MSc Thesis Task Description

Daniel Balint
candidate for MSc degree in Mechatronics Engineering

Intelligent Decision Optimization System for Agile Scrum Project Management

Agile Scrum is a process framework that focuses on continuously delivering and sustaining complex products at the highest possible value and has been widely used for project management in software development. Scrum product development is centered around end user feedback and iteratively delivering products in short development cycles called sprints, during which a usable, potentially releasable product increment with a new feature is created. Assembling sprint goals, laying out product pathways for the next cycle requires a sophisticated decision-making process from the Scrum team that factors in number of variables. Artificial intelligence can assist managers and team members by automating repetitive, high-volume tasks, by supporting the decision-making process to reduce turnaround time and to increase productivity and business value of deliverables. Leveraging the capabilities of machine learning, a decision-making AI is able to gather and analyze relevant information, diagnose problems and propose possible courses of action while emulating human proficiency.

This thesis introduces an intelligent decision support system that can assist Scrum teams during sprint planning to set goals and steer product development in a way that will deliver more value to their customers. The system will be able to evaluate end user reviews and stakeholder requests through deep-learning based natural language processing and will be able to estimate the effort required for the proposed design changes. By an optimization algorithm, based on effort and priority, the AI will select and organize backlog items into sprints to propose an optimal development plan.

Tasks to be performed by the student includes:

- Studying Scrum decision-making processes and identifying areas to be supported by the AI
- Laying out the workflow of the decision optimization system
- Implementing machine learning based feature extraction (NLP, e.g. word2vec) to identify backlog items
- Creating product backlog with story points by effort estimation using the extracted features
- Implementing an optimization algorithm to construct the ideal sprints from the backlog for a program iteration
- Integrating the decision support system on a chatbot
- Testing the system with data sets of customer reviews, reported bugs, design change requests

Supervisor at the department: Khalid M. Kahloot, PhD Student.
Budapest, 27 September 2020

Dr. Hassan Charaf
professor
head of department

Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Automation and Applied Informatics

Intelligent Decision Optimization System for Agile Scrum Project Management

Master's Thesis

Author
Dániel Bálint

Advisor
Ádám Kovács
Khalid Kahloot

May 23, 2021

Contents

HALLGATÓI NYILATKOZAT

Alulírott *Bálint Dániel*, szigorló hallgató kijelentem, hogy ezt a diplomatervet meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző(k), cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy autentikált felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Budapest, 2021. május 23.

Bálint Dániel
hallgató

Kivonat

Az agilis Scrum egy komplex termékek fejlesztésére és fenntartására létrehozott projektmenedzsment-keretrendszer, ami széles körben elterjedt szoftverfejlesztési területeken. A Scrum folyamatok középpontjában a felhasználói visszajelzések és termékek iteratív, rövid ciklusokban történő inkrementális fejlesztése áll. Az egy-két hetes fejlesztési ciklusok, úgynevezett sprintek során mindig egy potenciálisan kibocsátható, új funkcionalitással rendelkező vagy hibajavított termék inkrementum realizálódik. A sprint tervezés, a sprintcélok összeállítása a következő fejlesztési ciklushoz egy rendkívül kifinomult döntéshozatali folyamatot követel meg a Scrum csapattól, amelynek során számos változót kell figyelembe venni. Az ismétlődő, nagy volumenű feladatok mesterséges intelligenciával történő automatizálása nagy mértékben segítheti a menedzsereket és a fejlesztőket a döntéshozatali folyamat meggyorsításával, optimalizálásával, és így növelheti a csapat hatékonyságát valamint a kiadott termékek hozzáadott értékét.

Diplomatervem egy intelligens döntéstámogató rendszert mutat be, amely a sprint tervezés során segíti a Scrum csapatokat az optimális sprint célok kitűzésében és a termékfejlesztési folyamat irányításában annak érdekében, hogy magasabb hozzáadott értéket tudjanak szolgáltatni ügyfeleik számára. A rendszer képes egy mély tanulás (deep learning) alapú nyelvi modellen keresztül feldolgozni a termék backlog szöveges tételeit (pl. hibajelentések, bugok leírásai), és képes megbecsülni a a kívánt új funkcionalitás megvalósításához vagy probléma megoldásához szükséges relatív erőfeszítést, erőforrást (munkaóra, story-point). Dolgozatom során bemutatom, milyen kihívásokat állít a sprint tervezés és különösen a story-point becslés a Scrum csapatok elé, és hogyan lehet egy machine learning alapú modellt megalkotni ezen folyamatok automatizálására. Részletes leírást adok a döntéstámogató rendszer központi részéről, a story-point becslési modellről, amely során két lehetséges megoldás implementációját és hátterét mutatom be. Ezek közül az első egy LSTM-alapú architektúra, a másik pedig egy előtanított BERT (Bidirectional Encoder Representations from Transformers) modell finomhangolása. Kiértékelem és összehasonlítom a két megközelítést nyilvánosan elérhető, nyílt forráskódú projektek backlog adatainak átfogó gyűjteményével, és kitérek arra, hogy mik az egyes megoldások előnyei és hátrányai.

A Transformer alapú előtanított architektúrák forradalmasították a természetes nyelv feldolgozás, nyelvtechnológiák területét, és hatékony eszközként szolgálnak nagy teljesítményű machine learning megoldások fejlesztéséhez. Az implementált modellek összehasonlításával bemutatom, hogy a Transformer alapú neurális hálók nyelvi reprezentációs képességei hogyan nyilvánulnak meg a story-point becslés speciális alkalmazásában, és miért előnyös a BERT architektúra használata a javasolt döntéstámogató rendszerben.

Abstract

Agile Scrum is a process framework that focuses on continuously delivering and sustaining complex products at the highest possible value and has been widely used for project management in software development. Scrum product development is centered around customer feedback and iteratively delivering products in short development cycles, called sprints, during which a usable, potentially releasable product increment with a new feature is created. Assembling sprint goals, laying out product pathways for the next cycle requires a sophisticated decision-making process from the Scrum team that factors in number of variables. Artificial intelligence can assist managers and team members by automating repetitive, high-volume tasks, by supporting the decision-making process to reduce turnaround time and to increase productivity and business value of deliverables.

This thesis introduces an intelligent decision support system that can help Scrum teams during sprint planning to set goals and steer product development in a way that will deliver more value to their customers. The system is able to process the textual artifacts of a product backlog (e.g. descriptions of issue reports, bugs, user stories) through a deep learning-based language model and is be able to make an estimate of the effort required for completing a proposed new feature or resolving an issue. Throughout this thesis, I show what the challenges behind sprint planning, and specifically, effort estimation are and how a machine learning model can be formulated to automate these processes. I provide a detailed discussion on the central part of the decision support system, the story-point estimation model, and I present two distinct solutions for it, one of which is an LSTM-based architecture and the other is fine-tuning a pre-trained BERT (Bidirectional Encoder Representations from Transformers) model. I evaluate and compare the two approaches using a comprehensive collection of issue report datasets from publicly available, open-source projects and elaborate on what the benefits and drawbacks of each solution are.

Transformer-based pre-trained architectures revolutionized the field of natural language processing and provide a powerful tool for developing high-performance machine learning solutions. Upon comparing the implemented models, I show how the superior capabilities of Transformer-based representations manifest themselves in the specific application of effort estimation and why it is beneficial to use BERT in the proposed decision support system.

Chapter 1

Introduction

1.1 Scrum Process Framework

Scrum is an agile process framework that focuses on user feedback and the rapid, iterative delivery of business value for the customer, and it is mainly used in software development projects. Product development happens in short, few week long cycles called sprints, during which a new potentially releasable product increment with a new feature or bugfix is created. Before every development cycle, the Scrum team during the sprint planning meeting sets up the sprint goals by selecting certain items, tasks from the product backlog that should be completed in the upcoming sprint [1]. These could be issue reports, bugfixes, a new feature based on customer feedback or stakeholder request. Figure 1.1 demonstrates the typical flow of the Scrum development. The selection of this subset of items is a key momentum in the Scrum process, and it has decisive impact on the success of the project. To formulate an optimal sprint, a rigorous evaluation of the product backlog is needed by the product owner and the team, which involves the prioritization of items, decomposition or refinement of certain tasks and estimating the relative effort required by each of the tasks to be completed. During effort estimation, the team assigns to each item a relative value – known as a story point – that represents the relative effort needed to resolve the indicated issue, user story.

1.2 Project Scope

Laying out product pathways and sprint planning requires a sophisticated decision-making process that factors in number of variables, and it is heavily reliant on expert opinion. Although existing agile tools are useful, their support is limited to creating, managing, and tracking project artifacts, and visualising historical project data such as burn-down charts and other agile reports. Current agile project management tools lack advanced analytical methods that are capable of harvesting valuable insights from project data for prediction, estimation, planning and action recommendation. Many decision-making tasks in agile projects are still performed by agile teams

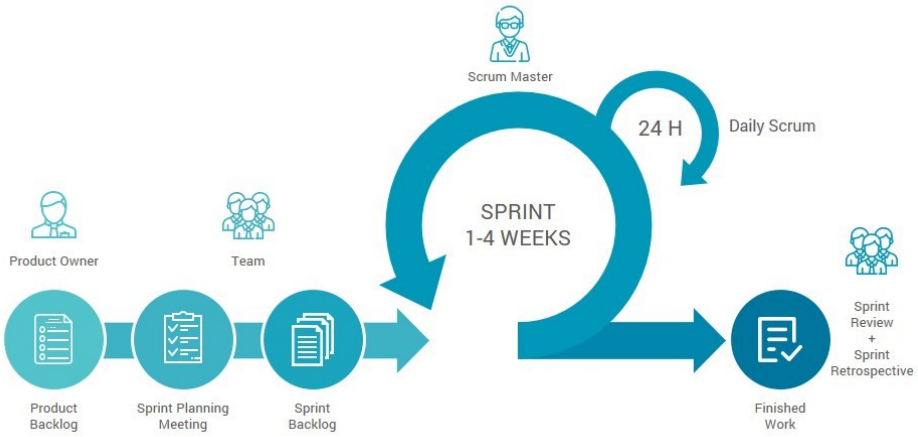

Figure 1.1: Scrum process framework [1].

without machinery support, but machine learning can assist the team by automating repetitive, high-volume tasks.

Scrum is centered around the product backlog, and one of the critical parts of the planning process is the effort estimation of the backlog items. This is in most cases exclusively based on the team members' expertise and experience that is inherently a limited and timely constrained subset of all the knowledge acquired by the company or the industry that the company operates in. Moreover, due to the continuous human resource flow across development firms and groups, the team's cumulative expertise fluctuates, and certain developers can have limited information about their product's history. However, a high-performance computational engine with machine learning capabilities, an AI agent, could process the entirety of a product's history and recorded backlogs in a timely manner and could process seemingly unlimited amount of industry specific documents, and using the gathered information, it could assist decision-making processes.

The problems mentioned previously and the lack supportive tools creates an opportunity for artificial intelligence to significantly improve the practice of Scrum project management. The goal of this research project is to develop a machine learning-based decision optimization system that could learn to predict the relative effort for each item using historic backlog data, and by using these effort estimates, can also formulate sprint plans and suggest the refinement of certain tasks.

In this thesis, I outline a conceptual solution for a Scrum decision support system, and I present its implementation in detail with special, predominant emphasis on the story-point estimation engine, which ultimately is the core of the concept. The effort estimation task of the issue reports is essentially a natural language processing problem, for which I propose two distinct language models, namely an LSTM-based architecture and a fine-tuned BERT model, and also discuss their build-up and implementation in high detail. I also provide an overview of the used datasets, the training process and the evaluation metrics. After evaluating both models with datasets from several different product backlogs of open-source projects, I discuss

the results and my experimental findings, and I compare the performance of the two solutions. For verification purposes, the code base of the implementation and the datasets are publicly available in a GitHub repository[1].

To present the theoretical background and relevant literature behind the project, I recap some of the recent proceedings concerning the topic of story-point estimation and the adoption of artificial intelligence and machine learning in agile processes, I give a summary of the field of natural language processing and present the state-of-the-art and some of the more commonly used NLP architectures especially focusing on LSTM by Hochreiter et al. [2], the Transformer architecture by Vaswani et al. [3] and BERT by Devlin et al. [4].

[1] https://github.com/b96daniel/ScrumSprintPlanning

Chapter 2

Background

2.1 Machine Learning in Agile

In recent years, machine learning and artificial intelligence had a major impact on software analytics and effort estimation in agile software development as a systematic literature review done by M. Diego Hernandez et al. [5] has revealed it in 2020, while back in 2014, according to Usman et al. [6], studies about effort estimation mostly investigated expert-based techniques, such as Planning Poker. Although Planning Poker is still one of the most frequently used effort estimation method in agile development, and human expertise will certainly be needed in the future as well, AI-based assistance in project management can be beneficial for agile teams.

In an article [7], H. K. Dam et al. outlined a number of important areas in Scrum management that remain challenging due to the lack of support by current agile tools, such as identifying backlog items, refining backlog items, sprint planning and pro-actively monitoring sprint progress and managing risks. Figure 2 demonstrates the possible roles of AI in agile development. They also proposed a concept of a machine learning-based project assistant to help development teams overcome the aforementioned challenges and to significantly improve the practice of agile management.

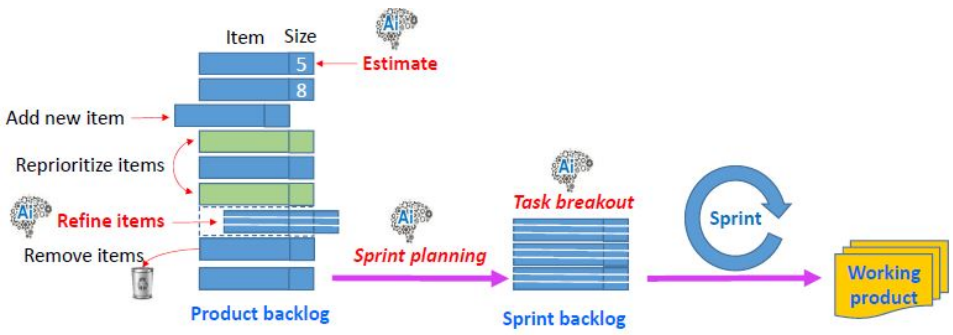

Figure 2.1: Possible AI support in agile [7].

2.2 Story-Point Estimation

The most crucial part of Scrum management is the sprint planning meeting and the effort estimation of user stories, during which usually a team assigns a relative value, i.e. story point, to each story that represents the relative effort (size and complexity of tasks) needed to resolve the indicated issue or implement the requested new feature. This story point approach is widely used among agile teams, and points are commonly assigned following the Fibonacci sequence [6]. It is important to note that story points are based on the team's past experience and knowledge and the consensus among team members, and they are by no means an objective measurement unit. Therefore, estimates for the same task can vary among different teams. Furthermore, estimates factor in a number of uncertainties regarding the team and the project besides the complexity and the amount of work that a task requires [8]. Considering the above mentioned reasons, creating an efficient story point estimation model is a critical and challenging part of developing a decision support system for Scrum. Several research papers and studies have been published around the topic of story point estimation.

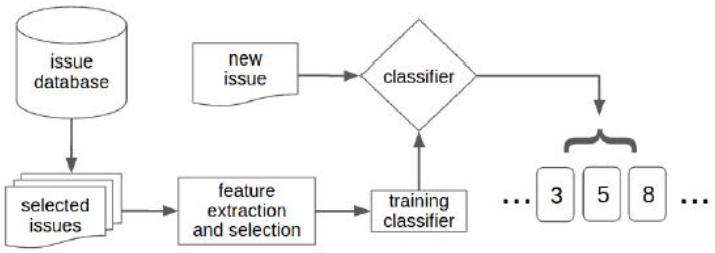

Figure 2.2: The model proposed by S. Porru et al.[9].

The estimation method proposed by S. Porru et al. [9] uses term frequency - inverse document frequency transformation (TF-IDF) to extract features from the natural text of issue reports, and then those features are used to train a machine learning classifier. TF-IDF is a statistical measure used to evaluate how important a word is to a document in a collection or corpus, and it is proportional to the number of occurrences of each term in the document and is offset by the number of documents in the collection that contain that term. After some feature engineering, a classifier can be trained with the issue datasets by exploiting the correlation between the TF-IDF weights in an issue report and its corresponding story point. Four different algorithm were evaluated for this purpose, namely, SVM (support-vector machine), K-nearest neighbour, Decision Tree and Naive Bayes. SVM proved to be the most accurate with an MMRE (Mean Magnitude of Relative Error) of 0.47. However, the datasets were relatively small and only included a few repositories. 5607 issues were gathered from 8 open source projects. In comparison, the experiment of Haugen [10] on 19 developers using Planning Poker resulted in an MMRE of 0.48 by considering the story points assigned by developers before and after task implementation.

