and learns the least (observe it has the highest loss after 9 epochs), all tasks show relative loss improvements as training continues. See Appendix A.1.1 for a log-snippet of the outputs produced by this model.

[8]`forward()` need not be rewritten if components allowed for data flows similar to task-wise `torch.nn.Sequentials`. However, this is often not the case in multitask-learners, and `forward()` would most often need to be rewritten as well.

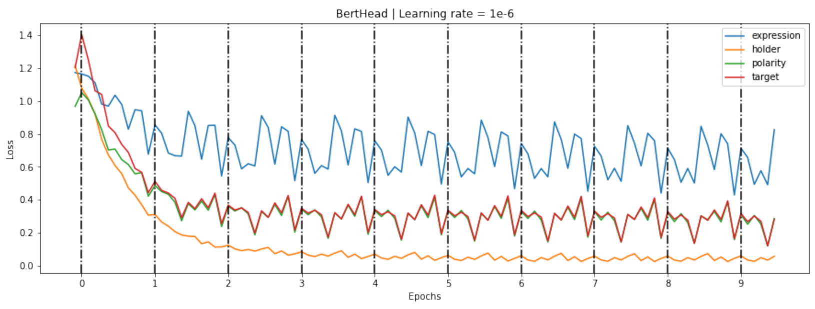


Figure 4.1: Same loss for all tasks.

Despite producing measurable decreases in loss, the various model configurations we initially tested achieved no score over 0 for our main aggregated F1 metric for many epochs. There were a range of possible explanations for this phenomenon. From potential bugs in our preprocessing, to the sheer lack of complexity of this simple model, we brainstorm and discuss some of the leading theories as to what was happening here below.

4.3.1.2 Subtask deactivation

Looking back at the results presented in *Applying Multitask Learning to Targeted Sentiment Analysis* (Pereira, Halvorsen and Guren 2021), we decided to test a variant of our architecture that only extracts targets and classifies polarities. Expression and holder annotations were excluded from that experiment's data set. Applying such a set-up to our own experiment would allow us to compare our results with theirs, ensuring our architecture was learning similar to the models presented there.

Luckily, our `BertHead` was built with flexibility in mind, and could handle such an architectural change. Subtask deactivation through model parameters is a functionality that was lacking from both the RACL and the IMN setups, each of which had rigid, unique data flows for their subtasks[9]. While some complex system may need to require specific subtasks, it is convenient to have the ability to train these simpler models on single tasks at a time.

Unfortunately, this in itself did not solve our problem. Our models trained only with target and polarity subtasks still evaluated to 0, even though loss decreased as expected. Compared to the results from *Applying Multitask Learning to Targeted Sentiment Analysis*, we deduced that there was an unseen bug somewhere in our base class. Subtask deactivation did however get us thinking about the importance of some tasks over others, and the different levels of difficultly each task carried.

4.3.1.3 Individual learning rates

The beauty of separate components for each subtask is that individual optimizers could be implemented and updated per task. Isolating subtask parameters

[9]To be fair here, we should note that IMN allowed for training on document level annotations as an auxiliary task, which could be turned on or off via a parameter, however the three subtasks were required for all training.

to each their own optimizer helps penalize components correctly during back-propagation, since only the loss relative to the model's performance on a single task is applied to that task's parameters. Separate task-wise optimizers in turn meant that individual learning rates could be configured and tuned per task.

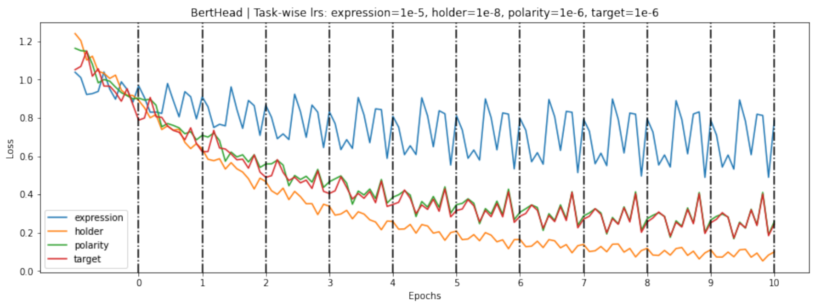


Figure 4.2: Task-wise learning rates with expression higher and holder lower than polarity and target.

Earlier, in Figure 4.1 we saw that our model learned holder-labels very quickly (with little fluctuation), and learned the least for the expression extraction subtask. We therefore set task-wise learnings rates accordingly: holder as very low to slow down learning here, expression higher than before to potentially learn more each step, and maintained the same rates for polarity and target. Figure 4.2 shows the resulting losses from this configuration for 10 epochs.

A similar pattern of losses is seen with task-wise learning rates as to those previously seen when all tasks had the same rates. Holder still decreased the fastest, followed by target and polarity, both of which decreased almost identically, and finally with expression decreasing the least, but with a high variation. Unique learning rates for holder and expression also had an effect on the unchanged polarity and target subtasks. We observe a flatter slope for all tasks, likely due to more noisy signal updates to our NorBERT2 parameters. It can be assumed that increasing expression's learning rate confused the model by penalizing the shared embeddings more with the highly variable expression loss, consequently allowing less information from our clearer signals from holder, target, and polarity through.

We let the same job for task-wise learning rates train for 200 epochs, to measure its effects over more iterations. After about 50 epochs, this model finally start giving some results for our main metric, albeit very small compared to results from the baselines. The best performance reached was a mere 0.11 at the end of this run. We did not however see any drops in our hard metric, telling us that this model could have probably trained for longer and performed (slightly) better. We stopped training after 200 epochs to limit resource usage, since performance for our metric seemed to reach an asymptote around 0.12.

Figure 4.3, shows the losses for these 200 epochs, We can see that expression loss slowly decreases, even with a higher learning rate than the rest of the tasks. Expression was a tricky subtask to calibrate, since the overall loss only decreased slightly, but loss between batches had a high variance. When referring back to the preliminary results presented in the paper introducing the NoReC$_{fine}$ data set (Øvrelid et al. 2020), we see that expression also scored quite low on the more strict metrics. From this, we concluded that expression must be a

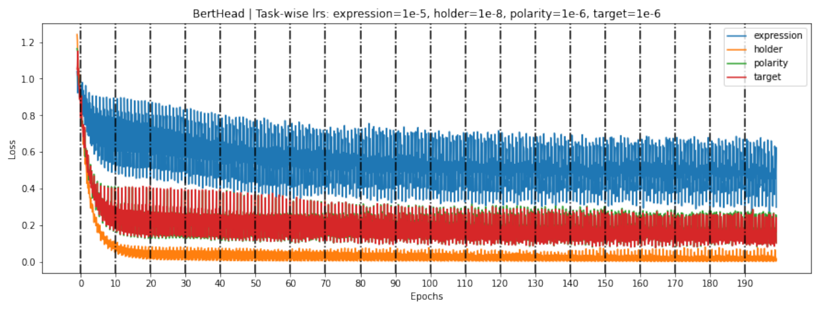


Figure 4.3: Task-wise learning rates for 200 epochs.

very difficult task to estimate, at least when exact-match scopes are needed for high performance. We decided this subtask needed more space to represent the intricacies of its labels, meanings we needed to increase our model complexity, picked up again in Section 4.3.3.

4.3.1.4 Debugged performance

During initial development, we assumed this simple model was performing poorly on our main metric due solely to a lack of complexity. This low score, along with the high variability between batches for some tasks, made interpreting this model's ability to perform our fine-grained sentiment analysis task difficult. We originally moved on from this `BertHead` model while it was still only giving 0 for our main metric. As will be mentioned in the next sections, increases in complexity did not solve our low performance in itself. However, the more complex models forced some debugging of our base class, specifically in our evaluation sequence, as well as our initializer methods.

When we came back and tested `BertHead` after the more complex models had been developed, and a few of these bugs removed, we found major performance improvements on this base model. The main bugs will be mentioned in later sections, according to where they were found during experimentation. We present some of the findings for the performance of the debugged `BertHead` in this section for consistency.

Figure 4.4 shows a run from one of the first studies checked after debugging, using similar parameter values as studies presented above. Already within the first five epochs, we saw jumps in performance close to the same range we would expect perfectly fine-tuned variations of our baselines (trained from their source code) would have been. When these plots were generated, we regained confidence that our `BertHead` model was properly built as a flexible base for more complex fine-grained sequence-labelers.

We could now also check the total effects of a BERT head truly has on the fine-grained sequence-labelers built for our experiment. In a small study we called "Frostbert", we froze the parameters for the internal NorBERT2 model, and trained only the task-wise linear components on their respective losses[10]. The results of this study showed that linear outputs can be trained to

[10] In all the previous experiments, we were using a default of `True` for optimizing over BERT parameters for all tasks.

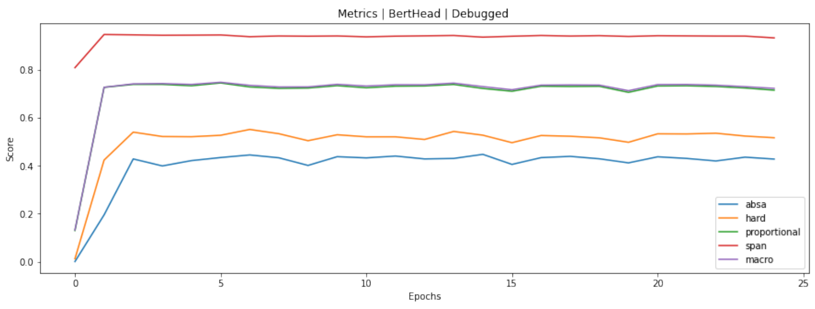


Figure 4.4: Jump in performance after debugging from more complex models.

achieve some performance over 0, but nothing compared to when NorBERT2 is fine-tuned.

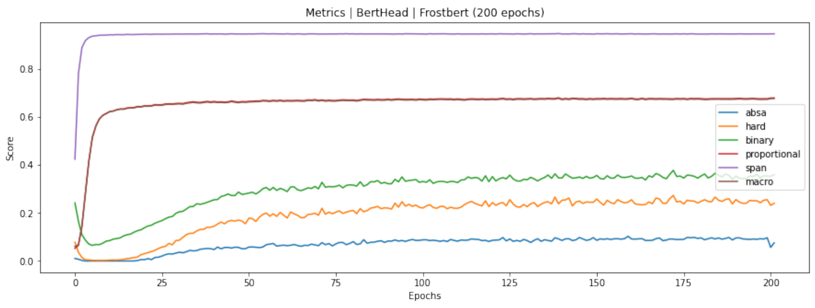


Figure 4.5: Metrics for Frostbert model over 200 epochs.

For the Frostbert study, we first checked if any gains could be found after 30 epochs. When stable, but small increases were measured for these shorter tests, we decided to let a Frostbert model train for many more epochs, just like above, to see how this performance play out over time. As Figure 4.5 shows, the scope-focused metrics increase from about 20 epochs until around 75 epochs, where they seem to stabilize near an asymptote.

Compared to the immediate jumps to about 0.40 on the debugged models with fine-tuning NorBERT2 already activated, the Frostbert models took much longer to score anything on our strict main metric. From this comparison, we can conclude that the use of NorBERT2 embeddings in a model focused on fine-grained sentiment analysis as a sequence-labeling task helps performance in two ways. The top aggregated F1 score the model achieved increased by over 30 percentage points, and number of iterations needed before reaching any such scores was shortened by at least 15 epochs.

4.3.2 `Study`: a class for hyperparameter searching

Before diving into the next architecture, we need to discuss the system we developed for cleaner hyperparameter searching. By this point in the experimentation, we began to encounter problems when storing logs for multiple hyperparameter searches running simultaneously. Jobs often took a few hours to run, so multiple instances of the same models, with slightly different hyperparameters were queued in parallel. However, with multiple jobs running at the same time, it was easy to mix up configurations, leading to messy logs and hard-to-decipher slurm outputs when certain configurations would fail.

As architectures grew more and more complex, so too would the number of hyperparameters to tune per model, thus exacerbating these problems. This ultimately motivated us to develop a `Study` class, that would read hyperparameters from a configuration file, train and evaluate the respective models, and find which of the specified parameter(s) gave the highest performance. This allowed for more structure about which jobs would be stored in which log files, and helped keep track of configurations through git-commits[11].

A configuration file was a JSON formatted dictionary containing all the parameters we wanted to specify for that model, along with which model we wanted to build, and the file-name prefix where the logs should be stored. The JSON files were stored in a folder off the project's root directory, called `studies/`. Every model was given their own directory within `studies/` to make sure configurations remained organized. Our `study.py` file would read in the JSON file specified via a command-line argument, find which hyperparameters were to be tested (i.e. those provided as lists), conduct the necessary runs to check those parameters, and save the model giving the highest score in a public directory on the server. That way, we could reuse model checkpoints that performed very well during a particular study in future experiments.

Search technique When running hyperparameter searches, there are a few different techniques developers can follow. An extensive *grid-search* takes in all the different values for the hyperparameters one would like to check, and trains and evaluates a model for every possible combination of those values. *K-fold cross-validation*, often used together with grid-searching, trains and tests a specific configuration on k different splits of the training data, the final result being an average of the scores found for each split. The two of these combined can be quite computationally expensive, quickly increasing job run time and resource consumption the more hyperparameters a developer wants to check[12]. We argue that a cross-validated grid-search should only be run when developers are certain everything in their model is working as it should be.

Our `Study` class took inspiration from these techniques, but followed a linear Big-O Notation, $O(p \cdot n)$, rather than $O(p^n)$ as describe above, where p is the number of hyperparameters checked in a particular study and n is the number of values tested per parameter. We achieved this lower computational budget by writing our `Study` class to only check a single hyperparameter at a time. To mimic the stability provided by cross-validation, we often included three runs at the end of a study, each "checking" the same value of epochs. This way, after a study was complete, we would ultimately have trained four models on the same configurations (one from the parameter search, and three from the epoch

[11]It was also helpful here that most development was carried out and tested locally, then pushed to our servers. Also writing these configuration files made sure that all previous configs were tracked in our version control.

[12]If a developer wants to check 3 hyperparameters, each with 4 values, using a 5-fold cross-validation, a total of $(4^3) \cdot 5 = 320$ models would need to be trained and evaluated. Considering a set-up where a single model takes only an hour to train, this search would need about two weeks to complete!

runs), which could be smoothed and plotted. As mentioned in Section 3.3.1, we used a default of 30 epochs for each hyper search to limit train time.

