Rheinisch-Westfälische Technische Hochschule Aachen
Lehrstuhl für Informatik 6
Prof. Dr.-Ing. Hermann Ney

Selected Topics in Human Language Technology and Machine Learning in WS 2021

Named Entity Recognition in Context

Marawan Emara

Matriculation Number 391 003

23rd March 2022

Supervisor: David Thulke

Contents

1 Introduction

As the content encapsulated within the internet, encompassing everything from vital biomedical research to simple social interactions, expands, so does the amount of unstructured, written data within it. For us to successfully manage that volume of knowledge, we must be able to provide a structured manner for the extraction of information from such unstructured data. Playing a key role in the planned composition of unstructured data, named entities aid us in the categorisation of proper names that denote real-world objects.

For computers and algorithms, the recognition of named entities can be particularly hard due to, for example, a now-resolved problem known as the "Out-of-Vocabulary" (OOV) Problem, which entails words that may be unknown within the vocabulary of common language models. A real-world example for the use of an OOV word would be searching for "Who is Alan Turing?" on Google. Here, "Alan Turing" would generally be an unrecognisable noun for the English lexicon. Nevertheless, the name remains an essential part of finding out the answer to the question. That example addresses the problem of question answering within the wider context of Natural Language Processing (NLP). Solutions for such a problem come in a multitude of varieties, such as its resolution through character-level or subword embeddings [1][3].

Other problems with recognising named entities in text involve the *ambiguity of segmentation*. Here, one needs to be able to define what the named entity is by recognising where the named entity begins and where it ends. Additionally, even when the named entity is properly recognised, one could stumble upon the problem of *type ambiguity*, where a named entity could be one of different categories.

Efficiently tackling those problems requires the use of various available mathematical and algorithmic models, all of which are gathered under the subtask of Information Extraction (IE) known as Named Entity Recognition (NER). In order to understand the contemporary models available to us, I will attempt to put Named Entity Recognition in context by first delivering some basic material on the building blocks of general Natural Language Processing and Deep Learning (DL) tools, after which I will layout two papers fundamental to our current understanding of NER, namely "Neural Architectures for Named Entity Recognition" [18] and "Contextual String Embeddings for Sequence Labelling" [1]. Both papers address different, yet entangled, parts of NER, with Akbik et al. (2018) focusing more on the *recognition* of the named entities, while Lample et al. (2016) skew towards the *grounding* of those named entities, i.e. the assigning of labels to the different named entities. Following a thorough description of both papers, I will accordingly layout more recent progress in the field of Named Entity Recognition by highlighting how some of the tools have advanced since the authoring of those two papers.

2 Foundational Blocks

To possess a proper comprehension of the later discussed papers, we need to initially apprehend the underlying concepts and models provided generally within the field of Natural Language Processing. Those concepts and models helped build the models now being widely used within NER, and as such perceiving how they work would further assist in the conceptualisation of contemporary models. In Section 2.1, we will begin by understanding the foundation of word embeddings, also known as word vectors or word representations, which is used by various models created afterwards as pre-trained input. Following that, and in Section 2.2, I will provide an insight into an essential building block of both machine learning and deep learning; the Artificial Neural Network and its derivative, the Recurrent Neural Network, both of which give us an insight into classification for Named Entity Recognition.

2.1 Classic Word Embeddings

Highlighting how word embeddings were first created helps us create a foundation and provide some context for latter sections. In their 2013 papers, Tomas Mikolov et al. [22][23] were able to improve the use of word representation effectively in Natural Language Processing, ensuring that their model is effectual enough for wide use within the field. Initially, defining a vocabulary of words for algorithmic use would have previously required the utilisation of what is known as *one-hot encoded vectors.* As an example, for a vocabulary of six words for the two sentences “I love Named Entity Recognition” and “I like named Entity Recognition”, we would have a vocabulary equivalent to {I, love, like, Named, Entity, Recognition}. Each word would then be assigned a vector where 1 would denote the word itself and 0 would denote every other word in the vocabulary: $I = \begin{bmatrix}1 & 0 & 0 & 0 & 0 & 0\end{bmatrix}$, $love = \begin{bmatrix}0 & 1 & 0 & 0 & 0 & 0\end{bmatrix}$, $like = \begin{bmatrix}0 & 0 & 1 & 0 & 0 & 0\end{bmatrix}$, $Named = \begin{bmatrix}0 & 0 & 0 & 1 & 0 & 0\end{bmatrix}$, $Entity = \begin{bmatrix}0 & 0 & 0 & 0 & 1 & 0\end{bmatrix}$, $Recognition = \begin{bmatrix}0 & 0 & 0 & 0 & 0 & 1\end{bmatrix}$.

This would essentially give us a six-dimensional vector space, with each word occupying its own dimension. In this sense, we are unable to properly differentiate the similarity between words. As in the example, **like** would be as similar, or rather dissimilar, to **love** as it is similar to **recognition** due to the cosine similarity of both being equivalent to 0. In reality, however, *like* and *love* have more of a similar meaning to one another than *like* and *recognition.*

As a solution to this problem, Mikolov et al. proposed the Skip-gram model which creates *distributed representation* for the words in a given vocabulary [22]. These distributed representations would mirror a many-to-many relationship between the word representations in a given context. This means that a word representation gains some similarity to the words surrounding it depending on how frequently they occur nearby one another in a set window. Conceptually, for a given corpus[1], we assign for each word x in its vocabulary V two random, fixed vectors; one for the *input*, denoted with v_x, and another for the *output*, denoted with v'_x[2]. Afterwards, we traverse the corpus word by word, marking the current position as position s. Using those words and their positions, we begin calculating the probability of the centre word x_s given its context words x_{s+j}. The centre word's vector v_x is then adjusted to maximise its average log probability given the context words.

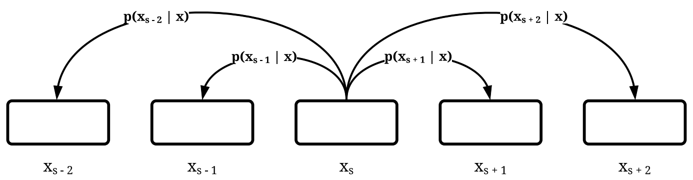

Figure 1: Illustration of the Skip-gram model's concept[3]. A window of 4 is shown in this illustration.

[1]The word *corpus* (plural *corpra*) means body in Latin, here it entails a body of text.

[2]The use of two different vectors here, one being for the centre (input) word and one being for the context (output) word, is to ensure the minimisation of the probability of a word given itself [9].

Here, the average log probability is defined through the formula for $x_1, x_2, \ldots, x_S$

$$L = \frac{1}{S} \sum_{s=1}^{S} \sum_{-c \leq j \leq c, j \neq 0} \log p(x_{s+j}|x_s) \tag{1}$$

With S denoting the size of the corpus. Furthermore, $p(x_{s+j}|x_s)$ is defined using the softmax function

$$p(x_O|x_I) = \frac{\exp({v'_{x_O}}^\top v_{x_I})}{\sum_{x=1}^{|V|} \exp({v'_x}^\top v_{x_I})} \tag{2}$$

Mikolov et al. further build onto the original paper in their second paper through the introduction of a hierarchical softmax and negative sampling [23]. The hierarchical softmax aids in increasing the efficiency of the model through its inherent attribute of using binary tree representations, with each node in the binary tree representation being the *relative probabilities* of the child nodes. This, therefore, decreases the amount of nodes calculated from $|V|$ to $log_2(|V|)$. As an alternative to the hierarchical softmax, Markov et al. also proposed negative sampling. That negative sampling helps train the model on fewer samples within the training corpus rather than training it on each pair of words within the whole corpus, making it more efficient. Negative sampling, which replaces the $\log p(x_O|x_I)$ term in the Skip-gram, is defined by them as

$$p(x|x_I) = \log \sigma({v'_{x_O}}^\top v_{x_I}) + \sum_{i=1}^{k} \mathbb{E}_{x_i \sim P_n(x)}[\log \sigma(-{v'_{x_i}}^\top v_{x_I})] \tag{3}$$

Where the sigmoid function

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{4}$$

and the noise distribution

$$P_n(x) = \Big(\frac{U(x)}{Z}\Big)^{3/4} \tag{5}$$

such that $U(x)$ is the *unigram distribution* of the words in the corpus, showing how frequent each word is, and Z is the normalisation factor as to give us an output between 0 and 1. We then take k random samples from the noise distribution $P_n(x)$ and distinguish those from the context (outer) words x_O using *logistic regression*, such as the sigmoid function mentioned in equation 4, which helps classify words into whether they are noise words or not.

