Expertise
and insight
for the future

Khanh Nguyen

Named Entity Recognition: Deep Learning with Automated Pipelines for Lead Processing

Metropolia University of Applied Sciences

Bachelor of Engineering

Information Technology

Bachelor's Thesis

6 May 2020

Author(s) Title Number of Pages Date	Khanh Nguyen Named Entity Recognition: Deep Learning with Automated Pipeline for Lead Processing 38 pages 6 May 2020
Degree	Bachelor of Engineering
Degree Programme	Information Technology
Specialisation option	Software Engineering
Instructor(s)	Olli Hämäläinen, Senior Lecturer

Over the past few years, various breakthroughs have been made in many artificial intelligence tasks due to the increasing popularity of artificial neural networks. Named entity recognition is a subtask of natural language processing, in which the aim is to detect and extract potential named entities from unstructured text. The goal of this thesis is to develop a functional Named Entity Recognition system using an artificial neural network for the company Vainu.

The end model was constructed by using different architectures of artificial neural networks, such as Recurrent Neural Network and Convolutional Neural Network. Some methods of transfer learning such as word embeddings were also applied. The trained model was then deployed as a microservice using Python and Docker. A training pipeline for the Named Entity Recognition model consisting of a continuous integration system with automated building and testing processes was also implemented.

Through many experiments and testing, the objective of this thesis was accomplished. The final model was able to perform the entity extracting task with high accuracy. With the new Named Entity Recognition application, Vainu gets a new AI that can be freely adapted to suit its requirements, increases the matching performance of the company and reduces the operation expense compared to using third-party software. The training pipeline was also implemented in a highly scalable way to ensure that new models for new languages can be added to the system with ease if necessary.

Keywords	NER, NLP, deep learning, CNN, RNN, ANN

Contents

List of Abbreviations

ANN	Artificial neural network.
CNN	Convolutional neural network. A specific type of network specialized in capturing local patterns of the input matrixes.
CRF	Conditional random field.
LSTM	A variant of RNN. This type of network has the ability to remember and forget sequences with great length.
NLP	Natural language processing.
RNN	Recurrent neural network. A special type of neural network that takes the input and calculates the output using the input in combination with its own hidden states. Mostly used for sequential input data.

1 Introduction

Vainu is a software as a service (SaaS) company founded in 2013. The company aims to build a large database of companies around the world with up-to-date and accurate information using artificial intelligence applications. The company collects and processes potential business leads from various public sources, such as internet articles, press releases, and financial statements. From these data, relevant information is extracted and used to update the related company profiles.

Named entity recognition (NER) is one of several natural language processing (NLP) tasks that involves detecting named entities from an unannotated chunk of text and classifying them into correct categories. Usually, entities such as organizations, people or locations are the main focus. NER is used in applications that involve keyword retrieval and analysis such as chat bots and search engines.

In order to identity the mentioned companies in the articles correctly, it is important to detect correctly all of the company names mentioned in the leads. Therefore, implementing a NER system with good performance is a vital step towards Vainu's goals. A good NER system can extract all the named entities from the input text with near-human performance while being much faster and cheaper.

Recently, neural networks and deep learning have gained a lot of popularity in many machine learning fields. Their ability to process data in any format (images, audios, texts) and detect complex patterns make deep learning the state-of-the-art machine learning algorithm today. Deep learning models, if provided with enough data, can learn to execute a wide variety of tasks, from simple digit recognition to more sophisticated ones such as speech recognition and self-driving cars.

The objectives of this thesis are as follows:

- To build a NER application for Vainu's lead processing system using neural networks.
- To implement a training feedback loop for the NER system, so that the system can re-evaluate and train itself automatically.
- To find the best deep learning model for the English language.

2 Theoretical background

The goal of this theoretical background is to explore the history of named entity recognition and some of the most common approaches for this task that have been made. Artificial neural network (ANN), its operation principle and some ANN variants that are used to implement the model for this thesis will also be introduced concisely.

2.1 Traditional approaches

2.1.1 Rule-based

A rule-based NER application detects named entities based on a set of rules implemented by software engineers. This system is usually implemented with the experiences and observations of linguists and language experts. These professionals can provide the rules which can be used to detect and classify named entities from the input text. (Hridoy, 2013).

As natural language rules are not machine-friendly, the software system built with this approach is usually very complicated and hard to maintain. A team of engineers with high competence is required for the implementation, because of the abstraction and complexity of the natural language rules.

2.1.2 Statistical models

Statistical models can be utilized on any task that requires data pattern detection including NER. The model, provided with enough input (text) and the output (named entities), can create a mapping from input to output, based on the statistical rules that it derived while observing the data. For NER, there were different approaches such as Hidden Markov Model (HMM) and Conditional Random Field (CRF). (Hridoy, 2013.) How these methods work is outside the scope of this thesis, but in general they were able to achieve remarkable accuracy and they are still being used nowadays.

The NER systems built using statistical models are easier to maintain, as there are no hard-coded rules that need to be updated manually. However, statistical models have

several drawbacks. They do not generalize well to natural language so their performance depends heavily on the amount of training data.

2.2 Artificial neural networks and deep learning

2.2.1 Artificial neural networks

Artificial neural networks (ANN) were created with the inspiration of biological brain neural networks. The average human brain consists of 86 billion neurons. These neurons form a large network of processing units that can handle information signals through electric pulses. (Azevedo et al., 2009.)

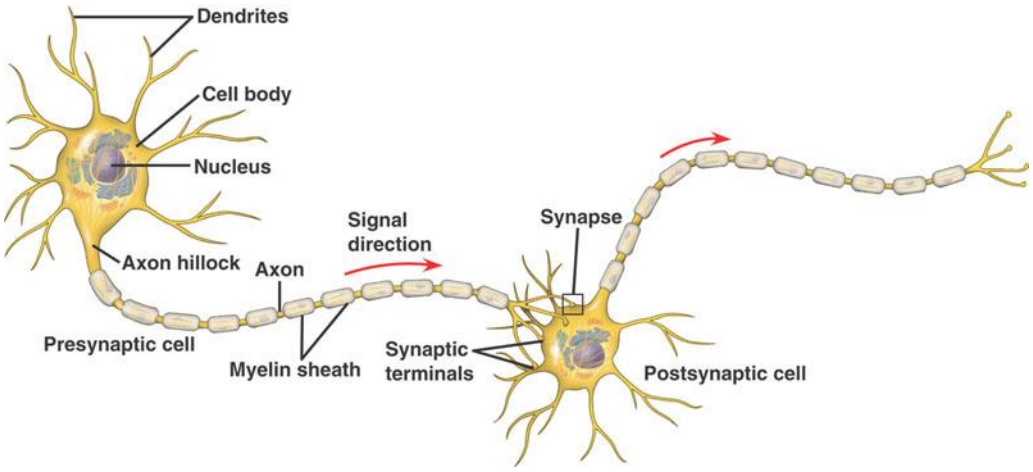

Figure 1. Brain neuron structure (copied from Gillam, 2015)

Figure 1 shows two connected biological brain neurons. A neuron receives the input signal from other neurons at its dendrites. The signal is then transmitted through the neuron body down to the synapse to meet the dendrites of the next neuron. However, the signal is only transmitted between the neurons if the signal strength surpasses a certain electrical threshold. By receiving, processing and transmitting different kind of signals continuously, the brain neural networks power all human thought processes, both conscious and unconscious.

To simulate the brain learning process and understand the underlying mechanism, many attempts have been made by scientists to replicate the neuron behavior with physical or mathematical models. The first artificial neuron was introduced by Warren McCulloch and Walter Pitts in 1943, using electrical circuits to model a primitive neural network. In

1949, Donald O. Hebb introduced the Hebbian Learning Rule, the foundation principle for artificial neural networks. In 1958, Frank Rosenblatt introduced the first perceptron, which highly resembles the modern perceptron, in his report “The Perceptron — a perceiving and recognizing automaton”. (Wang and Raj, 2017.)

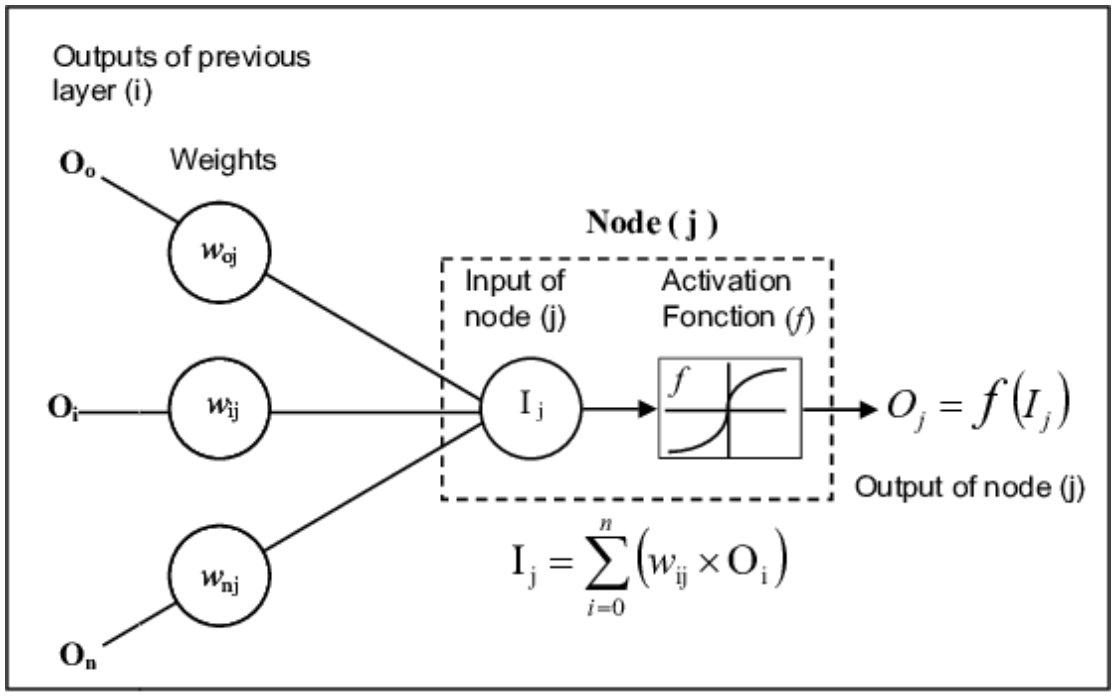

Figure 2. The basic element of a neural network: node computation. (Copied from Chokmani, 2017).

Figure 2 shows the structure of an artificial neuron (node) j. The edges connected to the input of the node are outputs from the nodes of the previous layer, with O_i representing the output from the node i. The circle on each edge represents a unique weight $w \in \mathbb{R}$, with w_{ij} being the weight from node i to node j. The node i takes Os and ws as inputs. The output of the neuron is calculated by the following equations:

$$I_j = \sum_{i=0}^{n} w_{ij} O_i + b \tag{1}$$

$$O_j = f(I_j) \tag{2}$$

The output of node j at layer k is calculated by taking all the outputs O from the preceding layers, usually $k-1$, and multiplying each output with the corresponding weight. The product is then summed across all the input nodes with the addition of the bias b to form I_j. The final output O_j is computed by applying a non-linear activation function f on I_j.

In an ANN model, the weights w and biases b are the parameters, which are usually initialized randomly. These parameters are optimized during the training process. ANN has a large number of parameters, making itself highly flexible. Moreover, it can be fitted on highly complex data.

The activation function f is usually a non-linear continuous function. One example is the Softmax function, which is usually used as the activation function at the output layer for a multiclass classification problem. The Softmax output is calculated by the following formula:

$$S(y_j) = \frac{e^{y_i}}{\sum_{j=1}^{K} e^{y_i}} \tag{3}$$

Softmax is a non-linear function that takes a vector of K real numbers as input, and outputs a probability distribution consisting of K probabilities. It takes the exponential function of each vector component and normalizes them by the sum of exponential functions. Softmax is also simple to differentiate, a necessary property in order to train the model efficiently.