A major downside to our $O(pn)$ hyperparameter searching is that we do not cover then entire hyperparameter space a model could possibly span over. In particular, we do not capture possible co-dependencies of different values across different hyperparameters. However, we argue that these linear checks provide enough coverage for the analysis we were trying to conduct in this experiment, while allowing us to maintain a reasonable resource budget.

We acknowledge that other search techniques that limit resource consumption but cover a broader area of a model's parameter space could have been implemented here. A very simple variation could have been a *random search* that randomly selected a proportion of possible parameter configurations to test simultaneously. This eliminates the linear checks of single parameters at a time, covering a large parameter space, while still maintaining an $O(pn)$ complexity. A *maximum likelihood guided search* could have also been implemented as a simpler form of a grid-search. Here, a mix of random searching and maximum likelihood gradients are guided by weights according to loss improvements (Welleck and Cho 2020). Neither of these were implemented in our project, as we chose to stick with the absolute simplest search technique to speed up development, a flaw that will be discussed further in 6.4.

4.3.3 Increasing complexity

Inspired by the downstream architecture BERT models are trained on, we decided the next natural iteration on our `BertHead` model was to add *long, short term memory recurrent neural networks* between the embedder and output layers for each task. This gave each subtask the freedom to learn patterns across various ranges of inputs, through the LSTM's internal gates, covered in Section 3.4.3.

Development on the `BertHead + LSTM` model, named `FgsaLSTM` in our project, started before any score over 0 for our main metric had yet been observed. With this in mind, we added a softer metric (token-wise F1 score averaged over subtasks) into our evaluation scheme, to get more perspectives of how learning was occurring in our models. Previous to this implementation, we were only sure our models were learning due to decreases in task-wise losses per epoch. Flat scores for our main metric gave us no insights on what the decreases in loss actually meant. The easier metric introduced now checked for any correctly labeled token, much more lenient than our exact match main metric[13].

4.3.3.1 Hyper-searching `FgsaLSTM`

The new components introduced in the `FgsaLSTM` model brought new hyperparameters we could tune in addition to those from `BertHead`, namely `bidirectional`, `hidden_size`, and `num_layers`. As can be assumed, parameter checks on hidden LSTM configurations increased model run time incrementally for every new layer, node, or direction the data was needed to flow through.

Losses per task behaved quite differently for `FgsaLSTM` models than for `BertHead`. In Figure 4.6, we see the losses for a single run of a study on hidden

[13]This metric was still an F1-score, so precision (how many of the estimated labels were actually correct) also captured how often our mode over predicted labels.

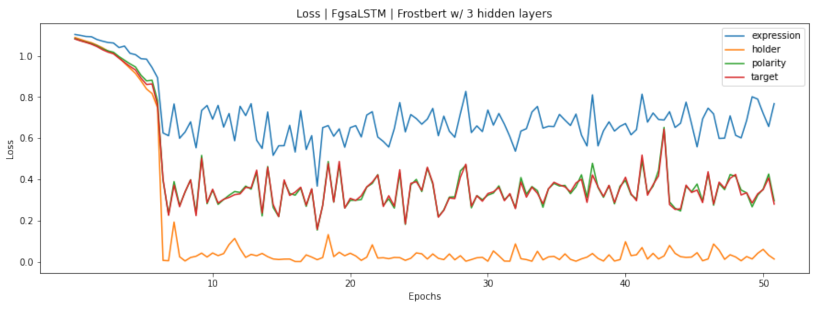


Figure 4.6: Task-wise loss for a single run on FgsaLSTM.

layers. We observe an initial stable decrease of losses, followed by a large drop around the sixth and seventh epochs. From here on out, we see high variance in the losses for all tasks, a pattern observed in many of these studies.

We tried to mitigate this loss oscillation by lowering the learning rates for all tasks, which is why Figure 4.6 was allowed to train for 50 epochs (instead of the default 30 epochs). However, after that initial drop early on during training, it seems all tasks struggled to learn anything valuable. Other studies gave similar results, hinting that a mere increase in complexity was not enough to solve our task. We concluded here that we needed to implement some communication between subtasks if we were going to have any chance of picking up on the true signal for each of the subtasks in our models.

While our newly implemented, less strict metric scored well over 0.60 for many of the configurations tested for `FgsaLSTM`, our main aggregated F1 score remained at 0 for the number of epochs we were checking. For some setups, specifically those with fewer hidden layers and node counts close to 768[14], we saw small blips of our main metric over 0. These lasted only a single epoch or two before falling back to zero, and usually occurred after about 30 epochs. When comparing metric performance against the losses for the same run, we found no correlation between these blips and loss.

These poor results along with the high variance in loss convinced us that this LSTM-based setup was not the correct direction we needed to take our models in. Mere complexity increases were not going to solve our 0-metric problem. We needed to isolate exactly what was giving such poor performance. Was it inevitable that these non-interactive systems were too simple to score well on a strict exact-match metric? Or was there some other bug in our base class that prevented any real learning from occurring?[15]

4.4 Baselines in PyTorch

At this stage of experimentation, it was problematic that RACL and IMN architectures trained on their source code gave results for the NoReC data, but none of our architectures built from a `BertHead` parent had yet given any results for the same metrics. This told us that a big puzzle piece was missing

[14]768 is the embedding size outputted from our internal NorBERT2 model.

[15]Spoiler alert: The latter was in fact true, but the final bug was not found and discussed until 4.5.

from our models, compared to the baselines. Some time was spent debugging this problem, specifically focusing on preprocessing, data-loading, optimizer configurations, and general data-flows during forward and backward steps. However, after all brainstormed sanity checks were conducted, our models were still scoring very low.

In order to check if the missing link was merely the lack of complexity of our `BertHead` and `FgsaLSTM` classes, we decided to reconstruct the baseline architectures in PyTorch, using `BertHead`. This gave us a standardized sandbox for comparing all the architectures used throughout our experiment. The idea here was that if we can get measurable results with as much overlap between models as possible, we could rule out eventual differences between frameworks or other errors innate in our flexible implementation.

In theory, this should not have been necessary. Architectures built in different frameworks (PyTorch, Tensorflow, Keras) should all be comparable between each other. We also knew that the source code for the baseline models were already tried and tested by other scientists around the world, ruling out any "unfairness" in favor of those models. Contrarily, our models were still largely a work in progress, and it was possible they were missing an important step somewhere that we were not able to find. Thus, reconstructions of the baselines in PyTorch were deemed necessary to eliminate any possible differences between models, except for their documented component nuances.

We would like to point out that this re-implementation required more than a simple plug-and-play configuration. These models are quite complex on top of the native framework discrepancies. Proper reconstruction required thorough understand of the motivation behind every little transformation, which was a time consuming step in our development.

This section details how re-implementation of both IMN and RACL affected our understanding of fine-grained sentiment analysis systems. The main architectural differences have already been covered in 3.5. Therefore, we will only comment on the important pieces, and rather focus on how these torchified architectures performed relative to our own and how they helped us in debugging our code.

4.4.1 Reproducibility nuances

Most papers documenting modern machine learning projects attempt to outline a reproducible experiment so that the any eventual state-of-the-art findings will be accepted by the experiment's respective community. However, when developing and documenting large-scale experiments, containing thousands of lines of code and even more logs and notes taken along the way, it is easy to get lost in the details of one's own understanding of the work. Abbreviated variable names, sparse comments, and limited doc-strings might be enough to remind the initial development team exactly what a piece of code is useful for. For completely new eyes trying to extrapolate the essential pieces of the projects, these abstractions are often quite hard to interpret. Such was the case for both the IMN and RACL architectures.

Both projects had a paper describing the general data flows of their system, along with visualizations on how the interacting components communicated with each other. Naturally, these explanations lacked the subtle details concerning intentions behind certain concatenations, normalizations, and other internal transformations of the data that could potentially have a huge impact on the information available between components. Additionally, many of the components were hand-written, requiring intensive studying to ensure that our interpretations of the components matched their true implementations.

4.4.1.1 Interesting architectural choices

When rewriting these architectures, we tried to include as many as possible of these transformations and hand-written components found from the source code in their respective PyTorch set-ups. Though, we admit, some of our understandings and implementations must have been different from the original,[16] because we found that our models trained better when a handful of these undocumented transformations were dropped.

Activation functions For example, in IMN, a soft-max was used after the linear outputs were generated, reducing the model output values for a given task to a probability distribution, summing up to 1. This made sure the model outputted logits, instead of non-normalized output values, i.e. with some values much larger than 1. When implemented in our PyTorch version of IMN, soft-maxing our outputs caused our strict performance metric to remain at zero again, with no hope of changing unless loss weights were incredibly high. Inspiration form RACL motivated us to drop these soft-max activations. This gave performance improvements larger than we had observed for any of our other architectures. Our reasoning for the increase in performance when these activation functions were dropped was that maybe it was affecting the amount of loss that got back-propagated per task, limiting penalization on parameters producing very wrong estimates. However, in such a case, tweaks to our learning rates should have then been able to fix this problem. We chose to leave the use of these activation functions as a configurable hyperparameter, for an analysis in future work.

Gold Transmission A key component, implemented in both IMN and RACL, that proved to be quite useful was the subtask transmissions. *Transmission* here is referred to as the passing out logits outputs from individual subtasks to attention inputs, sometimes blended with the gold labels included in the batch. IMN allowed a percentage of true labels to affect this transmission, according to the current epoch. When first browsing through the code, it almost looked like this was a form of cheating. However, comparisons between the two observed implementations provided evidence that these transmission steps were in fact *not* cheating. The influence that true labels had on attention inputs decreased as epochs increased. This feature was in fact documented in the IMN set-up, although was initially missed on our read-through of the experiment, since it was ony mentioned in the Appendix. The comments included in the RACL code helped clear up this misunderstanding by our part.

[16]A typical problem of reproducing previous systems is deciphering these undocumented components, sometimes producing unobservable system variations(Fokkens et al. 2013).

4.4.2 IMN (PyTorch)

It could be argued that increasing complexity when simpler models are scoring very low is a dangerous tactic. More complex architectures add more potential bugs to a system, making any future debugging even more laborious. In retrospect, we agree with such a claim due to our first hand experience with this cumbersome debugging. At the time of development, however, building a new version of a reliable architecture seemed like one of the only unchecked options left for solving our low performance issues.

Considering the discrepancies between frameworks and some hand-written components, initial implementation of IMN using a `BertHead` was relatively smooth. The first torchified baseline model was training and showing reasonable decreases in loss within a single workday of development. Admittedly, initial performance gave similar poor results to the other models we had tested.

Due to configurable optimizers built into the IMN source code, we decided to run a quick optimizer tuning study during initial development, to ensure we were using reasonable configurations. Through this hyperparameter search, we only observed changes in loss, since our metrics were still not giving reliable results. We concluded that our default optimizer `Adam` with an updated parameter value `weight_decay=0.5` gave the most reasonable loss outputs yet seen during experimentation[17].

4.4.2.1 Debugging help

Loss weights With our "tuned" optimizer using a weight-decay parameter, we determined it was necessary to test different loss weights on our setups. A loss weight tells the loss function (in our case a cross-entropy loss) which labels are more important to predict than others. A fun experiment on a model struggling to predict a label can be to set loss weights for that label incredibly high, and only measure recall (i.e. how many true labels the machine missed). Often, the system will learn to perfectly estimate those labels, but also give every other token in the sentence similar labels, even those tokens that should obviously not have any label. This at least hints to some peak loss weight value in between zero and the very high weight chosen, that balances label importance as best as possible.

Recall that our data set had three different attributes using BIO-labels, namely target, expressions and holder. Looking at the distribution of labels in each of these attributes individually, we found that about 90% of the tokens were labelled as $O = 0$, meaning these tokens were not in the scope of an expression, a holder, or a target, depending on the attribute being observed. While the difference between B and I labels could also be measured, we concluded the ratio between O versus B and I together was by far the largest, and therefore the most important to inform our models on.

Keeping this imbalance in mind, we set loss weights such that B and I were between 10 to 50 times as important as an O label. This change gave the first increases in our main F1 metric we observed during our experiment. The newfound importance of loss weights made us go back and test different loss

[17] This tuning by only observing losses later proved to give us a suboptimal configuration. This parameter value was deemed one of the last major bugs of our more complex models, documented in Section 4.5.2.

weights on both the `FgsaLSTM` and `BertHead`. These checks gave similar increases in performance. While we were no longer stuck at 0 for this main metric, no stable score over 0.05 was measured after any number of epochs tested. This meant there was still a bug in our system, even though some improvements for some label predictions were obtained.

First prediction bug Through observing outputs from different loss weights in a Jupyter Notebook sandbox, we stumbled onto a bug that was affecting all of our previous models built so far. Namely, the method generating model predictions had a faulty tab-index, causing the first predictions of a batch with sequence-length n to be copied n times. The rest of the predictions of the batch were disregarded, resulting in evaluations only for 1/32 of the data, much of which included mismatching padding elements.

Very high loss weights gave small increases in performance, likely due to a few first rows of batches that actually had some labels in them. Cleaning up this bug increased performance for our main metric to score around 0.10 after about 25 epochs for our best configurations of IMN. Similar results were measured for the simpler models detailed in 4.3. See Appendix B.1 for a full description of this bug.

It is important to note that this was not the only bug built into our base system. The remaining bugs were found during development for RACL and `FgFlex`. The rest of this section covering our torch-based IMN network presents and analyzes some initial results found after all these bugs were removed.

4.4.2.2 Hyperparameters

The tunable hyperparameters of our implementation of IMN, also included by the original authors, were shared and task-wise layers, interaction count, convolutional network dimensions, and optimizers. In addition to these parameters from the source code, we added a a few novel tunable parameters, to increase the flexibility of this setup, namely: configurable query, key and values for the polarity attention component, a scope finding auxiliary task, task-wise learning rates, loss weights, and trainable embeddings. Instead of going in-depth in all of the studies run for these hyperparameters, we chose a few of the most interesting ones that gave stable results, to discuss here.

Learning rate Pre-debugging studies had shown high variability of losses for each epoch, even when only measuring the same batch, i.e. the same data. Due to this, default learning rates were set very low, to `1e-8`, as an attempt to dampen the variability measured per epoch. In reality, such a low learning rate only slowed down learning. Therefore, when we were sure our model was debugged, we ran a new study on the learning rates needed for an IMN model. We checked everywhere from the default specified in the original set-up, all the way down to our very low `1e-8`.