A deep learning-based story point estimation (Deep-SE) approach by Choetkiertikul et al. [11] is utilizing word embedding and long short-term memory (LSTM) for document representation and using a Recurrent Highway Net (RHWN) [12] – a special recurrent neural network – for deep representation and a differentiable regression

to produce the story point estimates. This system is trainable from end-to-end, meaning that the data signals are passed from the raw text in user stories to the final output node, and the prediction error is propagated from the output node all the way back to the word layer. The model takes the title and the description of a backlog item exported from an issue tracking system and predicts the story point without any manual feature engineering.

The model in [11] was trained with a dataset of 23,133 issues in 16 different open-source projects and was evaluated against various different methods such as Random Forest, SVM, Automatically Transformed Linear Model (ATLM), Bag of Words. The Deep-SE was also compared against the existing Porru et al.'s TF-IDF [9] approach, and it outperformed all the above mentioned methods.

2.3 Natural Language Processing

Natural language processing, abbreviated as NLP, is a multi-disciplinary field of data science, linguistics and artificial intelligence that focuses on the interactions between human language and computers, and aims to develop algorithms and models that analyze speech and language, derive meaning from text, uncover contextual patterns, and therefore give the ability to machines to understand natural language, interpret the contents of documents, or ultimately, communicate with humans. Some popular applications of NLP, without exclusivity, include machine translation, speech recognition, voice-to-text, sentiment analysis, document summarization, question answering, natural language generation. This technology is absolutely imperative for our application since the decision support system has to extract meaningful insights from the unstructured textual data of product backlogs.

This discipline has been evolving over many decades, but neural networks and deep learning have significantly accelerated the advancements. In the early days, most NLP systems were based on symbolic methods, hand-written set of rules, such as rule-based parsing, morphology, semantics. However, in the late 1980s, 1990s, statistical methods and early machine learning methods became more prevalent, such as decision trees, support-vector machines, bag-of-words, n-gram, term frequency - inverse document frequency transformations [13]. These statistical methods require laborious feature engineering, hence they had become obsolete as the field shifted towards artificial neural networks.

The rise of deep learning has revolutionized the field of natural language processing, as it did to many other disciplines, and introduced a new paradigm in which NLP systems are designed to be end-to-end trainable and are able to learn sequence-to-sequence transformations directly, and therefore, obviating the need for intermediary feature engineering.

2.3.1 Sequence-to-Sequence Models

Since natural text is essentially a sequence of words, based on the principle of semantic composioniality [14] – that the meaning of a natural text is determined

by the meaning of its constituents (words) and the rules to combine them – the fundamental NLP task is to learn sequential input-output relations. Therefore, sequence-to-sequence models such as RNN-based encoder-decoder networks [15] or Transformer-based models [3] are commonly used for NLP applications. Recurrent neural networks (RNN) would be a reasonable choice to model relationships between words and to extract digitally interpretable meaning from a document because an RNN can take a sequence of variable length $(\mathbf{x_1}, ..., \mathbf{x_n})$ as an input and generate a sequence of outputs $(\mathbf{h_1}, ..., \mathbf{h_n})$ by using its recurrent connections, depicted in Figure 2.3. At each time t time step, the network takes $\mathbf{x}^{(t)}$ and $\mathbf{h}^{(t-1)}$ as inputs and outputs the next hidden state $\mathbf{h}^{(t)}$ as follows, where W and U are weight matrices and f is a nonlinear activation function [16].

$$\boldsymbol{h}^{(t)} = f\left(W\boldsymbol{h}^{(t-1)} + U\boldsymbol{x}^{(t)}\right) \tag{2.1}$$

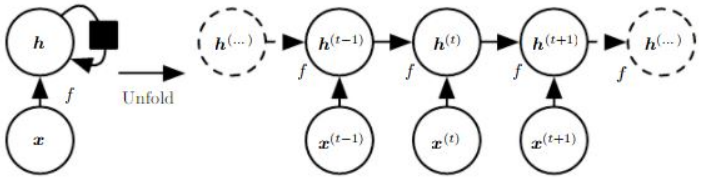

Figure 2.3: Recurrent Neural Network unfolded [16].

Note that RNNs can take variable-length input sequences, which is important since sentences obviously differ in length. Creating encoder-decoder architectures with multilayer RNNs enable us to map input sequences to output sequences that have a different length than the input, which is particularly useful in machine translation, for instance. With applying a classifier or regression layer on top of recurrent layers, we can utilize our network for plenty of NLP tasks, such as sentiment analysis or text categorization, et cetera. The use of a vanilla RNN, however, is not practical usually because of the exploding and vanishing gradient problem that makes it difficult to make use of information distant from the current point of processing [17]. Instead, long short-term memory or gated recurrent units (GRU) [18] are more commonly used, which were specifically developed to solve this problem. Attention-based, Transformer-based encoder-decoder models are also highly effective and prevalent in NLP applications. In fact, they have shown significantly better performance than their predecessors [3], and derivatives of the Transformer are now considered to be the state-of-the-art [19].

2.3.2 Word Embedding

These deep learning models are utilizing artificial neural networks for NLP, and therefore, each word of the input text has to be converted into a numerical fixed-length vector representation in order to capture that word's semantic properties. This process is done through a word embedding matrix or look-up table where each word of the vocabulary is encoded with its own unique low-dimensional vector representation such that words that are closer in the vector space are expected

to have a similar meaning [17]. For instance, in case of a $d = 3$ dimensional word embedding, the word *apple* could have a representation of $\mathbf{w}_{apple} = [0.5, 0.1, 0.3]$, and other words in the vocabulary that are often associated with it would be represented by a vector that have a low Euclidean distance to $\mathbf{w}_{apple}$. Consequently, embedding a vocabulary of a large number (K) of words on d dimensional space would result in a $d \times K$ sized embedding matrix, whereas in case of one-hot encoding, where each word have a binary representation of a K sized one-hot vector, the encoding matrix would be $K \times K$ in size.

Essentially, this word embedding method maps the huge vector space that has the size of the vocabulary to a continuous low-dimensional vector space. Word embeddings have to be trained by large corpora of unlabelled natural texts with language modeling, predicting the next word of a sequence based on the previous words. RNNs can be used for next word prediction by training them to produce the probability distribution over a V vocabulary given a sequence of words as input [20]. The probability distribution for the next word can be generated by taking the hidden state of the RNN in the last step and feeding it into a softmax function for every $\boldsymbol{w}_j \in V$ word vector as follows,

$$p\left(\boldsymbol{x}^{(t)} = \boldsymbol{w}_j \mid \boldsymbol{x}^{(t-1)}, \ldots, \boldsymbol{x}^{(1)}\right) = \frac{\exp\left(\boldsymbol{w}_j^T \boldsymbol{h}^{(t)}\right)}{\sum_{j=1}^{K} \exp\left(\boldsymbol{w}_j^T \boldsymbol{h}^{(t)}\right)} \tag{2.2}$$

where $\mathbf{x}^{(t-1)}, ..., \mathbf{x}^{(1)}$ is the input sequence of embedded word vectors, $\mathbf{x}^{(t)}$ is the representation of the next word, $\mathbf{w}_j$ is the embedded vector of the given word that we are calculating the probability for, $\mathbf{h}^{(t)}$ is the hidden state produced by the RNN at step t, and K is the size of the vocabulary [20]. If we calculate p for every word $(1, ..., j)$, we will obtain the probability distribution $\mathbf{P}^{(t)}$ for the next element of the sequence over the whole vocabulary, which we can summarize in a K-sized vector. With $\mathbf{P}^{(t)}$ and the $\mathbf{y}^{(t)}$ one-hot representation of the actual word at step t, we can calculate the L cross-entropy loss as follows,

$$L = -\mathbf{y}^{(t)} \cdot \log(\mathbf{P}^{(t)}) \tag{2.3}$$

which then we can back-propagate to adjust the RNN and embedding weights simultaneously with gradient descent.

$$w_{ij} = w_{ij} - \eta \frac{\partial L}{\partial w_{ij}} \tag{2.4}$$

2.3.3 Long Short-Term Memory

A popular approach for natural language processing is LSTM-based modelling. Long short-term memory [2] is a special variant of recurrent networks that can control how much information persists in its memory cell through the forget gate mechanism. The basic elements of an LSTM unit are the input, output and forget gates, which control the information flow, and the memory cell (c), which can store the information from previous time steps . An LSTM unit has actually four gates because the

input gate has two activation functions: the cell input activation (g) with a hyperbolic tangent activation, which controls how much to write to the memory cell, and the input gate activation (i), which controls whether to write to the memory cell. The resulting vectors are then element-wise multiplied with each other. The forget gate has a sigmoid activation
($\sigma(x) = \frac{1}{1+e^{-x}}$), and it controls how much of the previous information from the memory should be removed. The output gate also has a sigmoid activation, and it controls how much of the input information should be revealed [2].

In summary, the gates of the LSTM are taking the x_t element of a sequence and the previous hidden state h_{t-1} as inputs at each (t) time step, and their outputs control the two internal states of the LSTM unit – the cell memory c_t and the hidden state h_t. The cell memory is refreshed at each time step by forgetting a part of the old information and acquiring the new gated input. If the output of the forget gate is 0, then the previous data in the memory is completely erased, while if it is 1 the past memory remains intact.

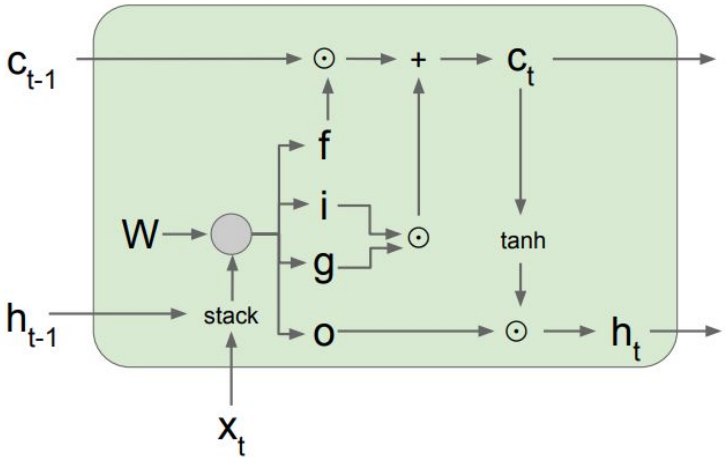

Figure 2.4: LSTM cell.

$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{pmatrix} W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix} \tag{2.5}$$
$$c_t = f \odot c_{t-1} + i \odot g$$
$$h_t = o \odot \tanh(c_t)$$

This unique architecture allows the LSTM to map a sequence into another sequence and to effectively learn long-term dependencies in sequential information. The forget gate effectively solves the problem of vanishing and exploding gradients, that vanilla RNNs suffer from, by controlling how much information can persist in the memory cell.

The LSTM-based encoder-decoder architecture was introduced by Sutskever et al. in 2014, [21], and they achieved on an English to French translation task from the WMT-14 dataset a BLEU score of 34.8 on the entire test set.

2.3.4 Attention Mechanism

In LSTM-based, or generally in RNN-based encoder-decoder architectures, the limited information that the context vector can carry turned out to be a bottleneck for machine translation tasks. Long sentences still pose a challenge for these models. In former seq2seq models, the encoder layers would encode the input sequences into a single fixed-size vector, the last hidden state, that ideally would contain all the relevant information needed by the decoder to produce its predictions. However, the context vector does not necessarily convey all the relevant information, and with this approach, each decoder steps receive the same information instead of the information relevant specific to that step. A solution was proposed in Bahdanau et al., 2014 [22], which introduced and refined a technique called Attention, that highly improved the efficacy of machine translation systems. This architecture consists of a bidirectional RNN as an encoder and a decoder that emulates searching through a source sentence during decoding, which allows the model to focus on the relevant parts of the input sequence as needed.

The proposed attention mechanism introduces a new input for each decoder step. Instead of passing the last hidden state of the encoding stage, the encoder passes the weighted sum $\mathbf{c_i}$ over all the encoder $\mathbf{h_j}$ hidden states to the decoder at each i time-step as follows,

$$c_i = \sum_{j=1}^{T_x} a_{ij} h_j \tag{2.6}$$

where T is the number of hidden states or words in the source sentence. The weight a_{ij} of each hidden state $\mathbf{h_j}$ is computed by

$$a_{ij} = \frac{\exp\left(e_{ij}\right)}{\sum_{k=1}^{T_x} \exp\left(e_{ik}\right)} \tag{2.7}$$

which is a softmax function over e_{ij}, which is the output of an alignment model that scores how well the inputs around position j and the output at position i match. The e_{ij} scores are calculated by

$$e_{ij} = f\left(s_{i-1}, h_j\right) \tag{2.8}$$

where $\mathbf{s}_{i-1}$ denotes the hidden state of the decoder in the previous step, and $\mathbf{h}_j$ is the current hidden state of the encoder [22]. These vectors are often called the query vector $\mathbf{q}$ and the key vector $\mathbf{k}$ respectively. Figure 2.5 demonstrates the attention model.

The alignment model f is implemented as a feed-forward neural network which is jointly trained with all the other components of the proposed system. Using this alignment approach, the a_{ij} values can be understood as probabilities, and e_{ij} as its associated energy, that reflects the importance of the encoder hidden state h_j with respect to the previous decoder hidden state s_{i-1} in deciding the next state s_i. Intuitively, this implements a mechanism of attention in the decoder, meaning

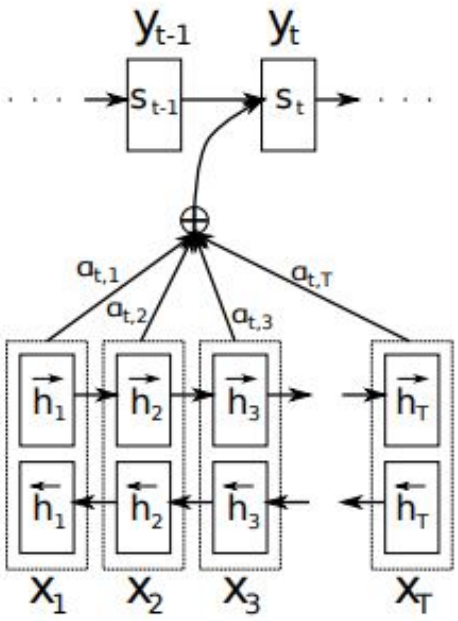

Figure 2.5: The graphical illustration of the attention model trying to generate the t-th target word y_t given a source sentence $(x_1, ..., x_T)$ [22].

that the decoder selects parts of the source sentence to pay attention to. By letting the decoder have an attention mechanism, we relieve the encoder from the burden of having to encode all information in the source sentence into a fixed-length vector [22]. Figure 2.6 provides a visual demonstration of the a_{ij} weights between the input and output sentence during a French-English translation task.

This attention model proposed by Bahdanau et al. achieved a BLEU score of 36.15 on the English to French translation task from the WMT-14 dataset with a sentence length of 50, outperforming the conventional encoder-decoder architectures on longer sentences and also achieving comparable results on short sentences [22].

Attention Types. The alignment model or scoring function f mentioned in Equation 2.8 can be implemented in different ways. For instance, as in the original attention model, it can be a feed-forward neural network (Equation 2.9) where the query and key are just concatenated and fed into the network.

$$f_{FNN}(\boldsymbol{q}, \boldsymbol{k}) = \boldsymbol{w}_2 \tanh\left(\boldsymbol{W}_1[\boldsymbol{q} : \boldsymbol{k}]\right) \tag{2.9}$$

The scoring function can also be simple dot product of $\mathbf{q}$ and $\mathbf{k}$ without weights (Equation 2.10), but in this case the vectors have to be of the same size. Another problem can be that the dot product is proportional to the vector dimensions. The so-called Scaled Dot Product (Equation 2.11) alleviates this problem by dividing the dot product with the square root of the size of the key vector [3].

$$f_{DP}(\boldsymbol{q}, \boldsymbol{k}) = \boldsymbol{q}^\top \boldsymbol{k} \tag{2.10}$$

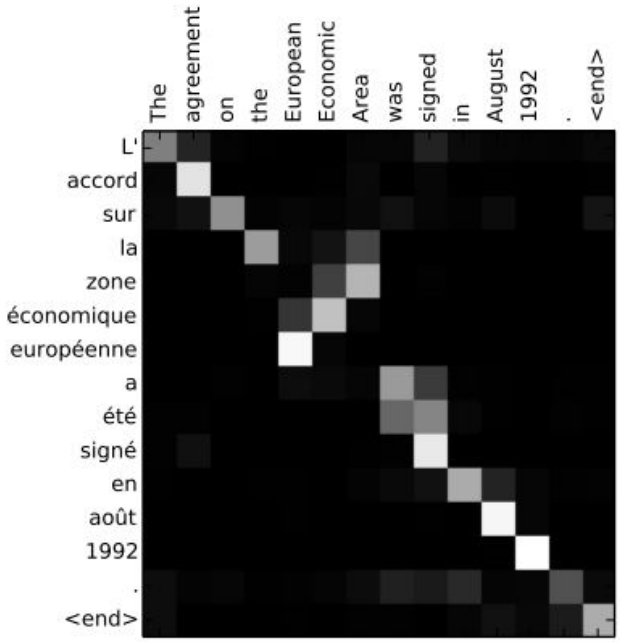

Figure 2.6: Each pixel represent the weight a_{ij} of the annotation of the j-th source word for the i-th target word [22].

$$f_{SDP}(\boldsymbol{q}, \boldsymbol{k}) = \frac{\boldsymbol{q}^\top \boldsymbol{k}}{\sqrt{|\boldsymbol{k}|}} \tag{2.11}$$

Previously it was established that the query and key vectors are hidden states from the decoder and the encoder respectively, but with a method called self-attention, attention can be computed from only one sequence of symbols [3]. This means that the query and key vectors are both coming from the same hidden states, and therefore, resulting in an attention-based sentence encoding that depends on all of its parts. Self-attention can be implemented as a standalone layer in both encoders and decoders, and it plays a pivotal role in the Transformer architecture.

