Universidade de Aveiro Departamento de Eletrónica,
Telecomunicações e Informática
2018

Pedro Ferreira de Matos

Reconhecimento de mutações genéticas em texto usando deep learning

Recognition of genetic mutations in text using deep learning

Universidade de Aveiro Departamento de Eletrónica, Telecomunicações e Informática
2018

Pedro Ferreira de Matos

Reconhecimento de mutações genéticas em texto usando deep learning

Recognition of genetic mutations in text using deep learning

"*We can't solve problems by using the same kind of thinking we used when we created them*"

— Albert Einstein

Universidade de Aveiro
2018

Departamento de Eletrónica,
Telecomunicações e Informática

Pedro Ferreira de Matos

Reconhecimento de mutações genéticas em texto usando deep learning

Recognition of genetic mutations in text using deep learning

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Mestre em Engenharia de Computadores e Telemática, realizada sob a orientação científica do Doutor Sérgio Guilherme Aleixo de Matos, Professor auxiliar do Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro.

Dedico este trabalho à minha familia.

agradecimentos /
acknowledgements

Agradeço a todos os meus colegas e professores que me proporcionaram conhecimentos e momentos fundamentais para o meu desenvolvimento ao longo destes anos.

Palavras Chave Deep Learning, NER, BI-LSTM-CRF, Redes Neuronais, Inteligencia Artificial

Resumo

Deep Learning é uma subárea de aprendizagem automática que tenta modelar estruturas complexas no dados através da aplicação de diferentes arquitecturas de redes neuronais com várias camadas de processamento. Estes métodos foram aplicados com sucesso em áreas que vão desde o reconhecimento de imagem e classificação, processamento de linguagem natural e bioinformática. Neste trabalho pretendemos criar métodos para reconhecimento de entidades nomeadas (NER) no texto usando técnicas de Deep Learning, a fim de identificar mutações genéticas.

Keywords Deep Learning, NER, BI-LSTM-CRF, Neural Networks, Artificial Intelligence

Abstract

Deep learning is a sub-area of automatic learning that attempts to model complex structures in the data through the application of different neural network architectures with multiple layers of processing. These methods have been successfully applied in areas ranging from image recognition and classification, natural language processing, and bioinformatics. In this work we intend to create methods for named-entity recognition (NER) in text using techniques of deep learning in order to identify genetic mutations.

Contents

List of Figures

List of Tables

Acronyms

AI Artificial Inteligence. 4–7

ANN Artificial Neural Networks. 6, 9, 14

BI-LSTM Bidirectional Long Short-Term Memory. 13, 17, 20, 21, 29, 31, 33, 34, 38, 48, 49

CNN Convolutional Neural Network. 8, 9, 14, 15, 17–20, 23, 29, 30

CPU Central Processing Unit. 7, 8

CRF Conditional Random Fields. 1, 21, 25, 27–31, 33, 34, 50

CV Computer Vision. 14

DL Deep Learning. 1, 4, 6–9, 17, 31, 43, 47, 50

GPU Graphics Processing Unit. 4, 7, 8

GRU Gated Recurrent Unit. 17, 19

HGVS Human Genome Variation Society. 1, 26, 27

IE Information Extraction. 3

LSTM Long Short-Term Memory. 9–11, 13, 17, 19–22, 24, 29, 35, 38, 39, 48, 49

ML Machine Learning. 4, 6

NER Named-Entity Recognition. 1–5, 9, 13, 17, 22, 25, 30, 49, 50

NLP Natural Language Processing. 21, 24

NLP Neuro-Linguistic Programming. 15, 17–19, 48

RNN Recurrent Neural Network. 9–11, 17–20, 22, 23, 29, 30, 35, 39

SGD Stochastic Gradient Descent. 43

SNLI Stanford Natural Language Inference. 18

SST Stanford Sentiment Treebank. 18

TF TensorFlow. 8, 9, 34–36, 51

VSM Vector Space Models. 16

Chapter 1

Introduction

This thesis is related with the creation of an algorithm to perform a Named-Entity Recognition (NER) task in a scientific corpus (biomedical domain) to find genetic mutations. This NER task is a sequence labelling problem in which is necessary to identify all the entities that are part of a genetic mutation. To solve this problem we decided to create two Deep Learning (DL) models, since DL excels at feature extraction and it is not necessary to complement with extensive hand-crafted rules or external dictionaries.

TmVar is a text-mining approach based on Conditional Random Fields (CRF) for extracting a wide range of sequences variants described at protein, DNA and RNA levels according to a standard nomenclature developed by the Human Genome Variation Society (HGVS). This work created a tool that is a high-performance method for mutation extraction from biomedical literature but it also created a dataset that contains biomedical documents and the respective annotations that can be used for works like our own. The dataset created by the tmVar work is the one we used in this thesis.

There are a lot of methods that can be used as an approach to text classification problems. TmVar used an approach based on CRF, another systems consist in the creation of dictionaries to have sources of external knowledge. There are three categories for NER systems: rule-based algorithms, machine learning algorithms and hybrid. Yet, most NER systems rely heavily on hand-crafted features and domain-specific knowledge in order to learn effectively from the small, supervised training corpora that are available. Even some machine learning approaches need to have some rules or features to treat more particular types of data that are not easily learnt by the models on its own. Our objective is to have the best possible results with the less hand-made features. The focus of our work was to evaluate the use of DL for this problem, comparing two types of models: one using word embeddings and the other using character embeddings. Character level embeddings are a common choice with biomedical corpus because of its complexity and being morphologically richer than a standard corpus. The relations between the characters can be useful in understanding the corpus and learn from it. Word models also have the problem of tokenization that can introduce some problems when treating the corpus and the fact that it is necessary to have word vectors to represent the tokens, otherwise some tokens will be seen as "unknown" to the model. For this reasons we believe that a character model can not only be easier to create but also achieve better perfomance on this sequence labelling task.

Both models created in this work were trained with the Keras framework [13] but one achieved higher performance than the other in terms of results. Each main model was trained in two configurations and both have a **BI-LSTM-CRF** as neural network. It consists of a bi-directional layer of LSTM cells with a CRF layer on top. This bi-directional layer allows the network to efficiently predict a result based on both past (forward states) and future features (backward states). The character model was trained with a mini-batch approach, in which sentences with similar length are grouped, and with a maximum sequence length approach. The word model was trained using two embeddings models: the BioNPLab word2vec model [14] and the GloVe model [15]. The best results were obtained with the character model and the mini-batch approach with an F-Measure of 87%, while the word model achieved 71.8%.

The first chapter contains information about the current state of the art in Deep learning for biomedical NER, offering context about the existing NER tasks and all the technologies surrounding the world of deep learning. It also explains what type of architectures are used in Deep learning and more specifically in tasks involving text classification. The second chapter will focus on the implementation done in this work, in particular the character and word models. It will also focus on what type of difficulties both models faced, the pre-processing done in each one and the results achieved by all the approaches. At last, it will be presented a conclusion that will review and comment the results presented here and explained in more detail in the second chapter. It will also discuss if the objective of creating a system with less to none hand-crafted rules was successful and what lines can be followed in future work.

Chapter 2

Background

2.1 Named-Entity Recognition

Information Extraction (IE) is the process of extracting specific information from any type of textual sources. The objective is to gather detailed and structured information that can be used to tasks like classification, integrated search or data-driven activities like mining for patterns or uncovering hidden relationships[16].

Today's technological advances have brought the possibility to access large amounts of textual information. However, it is very difficult to digest all the available information, especially because of its unstructured format. For this reason, IE is concerned with structuring all the relevant information from any given source. In other words, the goal of an IE system is to find and link the relevant information while ignoring the extraneous and irrelevant one[17]. Since a lot of today's information is available in natural language, an unstructured format, IE can help structuring the free-text information in a way that can be used by other tasks to mine knowledge out of it[18]. Even if the information is structured, it is necessary to create an algorithm or a system that can process this information, understand it and classify it on its own. This taks is a process of Named-Entity Recognition (NER).

Named-Entity Recognition (NER) is part of the process in Text Mining used for IE. While IE is a process that extracts information from unstructured text to provide more useful information about it, NER is a task of identifying proper names of people, locations, proteins, gene mutations, or other entities from natural language documents[19]. A domain-specific NER application may not be applicable for recognizing named-entities on other specific domains. For instance, Abner [20] will not perform well in processing military articles as they are designed for different domains[19]. While designing good features for NER systems requires a great deal of expertise and can be labour intensive, it also makes the taggers harder to adapt to new domains and languages since resources and syntactic parsers used to generate the features may not be readily available.

State-of-the-art NER systems rely heavily on hand-crafted features and domain-specific knowledge in order to learn effectively from the small, supervised training corpora that are available[21]. Some NER systems are comprised primarily of text parsers as in [22]. Another systems consists in the construction and use of gazetteers and dictionaries in order to have sources of external knowledge, even if they are not the core of the system but instead help in feature detection.

Algorithms for Named-Entity Recognition systems can be classified into three categories; rule-based, machine learning and hybrid [19]. A **Rule-Based** NER algorithm detects the named entity by using a set of rules and a list of dictionaries that are manually pre-defined by human. The patterns are mostly made up from grammatical, syntactic and orthographic features. Next, a **machine learning** NER algorithm normally involves the usage of ML techniques and a list of dictionaries. There are two types of ML model for the NER algorithms: supervised and unsupervised machine learning model. Unsupervised model does not require any training data, it is more closely aligned with what some call true artificial intelligence — the idea that a computer can learn to identify complex processes and patterns without a human to provide guidance along the way [23]. Unlike the methods before, supervised NER methods require a large amount of annotated data to produce a good NER system. ML methods are applicable for different domain-specific NER systems but it requires a large collection of annotated data. Hence, this might require high time-complexity to preprocess the annotate data[19]. Finally, a **hybrid** named entity recognition algorithm implements both the rule-based and machine learning methods [24]. Such method will produce a better result. However, the weaknesses of the rule-based are still unavoidable in this hybrid system. A domain-specific NER algorithm may need to customize the set of rules used to recognize different types of named entity when the domain of studies is changed.

Artificial Inteligence (AI) has evolved a lot since its first steps and it is becoming a powerful ally in NER systems. In the beginning technologies did not have the same capabilities as of today, so it was impossible to realize some tasks. Over the years, technology evolved, especially after 2015 due to the expansion of the Graphics Processing Unit (GPU) market, which lead to a faster parallel processing[25]. This expansion, together with the Big Data movement, were the perfect stimulator to a sub-field of AI and ML: Deep Learning (DL). In the figure 2.1 we can see a visual representation of this fields and how they are part of the world of Artificial Inteligence.

ML is, at its basic, the use of algorithms based on **statistical methods** to enable machines to learn from data and make predictions based on it[25]. ML is a subset of Artificial Inteligence but in its core is really just powerful math & prediction. This process is very different to the typical programming routines created by human hand, yet ML algorithms still need some guidance

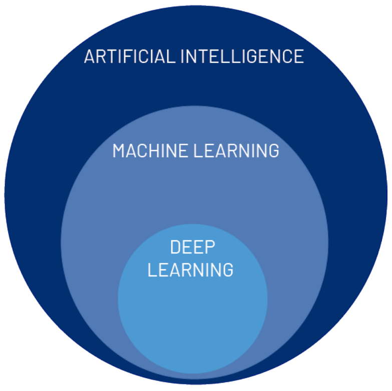


Figure 2.1: Fields of AI. Adapted from [1]

by the programmers in order to identify the features so that the model can learn better and see how the data is related. In order to perform these tasks, it is necessary to feed data in a format that a machine can read so that it can be learnt by its algorithms. These algorithms/techniques include linear regression, logistic regression, k-means clustering, decision trees, random forests, and more, and they can all be applied to a variety of problems. Some of these use cases are related to Data Science and can be applied in real-life problems like employing natural language processing in chat logs at online games to flag users that use offensive language or building any type of predictive model. But this still was not enough to replicate the most pure idea of Artificial Inteligence, something that was capable to process and learn information in a much more powerfull way and on its own, replicating the learning processes of our brain.

2.1.1 Evaluation

It is important to know if NER systems perform well enough for the task they were created. They can be evaluated in terms of *precision*, *recall*, and *f-measure*. Precision is the capability of the model not to label as positive a sample that is negative, recall is the capability of the model to find all the positive samples and f-measure can be interpreted as a weighted harmonic mean of the precision and recall.

In order to achieve this metrics we needed to calculate the number of *true positives*, *false positives* and *false negatives*. The *true positives* is the number of correct entities the model has predicted. If the model predicted 5 entities to be mutations, and only 3 are mutations, the number of true positives is 3. The *false positives* is the number of entities that were predicted as mutations but are not mutations. In another words, all the mutations the model predicted wrongly. In the previous example, if 2 of the 5 predicted mutations, are not mutations, this means that the number of false positives is 2. The *false negatives* is the subtraction between all the existing mutations and the true positives. The correct mutations are not the ones predicted by the model but the number of all the existing ones in the corpus. To calculate the precision, recall and f-measure the following formulas were used, where *tp_ all* is the number of all the true positives, *fp_ all* is the number of all the false positives and *fn_ all* is the number of all the false negatives:

```
Precision = tp_all / (tp_all + fp_all)
Recal = tp_all / (tp_all + fn_all)
F-measure = (2 * tp_all / (tp_all + tp_all + fp_all + fn_all))
```

2.2 Deep Learning

Deep Learning is a field of Machine Learning that is inspired by the way the human brain processes information. The most important aspect are the neurons and the interconnections between themselves, which are similar to a network like is portrayed at fig 2.2. The idea of creating a similar structure to the brain in order to allow Artificial Inteligence (AI) to process data and learn it on its own has existed for a long time but the technology available before did not allow for much. We can say that while ML needs the help of a human to indicate the appropriate features of the data to the algorithms and then learn from that data, DL creates an artificial neural network that can learn and make intelligent decisions on its own[26]. With this in mind, we can assume that Artificial Neural Networks (ANN) are the core of Deep Learning.

Figure 2.2: The way our brain processes information can be seen as a network [2]

In our brain, neurons are used to communicate with the rest of our body by sending information from one neuron to another, till it reaches the desired part of the body. ANN tries to replicate this by having at least three layers, where in each layer there are a lot of neurons that receive and process the information. The first layer is called **Input layer** because it only receives the initial data, and the last layer is called **Output layer** because it only gives the final output after all the data being processed. The layers in the middle are used to perform different functions on the data, in order to transform the inputs into something that the output layer can use. This layers are called **Hidden layers**. We can see a representation of this structure in the figure 2.3. The input and output layer are obligatory while the hidden layers can vary in number, conforming to the problem and the needs of each project.

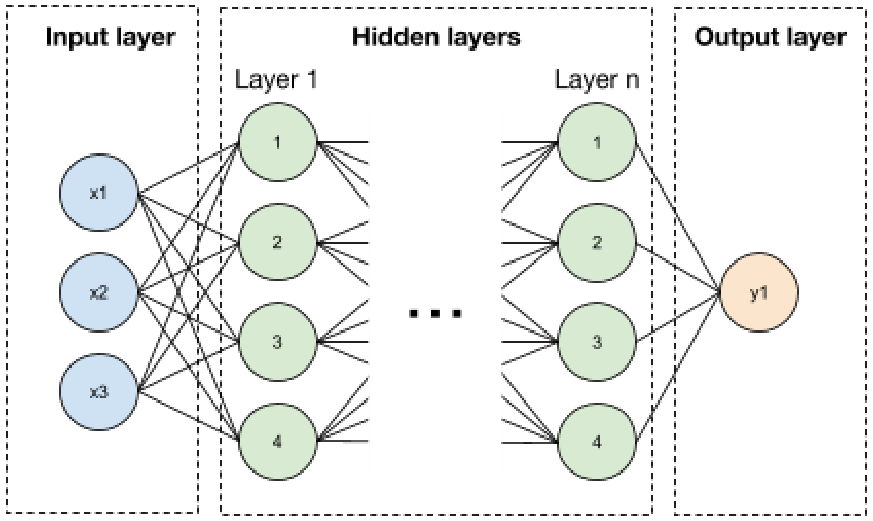


Figure 2.3: Neural Network structure

Nowadays it is easier to progress into Deep Learning because of the computational power available everywhere. Even if it is necessary a lot of GPUs to process really large amounts of data, anyone can try it at home and get decent results with the common GPUs of our machines or even with the CPU. Another reason why DL is so important is because the performance improves as the data increases, and today there's lots and lots of data. Neural Networks behave better when they have a lot of data so that they can understand it better [27]. The downside of this aspect is that the more data the more computation power is needed to train it, therefore needing also more time.

Deep Learning has a lot of uses nowadays, from image recognition or natural language generation, to recommendations in social media like Youtube or Netflix. At Google, it all started with the **Deep Mind** work, which was responsible for the creation of the **Alpha Go**, a famous game where the AI behind it was capable of defeating the world's best players [28]. Another one of Google's latest breakthrough involving DL is in the field of image analytics, more specifically in image enhancement [29]. This involves restoring or filling in detail missing from images, by extrapolating for data that is present, as well as using what it knows about other similar images. This examples are more specific and related to situations that are not dealt by the majority of the population. Yet, DL can be very important when dealing with the most famous social media networks that reach to millions, or even billions of users. Google uses it to provide useful recommendations on Youtube. The system monitors and records the viewing habits of the users as they stream content from their servers. Then the neural networks are put to work studying and learning everything about the users so that they can give betters suggestions to each user in order to keep them glued to the screen[30].

Deep Learning is particularly effective in feature detection [31]. Feature detection is a process of transforming all the learned knowledged into the creation of feature extractions to reduce the complexity of the data and make patterns more visible to learning algorithms to work. Deep Learning algorithms try to learn high-level features from its data like in image recognition where the neural network will try to learn the low-level features like nose, eyes, mouth, and then the high-level representation of a face[27]. Convolutional Neural Network (CNN) are used to perform this task and are very common.

2.2.1 Technologies used in Deep Learning

DL is a world on its own and is full of a technologies that can implement neural networks. The most famous are TF and Keras.