To our surprise, we found that the default configurations presented in the IMN paper scored among the worst of the range of rates tested. Learning rates of `1e-5` and `1e-6` gave the most stable results for a 30 epoch run. Both these rate gave initial jumps in performance around 10 epochs, due to gold transmission, then plateaus after the true-labels in transmission wore off. `1e-7` showed increases in our more strict metrics after a little over 17 epochs, remaining at zero up until that point.

From these findings, we concluded that a new default of `1e-5` should be used, allowing for the learning rate scheduler to reduce to stable `1e-6` if/when task-wise losses showed signs of increasing.

Layers Both shared convolutions and task-wise components had expandable layer counts, configurable via hyperparameters. It was through studying these parameters that we obtained our highest results for our own implementation of the IMN architecture.

A study checking values from 1 to 5 layers for each subtask, along with the same for shared components was run. Even with an easier epoch limit of 25, this study still required over 13 hours to run, given we tested 25 unique model configurations.

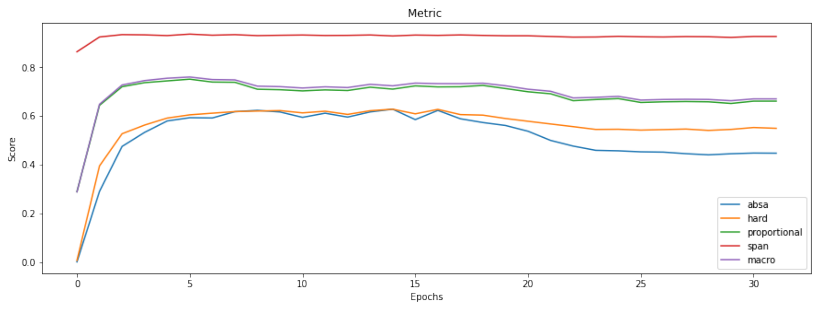


Figure 4.7: Best layers config smoothed over 5 runs; Shared: 3, Expression: 2, Polarity: 1, Target: 2

Figure 4.7 shows a smoothed output from four runs for this study, one where the configurations were tested for the first time, and the other three where epoch of 25 is the parameter being "studied". As can be seen in this figure, an initial high score around 9 or 10 epochs is reached for the development data, before a slight decline around 15 epochs followed by a plateau. At first glance, one would think this meant our model was over-fitting. However, keeping the gold transmission in mind, this decrease in performance at these early stages is really the weaning off of true labels being transmitted to attention inputs. Our main metric (blue) eventually plateaus around 0.44 after the gold transmission has completely worn off. The loss plots for this part of training (See Appendix A.2) show values relatively close to 0, compared to many of the other losses shown in models above.

Out of curiosity, we tried running this configuration on more epochs, to see how training plays out over more iterations. We see the peak at between 10 and 20 still only lasts for a maximum of 10 epochs. From epoch 25 onward, performance maintains the same plateaus observed in Figure 4.7.

While many of the layer configuration gave similar results, there were some outliers that should be noted here. For example, when the polarity task had more than a single layer, the final plateau around 25 never appeared, but metrics kept decreasing downward. Even though only single runs were check for values 2, 3, 4, and 5 here, we saw a clear pattern that performance metrics always continued downward after the first initial peak at 5 epochs. With this we assume that too many layers for polarity obscures the information found in the upstream subtask components for target and expression extraction, causing this drop in performance.

Another configuration to point out was target layers greater than 1. It seems like many of the plots are still trending upwards when our training was cut-off. We had already observed for the best configuration that eventually this second spike in performance reaches a peak, then starts to decline. However, because our epoch cut-off was chosen semi-arbitrarily (based off what seemed best for all configurations on all models), we could have maybe rerun this study with slightly longer epochs, eventually with an early-stopper, to see if more iterations would have actually given higher scores greater than those obtained by the best configuration found above. We let this extra analysis be saved for future studies, since our current high score was already quite high.

The final score for the 5-run averaged plot shown above after our 30 epochs was 0.44. This smoothed score exceeds any results our own novel models had reach, and seems to be well over the possible best results from the IMN source code. Given that every data set will give their own unique high scores, directly comparing this score to those high-scores obtained by the English is not entirely fair. However, this score does seem consistent with the best results presented for the IMN architecture with BERT embeddings in the baseline papers.

Other studies worth mentioning Before moving on to the analysis on the RACL architecture, we provide a quick comment on the query, key, and values studies for IMN along with the scope finder auxiliary task.

The ability to adjust the inputs to an attention mechanism was one of the novel architecture decisions we implemented in our version of IMN, and later carried over to our flexible `FgFlex` model for deeper analysis in future experiments. We felt it would be particularly interesting to study how these different attributes might be related through the context of their sentences. RACL argued for the same, and built rigid relations between all these attributes. We wanted to check which of these combination of attention relations gave the highest performance on an IMN set-up. Our findings showed us that the original IMN setup chose the best inputs possible for a single attention block, using gold-transmitted expression subtask information as query, and polarity information as the keys and values.

We built a scope finding auxiliary task when our models were still evaluating very low. The idea was to teach the model where scopes ended, in order to increase loss weights more, given that loss weights initially gave good results. The models we were able to test with this task activated showed that scope did have slight effects on task performance, though not much. Because we were focused on fixing our metric problems, we eventually deactivated this task by default, and never reran tests when all the bugs were found. We still believe that models reliant on high loss weights could benefit from scope predictions. Although, due to time restraints, further analysis on this had to be saved for future work.

4.4.3 RACL (PyTorch)

Reconstruction of the torch-based RACL model was similar to IMN, few handwritten components caused some confusion and difficulty when implementing.

The model was also built and ready to train within a day's work, although some bugs took up to a few weeks to hunt down and get rid of. As mentioned when discussing the source code in 4.2.2, a RACL model requires a considerable amount more of space to run. This is due to the extra attention components, and the stacked architecture.

When we started testing out our implementation of RACL, we were still dependent on loss weights to get any scores for our main performance metric. Through small adjustments to this system, we discovered a few other model details that were restricting learning. In this subsection, we describe how analysis on our torch-based RACL model unfolded, pointing out the bugs we found, and detailing a few of the hyperparameter searches done on the final, debugged model.

4.4.3.1 Initial performance

Low learning rates Probably the biggest constraint on learning for our first version of RACL was a very low learning rate of `1e-8`. We were still convinced that stable optimization required such small increments for every learning step. Comparing these initial studies with the debugged ones, we see that all training took much longer due to these excessively small rates, with no added benefits on stability. Slightly larger learning rates eventually gave the same amount of loss variability as the smaller rates, but trained much faster. We decided a learning rate of `1e-5` was more reasonable to use for this architecture, as well as many of the previous architectures.

From this, we learned that such a minute detail, like our over-confidence of such a small learning rate, can heavily affect any potential findings for a given architecture. When we thought we had already determined lower rates were necessary, potentially going back and checking new values seemed like a waste of time. Had we implemented a another hyperparameter search technique, we might have caught this constraint earlier on.

Loss weights Loss weight has been mentioned for some of the other architectures as the first hyperparameter studied that showed any results for our main performance metric greater than 0. RACL required the lowest value for this parameter for all of these architectures. While the other models needed loss weights between 10 and 30 to give their best results (in their buggy versions), RACL gave best results with weights somewhere between 5 and 7. This was surprising for us, since we would have expected this parameter to use more or less the same value for all architectures, dependent mainly on the distribution of annotations in the data set. We believe this lower dependency on loss weight is likely due to the attention relations between many of the subtasks. The increased communication between tasks assumably helped the model learn label importance for each subtask, based on how the same was learned for the others.

4.4.3.2 Debugging help

Activation functions The original RACL network did not include activation functions in their linear output components. This motivated us to try removing the same for our implementation of IMN. The resulting architectures showed jumps in performance, thus motivating us to build this functionality as configurable through a hyperparameter. While this may not particularly be considered a bug in itself, exclusion of activation functions proved beneficial in our novel `FgFlex` architecture as well. We note the future work could potentially further explore why this proved so beneficial for these architectures.

Opinion transmission The implementation of a gold-transmission-like feature in RACL, denoted *opinion propagation* in the original architecture, helped increase our understanding that the respective feature in IMN was not intentional cheating. However, RACL did not use any information from the true labels. The only information transmitted was the prediction outputs found in the expression subtask. This was good because when testing the main code, we only trained for 7 epochs before concluding scores were high enough to move on. If a gold-label transmission similar to IMN would have been implemented in RACL, with a default warm up constant of 5, about 55% of the true labels would have still been being transmitted to attention inputs. *That* would have been considered cheating!

The lines of the RACL source code where opinion-propagation is written also include a short explanation as to how their implementation differs from IMNs. We appreciated these comments, and learned that this feature was intended merely shared information between tasks, and in IMN's case, guide the complex attention block in the right direction of learning during the first few epochs. Due to this new understanding, we configured out set-ups to use a default constant of 3.5, and trained over 30 epochs, thus avoiding any cheating of true label transmission.

Normalization of attention inputs A potential bug still present in our final RACL architecture was normalization of the attention inputs: query, key and values. We included this feature in our implementation, using the PyTorch native `torch.nn.functional.normalize` method. From our experimenting, we did not see any increases or decreases in performance when this feature was added or removed. This tells us that either our implementation was wrong, or that this step had little to no effect on model performance. The final version of our RACL left these normalizations present all the places RACL included them in their original implementation.

4.4.3.3 Hyperparameters

Learning rate The learnings rates we tested for our RACL system ranged from `1e-3` down to `1e-7`. Our study found the same learning rate of `1e-5` to give the most stable results for both loss and performance metrics. See Appendix A.3 for all of the results from this study.

The optimal range of rates seemed to span over a similar range as for previous architectures, between `1e-5` and `1e-6` giving stable results. A rate of `1e-4` looked to be training well for the first few epochs, before dropping performance metrics to 0. Such behavior is expected for learning rates that are too high. Too large of learning steps can cause over-fitting on training data by misinterpreting that set's nuances as true annotation signals. We can also observe the loss for `lr=1e-4` is quite variable, compared to plots from smaller learning rates. A learning rate of `1e-6` gave performance metrics close to those from `1e-5`. However, when looking at the loss for this run, we see that no task is being learned optimally. The plots for `lr=1e-5` show steady decreases in loss for all tasks, and stable metric scores across all epochs. Therefore, this rate was also chosen as the optimal learning rate for our RACL architecture.

Stack count On our debugged RACL model, the stack count study we ran showed similar results for counts 1, 2, and 3. We would have expected the expanded stack counts would increase performance, since stacks allow for more information sharing between all subtasks, but this was not observed in our study on stacks. If a more complex system takes twice as long to run but achieves the same score as simpler one, then the simpler system is the preferred alternative. Such is the case with stack counts for RACL. Because we do not get any performance increases with stacks greater than 1, we conclude that single stacks for our implementation of RACL are the optimal value for this hyperparameter.

The best configuration of our RACL implementation found through hyperparameter searching achieved performance similar to `BertHead` and our IMN version. This tells us that the set-ups we covered in our studies provided little to no improvements on our overall task. Considering the original architecture was presented as better than the IMN+BERT model, we can conclude that there is a possibility future developers can unlock this untapped potential through more extensive hyperparameter searching.

4.5 Testing flexibility

With the perplexing components of the baseline source codes understood, recreated, and tested, model performances of the reconstructed baselines began to reach expected values, with IMN giving the best results. This meant we were ready to build our novel architecture, focused on flexibility. To do so, we extrapolated the best components of the baseline models, and added some adjustments we felt could possibly increase performance. `FgFlex`, the resulting model, was developed, trained, and tuned as the final engineering task of this experiment.

4.5.1 Novel flexibilities

In Section 3.5.5, we outlined the general structure of our `FgFlex` model. Much of this model was directly inspired by our torch-based baselines, although this second round of development over these systems helped synchronize even more of the distinct model components.

For example, initialization of standard component blocks, such as our internal convolution networks, attention mechanisms, and linear components, was refactored into methods in `BertHead`. Previous implementations rewrote each of these components in every model's unique `init_components()` method. Standardizing these initializations as part of the parent class allowed us to be sure that all models were using the exact same configuration for these components, thus eliminating any discrepancies between similar components across different models.

We also reset our default parameter value to those given best results in previous hyperparameter searching. An important example here was the ability to fine-tune NorBERT2 or not. Since we had yet to find an architecture that scored better when BERT fine-tuning was deactivated, we made sure our parent class defaulted to fine-tuning NorBERT2 for all classes. Any time we would want to run a new Frostbert study on a new architecture, these parameter changes were specified and tracked through our configuration files.

4.5.2 Discovering the final bugs

Again, when we started testing this architecture, loss weights were vital in order to achieve seemingly good model performance. Our `FgFlex` model needed weights much higher than RACL, with some optimal value somewhere around 33, compared to RACL's 7. Discussions with other members of the Language Technology Group at the University of Oslo showed us that this dependency on high loss weights was likely a flaw in our architecture, thus motivating to continue the hunt for bugs. These research group members convinced us that loss weights should not be the sole factor needed to lift a model's performance above 0, even for strict metrics.

Due to previous success with small-scale sandbox experimenting, another Jupyter Notebook was created for the new round of debugging. Here, we mimicked our training procedure, step-by-step, and observed how the data looked at every step.

Sub-word evaluations We found a difference between how our models were producing estimates versus how those predictions were being evaluated. Recall that a `BertTokenizer` object splits large words into smaller sub-words, based on the vocabulary in the tokenizer. Our models were then told to ignore all the tokens referring to sub-words, except for the first. This was to ensure that longer words were not penalized extra when wrong solely due to the fact they contained many sub-words.

While our models were ignoring these tokens during training, there was no such ignoring happening in our evaluation methods. Specifically, our main metric, implemented by the original RACL authors, and merely copied over to our experiment, checked for exact-match scopes, regardless of sub-words. This meant that we needed to adjust this metric to also ignore sub-words in the same manner as our models were. This fix had an effect on all our models, increasing performance by at least 20 percentage points each[18].