2.3.5 The Transformer

In the paper Attention All You Need, Vaswani et al. [3] presented a new encoder-decoder architecture that is solely based on attention, called the Transformer, which has achieved state-of-the-art results at the time of its publication in machine translation tasks, and today serves as basis for other high-performing architectures in various NLP applications. Its biggest benefit is that it allows for much more parallelization due to its attention mechanism. It does not need to process the sequential data in order, but instead it identifies the context for each position. Because of its advantages, the Transformer quickly replaced older, RNN or LSTM-based models in NLP problems.

The Transformer follows the typical encoder-decoder concept: it is built up from two sub-networks, the encoder and decoder, but it uses stacked self-attention and point-wise, fully connected layers for both. The model is auto-regressive at each step, meaning that it consumes the previously generated symbols as additional input when generating the next. Figure 2.7 illustrates the Transformer architecture.

Encoder-Decoder Layer Stack. The encoder sub-net is made up of a stack of $N = 6$ identical layers connected to each other, the output of a layer serving as input for the next. The layer count 6 is an arbitrary number chosen by the creators. Each layer is broken down to two sub-layers. The first is a multi-head self-attention mechanism, and the second is a simple, position-wise fully connected feed-forward network. Residual connection [23] is used around each of the two sub-layers with layer normalization, which means that the output of each sub-layer is $LayerNorm(x + Sublayer(x))$. The words of the input sequence are embedded into $d_{model} = 512$ dimensional vectors. Then, all embedded words of the sequence flow into the network in parallel. Hence the parallelization, a positional encoding takes place before the first encoder layer to inject information about the relative position of the words. After this, the list of vectors flow into the encoder sub-net. All sub-layers in the model produce outputs with $d_{model} = 512$ dimension, therefore the maximum length of an input sentence and the size of the embedded vectors is d_{model}.

The decoder also conveys a stack of $N = 6$ identical layers. However, in addition to the two sub-layers in each encoder layer, the decoder contains a third sub-layer in the middle, which performs multi-head attention over the output of the encoder stack and the output of the first sub-layer. Residual connections around each of the sub-layers are also used here, followed by layer normalization. An important feature of the decoder, which is different from the encoder, is that its first multi-head self-attention block takes already generated output word embeddings from the output sentence as inputs. Additionally, the first self-attention sub-layer in the decoder is masked to prevent positions from attending to subsequent positions. Masking and the fact that the output embeddings are offset by one position ensure that the predictions for position i can depend only on the known outputs at positions less than i [3].

Multi-Head Attention. In the self-attention sub-layers the Transformer employs scaled dot-product attention, which is based on Equation 2.11 mentioned previously. According to Vaswani et al. [3], an attention function can generally be described as mapping a query and a set of key-value pairs to an output, where the query, keys, values and output are all vectors, and the output is produced as a weighted sum of the values, where the weight assigned to each value is computed by a compatibility function of the query with the corresponding key.

The input vectors, at the first layer the word embeddings, are projected into queries $\mathbf{q}_i$ and keys $\mathbf{k}_i$ of dimension d_k and values of dimension d_v. For instance, from an

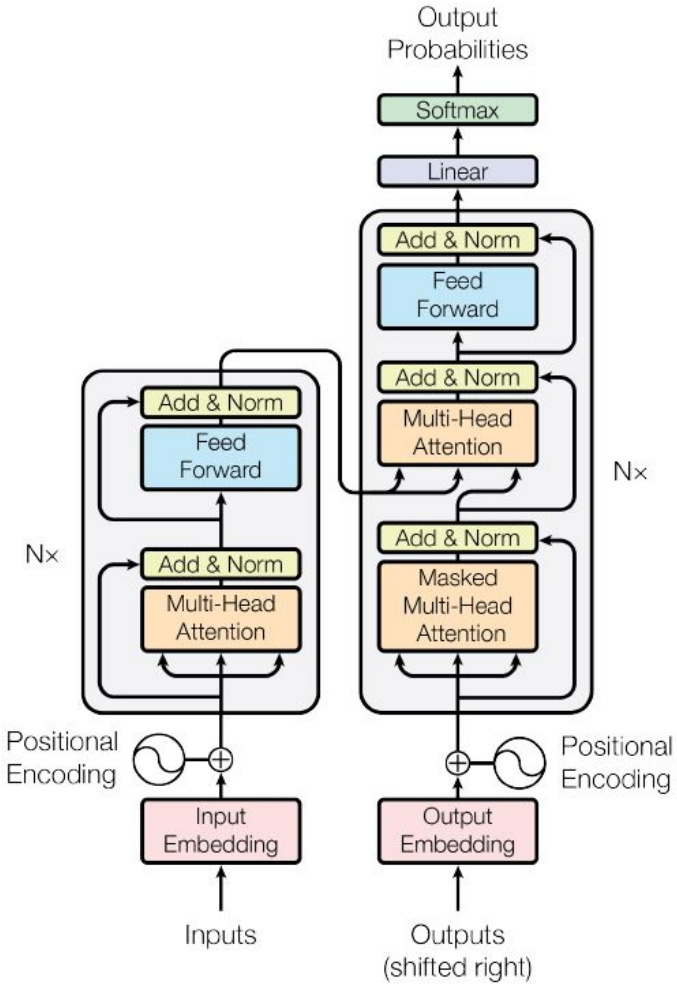

Figure 2.7: The Transformer architecture. The encoder stack is shown on the left, and the decoder stack is on the right [3].

input embedding $\mathbf{x}_0 \in \mathbb{R}^{d_{model}}$, the query vector can be computed by multiplying the input row vector with the query projection matrix $W_Q \in \mathbb{R}^{d_{model} \times d_k}$.

$$\boldsymbol{q}_0 = \boldsymbol{x}_0^T W_Q \tag{2.12}$$

Similarly, the key and value vectors can be calculated with the projection matrices $W_K \in \mathbb{R}^{d_{model} \times d_k}$, $W_V \in \mathbb{R}^{d_{model} \times d_v}$ respectively. Then to produce the scaled dot-product attention, the dot products of the query and key vectors are computed and divided by $\sqrt{d_k}$, then a softmax function is applied on the result to obtain the weights for the value vectors. These vector operations can actually be executed simultaneously on a set of vectors with matrix multiplication by packing the queries, keys and value vectors together into matrices Q, K, V respectively. Thus, the attention function will take the following form.

$$\text{Attention } (Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right) V \tag{2.13}$$

The diagram on the left of Figure 2.8 visualizes the calculation of the scale dot-product attention. Extending Equation 2.12 into matrix form, the $Q, K \in \mathbb{R}^{d_{model} \times d_k}$ and $V \in \mathbb{R}^{d_{model} \times d_v}$ matrices can be computed from the set of input vectors $X \in \mathbb{R}^{d_{model} \times d_{model}}$ as follows.

$$\begin{aligned} Q &= XW_Q \\ K &= XW_K \\ V &= XW_V \end{aligned} \tag{2.14}$$

There is not just a single attention function per layer however. There are multiple different projections of the queries, keys and values. Multi-head attention is employed, which means that there are several attention computations in parallel. The d_{model}-dimensional query, key and value vectors are divided into h equally sized pieces by using h different, learned linear projection matrices, which project the input vectors to d_k, d_k and d_v dimensions, respectively. Using this method, instead of applying attention over the whole $d_{model} = 512$ dimensional inputs at once, the computation is divided into $h = 8$ heads that process $\frac{d_{model}}{h} = 64$ dimensional query, key and value vectors. The outputs of all heads are then concatenated and once again projected as shown below.

$$\text{MultiHead}(Q, K, V) = \text{ Concat } (\text{head}_1, \ldots, \text{ head }_{\text{h}})\, W^O \tag{2.15}$$

where

$$\text{head }_{\text{i}} = \text{ Attention } \left(QW_i^Q, KW_i^K, VW_i^V\right) \tag{2.16}$$

where the projection matrices are $W_i^Q \in \mathbb{R}^{d_{model} \times d_k}$, $W_i^K \in \mathbb{R}^{d_{model} \times d_k}$ and $W_i^V \in \mathbb{R}^{d_{model} \times d_v}$, and $d_k = d_v = d_{model}/h = 64$. Multi-head attention allows the model to jointly attend to information from different representation subspaces at different positions and also does not increase the total computational cost compared to the single-head attention with full dimensionality [3]. The block diagram on the right of Figure 2.8 demonstrates the information flow and transformations of a multi-head attention unit.

In every encoder and decoder layer there is another sub-layer, which is a fully connected feed-forward network applied on top of every attention sub-layer with two linear transformations and a ReLU activation function.

$$\text{FFN}(x) = \max\left(0, xW_1 + b_1\right) W_2 + b_2 \tag{2.17}$$

Layer Normalization. On top of each sub-layer, layer normalization [24] is employed to handle unbalanced data scales in training data by re-scaling and re-centering input features around a uniform value. By doing so, the network is made more stable and faster. Layer normalization is used here over the sum of the sub-layer output and the original input as follows.

$$Y = LayerNorm(X + Sublayer(X)) \tag{2.18}$$

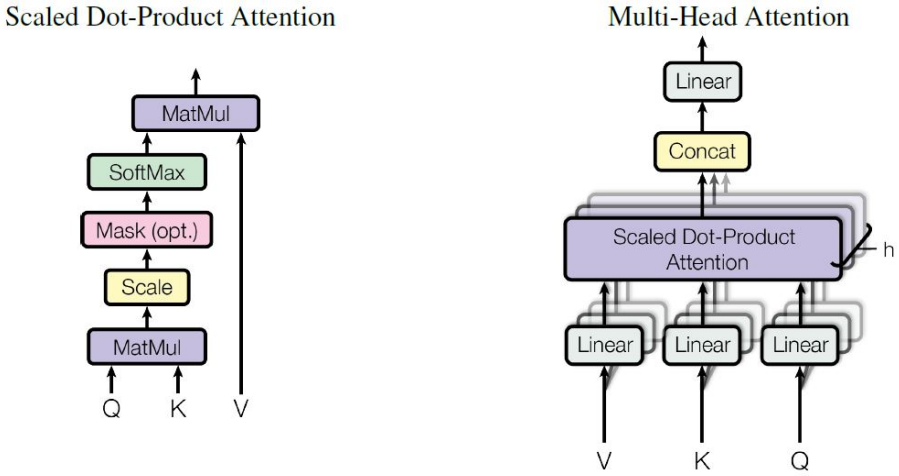

Figure 2.8: The Scaled Dot-Product Attention is shown on the left, and Multi-Head Attention consisting of several attention layers running in parallel is shown on the right [3].

Additionally, as a regularization technique, dropout [25] is applied to the output of each sub-layer before normalization and to the sums of word embeddings and positional embeddings in both the decoder and encoder stack. Dropout is a technique for addressing the problem of overfitting and slow propagation in large neural networks by reducing the number of parameters. The idea is to randomly drop connections and their weights from the neural network during training. In practice, this is achieved by a stochastic function that takes a weight matrix as input and replaces elements with 0 with a given probability. The dropout probability in the Transformer by default is $P_{drop} = 0.1$.

Positional Encoding. Since all elements of a sequence enters the network in parallel, positional encoding is needed before the first encoder layer to inject information about the relative or absolute position of words in the input sentence. This positional encoding has the same dimension as the word embedding layer, so they can be just simply summed. Positional encoding can be achieved in many ways; it can be absolute or relative, learned or fixed. In the Transformer, an absolute, fixed encoding is used, which is based on a sine and cosine functions with different frequencies.

$$\begin{aligned} PE_{(pos,2i)} &= \sin\left(pos/10000^{2i/d_{\text{model}}}\right) \\ PE_{(pos,2i+1)} &= \cos\left(pos/10000^{2i/d_{\text{model}}}\right) \end{aligned} \tag{2.19}$$

where $pos \in \{1, \ldots, d_{model}\}$ is the position of the word in the sequence and $i \in \{0, 1, \ldots, d_{model} - 1\}$ is the dimensional index for the embedding vectors. Figure 2.9 shows a visualization of positional encoding.

On the WMT-2014 English-to-French translation task, the Transformer model achieved a BLEU score of 41.0, outperforming all of the previously published single models, at less than 1/4 the training cost of the previous state-of-the-art model. It serves as a basis for today's top performing language models, such as XLNet,

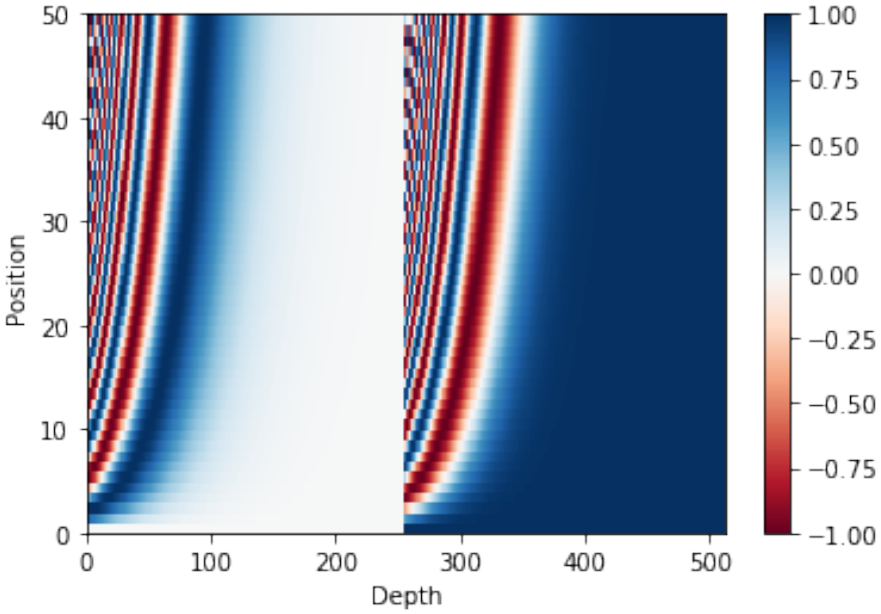

Figure 2.9: The 512-dimensional positional encoding for a sentence with a length of 50. Each row represents the embedding vector for a position.

GPT-3, and most importantly BERT, which will be discussed in further detail in Section 5.

Chapter 3

Concept

Based on the previous observations in Section 2.1, I outlined a basic concept of an AI-powered decision support system. The architecture consists of a Representation Engine that uses natural language processing (NLP) to extract digitally interpretable features from the user stories of the backlog and then feeds them to the Estimation Engine. The effort estimator predicts the story points for the backlog items, which then are used by the Planning Engine to formulate the next sprint, also taking inputs from the agile team into consideration. The system ideally would be integrated into an agile project management platform as a plugin.

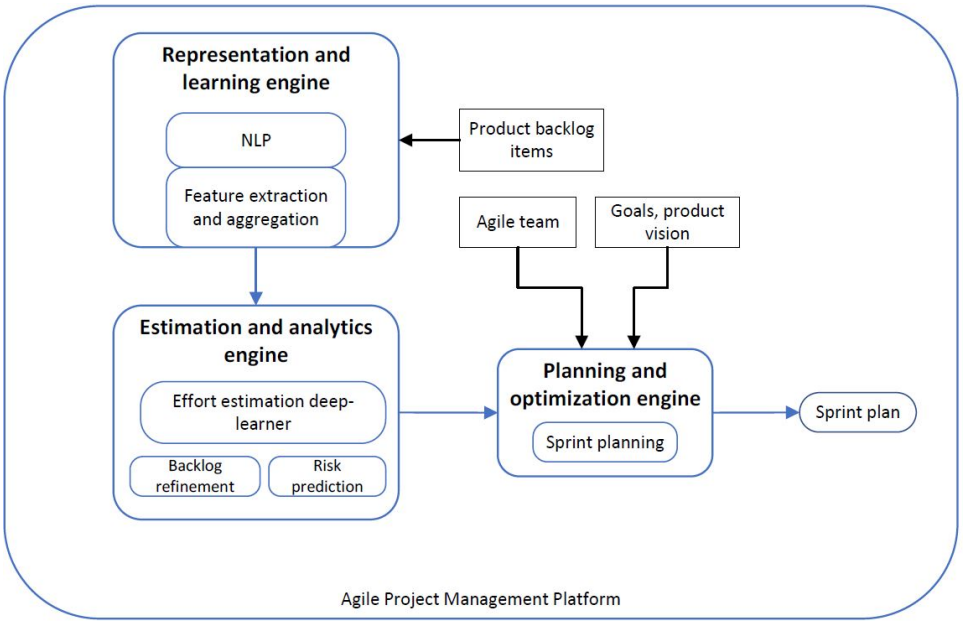

Figure 3.1: Architecture of the decision support system.