Although the paper provides two models for the so-called Word2Vec method, I have mainly focused on the Skip-gram due to its prevalence. The second model named "Continuous Bag of Words" (CBOW) is similar, using a *bag of context words* to predict the centre word. Overall, and to touch back on the main point, Mikolov et al.'s two papers partially aided in resolving the issue regarding similarity in words and resulted in a fairly high-quality benchmark that could be used to further advance tools used in NLP. Along with Mikolov et al., Penington et al.'s 2014 paper was able to further prove that words within word embeddings are able to accrue some sort of sense, or meaning, to the word vectors [26].

[3]Illustration similar to what is shown in Chris Manning's NLP Stanford Lectures https://web.stanford.edu/class/cs224n/slides/cs224n-2022-lecture01-wordvecs1.pdf.

2.2 Neural Networks

Having tackled the issues regarding word similarity and sense, we now arrive at another possible difficulty; the tagging of words within the given corpus to identify and then classify the different named entities within it. Traditionally, the solution could have come in the form of logistic classifiers, such as logistic regression or naive Bayes [24]. The problem herein lies within the constrictions of both logistic regression and naive Bayes, wherein both mathematical models are limited by their assumption of linearity between the inputs they are classifying. That, however, may not always be the case in NER, seeing that there various labels, as previously mentioned.

2.2.1 Feedforward Neural Networks

Similarly, another solution came in 1958 with the creation of the first artificial neural network by Rosenblatt in the form of the perceptron [32]. The single-layer perceptron, at its core concept, *learns* from a binary set of different input vectors and attempts to classify them into their respective binary sets linearly. What interests us, however, is the modern form of Feedforward Neural Networks (FFNNs), also called multilayer perceptrons.

That occurs by assigning a different weight to each input vector, summing all of the weighted vectors with the addition of a bias, then outputting the sum through a given function. In order to train the perceptron we must first gather our input and desired output vectors $\{(x_1, d_1), (x_2, d_2), \ldots, (x_n, d_n)\}$ and the randomly initialised weights at time t $\{w_1(t), w_2(t), \ldots, w_i(t)\}$, the perceptron would then output $y(t)$ such that

$$y_j(t) = f[w(t) \cdot x_j] \tag{6}$$

Wherein x_j would be n-th dimensional input vector and d_j would be n-th dimensional desired output vector. The weights would then be updated for the desired output d through

$$w_i(t+1) = w_i(t) + r \cdot (d_j - y_j(t)) \cdot x_{j,i}, \ 0 \leq i \leq n \tag{7}$$

Here r stands for the learning rate, which is set between 0 and 1, ensuring that the weight of the current parameters is maintained, while also taking into consideration the previous outputs without giving a heavy bias to those previous outputs. The output then converges around a linear separation for the inputted binary classes. A problem arises, however, with the single-layer perceptron, which has to do with its limitations. Due to only being able to classify binary classes linearly, the perceptron could not be used for most real-world cases of classification. Later on, variants of the perceptron would be developed to include multiclass classifications. To build further onto that, multiple layers of the perceptron could be used in tandem for the production of nonlinear classifiers. Here, the activation function would be a sigmoid, similar to the one introduced in Section 2.1 and shown in equation 8

$$\sigma(x) = \frac{1}{1+e^{-x}} \quad \text{or} \quad \sigma(x) = \tanh(x) \tag{8}$$

2.2.2 Recurrent Neural Networks

Despite all of those improvements, the perceptron, essentially a type of *Feedforward Neural Network* (FFNN), is unable to take context into consideration owing to a lack of time perception. That means that it is only able to view the current input and nothing else. The answer to whether such a problem could be solved came in the form of a *Recurrent Neural*

Network (RNN). Recurrent Neural Networks are able to take into account decisions made previously, thus changing its input from just the given input at time t to also including an input from time $t-1$ [33][14]. This creates a feedback loop within RNNs that is not found in FFNNs. Accomplishing the task of carrying information forward in time and the creation of so-called *long-term dependencies* is done through a *hidden state* at timestep t for embedding e_t denoted by h_t where

$$h_t = \sigma(W_e e_t + U_h h_{t-1}) \tag{9}$$

W_e signifies the weight matrix for the embeddings, just like in FFNNs, while U_h is the matrix that is used for the hidden-state-to-hidden-state operations. Using the h_{t-1} part of equation 9, we are able to immediately deduce that the neural network not only uses information from the previous hidden state, it also theoretically includes information from all hidden states before that.

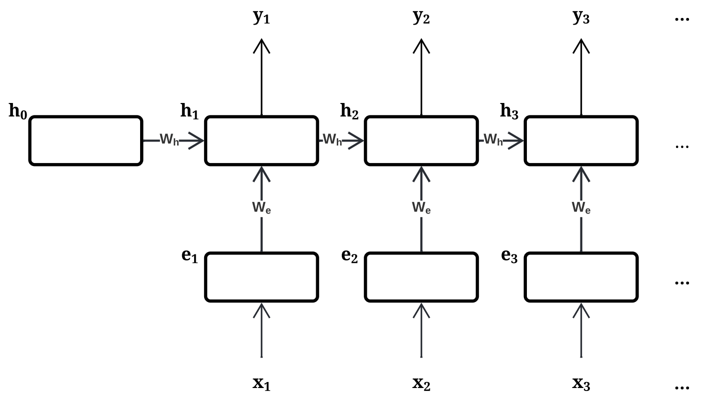

Figure 2: Illustration of a simple Recurrent Neural Network with three word embeddings as input[4].

Figure 2 showcases an abstract architecture of a RNN with three word embeddings as input. Formally described, from bottom to top, and with a vocabulary of words V,

$$x_t \in \mathbb{R}^{|V|} \tag{10a}$$

$$e_t = E \cdot x_t \tag{10b}$$

$$h_t = \sigma(W_e e_t + U_h h_{t-1} + b_1) \tag{10c}$$

$$y_t = S(U_h + b_2) \in \mathbb{R}^{|V|} \tag{10d}$$

$$J_t(\theta) = -\log y_t \quad \text{with } y_t \text{ for } x_{t-1} \tag{10e}$$

This fundamentally entails that for each word input shown in equation 10a, we transform it into the respective word embedding as shown in equation 10b. This step could either be learned with the dimensions set during training, or the embeddings could be received from a pre-trained model like Word2Vec as described in Section 2.1. We then take those word embeddings and calculate them with the hidden state, h_t, which is the sum of the weight matrix for the hidden states, W_h, multiplied by the previous hidden

[4]Illustration similar to the one used in Natural Language Processing with Deep Learning by Christopher Manning (https://web.stanford.edu/class/cs224n/slides/cs224n-2022-lecture05-rnnlm.pdf).

state, h_{t-1}, further added onto the weight matrix of each embedding, W_e, multiplied by the embedding, e_t, and finally added to the bias. All of this is placed into a sigmoid function and then passed onto the next step shown in equation 10d, which calculates the output distribution at timestep t through a softmax function [4], similar to the one defined in equation 2,

$$p(i|\vec{x}) = S(\vec{x})_i = \frac{\exp(x_i)}{\sum_{j=1}^{K} \exp(x_j)} \text{ for } i = 1, \ldots, K \text{ and } x = (x_1, \ldots, x_K) \in \mathbb{R}^K \tag{11}$$

For each timestep we are also able to calculate a loss through the loss function, also called cross-entropy function, $J_t(\theta)$.

Finally, and to accurately classify sequential input, RNNs rely on backpropagation of error and gradient descent. The specific variant of backpropagation used by RNNs is called backpropagation through time (BPTT). BPTT is executed by moving backwards from the final error, as showcased by the cross-entropy function, through the weights and inputs of every hidden state, then giving a part of the error to the weights. This occurs through the calculation of the partial derivative of the loss with respect to the partial derivative of each of the weights

$$\frac{\delta J_t(\theta)}{\delta W_h} = \sum_{i=1}^{t} \frac{\delta J_t(\theta)}{\delta W_h} \text{ for the } i\text{th timestep} \tag{12}$$

2.2.3 Long Short-Term Memory

One problem eventually arose with the RNNs described in Section 2.2.2. As the accumulated gradient is a product of all intermediate gradients in backpropagation, the accumulated gradient would tend to get smaller and smaller to the point of eventually vanishing [25]. Without knowledge of the gradient, which represents the error, we would be unable to properly adjust the weights to decrease the overall model error. This problem is called the Vanishing Gradient.