Weight decay Another "bug" we found was really a hyperparameter we set early on during development. Weight decay, originally tuned on our first suboptimal IMN implementation, seemed to restrict training significantly. A quick search on this parameter again suddenly showed major jumps in performance when the parameter was set to 0. This again had large effect on how our other models were training, and is as the last major bug we found before the final results for all architectures were found.

[18]When documenting this step, we realized that adjusting the metric code used in the baseline experiments was not the best solution to this problem. Even though it fixed our problem, it open the possibility for us to "cheat" with our metrics. Instead, we should have filtered out the ignored sub-word predictions in our `predict()`. More on this mistake in 6.3

4.5.3 Hyperparameter search

Even though our flexible model was our novel contribution with this experiment, we tried to limit hyperparameter search here to match that done on the other models, in order to remain fair. The following sections discuss in detail two of the novel searches to this architecture, followed by some relevant other findings discovered through these studies. A notebook containing all of the different searches checked, together with some discussion along the way, can be found in the project repository[19,20].

4.5.3.1 Attention relations

Inspired by the increased attention relations implemented in RACL, we wanted to further study which combination of subtask attentions provided the most efficient model performance. Every new attention relation adds over 1,500 new parameters to a model per stack, each of which required updates for every optimization step. Finding which of these relations between subtasks actually helped performance could help eliminate unnecessary complexity of future models.

Our `FgFlex` model defaulted to building an attention relation between all subtasks the model trains for. If a model was configured to train on all four subtasks (expression, holder, target, and polarity) with unspecified attention relations, it would contain 16 relations[21]. These models gave unsurprisingly volatile performance for our main metric. This behavior was expected, since these models included many factors that get updated for every batch over every epoch. The more dynamic elements a model contains, the more of a chance a really good, or really bad, change will be made, thus the more variation one should expect to see in the metrics. Best scores for these full attention relations seemed to be constantly better than our simplest `BertHead` architecture, and in the same general ball-park as our half-tuned IMN implementation, around 0.52 on our main metric.

We also tried to configure attention relations similar to those used in our IMN architecture. Surprisingly, only using polarity and expression attention relations gave relatively poor results, given model complexity. Here, scores similar to what would be expected from `BertHead` were obtained, about 0.44. This tells us that some of the smaller, internal data transformations of our `FgFlex` model were suboptimal compared to those in IMN. We were forced to accept this lower performance as a necessary price to pay in order to retain the flexibility of subtask deactivation novel to this architecture.

Relations mimicking those from RACL were tested as well. These included relations between target and expressions, expressions and target, and polarity with expression. Here, we saw similar results to those from the full attention relation defaults, though slightly more stable, meaning the performance did not oscillate much per epoch. These more stable results made this configuration the best of the three mentioned so far.

[19]https://github.com/pmhalvor/fgsa/tree/master/notebooks

[20]Similar notebooks were created for hyperparameter searching the other architectures as well, and can also be found in the repository.

[21]It could be important to point out again that the relation ["target", "expression"] differs from ["expression", "target"], since the first task is used as keys and values, while the second is used for queries.

Other relation-configurations, such as self-attention for all subtasks, singular attention relations between only 2 subtasks, and various alterations those mimicking the RACL setup were also studied. However, we did not extensively check every possible relation combination. In theory, there are an infinite number of relations that possibly could be checked, since multiple identical relations were also possible to configure. We decided against an extensive check for our experiment, and rather let future developers look further into these relations.

Out of all of these configurations, the best resulting attention relations we found for our `FgFlex` model, were an variation on the RACL relations. Including all of the same relations as RACL, this best configuration also included attention blocks using holder information. With targets and expressions as queries, these new relations used holders as keys and values. The last new relation used holder as query and polarity as keys and values.

4.5.3.2 Other studies worth mentioning

Multiple convlutional kernels We assumed that using multiple kernel sizes for our convolutional components would allow for our models to find patterns spanning over different lengths on our input sequences. This was partially inspired by the shared layer in the IMN architecture. However, our initial findings from hyperparameter searching gave no evident proof that these split components actually benefitted final model performance.

Layers The hyperparameter search on layers, resulting in the highest scores found for our IMN model, also proved to be a vital study to run on this structure. Here we found that increased layers seemed to help performance for almost all tasks. Lesser layers showed decreases in performance when gold transmission wore off. Contrarily, models containing many subtask layers maintained the high performance often reached within the first few epochs.

This told us that the information from gold transmission was being back-propagated into these task-wise layers, thus preserving vital information about what elements attention components should focus on.

Learning rate Optimal learning rates similar to that found for both our RACL and IMN models were found for FgFlex. `1e-5` gave best results, with `1e-6` giving a safe second place. This allowed for us to keep the learning rate scheduler activated without having to worry much about stagnant learning.

Stack The final study worth mentioning here was the check on stack count. We saw for IMN that interactions were often detrimental for performance. RACL showed moderate changes for counts up to 2, but dropped significantly when more than 2 stacks were used. In our FgFlex model, we found more stacks usually helped performance. It was actually through our stack study that we achieved the highest score measure in hyperparameter searching, namely 0.64 on our main metric.

4.6 Bug summary

Throughout development, our codebase was constantly changing due to new bugs being introduced, while old ones were still being weened out. As a result, new bug finds were attributed to almost every architecture developed. This was partially due to assumptions and reasoning that did not hold when the next, more complex architecture gave similar poor results as the previous. Since we could not entirely exclude a lack of complexity from explaining the low results, it was necessary to find if re-implementations of trustworthy models on our base model gave similar low scores, i.e. our motivation for baseline reconstruction. The reason new bugs kept being introduced, as old ones were removed, was that every new architecture added more moving parts, giving more places a new bug could hide.

This section provides a brief recap over the bugs found during experimentation. Even though all of these have been mentioned previously at their respective points of discovery during the experiment, we deemed a summary necessary for reproducibility purposes, warning any future developers from falling into the same traps we did. Important things that are mentioned for each bug are: when the bug was introduced, the reasoning behind introduction, how the bug was found, what effect it had on performance.

4.6.1 Prediction indention

During initial development of our `BertHead` class, we accidentally introduced the prediction indention bug, due to faulty tab-index. Found during development on IMN in torch, this bug had an effect on all models. When removed, we increased model performances up to 10 percentage points.

4.6.2 Learning rates

Low learning rates were decided necessary early during development of `FgsaLSTM`, partially motived by performance from `BetHead`, to avoid too much variability of loss for more noisy annotations. It was during hyperparameter searching of our first RACL version that we discovered learning was just as unstable with the low rate of `1e-8`, as for a few magnitudes higher, at `1e-5`. The effects this suboptimal configuration had on our models was that all training was taking much longer than needed. The increase to `1e-5` allowed for well configured models to be trained and ready for comparison by 30 epochs.

4.6.3 Opinion transmission

When developing our version of IMN, we struggled to understand exactly how gold-transmission should implemented. Later, our faulty initial implementation was corrected, thanks to the opinion propagation, built by the RACL developers. The effect our original implementation were not detrimental for initial performance on our IMN, however, we were unsure if the set-up was allowing leakage of true labels through for all epochs.

4.6.4 Activation functions

Our simpler models had no activation functions at the end of their respective linear components. These were first introduced with the IMN implementation. However, due to the fact the original RACL code did not use them, we made the functionality configurable via hyperparameters for our models. Through studying all three of our most complex models, we concluded that activation functions restricted model performance by up to 5 percentage points. Because this could be a result of our implementation, we make note of it here as a bug, and mention the feature as an area for future research.

4.6.5 Normalization of attention inputs

The original RACL implementation used L2-normalization for their attention inputs. We did not measure any benefit for this feature, although also included similar normalizations for our RACL and `FgFlex` models. We never were able to fully check this functionality, and therefore also leave if for potential future work.

4.6.6 Ignore sub-words in metrics

A large bug introduced in the first model built was the inclusion of tokens referring to sub-words in our predictions, yet telling the model to exclude these. This was not discovered until small scale experiments on our final model showed odd prediction scopes. The effect was measure to be up to 20 percentage point for each model trained. We admit that our fix was not the most scientific, since we changed the metric instead of changing the predictions. However, due to time restraints, we left our fix as-is, rather resorting to honest documentation around this flaw for any eventual reproducibility purposes.

4.6.7 Weight decay

Another suboptimal hyperparameter configuration found via tuning a faulty model, then carried over to cleaner models, was weight decay on our optimizer. This bug was not found until a hyperparameter search on our last model revealed large increases in performance when weight decay was set to 0. We could have maybe avoided this bug if we had run extensive hyperparameter searches on all of our models. However, to limit redundant resource consumption, such excessive tuning was deemed unnecessary for our experiment. In some of the more complex models, this results in up to 10 percentage point differences of performance results.

4.7 Chapter 4: Summary

In this chapter we outlined how our experiments unfolded. Experimentation started with outlining the preprocessing steps, along with checking the preprocessed Norwegian data on our baseline architectures. We then built simple models, followed by reconstructions of our baselines, and finally our flexible, novel architecture. Important hyperparameter searches along the way were presented and briefly discussed. The chapter finally rounds off by summarizing the main bugs that slowed down development throughout our experiments, together with how performance increased when the bugs were found and removed.

The next chapter will compare the best architectures found here, using the hypothesis test described in 3.3. While results for all architectures will be presented, a select few were chosen for a deeper evaluation. This allows us to cover models simpler and more complex than our baselines in a detailed analysis, while sparing the reader from redundant arguments.

CHAPTER 5

Experimental Results

In the previous chapter, we provided an outline of how our project unfolded. Inspired by two state-of-the-art sequence-labelers for fine-grained sentiment analysis, we built a flexible architecture for experimentation on the components necessary for high performing models. Some simple hyperparameter searching was done, on both the baseline models and our novel architectures. From the searches discussed in Chapter 4, we selected three optimized models for a more thorough comparison, namely the `BertHead` model, our implementation of the IMN architecture, and our `FgFlex` model. These three model types were selected to provide comparison between one simple architecture, one baseline, along with our novel architecture.

Chapter 5 will present the final results and evaluations for the three selected models. We start with some nuanced debates comparing individual components in Section 5.1. Here, we try to deduce the most useful components for building a fine-grained sequence-labeler. We already have provided some discussion around these components in previous sections, so Section 5.1 rather attempts to summarize these findings in a more concise, easier-to-digest, format.

After that, a quick presentation of the resource demands our top three models required is explored in Section 5.2. This gives the reader some insights on the computational impact our three best models have, in case future work is limited to a strict resource budget. For fair comparisons, we measure run time and memory consumption for these models on a fixed epoch size.

The final comparisons, including the final best results and discussions around what these findings tell us, are eventually presented in Section 5.3. In this section we compare the evaluation performance of these three models using our hypothesis test from 3.3. The hypothesis test helps us determine if any eventual increases we measured are statistically relevant, thus providing evidence that we succeeded in finding a better set-up for a fine-grained sentiment analysis sequence-labeler. This was necessary, due to varying performance over identical model configurations as a result of randomized parameter initializations, along with randomized shuffling of the data in our training set.

5.1 Component discussion

In **RQ2**, we were interested in finding which components provided increases in performance when used in a multitask-learning sequence-labeler for fine-grained sentiment analysis. In Chapter 3, we presented the main components used in the architectures used in our experimentation. Throughout Chapter 4, we briefly presented and discussed some initial results found on our implementation of these components. In this section, we aim to summarize these findings, giving the reader an overview of the importance of these components.

5.1.1 Linear layer versus LSTMs after BERT

Results from our two simplest models, `BertHead` and `FgsaLSTM`, provide a direct analysis on the benefits LSTMs have on contextual-embedded outputs. We saw little to no benefits from the use of LSTMs between our embedder and output layers, as seen via top performance for best configurations for each of our models. As we increased the number of layers in the LSTM components, we saw clear signs of decreasing performance. A plateau in performance is observed after around 10 epochs, hinting that no further learning would have occurred had we allowed more training iterations.

The observations that simple linear set-ups scored as good as our best LSTM set-up was quite surprising. Our reasoning for this is that the NorBERT2 embeddings are the main performance boosters of our models. Changing only the last few components in our pipeline has little to no relevance compared to the extensive BERT embeddings upstream. Therefore, we argue that when building simpler models, linear feed-forward outputs after a BERT embedder provide enough complexity.

5.1.2 Layers of convolutional nets

Through our hyperparameter tuning, we checked many configurations of convolutional layer counts per subtask. While individual results varied from architecture to architecture, we attempt to summarize these findings here to provide a starting point for future research.

Model	**Shared**	**Exp.**	**Holder**	**Polarity**	**Target**	**Stack**
`IMN`	3	2	-	1	2	3
`RACL`	-	2	-	1	3	1
`FgFlex`	3	2	2	1	2	2

Table 5.1: Final layer counts for interacting models

We often found that more than a single shared layer proved beneficial on our multitask-learners. Recall, the shared layer in our set-ups was the convolutional component after the embedder, but before the task-wise components. Performance shows signs of decreasing around 5 shared layers, although this was not robustly checked in our work. Variability of performance when studying shared layers spanned over a range of 5 percentage points. From this, we deduce that random initializations has more of an effect on performance than this particular hyperparameter, even though a slight increase in performance was found with more layers.

When looking at task-specific layers, we also observed slight increases in performance when more layers were used on our three BIO-labelling tasks (expression, holder, and target). Polarity showed a definite trend of decreasing performance as layer counts increased. Due to the weak indication of performance increases for BIO-subtasks with multiple layers, we argue that single layer configurations for these should provide enough space for subtask representation. However, in our final configuration of `FgFlex`, we used layer counts greater than 1 for these tasks due to the absolute best scores found in our simple hyperparameter searching.

5.1.3 Attention relations

We also studied different configurations of attention relations on subtask information. Specifically we compared the relations used in IMN, with those used in RACL, along with a handful of novel relations, including: self-attention, attention with holder information, and attention relations between all subtasks.

Our findings show that the relations used in IMN and RACL were best compared to other configurations with the same levels of complexity (1 relation in IMN and 4 relations in RACL). However, when working with a data set containing holder annotations, like NoReC$_{fine}$, it was beneficial to also include this information in these attention relations.