The Representation Engine processes the textual information and other attributes of the product backlog. Backlog items have numerous different attributes, such as the title and description of a user story, which are natural texts, but there are numeric properties as well, such as the priority or story point values. Occasionally, code fragments can also appear in an issue report. By NLP methods, meaningful feature

vector representation can be learned from the textual artifacts and code bases that can be used by other predictive models.

The Estimation Engine is responsible for providing useful predictive metrics such as effort estimates and performing other analytic functions such as backlog refinement or risk prediction. With machine learning methods, it is possible to build project specific prediction models from labeled historic project data, and they can learn the connections between the extracted features from the Representation Engine and the desired metric that is selected as the training label. The effort estimator learns from a team's previous relative effort used for older backlog items. It scans through the historic backlog data of the specific project and uses the story point values as training label.

Using the effort estimates and the prioritized backlog items, an optimization algorithm will suggest a sprint plan for the next development cycle also based on the team's sprint velocity (sprint size – the total effort committed for a sprint) and other inputs from the team.

Chapter 4

LSTM-based Estimation Model

The core of the outlined concept is the language representation and effort estimation engine, for which this thesis provides two distinct solutions that are later on evaluated and compared. The first approach to implement the estimator is an LSTM-based model, re-implementing the architecture described in a previous publication [11] that holds the latest state-of-the-art results for the specific task of story-point estimation as of writing this thesis. It is an end-to-end trainable language processing and effort estimation model that uses LSTM-based document representation and uses a Recurrent Highway Network and regression to produce the story-point estimates. As input, it takes the title and description fields of issue reports concatenated and converted into so-called word vectors. It outputs the relative effort required by the issue. This LSTM+RHWN model can learn to predict the story-points from raw text without the need for any manual feature engineering. The prediction error is propagated from the output node all the way back to the word layer. Figure 4.1 illustrates the architecture of the story-point estimation model.

4.1 Development Framework

With the rise of artificial intelligence, several open-source deep learning libraries have been developed, such as TensorFlow [26] from Google or PyTorch [27] from Facebook, to help developers implement neural architectures and to accelerate deployment and integration of machine learning models into their applications. These frameworks also often offer GPU and TPU support, which is essential for high-performance and large-scale solutions. Python became the number one programming language in the field of data science because of its flexibility, readability, modularity and the plethora of tools and modules that are available for it.

The implementation of the LSTM-based estimation model has been done in a Python-based framework, using several libraries, but most importantly Keras [28], which is an open-source Python library that is specifically designed for deep-learning applications and provides user friendly high-level APIs for artificial neural networks and deep learning architectures. It contains several implementations for commonly used neural layers, loss functions, activation functions, optimizers and other useful utilities. Presently, the exclusive backend for Keras (as of version 2.4) is TensorFlow,

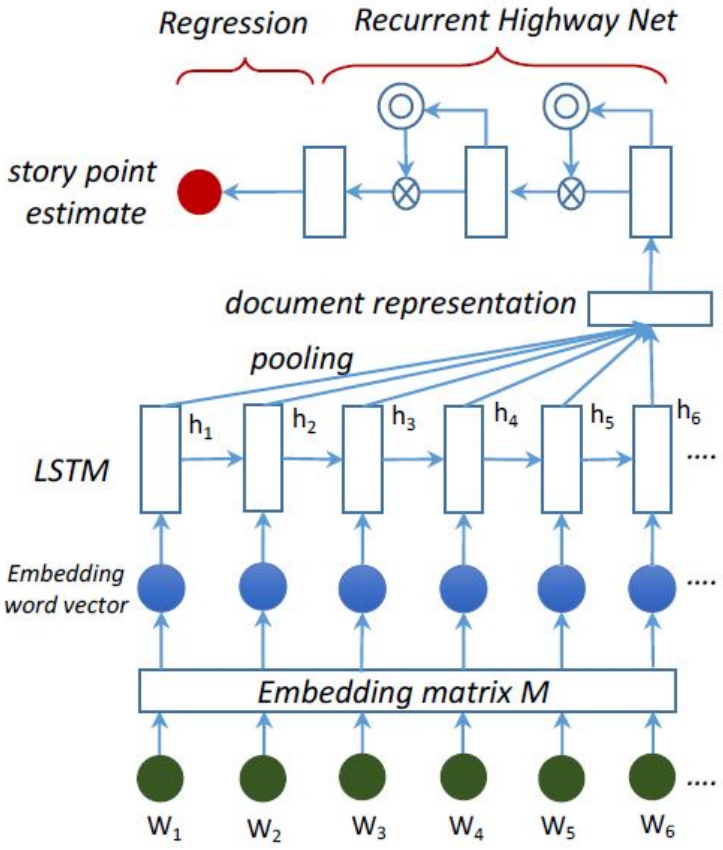

Figure 4.1: Effort estimation model [11].

which is essentially acts as the math engine behind Keras, doing the computational work, matrix calculations for the neural networks. Keras is basically a high-level interface for the TensorFlow library. However, since our model is based on a legacy code base from [11] that uses an older version of Keras, the backend for Keras is the Theano [29] library. There have been numerous other modules utilized as well, such as the NumPy [30] extension for array and matrix calculations, the pandas [31] library for data manipulations and handling the user-story datasets, or the scikit-learn [32] module for some network training utilities.

4.2 Word Embedding Layer

The first step of this approach is converting each word into fixed-length vectors ($word = [w_1, w_2 \dots w_d]$), thus performing the word-embedding, then feeding the word vectors as an input sequence to the LSTM layer which computes a vector representation for the whole document. For the word embedding, a look-up table or dictionary has to be created that stores a low-dimensional numerical vector representation for each word. This will be the so-called embedding matrix: $\boldsymbol{W_{emb}} \in \mathbb{R}^{d*V}$. The embedding dimension (d) is a tunable hyperparameter, and V is the vocabulary size. The embedding matrix has to be pre-trained by a large corpora of industry specific text. In Keras, the word embedding layer can be easily implemented by calling the *Embedding(output_dim = d, input_dim = vocab_size, input_length = inp_len,*

...) Layer class method, where we can pass the embedding size, the vocabulary size and sentence length, et cetera, as arguments.

4.3 Tokenization

Tokenization is the process of separating pieces of raw text, a string of input characters by a certain delimiter into smaller units (tokens) and assigning a unique ID, a simple integer to them so that a NLP algorithm can further process the text. Tokenization is a fundemental preprocessing step in natural language processing that breaks down the raw input text in a way that machines can understand. It needs to be done before the word embedding. The process results in a vocabulary or dictionary that holds all of the ID integers for their corresponding tokens, and these integers are then used by the word embedding layer.

Based on the delimiting rule, the tokens can be words, sub-words or characters. The most commonly used tokenization method is word tokenization, which splits the input string into individual words based on certain delimiters, such as white space, comma, dash, et cetera. As a result, a vocabulary is created that contains the unique IDs for every word in the training corpus. This is the method that was used in our LSTM estimation model.

The major drawback of word tokenization is the occurrence of out-of-vocabulary words during testing, which is inevitable due to the limited size of a vocabulary. These are new words that were not present in the training corpus, and therefore the vocabulary would fail to translate them into IDs. This problem is resolved by the so-called unknown token (UNK), which is used to represent any out-of-vocabulary word. A tokenization Perl script from the open-source machine translation package Moses [1] was used in the LSTM-based model. Inside the preprocessing Python script, the Perl script is invoked by the *Popen* class from the *subprocess* Python library.

4.4 LSTM Layer

As an encoder layer, an LSTM layer is used to produce the context vectors from the input sequences. The concatenated text from the title and description field of an issue report is fed into the LSTM layer token by token after being transformed by the word embedding matrix. Then the hidden state of the cell is used to produce the output vectors at each time step as the cell is trying to predict next element in the sequence. The input text from the issue reports can be arbitrarily long since the LSTM processes the input token by token until the end of the sequence. Because of the variable length of the input sequences, the output vectors from the LSTM have to be aggregated by a length invariant pooling to represent the whole text of the issue report by a single fixed-length vector. Mean-pooling is used for this process, but other types pooling, such as max-pooling or min-pooling were also tried. The produced document vector serves as input for the Recurrent Highway

[1] https://github.com/moses-smt/mosesdecoder

Network. Figure 4.2 shows the unfolding of an LSTM layer as it is trying to predict the next element of the sequence.

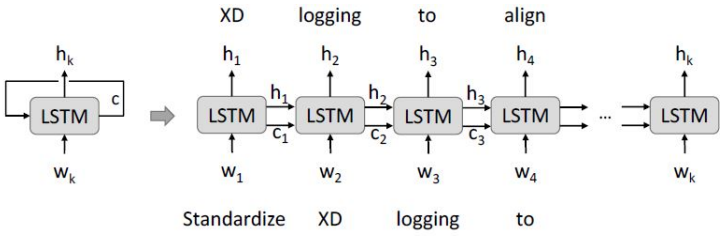

Figure 4.2: LSTM layer [11]

In Keras, an LSTM layer can be easily created by calling the *LSTM(input_dim, output_dim, dropout_U, dropout_W, ...)* Layer class method, specifying the input and output dimension, optional dropout probabilities, et cetera. To prevent overfitting, a dropout rate of 0.2 was set. For pooling the LSTM output sequence, on top of the LSTM layer a custom *PoolingSeq(Layer)* class had to be created, which is a child of the Layer class from keras.layers.

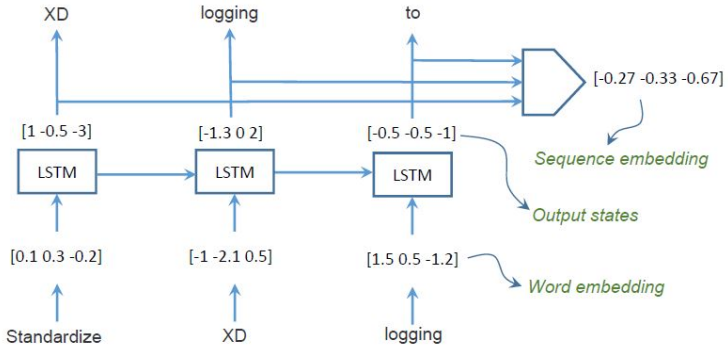

Figure 4.3: Pooling the output vectors [11]

4.5 Recurrent Highway Network

A multi-layer recurrent net is used to transform the document vector several times to acquire a deeper representation of the document. Linear passes, also known as Highways, are set up to help the gradients pass through to avoid vanishing and exploding gradients and to reduce parameters, and thus creating a Recurrent Highway Network (RHWN). Highway layers inside the recurrent transition is a superior method of increasing depth that enables the use of substantially more powerful and trainable sequential models efficiently, significantly outperforming traditional RNNs [12]. Figure 4.4 shows the architecture of an RHWN.

A Highway layer computation can be defined as

$$y = H\left(x, W_H\right) \cdot T\left(x, W_T\right) + x \cdot C\left(x, W_C\right) \tag{4.1}$$

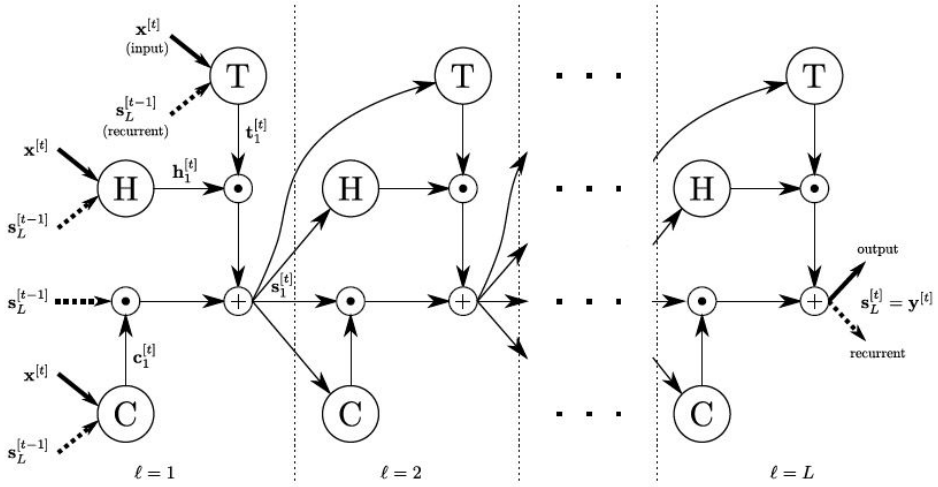

Figure 4.4: Recurrent Highway Network [12]

where $T(x, W_T)$ and $C(x, W_C)$ are the transform and carry gate with sigmoid activations, and $H(x, W_H)$ is nonlinear transformation with *tanh* activation and the products are taken element-wise [33].

In a standard recurrent net a state transition can be described as $\mathbf{y}^{[t]} = f\left(\mathbf{W}\mathbf{x}^{[t]} + \mathbf{R}\mathbf{y}^{[t-1]} + \mathbf{b}\right)$. Consequently, if s_l is an intermediate output at layer l with $\mathbf{s}_0^{[t]} = \mathbf{y}^{[t-1]}$, the hidden state of an intermediate layer in a Recurrent Highway Network can be defined as

$$\mathbf{s}_\ell^{[t]} = \mathbf{h}_\ell^{[t]} \cdot \mathbf{t}_\ell^{[t]} + \mathbf{s}_{\ell-1}^{[t]} \cdot \mathbf{c}_\ell^{[t]} \tag{4.2}$$

where

$$\mathbf{h}_\ell^{[t]} = \tanh\left(\mathbf{W}_H \mathbf{x}^{[t]} \mathbf{I}_{\{\ell=1\}} + \mathbf{R}_{H_\ell} \mathbf{s}_{\ell-1}^{[t]} + \mathbf{b}_{H_\ell}\right) \tag{4.3}$$

$$\mathbf{t}_\ell^{[t]} = \sigma\left(\mathbf{W}_T \mathbf{x}^{[t]} \mathbf{I}_{\{\ell=1\}} + \mathbf{R}_{T_\ell} \mathbf{s}_{\ell-1}^{[t]} + \mathbf{b}_{T_\ell}\right) \tag{4.4}$$

$$\mathbf{c}_\ell^{[t]} = \sigma\left(\mathbf{W}_C \mathbf{x}^{[t]} \mathbb{I}_{\{\ell=1\}} + \mathbf{R}_{C_\ell} \mathbf{s}_{\ell-1}^{[t]} + \mathbf{b}_{C_\ell}\right). \tag{4.5}$$

With the carry gate being defined as $C = 1 - T$ for simplicity, the intermediate layer can be written as follows.

$$\mathbf{s}_\ell^{[t]} = \mathbf{h}_\ell^{[t]} \cdot \mathbf{t}_\ell^{[t]} + \mathbf{s}_{\ell-1}^{[t]} \cdot (1 - \mathbf{t}_\ell^{[t]}) \tag{4.6}$$

From this equation, we can see that if $\mathbf{t}_\ell = 0$ is chosen, we are creating a clear linear pass by simply just copying the state. The output of this network is fed into a regressor (a feedforward net) which produces the story points. The number of hidden layers is a hyper-parameter that has to be tuned. In Keras, a recurrent highway layer can be easily created by calling the *Highway(activation = 'relu', init = 'glorot_normal', transform_bias = -1)* Layer class method, specifying the

activation function, biases and the initializer. A vanilla feed-forward layer can be instantiated by calling *Dense(output_dim = out_dim, activation = 'relu', init = 'glorot_normal')*, specifying the activation function, the output dimensions and the initializer.

To chain all the above mentioned layers together to form the desired topology, the Functional API from Keras can be used in a way to connect the next layer to its below layer with the *layer call* method as follows.

$$next\_layer = Layer(args)(below\_layer) \tag{4.7}$$

Then, with the *keras.Model* class, we can pack this layer chain into a model object with training and inference features. Therefore, the simplified algorithm to create the architecture of the estimation model would be:

Algorithm 1 Pseudo code of LSTM-based model build-up

```
def create_model(vocab_size, inp_len, emb_dim)
    input = Input(shape = (inp_len,) )
    embedding = Embedding(emb_dim, vocab_size, inp_len)
    embedding = embedding (input)
    lstm = LSTM (emb_dim, emb_dim) (embedding)
    pooled = PoolingSeq(poolmode = mean) (lstm)
    highway = Highway('relu') (pooled)
    top = Dense(out_dim = 1, 'relu') (highway)
    model = Model(input=[title, description,], output=top)
return model
```

Once the model is instantiated, it can be configured with losses and metrics with model.compile(), can be trained by model.fit(), or can perform a forward pass to make a prediction with model.predict().

4.6 Training

The training of the model is done in 2 steps. First, the word embedding matrix has to be trained by pre-training using a large corpus of sector or industry specific text. An unlabeled issue dataset containing more than 50000 data rows is used for pre-training for each repository. Then in the next step, the LSTM+Highway estimation model has to be trained for a specified project by previous issue reports with their training labels – the corresponding story points – and by using the word embedding matrix.