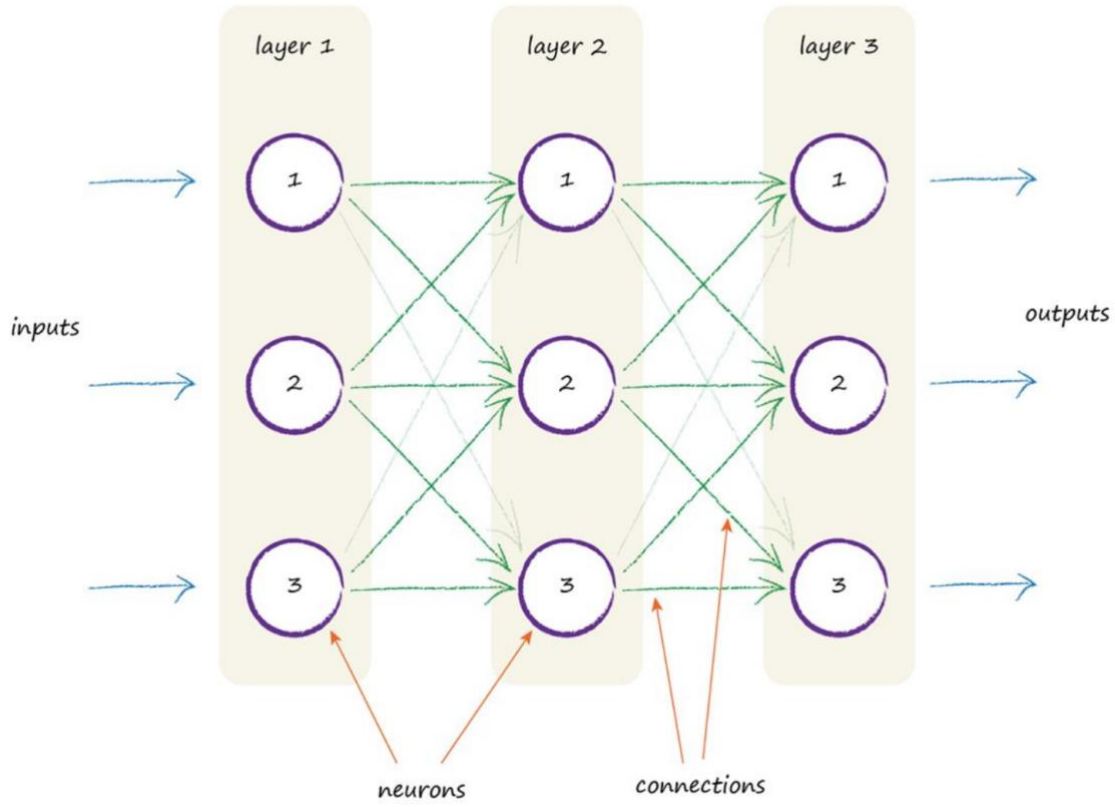

Figure 3. Neural network layers, neurons and connections. (Copied from Rashid, 2016).

Figure 3 shows the architecture of a simple ANN with five layers, including the inputs and outputs. The layers between input and output which are not directly observed are called hidden layers. The model from the figure has three hidden layers, each one consisting of three nodes. The term “deep learning” is used to describe any ANN that has more than two hidden layers.

2.2.2 Gradient descent

Gradient descent is an optimization method to find the minimum of a function. Given a function F and a point a where $F(a)$ is defined and differentiable, it is possible to find b with $F(b) < F(a)$ if $b = a - \nabla F(a)$, with $\nabla F(a)$ being the derivative of F at point a. In other words, if:

$$a_{n+1} = a_n - \gamma \nabla F(a) \tag{4}$$

Then $a_{n+1} \leq a_n$, if γ is small enough.

To a certain degree, an ANN can be represented as a non-linear mathematical function F with a large number of parameters. Given a dataset of input x and output y, the goal is to find F which can satisfy $F(x_i) \approx y_i, \forall i$. This can be achieved by initializing F with random parameters and gradient descent can be used to optimize the parameters using samples of x and y.

Gradient descent can only be used to minimize a function, so in order to apply gradient descent, another function J (loss function) is needed. The loss function takes the model output $F(x_i)$ and the actual output y_i as input and acts as an error measure function for F. In other words, if $J(F(x_i), y_i) \approx 0, \forall i$, then $F(x_i) \approx y_i, \forall i$. In practice, ANNs are trained by using gradient descent to find the parameters of F which can minimize $J(x)$. Depending on the kind of task and output the model has to solve, loss function has different forms to penalize the neural network.

2.2.3 Backpropagation with neural network

The first version of backpropagation was introduced in the 1970s by Seppo Linnainmaa as a general optimization method for performing automatic differentiation of complex nested functions (Linnainmaa, 1970). However, it was not adopted well by the community until 1986 when David Rumelhart, Geoffrey Hinton, and Ronald Williams proposed using backpropagation as the optimization method to describe how artificial neural networks are trained. Backpropagation quickly became the de facto standard to train neural networks up until today (Nielsen, 2015).

Backpropagation helps train neural networks by calculating the partial derivatives $\frac{\partial J}{\partial w}$ and $\frac{\partial J}{\partial b}$ of every weight w and bias b with respect to the cost function J. The algorithm is able to calculate $\frac{\partial J}{\partial w}$ and $\frac{\partial J}{\partial b}$ by utilizing the chain rule and product rule from differential calculus. By updating the ANN parameters with the derivatives using Gradient Descent, the network gradually reduces the output of the loss function J. When J decreases its value, the ANN increases its output accuracy and gradually learns the task.

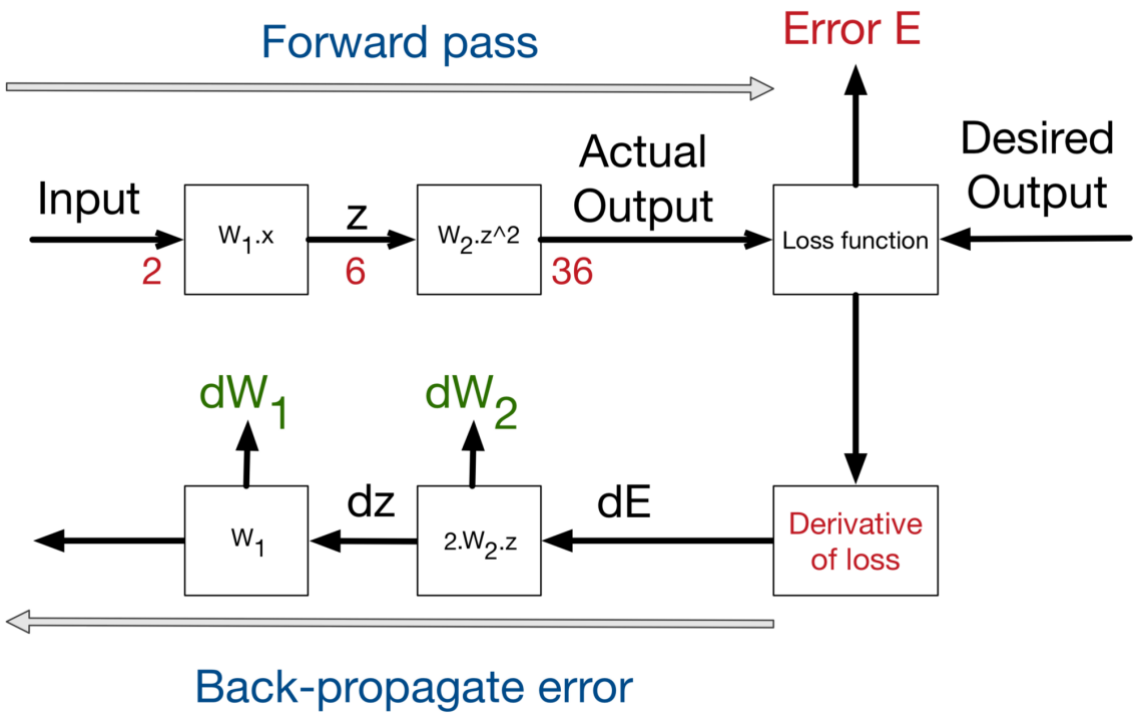

Figure 4. Diagram of forward and backward paths. (Copied from Moawad, 2018).

Figure 4 shows how backpropagation operates with neural networks. Before the training process, the ANN model is initialized with random parameters. The training process starts with the forward pass by feeding the model a training sample to calculate the model output. The loss function uses the model output and the desired output to calculate the error E. Afterwards, the partial derivative of E is calculated by taking the derivative of the loss function with respect to the model's output. By using backpropagation, the gradient is calculated gradually from the output layer back to each parameter in every layer of the ANN up until the input layer. Finally, every weight and bias in the model is updated with Gradient Descent.

2.2.4 Convolutional neural networks (CNN)

Convolutional neural networks (CNN) are neural networks that use a linear mathematical operation called convolution in place of general matrix multiplication in at least one of their layers. CNN is usually used for processing data that has a known grid-like structure. For instance, time-series data, where data are provided at regular time intervals, and image data, which can be represented as a 2-D matrix of pixels, are structures that can be processed by CNNs. (Goodfellow, 2016).

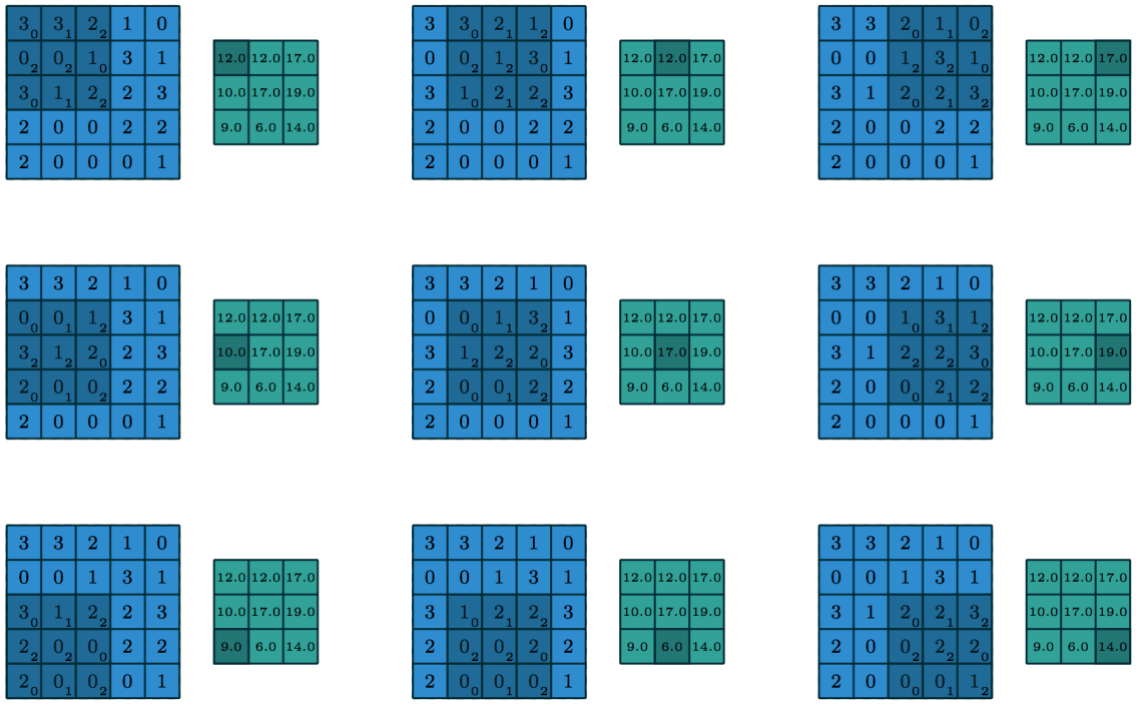

Figure 5. Computing the output values of a discrete convolution. (Copied from Dumoulin and Visin, 2018).

Figure 5 shows a convolution operation between a 5x5 matrix as the input and a 3x3 matrix kernel. The blue 5x5 matrix is the input feature map, the dark 3x3 region on the input shows the location of the kernel on the input and the green 3x3 matrix is the output. The regions that the kernel slides over during the computation are called local receptive fields. From left to right, when the operation starts, the kernel slides over the input feature map and calculates the output results by performing an elementwise multiplication between the kernel and the local receptive field and takes the sum of the result to create the convolution output. This convolution operation results in a 3x3 matrix.

According to Goodfellow (2016), CNN has three main properties: sparse interactions, parameter sharing, and equivariant representations. Unlike normal ANNs, where every node between adjacent layers are connected, every element of a convolution operation output is not affected by the whole input, only a part of it. This leads to the sparse interactions between the input and output. Because of the small size of kernels, the number of parameters that need to be stored and the number of operations that need to be computed are reduced. Although sparse interactions only allow a small number of input units to affect the output, it is possible to create a larger interaction by stacking multiple convolution layers on top of each other.

When the kernel slides across the input feature map, the kernel weights are used to calculate the output multiple times. In contrast to normal ANNs, where different parts of the input are connected with different parameters, CNNs allow the kernel weights to connect with the input at different locations. This property is called parameter sharing. With parameter sharing, the CNN kernels can process all receptor fields across the whole input feature map. The small and basic patterns from the input data are detected in this stage.