TensorFlow is an open source software library for numerical computation using data-flow graphs. It was originally developed by the Google Brain Team within Google's Machine Intelligence research organization for machine learning and deep neural networks research. As it is possible to see on the image 2.4, TF is supported in a lot of devices as CPU, GPU, and even mobile devices.

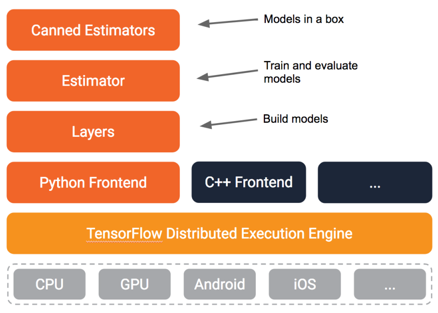


Figure 2.4: TensorFlow's architecture [3]

Its core is C++ but it offers more Frontends like Python. The Layers API provides a simpler interface for commonly used layers in DL models while the Estimator API makes training and evaluating distributed models easier. With TF the user has full freedom to create the ANN from scratch, but this is very difficult for users with low experience. TF is very low-level and the network needs to be all connected by the user, which means that in complex networks that involve lots of layers, users with low experience can have lots of difficulties.

On the other hand, **Keras** is a high level API written in Python and can be deployed on top of other AI technologies such as TensorFlow, Microsoft Cognitive Toolkit (CNTK), and Theano, as it is possible to see in fig fig 2.5. Keras is more user friendly than TF and offers more modularity and ease of extensibility. It is more suitable for users with less knowledge or if they need faster prototyping.

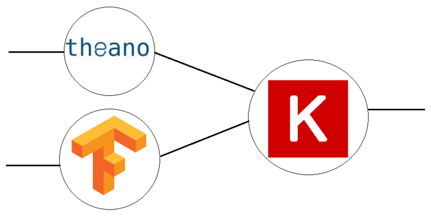


Figure 2.5: Keras built on top of TF or Theano [4]

2.2.2 Neural networks used in text classification

It was stated before that Deep Learning creates an ANN that can learn and make intelligent decisions on its own. This is very important because when we think that creating domain-specific NER algorithms is a very laborious process, it becomes obvious that the use of DL can help reduce the linguistic analysis knowledge required in the traditional NER models. There are various models that can be used in DL. We are going to talk about two: Recurrent Neural Network (RNN) and Convolutional Neural Network (CNN).

Recurrent Neural Network

RNN are networks with loops, allowing information to persist. In another words, this networks have "memory" [32]. These loops make RNN seem kind of mysterious. However, a RNN can be thought of as multiple copies of the same network, each passing a message to a successor [32] like in the following figure 2.6. This type of network is very successfull in a variety of problems: speech recognition, language modeling, translation, image captioning. . . However, the use of LSTM, a special type of RNN leads these tasks to much better results than the standard version(RNN). The use of LSTMs is very important because it has a long term memory while the traditional neural networks can't retain information. The problem with RNN is that they can only retain recent information which can be a serious problem when it is necessary to have past context to predict what is desired. That way to have a long term memory it is necessary to use LSTMs and not just the standard versions of RNNs.

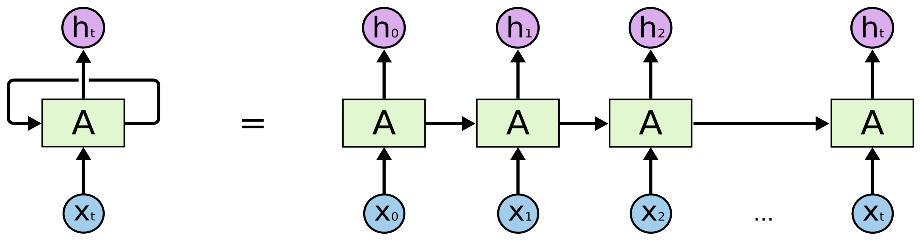


Figure 2.6: Recurrent Neural Networks fuction as a loop [5]

LSTMs contain information outside the normal flow of the RNN in a gated cell. Information can be stored in, written to or read from a cell. Those cells make decisions about when to store, write or read the information via gates that open and close as seen in 2.7. The gates act on signals they receive, blocking or passing information based on its strenght, which they filter with their own sets of weights. Those weights, like the weights that modulate input and hidden states, are adjusted via the recurrent networks learning process. That is, the cells learn when to allow data to enter, leave or be deleted through the iterative process of making guesses, backpropagating error, and adjusting weights via gradient descent [33].

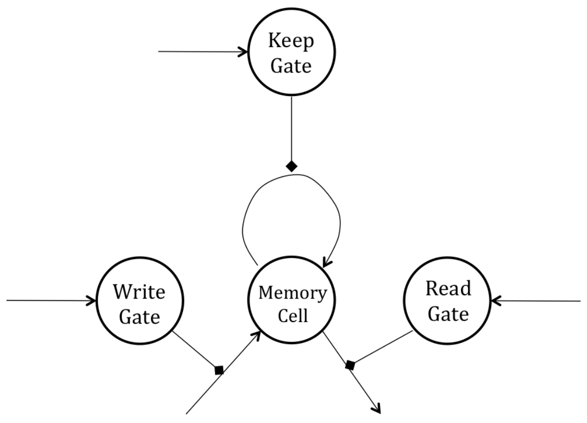


Figure 2.7: Gates to control the cell state [6]

Long Short-Term Memory are explicitly designed to overcome the long-term dependency problem. Their objective is to remember information for long periods of time and we are going to see in more detail how they achieve that. While RNNs only have a single tanh layer insider their modules, there are four as can be seen in image 2.8.

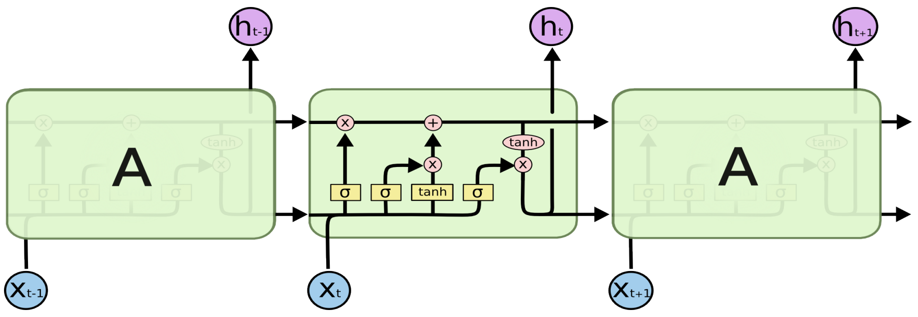


Figure 2.8: LSTM structure inside each module [5]

In the above diagram, each line carries an entire vector, from the output of one node to the inputs of others. The pink circles represent pointwise operations, like vector addition, while the yellow boxes are learned neural network layers. Lines merging denote concatenation, while a line forking denote its content being copied and the copies going to different locations. For a visual representation we can observe image 2.9

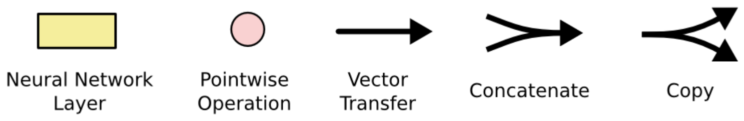


Figure 2.9: Notation used on the images describing LSTM structure [5]

The principal element of LSTM is the horizontal line running through the top of the diagram 2.10. This line is responsible for letting information just flow along it. Yet, this information can be changed, in particularly removed or added to the cell state, carefully regulated by structures we referenced before: the cell gates. This gates are responsible for blocking or letting the information go through. They are composed by a sigmoid neural net layer and a pointwise multiplication operation. The sigmoid layer can have output values between zero and one, where a value of zero means to block all the information/let nothing through and a value of one means to let the information pass the gate. As it was stated before, each LSTM has three gates, one to read, one to write and one to keep the information in the cell.

The first step in a cell is to decide what information it is going to be thrown away from the cell. This decision is made by the sigmoid layer called

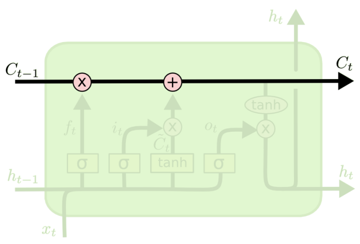
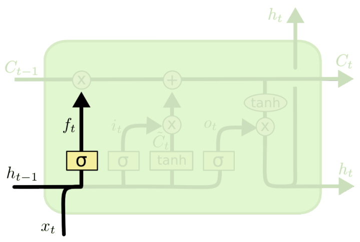


Figure 2.10: Line responsible for letting the information flow inside the cell [5]

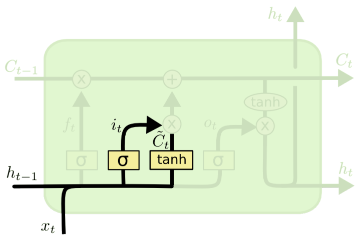
forget gate. It looks at **ht-1** and **xt**, and outputs a number between 0 and 1 for each number in the cell state **Ct-1**. The output 1 represents completely keep this while 0 represents"completely get rid of this 2.11.



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

Figure 2.11: Forget gate [5]

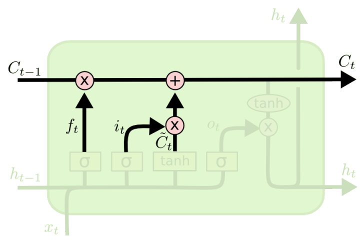
After deciding what information to forget, it is necessary to decide what new information is going to be stored in the cell. There are two parts in this step. First, a sigmoid layer called **input gate** decides which values will be updated. Second, a **tanh** layer creates a vector of new candidate values, $\tilde{\mathrm{C}}_\mathrm{t}$, that could be added to the state 2.12.



$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \ + \ b_i\right)$$
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \ + \ b_C)$$

Figure 2.12: Input gate [5]

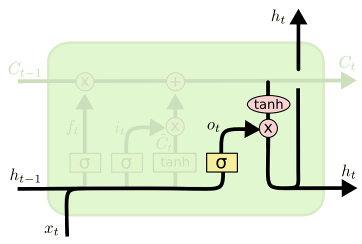
Now it is necessary to update the old cell state, **Ct-1**, into the new cell state **Ct**. The previous steps at the gates already decided what is necessary to do, now it is just necessary to actually do it. The old state is multiplied by **ft**, forgetting the things that were decided to forget. Then $\mathrm{i}_\mathrm{t}*\tilde{\mathrm{C}}_\mathrm{t}$ is added and this is the new value, scaled by how much we decided to update each state value 2.13.



$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

Figure 2.13: Change the values [5]

Finally, it is necessary to decide what the cell is going to output. The output will be based on the cell state, but will be a filtered version. First, it is run a sigmoid layer which decides what parts of the cell state are going to output. Then, the cell state is put through tanh (to push the values to be between -1 and 1) and multiply it by the output of the sigmoid gate, so that it only outputs the parts that were decided to 2.14.



$$o_t = \sigma\left(W_o\ [h_{t-1}, x_t]\ +\ b_o\right)$$
$$h_t = o_t * \tanh\left(C_t\right)$$

Figure 2.14: Output values [5]

There is a variant of LSTM neural networks, the Bidirectional Long Short-Term Memory (BI-LSTM). BI-LSTM simultaneously models each sequence in both the forward and backward directions while traditional LSTM only models in one direction. This enables a richer representation of data, since each token's encoding contains context information from the past and the future. The first LSTM learns the effect of previous words and the second learns the effect of future words. Since this type o network takes the context of the information in consideration by having context from the past and the present, it is very useful in NER tasks related to word classification/prediction. In the figure 2.15 it is possible to see the structure of a BI-LSTM network and how they have information propagation in both forward and backward directions in order for the network to gain past and future information about each word.

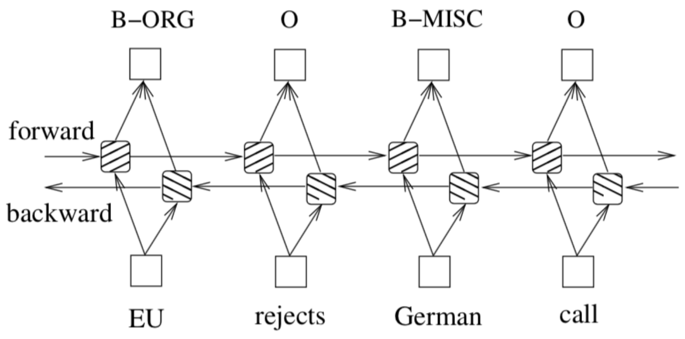


Figure 2.15: BI-LSTM structure [7]

Convolutional Neural Network

CNNs are a category of Neural Networks that have proven to be very effective in areas related to Computer Vision (CV), more specifically image recognition and classification. CNNs have been successful in identifying all types of objects, like faces and animals, and distinguish objects apart like traffic signals[34]. CNNs are just like the other Neural Networks. They are made up of neurons with learnable weights and biases. Each neuron receives several inputs, takes a weighted sum over them, pass it through an activation function and responds with an output. The whole network has a loss function [35]. CNN its a deep, feed-forward Artificial Neural Networks because information flows right through the model and there are no feedback connections in which outputs of the model are fed back into itself[36]. Each neuron in a layer receives input from a neighborhood of the neurons in the previous layer. Those neighborhoods, or local receptive fields, allow CNNs to recognize more and more complex patterns in a hierarchical way, by combining lower-level, elementary features into higher-level features. This property is called **compositionality**. For instance, edges can be inferred from raw pixels, edges can in turn be used to detect simple shapes, and finally shapes can be used to recognize objects[37]. Furthermore, the absolute positions of the features in the image do not matter. Only capturing their respective positions is useful for composing higher-level patterns. So, the model should be able to detect a feature regardless of its position in the image. This property is called **local invariance**. Both compositionality and local invariance are the two key concepts of CNNs.

CNNs operate over volumes. Unlike some neural networks, where the input is a vector, in CNNs the input is a multi-channeled image(3 channels). The inputs are related to the main building block of a CNN: **the convolution layer**. It computes the output of the neurons that are connected to local regions or receptive fields in the input, each computing a dot product between their weights and a small receptive field to which they are connected to in the input volume. Each computation leads to extraction of a feature map from the input image. As example, imagine having an image represented as a 5x5 matrix of values, and to take a 3x3 matrix and slide that 3x3 window or kernel around the image. At each position of that matrix, we multiply the values of your 3x3 window by the values in the image that are currently being covered by the window. As a result, we get a single number that represents all the values in that window of the images. This layer is used to filter: as the window moves over the image, it checks for patterns in that section of the image. This works because of filters, which are multiplied by the values outputted by the convolution[36]. **Pooling or Sub Sampling** is another important block in CNNs. The objective of subsampling is to get an input representation by reducing its dimensions, which helps in reducing overfitting. One of the techniques of subsampling is max pooling. With this technique, we select the highest pixel value from a region depending on its size. In other words, max pooling takes the largest value from the window of the image currently covered by the kernel. For example, we can have a max-pooling layer of size 2 x 2 and it will select the maximum pixel intensity value from 2 x 2 region. Finally, the **classification** is the last thing to be done and the objective is to flatten the high-level features that are learned by convolutional layers and combine all the features. It passes the flattened output to the output layer where it is used a softmax classifier or a sigmoid to predict the output classification label [36].

CNNs are mostly common in Image Classification but they also have a good performance when it comes to Neuro-Linguistic Programming (NLP) tasks. Instead of image pixels, the input for this type of tasks are sentences or documents represented as matrix[8] since the input in CNNs is a multi-channeled image(3 channels). Each row of the matrix corresponds to one token, typically a word, or a character. Each row is a vector that represents a word like in the figure 2.16. This vectors can be **word embeddings** like word2vec or Glove, but they can also be **one-hot vectors** that index the word into a vocabulary[8]. For example, for a sentence with 10 words using a 100-dimensional embedding the matrix would be 10x100.

2.2.3 Word embeddings

Neural networks do not recognize any input that is non-numerical such as words. In order to overcome this particularity it is necessary to convert words to vectors of values. However, it is not possible to convert to a random value. If words are treated as discrete atomic symbols, they will provide no useful information to the model regarding the relationships that exist between some of the words [38]. Using vector representations it is possible to overcome some of these obstacles.

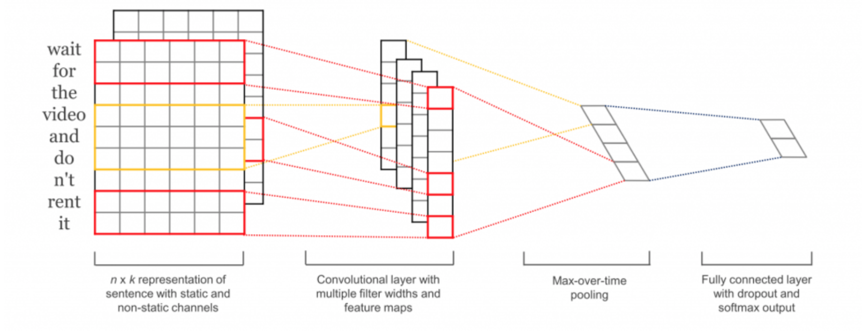

wait
for
the
video
and
do
n't
rent
it
n x k representation of sentence with static and non-static channels
Convolutional layer with multiple filter widths and feature maps
Max-over-time pooling
Fully connected layer with dropout and softmax output

Figure 2.16: Words separed by rows in CNN model [8]

Vector Space Models (VSM) represent words in a continuous vector space where semantically similar words are mapped to nearby points (embedded nearby each other). The different approaches that leverage this principle can be divided into two categories: *count-based* methods and *predictive* methods. Succinctly, *count-based* methods compute the statistics of how often some word co-occurs with its neighbor words in a large text corpus, and then map these count-statistics down to a small, dense vector for each word. *Predictive* models directly try to predict a word from its neighbors in terms of learned small, dense embedding vectors (considered parameters of the model). Two of the most popular word embedding models are part of this categories. **GloVe** [15] is a *count-based* method as described in the article: "Training is performed on aggregated global word-word co-occurrence statistics from a corpus, and the resulting representations showcase interesting linear substructures of the word vector space". On the other hand, **word2vec** is a *predictive model*. The **Bio NLPlab** is a word2vec approach were the vectors were induced from PubMed and PMC texts and their combination using the word2vec tool [14].