4.6.1 Pre-training

Pre-training is the process of creating the word embeddings and the initial weights of the LSTM by using unlabelled text from a large number of issue reports without

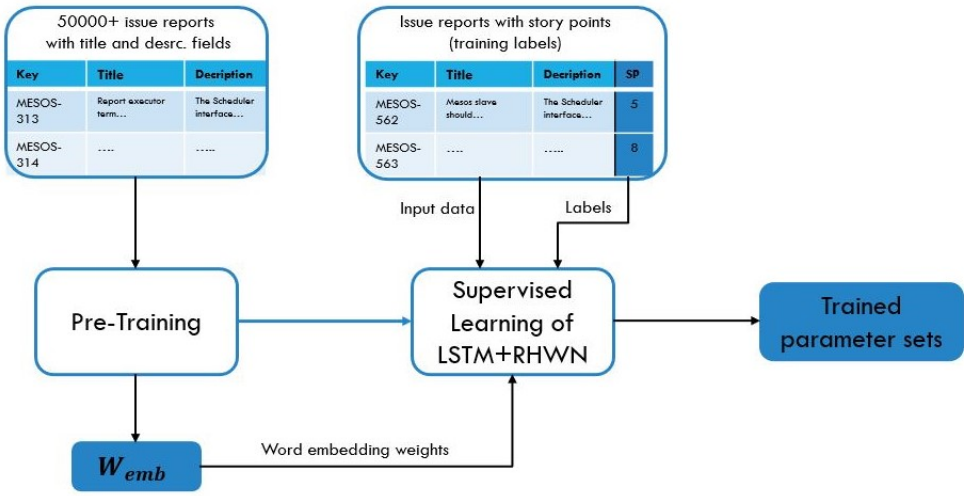

Figure 4.5: Training of the estimation model.

story-points. Pre-training leverages the predictability of natural language, and it is done through language modeling, by determining the probability distribution for the next word in a sequence based on the previous words as described in Section 2.3.2. Since this process is a classification problem, the language model is learned by minimizing the binary cross-entropy loss L, also known as log-loss.

$$L\left(\boldsymbol{y}_i, \hat{\boldsymbol{y}}_i\right) = -\boldsymbol{y}_i \log\left(\hat{\boldsymbol{y}}_i\right) \tag{4.8}$$

Using the loss, during back-propagation, the weights are adjusted by mini-batch gradient descent. Mini-batch gradient means that, instead of adjusting the weights after every training sample, the training samples are grouped together into small batches, and after calculating the losses and their gradients for a batch, we take the mean of those gradients and adjust the parameters with that. Modifying the weights after a batch containing $n > 1$ number of samples saves some computational cost compared to a modification after every single sample. If θ is our parameter set that needs to be optimized, and $L_i = L(\theta, \boldsymbol{y}_i, \hat{\boldsymbol{y}}_i)$ is the log-loss for the i^{th} sample, and we group n training sample into a batch, then the weights are computed as

$$\begin{aligned} \theta &= \theta - \eta \cdot \nabla_\theta L\left(\theta, y_{(i:i+n)}, \hat{y}_{(i:i+n)}\right) \\ \theta &= \theta - \frac{\eta}{n} \sum_{i=1}^{n} \nabla_\theta L_i \end{aligned} \tag{4.9}$$

where $\eta > 0$ is the learning rate, and $\nabla_\theta L_i$ is the gradient of the i^{th} sample over the θ parameter set. For example, for a particular θ_j weight, the gradient would be the following partial differential.

$$\nabla_{\theta j} L_i = \frac{\partial L_i}{\partial \theta_j} \tag{4.10}$$

Since the probability distributions are calculated over the entire vocabulary, the computational cost for each time step is $O(V)$, where V is the size of the vocabulary,

which can be in the hundreds of thousands for a large corpus. Therefore, to reduce this potentially huge computational cost, an approximate but very fast alternative was implemented based on Noise-Contrastive Estimation [34], which reduces the time to $O(M) << O(V)$, where M can be as small as 100.

The pre-training is done with approximately 50000 unlabeled issue reports for each repository shown in Table 4.3. These data tables contain three columns: the issue key, the title and the description of the issue. The latter two are concatenated into one sequence to form the textual inputs for pre-training with an average length of around 100 words per sample. This results in approximately 5 million data points per repository for pre-training. The batch size is chosen to be $n = 50$. The learning rate is configured to be $\eta = 0.01$. After configuring the hyper-parameters and compiling the model, the pre-training can be started by calling *model.fit()*.

4.6.2 Fine-tuning the LSTM

For our specific story-point estimation problem, the model has to be fine-tuned with labeled issue report datasets (Table 4.1) for each project. Since the LSTM-based model is performing a regression, the applied loss function is mean squared error (MSE):

$$\text{MSE} = \frac{1}{m}\sum_{i=1}^{m}(y_i - \hat{y}_i)^2 \tag{4.11}$$

where m is the dimensionality of the output vector, which is 1 in this case as the output of the network is a single scalar value of the story-point prediction. Mini-batch gradient descent is used to converge to the minimal loss in this case too. The applied datasets were split into training, validation and test sets at ratio of 60/20/20 respectively. Before training the model, we have to load the already pre-trained word embedding weights and the initialization weights for the LSTM layer. The training process is organized into epochs, iterations over the dataset. After each epoch, the model is evaluated on the validation set to optionally modify the learning rate or to decide when to terminate the training. The early stopping mechanism was adopted to end the training process if the loss does not improve for five consecutive epochs. The model is saved with the best performing parameter set and evaluated on the test dataset also.

4.7 Datasets

To train the model, issues that were estimated with story-points had to be collected from publicly available sources. Jira from Atlassian is a widely used agile project management platform that is also used for the development of several open-sorce software solutions. From the Jira issue tracking system [35], I exported a wide variety of issue report datasets from 15 different open-source projects which also served as a data source for the original implementation of the LSTM estimation model by Choetkiertikul et al [11]. These projects cover a versatile set of domains and include namely the Apache Mesos cluster manager, the Usergrid Backend-as-a-Service solution, the Appcelerator Studio and Titanium SDK mobile app development plat-

forms, the Aptana Studio web development platform, the DuraCloud storage solution, the Bamboo integration and release management platform, the Clover code coverage analysis tool, and also the Jira platform itself, LSST's Data Management tool, the Mule and Mule Studio integration platforms for SaaS, the Spring XD data analytics tool, the Talend Data Quality and Talend ESB data integration tools. These product backlogs are from 8 different repositories, which are Apache, Appcelerator, DuraSpace, Atlassian, LSST Corp, Mulsesoft, Spring and Talendforge. Table 4.1 and Table 4.2 summarize the statistics of the exported datasets such as minimum, maximum and median value and standard deviation of the story-points, and also demonstrates which repository a project belongs to.

Repository	Project	No. issues	SP Range	Median SP
Apache	Mesos	1680	1 - 40	3
	Usergrid	482	1 - 8	3
Appcelerator	Appcelerator Studio	2919	1 - 40	5
	Aptana Studio	829	1 - 40	8
	Titanium SDK	2251	1 - 34	5
DuraSpace	DuraCloud	666	1 - 16	1
Atlassian	Bamboo	521	1 - 20	2
	Clover	384	1 - 40	2
	JIRA	352	1 - 20	3
LSST Corp	DataMangement	4667	1 - 100	4
Mulesoft	Mule	889	1 - 21	5
	Mule Studio	732	1 - 34	5
Spring	Sping XD	3526	1 - 40	3
Talendforge	Talend Data Quality	1381	1 - 40	5
	Talend ESB	868	1 - 13	2

Table 4.1: Issue report datasets for training.

The number of issue reports in the entire dataset is adding up to 22147 overall. Only items that had been completed, i.e. had a Status field of *Closed*, *Done* or similar and had their Resolution field set to *Resolved* or *Fixed*, were exported. Issues that were given a zero or negative story point were filtered out. These data tables contain four columns: the issue key, the title and the description of the issue and their corresponding story points as labels.

For pre-training, sets of unlabeled issue reports had to be collected for all the 8 different repositories. These data tables only have the issue key, title and description columns, and the tables contain 10000-50000 records. Table 4.3 presents the statistics of the exported pre-training datasets. The reason for having only 8 pre-trained models for the 15 product backlog is that the projects that are under the same repository will use the same pre-trained word embeddings. The intuition behind this is that the applications that are developed under the same repository are generally serving the same sector or field and have similar vocabulary, use similar terminology.

Project	No. issues	Median SP	Std SP	Mean Length
Mesos	1680	3	2.42	181.12
Usergrid	482	3	1.4	108.6
Appcelerator Studio	2919	5	3.33	124.61
Aptana Studio	829	8	5.95	124.61
Titanium SDK	2251	5	5.10	205.9
DuraCloud	666	1	2.03	70.91
Bamboo	521	2	2.14	133.2
Clover	384	2	6.55	124.48
JIRA	352	3	3.51	114.57
DataMangement	4667	4	16.61	69.41
Mule	889	5	3.50	81.16
Mule Studio	732	5	5.39	70.99
Sping XD	3526	3	3.23	78.47
Talend Data Quality	1381	5	5.19	104.86
Talend ESB	868	2	1.5	128.97

Table 4.2: Training dataset statistics.

Repository	Number of issues	Mean Length
Apache	49802	111.42
Appcelerator	32789	93.23
DuraSpace	9807	67.53
Atlassian	49975	123.12
LSST Corp	24530	89.31
Mulesoft	9516	123.32
Spring	48493	98.47
Talendforge	49813	113.66

Table 4.3: Pre-training datasets.

4.8 Evaluation

4.8.1 Performance Metrics

The story-point estimation models were evaluated using different performance metrics such as Mean Absolute Error (MAE), Mean Magnitude of Relative Error (MMRE) and an arbitrary precision metric (Pre25) that just shows the number of predictions that had a relative error lower than 25%. These metrics are defined as:

$$MAE = \frac{1}{N} \cdot \sum_{i=1}^{N} \mid \text{TrueSP}_i - \text{PredSP}_i \mid \tag{4.12}$$

$$MMRE = \frac{1}{N} \cdot \sum_{i=1}^{N} \frac{\mid \text{TrueSP}_i - \text{PredSP}_i \mid}{\text{TrueSP}_i} \tag{4.13}$$

$$Pre25 = 100\% \cdot \frac{\#\left(MRE_i < 0.25\right)}{N} \tag{4.14}$$

where $PredSP$ is the story-point prediction of the model, $TrueSP$ is the actual story-point value, and N is the number of issues used to test a model. Averaging the absolute difference between the true value and the estimated value over the entire test set will result in the MAE. If we normalize the absolute prediction error with the true value, we gain the magnitude of relative error (MRE), and averaging it over the entire test set will give us the MMRE.

4.8.2 Hyper-parameter Settings

There are two important hyper-parameters that define the LSTM-based architecture: the word embedding dimension and the number of hidden layers in the recurrent highway layer. To tune these hyper-parameters, I ran story-point estimation experiments on the Appcelerator Studio dataset by fixing one hyper-parameter and varying the other, and I evaluated the mean absolute error (MAE) of the trained models with the validation set. The test were run for four different embedding sizes (10, 25, 50, 100) with nine different hidden layer counts (2, 3, 5, 10, 20, 30, 50, 75, 100). Figure 4.6 shows the results of the experiments. An embedding dimension of 50 with a hidden layer count of 10 results in the lowest mean absolute error, which is 1.31. However, if the number of hidden layers is 10, setting the embedding dimension to only 10 will also give a sub-optimal result of 1.32 MAE, which is just slightly worse than the 50-dimensional embeddings. Since a 10-dimensional embedding takes significantly less computational cost to pre-train and to train and provides almost equally good results, a hyper-parameter setting where the embedding dimension is 10 and the number of hidden layers is also 10 was selected.

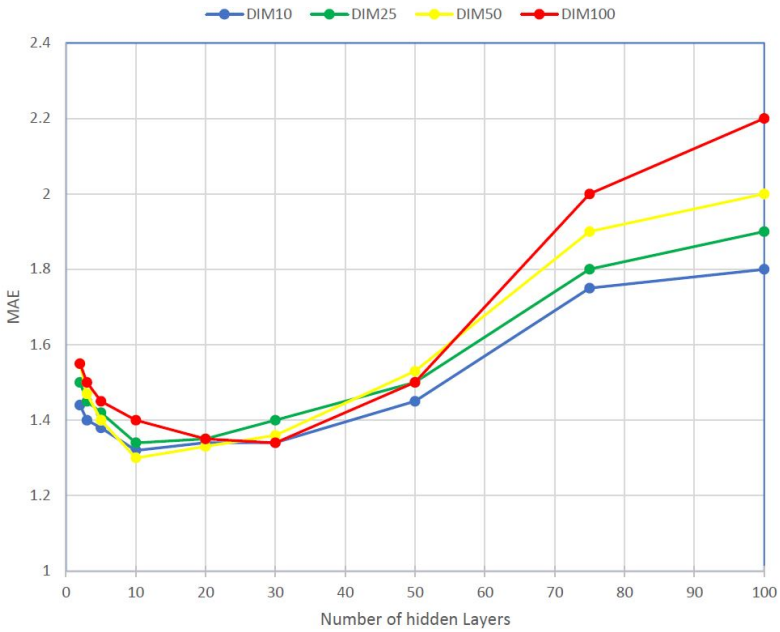

Figure 4.6: Effects of different hyper-parameter settings on mean absolute error during story-point estimation on the Appcelerator Studio dataset.

4.8.3 Results

After setting up the hyper-parameters, the models were pre-trained. Using the pre-trained embeddings, the models were fine-tuned for the story-point estimation task on the 15 different backlogs and evaluated on their test sets with the previously mentioned performance measures. The learning rate was set to $\eta = 0.01$, and a batch size of 100 was selected. The dropout rates for the LSTM and RHWN layer were set to 0.2 and 0.5 respectively.

Figure 4.7 and Figure 4.8 shows the learning curves for all the 15 models during fine-tuning. The validation losses are marked for every finished epoch. As can be seen, all of the models converge to a minimal loss, and most of the models reach their optimum after 10 epochs, but the early stopping criteria was not sensitive enough, and so some of the models were trained for a bit longer than needed. The training times varied between 8 and 30 minutes with an average of 14 minutes while running the models on a personal computer with a 3.6GHz Intel Core i7 quad-core CPU and 8GB RAM. The GPU was not utilized for the training of the LSTM-based estimation models. Pre-training took of course significantly more time: around five hours for each repository.

I compared the results to a baseline benchmark called Mean Effort, which essentially just returns the mean value of the training label set for each prediction. I also compared the results to an older method that was used for this same task in a previous research paper, namely the Term Frequency - Inverse Document Frequency method [9]. Table 4.4 summarizes the test results. Based on relative error, the LSTM-based model performed the best in the case of the Appcelerator Studio dataset, which had a MMRE of 0.3477 (a mean relative error of 34.77%), and the worst in the case of the Data Management dataset with a MMRE of 1.0607. The average MMRE of

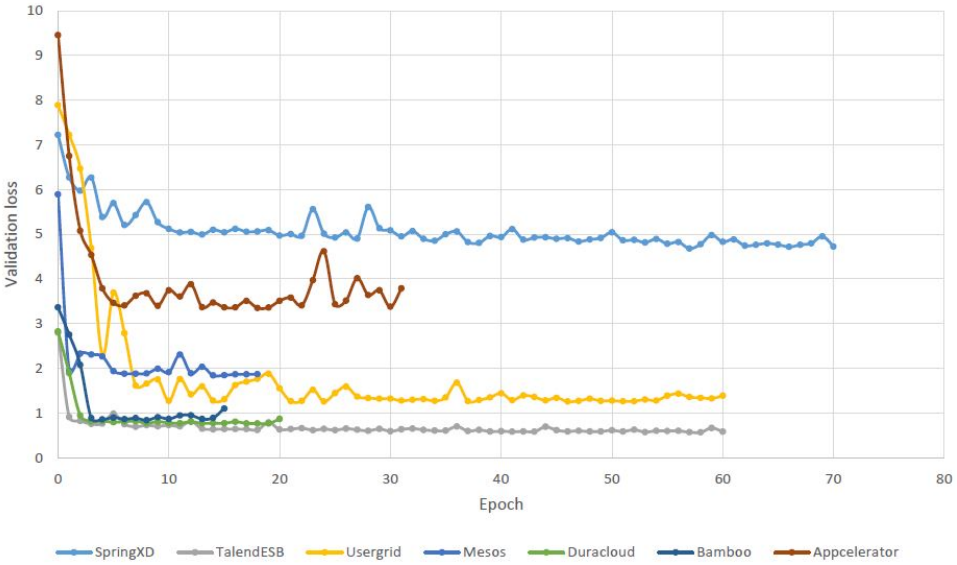

Figure 4.7: Learning curves of the estimation models for the projects SpringXD, TalendESB, Usergrid, Mesos, Duracloud, Bamboo and Appcelerator Studio.

the LSTM-based estimation model is 0.6621. The performance of the architecture varies significantly across the different projects, which can be caused by the different size and quality of the datasets. A low number of training samples, for example, could restrict the learning abilities of the model. However, if we were to examine the number of issues in the datasets and the performance of the corresponding models, we would find little to no correlation. Rather, the varied performance of the models can be attributed to the credibility and consistency of the true story-point values in the datasets, meaning that if there is no connection between the descriptions of the issues and their assigned story-points, if the story-points are assigned randomly or inconsistently in a backlog, then a neural network will not be able to learn dependencies between the textual information and the story-point labels of a dataset either, and it will not make good estimates.