In equation 10c I introduced the equation used to compute the hidden state for a RNN. Taking an abstraction from it by substitution word embedding e for an abstract input x we get $h_t = \sigma(W_x x_t + U_h h_{t-1} + b_1)$. Using that formula, alongside the use of the chain rule, we are able to deduce that

$$\frac{\delta h_t}{\delta h_{t-1}} = diag(\sigma(W_x x_t + U_h h_{t-1} + b_1)) \cdot W_h \tag{13}$$

Now we consider the loss $J_i(\theta)$ on the ith timestep with respect to the hidden state h_j on some arbitrary previous step j

$$\frac{\delta J_i(\theta)}{\delta h_j} = \frac{\delta J_i(\theta)}{\delta h_j} \prod_{j \le t \le i} \frac{\delta h_t}{\delta h_{t-1}} \tag{14a}$$

$$\Leftrightarrow \frac{\delta J_i(\theta)}{\delta h_j} = \frac{\delta J_i(\theta)}{\delta h_j} W_h^{i-j} \prod_{j \le t \le i} diag(\sigma(W_h h_{t-1} + W_x x_t + b_1)) \tag{14b}$$

Pascanu et al. showed that, when we take the matrix 2-norm for equation 14b, if the largest eigenvalue of $W_h < 1$, then the gradient $||\frac{\delta J_i(\theta)}{\delta h_j}||$ shrinks exponentially. The Vanishing Gradient subsequently points to a problem in RNNs: the further away an input is from the first input, the smaller the gradient in terms of BPTT, consequentially undoing what was meant to be the effect of long-term dependencies in RNNs. This showcases that it might be too difficult for RNNs to preserve certain information over many timesteps. The solution for that came in the form of a new variation of a Recurrent Neural Network which incorporates a *memory cell* to maintain long-term dependencies.

Long Short-Term Memory Networks (LSTMs) are able to do so using the addition of cell states to the hidden states. Alongside that, they also introduce gates to control the aforementioned cell states [12]. Formally put: for a sequence of inputs x_t, the LSTMs would compute a sequence of hidden states h_t and cell states c_t with

$$f_t = \sigma(W_f x_t + U_f h_{t-1} + b_f) \tag{15a}$$

$$i_t = \sigma(W_i x_t + U_i h_{t-1} + b_i) \tag{15b}$$

$$o_t = \sigma(W_o x_t + U_o h_{t-1} + b_o) \tag{15c}$$

$$\bar{c}_t = \tanh(W_c x_t + U_c h_{t-1} + b_c) \tag{15d}$$

$$c_t = f_t \circ c_{t-1} + i_t \circ \bar{c}_t \tag{15e}$$

$$h_t = o_t \circ \tanh(c_t) \tag{15f}$$

Each of the first three equations, equation 15a to equation 15c correspond to a different gate of the LSTM, with each controlling a different aspect of the hidden and cell states. The *forget gate* f_t controls which information is kept from the previous cell state, i.e. what should or should not be passed onto the current cell state. Gates i_t and o_t denote the *input* and *output gates* respectively, with the input gate controlling which information from the *new cell content* $\bar{c}_t$ is written to the cell state, while the output gate controls the information passed from the cell state to the hidden state. Finally, the new cell content contains all the information that is to be written to the *cell state* c_t, while the cell state forgets the information per the forget gate and writes the new cell content depending on the input gate settings. The *hidden state* h_t, which exists very similarly to the one in RNNs, reads the output as per the output gate.

Being able to control what is written to and read from the cell state, as well as when that occurs, allows for a memory-cell-like structure that enables long-range dependencies with a much lower probability of a vanishing gradient.

2.2.4 Labelling Using Neural Networks

Having understood how some different neural network models are built, one question remains regarding how a neural network ties into Named Entity Recognition. Each of the aforementioned neural network models is able to take in a sequence of word inputs then obtain an output by running each of the inputs individually through a number of hidden states. The last step, i.e. running the outputs through the softmax non-linearity function, is what aids us in gaining a full distribution of which label should be assigned to the input. The softmax non-linearity function conceptually gives us a likelihood, or probability, of the best label to be assigned to the respective input. Passing the neural network a new input would repeat that process for the new input, giving us, once again, a probability distribution with the most probable label for said input.

Labels used for different NER labelling models are most commonly either a person (PER), location (LOC), organisation (ORG), or geo-political entity (GPE) [15]. Additionally, numerical and temporal expressions, such as time, date, and currency, are also counted as named entities. On top of the labels, specific tagging methods are used in order to help with capturing the named entity's boundaries, as well as its type. An example of such a method would be BIO tagging [30] or one of its variations, IO and BIOES tagging. The original BIO tagging includes a label B for the *beginning* token of the named entity, a label I for tokens *inside* of the span of the named entity, and finally a label O for tokens *outside* the named entity. Words that do not constitute a named entity received the O label without a type label. The aforementioned neural networks ensure that the input words receive each a boundary and a type tag, for example

$$\underbrace{RWTH}_{\text{B-ORG}}\underbrace{Aachen}_{\text{I-ORG}}\underbrace{is}_{\text{O}}\underbrace{a}_{\text{O}}\underbrace{technical}_{\text{O}}\underbrace{university}_{\text{O}}\underbrace{in}_{\text{O}}\underbrace{Aachen}_{\text{B-LOC}}. \tag{16}$$

Figure 3: Word representations and their tags for the sentence "RWTH Aachen is a technical university in Aachen".

The IO tagging variation simply removes the B label and instead tags all tokens within the span of the named entity with the label I, while the BIOES tagging variation adds a label E for tokens at the end of the span of a named entity, then a label S for one-token-span entities.

3 Innovations in Named Entity Recognition

Now that we have obtained some understanding of two fundamental machine learning and NLP models, it should be easier to comprehend the innovations-of-their-time introduced in Section 3.1.2 and Section 3.2.2.

3.1 Input Embeddings

The process of Named Entity Recognition, or any generic tagging task, begins with the input. It is therefore crucial to first recognise what sort of input embeddings were introduced for use in different models. That is especially relevant due to the input embeddings playing a key role in the overall performance of any given model, while also highlighting how such a model can, as is the case for the model introduced by Lample et al. (2016), work without the need for language-specific resources or features. Language-specific resources may include concepts such as gazetteers, which is a geographically specific index of named entities like cities or organisations.

In the following two sections, 3.1.1 and 3.1.2, we will take a look at how differing types of input embeddings might look like. Both of which are then used by their respective authors in combination with the model introduced in Section 3.2.2.

3.1.1 Character-Level Embeddings

Lample et al. (2016) introduces the first of those novel models for the creation of word embeddings. Three different aspects could be attributed to this sort of process: firstly, the creation of the final word representations from character-level embeddings concatenated with word embeddings. Secondly, the use of pre-trained word embeddings for the initialisation of the lookup table is used in the first step. Finally, the utilisation of *dropout training* to reduce the bias to either of the first or second sets of word representations. Creating the first set of word representations is done to ensure spelling sensitivity, backed up by the fact that there exists a variation of the orthography or morphology in various languages for proper nouns and more specific names. Orthographic changes here would typically include a change in the spelling of a word, while morphological changes would refer to the smallest comprehensible units of a word (including prefixes, suffixes, etc.). One key difference of the character-level model used by Lample et al. (2016) was the relinquishing of pre-defined information, such as the definition of the language's prefixes and suffixes, while opting for the training for such bound morphemes[5]. Subsequently, and as previously mentioned, this clears the way for the ability to obtain representations that are independent of handcrafted, language-specific resources. Moreover, this sort of character-level model allows for world-level Out-of-Vocabulary Problems to be more easily handled [19][18].