Word2vec is a particularly computationally-efficient predictive model for learning word embeddings from raw text. It comes in two flavors, the Continuous Bag-of-Words model (CBOW) and the Skip-Gram model. Algorithmically, these models are similar, except that CBOW predicts target words (e.g. 'mat') from source context words ('the cat sits on the'), while the skip-gram does the inverse and predicts source context-words from the target words. This inversion might seem like an arbitrary choice, but statistically it has the effect that CBOW smoothes over a lot of the distributional information (by treating an entire context as one observation).

GloVe focus on the statistics of word occurrences in a corpus. Count-based models learn their vectors by essentially doing dimensionality reduction on the co-occurrence counts matrix. First it is construct a large matrix of (words x context) co-occurrence information. For each word/row it is counted how frequently the word is saw in some context/column in a large corpus. In the specific case of GloVe, the counts matrix is preprocessed by normalizing the counts and log-smoothing them.

2.3 Deep learning for biomedical NER

In this section we are going to describe and comment on four articles that talk about DL for biomedical NER. These articles are related to the use of neural networks to perform NER tasks and they explain what architecture was choosen and why. We also are going to talk about the tmVar[10] work and how it was done in order to understand how our results stand in comparison to the ones of the tmVar. The first article compares results between CNNs and RNNs neural networks. The objective is to compare the performance of both models in a serie of NLP tasks and to draw conclusions about when each model should be used and to what type of task. The second article is only about the performance of RNNs, in particular different types of LSTMs: there are LSTM, BI-LSTM and BI-LSTM-CRF models. Each model has its own characteristics and the objective is to understand which model performs well. The third article is about the CHEMDNER challenge [39] and the NER systems created to train on this dataset. The fourth article is about tmVar [10], a text-mining approach based on conditional random field (CRF) for extracting awide range of sequence variants described at protein, DNA and RNA levels(genetic mutations) according to a standard nomenclature developed by the Human Genome Variation Society. The tmVar dataset is also the dataset used in our models.

Comparison between CNNs and RNNs for NLP tasks

The first article[9] focuses on the differences between CNNs and RNNs, where the authors decided to compare both approaches in a various number of NLP tasks. CNNs are hierarchical and RNNs are sequential architectures. Since, by characterization one is hierarchical and another is sequential, it is tempting to choose CNNs for classification tasks like sentiment classification since sentiment is usually determined by some key phrases; and to choose RNNs for a sequence modeling task like language modeling as it requires flexible modeling of context dependencies. Yet, the current work in NLP literature does not support this clear division to choose one of the models. For example, RNNs perform well on document-level sentiment classification [40]; and this work [41] recently showed that gated CNNs outperform LSTMs on language modeling tasks, even though LSTMs had long been seen as better suited. In summary, there is no consensus in which type of deep neural network should be choosen for any particular NLP problem.

The three architectures tested were the following: one CNN, one LSTM and one Gated Recurrent Unit (GRU). GRU is a variant of LSTM that retains its resistance to the vanishing gradient problem, but the internal structure is simpler, and therefore is faster to train, since fewer computations are needed to make updates to its hidden states. The NLP tasks to be executed are the following: sentiment/relation classification, textual entailment, answer selection, question-relation matching in Freebase, Freebase path query answering and part-of-speech tagging. The three architectures can be seen in the following image 2.17.

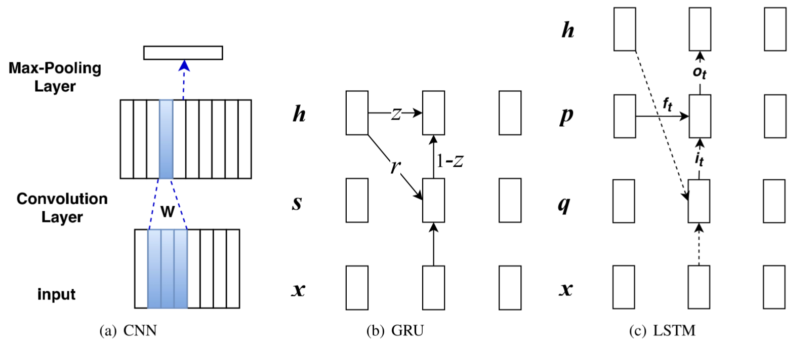


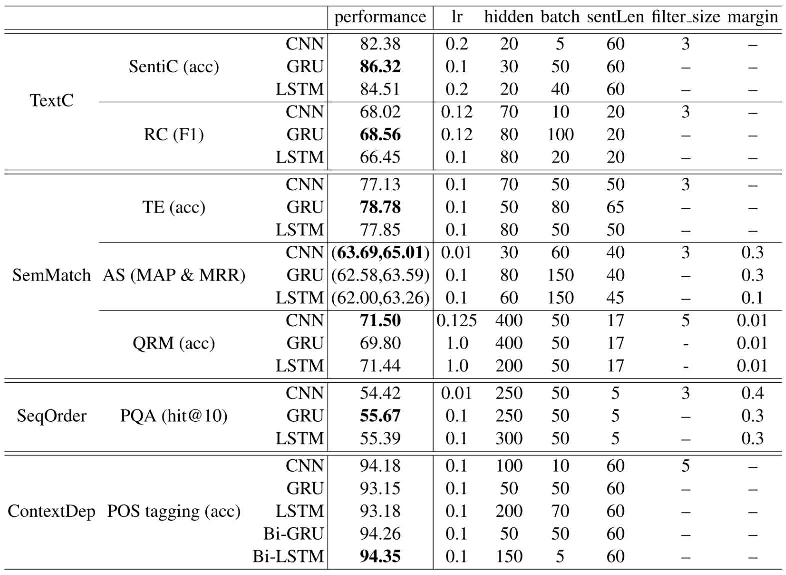
Figure 2.17: Three architectures in the Wenpeng's work [9]

There were seven tasks that served as comparison between the architectures:

- Sentiment classification task(SentiC) on Stanford Sentiment Treebank (SST)[42]
- Relation classification(RC) on SemEval 2010 task 8[43]
- Textual Entailment(TE) on Stanford Natural Language Inference (SNLI)[44]
- Answer selection (AS) on WikiQA[45]
- Question relation match (QRM) with WebQSP
- Path query answering (PQA) on the path query dataset released by [46]
- Part-of-speech tagging on WSJ

The tasks were organized in four categories. (i) **TextC**. Text classification, including SentiC and RC. (ii) **SemMatch** including TE, AS and QRM. (iii) **SeqOrder**. Sequence order, i.e., PQA. (iv) **ContextDep**. Context dependency, i.e., POS tagging. By investigating these four categories, the authors aimed to discover some basic principles involved in utilizing CNNs and RNNs. In order for this study to be fair, considering that the three architectures have different aspects and work in different ways, the experiments developed by the authors have the following desing: **(i)** Always train from scratch, no extra knowledge, no pretrained word embeddings. **(ii)** Always train using a basic setup without complex tricks such as batch normalization. **(iii)** Search for optimal hyperparameters for each task and each model separately, so that all results are based on optimal hyperparameters. **(iv)** Investigate the basic architecture and utilization of each model: **CNN** consists of a convolution layer and a max-pooling layer; **GRU** and **LSTM** model the input from left to right and always use the last hidden state as the final representation of the input. An exception is for POS tagging, were the authors also report bi-directional **RNNs** as this can make sure each word's representation can encode the words context of both sides, like the **CNN** does. **Hyperparameters** are tuned on dev: hidden size, minibatch size, learning rate, maximal sentence length, filter size (for CNN only) and margin in ranking loss in AS, QRM and PQA tasks.

The following table 2.18 shows the experimental results for all the tasks, models and corresponding hyperparameters. For **TextC**, *GRU* performs best on SentiC and similar to CNN in RC. For **SemMatch**, *CNN* performs best on AS and QRM, while *GRU* and *LSTM* outperforms CNN on TE. For **SeqOrder** (PQA), both *GRU* and *LSTM* outperform *CNN*. For **ContextDep** (POS tagging), *CNN* outperforms one-directional *RNNs*, but lags behind bi-directional *RNNs*.



			performance	lr	hidden	batch	sentLen	filter_size	margin
TextC	SentiC (acc)	CNN	82.38	0.2	20	5	60	3	–
		GRU	**86.32**	0.1	30	50	60	–	–
		LSTM	84.51	0.2	20	40	60	–	–
	RC (F1)	CNN	68.02	0.12	70	10	20	3	–
		GRU	**68.56**	0.12	80	100	20	–	–
		LSTM	66.45	0.1	80	20	20	–	–
SemMatch	TE (acc)	CNN	77.13	0.1	70	50	50	3	–
		GRU	**78.78**	0.1	50	80	65	–	–
		LSTM	77.85	0.1	80	50	50	–	–
	AS (MAP & MRR)	CNN	(**63.69,65.01**)	0.01	30	60	40	3	0.3
		GRU	(62.58,63.59)	0.1	80	150	40	–	0.3
		LSTM	(62.00,63.26)	0.1	60	150	45	–	0.1
	QRM (acc)	CNN	**71.50**	0.125	400	50	17	5	0.01
		GRU	69.80	1.0	400	50	17	-	0.01
		LSTM	71.44	1.0	200	50	17	-	0.01
SeqOrder	PQA (hit@10)	CNN	54.42	0.01	250	50	5	3	0.4
		GRU	**55.67**	0.1	250	50	5	–	0.3
		LSTM	55.39	0.1	300	50	5	–	0.3
ContextDep	POS tagging (acc)	CNN	94.18	0.1	100	10	60	5	–
		GRU	93.15	0.1	50	50	60	–	–
		LSTM	93.18	0.1	200	70	60	–	–
		Bi-GRU	94.26	0.1	50	50	60	–	–
		Bi-LSTM	**94.35**	0.1	150	5	60	–	–

Figure 2.18: Best results or CNN, GRU and LSTM in NLP tasks [9]

This work pretended to compare the three most used deep neural networks(CNN, GRU and LSTM) in some NLP tasks. The authors found that RNNs perform well and are robust in a broad range of tasks except when the task is essentially a keyphrase recognition as in some sentiment detection and question-answer matching settings. In addition, hidden size and batch size can make neural networls performance vary dramatically. This suggests that optimization of these two parameters is crucial to good performance of both CNNs and RNNs. The results achieved with RNNs were satisfactory and led us to choose the use of this type of architecture in our work, yet it is also possible to see that CNNs achieve good perfomances and have close results to the RNNs which indicates that they can also be used for text classification tasks.

Bidirectional LSTM-CRF Models for Sequence Tagging

In the second article[7] the objective is to analyze different neural network based models to sequence tagging task(like our own task). In this case the authors decided to use RNN and tested different models to analyze the results: LSTM networks; BI-LSTM networks; LSTM networks with a CRF layer (LSTM-CRF), **4)** and BI-LSTM networks with a CRF layer (BI-LSTM-CRF).

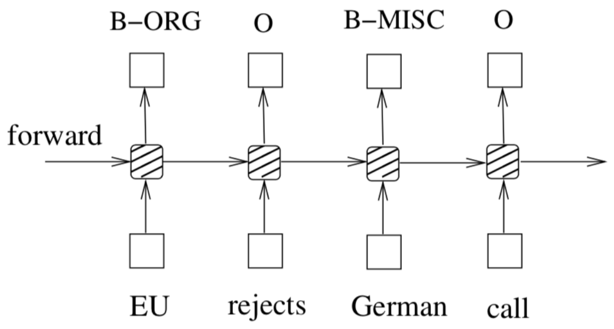


Figure 2.19: Example of lstm [7]

This article is very important because it gives a lot of insight about neural networks composed by all types of LSTM. Like it has been said before in this thesis, RNN have been producing promising results in a great variety of tasks, but more concrete in tasks including language models and speech recognition. LSTM allows the network to predict the current output making use of past features because of its **forward states** as seen in 2.19. Yet, the use of BI-LSTM allows the network to efficiently predict a result basing on both past features and future features(**backward states**) as seen in 2.20.

It is possible to observe in both pictures the major differences between

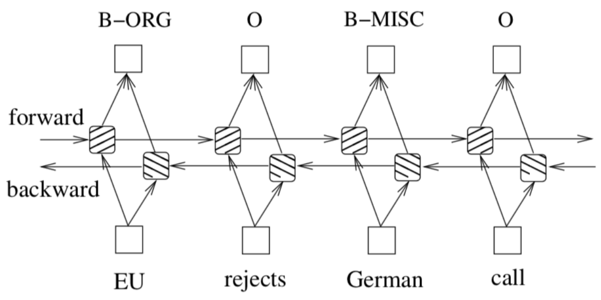


Figure 2.20: Example of bi-lstm [7]

both models: 1) a LSTM model will only forward the information and it is only possible to predict the output based on past features. 2) in a BI-LSTM model the information flows forward and backwards, allowing the model to predict the output based on past and future features. Theoretically this makes the BI-LSTM a better model than just an LSTM. In addition to a BI-LSTM model it is more and more standard to see the use of a Conditional Random Fields (CRF) layer. CRFs are a class of Statistical Model very used in pattern recognition. While another classifiers predict a label for a single sample without any consideration for its "neighbors", a CRF takes the context of the whole samples into consideration. For this reason its use became very widespread in Natural Language Processing (NLP) to predict sequences of labels for sequences of input samples. In the figure 2.21 it is possible to see that it is a BI-LSTM network with a CRF layer at the top. This CRF layer exists to predict the label of each input sample, but each output sample is connected, which means that the CRF layers takes the whole context into consideration when classifying a sample.

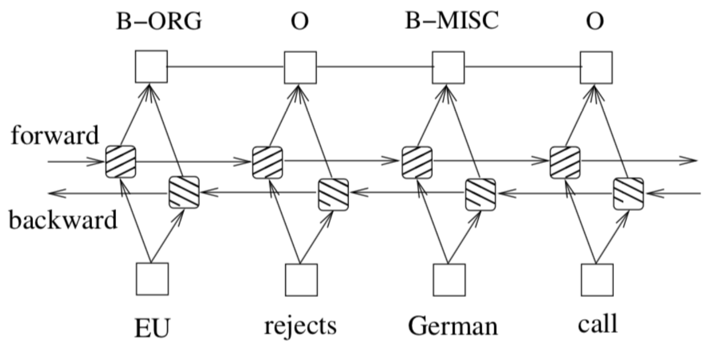
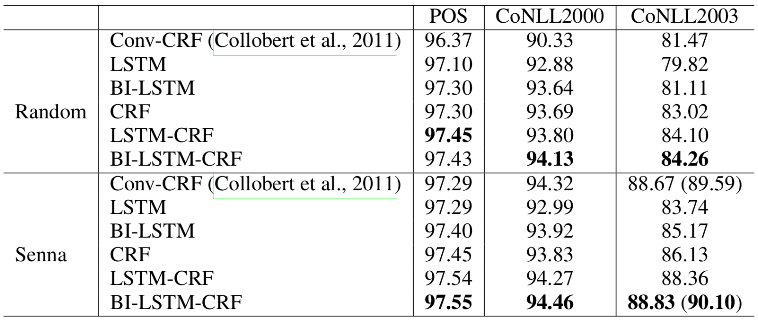


Figure 2.21: Example of bi-lstm-crf [7]

The datasets used to train the models were the following: Penn Tree-Bank (PTB) POS tagging, CoNLL 2000 chunking, and CoNLL 2003 named entity tagging. In order to use word embedding, the authors used Senna embedding[47] which has 130K vocabulary size and each word corresponds to a 50-dimensional embedding vector. They initialized the word embeddings in two different ways: *Random* and *Senna*. The feature sets are identical for each way, which means that the results will be solely dependent on the different networks, which help us to understand which LSTM architecture is better.

It is possible to analyze the results in the following figure 2.22: 1) LSTM is the weakest baseline for all three data sets. 2) LSTM-CRF models outperform CRF models for all data sets in both random and senna categories. This shows the effectiveness of the forward state LSTM component in modeling sequence data. 3) The BI-LSTM-CRF models further improve LSTM-CRF models and they lead to the best tagging performance for all cases except for POS data at random category. This results lead us to believe that **BI-LSTM-CRF** is the best architecture when dealing with RNNs.



		POS	CoNLL2000	CoNLL2003
Random	Conv-CRF (Collobert et al., 2011)	96.37	90.33	81.47
	LSTM	97.10	92.88	79.82
	BI-LSTM	97.30	93.64	81.11
	CRF	97.30	93.69	83.02
	LSTM-CRF	**97.45**	93.80	84.10
	BI-LSTM-CRF	97.43	**94.13**	**84.26**
Senna	Conv-CRF (Collobert et al., 2011)	97.29	94.32	88.67 (89.59)
	LSTM	97.29	92.99	83.74
	BI-LSTM	97.40	93.92	85.17
	CRF	97.45	93.83	86.13
	LSTM-CRF	97.54	94.27	88.36
	BI-LSTM-CRF	**97.55**	**94.46**	**88.83** (**90.10**)

Figure 2.22: Performance [7]

Putting hands to rest: efficient deep CNN-RNN architecture for chemical named entity recognition with no hand-crafted rules

This article[48] is about a work closer to ours. The authors know that most systems that perform NER tasks rely on hand-crafted rules or curated databases for data preprocessing, feature extraction and output postprocessing even if modern machine learning algorithms, such as deep neural networks, can automatically design the rules with little to none human intervention. Following this thinking, the authors created a model based on a combination of convolutional and stateful recurrent neural netwroks, without manually asserted rules, to perform NER tasks.