The optimal attention relation configuration we found was as follows:

```
[
    ["target", "expression"],
    ["target", "holder"],

    ["expression", "target"],
    ["expression", "holder"],

    ["holder", "target"],
    ["holder", "expression"],

    ["polarity", "target"],
    ["polarity", "holder"],
    ["polarity", "expression"]
]
```

5.1.4 Simplicity versus complexity

Our final results showed that the simplest `BertHead` model gave remarkably good results compared to our heavily complex models. As will be shown later in Section 5.3, our smoothed, final performance scores for this simple model did not fall outside of our 95% confidence interval. This means, we did not find enough statistical evidence to show that our baseline model performs better than `BertHead`, even though the average from those final 5 runs was slightly lower than our estimated population average for our optimal IMN set-up. [1]

Similarly, when increasing complexity from our IMN model to our `FgFlex` model, with more attention relations and higher stack counts, we did not find statistically significant increases in performance. This comparison will also be documented more thoroughly in 5.3. However, from this, we make the general claim that our simpler models are just as good as more complex ones.

[1]Had we defined a more lenient threshold (say a confidence interval drawn about a single standard deviation), then we could have claimed the difference in means was large enough to be considered evidence of a difference in expected performance. We did not use such a lenient performance threshold to reduce necessary computations.

5.2 Model efficiency

Furthering the argument from simpler models over more complex, we take a brief look at the practical efficiency of the models we tested. Here, we will compare the time a single epoch took to train for each of these models, along with their maximum memory usage during training.

Model	Epoch time (seconds)	Max memory (GB)
`BertHead`	54	10.694
`FgsaLSTM`	79	10.915
`IMN`	73	13.383
`RACL`	63	13.530
`FgFlex`	146	18.968

Table 5.2: Approximated resource requirements of our models

5.2.1 Run time

We look at the time a single epoch takes to process on GPU to get an estimate of a model's training speed. The second column in Table 5.2 shows these measurements taken on the time between the first and second epoch for each of the best models used for comparison.

We see that all the models take around a minute to train, except our optimal `FgFlex` model. This is likely due to the 2 stacks and many attention relations used in our optimal configuration of this model. Our IMN set-up was trained on 3 iterations as well, but only contained a single attention relation per interaction, thus limiting iteration time.

The slowest architecture was our `FgsaLSTM` model, clocking in at 6 seconds slower than the next closet model and a whole 25 seconds slower than our simple `BertHead`. Recall the discussion from 5.1.1, performance between `BertHead` and `FgsaLSTM` were very similar. The much slower iteration time is yet another argument against using LSTMs for non-interacting models.

It was interesting to see that our models with attention components were also faster than our LSTM model. One might be tempted to argue that our finding provide evidence that attention blocks are more efficient than LSTMs. However, we should point out that our `FgsaLSTM` model trained all 4 tasks (expression, holder, polarity, and target), whereas the IMN implementation was limited to only expression, polarity and target.

Even more surprising, our best RACL model was over 10 seconds faster per iteration than our best IMN model. Given the RACL set-up has 4 times as many attention components than IMN, we would have initially expected opposite results, with RACL as slower. Again, discrepancies between configurations provide an explanation. As shown in Table 5.1, our best IMN set-up used 3 interactions, versus only a single interaction in RACL.

5.2.2 Memory usage

We measure our models' memory requirements through the maximum GPU memory used during training. This metric is provided in the outputs of our Slurm jobs when executed on the Fox server. Checking the maximum memory used provides an insight on how many parameters are updated during a backward pass. Adding more model components, more stacks (not IMN iterations), or larger hidden sizes all have an effect on this metric. As can be seen in the third column of Table 5.1, as our model complexity increases, so too does this maximum memory used.

An interesting observation is the sizes between our two simple models, `BertHead` and `FgsaLSTM`. We would have expected a larger relative difference in maximum memory needed to update the LSTM parameters than for the linear based model. In a way, this tells us that our NorBERT2 head is massive compared to these final output structures. Which, in turn, could mean that we did not provide `FgsaLSTM` enough space to actually be useful. However, through our smaller scale hyperparameter searching, we could not find enough evidence to pursue very large configurations of the `FgsaLSTM` architecture.

Table 5.1 also shows very similar sizes of our IMN and RACL set-ups. Considering our model configurations, this strong resemblance in maximum memory usage seems reasonable. The IMN set-up used 3 iterations over it's task-wise components, meaning a back-propagation step would have needed to update these parameters again for every extra iteration. So, even though IMN only had a single attention component, the multiple updates due to these iterations would have required as much space as if completely new components were being updated.

It is interesting to point out that neither IMN or RACL is very much larger than our two simple models. This highlights how massive the NorBERT2 embedding layers are. As will be seen later in 5.3, the fact that `BertHead` is well over half the sizes of the other architectures is likely the reason this "simple" model performs so strongly.

Our best `FgFlex` configuration had both multiple stacks and more attention relations than our other interacting models. Thus, the larger maximum memory used during training was expected.

Unfortunately, we observed no direct corelation between memory consumption and final performance. We can say the same about iteration time, as our best performing model, IMN, achieved median results for both these metrics. However, increased complexity always increased both memory consumption and epoch time, as expected.

5.3 Final best results

The results we obtained through our experiments were underwhelming in reference to **RQ1**. On a broad level, we found that simpler sequence-labelers for fine-grained sentiment analysis were often as good, if not better than more complex ones. In this section, we explain this conclusion with help from averaged outputs from 5 runs on the same configurations to smooth variations in model performance due to random initializations.

Results for all of our models can be found in Table 5.3. The second column of this table shows the performance of each of our models on the development data, and the third column shows the held-out validation data. As explained in Section 4.1.1, these two sets play different roles in evaluation. The development set is used during training, giving us the metrics-plots seen in Chapter 4.

Model	Development data	Hold-out evaluation data
`BertHead`	0.4174	0.4152
`FgsaLSTM`	0.3877	0.3861
`IMN`	**0.4439**	**0.4438**
`RACL`	0.4201	0.4223
`FgFlex`	0.4036	0.4129

Table 5.3: Final performance results for all our models after 30 epochs, averaged over 5 runs.

The held-out evaluation was excluded until this final comparison between our optimized models.

If our models had been excessively tuned, we would have expected to see slightly lower performance for the evaluation data, due to increased risk of over-fitting. However, due to our linear, simplistic hyperparameter searching, we see no consistent discrepancies between results for each of these sets. In fact, some of the models performed better on the held out data set. This tells us that further tuning could be necessary on many of our optimized models.

5.3.1 Hypothesis test parameters

Recall our hypothesis test defined in Section 3.3. We built a normal distribution around estimated population parameters for the performance output from 100 randomly initialized models on our best baseline configuration. By setting a 95% confidence interval about our mean, we created α and β significance levels on this distribution. With this interval, we can determine if any differences in performance obtained by other architectures showed statistically significant improvements.

A visualization of our estimated distribution is shown in Figure 5.1. In this plot, we also show the averages from 5 runs for each of the architectures we tested. These smoothed performance scores give us an estimate as to how well each of our model setups fares compared to our distribution.

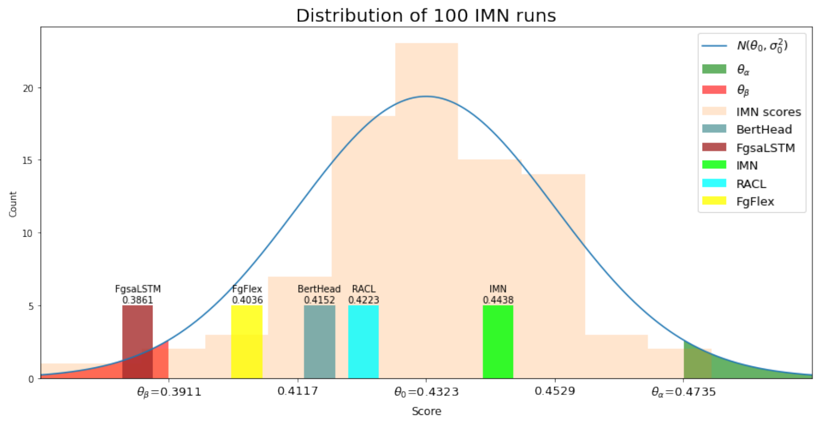


Figure 5.1: Population distribution with our top results.

Between our two baselines, our IMN implementation gave the best results on a 5-run average. We therefore approximated the expected value of the parameters $(\hat{\theta}_0, \hat{\sigma}_0)$ defining the distribution of IMN results with the help of 100 runs. We found our approximated population average $\theta_0 \approx \hat{\theta}_0 = 0.4323$ with an approximated standard deviation of $\sigma_0 \approx \hat{\sigma}_0 = 0.0206$. This set the limits of our 95% confidence interval to $\theta_\alpha = 0.4735$ for the upper boundary and $\theta_\beta = 0.3911$ for the lower boundary.

5.3.2 Simple models

As can be seen in Figure 5.1, our simple `BertHead` was the third best model out of the 5 we trained. However, it's averaged score fell safely within the confidence interval of the approximated population distribution of IMN performance models. This means that we could have expected to see the same score for another run of our best IMN configuration.

Our assumption as to why this simplest model scored so well is the underlying importance of NorBERT2 embeddings. Similar reasoning was argued earlier when looking into the resource demands of our models. We argue the most important model component for any of our fine-grained structures is the embedder. This argument is based partially on the fact that the internal NorBERT2 components are so large, require so much memory, bu also, due to the massive performance drops when we did not include these in our optimizers.

Our other simple model actually fell outside the scope of our confidence interval. While it does not say anything for certain, this very low score provides evidence that the `FgsaLSTM` model we built is expected to perform worse than our IMN implementation. A reason this model scores more poor than others could be that we did not check complex enough configurations of it. Future work could look into different variations of stacked LSTMs per task, or addition of some convolutional components. However, our findings show that single LSTM components were not very productive.

5.3.3 Interactive models

Of our interaction-focused architectures, our novel `FgFlex` scored the worst. This could have been completely random, as the smooth 5-run average for this model still falls within our confidence interval. Again, this means that another random selection of 5 IMN models could have also given a same score. So while we can make judgments on possible areas for improvement of our architecture, we have no statistical evidence that our `FgFlex` model is better or worse than our best baseline implementation.

The major difference between the final `FgFlex` architecture and the baselines was the increased complexity. Our flexible model was made to handle many more attention relations after subtask specific components, including a few on the new holder information. The poorer performance of our model could suggest that holder annotations do not provide enough new information to the model, compared to the increase in difficulty. However, such findings would not be expected, since evidence from other experiments argued that holder extraction is in fact beneficial, especially for polarity classification (Barnes, Velldal and

Øvrelid 2020).[2] Our findings suggest that holder extraction was not as helpful as previously presented, since no major increases in performance are found with the introduction of this extra annotation.

5.3.4 And our winner is..

As can be interpreted from Figure 5.1, we argue that the best model configuration we tested was our implementation of IMN. No other model provided evidence to accept our upper alternative hypothesis $H_{1,U}$, thus rejecting H_0.

If we chose to emphasize model efficacy according to train time and resource requirements, we might suggest that `BertHead` should also be considered a strong competitor. This argument is based on that fact that the `BertHead` model did not reject the null hypothesis by falling below the threshold θ_β. We can not make the argument that `BertHead` out-performs IMN, but it can be considered a strong competitor, with a slightly small computational footprint.

Our novel `FgFlex` was good for experimenting on different model configurations through abstract configuration files. Unfortunately, the limited tuning we were able to run did not reveal any novel interactions previously ignored by other architectures. A more extensive hyperparameter search on this model could potentially reveal more insights on what data annotations and model components are always useful, and which only help for some situations.

5.4 Chapter 5: Summary

In this chapter we saw the end results obtained from our experimentation. We compared some of the major components, arguing for simplicity over complexity in most cases. Resource consumption of our models was presented and discussed, also hinting toward sufficiency of simple models. Finally, we pointed out our best models, along with reasoning as to why they should be considered "best" based off the evidence we found.

With the results presented and discussed, we conclude our project. Chapter 6 explains where we could have potentially improved our experiment, and present some directions for future research on the fine-grained sequence-labelers presented in this experiment. The key insights are also presented as a summary of our findings.

[2] It could be noted here that the argument by Barnes, Velldal and Øvrelid (2020) was that expression and holder together helped in polarity classification.

CHAPTER 6

Conclusion

In this thesis, we looked at different sequence-labelling architectures focused on solving the task of fine-grained sentiment analysis for Norwegian. We used baselines previously developed for English, and extracted the important pieces to use in our novel flexible architecture. With a focus on reproducibility, we built our `FgFlex` model to evaluate varying configurations of interactions between the subtasks of our multitask-learner.

We open this concluding chapter with brief summary of our experiments in Section 6.1. Here, we outline the motivation for our project, together with previous work that inspired the direction of project development. We bring together the various building blocks that setup our experiment, and outline the methodology we followed to unlock the findings we discovered.

The key insights of our experiment are then summarized in Section 6.2. Here, we mention the most important components needed in fine-grained sequence-labelling, along with general lessons our experiments taught us. In short, we found that simple models out-performed of more complex ones, with a few exceptions for the previous state-of-the-art models. Although, even between the baselines, we saw that simplicity reigned supreme.

We do not claim to have executed perfect experimentation in this project. Section 6.3 scrutinizes the flaws we realized were a part of our project, and proposes how some of these could have been avoided. This list may not be entirely exhaustive, neglecting some of the smaller defects. However, it serves as a warning statement to future developers about potential bugs that can be avoided in further research on this topic. In this sense, our experiments constitute a negative result. Nevertheless, there is a growing awareness in the field of machine learning regarding the importance of publishing negative experiments to save others from repeating mistakes, as witnessed e.g. by the recent Workshop on Negative Results.[1]

Finally, in Section 6.4, we share some ideas on the directions future work can take. These include both next natural steps on our architectures, but also entirely new directions fine-grain sentiment analysis projects could pursue.

6.1 Project recap

Models built for fine-grained sentiment analysis aim to solve the complex task of detailed sentiment polarity classification at target level, together with target,

[1] https://insights-workshop.github.io/

expression, and sometimes holder extractions. Often, multitask-learning is employed to find each of these nuances of the opinions in an input.

Our project focused on solving fine-grained sentiment analysis as a sequence-labelling task. This decision was inspired by the IMN (He et al. 2019) and RACL (Chen and Qian 2020) architectures. Both these system split the main, complex task of fine-grain sentiment analysis into subtasks for each of the annotations included in their data sets. A multitask-learning architecture with different information sharing techniques was built for each of these systems. These models used convolutional networks, attention mechanisms, and contextual BERT embeddings to represent and extract labels for each task, for all the tokens in an input.