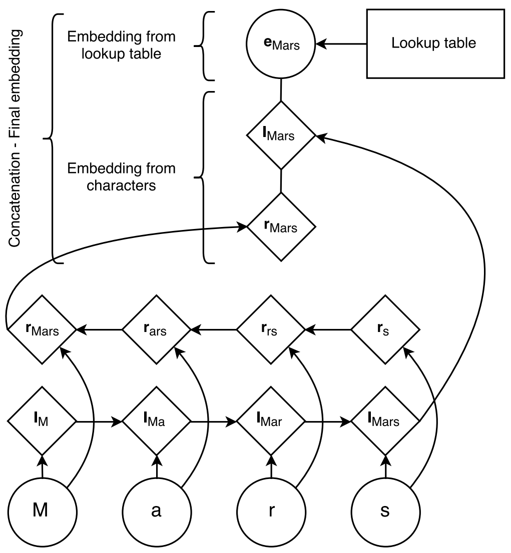

Figure 4: Architecture used for the input embeddings. The individual characters of the word representation are first inputted into a Bi-LSTM, with the output being concatenated with the original word from the lookup table [18].

The generation of a word embedding, as seen in Figure 4, begins with the setting up of a randomly initialised word representations lookup table which contains all of the word representations in the corpus. Character representations are then used as inputs for a bidirectional Long Short-Term Memory (LSTM) neural network model, similar to the process described in Section 2.2.3 and further explained in Section 3.2, concatenating the outputs of the forward and backward LSTMs to form the word representations. The word representation outputs from the Bi-LSTM are then concatenated with the respective word representation taken from the lookup table. Should a character-level concatenated word not have a respective word representation, such as OOV names, it is mapped to an "*UNK* embedding" [18]. Due to the inherent nature of LSTMs, as is with RNNs, being biased to their most recent inputs (due to the decreasing gradient), prefixes and suffixes of words

[5]Bound morphemes are morphemes that are dependent on other morphemes.

tend to be accurately represented.

In initialising the previously mentioned word representations' lookup table for the final concatenation, Lample et al. (2016) opted for the application of pre-trained word embeddings over randomly initialised ones, like the one used for the character-level embeddings. In doing so, they achieved "significant improvements" [18]. For those pre-trained embeddings, a variation of the Word2Vec word embeddings [22], as previously explained in Section 2.1, as well as the skip-n-gram embeddings developed by Ling et al. in 2015 [20], which adds *variable attention* to the CBOW variation of the Word2Vec, making the word embeddings more contextualised.

The final aspect of the input embeddings included ensuring a balance between both of the previously mentioned representations. That occurred through *dropout training* [18][11]. Dropout training regularises the different representations by randomly ignoring some of the outputs [35]. This essentially treats the outputs as nodes with different connectivity. What this does, in turn, is that the odels corrects the missing information, which causes the overall model to become more vigorous and generalised between the different intake models [18].

3.1.2 Contextual String Embeddings

In the models we have seen so far, word tokens typically carry the same word embedding, even if the context, or type, of that word token differs. As previously seen in Figure 3, the word *Aachen* has two different contexts in the sentence shown; once where it acts as a name for an organisation and once where it acts as a name for a location. There, the word Aachen would have the same word vector despite the change in context. Leveraging the change in context when undergoing the tagging of words could, potentially, greatly improve a model's ability to tag named entities by their type correctly. That was indeed the result achieved by Akbik et al.'s 2018 paper, "Contextual String Embeddings for Sequence Labelling", which achieved then-state-of-the-art results [1].

So far, we have been introduced to *classical word embeddings*, such as Word2Vec [23] in Section 2.1, and *character-level embeddings*, such as the one by Lample et al. (2016) [18] in Section 3.1.1. Classical word embeddings enable us to give some meaning to the words, in terms of both syntactically and semantically meaning. That is previously shown through the ability to capture the similarity between word representations. Character-level embeddings build onto that ability to include semantic and syntactic meaning by adding the capacity to include subword features, such as the detection of prefixes and suffixes. In Akbik et al. (2018), they introduce a method to further handle context through the use of character sequences, on top of the aforementioned abilities. Though previous models have been introduced to handle such context for words [27][28], they have not done so on a character-level and thus have lacked the capability to handle problems such as the OOV problem, word misspellings, or the handling of bound morphemes.

Contextual string embeddings, as they were named by the authors [1], are able to automatically detect a plethora of semantic and syntactic properties, such as the ones already previously described for word-level and subword-level embeddings. Even though they are character-dependent, all such properties have been proven to exist for embeddings created from characters, with the addition of sentiment [36][10][16][29].

As with other approaches in this paper so far, the authors have opted in for the use of a bidirectional LSTM when creating their embeddings [1]. The bidirectional LSTM is fed characters with the goal of training the model to predict the upcoming character based on the previously available characters. In Contextual String Embeddings, the model does not constrain itself to the characters within a word, but rather crosses both words and sentences to be able to predict upcoming characters as it traverses the entire corpus. Such prediction is calculated through a probability distribution formed by a softmax function (see equation 2). Generally, the neural network use by Akbik et al. (2018) uses the exact same one as in sections 2.2.2 and 2.2.3, in addition to equations 22a, 22b, and 22c, which are introduced later in Section 3.2.2. The outputs of the hidden states are taken at the last character of the whole input sequence in regards to the forward LSTM while being taken at the beginning character of the whole sequence in regards to the backwards LSTM. Those outputs are then concatenated, giving us the full contextual string embeddings [1], with each word being:

$$w_i = \begin{bmatrix} \overrightarrow{h}_{s_{i+1}-1} \\ \overleftarrow{h}_{s-1} \end{bmatrix} \tag{17}$$

Where the word strings begin each at position indices $s_0, s_1, \ldots, s_n$. In Figure 5, we are able to see how such a concatenation as w_i is formed.

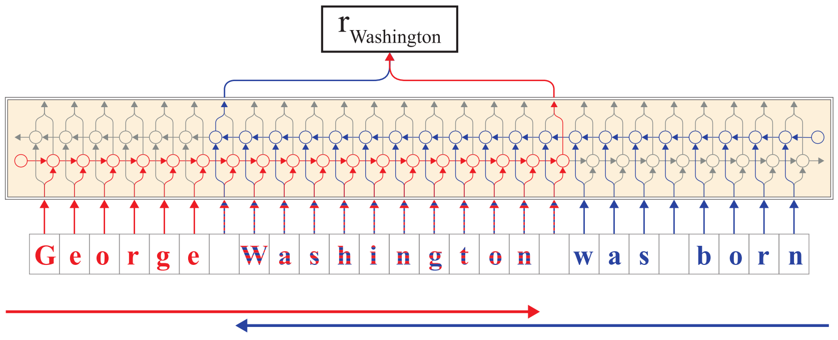

Figure 5: For the centre word shown, that being “Washington”, the whole left context from the beginning is taken into account by the forward LSTM (in red), while the whole right context is taken into account by the backwards LSTM. This information is then taken from each LSTM’s output and concatenated into that specific word’s embedding [1].

For the sequence labelling task, which could be either Named Entity Recognition or Part-of-Speech tagging, Akbik et al. (2018) used a bidirectional LSTM-CRF similar to the one described later in Section 3.2.2 [13][1]. Finally, the authors concatenated their embeddings with word-level embeddings, similar to what was done in Section 3.1.1.

3.2 Improvements on Neural Architectures

Having comprehended the sort of inputs being used, we now move onto the main architecture used in both “Neural Architectures for Named Entity Recognition” [18] and “Contextual String Embeddings for Sequence Labelling” [1] for the labelling of Named Entities, the Bi-LSTM-CRF model. In their 2016 paper, Lample et al. (2016) were able to produce a then-state-of-the-art model that combines three different aspects: a bidirectional Long Short- Term Memory model, supported by a Conditional Random Field, with the whole neural network taking in word-level tokens created from character-level embeddings as input [18]. Their model was able to become one of the first that was no longer reliant on language-specific sources, which require a considerable amount of resources to develop, and with that removing a substantial challenge in NER. In the next few sections, I will be going into detail regarding the Bi-LSTM-CRF model, which could be used in tandem with word embeddings created through the models mentioned in Section 3.1.