To facilitate the development of new and superior NER systems, BioCreative announced the CHEMDNER challenge[39], which ended in 2015. To create this dataset a team of experts has produced an extensive manually annotated corpus covering various chemical entity types, including systematic and trivial names, abbreviations and identifiers, formulae and phrases. Due to many difficulties inherent to chemical entity detection and normalisation, manual annotation yields the inter-annotator agreement score of 91%, which is regarded as the theoretical limit for any system trained on this corpus(the work here presented achieve results near this level). The dataset is the **CHEMDNER** corpus and it contains ten thousand abstracts from eleven chemistry-related fields of science with over 84k manually annotated chemical entities (20k unique) of eight types:

- ABBREVIATION (15.55%)
- FAMILY (14.15%)
- FORMULA (14.26%)
- IDENTIFIER (2.16%)
- MULTIPLE (0.70%)
- SYSTEMATIC (22.69%)
- TRIVIAL (30.36%)
- NO CLASS (0.13%)

The authors decided to use three types of neural networks: 1) One-dimensional (1D) convolutional neural networks; 2) recurrent neural networks; 3) time-distributed dense (fully-connected) networks. The 1D CNNs are trainable feature extractors applied along a sequence evolving in time. The RNNs were used because they are highly powerful trainable state machines theoretically capable of modelling relationships of arbitrary depth to process CNN extracted features. In order to improve performance, it is common to use bidirectional RNNs that process sequences in both directions(forward and backwards). Finally, the use of a time-distributed fully connected network with the sigmoid activation function to generate label probabilities.

The authors have two ways to treat the data that influencied our work: the first is the most natural solution and implies grouping and encoding (i.e. representing as numeric tensors) equally sized texts together; the second uses zero-padding (artificially increasing length by appending zeros to numerically encoded sequences) but this method is less flexible, because it introduces a sentence length limit and requires a sentence segmentation model. In terms of tokenization the authors cleary indicate how important this task is. Yet, tokenization can introduce severe merged/overlapping entities and it is very important to use an adequate tokenizer with rules finely adjusted for the task at hand. After having the model structured the authors needed to take care of the most important aspect in NLP tasks: tokenization. Tokenization can introduce severe merged/overlapping entities and it is very important to use an adequate tokenizer with rules finely adjusted for the task at hand. There are two major groups of token encoding strategies: morphology aware (character-level) and unaware (word-level). While word-level encodings are efficient for morphologically rigd corpora (e.g. standard English texts), morphologically rich biomedical and chemical literature introduces many infrequent words and word-forms, resulting in high out-of-vocabulary (OOV) rates. Consequently, most CHEMNDER participants have additionally (or exclusively) used morphology aware-encodings, targeting various manually designed character-level features. This happens because it is very hard to create an optimal tokeniser equally adequate for recovering standard vocabulary and diverse chemical entities, since they have different underlying morphology - a tokenizer has to be context-aware. Because of this, the authors developed their own model based on a "break and stitch" strategy: a primary extra-fine segmentation followed by a refinement step trying to recover target entities.

The models trained by the authors had two input nodes: one for pre-trained word-level embeddings and another for encoded token strings. The strings were encoded as integer vectors containing character identifiers. They trained 300-dimensional Glove embeddings with default configurations on a corpus of random PubMed abstracts from the same categories as the CHEMDNER abstracts. Character-level embeddings were optimised during training. It consists of a trainable linear character-embedding layer transforming vectors of character codes into matrices of 32-dimensional character embeddings. These word matrices are then processed by a standard biGRU (16 cells) layer producing a 32-dimensional vector per token. Instead of concatenating word- and character-level embeddings before feeding them into a single CNN or RNN block, the authors used two separated layers of deep 1D CNNs for each embedding type to increase the number of degrees of freedom without using too many convolutional filters. Features extracted by these independent blocks were subsequently concatenated and fed into a two-layers deep HS-biGRU. At the output, the labelling method resembles the widely used *IOB* scheme with three mutually exclusive labels: Inside/Outside/ Beginning (of an entity).

The networks were trained for 40 epochs with a callback saving weights upon improvements in performance on the validation dataset. During testing, the CHEMDNER chemical entity mention (CEM) subtask was the target. The results from the tokenizer were very satisfactory: out of 25347 annotated entities in the testing dataset less than 0.19% spanned the same token; at the same time the tokeniser had a recall of 91.75% and precision of 93.32%. Therefore, it was able to accurately recover most of the annotated entities. In terms of performance GRUs trained and converged faster and showed slightly better performance on the testing dataset than LSTMs.

Convolutional layers were crucial for good performance. On average, replacing the CNN-layers with one or two hs-biGRU layers reduced the F-score by 1.5-2.3% and hampered the training process. On the CHEMDNER CEM subtask the fully-featured network has gained the F-score of 88.7%. Therefore, it outperforms all models submitted for the CHEMDNER task by a significant margin, though the edge over ChemDataExtractor is less impressive as it is possible to see in 2.1. Considering the inter-annotator agreement score of 91%, the model demonstrates near human performance.

Model	**Precision %**	**Recall %**	**F1-Score %**
model in study	88.6	88.8	88.7
ChemDataExtractor	89.1	86.6	87.8
tmChem(173)	89.2	85.8	87.4
(231)	89.1	85.2	87.1
LeadMine(179)	88.7	85.1	86.9
(184)	92.7	81.2	86.6
Chemspot(198)	91.2	82.3	86.7
Becas(197)	86.5	85.7	86.1
(192)	89.4	81.1	85.1
BANNER-CHEMDNER(233)	88.7	81.2	84.8
(185)	84.5	80.1	82.2

Table 2.1: CHEMDNER challenge team IDs are given in parenthesis in the Model column (where available). ChemDataExtractor performance scores reported by the authors

The model presented in this article for chemical NER in biomedical texts was trained and evaluated on the CHEMDNER corpus and achieved high values in performance, proving that chemical NER can be done efficiently with no manually created rules or curated databases whatsoever. Yet, the authors advocate the use of specialised trainable tokenizers to make sure that all the words and word-forms are found since the biomedical and chemical literature is morphologically very rich, which results in high out-of-vocabulary rates. In particular, the use of specialised trainable tokenizers and stateful recurrent neural networks.

tmVar: a text mining approach for extracting sequence variants in biomedical literature

tmVar is a text-mining approach based on Conditional Random Fields for extracting a wide range of sequence variants described at protein, DNA and

RNA levels(genetic mutations) according to a standard nomenclature developed by the Human Genome Variation Society (HGVS). One of the most important research issues is gene/protein and disease relationship analysis. Sequence variation plays the key role between gene and disease. Therefore, identifying sequence variation is one of the major approaches for characterizing gene–disease relationships, with many study results subsequently reported in scientific publications. As such, text mining mutation-related information from the literature has become an increasingly important task in many downstream bioinformatics applications. Despite some reported success in identifying specific mutation types or identifiers, such as dbSNP RS numbers, mutation identification from free text in general remains a challenge because most mutations are not described in accordance with standard nomenclature ($<$25%in tmVar corpus) and only few are mentioned with standard database identifiers, such as dbSNP RS numbers ($<$10% in tmVar corpus). To the opposite, it is common to see the same mutation described in many different non-standard ways in the literature. Despite different scopes, with regard to methods for mutation detection, most systems rely on manually derived regular expressions. For instance, for detecting protein point mutations (e.g. **A42G**) from text, MutationFinder [49] was developed, which contains $>$700 regular expression patterns and achieves state-of-the-art performance of 90% in F-measure.

TmVar is unique in extracting mutations of many types that are not considered by previous methods. Existing methods, such as MutationFinder either exclusively aim for extracting point mutations in proteins or are limited to a few mutation types, such as substitution and deletion in both proteins and genes. To the knowledge in hand, tmVar is the first attempt to identify various mutation types according to a standard nomenclature endorsed by the HGVS for the description of sequence variants (mutations). Similar to MutationFinder, tmVar also contributes to the text-mining community a large corpus (500 PubMed abstracts) of manually annotated raw and normalized mutation mentions. A raw mutation extraction is normalized when individual mutation components are identified and standardized when applicable. For instance, *'Arg987Ter'* (PMID: 22188495) is normalized as *'pjRj987jX'* to denote the replacement of an arginine residue at position 573 by termination codon, where a single letter 'p' is added to indicate the mutation type, and the standard one-letter codes are used (with their respective positions in the normalized notation) to represent the wild-type and mutant residue. The tmVar corpus covers many kinds of mutations not previously considered, such as *'p.Pro246HisfsX13'* (PMID: 21738389) and *'IVS3+1G/A'* (PMID: 15111599).

As shown in figure 2.23, the system first performs tokenization on the input text as pre-processing. Next, the system extracts mutation mentions from text using a CRF-based approach, followed by some post-processing steps. As illustrated in the figure, instead of extracting a mutation mention such as *c.2708_2711delTTAG* as a whole, the CRF module identifies each mutation component (e.g. 'del' as the mutation type) individually. Finally, it is implemented a post-processing module to handle some rare mutation formulas and nature language mentions that are not curated in the corpus.

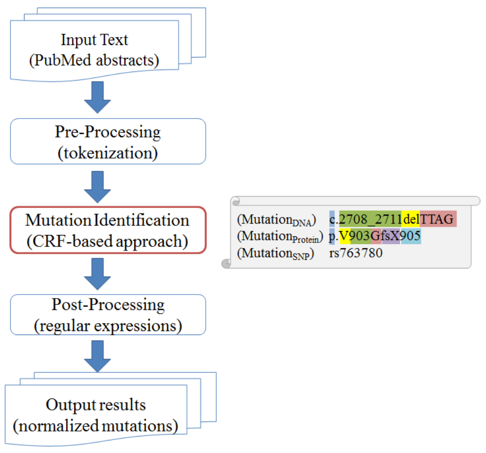


Figure 2.23: The system overview that includes three major components: pre-processing (tokenization), mutation identification (CRF) and post- processing (regular expression patterns) [10]

The pre-processing consists in a tokenizer that divides text input into a sequence of tokens, which generally correspond to 'words'. However, to capture individual components within a mutation mention, it is performed tokenization on a finer level than traditional methods [50] that separate input text by space or punctuation. Special characters (e.g. '-', '*'), numbers, lowercase letters and uppercase letters are divided as separate tokens. For instance, instead of regarding the mention *'c.2708_2711delTTAG'* as one token, it is split into seven pieces. The CRF module that takes care of the mutation identification is the more complex. As forementioned, the mutation identification problem was seen as a sequence-labeling task. In particular, each mutation component was considered as an individual label, such that every mutation mention becomes a sequence of labels. Accordingly, was adapted a probability-based sequence detection CRF model [51], which defines the conditional probability distribution $P(Y|X)$ of label sequence Y given observation sequence X. Unlike the traditional **BIO** labeling models, which labels each token as being the beginning of **(B)**, the inside of **(I)** or entirely outside **(O)** of a span of interest, it is used 10 different labels (fig 2.24) for describing mutation elements (i.e. tokens within the mutation mentions) based on the HGVS nomenclature, and one additional label **(O)** for all tokens outside a mention.

The CRF module contains six different types of features to help achieve better performance: 1) dictionary features; 2) general linguistic features; 3)

Mutation types	A	P	T	W	M	F	S	D	R	I
Substitution	●	●	●	●	●					●
Deletion	●	●	●		●					●
Insertion	●	●	●		●					●
Insertion/deletion	●	●	●		●					●
Duplication	●	●	●	●				●		●
Frame shift	●	●	●	●	●	●	●			●
RS number									●	

Figure 2.24: The 10 different labels for tokens within mutation mentions: reference sequence (A); mutation position (P); mutation type (T); wild-type (W); mutant (M); frame shift (F); frame shift position (S); duplication time (D); SNP (R); other inside mutation tokens (I) [10]

character features; 4) semantic features; 5) case pattern features; 6) contextual features. Yet, the CRF model still misses a few mentions. To minimize the number of false negatives, the model takes the mentions extracted by the CRF module and translates them into regular expression patterns to find additional mentions of similar kind(post-processing). Two rules were applied to make the translated pattersn more generalizable: **(i)** all numerical digitals become '[0-9]+'; **(ii)** all lowercase and uppercase letters become '[a-z]' and '[A-Z]', respectively, except three special tokens IVS, EX and RS. As a result, '*c.IVS64þ5C4G*' is translated to '[a-z]\.IVS[0-9]+\+[0-9]+[A-Z]\>[A-Z]'.

The methods developed in tmVar were compared to MutationFinder. In addition to the public software, MutationFinder also has a large corpus where both raw mentions and normalized annotations are available, which allowed to perform cross-comparisons between tmVar and MutationFinder on two different gold-stardand datasets. The methods developed in tmVar were compared with MutationFinder for precision, recall and F-measure. MutationFinder was designed exclusively for detecting protein point mutation, yet the table reports its performance on all mutations, as well as just protein point mutations, when using the tmVar test corpus. As such, there are two rows of results in table 2.2 for MutationFinder.

As can been seen in table 2.2 and 2.3, tmVar method achieved consistently higher F-measures than MutationFinder on two independent datasets. On the other hand, when benchmarked on the tmVar corpus, MutationFinder's results (2.2) dropped significantly from the performance on its own corpus (2.3), especially in recall, even though the evaluation was limited to its extraction scope (protein point mutation). The results achieved at tmVar suggest that it is a high-performance method for mutation extraction from biomedical literature.

The authors created a CRF-based machine-learning method for mutation extraction from biomedical text with high performance. The method complements and extends existing methods in extracting a wide range of dif-

	Methods	P(%)	R(%)	F(%)
All mutations	MutationFinder	91.66	33.21	48.76
	MutationFinder[a]	89.66	69.15	78.08
	tmVar	**91.38**	**91.40**	**91.39**
Normalized mutations	MutationFinder	84.21	25.29	38.90
	MutationFinder[a]	84.09	63.25	72.20
	tmVar	**87.74**	**87.46**	**87.60**

Table 2.2: Results on the tmVar test set in terms of precision (P), recall (R) and F-measure (F)

	Methods	P(%)	R(%)	F(%)
All mutations	MutationFinder	98.41	81.92	89.41
	tmVar	**98.80**	**89.62**	**93.98**
Normalized mutations	MutationFinder	**98.47**	80.63	88.66
	tmVar	97.58	**83.96**	**90.26**

Table 2.3: Results on theMutationFinder corpus in terms of precision (P), recall (R) and F-measure (F)

ferent types of sequence variants in scientific publications. This work showed how tmVar can compete with another mutations extraction tools like MutationFinder and achieve great performance on PubMed abstracts. The corpus created for this work contained 500 documents that are now at the disposal of the community. In our case, this 500 documents became our dataset and we will compare the results obtained with the ones of tmVar. It is also important to note that the tmVar work contains a CRF approach that is complemented with hand-crafted features and even uses regular expressions as post-processing method to achieve this results while our work will try to rely only on neural networks.

General Conclusions

The first article was important to help us understand the different neural networks architectures and when they should be used. Even if both RNNs and CNNs have good performance in the various task tested, RNNs seem to have advantage on text classification tasks and because of that we decided to go with that architecture even if both can be used as the article stated. The second article compared different types of RNNs architecture based on LSTMs. They tested normal LSTMs, BI-LSTM, LSTMs with CRF layer and BI-LSMT-CRF. The results cleary showed that having a bi-lstm neural network instead of just a lstm one achieves a better performance and should be the used in any sequence classification task. But when adding a CRF layer on top of a bi-lstm neural network the results are even better and were the best in all but one of the tasks tested. This article set our mind in deciding to use a BI-LSTM-CRF model for our task. The third and fourth are articles that talk about works similar to ours and helped us understand the process of creating NER models based on machine learning approaches. On the third article the authors claim that no hand-crafted rules were used in the model and the results are all based in the features extracted by the theirs neural network. The model relies in both CNNs and RNNs and achieve very good results in the challenge they participated showing that neural networks can achieve top performances without hand-crafted rules or hand extracted features. The last article does not contain a machine learning approach, yet the tmVar work created the dataset we are going to use. The fact that they have a CRF approach with features extraction and post-processing is good because we can compare this type of approach with our neural network model free of hand-crafted rules. If our model can achieve a close or better performance than tmVar it is very good because tmVar is one of the best methods for mutation extraction as the results in comparison to Mutation-Finder show.

Chapter 3

Implementation

3.1 Introduction

The problem of this thesis is focused in the recognition of genetic mutations in biomedical corpus. Mutations have a very distinct way of being represented, as happens with most biomedical literature, which leads to being very difficult to create hand-crafted rules to detect them. For this reason, the use of Deep Learning becames important since it excels in pattern recognition, leading to a good performance in detecting mutations with little to none hand-crafted rules.

Our model focus on the creation of Deep Learning models based on characters and word embeddings. **BI-LSTM-CRF** is the type of neural network used to create the models and both have three BI-LSTM layers with a CRF on top. The bi-directional layer allows the network to efficiently predict a result based on both past (forward states) and future features (backward states), and the CRF layer takes the context of the whole results into consideration before predicting a final output for each input sample. Both our models rely only on the neural network to extract features from the data and there are no hand-crafted rules.

3.1.1 Dataset

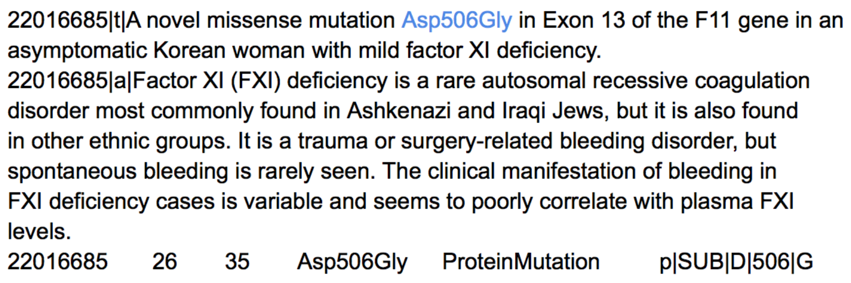
The dataset used was the one from tmVar[10]. The full corpus contains 500 documents and 1410 identified mutations (all other words are considered non-mutations). The dataset already comes divided in train and test dataset: **334** documents and **967** mutations as training dataset, and **166** documents and **464** mutations as test dataset to evalute the model as it is possible to see in 3.1.