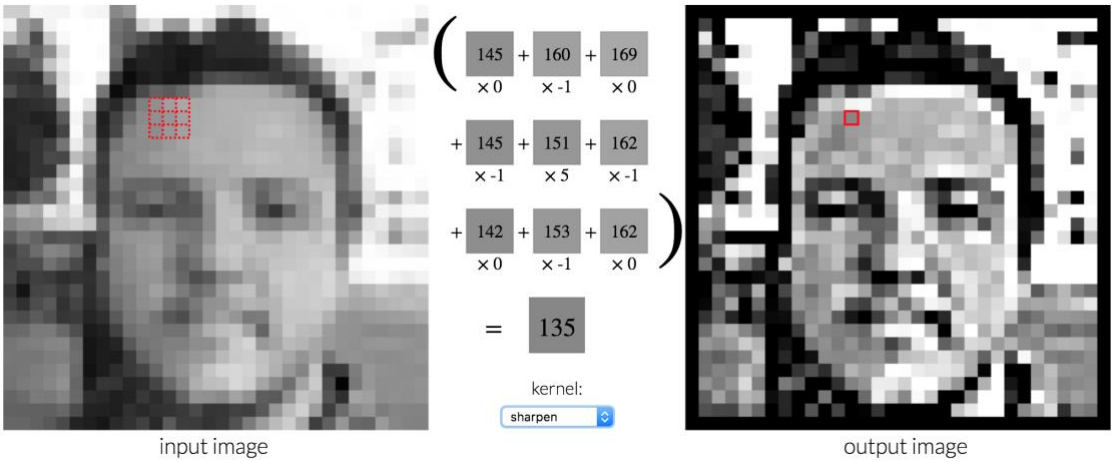

Figure 6. Image processing using sharpen kernel. (Copied from Powell, 2015).

Figure 6 shows an example of a convolution operation using a sharpening matrix kernel. The image on the right is formed by the convolution between the matrix $\begin{pmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{pmatrix}$ and the input image on the left with zero padding. The red square on the input image is the location that the kernel currently slides on and the pixel on the output image has the value of the convolution output. This results in another image with much sharper edges between each image region. There are many more types of kernels in image processing, each with its own usage. Like other neural network parameters, CNN kernels are not predefined, but initialized randomly at the start and optimized during the training phrase.

The CNN is used as a part of the character encoding of the final ANN model, generating the word's encoded vector from its character embedding matrix. By using small multiple kernel sizes, the CNN layer can detect potential semantic patterns from the groups of characters that formed the words. The CNN layer outputs are then combined with other layer outputs to form the word representation vectors.

2.2.5 Recurrent neural networks (RNN)

RNN is a variant of neural networks specialized in processing sequential input. Normal neural networks are typically combinational, which means the output is calculated using only the present input. On the other hand, an RNN has its own memory and calculates its output using both this memory and the present input. As a result, the output of an RNN depends on the present input and the history of the input. This property makes RNNs the ideal choice for processing sequential data. In fact, RNNs are mainly used in tasks where the input follows sequential structure such as NLP, audio recognition, time series analysis, etc.

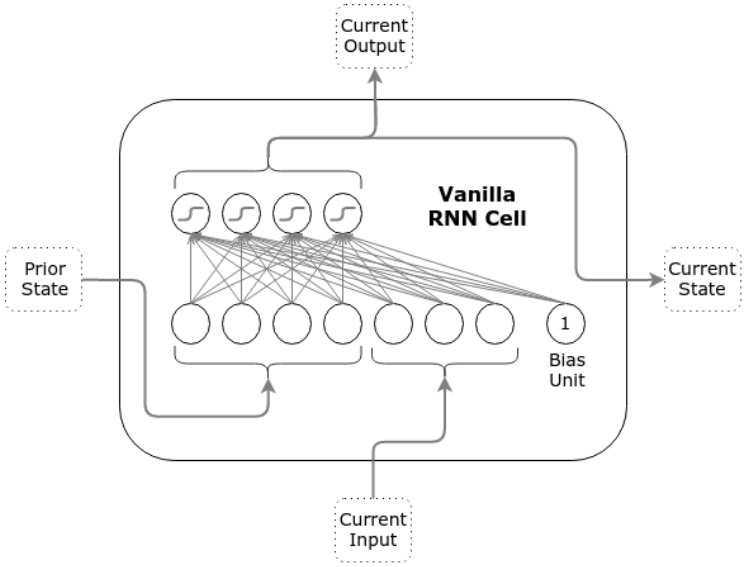

Figure 7. Basic RNN cell (copied from Pitis, 2016)

Figure 7 displays the structure of a basic RNN cell. At every timestep i, with $i > 0$, the RNN takes both the input x_i and its previous hidden state s_{i-1} to calculate the output y_i and the hidden state s_i. The hidden state s_i is also passed on to the next timestep to compute y_{i+1} and s_{i+1}, up until the last input timestep. Because network weights and bias are reused at every timestep, RNN also has the parameter sharing property.

In practice, a basic RNN is rarely used in modern deep learning applications. Because the output is calculated by using the input in every time step, the simple structure of a basic RNN causes many gradient-related problems, most noticeably vanishing gradients, which reduces the number of the gradients for the early time steps to arbitrarily small, making the network very difficult, even impossible to train (Bengio, 1994).

In 1997, the first concept of the long short-term memory (LSTM) network was introduced. LSTM is a variant of RNN which was designed to avoid the problem with long-term dependencies while still being able to process the sequential input effectively. (Hochreiter and Schmidhuber, 1997.)

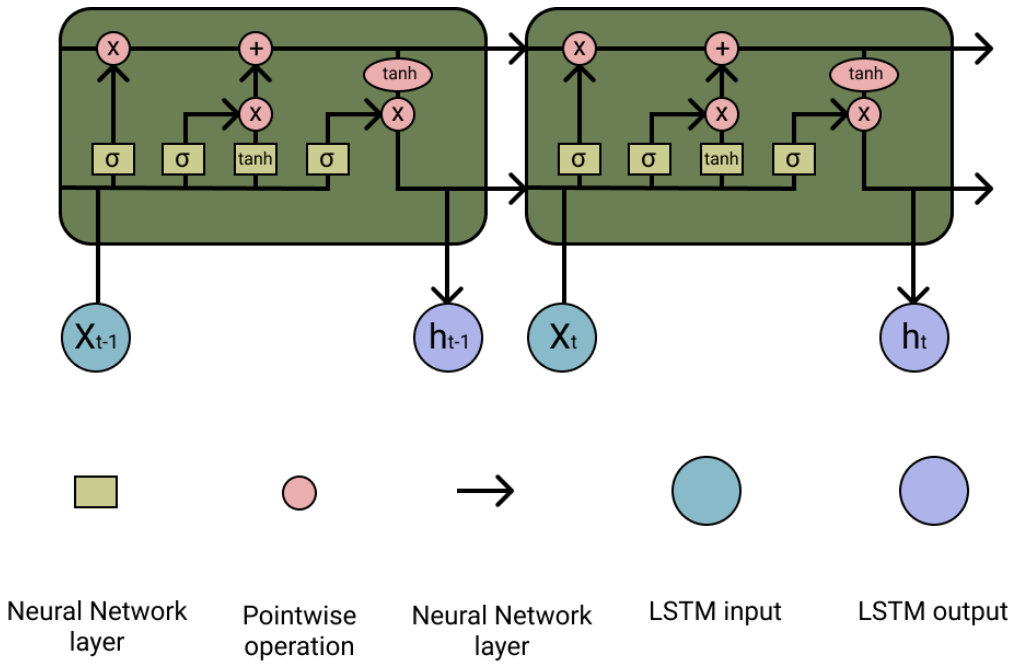

Figure 8. Long short-term memory (LSTM) network cell and notations (Adapted from Colah, 2015)

Figure 8 displays an LSTM cell architecture. From left to right is the LSTM cell state of the two timesteps $t-1$ and t. The horizontal arrows connecting one timestep to another on the top of the diagram are the cell states C_t. For each timestep t, the LSTM network can modify its cell state C_t by adding, subtracting or scaling the information it keeps in the cell state depending on the input at that timestep X_t, the cell state from the previous timestep C_{t-1} and the previous output h_{t-1}. Each yellow square is a neural network layer.

The neural network layer in the bottom left of the diagram is called the forget gate layer. It takes both X_t and h_{t-1} as input, decides which element from the concatenated matrix to keep by running the output through the sigmoid function σ to create the forget matrix f_t:

$$f_t = \sigma\left(W_f . [h_{t-1}, X_t] + b_f\right) \quad (5)$$

where W_f is the weight matrix and b_f is the bias vector. The forget matrix f_t has the same size as the cell state C_t, with every element ranging from 0 to 1. The cell state C_t is then multiplied elementwise with the forget matrix f_t to decide which element from the old state should be kept, and which should be forgotten:

$$C_t = f_t * C_{t-1} \tag{6}$$

The neural network layer on the right of the forget gate layer is called the input gate layer. It uses the same input as the forget gate layer with a different set of parameters. Its output, the input matrix i_t is calculated as:

$$i_t = \sigma(W_i.[h_{t-1}, X_t] + b_i) \tag{7}$$

where W_i and b_i are the weights and the bias of the input gate layer. Although the calculation steps are similar to f_t, i_t is used to decide which element should be added to the cell state C_t.

This LSTM cell contains another layer, which is marked as the tanh layer next to the input gate layer. This layer uses the tanh activation function to calculate the new cell state candidate $\tilde{C}_t$:

$$\tilde{C}_t = \tanh(W_C.[h_{t-1}, X_t] + b_C) \tag{8}$$

W_C is the weight matrix and b_C is the bias of the input gate layer. Next, the current cell state C_t will be updated by adding $\tilde{C}_t$, scaled by the input gate output i_t:

$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t \tag{9}$$

After this step, the calculation of C_t is completed. The final step is to calculate the output h_t of the current timestep, using C_t. Again, the sigmoid function is used to calculate the output matrix o_t. h_t is then calculated by multiplying the output matrix o_t with the current cell state:

$$o_t = \sigma(W_o.[h_{t-1}, X_t] + b_o) \quad (10)$$

$$h_t = o_t * \tanh(C_t) \quad (11)$$

where W_o and b_o are the output gate parameters.

2.2.6 Linear conditional random fields (CRF)

The goal of the ANN model built in the study described in this thesis is to detect any potential named entities from the input sequence. In other words, the model reads the input text and predicts whether each word in the text belongs to a named entity. If it is, the model will classify the named entity into the correct categories. Since each word can be classified as one of many categories, this task can be labeled as a multiclass classification problem.

This thesis uses the CoNLL-2003 Shared Task English dataset as the benchmark method for the English NER model. The dataset consists of 22,137 annotated sentences. Each sentence is tokenized into a list of words and each word has a label which indicates whether that word is inside a named entity or not. If that word is a named entity, the label also indicates the entity category. The CoNLL-2003 dataset includes named entities of four types: Person (PER), Organization (ORG), Location (LOC) and miscellaneous names (MISC) (Sang and Meulder, 2003).

For normal multiclass classification problems, the Softmax function is usually used as the activation function in the output layer to produce the probability distributions of the potential labels. However, this is not the best approach for NER, because Softmax only computes the output probability distribution for each tag based on its local input. In other words, when computing the probability distribution for one word, it only uses that word's features, while information about its neighboring words is not considered. This method is not optimal, because the output at each word is not independent. Using Softmax will most likely make the model skip over potential important information at other words.

Given a sequence of words $w_1, w_2, \ldots . w_m$; a sequence of score vectors $s_1, s_2, \ldots s_m$ and a sequence of tags $y_1, y_2, \ldots y_m$, a linear-chain CRF defines a global score $C \in \mathbb{R}$:

$$C(y_1, y_2, \dots y_m) = b[y_1] + \sum_{t=1}^{m} s_t[y_t] + \sum_{t=1}^{m-1} \boldsymbol{T}[y_t, y_{t+1}] + e[y_m] \qquad (12)$$

Where $\boldsymbol{T}$ is the transition matrix that captures the dependencies between two neighboring timesteps. $\boldsymbol{T}[y_t, y_{t+1}]$ represents the transition score from y_t to y_{t+1}. b is the beginning score vector with m components and $b[y_1]$ represents the score for beginning the sequence with the tag y_1. Likewise, $e[y_m]$ represents the score of ending the sequence with the tag y_m. Both e, b and $\boldsymbol{T}$ are trainable parameters. The score vector for each input word is calculated by the model for each word and used as input for the CRF layer.