3.2.1 Conditional Random Fields

So far, the neural network models discussed within Section 2.2.1, Section 2.2.2, and Section 2.2.3 all work on labelling the individual elements of the input sequence independently. Meaning that each classification decision made in those models is conditionally independent of any other decision, even when it takes the loss into account through a method like BPTT. Named entities, however, usually exist in a sequence of interdependent tokens. Put concretely, the named entities are in a syntactically defined order. For example, an I-PER label cannot exist following an O label when using the BIO tagging method; it must always exist after a B-PER label. That is something that a neural network like the LSTM model might not be able to take into account due to its independent decision-making process. Solving such a problem requires generalisation over the whole sequence of inputs to ensure that the tagging decisions take the surrounding context of other tagging decisions into account. Conditional Random Fields (CRFs) [17] essentially do so. Expressly, given a sequence of inputs $x_1^n = (x_1, x_2, \dots, x_n)$ and a sequence of predictions $y_1^n = (y_1, y_2, \dots, y_n)$, we would then obtain the overall probability distribution $p(y_1^n|x_1^n) = p(y_1, y_2, \dots, y_n|x_1, x_2, \dots, x_n)$.

The simplest variation of CRFs, the same one used in Lample et al.'s 2016 paper, is the linear chain CRF. In 'regular', or rather token-level, classification approaches, the distribution would be computed as follows through the softmax function

$$p(y_1^n|x_1^n) = \prod_n p(y_n|x_n) = \prod_n \frac{\exp(y_n x_n)}{\sum_{i=1}^n \exp(x_i)} \tag{18}$$

A linear chain CRF, on the other hand, would do *sequence classification* through

$$p(y_1^n|x_1^n) = \frac{\exp(w \cdot \Phi_f(x_1^n, y_1^n))}{\sum_{y' \in Y} \exp(w \cdot \Phi_f(x_1^n, y_1^n))} \tag{19a}$$

$$\Phi_f(x_1^n, y_1^n) = \sum_n \phi_f(y_{n-1}, y_n, x, n) \tag{19b}$$

$$p(y_1^n|x_1^n) = \frac{\prod_{i=1}^n \exp(f(y_{i-1}, y_i, x, i))}{\sum_{y' \in Y} \prod_{i=1}^n \exp(f(y'_{i-1}, y'_i, x, i))} \tag{19c}$$

Where feature vector $\phi_f(x_1^n, y_1^n)$ represents the undirected connections between the different outputs of the Conditional Random Field, such that the probability assigned to y_n would dependent on the probability assigned to y_{n-1} and vice versa. The feature vector is defined by a set of feature functions $\phi_{f1}, \phi_{f2}, \dots, \phi_{fn}$ which analyses the whole sequence X, the current output y_n, the previous output y_{n-1}, and finally the current position in the sentence n. Y also presents all possible tag sequences. Note that the normalisation constant for CRFs differs from regular classification insofar that it is dependent on not only the input sequence x_1^n, but also the parameters set for the model.

One extremely important point to notice here is that the sum over all possible tag sequences needs to be calculated for the overall probability distribution. That calculation, however, is not trivial, and would result in a time complexity of $O(|Y|^n)$ if naively calculated. In practice, the solution to reducing that complexity comes from the use of dynamic programming. An algorithm named the forward (or backward) algorithm fundamentally stores results for the tag sequences calculated early on in the CRF and reuses those results down the line, saving computational power.

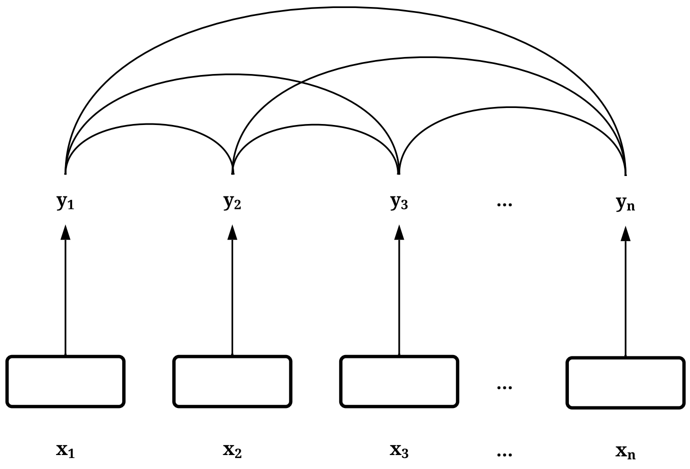

Figure 6: Illustration showing how the outputs in a Conditional Random Field form a generalised conditional distribution for the entire sequence instead of for single inputs and their respective outputs. The lines connecting $y_1, y_2, y_3, \ldots, y_n$ are the undirected connections that showcase the interdepencency of the outputs.

In order to predict the correct sequence of tags, it is essential to maximise its probability used the above prescribed equation. That is called the *inference* with a linear chain CRF and is done through

$$y^* = \underset{\mathbf{y}}{\text{argmax}} \left\{ p(y_1^n | x_1^n) \right\} \tag{20}$$

Seeing that we only need the score for a particular labelling sequence, the exponential and denominator could be disposed of

$$y^* = \underset{\mathbf{y}}{\text{argmax}} \text{ score } (y_1^n | x_1^n), \text{ where score } (y_1^n | x_1^n) = \sum_j w \cdot \Phi_f(x_1^n, y_1^n) \tag{21a}$$

$$\Leftrightarrow \; y^* = \underset{\mathbf{y}}{\text{argmax}} \left\{ \prod_{i=1}^{n} f(y_{i-1}, y_i, x, i) \right\} \tag{21b}$$

Here, we try all of the possible sequences of tags and get the maximum one. In practice, however, that would be computationally exhaustive, and thus not feasible. Therefore, $\text{argmax}_{\mathbf{y}}$ here could also be computed using a backtracker which iteratively takes the saved maximum arguments of the previous steps. This maximisation of the correctly outputted features essentially maximises the probability of the correct sequences of tags being returned. Therefore, overall, we are able to achieve a generalised sequence of labels for our initial sequence of inputs.

3.2.2 The LSTM-CRF Model

Having discussed how both LSTMs and CRFs were created, and their general purposes, we could comfortably begin to comprehend the bidirectional LSTM-CRF model proposed in "Neural Architectures for Named Entity Recognition", as well as its impact on the model's effectiveness. In order to capture context in both directions, bidirectional LSTMs (Bi-LSTMs) are used instead of a simple LSTM. The concept for Bi-LSTMs is to have two separate LSTMs; one going forward for context coming after the centre word, while the other goes backward for context preceding the centre. Both LSTMs are then concatenated in hidden states respective to the input embeddings. Concretely we would have

$$\overrightarrow{h}_t = \text{LSTM}_{\text{FW}}(\overrightarrow{h}_{t+1}, x_t) = l_i(\overrightarrow{h}_{t+1}, x_t) \tag{22a}$$

$$\overleftarrow{h}_t = \text{LSTM}_{\text{BW}}(\overleftarrow{h}_{t-1}, x_t) = r_i(\overrightarrow{h}_{t-1}, x_t) \tag{22b}$$

$$h_t = [\overrightarrow{h}_t; \overleftarrow{h}_t] \tag{22c}$$

With LSTM_{FW} denoting that a forward step of the LSTM needs to be completed, while LSTM_{BW} denotes that a backwards step of the LSTM needs to be completed. The Bi-LSTM enables us to gain a wider view of the context surrounding a word, thus enabling higher accuracy in tagging.

On top of the Bi-LSTM model, Lample et al. (2016) [18] also make use of a CRF model that takes in the output from the hidden states of the Bi-LSTM, ignoring the output distribution that comes from it. This model was first introduced in 2015 by Huang et al. [13], though changes are made to make it *hierarchical* with a CRF stacked on top of the Bi-LSTM. As seen in Figure 7, the input for the CRF would be the sequence of concatenations of word embeddings c_i. The CRF would finally produce the needed sequence of outputs y using the outputs passed from the Bi-LSTM as inputs, which would be the final result. Lample et al. (2016) found additionally that attaching an extra hidden layer between the concatenations and the CRF, instead of feeding the concatenations directly as inputs, tended to result in better outcomes.