As we can see in figure 3.1 the corpus can be divided into two parts. The **first** are the sentences that contains the **title** and **abstract** from the corpus, together with the **id** to facilitate the identification. The **second** part consists in all the genetic mutations that are presented in the sentences, its

Dataset	Nº Docs	Nº Mut
Train	334	967
Test	166	464

Table 3.1: Table with the ratio of documents and mutations

position(offset) in the sentence and what type of mutation it is. As we can see, the mutations format is very distinct to standard words, as they can have letters and digits, or even pontuaction marks, which makes them very difficult to identify just by following semantic rules.


22016685|t|A novel missense mutation Asp506Gly in Exon 13 of the F11 gene in an asymptomatic Korean woman with mild factor XI deficiency.
22016685|a|Factor XI (FXI) deficiency is a rare autosomal recessive coagulation disorder most commonly found in Ashkenazi and Iraqi Jews, but it is also found in other ethnic groups. It is a trauma or surgery-related bleeding disorder, but spontaneous bleeding is rarely seen. The clinical manifestation of bleeding in FXI deficiency cases is variable and seems to poorly correlate with plasma FXI levels.

```
22016685    26    35    Asp506Gly    ProteinMutation    p|SUB|D|506|G
```

Figure 3.1: Example of one document from the corpus(PubMed ID 22016685). The highlighted word corresponds to a mutation mention

The objective is to create two models, one based on character and another on word level embeddings, that can successfully predict all the labels in a sequence of inputs(sentences from the corpus). Both models need preprocessing in the corpus in order to convert it to a proper input that a neural network can recognize. The word model needs a tokenizer to split the corpus of each document into a set of words and word-forms(tokens). Yet, the tokenization can introduce problems like the place of the splitting and it can consider some tokens as not part of words or even introduce severe merged/overlapping entities. If we were dealing with a corpus that only had english words and correct pontuation, a tokenizer could be useful and perform close to perfection tokenization of the corpus. But in this case, the corpus is very complex because of the mutations structure and the morphologically rich biomedical corpus, which is very dificult for a standard tokenizer to properly understand. Tokenization is a very difficult task in corpus as this one and because of that it is very usual to choose models with character level embeddins so that it is not necessary to deal with the tokenization.

The tmVar dataset helps with the mutations labelling because of the offsets presented at each mutation. In the figure 3.2 it is present an example where it is possible to see all the mutations that belong to a document and their attributes like the offset and type.

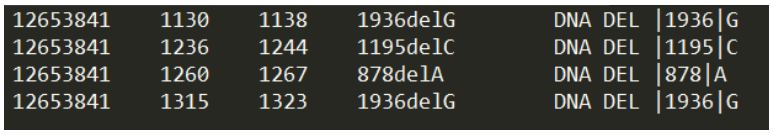

```
12653841    1130    1138    1936delG        DNA DEL |1936|G
12653841    1236    1244    1195delC        DNA DEL |1195|C
12653841    1260    1267    878delA         DNA DEL |878|A
12653841    1315    1323    1936delG        DNA DEL |1936|G
```

Figure 3.2: Example of the labels

The offsets present in the example are the following: (1130-1138; 1236-1244; 1260-1267; 1315-1323). This offsets are the positions of the first and last characters of that mutation in the document and all the characters between this positions belong to that mutation. With this information it is simpler to label all the tokens or characters in the corpus. The tokens/characters between this positions will be tagged as mutations, while the others will be tagged as non-mutations. The characters are correctly marked with the *BIO* tags. The tag *"B"* stands for the first character of the mutation, marking it beginning. The tag *"I"* stands for a character inside the mutation but not the first one. For last, the tag *"O"* stands for a character non-belonging to a mutation. In the tokenization (word model), it is used the *SOBIE* (sometimes known as BIOES) tags, where *"O"* marks a token that is not part of an entity, *"S"* marks a token that is the whole of an entity (a "singleton"), *"B"* marks a token at the beginning of an entity, *"I"* marks one inside an entity, and *"E"* marks one at the end.

3.2 Models

The primary objective of this thesis is the creation of a word and a character based model, in order to compare the performance of both models. The models are very similar because they both have Bidirectional Long Short-Term Memory (BI-LSTM) layers with a Conditional Random Fields (CRF) layer on top like it is described in the figure 3.3. Each cell of the input layer contains a token or a character since this is a sequence labelling problem and it is necessary to classify all the inputs.

In theory it is rather plausible to predict that the character model can present better results since the mutation's syntax and the composition of a biomedical corpus are very complex. To create a good word based model it is necessary to have a very good tokenizer and a good word embedding model. Since the tokenization can be a stressful task, the creation of character based model that uses the characters and their surrounding ones to understand the corpus can be an easier task and can achieve good results in successfully labelling the biomedical corpus.

At first, we used two tokenizers: the Keras Tokenizer [52] and NLTK Tokenizer [53]. The Keras Tokenizer produced better results in tokenizing the

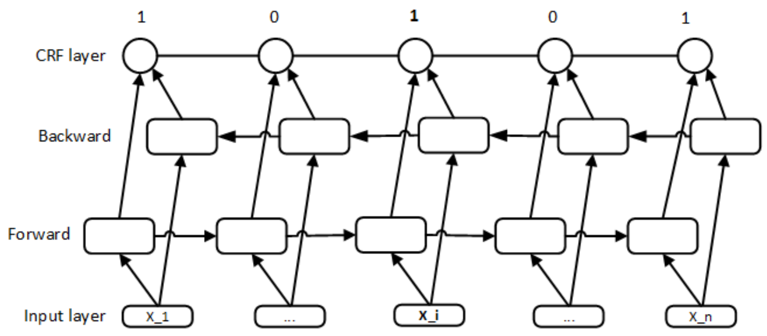
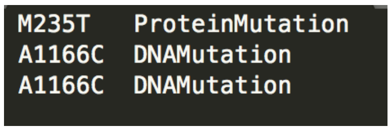


Figure 3.3: Neural Network structure with BI-LSTM and CRF layer on top of it [11]

sentences in order to get the mutations properly splitted into tokens, yet, the accuracy of this tokenizer was very low. From the 1692 total existing mutations, the keras tokenizer only found around 400 mutations, which mean that all other existing mutations would be considered as non-mutations when assigning labels to the tokens. Continuing with this tokenizer would lead to give false negatives to the model, which would be a problem when classifying values. For example, a mutation that the tokenizer failed to isolate, would be considered as a non-mutation, but when the same word would be given by a user to the model it would be incorrectly classified and therefore giving a incorrect prediction to the user. The second tokenizer used was the NLTK Tokenizer with **Keyphrase Extraction**. The idea was that if the tokenizer had the existing mutations as keyphrases that need to be isolated, it would achieve more accuracy. Yet, as stated before, the syntax of mutations is very complex and because of all the special characters it contains, even a tokenizer with keyphrases would achieve low results and not be able to extract all the mutations available in the documents. After finishing the character based model there was some time dedicated to see how other models from different tasks/problems dealt with the tokenization to sucessfully create a word based model and we came in contact with a python version of the **OSCAR4** tokenizer [54]. This tokenizer is very important because the process of tokenization is based on biomedical corpus to deal better with its characteristics like punctuation and combinations of numbers and letters. It also considers the offsets of each mutation in the corpus, and after the tokenization it is able to indicate which tokens were related to the same entity.

The final models in this thesis were done with the Keras library because of its simplicity and being more user friendly. But in a first approach the TensorFlow technology was used in order to gain more insigh about neural networks and how they work since Keras decreases the complexity but also leaves the users with less knowledge about neural networks. The experiments done can be divided in two parts: **1)** in the first part the experiments were made using the TensorFlow (TF) library to create the model; **2)** in the second part the models were done with the Keras library because of its reduced complexity and faster prototyping.

In this project the models done in TF served the purpose of giving more insight about neural networks in order to understand how to create one from scratch and not just connecting layers like if they were legos as it happens with Keras. Some of the small models in TF were done with this corpus but with different tasks than the one of the final model (sequence labelling). The first TF model only trained with a mutation as input and its classification as label. There was no sequence labelling, but only a single input and the corresponding classification as output.

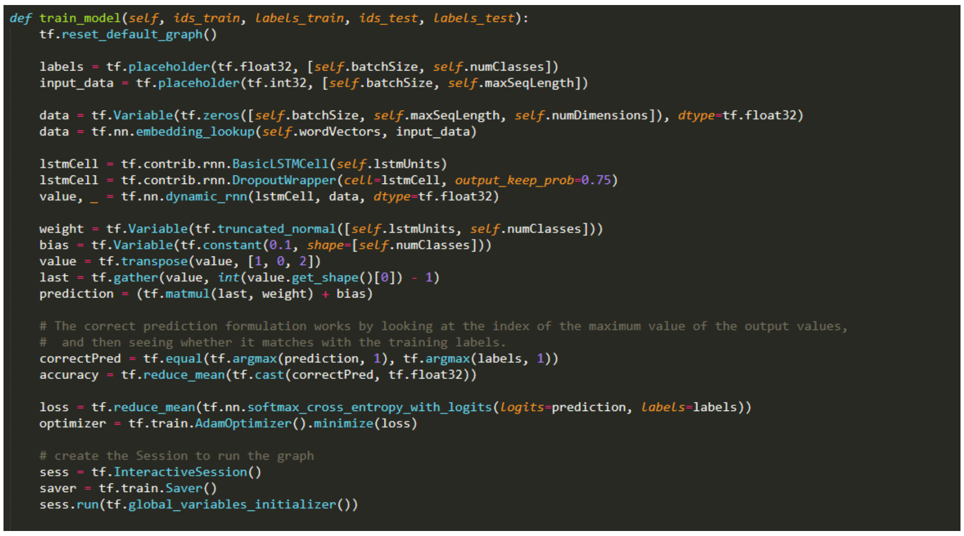


```
M235T   ProteinMutation
A1166C  DNAMutation
A1166C  DNAMutation
```

Figure 3.4: Inputs on first model

This type of model gave more insight in how the data needs to be correctly fetch in each batch and then passed into the model. The data is stored in vectors and matrixes in TF and it is necessary to have a clear understanding of this structures. In the following figure 3.5 it is possible to see the number of lines needed to create a neural network structure. In this figure it is being create an RNN with LSTM as cells. It is also necessary to define the embeddings, weight, accuracy, optimizer while in Keras all this fields are passed as arguments to the fuction that creates the neural network. In Keras the same structure could be defined with just two or three lines. With this example it is easier to understand why Keras offers faster prototyping.

With the knowledge gained from the first project in TF it was possible to start a second project and go a little further in terms of complexity. It was possible to use the whole dataset and the corpus of each document as input together with information saying if that corpus contained mutations or not (1 or 0 to identify if the corpus contained any mutation or not, respectively). After the training phase, when giving a new corpus to the network, it would give as output a '1' or '0' response in case that corpus had a mutation or not. With this project it became very clear that dealing with more complex inputs and building more complex networks was a difficult task for the beginner skills and because of that the technology used changed to the Keras library. Even if the final model is all done in Keras, the use of TF was very important and it gave more insight about neural networks, how they work, what they need and how each block connects to the others. TF also offers more flexibility and total control of the neural network. It is recommended to learn TF because in the long term it becames more powerfull than Keras and offers more. It is also important to gain this little knowledge of how



```python
def train_model(self, ids_train, labels_train, ids_test, labels_test):
    tf.reset_default_graph()

    labels = tf.placeholder(tf.float32, [self.batchSize, self.numClasses])
    input_data = tf.placeholder(tf.int32, [self.batchSize, self.maxSeqLength])

    data = tf.Variable(tf.zeros([self.batchSize, self.maxSeqLength, self.numDimensions]), dtype=tf.float32)
    data = tf.nn.embedding_lookup(self.wordVectors, input_data)

    lstmCell = tf.contrib.rnn.BasicLSTMCell(self.lstmUnits)
    lstmCell = tf.contrib.rnn.DropoutWrapper(cell=lstmCell, output_keep_prob=0.75)
    value, _ = tf.nn.dynamic_rnn(lstmCell, data, dtype=tf.float32)

    weight = tf.Variable(tf.truncated_normal([self.lstmUnits, self.numClasses]))
    bias = tf.Variable(tf.constant(0.1, shape=[self.numClasses]))
    value = tf.transpose(value, [1, 0, 2])
    last = tf.gather(value, int(value.get_shape()[0]) - 1)
    prediction = (tf.matmul(last, weight) + bias)

    # The correct prediction formulation works by looking at the index of the maximum value of the output values,
    #  and then seeing whether it matches with the training labels.
    correctPred = tf.equal(tf.argmax(prediction, 1), tf.argmax(labels, 1))
    accuracy = tf.reduce_mean(tf.cast(correctPred, tf.float32))

    loss = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(logits=prediction, labels=labels))
    optimizer = tf.train.AdamOptimizer().minimize(loss)

    # create the Session to run the graph
    sess = tf.InteractiveSession()
    saver = tf.train.Saver()
    sess.run(tf.global_variables_initializer())
```

Figure 3.5: Model structure with TF

neural networks work because it is easier and better to learn the basics in TF and then switch to Keras to faster and simpler production, than starting with Keras and then being very difficult to make the bridge and understand the complexity of TF.

3.2.1 Character Model

The first step in the creation of a character model is the pre-processing. First it is necessary to read all the corpus and labels; the corpus will go to a dictionary where the key is the document id and the value is the corpus, and the mutations will go to a dictionary where the key is the document id and a list of values contains all the mutations in that document. Then, after having all the corpus and mutations in dictionaries it is necessary to transform the corpus of each document to a list of characters. Since both dictionaries have the doc id as key, it is possible to transform the corpus into a list of characters and at the same time to create a list based on that characters, but with the labels associated. In order to do so, it is necessary to get the position of each mutation in the document, which are stored in the dictionary of mutations. Then, it is necessary to get the list of characters and to create a list of the same size and fill it with *0's*. Since our tags are *BIO*, the *'0'* will stand for a character that does not belong to any mutation. After the list being full, it is necessary to check all the positions of the mutations, and to go to the list and change the values in that positions. The first character of each mutation will always have a *'B'* to indicate that a mutation starts in that character, and the other characters will have a *'I'* to indicate that they are inside the mutation. To a list of characters like this:

[r,e,f, ,t,o, ,A,s,p,5,0,6,G,l,y] the function would return also a list of labels like this:[0,0,0,0,0,0,0,B,I,I,I,I,I,I,I], where *Asp506Gly* is a mutation and the other entities are non-mutations.

The corpus needs to be splited into sentences so that the inputs feed to the network are not to big. After labelling each character, there is an algorithm that fetches the corpus and labels from the same document and will search(in the corpus) for all the sequences of ". ". This sequence indicates the end of a sentence and the beginning of another. The algorithm saves the positions where these sequences happens, and with them it is possible to know all the sentences of the corpus. Then the same positions are used in the labels dictionary and the list is split in the same positons. To finalize, the structure of the dictionaries is changed, instead of having a list of characters as value, the dictionaries will have a list of lists of characters has values, where each list of characters is a sentence of that corpus.

To finalize the pre-processing, both dictionaries are merged into one list of dictionaries. Having all the information in one list make it easier to fetch data into the neural network inputs because at this point, since all the corpus and labels have been treated, there is no need for other attributes like document id. The new list has at each element a new dictionary and this dictionary will have under the field *"corpus"* the list of lists of chars from the corpus, and under the field *"labels"* the list of lists of labels for each character.

The flow of this function can be seen in the following fig 3.6.

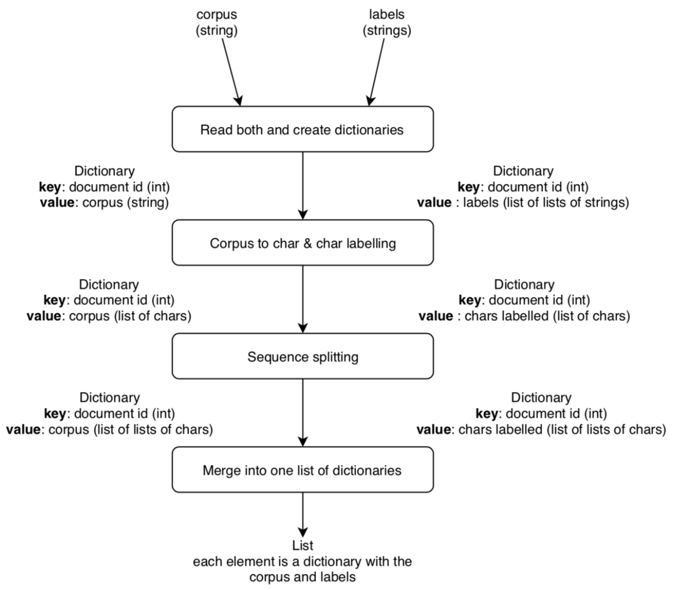

corpus (string)
labels (strings)
Read both and create dictionaries
Dictionary key: document id (int) value: corpus (string)
Dictionary key: document id (int) value : labels (list of lists of strings)
Corpus to char & char labelling
Dictionary key: document id (int) value: corpus (list of chars)
Dictionary key: document id (int) value : chars labelled (list of chars)
Sequence splitting
Dictionary key: document id (int) value: corpus (list of lists of chars)
Dictionary key: document id (int) value: chars labelled (list of lists of chars)
Merge into one list of dictionaries
List each element is a dictionary with the corpus and labels

Figure 3.6: Diagram of the seq_to_char function

Architecture

Now it is time to talk about the model's architecture. After receiving the list of dictionaries with all the information (both corpus and labels in characters) the model can start to organize the data and all the dependencies to feed the model. After receiving the training dataset, it is necessary to take in consideration that the model does not receive characters as inputs. It is necessary to transform each character in a numeric value. To make this transformation our model has an alphabet of all the characters presented in the dataset, which is the following:

```
abcdefghijklmopqrstuvwxyz
ABCDEFGHIJKLMOPQRSTUVWXYZ
0123456789
~ { } [ ] % + . \\ ; > < - \# , ' " = ( ) ? / : _ *
```

It contains all the letters in lower and upper case, all the digits and the symbols here present. It is very important that in such scenario there is a distinction between letters in lower and upper case because mutations have a very particular distinct and complex way of being represented and the use of different case letters in a specific context can be the very useful to identify a mutation. After the alphabet being defined it is very import to assign a number to each character in the alphabet so that the neural network can recognize the inputs. If we were to follow the sequence in the alphabet presented earlier, the character *'a'* would have the value *1*, the character *'b'* would have the value *2*, etc...