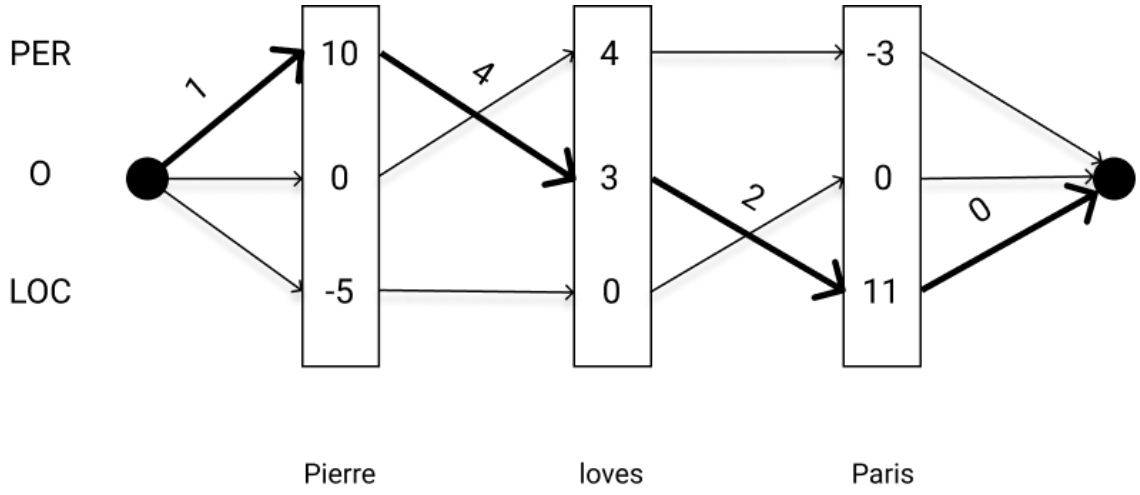

Figure 9. CRF tagging scheme. (Adapted from Genthial, 2017).

Figure 9 shows a simple case of a linear chain CRF when tagging a text sequence. The two black dots mark the beginning and the end of the text. The score vector s of every word is displayed as the rectangle above the word, with each component representing the score of that word for the possible tags on the left. The total number of tags is also the size of s. The arrows between each tag symbolize the transitions, and the number above each arrow is the transition score from the transition matrix $\boldsymbol{T}$. A tag sequence is formed by a collection of transitions from the beginning of the sequence, with the score equal to the sum of all transition scores component. CRFs iterate through possible tag sequences for the input, calculates the score for each sequence and outputs the one with the highest score. The score vector s is generated by the ANN. e, b and $\boldsymbol{T}$ are parameters that are optimized based on the data that the model observes.

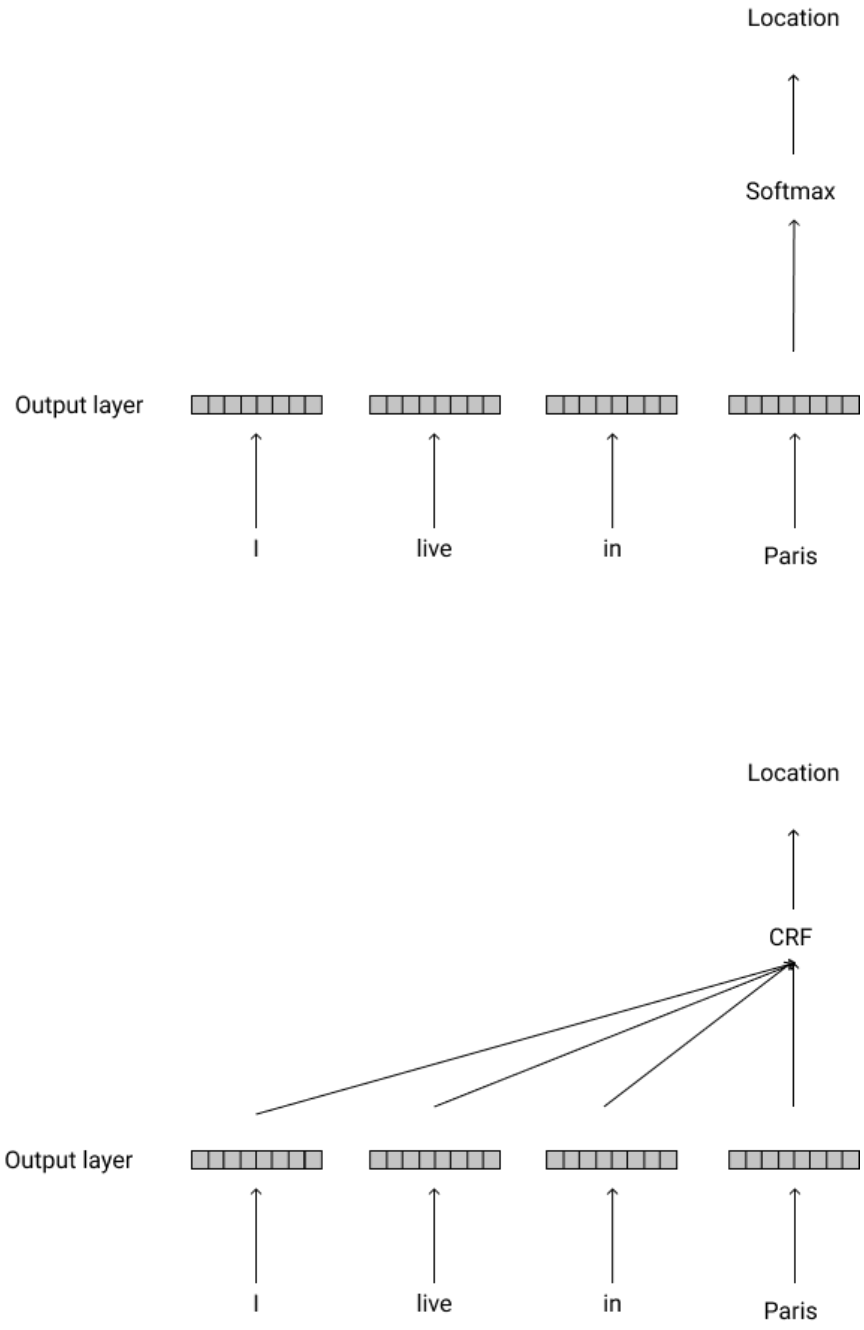

Figure 10. Tagging example: CRF vs Softmax

Figure 10 shows the difference between CRF and Softmax when classifying the word “Paris” from the input sequence “I live in Paris”. Softmax computes the output using only the features generated for the word “Paris”, so the model can learn “Paris” is a location name. CRF, on the other hand, takes the input information from not only the word “Paris” but also from the rest of the sequence. So not only the model learns that “Paris” is a location, but it can also learn that the word coming after “I live in” is more likely to be a

location. In other words, CRF improves the ability of the model to learn and generalize, especially for sequence tagging.

2.3 Summary of theoretical background

The goal of this theoretical background was to explore the history and approaches of Named Entity Recognition as well as to introduce the readers to some concepts and basic structures of artificial neural networks.

Named Entity Recognition has always been a challenging task for machine learning because of its machine unfriendly nature. It is shown from many experiments that the state-of-the-art Deep Learning models with their flexibility and ability to process any form of digital data structures has already surpassed their predecessors in performance by a high margin, not only in NLP but also in other tasks.

3 Implemented model architecture

All RNN networks used in the model for this thesis are bidirectional LSTM networks, which consist of two LSTM layers. One processes the input in forward and the other in backward direction. The output of each timestep from the forward and backward RNN networks are then concatenated to form the bidirectional outputs. Unlike the unidirectional LSTM, where the network only has access to the past timesteps, bidirectional LSTM can use information from both the past and the future timesteps.

The architecture of the final ANN model can be divided into three main components: input, feature extraction and output. The model receives its input from the input layer. The word feature vectors are derived from the input during feature extraction stage. Finally, the extracted features are processed to form the sequence context vectors, which are used as inputs for the CRF layer. The CRF layer uses the context vectors to predict the final output.

3.1 Input layer

To make a decision whether a word from a sentence is an entity or not, one usually takes the semantic information and the morphologic information of that word into consideration. Likewise, the ANN model needs to be able to process the data to make accurate predictions. The morphology of a word can be constructed from the characters that it contains and their order. Semantic information is a more challenging problem as there are no straightforward methods to perfectly quantize this information. However, there have been several approaches that can partially solve this challenge. They will be explained later in this chapter.

The inputs for the ANN include the character indices and the token indices. Each input sequence s with m words (tokens) is transformed into a vector of token indices $t \in \mathbb{N}^m$ and a matrix of character indices $\boldsymbol{C} \in \mathbb{N}^{m \times n}$. The token index vector t has the same length as the input sequence, with each element t_i representing the index of the word at position i from s, following a predefined vocabulary. If the input token is not in the vocabulary list, a wildcard token will be used as replacement. The character indices matrix $\boldsymbol{C}$ has m rows and n columns, where $\boldsymbol{C}_{ij}$ represents the character index of the jth character of the ith word. n is the maximum length of the tokens in s. Rows with length less than n are padded with 0 at the end.

3.2 Feature extraction

3.2.1 Word embedding

In image and audio domain, ANN models can process the digital data directly because these data usually contain all the necessary information. For image recognition tasks, an ANN model can process the raw pixel values directly as the input. The same also happens with speech recognition system, where the model can use the audio signal amplitude and frequency as the input. However, this property does not apply to words. In machine language, words are represented as discrete symbols which do not carry any useful information, especially in semantic sense. Since the word symbols are discrete, the model does not have any means to exploit the relationships between different words and generalize the learning process. For instance, when the model learns the word “machine”, it cannot use the new information to update its understanding of other related words such as “washing machine” or “robot”. Without generalization, the model has to learn each word one by one, which increases the learning time and the required training data significantly. Word embedding was created in order to tackle this problem.

Word embedding is a mapping technique that maps a word from a predefined vocabulary to a distributed and dense word vector with fixed length. The word vectors are not manually picked, but rather learned by different methods (Schnabel, 2015).

In 2013, Tomas Mikolov and his team from Google introduced word2vec, a word embedding algorithm which uses a simple ANN model to generate and optimize word vectors using a large text corpus (Mikolov, 2016). The team created a two-layer neural network which learns to predict the next most likely word from the input word. Before the training phrase, the model initialized random real vectors as projections for each word it came across. During the training phrase, the vectors are updated with gradient descent and backpropagation, in order to give the best prediction. Despite its simplicity, the word vectors trained with this method are capable of capturing the word's contextual and semantic information as well as its relationship with other words. Some other approaches have been proposed since then. Most of them were able to reach remarkable results such as GloVe, Fasttext, etc. Most of them use neural networks as a means to optimize the word vectors.

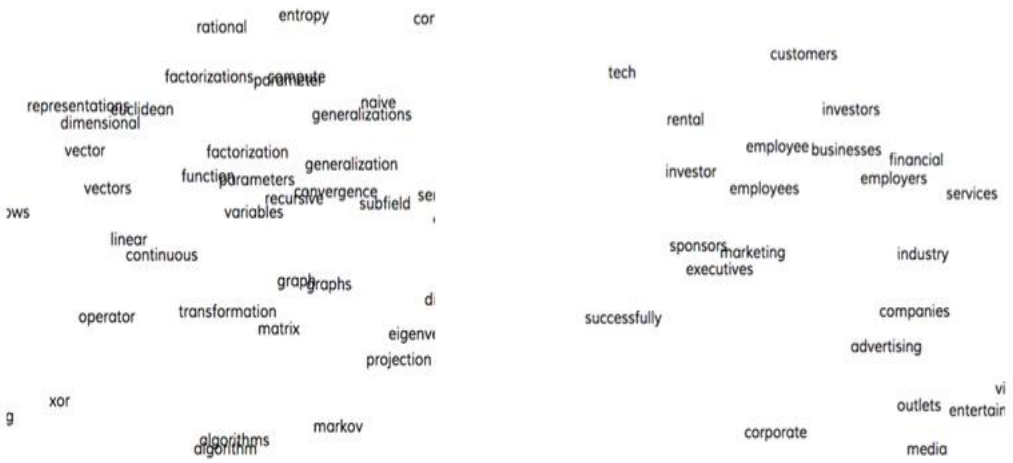

Figure 11. Clusters of semantically similar words emerging when the word2vec vectors are projected down to 2D using t-SNE. (Copied from Heuer 2016).