We chose to iterate on these architectures, building a flexible synthesized version of the two baselines. Our novel `FgFlex` model could then easily test different relational connections between the subtasks of a model. This drove research on exactly which annotations had something to learn from others, allowing future developers to avoid unnecessary attention relations that slow down training, but provide little benefits on performance.

After a simplified hyperparameter search, we compared optimized versions of each model we developed for this thesis with the help of an approximated expected performance distribution of our best baseline. While we failed to find a proper configuration of our more complex models, we found that very simple models score about as well as our best baseline. This told us that while these baselines could provide slight improvements on our strict metrics, much of the heavy lifting for high performing models was achieved by our NorBERT2 head.

6.2 Key components

As such a complex task, fine-grained sentiment analysis requires good representations of the token-level elements of an input sequence, as well as smart internal interactions between model subtasks to produce proper model outputs. In this section, we outline the key components needed for these proper representations. While our experiment explored more than just the two mentioned here, we deemed attention and BERT heads as the most important components for fine-grained sequence-labelling.

6.2.1 Attention

Our baselines models already showed that attention mechanisms can be very useful to allow a model's subtasks to pass information to each other. Attention components make use of cross-multiplication of the subtask outputs. This helps uncover contextual information about the indexes of these output tensors, thus helping the model learn relations between information found from each task-wise component.

Our findings suggest that these attention relations are helpful when applied to some subtasks, but not necessarily between all of them. In our final `FgFlex` model, we saw that additional attention relations did not provide the boosts in performance we would have initially expected when reading the results presented in the IMN and RACL papers (He et al. 2019, Chen and Qian 2020). The `FgFlex` model can be used to test any possible subtask configuration setup through hyperparameters.

6.2.2 BERT head

For comparative purposes, we trained a simple `BertHead` model, that only applied a single linear output component to upstream BERT embeddings. This model scored close to the more complex, interacting models, as shown in our final results. Such a high performance of a model with very simple downstream tasks reinforces what many modern NLP projects also show, BERT embeddings are very helpful for model performance. Unfortunately, we did not check any simpler embedding set-ups to strengthen this claim. However, our hyperparameter studies where fine-tuning our NorBERT2 embedding was deactivated showed major drops in performance for all architectures, especially the simplest one, `BertHead`.

6.3 Room for improvement

Decisions made throughout project development shaped the results obtained in this experiment. These included everything from solutions to found bugs up to component selection for our novel architectures. When choosing a proposed solution, we considered possible new problems that could arise as a result of the solution. For example, implementation of our `BertHead` class affected the overall end flexibility of our more complex models. We therefore needed to consider a model's expected scalability when writing the new classes.

Unfortunately, in some cases, potential side-effects of a solution were not entirely forseeable when the solution was decided upon. This consequently lead to some decisions being made mainly for project progression, rather than remaining stuck at a single cross-road, with two dark paths ahead.

This section presents and discusses some of these vital decisions, both the good and bad. The decisions are split into three main parts of our project according to where they had larges effects: engineering, experiment, and evaluation. We try to point out areas of improvement, where we would choose differently if we were to restart the experimentation process. These can be interpreted as warnings for future developers embarking on a similar experiments in the future.

6.3.1 Engineering improvements

This subsection presents the improvements that could have been made on the portion of our project dealing with model construction and implementation. We touch on code specific details like class inheritance, testing, and merge-readiness of development branches, but also higher level questions like how necessary it was to rebuild baseline models.

Complex models inheiriting from base The fact that we chose to define a base class that our more complex models would inherit was only one of the possible approaches we could have taken to model structure. We could have alternatively rewritten every model as a standalone class, with many of the core elements just copied over from previous models.

In a previous project we took part in, the latter approach was used due to large expected differences between architectures (Pereira, Halvorsen and Guren 2021). With every model reliant only on the scope of its class, the models were easy to move between files, in case large directory organization changes were necessary throughout development. However, they caused headaches when bugs within a method used in all models were found. Updates to all of the identical methods in each of the respective models were needed to fully get rid of the bug. When working with many models, it became tedious to comb through the clutter and ensure updates were made everywhere.

Our choice of models inheriting base methods from a simpler parent class ensured that any updates to these methods fixed all models with the update. Consequently, when building more complex component interactions, we found ourselves altering pieces of this base class to handle the different variations that would come downstream. We consider these add-hoc additions to the base methods as suboptimal implementations for our models. To improve scalability of this inheritance-based structure of our code, we would recommend that a total reconstruction of our interacting `init_components()` method and `init_optimizers()` should be done to handle our various architectures more generally.

Test functions When the project started, we hoped to write test methods for all the steps along the way during development. These would provide sanity checks, ensuring that our data and models looked and worked the way they were intended to during implementation. Through diverse experience in both industry and academia, we knew that robust testing should be a cornerstone of any code-based project.

However, as our timeline progressed, we quickly found that development of proper test coverage often took as long as the experimental engineering itself. This stagnated development during the first few weeks, and was eventually down prioritized to ensure we would maintain a reasonable development schedule. With more developers on hand, these test functions could have helped achieved a more streamlined experimentation process.

Merging development branches The use of a source-control tool helped us track all updates and changes made to our systems. During initial development, we completely built and tested models on their respective `dev/` branches, and only merged these into our master project branch when reasonable results were obtained. We tried to also create branch snapshots as stable versions of the code when merged into master, for example `stable/BertHead`.

However, as the number of models continued to grow, so too grew the number of bugs found in previous merges. This resulted in cluttered `stable/` branches, and we eventually dropped the idea all together. The clutter could have been avoided if we had clearly defined guidelines as to when a merge was necessary, along with version-labelling each new `stable/` branch created. This could have been planned for before starting project development, and would have required slightly more time for documentation during development. With proper documentation of each version, switching between different stable versions of our codebase would have been more organized, and would have been beneficial in any eventual re-checking of previous implementations.

Reconstructing baselines The last point to discuss under engineering improvements was the necessity of re-implementing the baselines in our framework. While this topic has been discussed a few places through-out the text, we figured it was also a vital point to mention under improvement, since it consumed much of our development time.

Originally argued for as a debugging solution, one could argue that debugging could have been done without this re-implementation. The final bugs we found affected how our predictions were being measured, and thus were only a part of the `BertHead` class, our first model implemented. Had we done the small scale step-by-step analysis that discovered these bugs earlier on, we could have avoided this baseline reconstruction detour altogether.

On the contrary, building these baselines allowed us to eliminate any potential ambiguities between component initializations. Rewriting everything in PyTorch gave a standardized foundation for more pure comparisons of the architectural components of these systems. It also provided a nice stepping stone for development of our novel flexible architecture, since much of the `FgFlex` model is parallel to these baselines.

In conclusion, we argue that the advantages of the decision to rewriting our baselines out-weighted the disadvantages. Even though it could have saved us time during development, this reconstruction gave our experiment an edge over one where mere comparisons of the original baseline implementations were used thanks to the standardized foundation our set-up provided.

6.3.2 Experimenting

The actual experiments conducted as a part of this thesis could have also been improved upon. In this section, we discuss the reasoning behind our ad hoc evaluation debugging, some decisions around our hyperparameter search technique, along with how we defined the threshold to end tuning of our baseline models on new data.

Linear parameter search versus others While previously discussed in Section 4.3.2, we briefly touch on our choice of hyperparameter search technique again as a potential area for project improvement. As mentioned, we ruled out grid-searching with cross-validation techniques due to the exponential increase in resources required to run a single job. We argued that this technique is fine to use when developers are sure the models they are running are correctly implemented. However, since our models were works-in-progress throughout most of the experiment, we decided to take a less demanding approach.

Other techniques, such as random or guided parameter searches could have also been used. We have no argument against either of these techniques, as they both cut resource consumption, while covering more of the potential parameter-configuration space. We chose here the simplest method of searching for initial development, and stuck to it throughout our experiment.

If we were to start project development over again, we might argue to rather implement a random or guided hyperparameter search from the beginning. This would help unlock even more of the potential of the new models being tested.

Hyperparameter value selection Sticking to the topic of hyperparameter searching we can comment on how selection of the best values could have also had an effect on our potential best configurations.

Through our experiments, we found that any given configuration could achieve different results, within a 5 to 10 percentage point range. When evaluating different values for certain hyperparameters, it was often hard to conclude which configuration, if any, actually provided benefits. The exceptions here were those parameter values that lead to large reductions of performance.

This is a direct consequence of our chosen hyperparameter-tuning method, discussed in the paragraphs above. Because a single model's performance was more noisy than any real increases most hyperparameter values provided, we were forced to select configurations from single best runs. Such a selection process cannot ensure optimal tuning for hyperparameters with weaker signals.

To mitigate such a problem, we propose using a 5-fold cross-validation technique with our simple selection technique (or even better a random or guided search as mentioned above). Cross-validation would provide a smoother signal for new search, since it would train 5 models on slightly smaller data sets with the desired configurations.

Early hyperparameter seaching unfinished models At an early stage in our experiment, we ran a hyperparameter search on a buggy model. The resulting "optimal" parameters eventually proved to limit performance throughout the subsequent steps of development. We advise future developers to be wary of trusting fine-tuned parameter values found early during development. While some tunning may be necessary, for example to find proper learning rate magnitudes, new checks should be when large bugs are discovered, to ensure that the tuned values are still optimal.

Sequence-labelling task Early on during our experimentation, we committed to solving fine-grained sentiment analysis as a sequence-labelling task. This was inspired by our chosen baselines. As our project developed, we also stumbled into other methods for solving fine-grained sentiment analysis, for example through dependency graph parsing (Jeremy Barnes, Kurtz et al. 2021). We would like to point out that these other solutions seem to be promising methods for solving FGSA.

6.3.3 Evaluation

Our evaluation schema was not entirely flawless either. We will briefly touch on what we found most difficult with this step in our project here.

What is an improvement? As mentioned in the discussion around hyperparameter searching, we often found ourselves struggling to determine that a given hyperparameter configuration was better than others tested. There is no simple solution here either. Cross-validation would have increased our run times by the number of folds we ran it over. Hypothesis test based comparisons like the one used in Chapter 5 could have given us more confidence on certain improvements, but would have also required much more time and computational resources. We acknowledge that our choice of hyperparameter searching did not invoke confidence in our optimally tuned configurations, and should thus be improved upon in future work.

Final evaluation We built an approximated normal distribution for a population of expected model results using 100 runs on our best baseline. Using the Central Limit Theorem (Berger and Casella 2001), we decided that 100 runs should be sufficient to locate the expected value of our population mean and standard deviation.

However, 100 was just an arbitrary number chosen much higher than any runs we expected to execute on our final model comparisons. To fully satisfy the CLT, we would have needed to run infinite runs for our model. A better approach could have been to plot the average of the found results along the way, and find a proper cut-off point when the average seemingly converged beyond 4 decimal points. This would have ensured us that we found a good enough approximation for our distribution.

6.4 Future work

In the final section of our thesis, we present the directions future work on this topic can take. These inspirations were found along the way, but were not implemented in our experiment, due to lack of time and resources.

Hyperparameter search As mentioned multiple times already, alternative techniques for hyperparameter searching should be used for future work. Our simplified approach did not provide the confidence we needed to be sure were were tuning optimally. Future work should try implementing a MLE-guided parameter search (Welleck and Cho 2020) or a form of randomized parameter search to cover more of potentially correlated hyperparameter values.

Initialization constants on model parameters The original implementations of our baseline models included Xaiver initialized parameters in their architectural components. We dropped this step due to a lack of knowledge around proper implementation in our framework. This could have had an impact on how our activation functions were affecting model performance. We would recommend future work should look into initialization constants to see if any gains can be achieved through them.

Task-wise LSTM shapes Our results showed the our LSTM based model performed the worst of all models. This could seem surprising, since similar components have shown to be beneficial for fine-grained sentiment analysis in previous work (Barnes, Velldal and Øvrelid 2020). We might argue that more task flexibility could have been implemented here, allowing task-specific configurations of hidden sizes to be tested and tuned.

Scope finder Our implementation of IMN tried using a scope-finding auxiliary task to help mitigate large over-confidence of labels due to a reliance on large weight losses. While we eventually dropped the large loss weights thanks to debugging, we grew curious as to how much performance increases can come from scope estimations. It could be interesting to further pursue this auxiliary task, potentially adding it to the next iteration of our `FgFlex` model.

Precision and recall focused loss weights Another idea that was brainstormed but never implemented when our models relied on loss weights was precision and recall based weights. In Section 3.2.1, we explained that precision is a measurement of the proportion of predicted labels that are actually true labels, and recall is the measurement of the proportion of true labels that were left out of predictions. Both of these are key components in an F1 score. Since we were comparing our models over an F1 based metric, it would be possible to implement two types of loss, one focusing on precision and the other on recall. By setting label weights very high, we ensure that recall will score very high, but probably ruin the chances of precision to score well. By setting empty token weights higher than label weights, we ensure our model does not predict a label as present unless the model is very confident the label should be present. Alternating between these losses based on epoch count could help optimize the model to score well on the main F1 metric. We never implemented such a loss, but wanted to point it out in case future developers also realize there are potential gains to be made from increased loss weights.

6.5 Chapter 6: Summary

This concludes our project for fine-grained sentiment analysis for Norwegian. We have presented and discussed from previous works in this field, and taken inspiration from the best of these past projects. We outlined our experimental framework, touching in on the systems we used to develop, maintain, and run our architectures. A detailed walk-through of how our experiment unfolded followed, together with the bugs our models picked up along the way, and what effects these had on our results. A structured, statistical comparison of our best architectures was presented, as well as the computational needs the different models required. We attempted to view our project through an impartial perspective, to honestly detail the mistakes we made, and warn future developers against the potential flaws one might accidentally include in such a project. We hope this project can serve as a stepping stone for similar further research for fine-grained sentiment analysis.