After having the alphabet and the numbers attributed to each character, it is time to create the model. The structure have been described in the beginning and consists on one **input layer** that can have a fixed shape (in case where each sentence has the same length) or can have a shape that will change at each input(differente lengths for the sentences). The input layer connects to an **Embedding Layer** that will turn positive integers (the number that each character has been mapped to) into dense vectors of fixed size. The objective is that the embedding layer will map characters that are related to similar regions and with this particularity the neural network can learn the data better. After the embedding layer, the network will have three BI-LSTM layers, the first one with 128 units and the other two with 64 units. Each LSTM returns full sequences between units and not just the last output, the dropout is 0.5 and the merge mode between the three layers is concatenation. After this layers, it is necessary to have a **Time Distributed** layer in order to keep a *many-to-many* relation between the inputs and the outputs. In another words, the objective of this model is to label all the input sequence and in order to do so, it is necessary to have the LSTM layers returning the full sequence of input and to have a Time Distributed layer to make sure that each input will have the correct output presented in the same sequential order. For example, considering the inputs (a1,a2,a3,a4...aN) and the outputs (b1,b2,b3,b4...bN): the Time Distributed layer receives the full sequence from the LSTM layers and guarantee that the outputs will be presented in the same sequential order that the inputs were given to the network. Otherwise, the only output would be the last label "bN". Time Distributed layers are very useful in labelling sequence models. It is possible to see in fig 3.7 the different type of input/output combinations possible for RNNs networks (LSTMs make part of this type of networks as said before). The Time Distributed layer allows us to build models that do the one-to-many and many-to-many architectures. In the case of our model, it is necessary to do the many-to-many architecture. To finish the network it is necessary the use of a CRF layer.

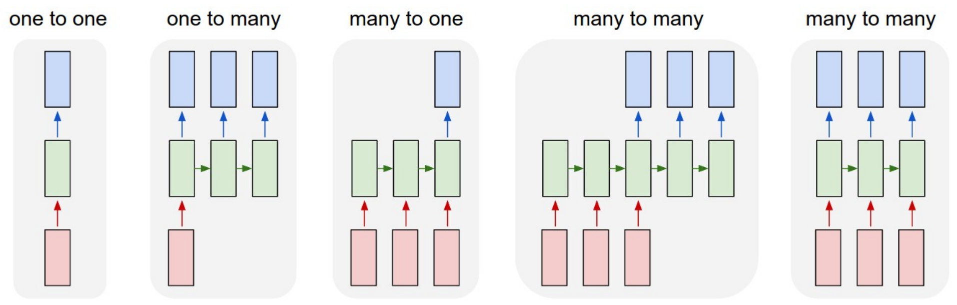


Figure 3.7: The several types of input/output combinations RNNs are capable of [12]

In order to get the inputs ready for the neural network, it is necessary to convert all the sequences of characters and labels into sequences of numbers. The sequences of characters from the corpus need to be mapped to the respective numeric value of each character (the dictionary based on the alphabet) and the sequences of characters from the labels need to be mapped to numeric values, and to do so there are two new dictionaries that are going to help in the mapping.

```
{'O': 1, 'B': 2, 'I': 3, 'X': 0}
```

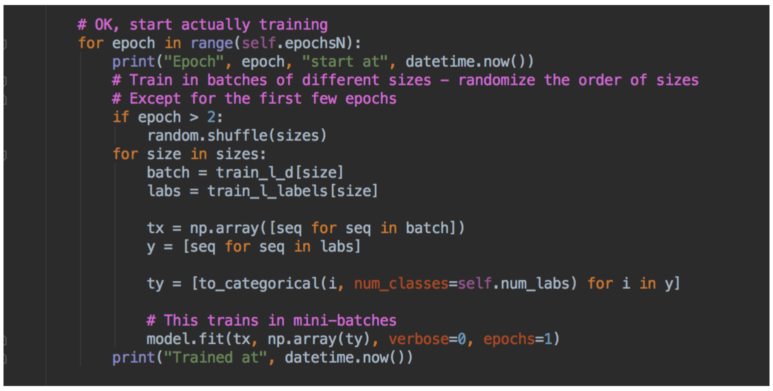
```
{'O': 0, 'B': 1, 'I': 2}
```

The first dictionary is used with an approach that uses padding in all sequences so that all have the same length and the second dictionary is used in an approach named as "mini-batch approach" because it groups sequences with the same length together. Since the labelling system is *BIO* as discussed before, it is easy to notice that the first dictionary contains one more label than it should. This happens because the first approach is based in finding the biggest sequence (the sequence with the biggest lenght of characters) and to pad all the others to that lenght. This approach is very usual and it is better than finding an intermediate value that would require to truncate some sequences and it would result in loss of information. The downside of this method is that the model takes much more time to train because of the big dimensions of its inputs. For example, if the dataset had 200 sentences and the biggest sentence contained 300 characters while the others sequences are mostly around 100 characters, all the 200 sequences would be padded to the 300 characters. Because of this padding it is necessary to make a distinction between the labels that are part of the original sequence and the labels that are being added. Since the padding adds 0's to the sequence, the dictionary will map the value 0 to an 'X' in order to not confuse the model with this new label and the label that indicate that a character does not belong to a mutation. While in the first dictionary the label to indicate that is the number 1, in the second dictionary the same label corresponds to the value 0.

The second dictionary is used in the second approach. This approach relies on using groups of inputs with the same length as inputs to the neural network and it is called **mini-batchs**. In this approach, sentences with similar length are grouped and fed together as inputs to the network. If the dataset is composed by 5 sentences with 65, 82, 120, 200 and 120 characters we are going to create 4 groups: one for the sentence with 65 characters, another for the sentence with 82 characters, another for the sentence with 200 characters and the last one with the two sentences with 120 characters. By doing this, the neural network will have inputs with multiple dimensions and will not be necessary to pad or truncate any sentence in order to respect the sequence length established. This means that the batch size will vary at each length. If one length is 40 and there are 32 sequences with that lenght, those 32 sequences are fed at the same time to the neural network. The sentences of each batch can be fed in any order. This is done in our model to prevent the model from receiving always the data in the same order, which could influence the training.

In order to create this two approaches it is necessary to change the code and the way the train is done in each approach. It is possible to see in fig 3.8 that to reach the most important function **model.fit** it is necessary to do a little bit of coding. First of all, since we only want to feed the network with sequences of the same length, it is necessary to create a cycle to represent all the epochs wanted (epoch is a full passage on the dataset) because the objective is to first give all the sequences in groups and only then repeat them again for the number of times necessary. In the first three epochs the order in which the sequences are given is always the same, but after these three epochs it is made a shuffle in order to change the order of the sequences so that the model do not get used to always seeing the same data in the same order and this way try to maximize the learning process.

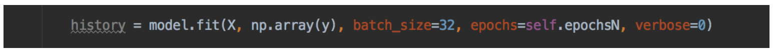
In the approach with the maximum length of the sequences as the input



```
# OK, start actually training
for epoch in range(self.epochsN):
    print("Epoch", epoch, "start at", datetime.now())
    # Train in batches of different sizes - randomize the order of sizes
    # Except for the first few epochs
    if epoch > 2:
        random.shuffle(sizes)
    for size in sizes:
        batch = train_l_d[size]
        labs = train_l_labels[size]

        tx = np.array([seq for seq in batch])
        y = [seq for seq in labs]

        ty = [to_categorical(i, num_classes=self.num_labs) for i in y]

        # This trains in mini-batches
        model.fit(tx, np.array(ty), verbose=0, epochs=1)
    print("Trained at", datetime.now())
```

Figure 3.8: Code needed to put the network training with mini-batchs

length, all the data is first padded to the same lenght and then it is only necessary to put the model training with the correct batch size and number of epochs as can be seen in fig 3.9. The complexity and quantity of code necessary is smaller then when using mini-batchs.



```
history = model.fit(X, np.array(y), batch_size=32, epochs=self.epochsN, verbose=0)
```

Figure 3.9: Code needed to put the network training after all sequences being padded

Results

In order to understand if the approaches were identical in terms of results, both were trained with 10 epochs and 20 epochs at first like it is possible to see in table 3.2.

	10 epochs			20 epochs		
	Precision	Recall	F	Precision	Recal	F
Mini-Batch	0.8392	0.6185	0.7122	0.8202	0.8556	0.8375
Max Seq	0.5344	0.5517	0.5429	0.8304	0.8125	0.8214

Table 3.2: Results from both models with 10 and 20 epochs

The results are divided in 3 metrics: **Precision**, **Recall** and **F-Measure**. In this work, they were calculated by hand and no external library was used to measure the results. Analyzing the results it is possible to see that the mini-batch approach had better results both on the training with 10 epochs and 20 epochs. The maximum sequence length produces very bad results with a train of 10 epochs and it can be derived to the model dealing with more complexity because of all the sequences that are padded and need to produce extra labels that are "rubish". Maybe 10 epochs are a small number for the model to figure out what characters are part of the corpus and what characters were added because of the padding. Yet, for 20 epochs the maximum sequence approach produces results closer to the values of the mini-batch approach with a little more precison but the recal being substantially lower. A very important and transverse conclusion is that both approachs are dealing with a small dataset that only contains 334 documents (there are other similar studies with datasets containing around hundreds of thousands documents) which means that one epoch contains few information and it is necessary to increase the number of epochs in order to maximize the learning process of the neural network.

The previous approach was based on training the model for a number of epochs and then use the test dataset to see the results, in this case the f-measure, precision and recall. While this is a valid method to analyze the model's performance, it was created another method to train and test the data at each epoch but it is only valid for the mini-batch model. Since the mini-batch model trains in batchs of different sizes and the epoch cycle is "hand made", at the end of each epoch, it is possible to test the model with the test dataset to see the values of each metric for the current epoch. Each value is stored and only replaced when a better value is achieved. In another words, we can run the model for 40 epochs, but if the epoch that produced better results is the epoch number 32, the model in that epoch is the one that is going to be saved. This is inspired by the model checkpoint callback offered by Keras, that saves the better epoch. Yet, since our model needs to have these methods implemented by hand, it is necessary to save only the best epoch when the values are the best, and then replace it if another epoch produces better results. This approach saves a lot of time in finding the best model, because it compares the results from all the epochs, while training the model for a number of epochs and saving the final results and weights would force us to run the model more times for different number of epochs in order to compare results.

The following table 3.3 contains the best results from a mini-batch model trained for 40 epochs with the described approach. It is possible to see that best epoch was the epoch number 37, with a f-measure of **0.874** while the more close was epoch number 30, with a f-measure of **0.851**. Between these two epochs, the precision is lower in the last one by a small margin, yet the recal is higher by a far superior margin than the one between the precision of both epochs. In fact, this was the epoch with the highest recal and the second highest precision. Yet, this is epoch number 37 and in case we had not used this approach, the model saved would have had 40 epochs and the results would be inferior to the ones registered by this epoch.

Epoch N^{o}	Precision	Recal	F
23	0.785	0.834	0.809
25	0.861	0.838	0.849
30	0.884	0.821	0.851
37	0.881	0.866	**0.874**

Table 3.3: Results with this approach of training and testing each epoch

Choosing the correct batch size(hyperparameter) for a model is a matter of trying and watch the results even if some guidelines can be followed. A larger number of batch size will increase the available computational parallelism, but a small number of batches has been shown to provide improved generalization performance [55]. Deep Learning optimization is typically based on Stochastic Gradient Descent (SGD). The use of larger batch sizes is to improve the parellelism of SGD in order to increase both the efficiency of current processors and to allow the distributed processing on a larger number of nodes. This is important because of the training of huge datasets where it is very important to use the hardware to the maximum and to take the less time possible. But, the use of small batch sizes has been shown to improve generalization performance and optimization convergence [56] and overral, the experiments conducted in [55] support the conclusion that using small batch sizes for training benefits in terms of range of learning rates that provide a stable convergence of the model and achieve bet test performance for a given number of epochs. This information helped us establish the number of batch sizes as 32.

Mini-Batch			Max-Seq		
Precision	Recal	F	Precision	Recal	F
0.881	0.866	0.874	0.844	0.868	0.856

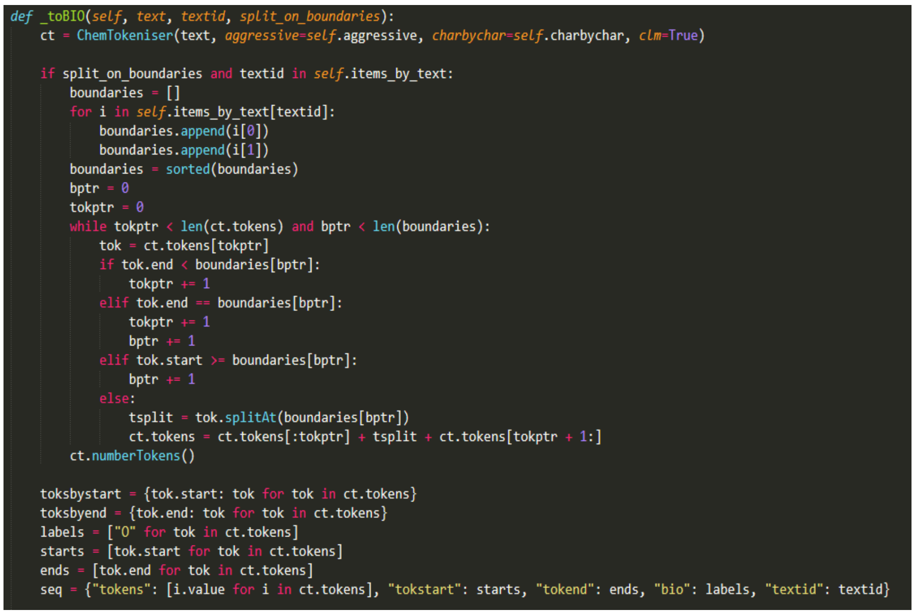
Table 3.4: Best results for both approaches

Comparing the results in both approaches, it is possible to see that the results for the mini-batch model are better in all the measures, except the recall, but it is almost the same there. But it is important to note that when training a mini-batch model, we can train for 40 or 60 batches, but it is possible to save the model with the best epoch. This is very important because allow us to always have the best possible model in that range of epochs. This is also possible to realize with Keras for the second approach with one of their callbacks *'EarlyStopping'*. This callback stops training when a monitored quantity has stopped improving (loss or accuracy), yet the usage of this callback with our model was very difficult to conciliate because our accuracy and loss values in training are quickly to converge due to the small dataset and the fact that the number of characters that have the tag *'0'* is the majority of all the characters it is very easy for the network in its earlier stages to predict almost everything as *'0'* and still achieve a great accuracy which would activate the callback and would stop the training. The dowside of one aspect in relation to the other is the times both take to train. A model with a padding so big as 751 takes a lot of time to train in comparison to the mini-batch approach where each sequence is trained with its natural lenght and because of that it is faster.

3.2.2 Word Model

All of the architectural aspects described in the character model are the same for this one, except the use of tokenization(pre-processing) and word embedding models. The tokenization was a big problem in the creation of a proper word model because of the difficulties in parsing the scientific text due to it is particularities and complex words that contain a mix of letters in lower and uppercase, together with numbers and pontuaction. Due to this particularity the use of standard tokenizers like *NLTK Tokenizer* and even the *Keras Tokenizer* were insufficient to successfully tokenize the scientific corpus. With this problems in mind, there was an article that was helpful in understanding how to perform a proper tokenization based on a scientific corpus and the locations of some of the keywords that needed to be treated as proper tokens [57].

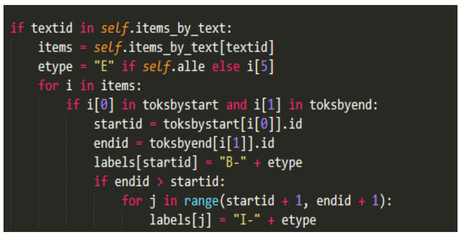
This tokenizer is responsible for finding the entities that are mutations in the corpus and to tokenize the corpus but keep all the tokens that are part or are a mutation properly marked with the correct tags. The tokenizer scroll through all the documents and in each document it saves the corpus/string which needs to be tokenize but it also saves all the mutations and it is location/offset where the mutation starts and ends. With this information the tokenizer knows which words or group of words need to be properly tagged after being tokenized. This is importance because with a normal tokenizer would be impossible to tokenize the text and then know which tokens belong to which mutations since stardand tokenizers only care about tokenize the text and do not pay attention to the tokenization of words that once are in tokens need to be presented together to make sense of the information. What this tokenizer does in detail is to use the ChemTokenizer that is specialized in tokenization for chemistry and related text (in our case the mutation text fits this expertise) to tokenize the corpus in tokens with the knowledge it has of scientific text structure and returns a list of all the tokens of the corpus and their offsets. Then, this tokens and their locations are confronted with the offsets of each mutation in order to properly organize the tokens and to make sure the tokens that form a mutation are divided in a way that they are not joined with another tokens that do not make part of the mutation. This is done in the following function 3.10.



```python
def _toBIO(self, text, textid, split_on_boundaries):
    ct = ChemTokeniser(text, aggressive=self.aggressive, charbychar=self.charbychar, clm=True)

    if split_on_boundaries and textid in self.items_by_text:
        boundaries = []
        for i in self.items_by_text[textid]:
            boundaries.append(i[0])
            boundaries.append(i[1])
        boundaries = sorted(boundaries)
        bptr = 0
        tokptr = 0
        while tokptr < len(ct.tokens) and bptr < len(boundaries):
            tok = ct.tokens[tokptr]
            if tok.end < boundaries[bptr]:
                tokptr += 1
            elif tok.end == boundaries[bptr]:
                tokptr += 1
                bptr += 1
            elif tok.start >= boundaries[bptr]:
                bptr += 1
            else:
                tsplit = tok.splitAt(boundaries[bptr])
                ct.tokens = ct.tokens[:tokptr] + tsplit + ct.tokens[tokptr + 1:]
        ct.numberTokens()

    toksbystart = {tok.start: tok for tok in ct.tokens}
    toksbyend = {tok.end: tok for tok in ct.tokens}
    labels = ["O" for tok in ct.tokens]
    starts = [tok.start for tok in ct.tokens]
    ends = [tok.end for tok in ct.tokens]
    seq = {"tokens": [i.value for i in ct.tokens], "tokstart": starts, "tokend": ends, "bio": labels, "textid": textid}
```

Figure 3.10: Function that divides the tokens in the correct way to respect the offsets of each mutation

Yet, in this part the tokenizer is only worried in making sure that the tokens are at their correct locations and that the tokens that make part of mutation are properly isolated. Because of this, it is possible to see that all the labels contain the tag *'O'*, which means that the tokens do not have their correct tags. This is only done after the corpus being properly tokenized to make sure that the tags attributed to each token are final and that the tokens offsets will not suffer modifications. The assignment of a tag to each token is done in the following code 3.11.