Figure 11 displays word2vec's 2-dimensional projection. Several word clusters can be observed. Words in a cluster share at least one common feature: the semantic information or the connection in the word context. More clusters have been found while observing word2vec vectors such as occupations, cities, numbers, countries, etc. By using word embeddings as input, the model can detect the relationship between the words of every cluster and generalize better.

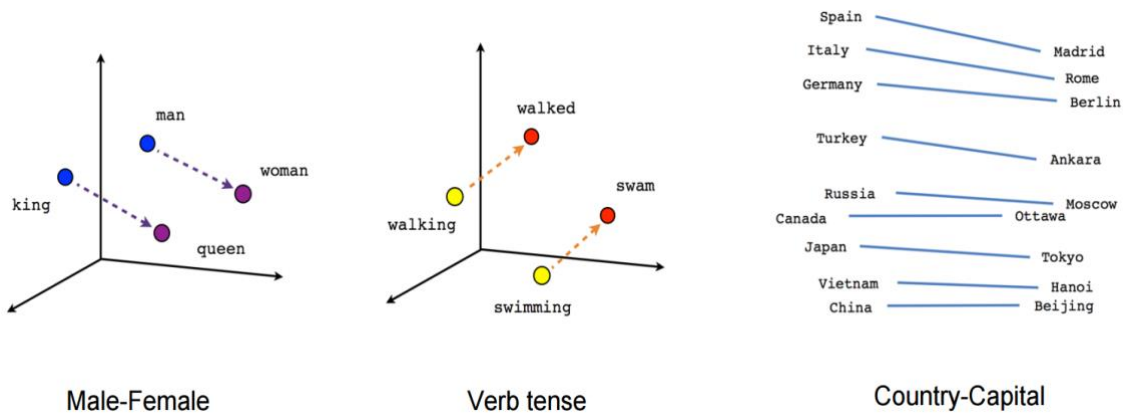

Figure 12. Relationships of word vectors (Copied from Google, 2018)

Figure 12 shows some relationship observations in trained word vectors using the word2vec algorithm. The semantic relationship between gender-specific words such as "king" and "queen", "man" and "woman" represented by the vector difference between

them are very similar. This also happens with verb tenses or Country-Capital relationship although the vectors can slightly differ, depending on the dataset that the word embeddings were trained on. Although they are not perfect, word embeddings do provide the ANN model a means to capture the natural language semantic information.

The word embedding vectors used in neural networks can be randomly initialized or pre-trained. The model's parameters are then optimized on top of the embedding layer, thus reducing the task complexity and increasing the speed of the training process. This is called "transfer learning". Since the size of CoNLL-2003 dataset (see Sang and Meulder 2003) is quite limited, pre-trained word embeddings were used. Facebook's FastText pre-trained word vectors are used in the word embedding layer (Joulin, 2016). Fasttext has a high volume of vocabulary and is available for many languages. The English vocabulary size is 2,000,000. This guarantees that most of the English words that might appear in the input are contained.

In neural networks, an embedding layer is a mapping layer where input words are embedded into a continuous vector space. In the ANN model for this thesis, the word embedding layer is where an input sentence with n words is converted into an embedding matrix with n rows and 300 columns, formed by stacking the word vector of each word in the sentence on top of each other.

During the training, the word embedding layer is frozen, which means the backpropagation does not update this layer, keeping the word vectors intact. This reduces the size of the model and increases the training speed by a good amount, making the model faster to train and easier to deploy.

3.2.2 Character embedding

As word embedding helps the model understand words, character embedding layers give the model meaningful representation of characters. Although characters do not have the same sparsity as words, having a character embedding layer can help the model understand the words better, especially if the input word does not appear in the word embedding vocabulary.

Character embeddings enable the ANN model to build character morphologic features of any word, given that the word is constructed with characters that the model was trained

on. In many cases such as phone numbers or emails, the character encoding layer can give important information to help the model make tagging decisions. The character embedding layer for the final model has a size of 64.

3.2.3 Character encoding layer

Encoding layers have originated from “auto encoder”, an ANN architecture that is used in data compression and feature extraction. The auto encoder model is trained with the task to encode/decode the input, in which it learns to project the input into a lower dimensional representation using the encoder and rebuilding the original input from the compressed projection with the decoder. In order to encode and decode data efficiently, the neural network is forced to learn to extract relevant information from its input in an accurate way.

In the ANN model of this thesis, the character encoding layer is used to extract the input word morphologic features from the character embedding matrix. To be more precise, the encoding layer creates the morphologic vector for the word from its character embedding matrix input while making sure that all the important information from character embeddings are encoded in that vector.

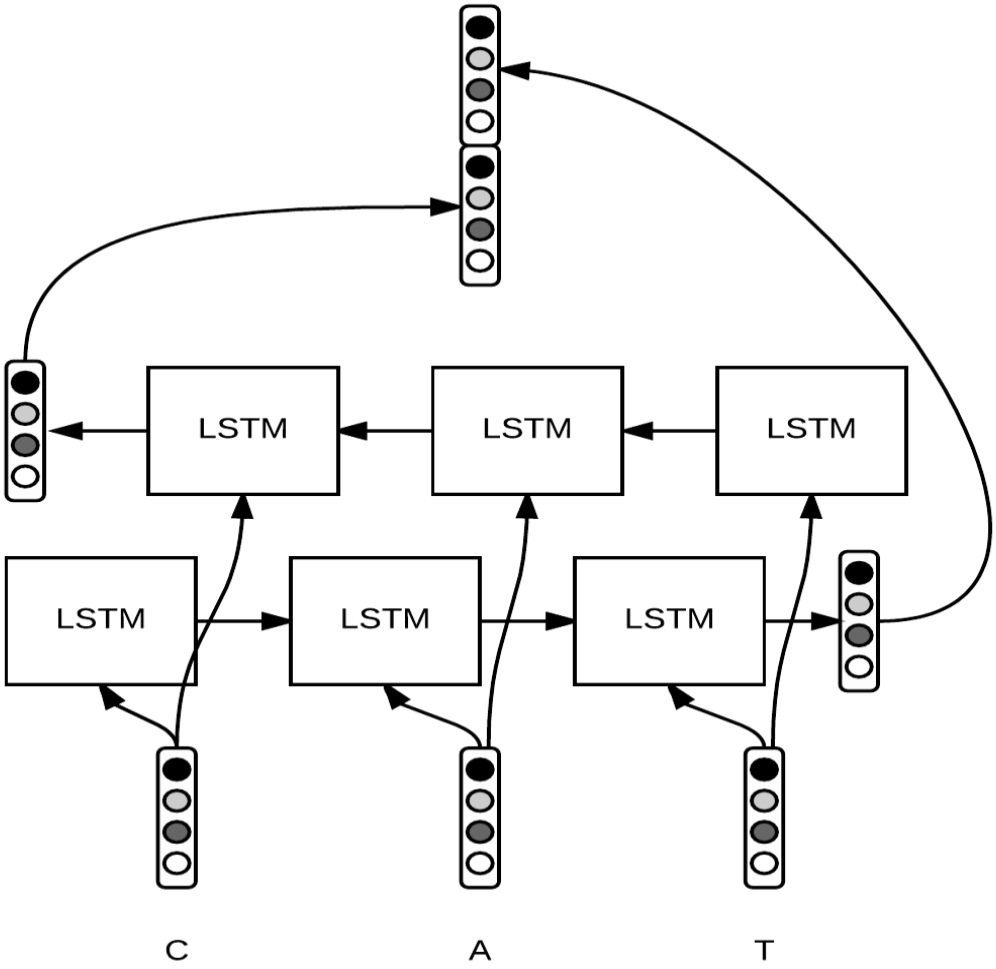

Figure 13. Word level representation from character embeddings. (Copied from Genthial, 2017)

Figure 13 shows how the encoder creates the word presentation vector for the word “cat”. The character embedding for each character is fed into the bidirectional LSTM encoder with two directions, forward and backward. The output of the last timestep (character) of each direction is then concatenated into a vector that carries the encoded information for the word “cat”. An LSTM encoder is capable of capturing sequential patterns from the word’s characters.

With enough training data, the character encoder can learn to capture the character patterns that entities may or may not contain. For instance, in English, words ending with “ese”, i.e. “Chinese”, “Japanese”, and “Vietnamese”, are more likely to be nationalities. Either a CNN or an RNN can be used for the character encoder. For this ANN structure, both architectures are used since experiments show that their combination yields the

best result. The vectors generated from the CNN and RNN encoder are concatenated to create the final encoded vector. The character encoder can also handle the case of misspelling words, emoticons, or words that do not exist in the word embedding layer.

3.3 Output

3.3.1 Context processing

After the character encoded vectors were created, they are concatenated with the word embeddings to form the word presentation vectors, which include the word's semantic and morphological information. These vectors should contain all the potential information that each word might contain. However, sentences cannot be understood by reading each word separately. They need to be processed as a whole to make connections between each word.

Homonyms are common in many languages. In the English dictionary, there is rarely any word that only has one definition. For instance, the word "crane" can be used to describe either a species of birds or a construction machine. This ambiguity also applies to named entities since many named entities share the same name. For instance, the word "Nokia" can refer to the famous company Nokia or the city Nokia in Finland. Because these words share an identical spelling, they also have the same word representation vectors. The word presentation vectors are still deficient, because they contain only word-related features.

When reading a sentence, in order to identify the accurate meaning of a word, one often uses the context clues derived from the rest of the sentence. For instance, in the sentence "the cranes fly towards the horizon", the rest of the sentence "fly towards the horizon" tell the reader that this "crane" is much more likely to be a bird. Likewise, the model needs another stage to correctly identify the context information from the input text. This can be achieved by using another RNN layer to process the word representation vectors.

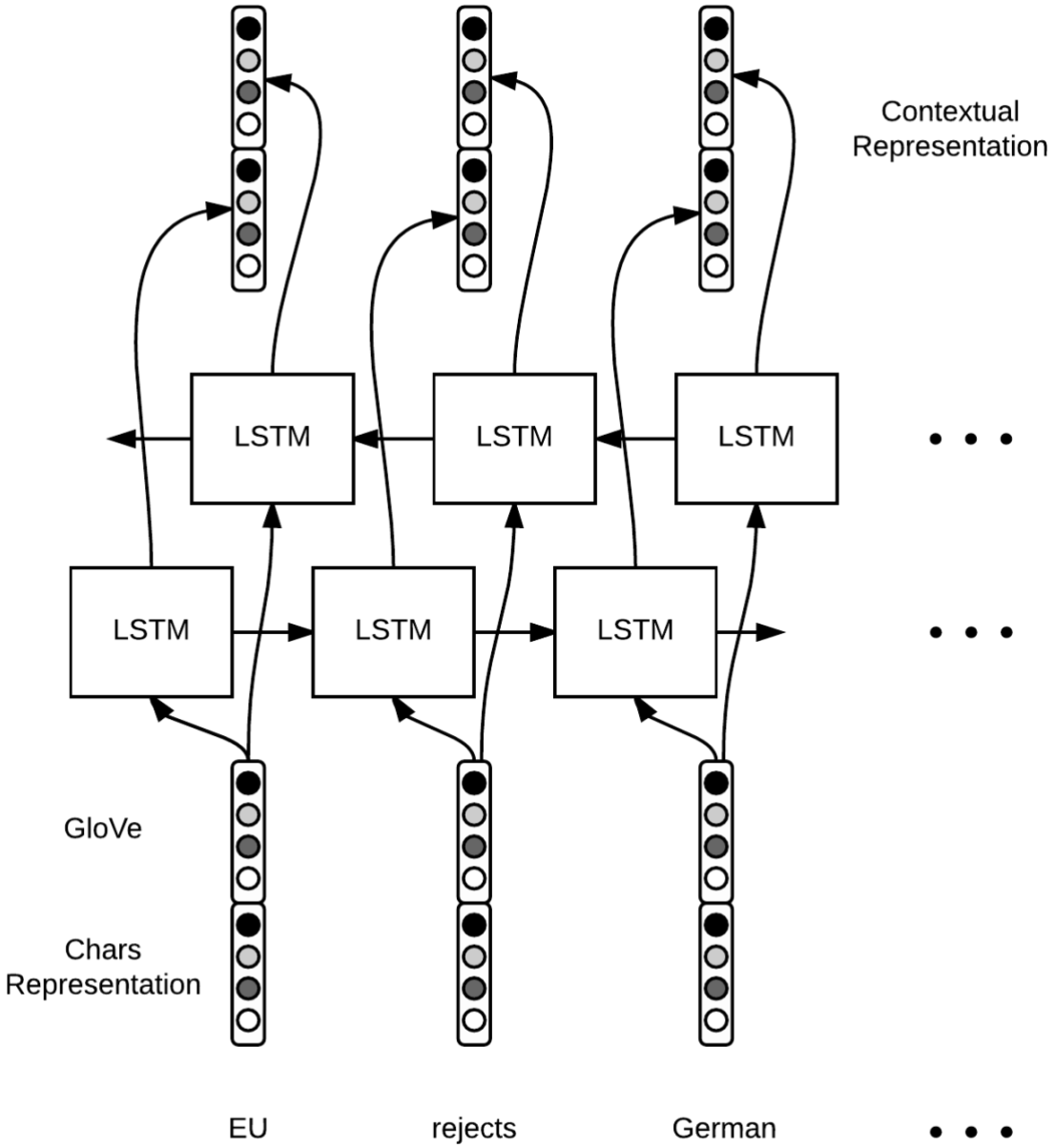

Figure 14. Bidirectional LSTM on top of word representation to extract contextual representation of each word. (Copied from Genthial, 2017).