Appendices

APPENDIX A

Results from our experiment

A.1 Experimentation

A.1.1 BertHead loss

Below we see a snippet of the loss outputs of a BertHead model, trained early during experimentation. This is only the loss provided from a single batch, but still shows how our model is learning. During the first epoch, we see decreases in loss everywhere between a tenth of the previous values (for holder), to half the initial values (for expression). The variation between subtasks tells us it's easier for our model to learn holders than any other task. We also see that target and polarity show quite similar decreases in loss, which can be expected, since they are labeled over the same tokens. Notice also the large initial dip after the first epoch, followed by a plateau for the rest of the epochs. This is similar to the behavior of RACL, and is likely due to large corrections in the BERT embedder in the model. Timestamps are included to show training times on this setup, which was initially run on CPU.

```
15:52:01,455 [    INFO] Fitting model... (dev.py:95)
15:52:37,961 [    INFO] Epoch:  0 Batch:  0 (model.py:399)
15:52:37,962 [    INFO] expression loss:1.0552302598953247 (model.py:401)
15:52:37,962 [    INFO] holder     loss:1.0435622930526733 (model.py:401)
15:52:37,962 [    INFO] polarity   loss:0.9089418053627014 (model.py:401)
15:52:37,962 [    INFO] target     loss:1.385927438735962 (model.py:401)
18:14:19,179 [    INFO] Epoch:  1 Batch:  0 (model.py:399)
18:14:19,179 [    INFO] expression loss:0.7729575037956238 (model.py:401)
18:14:19,179 [    INFO] holder     loss:0.1621379554271698 (model.py:401)
18:14:19,179 [    INFO] polarity   loss:0.3774401545524597 (model.py:401)
18:14:19,179 [    INFO] target     loss:0.3899977505207062 (model.py:401)
20:36:23,854 [    INFO] Epoch:  2 Batch:  0 (model.py:399)
20:36:23,855 [    INFO] expression loss:0.7536727786064148 (model.py:401)
20:36:23,855 [    INFO] holder     loss:0.08016534894704819 (model.py:401)
20:36:23,855 [    INFO] polarity   loss:0.33955639600753784 (model.py:401)
20:36:23,855 [    INFO] target     loss:0.34202834963798523 (model.py:401)
22:58:27,850 [    INFO] Epoch:  3 Batch:  0 (model.py:399)
22:58:27,850 [    INFO] expression loss:0.7385057210922241 (model.py:401)
22:58:27,850 [    INFO] holder     loss:0.06342285126447678 (model.py:401)
22:58:27,850 [    INFO] polarity   loss:0.3324304521083832 (model.py:401)
22:58:27,850 [    INFO] target     loss:0.3368951678276062 (model.py:401)
01:20:20,290 [    INFO] Epoch:  4 Batch:  0 (model.py:399)
01:20:20,290 [    INFO] expression loss:0.7114876508712769 (model.py:401)
01:20:20,290 [    INFO] holder     loss:0.05775010213255882 (model.py:401)
01:20:20,290 [    INFO] polarity   loss:0.3238269090652466 (model.py:401)
01:20:20,290 [    INFO] target     loss:0.33151718974113464 (model.py:401)
```

A.1.2 BertHead learning rate

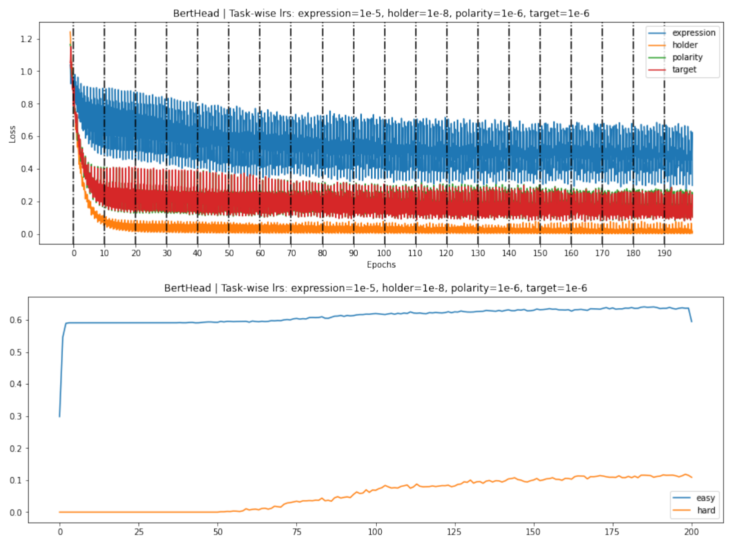


Figure A.1: Loss and metrics for BertHead with task-wise learning rates.

A.2 IMN layers study (debugged)

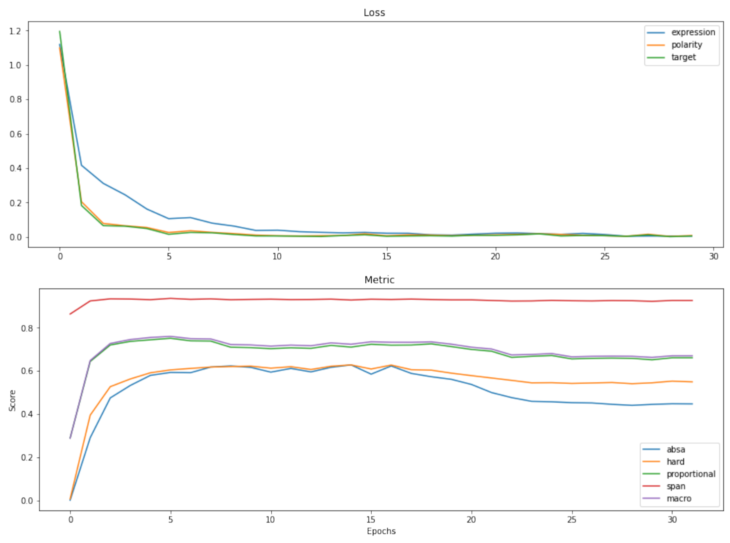


Figure A.2: 4-run-smoothed loss and metrics for IMN with best layer configuration.

A.3 RACL learning rate study (debugged)

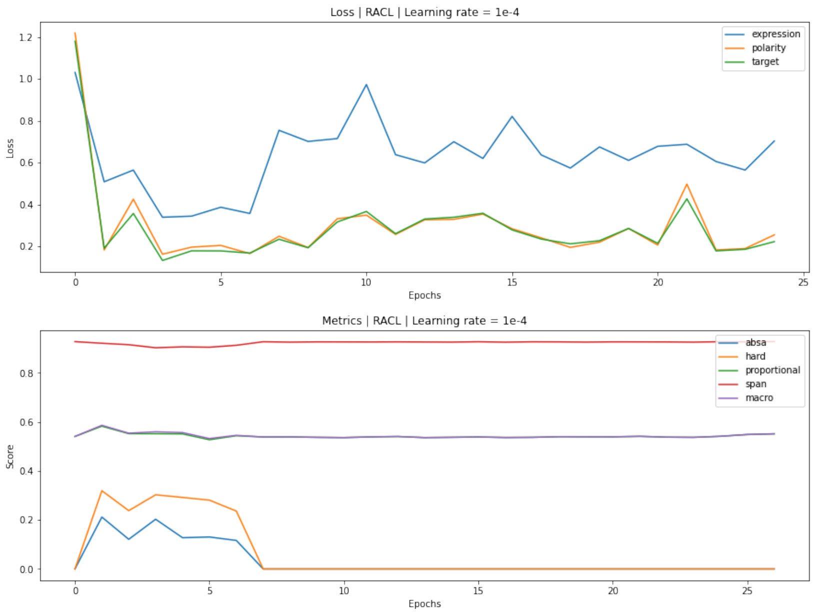


Figure A.3: Loss and metrics for RACL with learning rate=1e-4.

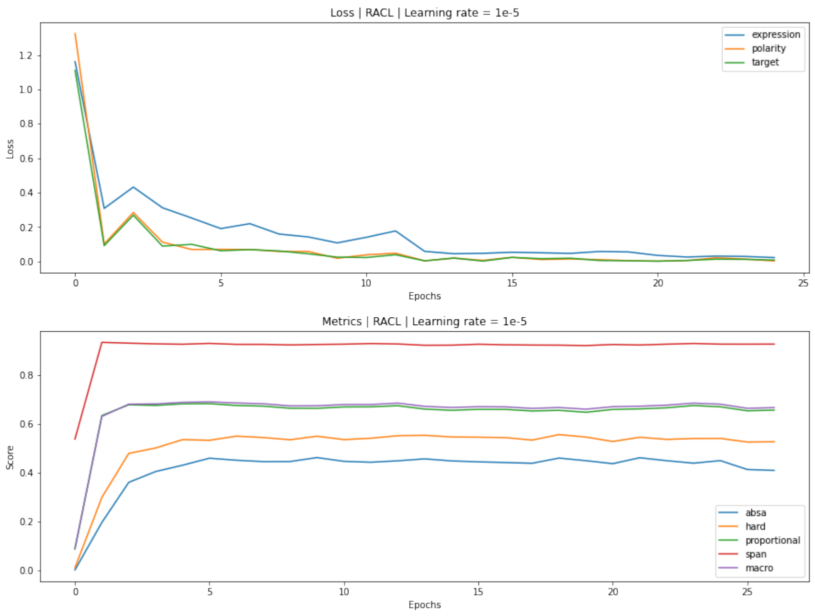


Figure A.4: Loss and metrics for RACL with learning rate=1e-5.

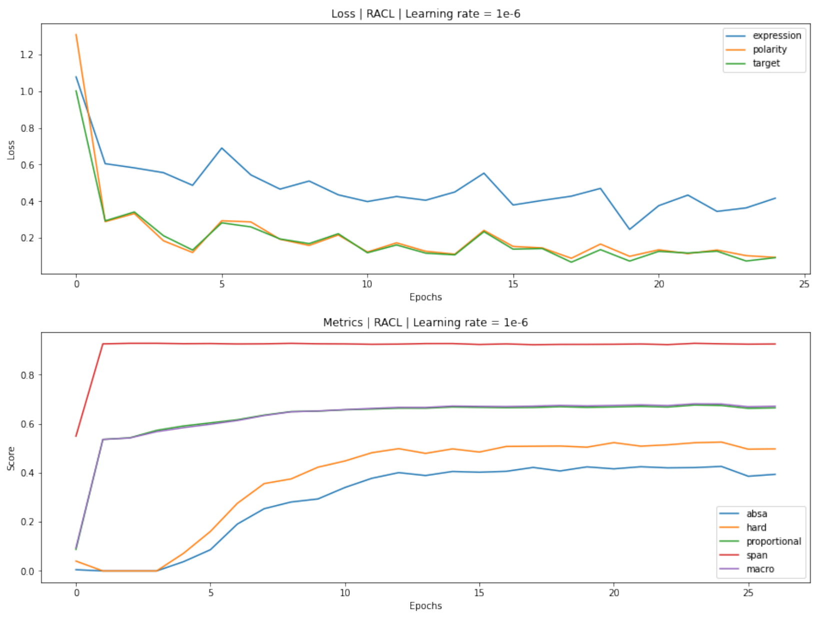


Figure A.5: Loss and metrics for RACL with learning rate=1e-6.

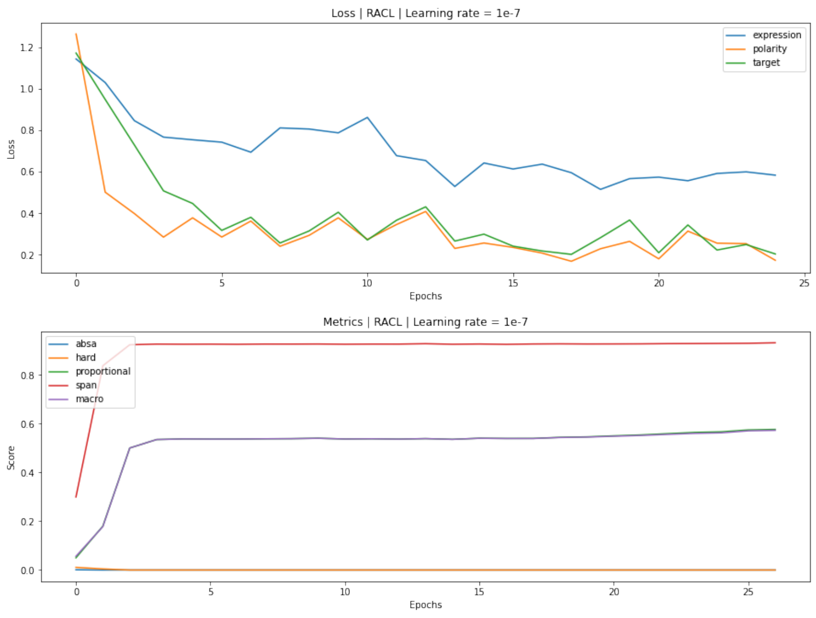


Figure A.6: Loss and metrics for RACL with learning rate=1e-7.

APPENDIX B

Bug details

B.1 Detailed description of the main prediction bug

Recall that our `BertHead` class was built to avoid rewriting much of the boilerplate code necessary in a neural network, i.e. the fit method, initialization methods, as well as evaluation and prediction methods.

Our base class included an `evaluate` method to measure model performance over a handful of different metrics. These evaluation techniques, discussed previously in 3.2, relied a method called `predict()` to make predictions on a test data set used for evaluation.

In our original implementation, our predict method took in a batch of validation data. It then fed this data through a forward-call to produce model outputs. These outputs were then converted from their tri-state logits to their estimated token-label[1]. To iterate through all the tokens of a batch's output, we needed to write a few nested for-loops in this prediction-method.

The bug of this method was a single faulty tab-index in one of these for-loops. The nested for-loop iterating over tokens in a single sentence of a batch was meant to be broken when the padding tokens of that sentence were reached. This was built to add all predicted tokens for a given input to their respective subtask lists, without including any of the padding from being batched. Instead, we called `break` on one of the outer for-loops, causing predictions from the same first sentence in every batch to be added to the respective task-wise lists n times, where n was the length of sequences for that batch. In other words, our prediction method was returning predictions from the first row of every batch n times, then skipping all the other sentences.

When we added very high loss weights, we saw small increases in performance, likely due to a few first rows of batches that actually had some labels in them. Since many input sentences in our data set did no have any labels, performance would have still be very low. This is because a sentence with no labels returns an F1 score of 0, and our final F1 scores were averages over all the batch-wise F1 scores. An underlying assumption with such an evaluation scheme is that at least one input sentence with an opinion would be included in every random selection of 32 input sentences. To be more sure that all batches had the ability to score over 0, we could have used a larger batch-size. This would then cost

[1]Tri-state logits for a given token contain a probability for each possible state a token could have. These were produced for all tokens in all sentences in a batch for each subtask in the outputs. Refer back to 3.5 for more details on model outputs.

more required memory per batch when training, since more inputs would need to be updated for during the optimization step.