With that being said, as it can be seen from the learning curves too, the LSTM-based estimation model was able to learn connections between the issue descriptions and their labels, other than learning the mean of the labels, in all of the cases since it significantly outperformed the mean effort baseline benchmark in every dataset. A study done by N.C. Augen [10] found that the human relative error using planning poker estimation method is around 0.47. In 7 out of the 15 cases (Mesos, Appcelerator, Apatana, DuraCloud, Bamboo, Jira and Talend ESB), the LSTM model has provided encouraging results with a similar or lower MMRE than the aforementioned human predictions. However, overall, it has higher relative error than the expert-based method.

$$\begin{aligned} MMRE_{LSTM} &= 0.6621 \\ MMRE_{human} &= 0.47 \\ MAE_{LSTM} &< MAE_{benchmark} \end{aligned} \tag{4.15}$$

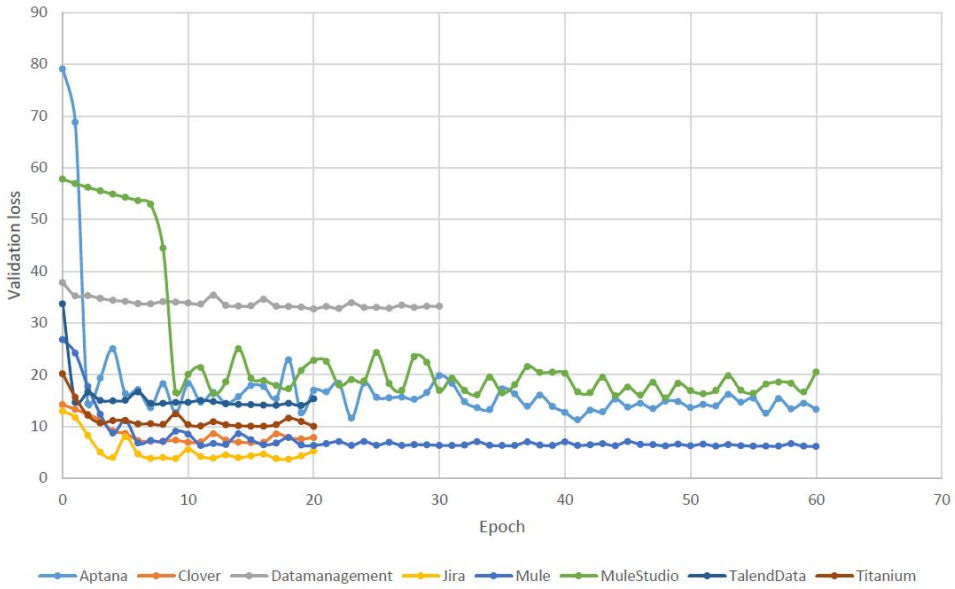

Figure 4.8: Learning curves of the estimation models for the projects Aptana Studio, Clover, Datamanagement, Jira, Mule, Mule Studio, Talend Data and Titanium.

	LSTM-based Model			Mean Effort	TF-IDF
Project	MAE [-]	MMRE [-]	Pre25 [%]	MAE [-]	MAE [-]
Mesos	1.0819	0.4643	33.5312	1.64	1.23
Usergrid	1.0383	0.5671	40.8163	1.48	
Appc. Studio	1.3233	0.3477	50.5119	2.08	1.28
Aptana Studio	2.7340	0.4105	36.9697	3.15	5.69
Titanium SDK	2.0640	0.6323	39.3805	3.05	
DuraCloud	0.6708	0.4424	34.8148	1.30	
Bamboo	0.7641	0.5256	33.9623	1.75	
Clover	2.3115	0.7772	25.3165	3.49	
JIRA	1.4654	0.5072	31.9444	2.48	
DataM	4.3338	1.0607	18.1818	5.29	
Mule	2.0368	0.9717	31.6384	2.59	3.37
Mule Studio	3.0857	0.8730	33.1081	3.34	
Sping XD	1.7800	0.9156	23.3380	2.27	1.86
Talend DataQ	2.9216	0.9465	29.7101	4.81	
Talend ESB	0.6978	0.4908	40.3409	1.14	

Table 4.4: Evaluation of results.

Chapter 5

BERT-based Estimation Model

The second solution presented for the story-point estimator is based on a state-of-the-art language representation model called BERT, that is, Bidirectional Encoder Representations from Transformers proposed by Devlin et al. from Google Research in 2018 [4]. The release of BERT marks a milestone for the machine learning community as it broke numerous records on NLP tasks such as question answering (SQuAD v1.1, SQuAD v2.0 [36]), natural language inference or general language understanding (GLUE [37]), and it provides a powerful out-of-the-box tool for developers who are building machine learning models that involve natural language processing. The outstanding performance of BERT and its practicality is a result of a paradigm shift in NLP to transfer learning, which allows for quicker development of large scale machine learning applications with minimal task-specific fine-tuning of massive pre-trained models.

5.1 BERT

The BERT architecture is derived from the so-called Transformer [3] architecture, an attention-based neural machine translation model, and it is designed to be pre-trained bidirectionally (both left and right context in all layers) by a large corpus of unlabeled text. The pre-trained model then can be fine-tuned with one additional layer for a broad range of classification, regression problems without significantly altering the baseline architecture. There are various different already pre-trained BERT models available in the public domain for us to choose from and fine-tune for our specific application [19]. In this section, I discuss how the creators of BERT built up and pre-trained their original model.

5.1.1 BERT Architecture

The basis of the BERT architecture is the encoder stack of the Transformer, which was discussed in Section 2.3.5. The standard $\text{BERT}_{\text{BASE}}$ model consists of 12 Transformer encoder layers, 12 self-attention heads per layer with a hidden size of 768 and a total number of 110 million parameters, while the larger $\text{BERT}_{\text{LARGE}}$ has 24

encoder layers, 16 self-attention heads per layer with a hidden size of 1024 and a total number of 340 million parameters. Figure 5.1 illustrates the BERT architecture. It also, of course, has an input embedding layer that includes token embedding and position encoding inherited from the Transformer, but what is unique about BERT's input embedding is that it also has a segment encoding layer that enables model to handle sentence pairs (question and answer, for example) in one token sequence, which is particularly useful in question answering or natural language inference tasks. The input sequence is represented by the sum of the token, segment and position embeddings. This input embedding layer is able to unambiguously represent both a single sentence and a pair of sentences. From our story-point estimation task's point of view, it is important to note that a "sentence" here does not necessarily mean a linguistic sentence, but rather a contiguous text of any length below the maximum length. For example, it can be a description of an issue report from our dataset containing multiple linguistic sentences.

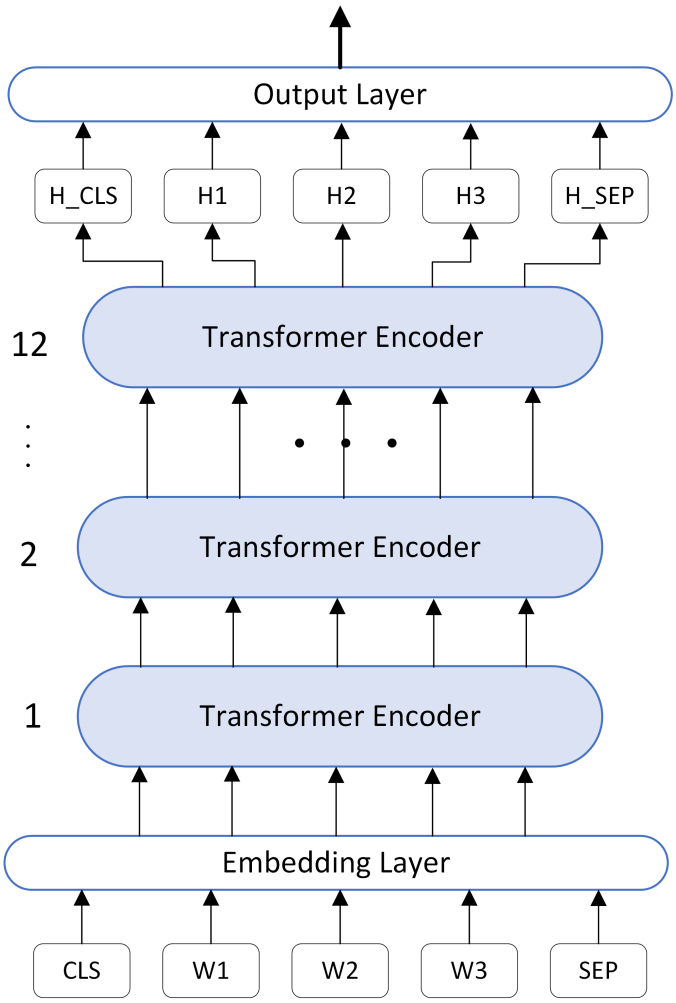

Figure 5.1: The layer stack of BERT.

For tokenization, WordPiece embedding [38] is used, which has a vocabulary of 30000 tokens, but it contains tokens for words, sub-words and individual characters as well, so it can tokenize every element of an input. A special [CLS] classification token is put at the beginning of every sequence. The final hidden state (H_{CLS})

corresponding to this token is used as the aggregate sequence representation for classification tasks. In case of a sentence pair input, the two sentences are separated by a special [SEP] separator token, which is placed at the end of both sentences. It is important that even in one-sentence tasks a [SEP] has to be placed at end of the sentence. The maximum length of an input sequence is $d_{model} = 512$ just as in the Transformer. Figure 5.2 demonstrates the input representation.

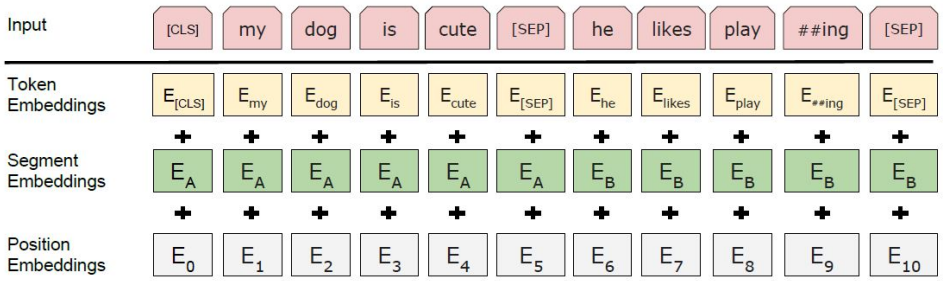

Figure 5.2: Input representation of BERT [4].

There is an additional feed-forward layer put on top of the encoder layers which is task specific, usually a simple fully-connected neural layer, and is trained during fine-tuning. It could be, for example, a single dense layer with a softmax output for classification tasks.

5.1.2 Pre-training BERT

Since an input sequence is fed into the network at the same time in parallel instead of sequentially left-to-right or right-to-left, the model is considered to be non-directional or bidirectional, as the authors would call it. This characteristic allows the model to represent words based on all of their surroundings (left and right context), but this also means that the traditional next-word prediction language model does not work here for pre-training. To overcome this problem, the authors of BERT proposed a new method of using two unsupervised tasks to pre-train the model, namely a masked language model (Masked LM) and a next sentence prediction task (NSP).

The Masked LM is an effective way to train non-directional representations by randomly masking a certain percentage (15% in this case) of the input tokens and then trying to predict those masked tokens with a classification output layer. To facilitate this task, the final output vectors from the BERT encoder stack are fed into a simple feed-forward layer with a softmax output over the vocabulary that produces the probabilities for the original values of the masked words. In the feed-forward layer the final hidden states are multiplied by the embedding matrix, and thus, converted back into vocabulary dimension, and then the softmax output function calculates the probability distribution over the vocabulary. The loss is only computed for the masked tokens, which is then back-propagated to optimize the weights. As a consequence of ignoring the outputs for non-masked tokens, which is 85% of the output, the convergence of the model is slower compared to directional models, but it results in a higher contextual understanding. Figure 5.3 illustrates the Masked LM method.

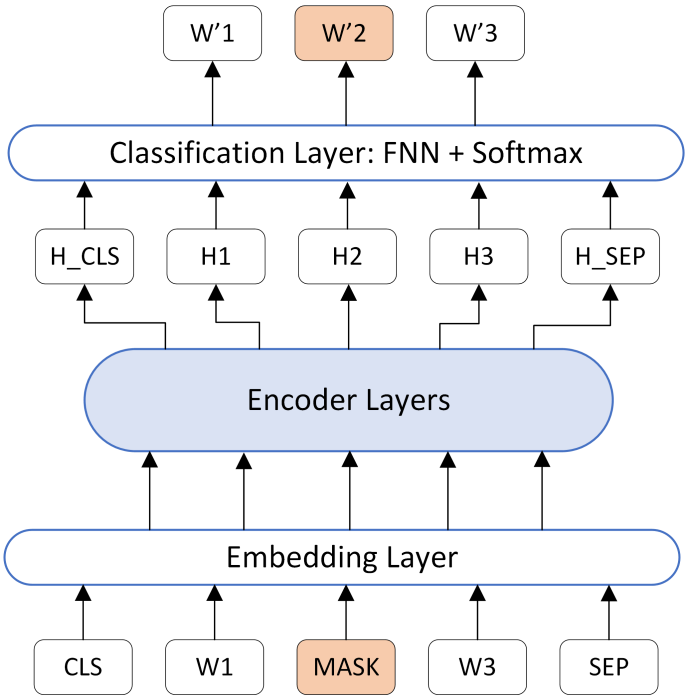

Figure 5.3: Masked language modeling with BERT.

For the model to understand relationships between sentences, it is also pre-trained on a next-sentence prediction task, which is implemented by selecting A and B sentence pairs from the training corpus in a way that 50% of the time sentence B actually follows A and 50% of the time it is a random sentence from the corpus that does not follow A. The training label for the sentence pairs is a simple binary value annotating if B is the next sentence after A, and the C_{CLS} output is used to predict this label by binary classification. During pre-training BERT, the masked language modeling and the next-sentence prediction is performed together while trying to minimize the combined loss of the two classification problem. Figure 5.4 shows the overall pre-training process with the Masked LM and NSP method combined.

The original English $\text{BERT}_{\text{BASE}}$ and $\text{BERT}_{\text{LARGE}}$ models were pre-trained with the BooksCorpus [39] dataset containing 800 million words and with the text passages of the English Wikipedia (roughly 2500 million words). There are now models for as much as 104 languages. Pre-training BERT is computationally expensive. It took 4 days to pre-train the base model on 4 cloud TPUs and also 4 days to pre-train the large model on 16 TPUs, but this results in a deep bidirectional architecture with enhanced language understanding capabilities that allows the same pre-trained model to be effectively used in a broad set of NLP applications. The BERT models have outperformed their directional transfer learning predecessors such as the OpenAI GPT [40] and ELMo [41] models on multiple benchmarks.

5.1.3 GLUE Benchmark

The General Language Understanding Evaluation (GLUE) benchmark [37] is a collection of nine natural language understanding tasks that is used to measure the

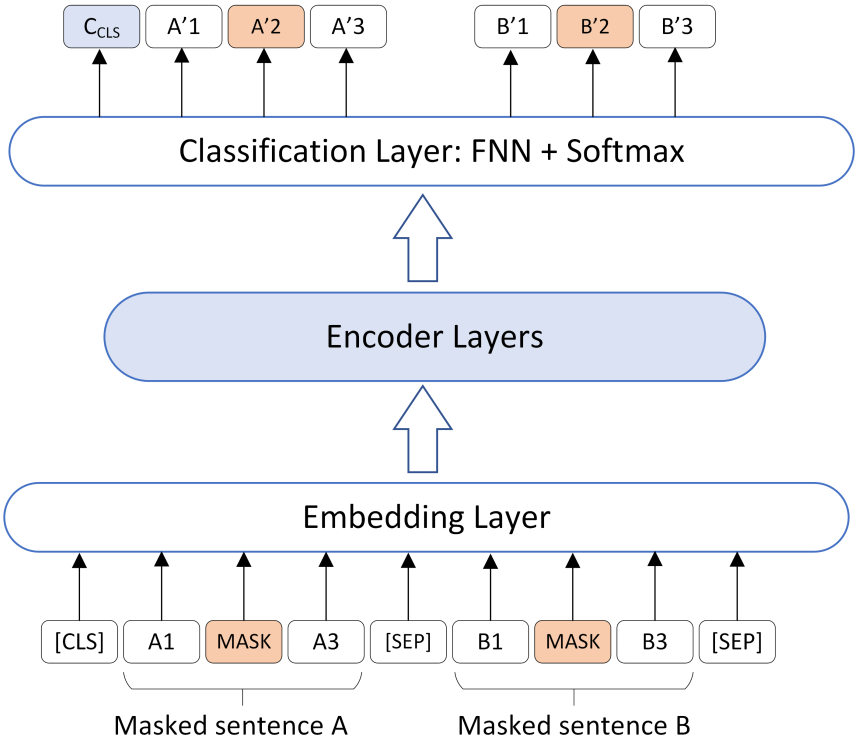

Figure 5.4: Pre-training BERT with NSP and Masked LM.

general linguistic capabilities of language models that are not constrained to a single specific application. The benchmark includes two single-sentence tasks, three similarity and paraphrase tasks and four inference tasks. According to the official GLUE benchmark leaderboard[1], $\text{BERT}_{\text{BASE}}$ and $\text{BERT}_{\text{LARGE}}$ have achieved a GLUE score of 78.3 and 80.5 respectively, beating the previous state-of-the-art, OpenAI GPT's 72.8 score and the 70.0 score of ELMo.