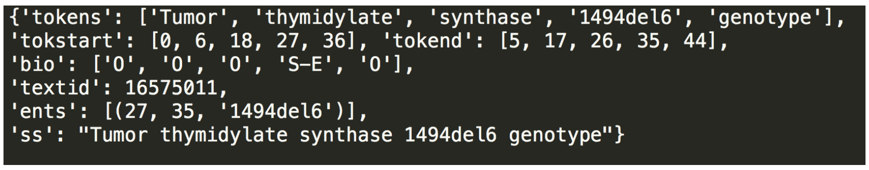


```python
if textid in self.items_by_text:
    items = self.items_by_text[textid]
    etype = "E" if self.alle else i[5]
    for i in items:
        if i[0] in toksbystart and i[1] in toksbyend:
            startid = toksbystart[i[0]].id
            endid = toksbyend[i[1]].id
            labels[startid] = "B-" + etype
            if endid > startid:
                for j in range(startid + 1, endid + 1):
                    labels[j] = "I-" + etype
```

Figure 3.11: Function that assigns a tag to each token

In the dataset used in these models, it is possible to see how the tokens that identify mutations have the locations of the first character of the token and the last character of the token in the corpus like it was exemplified in figure 3.1. It is then necessary to have a tokenizer that will respect the locations of the tokens identifying the mutations. The tokenizer used in the work done by [57] is a modified version of a Python translation of the Oscar4 Tokenizer [54]. This tokenizer takes in consideration the locations of the mutations when tokenizing the corpus and it will save the location/index of the first character of each token, and classifies each token in the following notation: *SOBIE* (sometimes known as BIOES) tags, where *'O'* marks a token that is not part of an entity, *'S'* marks a token that is the whole of an entity (a "singleton"), *'B'* marks a token at the beginning of an entity, *'I'* marks one inside an entity, and *'E'* marks one at the end.

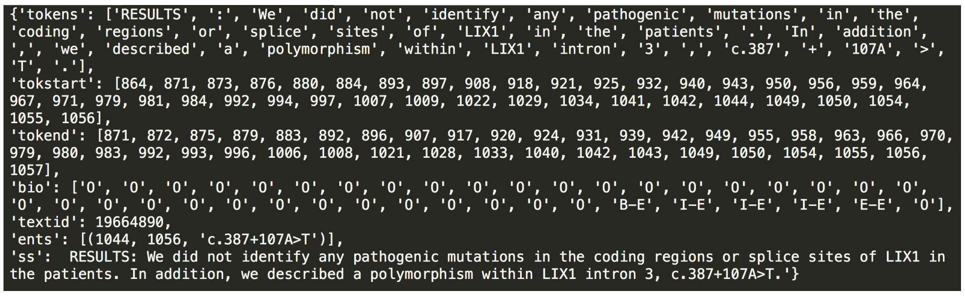
In the following figure 3.12 it is possible to see all the fields returned by the tokenizer. In this example it is assumed that the corpus is only those 5 words and is represented under the field *"ss"*. The field *"tokens"* contains the tokens the tokenizer processed from the corpus and the fields *"tokstart"* and *"tokend"* represent the index of the the first character and the index of the last character of each token in the corpus and it is possible to see in this example that the index of the characters are exactly the same of the ones in the field *"ents"* that is the exactly information(tokens and indices) presented in the training dataset. The tokenizer takes this inputs from the dataset in order to know each tokens make part of a mutation token. In this example this is not cleary because the only mutation was divided in a single token ("singleton"). The field *"bio"* contains the tags of each token present.



```
{'tokens': ['Tumor', 'thymidylate', 'synthase', '1494del6', 'genotype'],
'tokstart': [0, 6, 18, 27, 36], 'tokend': [5, 17, 26, 35, 44],
'bio': ['O', 'O', 'O', 'S-E', 'O'],
'textid': 16575011,
'ents': [(27, 35, '1494del6')],
'ss': "Tumor thymidylate synthase 1494del6 genotype"}
```

Figure 3.12: Example of the output given by the tokenizer

This following example 3.13 contains a specific mutation that is divided in 5 tokens. This mutation starts in the index 1044 and ends in the index 1056, yet it contains more than just one token in this range. The mutation *"c.387+107A>T"* is divided in the following tokens: *'c.387'*, *'+'*, *'107A'*, *'>'* and *'T'*. The first token has the tag *'B-E'* because it is the first token of the mutation and the last token has *'E-E'* because it is the last. All the intermediate tokens have the tags *'I-E'*. With this example it is possible to see how the tokenizer saves all the tokens related to the same mutation, eliminating the problems of the others tokenizers that were not capable to save information that would relate a set of tokens to the same mutation.



```
{'tokens': ['RESULTS', ':', 'We', 'did', 'not', 'identify', 'any', 'pathogenic', 'mutations', 'in', 'the',
'coding', 'regions', 'or', 'splice', 'sites', 'of', 'LIX1', 'in', 'the', 'patients', '.', 'In', 'addition',
',', 'we', 'described', 'a', 'polymorphism', 'within', 'LIX1', 'intron', '3', ',', 'c.387', '+', '107A', '>',
'T', '.'],
'tokstart': [864, 871, 873, 876, 880, 884, 893, 897, 908, 918, 921, 925, 932, 940, 943, 950, 956, 959, 964,
967, 971, 979, 981, 984, 992, 994, 997, 1007, 1009, 1022, 1029, 1034, 1041, 1042, 1044, 1049, 1050, 1054,
1055, 1056],
'tokend': [871, 872, 875, 879, 883, 892, 896, 907, 917, 920, 924, 931, 939, 942, 949, 955, 958, 963, 966, 970,
979, 980, 983, 992, 993, 996, 1006, 1008, 1021, 1028, 1033, 1040, 1042, 1043, 1049, 1050, 1054, 1055, 1056,
1057],
'bio': ['O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O',
'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'B-E', 'I-E', 'I-E', 'I-E', 'E-E', 'O'],
'textid': 19664890,
'ents': [(1044, 1056, 'c.387+107A>T')],
'ss':  RESULTS: We did not identify any pathogenic mutations in the coding regions or splice sites of LIX1 in
the patients. In addition, we described a polymorphism within LIX1 intron 3, c.387+107A>T.'}
```

Figure 3.13: Example of the output given by the tokenizer - 2

Architecure

The tokens are words or parts of words, or even pontuaction and as described before neural networks do not recognize inputs of words/characters. It is necessary in the characters case to create a map that converts the character to a number and in case of words it is possible to use word embeddings. Word embeddings are a type of word representation that allows words with similar meaning to have a similar representation. Word embeddings are a class of techniques where words are represented as vectors. Each word is mapped to one vector and the vectors are learned in a way that resembles a neural network, and this is why this technique is used in the DL. The key aspect of this approach is the idea of using a dense distributed representation for each word. This distributed representation is learned based on the usage of words. This means that words that are used in similar ways/contexts have similar representations.

This model was tested with two types of word embeddings to see which one achieved better results. The first group of word embeddings were induced from a combination of PubMed and PMC texts with texts extracted from a English Wikipedia dump [14]. The second group of word embeddings were from Global Vectors for Word Representation, or GloVe, algorithm that is an extension to the word2vec method for efficiently learning word vectors. Classical vector space model representations of words were developed using matrix factorization techniques such as Latent Semantic Analysis (LSA) that do a good job of using global text statistics but are not as good as the learned methods like word2vec at capturing meaning and demonstrating it. GloVe is an approach that marries both the global statistics of matrix factorization techniques like LSA with the local context-based learning in word2vec. Rather than using a window to define local context, GloVe constructs an explicit word-context or word co-occurrence matrix using statistics across the whole text corpus. The result is a learning model that may result in generally better word embeddings. The objective in using both these models is to understand if the use of word embeddings that were induced from documents with scientific language produces better results than using the GloVe embeddings which are induced from a ample type of documents. With the embeddings from Bio Nlplab [14] it was possible to convert **94639** words to word embeddings, while only **1437** words were unknow to the embeddings.

Results

With a word model, the words are already formed and the neural network does not need to understand how the characters are related between themselves to form words. Because of this, it is interesting to compare the results between a neural network with multiple BI-LSTM-CRF layers and one with just one BI-LSTM-CRF layer. In the character model is very important to have multiple BI-LSTM layers with CRF in order for the network to understand the features internally. To train the network was used the same approach of using mini-batchs when training the network for characters. This approach will train in batches all the sequences with the same lenght and it allow us to at the end of each epoch to test the results for the test dataset in order to see which epoch produces the best results. The following results 3.5 were achieved when training a neural network for the maximum number of 40 epochs and with just one LSTM-CRF layer.

It is possible to see that the results are much better for the model using the Bio NPLAB word vectors. One of the reasons is the quantity of words presented in each model. The **Bio NPLAB** word vectors contain around 5M of words while the **GloVe** word vectors used (glove 6B) contains aroud 400K of words. Yet, the biggest model from GloVe contains 2.2M, still less than half of the quantity from Bio NPLAB. The quantities in each model may affect the outcome, yet the **GloVe** word vectors are the standard for the NLP tasks. This shows how difficult it is to tokenize and process scientific text and that the **Bio NPLAB** word vectors contain more words related with the dataset used.

Bio NPLAB				GloVe			
Epoch	Precision	Recal	F	Epoch	Precision	Recal	F
19	0.753	0.642	0.693	23	0.5666	0.513	0.538

Table 3.5: Results for word model with one BI-LSTM-CRF layer

After getting the results with just one BI-LSTM-CRF layer was important to see how they would improve with multiple layers. As explained before in the character model was important to use multiple BI-LSTM-CRF layers to make sure that the network would cleary understand how the characters are related. Since the results on the word model with just one BI-LSTM-CR layer were lower, it was important to see if more layers would improve or if the tokenization of scientific text was a problem that would require another approach. In the following picture it is possible to see the results in the following 3.6. The results improved a bit, in particular in the *recall* measure. With this results it is not possible to conclude that it is better to have a multiple LSTM layer if we aim to have better results in a case where the tokenization is particularly difficult and most tokens contain words or part of words that are very specific to the scientific context and do not have representation on the word embedding models.

Bio NPLAB				GloVe			
Epoch	Precision	Recal	F	Epoch	Precision	Recal	F
39	0.672	0.772	0.718	31	0.564	0.526	0.544

Table 3.6: Results for word model with multiple(3) BI-LSTM-CRF layers

The architecture of the word model is exactly the same as the one of the character model. Both have three BI-LSTM layers with a CRF on top to predict the final output of each sample. The best results in each model were trained with a mini-batch approach that allowed us to save the best epoch. Yet, the results are far superior in the character model. The only big difference between these two models are the pre-processing. In this case, the word model relies on tokenization to create tokens and a word embedding model to represent those tokens in a way the neural network can understand. The embedding model with best results was **Bio NPLAB** that contains embeddings extracted from biomedical corpus and contains around five million embeddings. The fact that tokenization is the biggest issue in this model confirms the concerns of [48] in regard to the creation of proper tokenizers to NER tasks in biomedical corpus. In this case, would be interesting to see in the future if another tokenizer could achieve better results or if it was necessary to introduce hand-crafted rules in this model to achieve the same performance of the character model.

Chapter 4

Conclusion

The objective of the work described in this thesis was to create a NER system/algorithm that could performe a sequence labelling task in the tmVar[10] dataset. To perform this task we have chosen to create two Deep Learning models and compare the performance of both. The main objective in using Deep Learning was to create a system that had none to minimal hand-crafted rules/features to help understanding the data, since Deep Learning excels other technologies in feature detection. We created two models, one with character and another with word embeddings. Each main model was trained in two configurations. The character model was trained with a mini-batch approach, in which sentences with similar length are grouped, and with a maximum sequence length approach. The word model was only trained with a mini-batch approach. While the character embeddings were created during the training, the word embeddings used came from two different models: the BioNPLab word2vec models[14] and the GloVe model[15].

The results obtained show that it is possible to achieve good perfomance without the presence of hand-crafted rules. The tmVar system produced a F-measure of 0.914 with CRF methods and the help of post-processing methods. In this work, we relied only in a neural network to understand the dataset and achieved as best result 0.874 of F-measure as can be seen in table 4.1. Our best result was achieved in the character model and we believe that such results are because of the biomedical corpus being morphologically richer than a standard corpus. The mini-batch approach is the suggested one because it trains in lower time, since there is no padding in the sentences. The fact that the word model could not find embeddings for all the tokens could be determinant in the lower results presented by the word model, but it is possible to see that the model using the BioNPL embeddings(trained on biomedical corpura) achieved higher results that the GloVe embeddings(trained on standard corpura). Another problem for the word model could be the tokenization that is a very important aspect in the creation of any word model. Remais to be seen if another tokenizer could increase the performance of this model.

Model	Precision	Recal	F-Measure
tmVar [10]	0.914	0.914	0.914
Char-Mini-Batch	0.881	0.866	0.874
Char-Max-Seq	0.844	0.868	0.856
Words-BioNLP	0.672	0.77	0.718
Words-Glove	0.564	0.526	0.544

Table 4.1: Final Results

In this thesis our objective was to create the models using TF as the technology but to do so was necessary a longer time than the one we had at our disposal. TF is more complete and allows to do more than Keras, yet it also requires more knowledge and more time to learn it, which lead us to change the technology to Keras in order to have faster prototyping. Ideally we would have wanted to train our models with another datasets and to see if the results sustained or if it was necessary to make introduce hand-made rules to achieve similar results to the ones obtained before. Yet, training a neural network takes a lot of time and due to the shared access of the machine used for training, it was not possible to train another datasets and models. The word model also had a worst performance than the character model and it would be interesting to see if with a few hand-made rules it could be possible to increase the performance. But as said before, this models take a lot of time to train and to debug when something goes wrong, which prevented us to do everything we wanted.

With all of this in mind, it would be interesting to see if with very few hand-crafted rules the character model could overcome the tmVar in terms of perfomance, showing the power of neural networks in text classification and complex corpuras like the biomedical one. It would also be important to see what is needed to increase the performance of the word model and how much more complexity it would need to reach closer levels to the character one. It would also be important to see how the current model performs with another specific datasets to deduce that neural networks are powerfull enough in feature extraction and can help creating systems that rely less and less in rules created by the programmers and that are different for each problem.

Bibliography

[1] M. Manning, "Why all the focus on artificial intelligence?" [Online]. Available: http://www.tgdaily.com/technology/why-all-the-focus-on-artificial-intelligence

[2] M. Sharma, "The fault in our approach: What you're doing wrong while implementing recurrent neural..." Oct 2016. [Online]. Available: https://medium.com/emergent-future/the-fault-in-our-approach-what-youre-doing-wrong-while-implementing-recurrent-neural-network-lstm-929fbe17723c

[3] J. Allaire, "R interface to tensorflow estimators." [Online]. Available: https://tensorflow.rstudio.com/tfestimators/

[4] A. Cheremskoy, "Keras, theano and tensorflow on windows and linux," May 2017. [Online]. Available: https://gettocode.com/2016/12/02/keras-on-theano-and-tensorflow-on-windows-and-linux/

[5] "Understanding lstm networks." [Online]. Available: http://colah.github.io/posts/2015-08-Understanding-LSTMs/

[6] "Nikhil buduma | a deep dive into recurrent neural nets." [Online]. Available: http://nikhilbuduma.com/2015/01/11/a-deep-dive-into-recurrent-neural-networks/

[7] Z. Huang, W. Xu, and K. Yu, "Bidirectional LSTM-CRF Models for Sequence Tagging," 2015. [Online]. Available: http://arxiv.org/abs/1508.01991

[8] D. Britz, "Understanding convolutional neural networks for nlp," Jan 2016. [Online]. Available: http://www.wildml.com/2015/11/understanding-convolutional-neural-networks-for-nlp/

[9] W. Yin, K. Kann, M. Yu, and H. Schütze, "Comparative Study of CNN and RNN for Natural Language Processing," 2017. [Online]. Available: http://arxiv.org/abs/1702.01923

[10] C. H. Wei, B. R. Harris, H. Y. Kao, and Z. Lu, "TmVar: A text mining approach for extracting sequence variants in biomedical literature," *Bioinformatics*, vol. 29, no. 11, pp. 1433–1439, 2013.

[11] C. Paper and L. D. Viet, "Deletion-based sentence compression using Bi-enc-dec LSTM," no. October, 2017.

[12] keras team, "When and how to use timedistributeddense · issue 1029 · keras-team/keras." [Online]. Available: https://github.com/keras-team/keras/issues/1029

[13] "Keras: The python deep learning library." [Online]. Available: https://keras.io/

[14] S. Pyysalo, F. Ginter, H. Moen, T. Salakoski, and S. Ananiadou, "Distributional Semantics Resources for Biomedical Text Processing," *Proceedings of LBM*, pp. 39–44, 2013.