Figure 14 describes the context vector generating process using bidirectional LSTM. In the figure, GloVe represents the word embedding vector and the character representation is the output of the previous character encoding layer. These two vectors are concatenated to form the final word representation vectors, which are used as input for the token bidirectional LSTM in order to generate the word's contextual features. Unlike the character encoding layer, which only takes the output at the last timestep, the

output of this layer will be taken at every timestep. These output vectors contain the information about the word and the contextual information it carries.

3.3.2 Output layer

After the word features of the whole input sequence are extracted and the context vectors are generated, the model can use the data to compute the tagging output. The contextual vectors are fed into several layers of fully connected networks to compute the potential score for each of the available tags. The scores then go through the linear chain CRF layer, which will output the tagging sequence with the highest score for the input text.

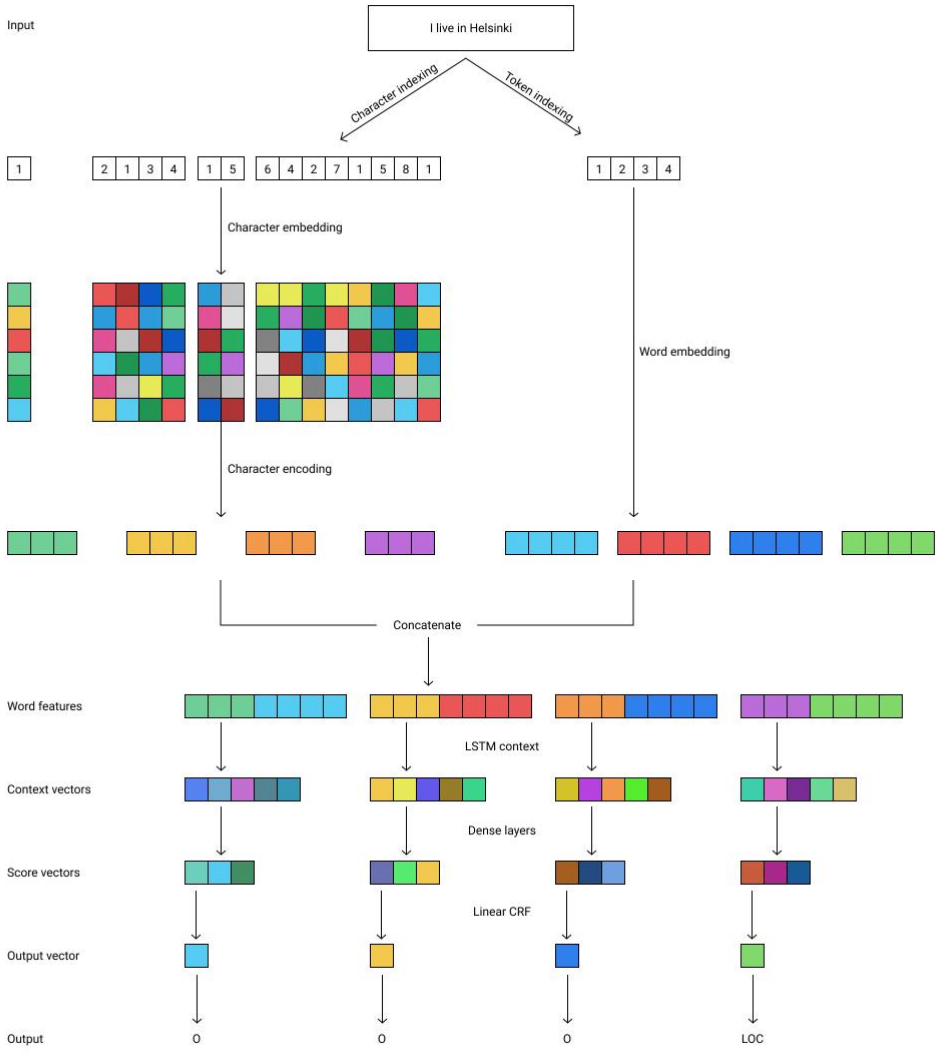

Figure 15. Final ANN model architecture

Figure 15 shows the architecture of the final NER model. Every input sentence is transformed into numerical data which consists of two vectors: the character indices and token indices. The character indices are fed into a character embedding layer, followed by a character encoding layer to construct the word morphologic feature vectors. The token indices are transformed into word embedding vectors, which carry most of the semantic information of the words. The semantic vectors are then combined with the morphologic vectors to form the word feature vectors. These feature vectors are then processed by another LSTM layer to generate the final context vectors. The context vectors are then projected into the score vectors by several dense layers, which will be fed into the output CRF layer to calculate the model's prediction.

4 Implementation details

4.1 Data annotation

4.1.1 BIO tagging

In order to store the output for each input sequence, the BIO tagging format (beginning, inside, outside) is used. This format was introduced in 1995. (Ramshaw and Marcus, 1995). It makes sure that named entities with more than one word can also be recorded without any potential conflicts.

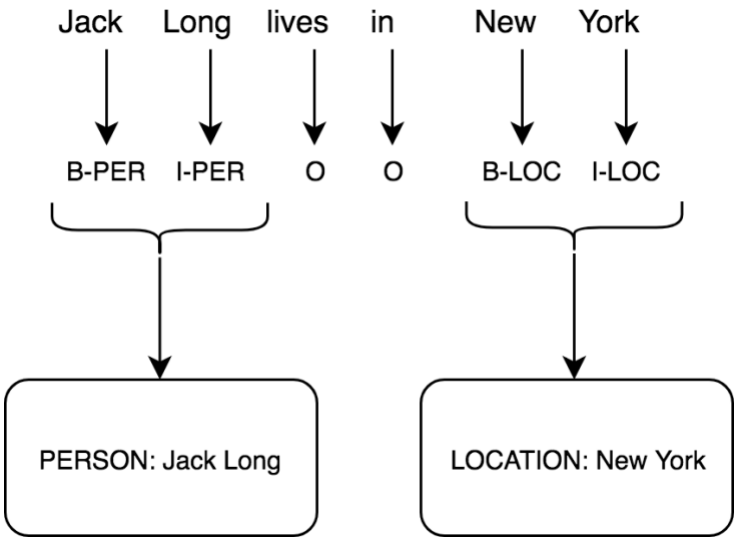

Figure 16. A Bio tag scheme example

Figure 16 shows a case of a BIO tag for a simple sentence. The B (begin) prefix marks the beginning of the entity, and I (inside) indicates that the current word is a part of the preceding entity. O (outside) marks the words that do not belong to any entities. The model is also trained to output BIO tags for every input sequence.

4.1.2 Database architecture

The training data is expected to come in high volume and expand continuously. Moreover, because the model architecture is updated occasionally, the training data structure might also change overtime. Therefore, it is best practice to use a NoSQL database such as MongoDB to store the training set.

MongoDB is a NoSQL database which is document-oriented, simple to use and easy to scale. Instead of storing the data with columns and rows like tradition relational database, MongoDB store data objects as JavaScript Object Notation (JSON) documents, which indicates that each record is stored as a JavaScript object in a MongoDB collection. MongoDB databases also have high performance and high ability (Jayaram, 2016.)

The training set, generated by humans, is stored in a MongoDB collection. Every training sample contains a tokenized article as input and the tag label for each token as output. The labels are annotated manually by the human workers. Also, other information about the article such as language, date, tags and the worker's data are also collected and saved. These data are then used to build a performance profile for each worker, which will then be used as a reference during the automated cross validation between the answers.

4.2 NER service deployment

4.2.1 Training process

The model was implemented using the programming language Python, a high-level programming language that is used to implement most of Vainu's services. It comes with many machine learning and scientific computing libraries such as NumPy, scikit-learn, Tensorflow, Pytorch, etc. The ANN model was constructed and trained with Tensorflow, an open source deep learning framework from Google.

As the name suggests, Tensorflow is built around objects called tensors, a generalization of vectors and matrices to potentially higher dimensions (Google, 2018). A tensor is an algebraic object with n dimensions, with n varying from 0 to any natural number.

Tensorflow was designed following the principle of static computation graphs. Each ANN model is represented as a computation graph. Each graph is constructed using three main components: Placeholder, Variable and Operation. Placeholders are the tensors that will receive their value from outside the graph. They are used for storing the input and output data. The model parameters (weight, bias) are stored as Variables, while the Operation defines the operations between tensors.

To train a model with Tensorflow, the model graph must be constructed in advance. First, Variables and Placeholders are initialized as model tensors with size and data type. Operations between tensors are also defined in this process. The loss function and model optimization step are also defined in this stage as an Operation between the model output and the expected output. Afterwards, during the training phrase, training data are fed to the Placeholders and they are used along with the Variables to compute the model output (forward pass). All of the optimization processes are handled automatically by the Tensorflow framework with automatic differentiation.

The model is trained using the mini-batch gradient descent with the size of 16. During the training process, the training data are shuffled. For each training step, a batch of 16 random samples are sampled from the training set. Mini-batch gradient descent reduces overfitting and speeds up the training process.

The training process continues until all of the training samples are consumed (one epoch). Typically, each training session takes 20 epochs on average, which takes approximately 2 hours using three NVDIA Quadro P3000 GPUs.

Overfitting is an error which can happen during the training process when the model becomes too biased with the training data. Overfitting reduces the model's predictive performance when making prediction on real, unseen data. To prevent overfitting, 10% of the training data is randomly sampled to form the validation set when the training starts. This validation set will be used to evaluate the performance regularly during the training process to hypothesize the model's performance on unseen data. When the validation performance begins to drop, the model is very likely to start being overfit. The training process is stopped when this happens.

4.2.2 Details of application implementation

The NER model is implemented as a Python class. Each model instance is an object inherited from this class. The model structure is controlled using different constants called hyper parameters. Models with a different set of hyper parameters usually have different performances and training speed.

```
hyper_parameters = {
    "deploy": False,
    "force_deploy": False,
    "character_embedding_dimension": 50,
    "token_embedding_dimension": 300,
    "token_lstm_hidden_state_dimension": 400,
    "feedforward_after_lstm_layers": "300",
    "learning_rate": 0.0015,
    "filter_sizes": "2,3,4,5,6",
    "filter_number": 60,
    "dropout": 0.32,
    "cnn_dropout": 0,
    "character_lstm_hidden_state_dimension": 200,
    "eval_per_epoch": 3,
    "epoch_number": 30,
    "language": "fi",
    "gpu_device": 1,
    "batch_size": 8,
    "elmo": False,
    "layer_norm": True,
    "use_old_data": True,
    "auto_validate": True,
}
```

Figure 17. Sample hyper parameters

Figure 17 displays some sample hyper parameters that are used to initialize an instance of the NER model. The model properties such as number of layers, number of nodes on each layer, and learning rate can be controlled using these hyper parameters. This approach simplifies the process of building and testing different model variants.

With many variables to control the model architecture and the training process, it is necessary to try different sets of hyper parameters for every machine learning problem to find the best model for the task. This hyper parameter tuning process is handled automatically by a third-party library when the model is being trained.

After the training process finishes, the model hyper parameters and parameters are saved and uploaded to an Amazon S3 Bucket. The NER application is implemented as an HTTP application. When the application starts, it will scan the S3 Bucket to download the hyper parameters and parameters of the model's latest version. The hyper parameters are used to initialize the model object and the parameters are used to restore the model to its optimized state.