Because this bug only affected our evaluation method, our model losses were in fact showing true learning occurring internally. We just were not measuring model predictions properly, thus obscuring our performance metrics. Some of our initial hyperparameter tuning was done using changes in loss. These were therefore still relevant studies, even though they produced low metric scores.

With this fix, we again reran tests on our simpler models, and found that performance increased dramatically. Because of the similar performance of our `BertHead` and `FgsaLSTM` after the prediction bug was found, we focused mainly on the simplest of the two, namely `BertHead` for future comparisons. This bug was found and commented on in pull-request #24 in our project repository[2].

[2] https://github.com/pmhalvor/fgsa/pull/24#discussion_r832663340

Bibliography

Abu Sheikha, Fadi and Inkpen, Diana (2010). 'Automatic classification of documents by formality'. In: *Proceedings of the 6th International Conference on Natural Language Processing and Knowledge Engineering(NLPKE-2010)*, pp. 1–5.

Agerri, Rodrigo et al. (2013). 'OpeNER: Open Polarity Enhanced Named Entity Recognition'. In: *Procesamiento del Lenguaje Natural* vol. 51, no. 0, pp. 215–218.

Bahdanau, Dzmitry, Cho, Kyunghyun and Bengio, Yoshua (2016). *Neural Machine Translation by Jointly Learning to Align and Translate*. arXiv: 1409.0473 [cs.CL].

Bahl, Lalit R., Jelinek, Frederick and Mercer, Robert L. (1983). 'A Maximum Likelihood Approach to Continuous Speech Recognition'. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* vol. PAMI-5, no. 2, pp. 179–190.

Barnes, J. et al. (May 2021). 'IN5550 Teaching Workshop on Neural Natural Language Processing'. In: *Proceedings of the Third IN5550 Workshop on Neural Language Processing (WNNLP 2021)*. Oslo, Norway: Language Technology Group.

Barnes, Jeremy, Badia, Toni and Lambert, Patrik (May 2018). 'MultiBooked: A Corpus of Basque and Catalan Hotel Reviews Annotated for Aspect-level Sentiment Classification'. In: *Proceedings of the Eleventh International Conference on Language Resources and Evaluation (LREC 2018)*. Miyazaki, Japan: European Language Resources Association (ELRA).

Barnes, Jeremy, Kurtz, Robin et al. (Aug. 2021). 'Structured Sentiment Analysis as Dependency Graph Parsing'. In: *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*. Online: Association for Computational Linguistics, pp. 3387–3402.

Barnes, Jeremy, Øvrelid, Lilja and Velldal, Erik (2021). 'If you've got it, flaunt it: Making the most of fine-grained sentiment annotations'. In: *ArXiv* vol. abs/2102.00299.

Barnes, Velldal and Øvrelid (Nov. 2020). 'Improving sentiment analysis with multi-task learning of negation'. In: *Natural Language Engineering* vol. 27, no. 2, pp. 249–269.

Bergem, Eivind (2018). *Document level Sentiment Analysis for Norwegian.*

Berger, Roger and Casella, George (June 2001). *Statistical Inference*. 2nd ed. Florence, AL: Duxbury Press.

Bojanowski, Piotr et al. (2017). *Enriching Word Vectors with Subword Information*. arXiv: 1607.04606 [cs.CL].

Chen, Zhuang and Qian, Tieyun (2020). 'Relation-aware collaborative learning for unified aspect-based sentiment analysis'. In: *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pp. 3685–3694.

Church, Kenneth Ward and Hanks, Patrick (June 1989). 'Word Association Norms, Mutual Information, and Lexicography'. In: *27th Annual Meeting of the Association for Computational Linguistics*. Vancouver, British Columbia, Canada: Association for Computational Linguistics, pp. 76–83.

Devlin, Jacob et al. (June 2019). 'BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding'. In: *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. Minneapolis, Minnesota: Association for Computational Linguistics, pp. 4171–4186.

Dua, Dheeru and Graff, Casey (2011). *Amazon Commerce reviews set Data Set*.

Dumoulin, Vincent and Visin, Francesco (2018). *A guide to convolution arithmetic for deep learning*. arXiv: 1603.07285 [stat.ML].

Ettinger, Allyson (2020). 'What BERT Is Not: Lessons from a New Suite of Psycholinguistic Diagnostics for Language Models'. In: *Transactions of the Association for Computational Linguistics* vol. 8, pp. 34–48.

Fokkens, Antske et al. (Aug. 2013). 'Offspring from Reproduction Problems: What Replication Failure Teaches Us'. In: *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Sofia, Bulgaria: Association for Computational Linguistics, pp. 1691–1701.

Georgakopoulos, Spiros V et al. (2018). 'Convolutional neural networks for toxic comment classification'. In: *Proceedings of the 10th hellenic conference on artificial intelligence*, pp. 1–6.

Ghiasi, Golnaz, Lin, Tsung-Yi and Le, Quoc V (2018). 'DropBlock: A regularization method for convolutional networks'. In: *Advances in Neural Information Processing Systems*. Ed. by Bengio, S. et al. Vol. 31. Curran Associates, Inc.

Gupta, Ayush et al. (2021). *Hostility Detection and Covid-19 Fake News Detection in Social Media*.

Harris, Zellig (1954). *Distributional Structure*.

He, Ruidan et al. (2019). 'An Interactive Multi-Task Learning Network for End-to-End Aspect-Based Sentiment Analysis'. In: *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*. Association for Computational Linguistics.

Hu, Minghao et al. (July 2019). 'Open-Domain Targeted Sentiment Analysis via Span-Based Extraction and Classification'. In: *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*. Florence, Italy: Association for Computational Linguistics, pp. 537–546.

Hu, Minqing and Liu, Bing (Aug. 2004). 'Mining and summarizing customer reviews'. In: pp. 168–177.

Jurafsky, Dan and Martin, James (2020). *Speech and Language Processing.* Stanford University, pp. 97–128.

Klinger, Roman and Cimiano, Philipp (May 2014). 'The USAGE review corpus for fine grained multi lingual opinion analysis'. In: *Proceedings of the Ninth International Conference on Language Resources and Evaluation (LREC'14).* Reykjavik, Iceland: European Language Resources Association (ELRA), pp. 2211–2218.

Kummervold, Per E et al. (2021). 'Operationalizing a National Digital Library: The Case for a Norwegian Transformer Model'. In: *Proceedings of the 23rd Nordic Conference on Computational Linguistics (NoDaLiDa).* Reykjavik, Iceland (Online): Linköping University Electronic Press, Sweden, pp. 20–29.

Kutuzov, Andrey et al. (May 2021). 'Large-Scale Contextualised Language Modelling for Norwegian'. In: *Proceedings of the 23rd Nordic Conference on Computational Linguistics (NoDaLiDa).* Reykjavik, Iceland (Online): Linköping University Electronic Press, Sweden, pp. 30–40.

Lai, Siwei et al. (2015). 'Recurrent Convolutional Neural Networks for Text Classification'. In: *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence.* AAAI'15. Austin, Texas: AAAI Press, pp. 2267–2273.

Liu, B. (2012). *Sentiment Analysis and Opinion Mining.* Morgan and Claypool Publishers, p. 168.

Martin, Louis et al. (July 2020). 'CamemBERT: a Tasty French Language Model'. In: *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics.* Online: Association for Computational Linguistics, pp. 7203–7219.

Merchant, Amil et al. (2020). 'What happens to bert embeddings during fine-tuning?' In: *arXiv preprint arXiv:2004.14448.*

Mikolov, Tomas et al. (2013). *Efficient Estimation of Word Representations in Vector Space.* arXiv: 1301.3781 [cs.CL].

Mæhlum, Petter et al. (Oct. 2019). 'Annotating evaluative sentences for sentiment analysis: a dataset for Norwegian'. In: *Proceedings of the 22nd Nordic Conference on Computational Linguistics.* Turku, Finland: Linköping University Electronic Press, pp. 121–130.

Pang, Bo and Lee, Lillian (July 2004). 'A Sentimental Education: Sentiment Analysis Using Subjectivity Summarization Based on Minimum Cuts'. In: *Proceedings of the 42nd Annual Meeting of the Association for Computational Linguistics (ACL-04).* Barcelona, Spain, pp. 271–278.

Peltarion (2020). *How to get meaning from text with language model BERT.*

Pereira, F., Halvorsen, P. and Guren, E. (May 2021). 'Applying Multitask Learning to Targeted Sentiment Analysis using Transformer-Based Models'. In: *Proceedings of the Third IN5550 Workshop on Neural Language Processing (WNNLP 2021).* Oslo, Norway: Language Technology Group.

Peters, Matthew E. et al. (2018). *Deep contextualized word representations.* arXiv: 1802.05365 [cs.CL].

Pontiki, Maria, Galanis, Dimitris, Papageorgiou, Haris et al. (June 2016). 'SemEval-2016 Task 5: Aspect Based Sentiment Analysis'. In: *Proceedings of the 10th International Workshop on Semantic Evaluation (SemEval-2016).* San Diego, California: Association for Computational Linguistics, pp. 19–30.

Pontiki, Maria, Galanis, Dimitris, Pavlopoulos, John et al. (Aug. 2014). 'SemEval-2014 Task 4: Aspect Based Sentiment Analysis'. In: *Proceedings* *of the 8th International Workshop on Semantic Evaluation (SemEval 2014)*. Dublin, Ireland: Association for Computational Linguistics, pp. 27–35.

Qu, Lizhen (2013). 'Sentiment analysis with limited training data. Meinungsanalyse mit begrenzten Trainingsdaten'. In.

Ramesh, Aditya et al. (2022). *Hierarchical Text-Conditional Image Generation with CLIP Latents*.

Rogers, Anna, Kovaleva, Olga and Rumshisky, Anna (2020). *A Primer in BERTology: What we know about how BERT works*. arXiv: 2002.12327 [cs.CL].

Ruder, Sebastian, Ghaffari, Parsa and Breslin, John G. (2016). *A Hierarchical Model of Reviews for Aspect-based Sentiment Analysis*. arXiv: 1609.02745 [cs.CL].

Salton, Gerard (1991). 'Developments in Automatic Text Retrieval'. In: *Science* vol. 253, no. 5023, pp. 974–980.

Santos, Cicero dos and Gatti, Maira (Aug. 2014). 'Deep Convolutional Neural Networks for Sentiment Analysis of Short Texts'. In: *Proceedings of COLING 2014, the 25th International Conference on Computational Linguistics: Technical Papers*. Dublin, Ireland: Dublin City University and Association for Computational Linguistics, pp. 69–78.

Scheible, Raphael et al. (2020). *GottBERT: a pure German Language Model*. arXiv: 2012.02110 [cs.CL].

Shi, Tianze and Liu, Zhiyuan (2014). *Linking GloVe with word2vec*. arXiv: 1411.5595 [cs.CL].

Socher, R. et al. (2013). 'Recursive Deep Models for Semantic Compositionality Over a Sentiment Treebank'. In: p. 12.

Tang, Duyu, Qin, Bing and Liu, Ting (Sept. 2015). 'Document Modeling with Gated Recurrent Neural Network for Sentiment Classification'. In: *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*. Lisbon, Portugal: Association for Computational Linguistics, pp. 1422–1432.

Toprak, Cigdem, Jakob, Niklas and Gurevych, Iryna (July 2010). 'Sentence and Expression Level Annotation of Opinions in User-Generated Discourse'. In: *Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics*. Uppsala, Sweden: Association for Computational Linguistics, pp. 575–584.

Turney, Peter (July 2002). 'Thumbs Up or Thumbs Down? Semantic Orientation Applied to Unsupervised Classification of Reviews'. In: *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*. Philadelphia, Pennsylvania, USA: Association for Computational Linguistics, pp. 417–424.

Vaswani, Ashish et al. (2017). 'Attention is all you need'. In: *Advances in neural information processing systems*, pp. 5998–6008.

Velldal, Erik et al. (May 2018). 'NoReC: The Norwegian Review Corpus'. In: *Proceedings of the Eleventh International Conference on Language Resources and Evaluation (LREC 2018)*. Ed. by chair), Nicoletta Calzolari (Conference et al. Miyazaki, Japan: European Language Resources Association (ELRA).

Walker, Nicholas (May 2021). 'To BERT or not to BERT: Design Choices Matter in Targeted Semantic Analysis .' In: *Proceedings of the Third IN5550 Workshop on Neural Language Processing (WNNLP 2021)*. Oslo, Norway: Language Technology Group.

Welleck, Sean and Cho, Kyunghyun (2020). *MLE-guided parameter search for task loss minimization in neural sequence modeling.*

Wiebe, Janyce, Wilson, Theresa and Cardie, Claire (2005). 'Annotating Expressions of Opinions and Emotions in Language'. In: *Language Resources and Evaluation.*

Wiedemann, Gregor et al. (2019). 'Does BERT make any sense? Interpretable word sense disambiguation with contextualized embeddings'. In: *arXiv preprint arXiv:1909.10430.*

Xue, Linting et al. (June 2021). 'mT5: A Massively Multilingual Pre-trained Text-to-Text Transformer'. In: *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies.* Online: Association for Computational Linguistics, pp. 483–498.

Yu, Hong and Hatzivassiloglou, Vasileios (2003). 'Towards answering opinion questions: Separating facts from opinions and identifying the polarity of opinion sentences'. In: *Proceedings of the 2003 conference on Empirical methods in natural language processing*, pp. 129–136.

Zhang, Xiang, Zhao, Junbo and LeCun, Yann (2016). *Character-level Convolutional Networks for Text Classification.* arXiv: 1509.01626 [cs.LG].

Zhang, Xiaoqian et al. (2011). 'Polarity Shifting: Corpus Construction and Analysis'. In: *2011 International Conference on Asian Language Processing*, pp. 272–275.

Zhang, Ye and Wallace, Byron (2016). *A Sensitivity Analysis of (and Practitioners' Guide to) Convolutional Neural Networks for Sentence Classification.* arXiv: 1510.03820 [cs.CL].

Øvrelid, Lilja et al. (May 2020). 'A Fine-grained Sentiment Dataset for Norwegian'. English. In: *Proceedings of the 12th Language Resources and Evaluation Conference.* Marseille, France: European Language Resources Association, pp. 5025–5033.