From the perspective of our story-point estimation model, BERT's performance on the two single-sentence GLUE benchmark tasks (SST-2, CoLA [37]) are of particular interest since these tests are about classifying single sequences using the CLS output of BERT. The SST-2 is a sentiment analysis task on the Stanford Sentiment Treebank [42] dataset, which consists of sentences from movie reviews and human annotations of their sentiment. The CoLA test is a grammatical acceptability judgment problem, the binary classification of which sentences are grammatically correct or not, based on the dataset of The Corpus of Linguistic Acceptability [43]. The large BERT achieved a score of 94.9 and 60.5 on the SST-2 and CoLA benchmark respectively, significantly outperforming the OpenAI GPT (91.3, 45.4) and ELMo (90.4, 36.0) models. For all of the above reasons, the pre-trained BERT architecture seems like a promising choice for the story-point estimation model.

[1]https://gluebenchmark.com/leaderboard

5.2 Implementation of BERT

To realize the BERT-based estimation model, a pre-trained BERT model have to be fine-tuned with the datasets shown in Table 4.1. There are thousands of pre-trained Transformer-based models in over 100 langauges accessible in the *Huggingface Transformers*[2] library from which we can choose the desired ones. Table 5.1 shows some of the available ones with their performance measures.

Model		No. Parameters	Performance		
			GLUE	CoLA	SST-2
BERT (by Google)	base	110M	78.3	52.1	93.5
	large	340M	80.5	60.5	94.9
RoBERTa (by Facebook AI)		355M	88.1	67.8	96.7
DeBERTa (by Microsoft)		355M	90.8	71.5	97.5
DistilBERT		66M	77	51.3	91.3
OpenAI GPT-2		1500M	72.8	45.4	91.3

Table 5.1: A few of the popular pre-trained models available in the public domain.

The Huggingface Transformers library provides high-level unified APIs to download and use the pre-trained models for our application, fine-tune them with our own dataset. This library is dedicated to supporting Transformer-based architectures and facilitating the distribution of pre-trained models through a centralized hub. The models are implemented using the most popular Python libraries, PyTorch and TensorFlow, with seamless integration between them [19]. To implement BERT for the story-point estimation model, I have used the PyTorch interface because it provides a nice balance between the high-level APIs and TensorFlow code and also includes practical pre-built modifications for BERT's top layer, such as the BertForSequenceClassification class, and other task-specific classes for tokenization, preprocessing, et cetera. PyTorch is an open-source machine learning framework that offers tensor computing capabilities with optional CUDA[3]-enabled GPU acceleration, which is immensely useful for fine-tuning BERT, and also provides building blocks and tools for deep neural networks. Although Python interface is used primarily for PyTorch, the Huggingface Transformer models are compatible with TorchScript, which is an intermediate representation that can be also run in a high-performance, production friendly environment such as C++.

In order to implement BERT for our estimation model, first, the training datasets, the tokenizer and the pre-trained model have to be loaded, then the datasets have to be preprocessed, parsed and tokenized, split into training and test sets. After configuring the hyper-parameters, samplers, the optimizer and other training settings, the model can be fine-tuned with the our datasets in a training loop for a few epochs. At the end of each epoch, the model is evaluated in the validation phase. The flowchart in Figure 5.5 summarizes the fine-tuning algorithm.

[2]https://github.com/huggingface/transformers

[3]Computer Unified Device Architecture by Nvidia: https://developer.nvidia.com/cuda-zone

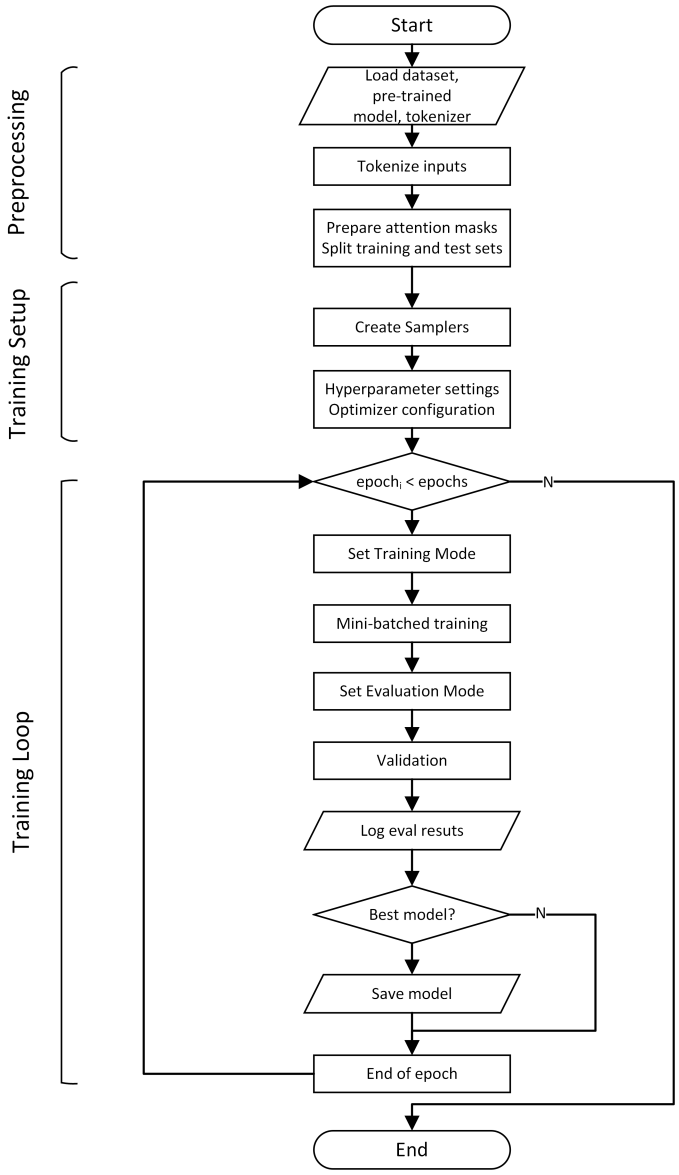

Figure 5.5: Flowchart of the fine-tuning algorithm.

5.2.1 Preprocessing

After loading our dataset and extracting the issue titles, descriptions and labels, the textual inputs are tokenized with the WordPiece-based BertTokenizer class. The tokenizer contains the most frequent English words and also all of the English characters and some sub-words, which are used to handle out-of-vocabulary words. It also appends the special [CLS] and [SEP] tokens to the input sequences. As an illustration, an example sentence would be tokenized the following way:

This is a sentence embedding.

< [CLS], 'This', 'is', 'a', 'sentence', 'em', '##bed', '##ding', '.', [SEP] >

< 101, 2023, 2003, 1037, 6251, 7861, 8270, 4667, 102 >

As it can be seen in this example, the word *embedding* is not in the vocabulary of BertTokenizer, and therefore it is dismantled into word pieces that are present in the vocabulary. The suffixes following the first word piece are appended with a double hash symbol to indicate they are sub-words, not separate words.

After tokenization, the sequences have to be truncated or padded with padding tokens (zeros) to the maximum input length, which is 512 tokens. The *pad_sequeces* utility from the *keras.preprocessing* module will do this for us. Since the BERT vocabulary does not use the ID 0, an attention mask have to be used to make it explicit for the model which tokens are padded and therefore should not be attended. The dataset is then split into a training set and validation set in a ration of 90/10 respectively. The labels and inputs have to converted to Torch tensors from NumPy arrays because the PyTorch implementation of BERT is used.

5.2.2 Fine-tuning BERT

Story-point estimation is a regression problem, and fine-tuning BERT for a regression task is possible by configuring a BertForSequenceClassification object with 1-dimensional labels ($num\_labels = 1$). BertForSequenceClassification is a pre-built task-specific BERT model class that inherits from the PreTrainedModel base class and implements a classification/regression head on the top of the basic BERT architecture via a simple linear layer the top of the pooled output. By specifying the output dimension of the classifier to be 1, we are trivially configuring it for regression, and the model will calculate the mean squared error as a loss instead of the cross-entropy loss. In order to acquire the model with pre-trained weights from a pre-trained model configuration, we have to instantiate it by calling the *from_pretrained()* method, which will download the desired pre-trained configuration given by the URI (Uniform Resource Identifier) passed as an argument.

Fine-tuning a BERT model is a memory demanding task beacuse of the sheer size of the model. Using the DataLoader utility from PyTorch to iterate over our dataset during training can help to offload the RAM. The DataLoader simplifies the handling of data samples and also improves readability by providing an iterable object around our dataset, and it also helps to save some memory during training because, unlike a for loop, with an iterator, the entire dataset does not need to be loaded into memory. Additionally, it also takes care of splitting our data into batches and shuffling them if necessary, which is useful since we are passing the samples in mini-batches to the model. To instantiate a DataLoader, we have to specify a sampler, the batch size and our dataset. The PyTorch SequentialSampler utility is used for sampling, and the batch size is chosen to be 16 as recommended by the authors of BERT. A DataLoader is instantiated for both the training and validation set.

In order to setup the fine-tuning process, the optimizer and learning rate scheduler has to be configured. I used the Adam optimizer [44] with a L2 weight decay of 0.01 to tune the weights. The parameters of the stored model have to be passed to the optimizer, and the learning rate is set to be 2×10^{-5}. To reduce the risk of overfitting, a linear learning rate scheduler is applied to decrease the learning rate to zero from its initial value step by step.

With all that being set up, the training process can be started. Since we are running the training on the GPU, the model have to be moved to the GPU with the CUDA API. From the perspective of computation time, it is critical to run the model on a GPU since training such an enormous neural network as BERT with hundreds of millions of parameters is a compute-intensive task. The forward pass calculations, back-propagation and weight optimization are large matrix operations essentially, which can be parallelized. By leveraging the parallel processing capabilities of GPUs, we can significantly reduce training times.

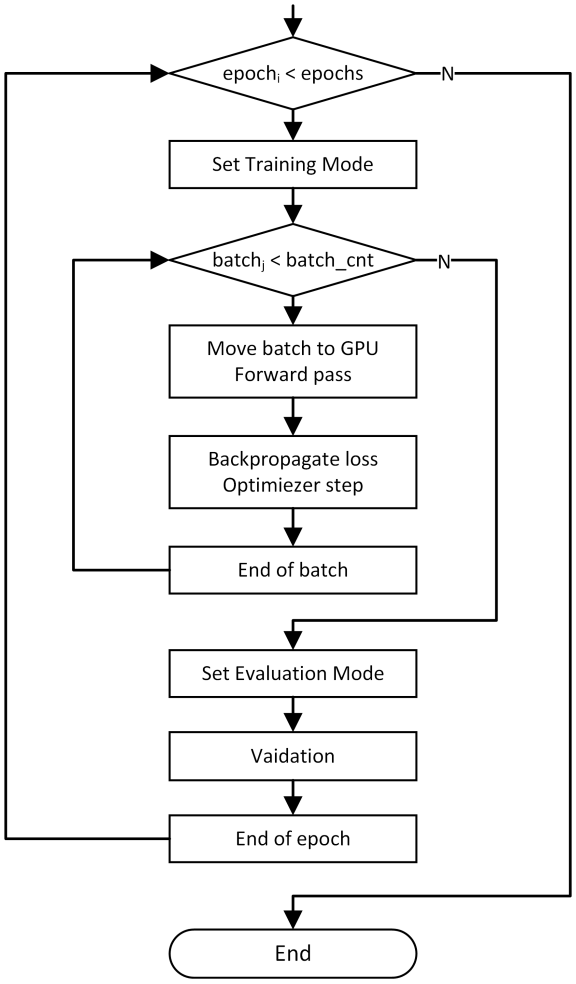

Figure 5.6: Flowchart of the training loop.

Training Loop. To fine-tune the model, I iterate through the dataset in a training loop for 10 epochs. The samples are passed to the model in small batches of 16 to facilitate mini-batched gradient descent and further reduce computational cost. For each epoch, there is a training and validation phase. The first step in the loop is to put the model into training mode by calling *model.train()* because dropout layers, batchnorm layers, et cetera, behave differently during training and evaluation mode. Then, there is an inner loop for the DataLoader to iterate through our dataset batch by batch. Every step, it provides a batch of inputs, attention masks and labels, which we then have to move to the GPU for acceleration. After this, a forward pass is performed by feeding the input data into the network. It is important to

clear out the gradients from the previous cycle with the *model.zero_grad()* function before passing a batch because PyTorch models accumulate gradients by default unless they are explicitly cleared. The model produces the outputs for the samples of the batch, and the mean squared errors are calculated with the labels. The loss is propagated backward to calculate the gradients, and by calling *optimizer.step()*, the parameters are updated. The learning rate is updated by the scheduler.

After executing these steps for all of the batches, the model enters validation phase. The model is set to evaluation mode, and a forward pass is performed on the validation set similarly to the training phase. There is no backward pass or gradient calculations here, the output values have to be moved back to the CPU to manually calculate the loss (MSE) and the relative error (MRE) of the predictions. The training loop is executed for the number of epochs specified, and the best performing (lowest MMRE) model is saved. The flowchart in Figure 5.6 summarizes the steps of the training loop.

5.3 Evaluation of BERT

5.3.1 Experimental Setting

The experiments were run on a Nvidia Tesla T4 cloud GPU in the Google Colaboratory environment. The same dataset that was used for the LSTM-based model (Table 4.1) was used for the training of the BERT models. Initially, I used the uncased base version of BERT as a pre-trained model to be fine-tuned, but I have tried other alterations of BERT also, such as the uncased $\text{BERT}_{\text{LARGE}}$, RoBERTa [45] or DeBERTa [46]. Thanks to the unified interfaces of the Huggingface Transformers library, running experiments for all of these different Transformer-based models does not require any major changes to the code base other than the *from_pretrained()* calls. The hyper-parameters and the training loop were setup in the same way as described in Section 5.2.2 for all of the experiments. I fine-tuned the models for 10 epochs. Generally, the models reached an optimum in 5-6 epochs, as can be seen from the plateau of the learning curves. Figure 5.7 and Figure 5.8 shows the learning curves of the $\text{BERT}_{\text{BASE}}$ models for all 15 datasets during fine-tuning. The vertical axis measures the average training loss (average MSE of the predictions) after each epoch.

RoBERTa. The Robustly Optimized BERT Pretraining Approach, abbreviated as RoBERTa, was proposed by Liu et al. [45], researchers at Facebook AI. It is a variation of the original $\text{BERT}_{\text{LARGE}}$ model that modifies some key hyper-parameters and removes the next-sentence prediction objective from the pre-training procedure and trains with much larger mini-batches and learning rates. It has exceeded the performance of $\text{BERT}_{\text{LARGE}}$ on the GLUE benchmark with a score of 88.1, but more importantly for our application, it has achieved better results on the singe-sentence tasks, CoLA and SST-2, with scores of 67.8 and 96.7 respectively.

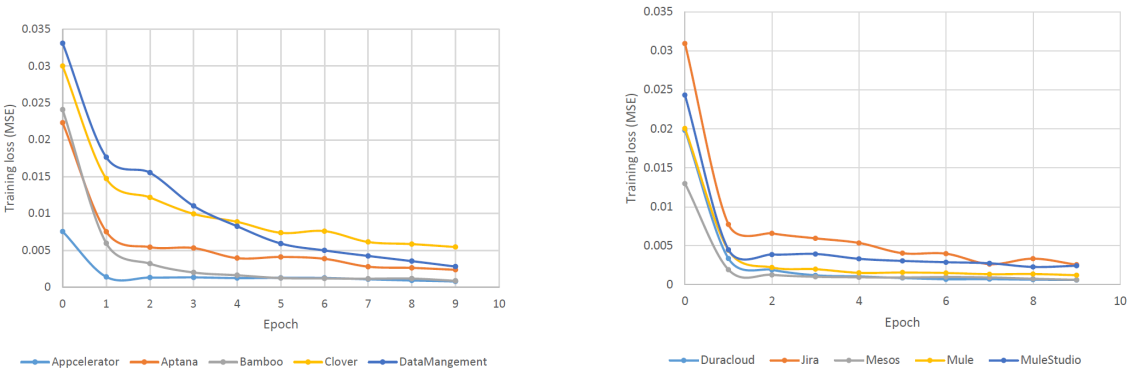

Figure 5.7: Learning curves of the BERT model during fine-tuning for the Appcelerator Studio, Aptana, Bamboo, Clover, DataMangement, Duracloud, Jira, Mesos, Mule and MuleStudio projects.