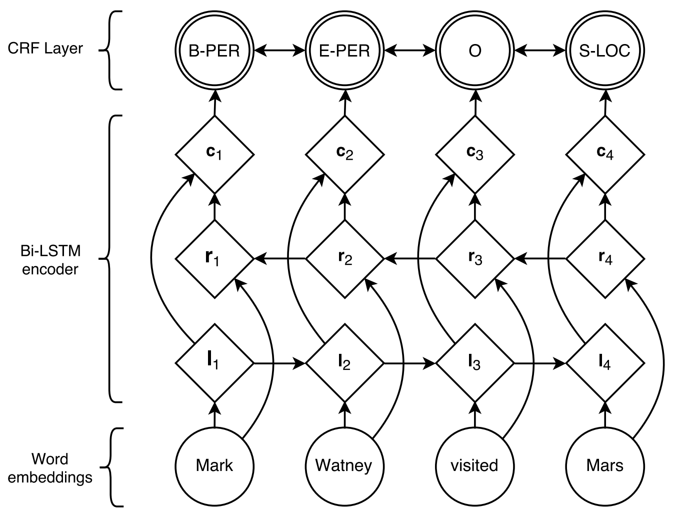

Figure 7: Architecture of the Bi-LSTM-CRF as introduced by Lample et al. (2016) Here, l_i is a representation of the LSTM_{FW}, while r_i represents LSTM_{BW}. c_i is the concatenation of both [18].

In the labelling of their output tokens, BIOES tagging was chosen over the typical BIO tagging, both of which were previously discussed in Section 2.2.4. The reason behind doing so was that the use of the BIOES tagging method would result in better performance by the overall model [31][7].

3.3 Evaluating the Models

Understanding the impacts of the models described in sections 3.2 and **??** requires that we understand the benchmarks that they were attempting to overcome. Furthermore, in order to understand those benchmarks, we must subsequently, and initially, understand how the metrics are set and what they are. In Section 3.3.1, we will take a look at the datasets used to compare both models to others in the field, then begin to comprehend what they achieved in Section 3.3.2.

3.3.1 Understanding the Evaluation Metrics and Datasets

One very common factor between various NER papers' results is that they are usually compared using the *F1 score* from testing on the *CoNLL-2003 dataset*. In order to understand what that entails, we must first comprehend what the different datasets within CoNLL-2003 are, how they are used for testing, and how the F1 score is then measured.

CoNLL-2003 is a dataset released in 2003 in a shared task revolving around language-independent named entity recognition [34]. The dataset includes, in total, eight different files that cover both the English and German languages. Corresponding to each language is "a training file, a development file, a test file, and a large file with unannotated data" [34]. All of which contain the three of the four main named entities previously discussed in Section 2.2.4; persons, location, and organisations. Alongside that, they have also included a miscellaneous category. The training file would then be used by authors for the training of their models, while the development file would be used for ensuring that the right parameters are set. Finally, the models would be tested against the test file, with the results being taken from that and the F1 score calculated for those results to compare them against other models and benchmarks.

English data	Articles	Sentences	Token	LOC	MISC	ORG	PER
Training set	946	14,987	203,621	7,140	3,438	6,321	6,600
Development set	216	3,466	51,362	1,837	922	1,341	1,842
Test set	231	3,684	46,435	1,668	702	1,661	1,617

Table 1: From left to right, the columns denote the amount of: articles, sentences, tokens, named location entities, named miscellaneous entities, named organisation entities, and named person entities in the CoNLL-2003 English language datasets [34].

German data	Articles	Sentences	Token	LOC	MISC	ORG	PER
Training set	553	12,705	206,931	4363	2,288	2,427	2,773
Development set	201	3,068	51,444	1181	1,010	1,241	1,401
Test set	155	3,160	51,943	1035	670	773	1,195

Table 2: Table the denotes the same as Table 1, but for the CoNLL-2003 German language datasets [34].

When evaluating one's model, the total number of entities would be counted for each of the types of the named entities shown above, then compared to the actual number, as shown in tables 1 and 2. Afterwards, an F1 score would be calculated. The F1 score is a variation of the F_β score [37], which is as follows:

$$F\beta = \frac{(\beta^2 + 1) \cdot precision \cdot recall}{(\beta^2 \cdot precision) + recall} \tag{23a}$$

$$F1 = \frac{2 \cdot precision \cdot recall}{precision + recall}, \beta = 1 \tag{23b}$$

$$\Leftrightarrow \; F1 = \frac{\mathrm{tp}}{\mathrm{tp} + \frac{1}{2}(\mathrm{fp} + \mathrm{fn})} \tag{23c}$$

With tp denoting true positives, meaning correctly predicted outputs of the positive class, fn denoting false negatives, meaning incorrectly predicted outputs of the negative class, and fp denoting false positives, meaning meaning incorrectly predicted outputs of the positive class. Precision elicits the number of true positives as a fraction of all positives, both true and false, while recall means the number of true positives as a fraction of the total number of positives, i.e. false negatives and true positives. The F1 score would return a value between 0 (worst possible score for a classifier) and 1 (considered a perfect classifier).

3.3.2 Evaluation of the Two Different Models

Now that we have had a general overview of the CoNLL-2003 dataset and the F1 score, we can take a look at how both the Bi-LSTM-CRF model introduced by Lample et al. (2016) and the model introduced for contextual string embeddings by Akbik et al. (2018) were both state of the art models in Named Entity Recognition at the time of their release.

Model	**NER-English F1-Score**	**NER-German F1-Score**
Bi-LSTM-CRF$_{\text{[Huang et al. (2015)]}}$	90.1*	82.3*
Bi-LSTM-CNN$_{\text{[Chiu et al. (2016)]}}$	91.6*	n/a
Bi-LSTM-CRF$_{\text{[Lample et al. (2016)]}}$	90.9	83.8*
Bi-LM-Bi-RNN$_{\text{[Peters et al. (2017)]}}$	91.9*	n/a
Bi-LM$_{\text{[Peters et al. (2018)]}}$	92.2*	n/a
Bi-LSTM-CRF$_{\text{[Akbik et al. (2018)]}}$	93.1*	88.3*

Table 3: Comparing different models for the CoNLL-2003 English and German datasets in Named Entity Recognition, all of which were considered state of the art at the time of their publishing. Notice that Lample et al. (2016) did not overcome the highest F1 score at the time of its publishing, but was considered state of the art nevertheless [13][5][18][27][28][1]. * = highest F1 score at the time of publishing.

Table 3 shows us how well each of the aforementioned models worked in the context of models that precede them. Though Lample et al. (2016) did not necessarily overcome the highest F1-score set by Chiu et al (2016) in the CoNLL-2003 English dataset at the time of its publishing, it set the state of the art benchmark for the CoNLL-2003 German dataset [5][18]. Akbik et al. (2018), on the other hand, set the new state of the art benchmarks in both datasets, overcoming all preceding models [1].

Furthermore, and though it is not clear in the table, both Lample et al. (2016) and Akbik et al. (2018) have shown that the use of certain features, such as pre-trained language models and character-level embeddings, enhances the models in terms of overall performance [18][1]. Those improvements can be seen in the ablation studies of each of the architectures, wherein a model's performance is studied further through the removal of specific components and then the testing of the outcome model, as to better comprehend how much each component of the model contributes to the overall performance.

Embedding + Architecture	**NER-English F1-Score**	**NER-German F1-Score**
PROPOSED$_{+WORD}$		
+Bi-LSTM-CRF	93.1	88.2
+Map-CRF	90.2	85.2
+Map	79.9	77.0
PROPOSED		
+Bi-LSTM-CRF	92.0	85.8
+Map-CRF	88.6	82.3
+Map	81.4	73.9
CLASSIC WORD EMBEDDINGS		
+Bi-LSTM-CRF	88.5	82.3
+Map-CRF	66.5	72.7
+Map	48.8	57.4

Table 4: Shown in this table are the ablation studies of Akbik et al.'s Flair Embeddings in comparison to Classic Word Embeddings (as described in Section 2.1). Table taken from "Contextual String Embeddings for Sequence Labelling" [1].

As seen in Table 4, Akbik et al. (2018) expiremented with different neural architectures alongside different embeddings to show the effectiveness of their proposed approach. It is clear that Contextual String Embeddings, or Flair Embeddings, perform better throughout all of the differing architectures. These results highlight the point that Flair Embeddings are an improvement to Classic Word Embeddings regardless of the neural architecture being used for labelling.

4 Related Work

4.1 Progress in Neural Architectures

At the same time that Akbik et al. (2018) was being authored, another paper was published that would change the type of models that occupy the state of the art ranks within Machine Learning at large, especially in Named Entity Recognition as well. In that paper, Vaswani et al. (2017), developed the concept behind transformers [38]. Following "Contextual String Embeddings for Sequence Labelling", almost every new best performing model has been based on Vaswani et al. (2017)'s transformers.