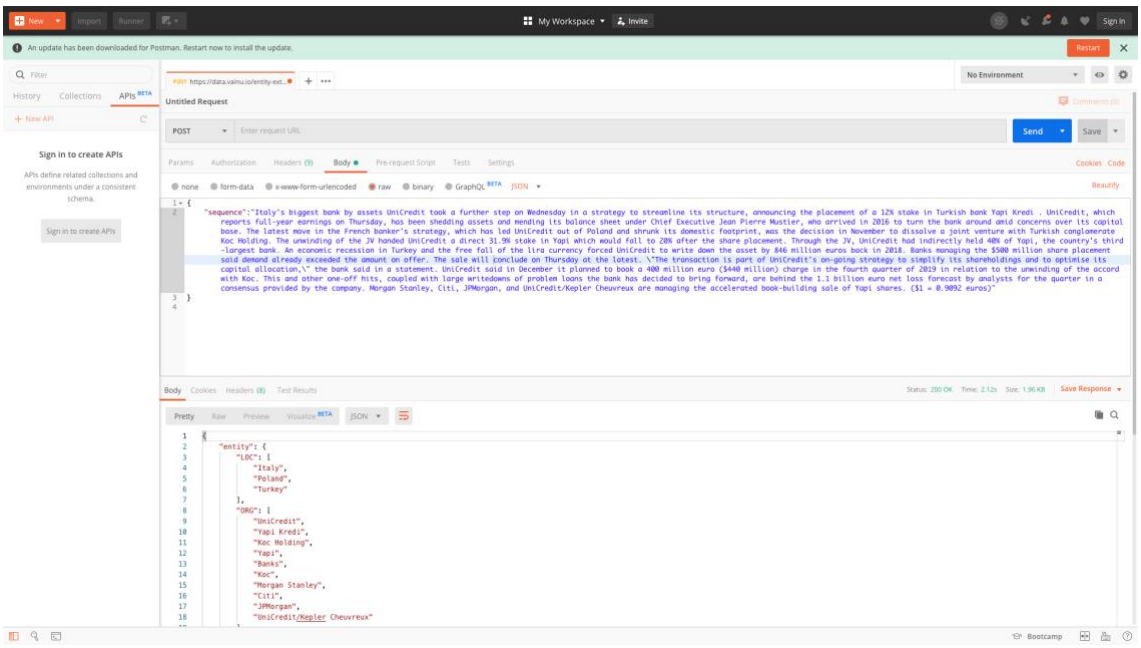

Figure 18. Microservice input/output

Figure 18 shows an example of the microservice input and output. The application receives the text input through the HTTP requests. It converts the text into numerical format, runs the input through the model to take the prediction and returns the model prediction to the caller. For the backend implementation, Flask, a micro-framework for web server, is used.

4.2.3 Microservice

Microservice is a new approach to software development that has gained a considerable amount of popularity in modern software development. An application that follows the microservice pattern is constructed as a collection of small services built separately, each service performing a specific function. These micro components allow multiple teams to work simultaneously, each one focusing on a single feature or on a branch of the application while being independent of others. Additionally, since the software is divided into multiple small modules, the reusability of every module is increased, along with the simplicity of maintenance, upgrading and deploying of each component.

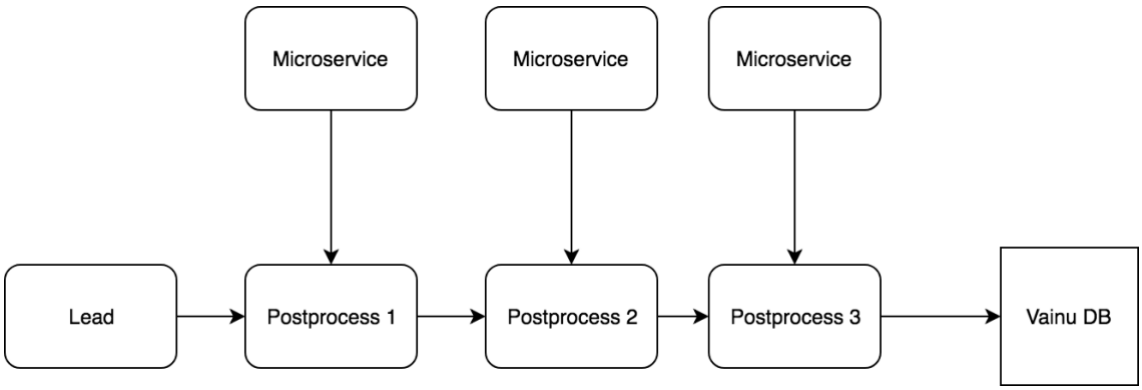

Figure 19. Lead processing with microservices

Figure 19 describes how microservices are used in the automated lead processing system of Vainu. Each microservice has one task, such as classifying and matching. The NER model is deployed as a part of the prospect tagging process by detecting and extracting the potential company names from the lead content. The extracted named entities are then used to match with the company names in the Vainu database to tag the mentioned companies to the lead object.

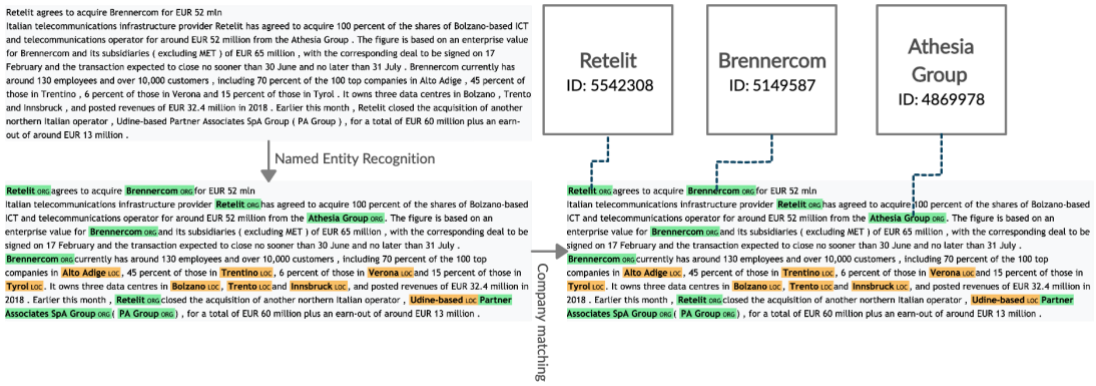

Figure 20. Company matching with NER

Figure 20 displays how a sample lead is processed in Vainu's company matching system. The NER application detects all potential named entities from the lead's content and title. The company names are then extracted and used along with other data such as location and context. In order to search for the best matches from Vainu's company database. When a match is found, the lead is added to the company profile.

4.2.4 Docker and Kubernetes

With the growth of its software infrastructure, Vainu adopted Kubernetes and Docker to minimize the maintenance workload. Docker is an open source platform to design, implement and deploy applications. Unlike normal applications, where the software only contains the source code, Docker allows developers to ship the application with all the dependencies it needs as a built package. This eliminates the compatibility issues that often happen when the deployment machine does not run the same environment system as the development machine. Applications packaged using Docker can run on any system with Docker without any potential conflicts. A Docker image is a unit of software that contains the code and all of its dependencies. Docker images become containers at runtime, when they run on the Docker Engine.

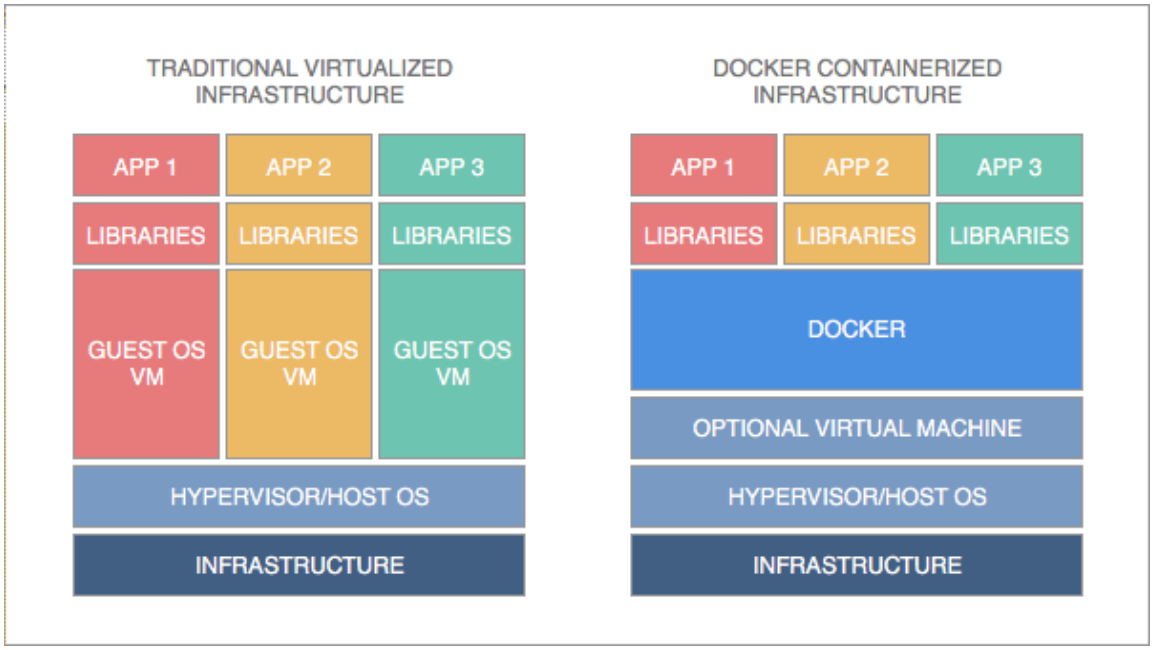

Figure 21. Docker infrastructure vs traditional virtualization. (Copied from Hackett 2015).

Figure 21 explains the main differences between Docker and traditional virtualization (virtual machine). On the other hand, applications packaged with Docker can run with the Linux kernel of the host machine. This reduces the size of the applications, making them much faster to run.

In 2014, Google open source Kubernetes, a platform for managing workloads and services. By design, Kubernetes only works with Docker containers as the application unit. Kubernetes also comes with different tools and services to simplify the application

management process such as application autoscaling and load distributing between application instances. Other utilities such as log examination, application secret management, and job management are also integrated into this open source platform. (Gerrard, 2019.)

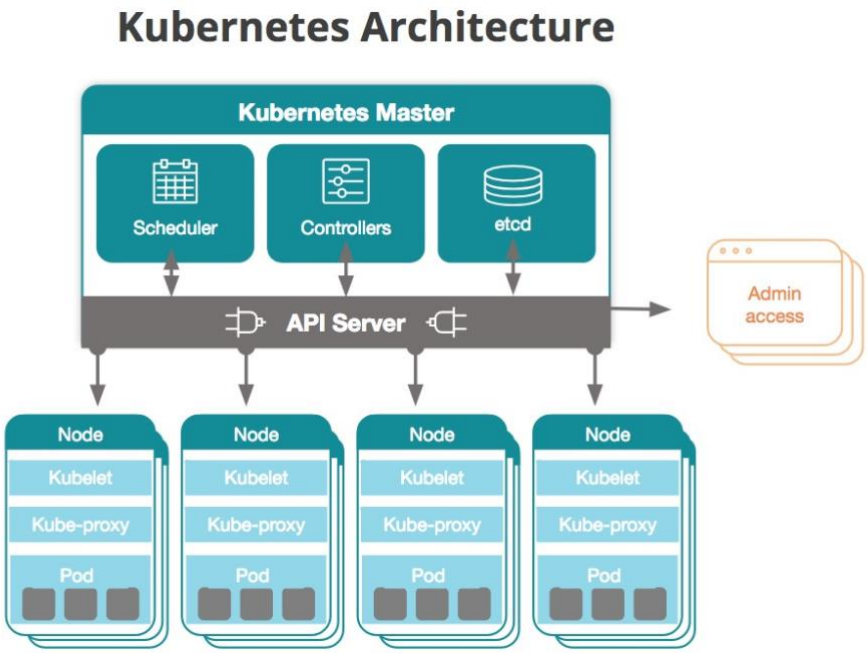

Figure 22. Kubernetes cluster architecture. (Copied from Gerrard 2018).

Figure 22 describes the basic architecture of a Kubernetes cluster. A Kubernetes cluster contains multiple nodes, which are usually hosted on Linux virtual machine instances. Each node contains multiple pods. Each pod is an application instance which is hosted as a Docker container. The nodes hosted by different computers are connected and controlled by the Kubernetes Master. The infrastructure managed by Kubernetes helps developers administrate their applications much more efficiently and with less effort.

4.3 Training pipelines

At the start, the ANN model is trained with approximately 1,000 training samples for every language, which is sufficient for the model to learn and detect named entities with an adequate performance. To improve the model's performance further and to make sure the model is trained and improved continuously, a training feedback loop is implemented.

Leads are randomly sampled from the Vainu database and published to the human workers to manually label.