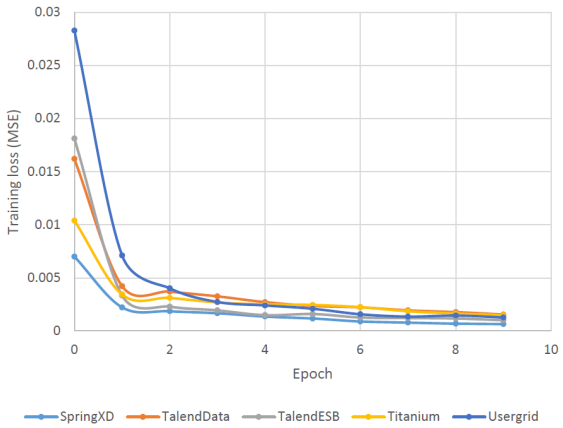

Figure 5.8: Learning curves of the BERT model during fine-tuning for the SpringXD, TalendData, TalendESB, Titanium, Usergrid projects.

DeBERTa. The Decoding-enhanced BERT with Disentangled Attention, abbreviated as DeBERTa, was proposed by Pengcheng He et al. [46], researchers at Microsoft, and it is an alteration of the BERT architecture. It is based on the RoBERTa model supplemented with two novel techniques: disentangled attention mechanism, where each word is represented using two vectors that encode its content and position, and enhanced mask decoders replacing the output softmax layer to predict the masked tokens during pre-training. This model has achieved slightly better results than RoBERTa on the GLUE benchmark. See Table 5.1 for a summary on the benchmark performance of the previously mentioned models.

5.3.2 Results

After fine-tuning a $\text{BERT}_{\text{BASE}}$, $\text{BERT}_{\text{LARGE}}$, RoBERTa and DeBERTa model for each of the 15 projects, I evaluated their performance on the validation set. The accuracy of a story-point prediction is measured by its absolute relative error (or

magnitude of relative error) from the actual value. The performance of an estimation model is associated with the average absolute relative error of its predictions on the validation set, or in other words, the mean magnitude of relative error (MMRE).

$$MMRE = \frac{1}{N} \cdot \sum_{i=1}^{N} \frac{| \text{TrueSP}_i - \text{PredSP}_i |}{\text{TrueSP}_i} \tag{5.1}$$

Table 5.2 summarizes the performance of the 4 model types on all the 15 different datasets.

	Mean Magnitude of Relative Error			
Dataset	$\text{BERT}_{\text{base}}$	$\text{BERT}_{\text{large}}$	RoBERTa	DeBERTa
Mesos	0.504	0.636	0.503	0.575
Usergrid	0.486	0.481	0.396	0.454
Appc. Studio	**0.393**	**0.374**	0.383	**0.403**
Aptana Studio	0.896	1.026	0.701	1.116
Titanium SDK	0.637	0.761	0.616	0.837
DuraCloud	0.585	0.492	**0.378**	0.392
Bamboo	0.545	0.548	0.444	0.503
Clover	1.017	0.979	0.789	*1.700*
Jira	0.902	0.479	0.493	0.509
DataMangement	0.921	*1.147*	*1.036*	1.241
Mule	0.650	0.718	0.677	0.684
Mule Studio	0.816	0.817	0.735	0.762
Sping XD	0.640	0.584	0.521	0.611
Talend DataQ	*1.087*	1.034	0.903	0.922
Talend ESB	0.550	0.515	0.520	0.547
Mean	**0.709**	**0.706**	**0.606**	**0.750**

Table 5.2: Results from the fine-tuned $\text{BERT}_{\text{BASE}}$, $\text{BERT}_{\text{LARGE}}$, RoBERTa and DeBERTa models.

As it can be seen in the table, the best performing model was RoBERTa with a 0.606 mean MMRE across all the datasets. It was the most accurate in the DuraCloud project, having a relative error as low as 37.8%, which is significantly better than the performance of the original BERT models. RoBERTa was the most inaccurate in the case of the DataManagement dataset, with a relative error of 103.6%. In 8 out of the 15 cases (Mesos, Usergrid, Appc. Studio, DuraCloud, Bamboo, Jira, Spring XD and Talend ESB), it has achieved similar or better accuracy compared to human estimations (0.47 MMRE [10]). $\text{BERT}_{\text{LARGE}}$ with its 0.706 mean MMRE did not bring much improvement in accuracy compared to $\text{BERT}_{\text{BASE}}$ with 0.709. Both of them had the lowest relative error with Appcelerator Studio project, and similarly to the other two models, they have the worst efficacy in the DataManagement, Talend Data Quality and Clover datasets. The performance metrics are highly variable among the different datasets just like in the LSTM case, and the best and worst performing cases are seem to be the same in each model types.

The RoBERTa model has outperformed the $\text{BERT}_{\text{LARGE}}$ model by 14.16% on these story-point estimation tasks, which would be consistent with its superiority on the GLUE benchmark. However, the opposite is the case with DeBERTa, which has a 6.23% higher mean MMRE (0.750) than $\text{BERT}_{\text{LARGE}}$, despite having even higher benchmark ratings than RoBERTa. The relative under-performance of DeBERTa could be a matter of further studies.

5.3.3 Comparison

In the previous section, I have established that RoBERTa is the best performing model for the estimation engine among the studied BERT-based architectures. Now, I compare it to the first solution, the LSTM-based estimation model. Table 5.3 presents the MMRE metrics for the LSTM-based model and RoBERTa in each dataset, and also notes the percentage-wise change in the relative error that RoBERTa achieved.

	Mean Magnitude of Relative Error		
Dataset	LSTM-based Model	RoBERTa	Change
Mesos	0.464	0.503	8.34%
Usergrid	0.567	0.396	-30.17%
Appc. Studio	0.348	0.383	10.01%
Aptana Studio	0.411	0.701	70.77%
Titanium SDK	0.632	0.616	-2.58%
DuraCloud	0.442	0.378	-14.56%
Bamboo	0.526	0.444	-15.53%
Clover	0.777	0.789	1.52%
Jira	0.507	0.493	-2.80%
DataMangement	1.061	1.036	-2.33%
Mule	0.972	0.677	-30.33%
Mule Studio	0.873	0.735	-15.81%
Sping XD	0.916	0.521	-43.10%
Talend DataQ	0.947	0.903	-4.60%
Talend ESB	0.491	0.520	5.95%
Mean	**0.662**	**0.606**	**-8.44%**

Table 5.3: Comparison between the performance of the LSTM-based estimation model and the RoBERTa-based estimation model.

The RoBERTa-based estimation model achieved significant improvement (decrease) in MMRE in the case of the Usergrid (-30.17%), DuraCloud (-14.56%), Bamboo (-15.53%), Mule (-30.33%), MuleStudio (-15.81%), SpringXD (-43.10%) product backlogs, and it suffers from a significant increase in MMRE in the case of the Appcelerator (10.01%) and Aptana Studio (70.77%) projects compared to the LSTM-based model. Overall, RoBERTa has sightly outperformed the LSTM-based model by decreasing the mean MMRE across the datasets by 8.84% to 0.606. However,

this improvement comes at a cost. Fine-tuning a BERT model requires much more compute power and memory than training the LSTM model. As a comparison between compute costs, Table 5.4 presents the training times for the two models for each dataset.

	Training time	
Dataset	LSTM-based Model	RoBERTa
Mesos	17 min	24 min
Usergrid	11 min	19 min
Appc. Studio	19 min	28 min
Aptana Studio	14 min	20 min
Titanium SDK	24 min	35 min
DuraCloud	14 min	22 min
Bamboo	11 min	19 min
Clover	10 min	17 min
Jira	8 min	14 min
DataMangement	30 min	53 min
Mule	17 min	26 min
Mule Studio	13 min	20 min
Sping XD	29 min	41 min
Talend DataQ	15 min	25 min
Talend ESB	12 min	17 min
Mean	**14 min**	**25 min**

Table 5.4: Training times for the LSTM-based model and the RoBERTa-based model.

The fine-tuning of the RoBERTa models took on average 25 minutes, which exceeds the 14-minute average training time of the LSTM-based models by more than 70% even though the BERT models were fine-tuned with much more powerful resources than the LSTM models (an Nvidia T4 datacenter GPU versus an Intel i7-Core CPU of a personal computer).

On the other hand, because of the modularity of the Huggingface Transformers library and their unified interfaces, it is much easier to implement and integrate BERT models, and switching between models or implementing potentially new models can be done in straightforward, simple manner through the high-level interfaces. Additionally, pre-trained BERT models are available in multiple languages, therefore, the effort estimation engine would not necessarily be restricted to English product backlogs, it could be implemented in other languages too. For all of these reasons, I will choose the RoBERTa-based model for the story-point estimation engine that will be integrated into the decision support system.

Chapter 6

Conclusion

Throughout this thesis, I have provided an overview on basic Scrum processes and identified possible areas and challenges that could be supported by machine learning, specifically pointing out the problems of sprint planning and software effort estimation. I studied the latest proceedings that are pursuing to implement some type of automation to support these processes and presented existing story-point estimation methods. I gave an extensive literature review on the field of natural language processing, its history, theoretical background and the latest state-of-the-art technologies and deep-learning architectures used in the field as NLP techniques are the key to extract insights for business intelligence applications from textual data. Based on my findings, I have proposed a concept for a machine learning-based decision support system that can process the textual artifacts of a product backlog and can automate the effort estimation of backlog items, and thus, accelerate some key agile processes. I have identified that a story-point estimation model is the critical part for creating such a system and proposed and implemented two distinct approaches to tackle this problem. The first was a re-implementation of an LSTM architecture that had previously achieved state-of-the-art results on story-point estimation tasks, and the other solution was fine-tuning pre-trained BERT models for this specific regression problem.

Upon evaluating and comparing the two solutions, I have found promising results. According to my literature review and to my current knowledge, this thesis is the first to present an implementation of a BERT architecture for the specific problem of story-point estimation, and moreover, the RoBERTa-based model has produced better results than the already existing, LSTM-based solution and three other BERT-based counterparts as well on the applied issue report datasets. Both of the solutions had an accuracy comparable to human predictions in around half of the cases, which affirms the eligibility of NLP applications in agile processes. Transformer-based pre-trained architectures revolutionized the field of machine learning and provide a powerful tool for developing scalable, high-performance language processing solutions. Once again they have proven their efficacy through the specific application of story-point estimation. Leveraging the capabilities of Bidirectional Encoder Representations from Transformers, I have successfully built the basis of an agile decision support system that could be a great value-add for Scrum practitioners. I am looking forward to integrate my effort estimation solution into a cloud-based service or as

an add-on for a project management platform to make it available for the developer community in the future.

Acknowledgements

The author would like to express his thanks to Adam Kovacs and Khalid Kahloot for their support as scientific advisors.

Bibliography

[1] K. Schwaber and J. Sutherland, *The Scrum Guide*. Creative Commons, 2020.

[2] S. Hochreiter and J. Schmidhuber, "Long short-term memory," 1997.

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

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

[5] M. Fernandez-Diego, E. R. Mendez, F. Gonzalez-Ladron-De-Guevara, S. Abrahao, and E. Insfran, "An update on effort estimation in agile software development: A systematic literature review," *IEEE Access*, vol. 8, pp. 166768–166800, 2020.

[6] M. Usman, E. Mendes, F. Neiva, and R. Britto, "Effort estimation in agile software development: A systematic literature review," 2014.

[7] H. Dam, T. Tran, J. Grundy, A. Ghose, and Y. Kamei, "Towards effective ai-powered agile project management," 2018.

[8] M. Cohn, *Agile Estimating and Planning*. USA: Prentice Hall PTR, 2005.

[9] S. Porru, A. Murgia, S. Demeyer, M. Marchesi, and R. Tonelli, "Estimating story points from issue reports," 2016.

[10] N. C. Haugen, "An empirical study of using planning poker for user story estimation," in *AGILE 2006*, 2006.

[11] M. Choetkiertikul, H. K. Dam, T. Tran, T. Pham, A. Ghose, and T. Menzies, "A deep learning model for estimating story points," *IEEE Transactions on Software Engineering*, vol. 45, no. 7, pp. 637–656, 2019.

[12] J. Zilly, R. Srivastava, J. Koutník, and J. Schmidhuber, "Recurrent highway networks," 07 2016.

[13] J. Hutchins, "The history of machine translation in a nutshell," 2006.

[14] F. Pelletier, "The principle of semantic compositionality," *Topoi*, pp. 11–24, 1994.

[15] K. Cho, B. van Merrienboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using rnn encoder-decoder for statistical machine translation," 2014.

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

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

[18] J. Chung, C. Gulcehre, K. Cho, and Y. Bengio, "Empirical evaluation of gated recurrent neural networks on sequence modeling," 2014.

[19] T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, T. Rault, R. Louf, M. Funtowicz, J. Davison, S. Shleifer, P. von Platen, C. Ma, Y. Jernite, J. Plu, C. Xu, T. L. Scao, S. Gugger, M. Drame, Q. Lhoest, and A. M. Rush, "Transformers: State-of-the-art natural language processing," in *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, (Online), pp. 38–45, Association for Computational Linguistics, Oct. 2020.

[20] Y. Bengio, R. Ducharme, P. Vincent, and C. Janvin, "A neural probabilistic language model," *J. Mach. Learn. Res.*, vol. 3, Mar. 2003.

[21] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," 2014.

[22] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," 2014.

[23] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," 2015.

[24] J. L. Ba, J. R. Kiros, and G. E. Hinton, "Layer normalization," 2016.

[25] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," vol. 15, no. 1, 2014.

[26] "Tensorflow." https://tensorflow.org/. Accessed: 2020-10-30.

[27] "Pytorch." https://pytorch.org/. Accessed: 2020-10-30.

[28] "Keras." https://keras.io. Accessed: 2020-10-30.

[29] "Theano." https://github.com/Theano/Theano. Accessed: 2020-10-30.

[30] "Numpy." https://numpy.org/. Accessed: 2020-10-30.

[31] "Pandas." https://pandas.pydata.org/. Accessed: 2020-10-30.

[32] "Scikit-learn." https://scikit-learn.org/stable/. Accessed: 2020-10-30.

[33] R. K. Srivastava, K. Greff, and J. Schmidhuber, "Highway networks," *CoRR*, vol. abs/1505.00387, 2015.

[34] M. U. Gutmann and A. Hyvärinen, "Noise-contrastive estimation of unnormalized statistical models, with applications to natural image statistics," vol. 13, no. null, 2012.

[35] "Jira issue tracker." https://issues.apache.org/jira/secure/Dashboard.jspa. Accessed: 2020-10-30.

[36] P. Rajpurkar, J. Zhang, K. Lopyrev, and P. Liang, "Squad: 100,000+ questions for machine comprehension of text," 2016.

[37] A. Wang, A. Singh, J. Michael, F. Hill, O. Levy, and S. R. Bowman, "Glue: A multi-task benchmark and analysis platform for natural language understanding," 2019.

[38] Y. Wu, M. Schuster, Z. Chen, Q. V. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, K. Macherey, J. Klingner, A. Shah, M. Johnson, X. Liu, L. Kaiser, S. Gouws, Y. Kato, T. Kudo, H. Kazawa, K. Stevens, G. Kurian, N. Patil, W. Wang, C. Young, J. Smith, J. Riesa, A. Rudnick, O. Vinyals, G. Corrado, M. Hughes, and J. Dean, "Google's neural machine translation system: Bridging the gap between human and machine translation," 2016.

[39] Y. Zhu, R. Kiros, R. Zemel, R. Salakhutdinov, R. Urtasun, A. Torralba, and S. Fidler, "Aligning books and movies: Towards story-like visual explanations by watching movies and reading books," 2015.

[40] A. Radford and K. Narasimhan, "Improving language understanding by generative pre-training," 2018.

[41] M. E. Peters, M. Neumann, M. Iyyer, M. Gardner, C. Clark, K. Lee, and L. Zettlemoyer, "Deep contextualized word representations," 2018.

[42] R. Socher, A. Perelygin, J. Wu, J. Chuang, C. D. Manning, A. Ng, and C. Potts, "Recursive deep models for semantic compositionality over a sentiment treebank," in *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing*, (Seattle, Washington, USA), pp. 1631–1642, Association for Computational Linguistics, Oct. 2013.

[43] A. Warstadt, A. Singh, and S. R. Bowman, "Neural network acceptability judgments," *arXiv preprint arXiv:1805.12471*, 2018.

[44] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," 2017.

[45] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, "Roberta: A robustly optimized bert pretraining approach," 2019.

[46] P. He, X. Liu, J. Gao, and W. Chen, "Deberta: Decoding-enhanced bert with disentangled attention," 2021.