In order to obtain a better idea of how neural architectures progressed since Akbik et al. (2018), we will get a quick overlook of transformers and how they work in Section 4.1.1, after which a few transformer-based models will be described in Section 4.1.2.

4.1.1 Transformers

Despite attempting to deal with the Vanishing Gradient problem for RNNs, LSTMs and other similar variations of RNNs were still very limited in dealing with long-term dependencies. Not only that, but they also did not possess the ability to process the inputs given to them in parallel due to being inherently sequential. Nevertheless, with the ever more frequent use of Graphics Processing Units (GPUs) for higher performance computation, it was essential to develop a model which could process inputs in parallel. Vaswani et al. (2017) noticed that *attention*, a concept introduced for RNNs which allows alignment (identifying parts of the respective inputs that correspond to the output sequence) and translation (selection of the appropriate output based on the information received from alignment) [2], could be used without the need for the recurrence of the neural network [38].

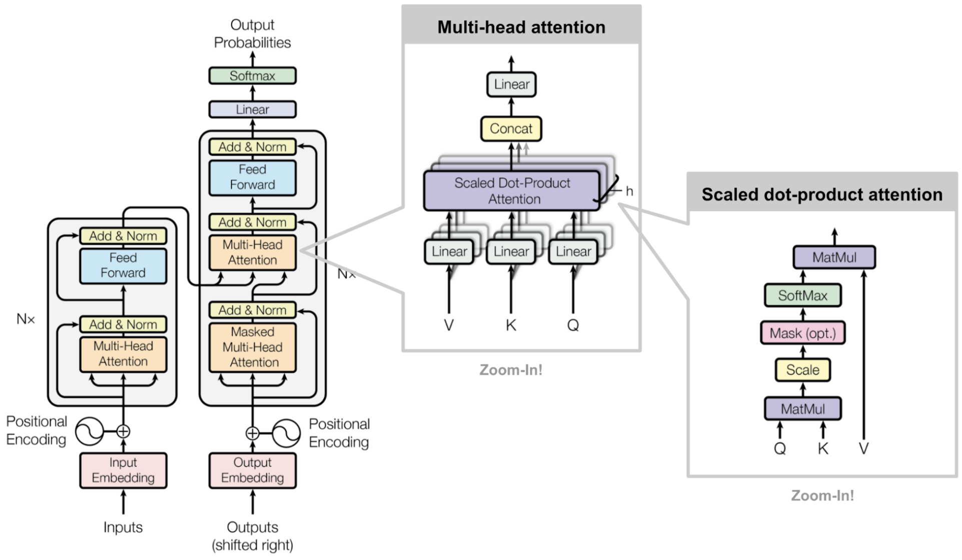

Figure 8: Transformer model architecture, showing the multi-head attention and scaled dot-product attention in perspective [38][6].

[6]Illustration credits go to https://deepfrench.gitlab.io/deep-learning-project/.

Comprehending how a transformer works requires some background knowledge about *encoders and decoders*, as well as what attention entails. Transformers make use of what could be construed as the simplest type of attention; dot-product attention. In dot-product attention, one inputs into an attention function a query q, a set of key-value pairs k, v, each have a dimension of d_k and d_v respectively, all of which are packed into matrices, Q, V and K. The attention function then outputs the weighted sum of values where the weight of each value v is an inner product of the query q and respective key k (normalised over d_k to curb a decrease in the softmax gradient, thus making it *scaled dot-product attention*) [2][38]:

$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V \tag{24}$$

Attention is the first part of a complete transformer block, with the input word vectors being the queries, keys and values. That implies that the word vectors select one another, therefore its naming as *self-attention.* Adding the ability for parallel computation for attention was done through the *multi-head attention* sub-model. This part of the overall model would compute multiple attention functions of the linearly projected queries, keys and values using the dimensions of the keys and values, then concatenating them into the final values as seen in Figure 8. A *residual connection* is employed from the inputs to the output of the multi-head attention mechanism, adding them together, then finally normalising the results. Those results are then inputted into a similar mechanism, substituting the multi-head attention mechanism with an FFNN [38]. This would be considered the second part of the complete transformer block, many of which make up the entire transformer model.

Complete transformer blocks were then deployed into Encoders-Decoders, producing the overall model. Encoder-Decoders are fundamentally made up of two blocks: one that reads the input sequence and encodes it into a fixed-length vector, while the other decodes that fixed-length vector and outputs the predicted sequence [6]. In Transformers, however, that sort of architecture deviates into the model seen in Figure 8. The transformer block was then stacked six times over for the encoder and the decoder. One variation was made for the decoder within the sub-model, wherein an extra multi-head attention mechanism was added with the output of the encoder stack being taken in as its input. Additionally, self-attention within the decoder was changed to disallow the computation of ensuing positions, thus predictions would only be dependent on outputs of a previous position [38].

In the end, transformers were able to achieve parallelism that was not discovered well enough beforehand. Subsequently, progress within NER also improved down the line with the new transformer-based models.

4.1.2 Transformer-Based Models

Understanding how transformers improved the state of Named Entity Recognition means knowing what sorts of architectures were developed following its release, as well as how well they performed. One example of that is the widely used masked language model, *BERT* [8].

Traditionally, language models were only unidirectional; using either the left or right context. That mostly relied on the need for probability distributions that were robust, and the need to avoid cross-talk, where a word representation would be concatenated with itself in a higher layer. Nevertheless, the understanding of languages typically requires bidirectional context, so the need for it never disappeared. The solution came with the use of the transformer encoder with the addition of masking a percentage of the input words, then predicting the masked words. The language model is then simultaneously trained on the left context and on the right context and finally concatenated. This step, called pre-training, enables BERT to learn the underlying rules of the language and context being used.

Following pre-training, BERT is then fine tuned in order for the masked language model to learn the task at hand, in this context the task would be NER. Fine tuning includes the removal or resetting of the existing BERT output layer, swapping it out with an empty output layer, set to zero. Now, with BERT's hidden layers still containing the information it learned in pre-training, the model is trained on the NER task, enabling the output layer to learn how to label named entities.

BERT, by itself, was unable to overcome the Flair embeddings model by Akbik et al. (2018), though it came close with an F1-score of 92.80 for BERT Large against Flair embeddings' F1-score of 93.04. It was rather architectures that built onto BERT that eventually overcame Flair embeddings and began to set higher state of the art F1 scores for Named Entity Recognition. The first of such BERT-dependent models that broke the ceiling placed by Flair embeddings was Cross-sentence Contexts [21]. In their paper, Luoma et al. (2020) utilised BERT alongside the learning of cross-sentence context to achieve better results in the labelling of named entities. Predictions are made for one sentence at a time using BERT's set window size. Different predictions are then made by moving the sentence around, with majority voting being used to pick the more probable labels for the tokens [21].

Another model that achieved a higher score than the best F1-score at the time of its publishing was LUKE, also based on BERT and transformers [40]. Yamada et al. (2020) trained their model using a pre-trained task, which was based on BERT. Alongside that, they also proposed an *entity-aware* self-attention mechanism that takes into account the types of tokens when computing the attention:

$$y_i = \sum_{j=1}^{n} \text{softmax}(\frac{Kx_j^\top Qx_i}{\sqrt{L}}) \tag{25}$$

for input vectors $x_1, x_2, \dots, x_n$ and output vectors $y_1, y_2, \dots, y_n$, where also $Q, K, V \in \mathbb{R}^{L \times D}$ [40]. This sort of model overcame both Flair embeddings and the previous best, Lumoa et al. (2020)'s Cross-sentence Context.

4.1.3 Working with Embeddings

Tying into transformers is the state of the art model with the current highest F1-score, ACE [39]. The idea brought on by Wang et al. (2021) built on previous work seen in earlier sections, such as in Section 3.1, as well as adding BERT onto it. The premise is that concatenating different pre-trained contextualised embeddings creates powerful models for prediction. Concatenations thus far have been done through manual selection of the pre-trained embeddings, depending on the task and collection of embeddings. Automated Concatenation of Embeddings proposes a solution for that by automatically sampling the most effective individual embedding types available for the task-at-hand, then updates its set of choices based on a certain response. This sort of reinforcement learning optimises to find the best possible concatenation for use, all depending on the accuracy of the different models.