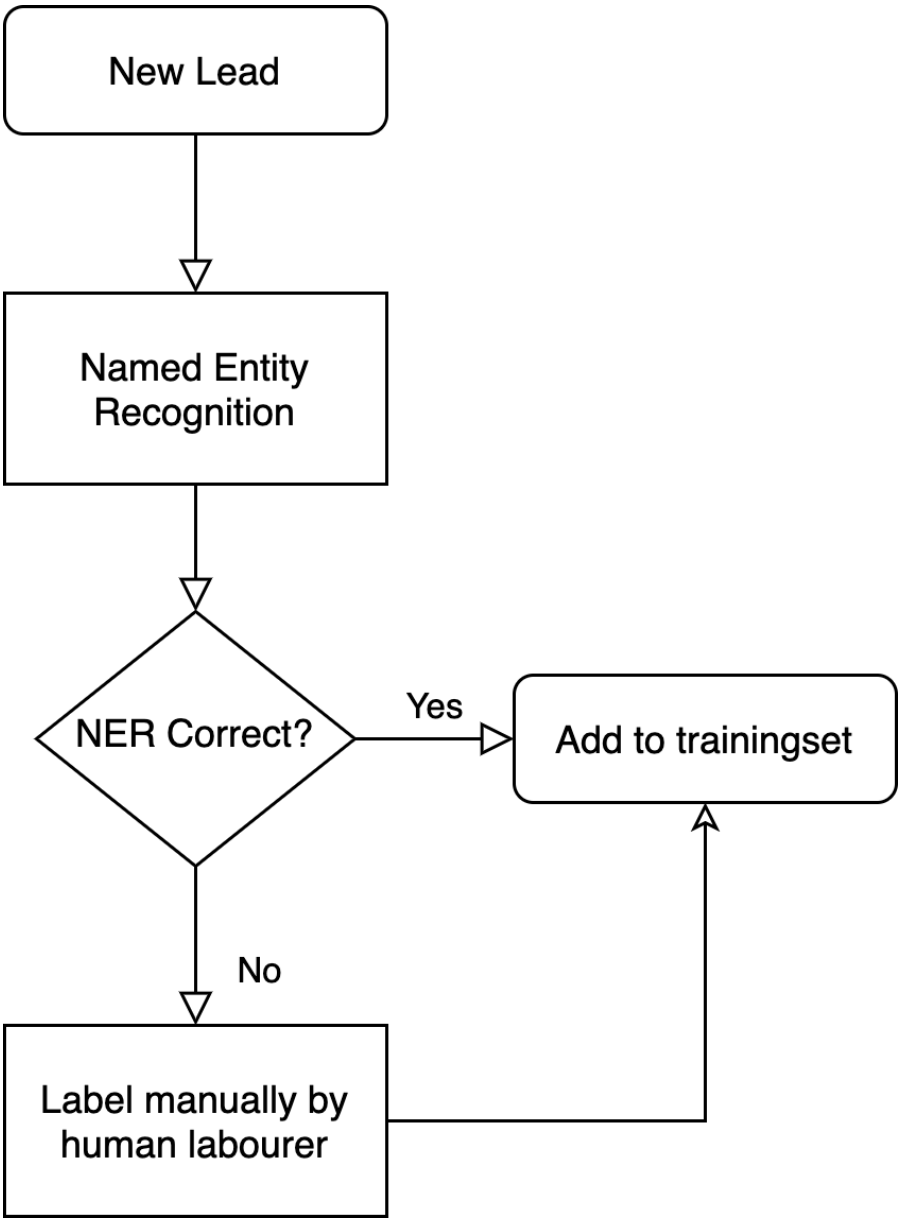

Figure 23. NER training pipeline

Figure 23 describes the automated pipeline for expanding the NER training set. When the NER model processes a new lead, the model output is stored in that lead object. The output is then validated by human workers occasionally. If the output is correct, the lead and the output will be added to the training set as a new training sample. If the annotation is inaccurate, the lead will be sent to get annotated by human laborers to get the correct output. More training data increases the NER performance eventually until the point where little human interaction is needed.

5 Results

After the training pipeline was implemented, a NER model was trained for each language that Vainu supports. The English NER model was benchmarked with the CoNLL-2003 dataset (see Sang and Meulder 2003). In the end, it was able to reach a good overall score.

System	Publication	Results
FIJZ	Florian, Ittycheriah, Jing and Zhang (2003)	88.76%
Baseline	Tjong Kim Sang and De Meulder (2003)	59.61%
BI-LSTM-CRF	Huang et al. (2015)	90.10%
Vainu NER		90.32%

Table 1. Vainu NER test f1 score

English is not a complicated language in terms of syntax and grammar rules, and the English NER task is the most straightforward task out of the seven languages that Vainu supports. Therefore, the English NER performance is the highest one compared to all the NER models that were implemented for the study this thesis discusses. Some other agglutinative languages can be much more challenging. Finnish, for instance, uses different kinds of suffixes depending on the context to express the grammatical information. In this case, word embeddings are not as effective, because the limited number of vocabularies cannot capture all the possible semantic information of words with different suffixes. This is the drawback that static word embeddings carry.

6 Conclusion

The goal of this thesis was to build a state-of-the-art deep learning NER system for lead processing. In addition, an automated learning pipeline was implemented, giving the system the capability to learn and improve continuously with little human interaction needed.

With architectures such as CNN, RNN, and LSTM deep learning was obviously the most effective machine learning algorithm, not only with Computer Vision, Audio Synthesizing but also with Natural Language Processing.

Before the Named Entity Recognition system was put into production, Vainu had to tackle the task by using 3rd party software (Google Cloud API). While this approach did work at some level, it was not cost-effective with the gigantic number of leads that needed to be processed. Google NER also did not support all the languages that Vainu requires and it was not specialized for business articles.

Because of Tensorflow's static graph building syntax, practical implementation is a challenging task, even if the user is familiar with the theory of neural networks and deep learning. Fortunately, Tensorflow's low level syntax requires a solid knowledge background to get started and understand, but when one can grasp its principle, the rest of the framework should become very intuitive.

At the time of completing this thesis, multiple breakthroughs in NLP and deep learning had been made. One of the honorable mentions includes BERT, a new method for word embedding (Devlin et al. 2018). These approaches are proven to work and can boost the score of any NLP system by a considerate margin. Currently, work is being done to integrate this new system into the Vainu NER model.

References

Azevedo F. A., Carvalho L. R., Grinberg L. T., Farfel J. M., Ferretti R. E., Leite R. E., Jacob Filho W., Lent R., Herculano-Houzel S. (2009). Equal numbers of neuronal and nonneuronal cells make the human brain an isometrically scaled-up primate brain. Journal of Comparative Neurology 513(5), 532–541.

Bengio Y., Simard P. and Frasconi P. 1994. Learning Long-Term Dependencies with Gradient Descent is Difficult. URL: https://pdfs.semanticscholar.org/bf49/4f7c293aa217a97a3548169d1057813a967b.pdf Accessed on March 10, 2019.

Chokmani K. 2007. Estimation of River Ice Thickness Using Artificial Neural Networks [online]. June 19 - 22, 2007. URL: https://www.researchgate.net/publication/255629329_Estimation_of_River_Ice_Thickness_Using_Artificial_Neural_Networks. Accessed on March 11, 2019

Colah C. 2015. Understanding LSTM Networks [Online]. August 27, 2015. URL: https://colah.github.io/posts/2015-08-Understanding-LSTMs/. Accessed on November 17, 2018.

Devlin J., Chang M., Lee K. and Toutanova K. 2018. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. arXiv:1810.04805v2 [cs.CL]. May 24, 2019. URL: https://arxiv.org/pdf/1810.04805.pdf. Accessed on April 20, 2020

Dumoulin V. and Visin F. 2018. A guide to convolution arithmetic for deep learning [online]. arXiv:1603.07285v2 [stat.ML]. January 11, 2018. URL: https://arxiv.org/pdf/1603.07285.pdf. Accessed on March 10, 2019.

Genthial G. 2017. Sequence Tagging with Tensorflow [online]. April 5, 2017 https://guillaumegenthial.github.io/sequence-tagging-with-tensorflow.html. Accessed on December 2, 2018.

Gerrard A. 2018. What Is Kubernetes? An Introduction to the Wildly Popular Container Orchestration Platform [online]. July 25, 2018. URL: https://blog.newrelic.com/engineering/what-is-kubernetes/. Accessed on December 7, 2018.

Gillam P. 2015. Nerve Cells and Synapses: A* understanding for iGCSE Biology [online]. URL: https://pmgbiology.com/2015/02/18/nerve-cells-and-synapses-a-understanding-for-igcse-biology/. Accessed on November 10, 2019.

Goodfellow, I. J., Bengio, Y. and Courville A. 2016. Deep Learning [online]. MIT Press. URL: https://www.deeplearningbook.org/. Accessed on October 29, 2019.

Hackett W. 2015. Docker—a brief introduction [online]. July 15, 2015. URL: http://willhackett.blog/docker-a-brief-introduction. Accessed on April 29, 2019.

Heuer H. 2016. Text comparison using word vector representations and dimensionality reduction [online]. Proceedings of the 8th European Conference on Python in Science. July 2, 2016. URL: https://arxiv.org/pdf/1607.00534. Accessed on March 10, 2019.

Hochreiter S. and Schmidhuber J. 1997. Long Short-Term Memory [online]. Neural Computation 9 (8): 1735 - 1780. URL: http://www.bioinf.jku.at/publications/older/2604.pdf. Accessed on August 1, 2019.

Jayaram P. 2016. When to Use (and Not to Use) MongoDB [online]. November 30, 2016. URL: https://dzone.com/articles/why-mongodb. Accessed on February 27, 2019.

Hridoy J.M. 2013. A Study on The Approaches of Developing a Named Entity Recognition Tool. International Journal of Research in Engineering and Technology (IJRET) 2(2): 58-61. December 2013. URL: https://ijret.org/volumes/2013v02/i14/IJRET20130214011.pdf. Accessed on March 10, 2019.

Joulin, A., Grave, E., Bojanowski, P. and Mikolov, T. 2016. Bag of Tricks for Efficient Text Classification [online]. arXiv:1607.01759 [cs.CL]. August 9, 2016. URL: https://arxiv.org/pdf/1607.01759.pdf. Accessed on January 10, 2020.

Linnainmaa, S. 1970. The representation of the cumulative rounding error of an algorithm as a Taylor expansion of the local rounding errors. Master's Thesis (in Finnish), University of Helsinki, 6–7.

Moawad, A. Neural networks and back-propagation explained in a simple way [online]. URL: https://medium.com/datathings/neural-networks-and-backpropagation-explained-in-a-simple-way-f540a3611f5e. Accessed on December 1, 2018.

Nielsen, M.A. 2015. Chapter 2: How the Backpropagation Algorithm Works [online]. Neural Networks and Deep Learning. Determination Press. URL: http://neuralnetworksanddeeplearning.com/chap2.html. Accessed on March 1, 2020.

Pitis S. 2016. Written Memories: Understanding, Deriving and Extending the LSTM [online]. July 26, 2016. URL: https://r2rt.com/written-memories-understanding-deriving-and-extending-the-lstm.html Accessed on April 20, 2020

Powell, V. Image Kernels Explained Visually [online]. January 29, 2015. URL: http://setosa.io/ev/image-kernels/. Accessed on March 10, 2019.

Ramshaw L.A and Marcus M.P. 1995. Text Chunking using Transformation-Based Learning [online]. arXiv:cmp-lg/9505040. URL: https://arxiv.org/pdf/cmp-lg/9505040.pdf. Accessed on January 25, 2020.

Rashid, T. 2016. Make your own neural network [electronic book], 29 March 2016. URL: https://github.com/ProWhalen/AndrewNg-ML/blob/master/Make%20Your%20Own%20Neural%20Network.pdf. Accessed on December 1, 2019.

Sang E.F.T.K and Meulder F.D. 2003. Introduction to the CoNLL-2003 Shared Task [online]. URL: https://www.aclweb.org/anthology/W03-0419.pdf. Accessed on January 15, 2020.

Schnabel, T., Labutov, I., Mimno, D. and Joachims T. 2015. Evaluation methods for unsupervised word embeddings. Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP), September 17-21, 2015; pp. 298-307. URL: https://www.aclweb.org/anthology/D15-1036. Accessed on July 17, 2019.

Wang, H. and Raj, B. 2017. On the Origin of Deep Learning [online]. arXiv:1702.07800v4 [cs.LG]. March 3, 2017. URL: https://arxiv.org/pdf/1702.07800.pdf. Accessed on October 5, 2019.