[15] J. Pennington, R. Socher, and C. D. Manning, "GloVe : Global Vectors for Word Representation." [Online]. Available: https://nlp.stanford.edu/pubs/glove.pdf

[16] R. Gaizauskas and Y. Wilks, *Information Extraction: Beyond Document Retrieval*, 1998, vol. 3, no. 2.

[17] A. Téllez-Valero, "A machine learning approach to information extraction," *... and Intelligent Text ...*, 2005. [Online]. Available: http://link.springer.com/chapter/10.1007/978-3-540-30586-6_58

[18] R. J. Mooney and U. Y. Nahm, "Text mining with Information extraction," *Proceedings of the 4th International MIDP Colloquiu*, no. September 2003, pp. 141–160, 2003.

[19] H. Song, B. Yang, and X. Liu, "Advanced Data Mining and Applications," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 8346, no. PART 1, pp. 25–35, 2013. [Online]. Available: http://www.scopus.com/inward/record.url?eid=2-s2.0-84893105379&partnerID=tZOtx3y1

[20] B. Settles, "ABNER: An open source tool for automatically tagging genes, proteins and other entity names in text," *Bioinformatics*, vol. 21, no. 14, pp. 3191–3192, 2005.

[21] G. Lample, M. Ballesteros, S. Subramanian, K. Kawakami, and C. Dyer, "Neural Architectures for Named Entity Recognition," 2016. [Online]. Available: http://arxiv.org/abs/1603.01360

[22] G. Zhou and J. Su, “Named entity recognition using an HMM-based chunk tagger,” *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics - ACL ’02*, no. July, p. 473, 2001. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1073083.1073163

[23] R. Sathya and A. Abraham, “Comparison of Supervised and Unsupervised Learning Algorithms for Pattern Classification,” *International Journal of Advanced Research in Artificial Intelligence*, vol. 2, no. 2, pp. 34–38, 2013.

[24] R. Srihari, C. Niu, and W. Li, “A hybrid approach for named entity and sub-type tagging,” *Proceedings of the sixth conference on Applied natural language processing* -, pp. 247–254, 2000. [Online]. Available: http://portal.acm.org/citation.cfm?doid=974147.974181

[25] M. Copeland, “The difference between ai, machine learning, and deep learning? | nvidia blog,” Dec 2017. [Online]. Available: https://blogs.nvidia.com/blog/2016/07/29/whats-difference-artificial-intelligence-machine-learning-deep-learning-ai/

[26] “A simple way to understand machine learning vs deep learning.” [Online]. Available: https://www.zendesk.com/blog/machine-learning-and-deep-learning/

[27] “Deep learning vs. machine learning – the essential differences you need to know!” [Online]. Available: https://www.analyticsvidhya.com/blog/2017/04/comparison-between-deep-learning-machine-learning/

[28] J. Russell, “Google’s alphago ai wins three-match series against the world’s best go player,” May 2017. [Online]. Available: https://techcrunch.com/2017/05/24/alphago-beats-planets-best-human-go-player-ke-jie/

[29] . . p. U. Sebastian Anthony Feb 7, “Google brain super-resolution image tech makes "zoom, enhance!" real,” Feb 2017. [Online]. Available: https://arstechnica.com/information-technology/2017/02/google-brain-super-resolution-zoom-enhance/

[30] P. Covington, J. Adams, and E. Sargin, “Deep neural networks for youtube recommendations,” in *Proceedings of the 10th ACM Conference on Recommender Systems*, New York, NY, USA, 2016.

[31] J. Berkman, “Machine learning vs. deep learning.” [Online]. Available: https://www.datascience.com/blog/machine-learning-and-deep-learning-what-is-the-difference

[32] "Understanding lstm networks." [Online]. Available: http://colah.github.io/posts/2015-08-Understanding-LSTMs/

[33] S. t. Chris V. Nicholson, Adam Gibson, "A beginner's guide to recurrent networks and lstms." [Online]. Available: https://deeplearning4j.org/lstm.html

[34] Ujjwalkarn, "An intuitive explanation of convolutional neural networks," May 2017. [Online]. Available: https://ujjwalkarn.me/2016/08/11/intuitive-explanation-convnets/

[35] H. Pokharna, "The best explanation of convolutional neural networks on the internet!" Jul 2016. [Online]. Available: https://medium.com/technologymadeeasy/the-best-explanation-of-convolutional-neural-networks-on-the-internet-fbb8b1ad5df8

[36] A. Sharma, "Convolutional neural networks in python with keras," Dec 2017. [Online]. Available: https://www.datacamp.com/community/tutorials/convolutional-neural-networks-python

[37] A. J. Tixier, "Introduction to CNNs and LSTMs for NLP," pp. 1–10, 2017.

[38] "Vector representations of words | tensorflow." [Online]. Available: https://www.tensorflow.org/tutorials/word2vec

[39] M. Krallinger, O. Rabal, F. Leitner, M. Vazquez, D. Salgado, Z. Lu, R. Leaman, Y. Lu, D. Ji, D. M. Lowe, R. A. Sayle, R. T. Batista-Navarro, R. Rak, T. Huber, T. Rocktäschel, S. Matos, D. Campos, B. Tang, H. Xu, T. Munkhdalai, K. H. Ryu, S. Ramanan, S. Nathan, S. Žitnik, M. Bajec, L. Weber, M. Irmer, S. A. Akhondi, J. A. Kors, S. Xu, X. An, U. K. Sikdar, A. Ekbal, M. Yoshioka, T. M. Dieb, M. Choi, K. Verspoor, M. Khabsa, C. L. Giles, H. Liu, K. E. Ravikumar, A. Lamurias, F. M. Couto, H.-J. Dai, R. T.-H. Tsai, C. Ata, T. Can, A. Usié, R. Alves, I. Segura-Bedmar, P. Martínez, J. Oyarzabal, and A. Valencia, "The chemdner corpus of chemicals and drugs and its annotation principles," *Journal of Cheminformatics*, vol. 7, no. 1, p. S2, Jan 2015. [Online]. Available: https://doi.org/10.1186/1758-2946-7-S1-S2

[40] D. Tang, B. Qin, and T. Liu, "Document Modeling with Gated Recurrent Neural Network for Sentiment Classification," *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, no. September, pp. 1422–1432, 2015. [Online]. Available: http://aclweb.org/anthology/D15-1167

[41] Y. N. Dauphin, A. Fan, M. Auli, and D. Grangier, “Language Modeling with Gated Convolutional Networks,” 2016. [Online]. Available: http://arxiv.org/abs/1612.08083

[42] R. Socher, A. Perelygin, and J. Wu, “Recursive deep models for semantic compositionality over a sentiment treebank,” *Proceedings of the ...*, pp. 1631–1642, 2013.

[43] I. Hendrickx, S. N. Kim, Z. Kozareva, P. Nakov, D. Ó Séaghdha, S. Padó, M. Pennacchiotti, L. Romano, and S. Szpakowicz, “SemEval-2010 Task 8 : Multi-Way Classification of Semantic Relations Between Pairs of Nominals,” *Computational Linguistics*, no. June 2009, pp. 94–99, 2010. [Online]. Available: http://www.aclweb.org/anthology/S10-1006

[44] S. R. Bowman, G. Angeli, C. Potts, and C. D. Manning, “A large annotated corpus for learning natural language inference,” no. September, pp. 632–642, 2015. [Online]. Available: http://arxiv.org/abs/1508.05326

[45] Y. Yang, W.-T. Yih, and C. Meek, “WIKIQA: A Challenge Dataset for Open-Domain Question Answering,” *Proceedings of EMNLP 2015*, no. September 2015, pp. 2013–2018, 2015.

[46] K. Guu, J. Miller, and P. Liang, “Traversing Knowledge Graphs in Vector Space,” no. September, pp. 318–327, 2015. [Online]. Available: http://arxiv.org/abs/1506.01094

[47] [Online]. Available: http://ronan.collobert.com/senna/

[48] I. Korvigo, M. Holmatov, A. Zaikovskii, and M. Skoblov, “Putting hands to rest : efficient deep CNN - RNN architecture for chemical named entity recognition with no hand - crafted rules,” *Journal of Cheminformatics*, pp. 1–10, 2018. [Online]. Available: https://doi.org/10.1186/s13321-018-0280-0

[49] J. G. Caporaso, W. A. Baumgartner, Jr, D. A. Randolph, K. B. Cohen, and L. Hunter, “Mutationfinder: a high-performance system for extracting point mutation mentions from text,” *Bioinformatics*, vol. 23, no. 14, pp. 1862–1865, 2007. [Online]. Available: http://dx.doi.org/10.1093/bioinformatics/btm235

[50] J. J. Webster and C. Kit, “Tokenization as the initial phase in nlp,” in *Proceedings of the 14th Conference on Computational Linguistics - Volume 4*, ser. COLING ’92. Stroudsburg, PA, USA: Association for Computational Linguistics, 1992, pp. 1106–1110. [Online]. Available: https://doi.org/10.3115/992424.992434

[51] J. Lafferty, A. McCallum, and F. C. N. Pereira, “Conditional random fields: Probabilistic models for segmenting and labeling sequence data,” *ICML ’01 Proceedings of the Eighteenth International Conference on Machine Learning*, vol. 8, no. June, pp. 282–289, 2001. [Online]. Available: http://repository.upenn.edu/cis_papers/159/%5Cnhttp://dl.acm.org/citation.cfm?id=655813

[52] [Online]. Available: https://keras.io/preprocessing/text/

[53] “nltk.tokenize package.” [Online]. Available: http://www.nltk.org/api/nltk.tokenize.html

[54] D. M. Jessop, S. E. Adams, E. L. Willighagen, L. Hawizy, and P. Murray-rust, “OSCAR4 : a flexible architecture for chemical text-mining,” pp. 1–12, 2011.

[55] D. Masters and C. Luschi, “Revisiting Small Batch Training for Deep Neural Networks,” pp. 1–18, 2018. [Online]. Available: http://arxiv.org/abs/1804.07612

[56] N. S. Keskar, D. Mudigere, J. Nocedal, M. Smelyanskiy, and P. T. P. Tang, “On Large-Batch Training for Deep Learning: Generalization Gap and Sharp Minima,” pp. 1–16, 2016. [Online]. Available: http://arxiv.org/abs/1609.04836

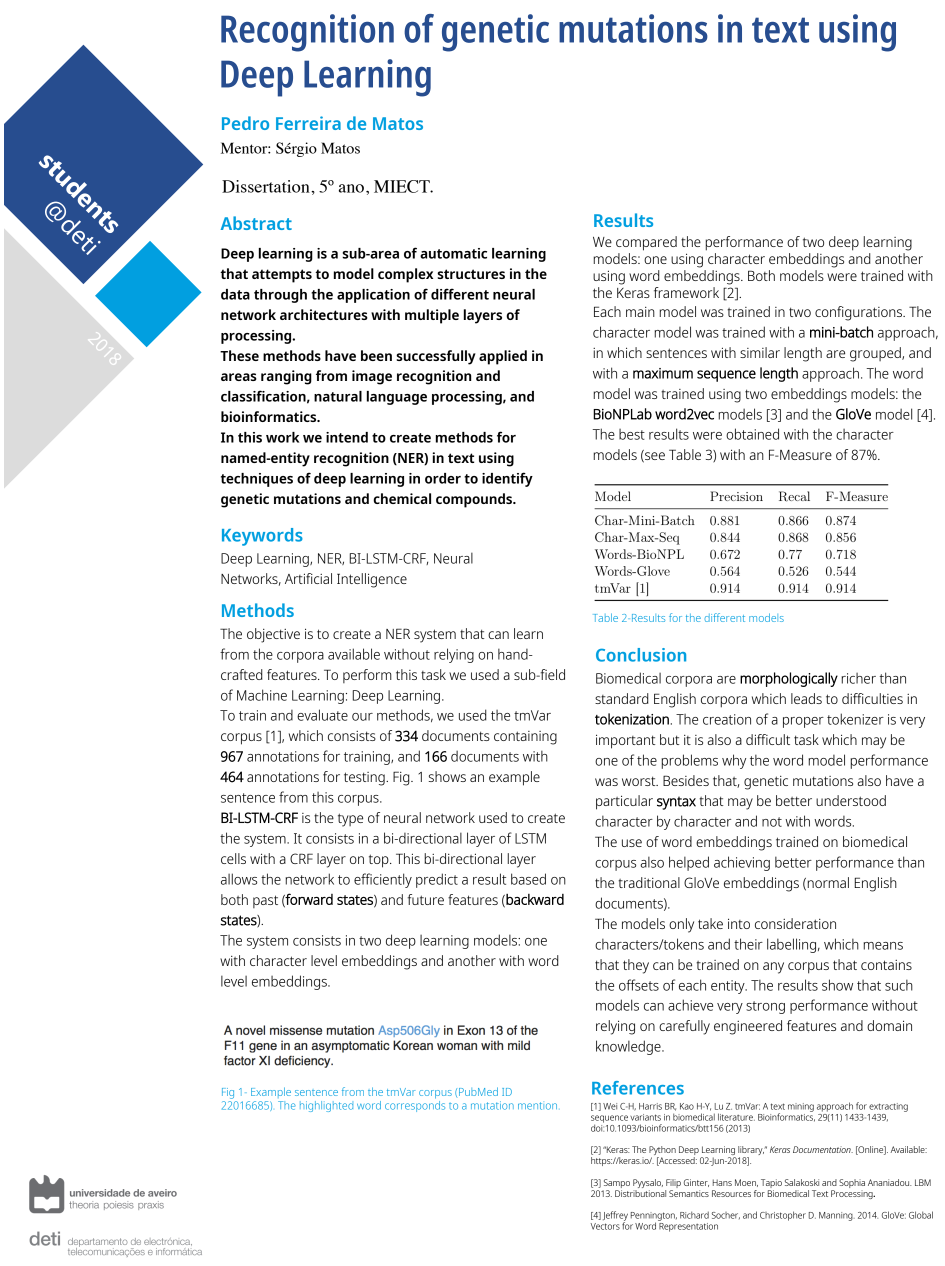
[57] P. Corbett and J. Boyle, “Chemlistem - chemical named entity recognition using recurrent neural networks,” pp. 61–68.

Appendix A

Appendix

A.1 Public Presentations

The project developed in this dissertation was presented in a public event, during 2018, called **students@deti** in the DETI department of UA. The following poster A.1 was created to present the project in the event.



students
@deti
2018

Recognition of genetic mutations in text using Deep Learning

Pedro Ferreira de Matos
Mentor: Sérgio Matos

Dissertation, 5º ano, MIECT.

Abstract

Deep learning is a sub-area of automatic learning that attempts to model complex structures in the data through the application of different neural network architectures with multiple layers of processing.
These methods have been successfully applied in areas ranging from image recognition and classification, natural language processing, and bioinformatics.
In this work we intend to create methods for named-entity recognition (NER) in text using techniques of deep learning in order to identify genetic mutations and chemical compounds.

Keywords

Deep Learning, NER, BI-LSTM-CRF, Neural Networks, Artificial Intelligence

Methods

The objective is to create a NER system that can learn from the corpora available without relying on hand-crafted features. To perform this task we used a sub-field of Machine Learning: Deep Learning.
To train and evaluate our methods, we used the tmVar corpus [1], which consists of 334 documents containing 967 annotations for training, and 166 documents with 464 annotations for testing. Fig. 1 shows an example sentence from this corpus.
BI-LSTM-CRF is the type of neural network used to create the system. It consists in a bi-directional layer of LSTM cells with a CRF layer on top. This bi-directional layer allows the network to efficiently predict a result based on both past (forward states) and future features (backward states).
The system consists in two deep learning models: one with character level embeddings and another with word level embeddings.

A novel missense mutation Asp506Gly in Exon 13 of the F11 gene in an asymptomatic Korean woman with mild factor XI deficiency.

Fig 1- Example sentence from the tmVar corpus (PubMed ID 22016685). The highlighted word corresponds to a mutation mention.

Results

We compared the performance of two deep learning models: one using character embeddings and another using word embeddings. Both models were trained with the Keras framework [2].
Each main model was trained in two configurations. The character model was trained with a mini-batch approach, in which sentences with similar length are grouped, and with a maximum sequence length approach. The word model was trained using two embeddings models: the BioNPLab word2vec models [3] and the GloVe model [4].
The best results were obtained with the character models (see Table 3) with an F-Measure of 87%.

Model	Precision	Recal	F-Measure
Char-Mini-Batch	0.881	0.866	0.874
Char-Max-Seq	0.844	0.868	0.856
Words-BioNPL	0.672	0.77	0.718
Words-Glove	0.564	0.526	0.544
tmVar [1]	0.914	0.914	0.914

Table 2-Results for the different models

Conclusion

Biomedical corpora are morphologically richer than standard English corpora which leads to difficulties in tokenization. The creation of a proper tokenizer is very important but it is also a difficult task which may be one of the problems why the word model performance was worst. Besides that, genetic mutations also have a particular syntax that may be better understood character by character and not with words.
The use of word embeddings trained on biomedical corpus also helped achieving better performance than the traditional GloVe embeddings (normal English documents).
The models only take into consideration characters/tokens and their labelling, which means that they can be trained on any corpus that contains the offsets of each entity. The results show that such models can achieve very strong performance without relying on carefully engineered features and domain knowledge.

References

[1] Wei C-H, Harris BR, Kao H-Y, Lu Z. tmVar: A text mining approach for extracting sequence variants in biomedical literature. Bioinformatics, 29(11) 1433-1439, doi:10.1093/bioinformatics/btt156 (2013)

[2] "Keras: The Python Deep Learning library," *Keras Documentation*. [Online]. Available: https://keras.io/. [Accessed: 02-Jun-2018].

[3] Sampo Pyysalo, Filip Ginter, Hans Moen, Tapio Salakoski and Sophia Ananiadou. LBM 2013. Distributional Semantics Resources for Biomedical Text Processing.

[4] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. 2014. GloVe: Global Vectors for Word Representation

universidade de aveiro
theoria poiesis praxis
deti departamento de electrónica, telecomunicações e informática

Figure A.1: The poster presented in the students@deti event.