4.2 Evaluating the Progress

Similar to the results in Section 3.3.2, BERT and models that were based on transformers have shown improvements in the state of the art benchmarks on their release. Though BERT and its variations did not set the highest F1-score for CoNLL-2003, models that used BERT pre-trained language models did so down the line.

Model	**NER-English F1-Score**	**NER-German F1-Score**
Bi-LSTM-CRF$_{\text{[Lample et al. (2016)]}}$	90.9	83.8*
Bi-LSTM-CRF$_{\text{[Akbik et al. (2018)]}}$	93.1*	88.3*
BERT$_{\text{[Devlin et al. (2018)]}}$	92.8	n/a
Cross-Sentence Context$_{\text{[Luoma et al. (2020)]}}$	93.7*	87.3
LUKE$_{\text{[Yamada et al. (2020)]}}$	93.9*	n/a
ACE$_{\text{[Wang et al. (2021)]}}$	94.6*	88.4*

Table 5: Comparing different models for the CoNLL-2003 English and German datasets in Named Entity Recognition, all of which were considered state of the art at the time of their publishing [13][5][18][27][28][1][8][5][39]. * = highest F1 score at the time of publishing.

One very important aspect to note in regards to all of the different models shown in Table 5 is that as the years progress, and as new, differing models are presented, the corpra used for training increase in size. This has the potential to inflate a model's performance beyond changes in the architecture itself. Therefore, it is essential to keep that in mind when comparing the differing models shown in the aforementioned table.

5 Conclusion

The rapid progress in the fields of Natural Language Processing and Named Entity Recognition continues to push for higher standards within both fields. As seen throughout this paper, such progress does not come alone but is rather tied to advancements in general machine and deep learning models and mechanisms, such as the evolution in neural architectures. With the continued development of Named Entity Recognition comes also the evolution in knowledge and information retrieval, as well as other areas such as algorithms' abilities to answer questions efficiently and effectively.

Moreover, and though it is clear that Transformers may generally outperform Bidirectional LSTM-CRF models, it is unknown at this point in time whether newer models would enable further development in Named Entity Recognition, and whether other models, such as Convolutional Neural Networks, would aid in that development. On top of that, and despite the irrevocable progress in the field, it is also unknown whether a certain threshold would be met which newer models, trained on bigger corpra, would be unable to finally cross, thus instituting a maximum threshold for Named Entity Recognition as a whole.

References

[1] Alan Akbik, Duncan Blythe, and Roland Vollgraf. Contextual string embeddings for sequence labeling. In *Proceedings of the 27th International Conference on Computational Linguistics*, pages 1638–1649, August 2018.

[2] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

[3] Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomas Mikolov. Enriching Word Vectors with Subword Information. *Transactions of the Association for Computational Linguistics*, 2012.

[4] John Bridle. Probabilistic interpretation of feedforward classification network outputs, with relationships to statistical pattern recognition. In *Neurocomputing*, pages 227–236, 1990.

[5] Jason Chiu and Eric Nichols. Named entity recognition with bidirectional LSTM-CNNs. *Transactions of the Association for Computational Linguistics*, 2016.

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

[7] Hong-Jie Dai, Po-Ting Lai, Yung-Chun Chang, and Richard Tzong-Han Tsai. Enhancing of chemical compound and drug name recognition using representative tag scheme and fine-grained tokenization. *Journal of cheminformatics*, 2015.

[8] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 4171–4186, June 2019.

[9] Yoav Goldberg and Omer Levy. word2vec Explained: deriving Mikolov et al.'s negative-sampling word-embedding method. *arXiv preprint arXiv:1402.3722*, 2014.

[10] Alex Graves. Generating sequences with recurrent neural networks. *arXiv preprint arXiv:1308.0850*, 2013.

[11] Geoffrey Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. *arXiv preprint arXiv:1207.0580*, 2012.

[12] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 1997.

[13] Zhiheng Huang, Wei Xu, and Kai Yu. Bidirectional LSTM-CRF models for sequence tagging. *arXiv preprint arXiv:1508.01991*, 2015.

[14] Michael Jordan. Serial order: A parallel distributed processing approach. Technical report, California Univ., San Diego, La Jolla (USA). Inst. for Cognitive Science, March 1986.

[15] Dan Jurafsky and James Martin. *Speech and Language Processing.* https://web.stanford.edu/ jurafsky/slp3/, 2022.

[16] Andrej Karpathy, Justin Johnson, and Li Fei-Fei. Visualizing and understanding recurrent networks. *arXiv preprint arXiv:1506.02078*, 2015.

[17] John Lafferty, Andrew McCallum, and Fernando Pereira. Conditional random fields: Probabilistic models for segmenting and labeling sequence data. 2001.

[18] Guillaume Lample, Miguel Ballesteros, Sandeep Subramanian, Kazuya Kawakami, and Chris Dyer. Neural architectures for named entity recognition. In *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 260–270, June 2016.

[19] Wang Ling, Tiago Luís, Luís Marujo, Ramón Fernandez Astudillo, Silvio Amir, Chris Dyer, Alan Black, and Isabel Trancoso. Finding function in form: Compositional character models for open vocabulary word representation. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, 2015.

[20] Wang Ling, Yulia Tsvetkov, Silvio Amir, Ramon Fermandez, Chris Dyer, Alan Black, Isabel Trancoso, and Chu-Cheng Lin. Not all contexts are created equal: Better word representations with variable attention. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pages 1367–1372, 2015.

[21] Jouni Luoma and Sampo Pyysalo. Exploring cross-sentence contexts for named entity recognition with BERT. *arXiv preprint arXiv:2006.01563*, 2020.

[22] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word Representations in vector space. *Advances in Neural Information Processing Systems*, October 2013.

[23] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffrey Dean. Distributed representations of words and phrases and their compositionality. *Advances in Neural Information Processing Systems*, October 2013.

[24] Andrew Ng and Michael Jordan. On discriminative vs. generative classifiers: A comparison of logistic regression and naive bayes. In *Advances in neural information processing systems*, pages 841–848, 2002.

[25] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. On the difficulty of training recurrent neural networks. In *International Conference on Machine Learning*, pages 1310–1318, 2013.

[26] Jeffrey Pennington, Richard Socher, and Christopher Manning. Glove: Global vectors for word representation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing*, pages 1532–1543, 2014.

[27] Matthew Peters, Waleed Ammar, Chandra Bhagavatula, and Russell Power. Semi-supervised sequence tagging with bidirectional language models. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics*, pages 1756–1765, April 2017.

[28] Matthew Peters, Waleed Ammar, Chandra Bhagavatula, and Russell Power. Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies. In *Proceedings of NAACL-HLT*, pages 2227–2237, June 2018.

[29] Alec Radford, Rafal Jozefowicz, and Ilya Sutskever. Learning to generate reviews and discovering sentiment. *arXiv preprint arXiv:1704.01444*, 2017.

[30] Lance Ramshaw and Mitchell Marcus. Text chunking using transformation-based learning. In *Natural language processing using very large corpora*, pages 157–176. 1995.

[31] Lev Ratinov and Dan Roth. Design challenges and misconceptions in named entity recognition. In *Proceedings of the Thirteenth Conference on Computational Natural Language Learning*, pages 147–155, 2009.

[32] Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 1958.

[33] David Rumelhart, Geoffrey Hinton, and Ronald Williams. Learning representations by back-propagating errors. *Nature*, 1986.

[34] Erik Sang and Fien De Meulder. Introduction to the CoNLL-2003 shared task: Language-independent named entity recognition. *arXiv preprint cs/0306050*, 2003.

[35] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 2014.

[36] Ilya Sutskever, James Martens, and Geoffrey Hinton. Generating text with recurrent neural networks. In *International Conference on Machine Learning*, 2011.

[37] C.J. van Rijsbergen. *Information Retrieval.* Buttersworth, 1975.

[38] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems*, 2017.

[39] Xinyu Wang, Yong Jiang, Nguyen Bach, Tao Wang, Zhongqiang Huang, Fei Huang, and Kewei Tu. Automated Concatenation of Embeddings for Structured Prediction. *rXiv preprint arXiv:2010.05006*, 2021.

[40] Ikuya Yamada, Akari Asai, Hiroyuki Shindo, Hideaki Takeda, and Yuji Matsumoto. Luke: deep contextualized entity representations with entity-aware self-attention. *arXiv preprint arXiv:2010.01057*, 2020.