École Normale Supérieure de Rennes
Département de Mathématiques

Deep Learning for Natural Language Processing Internship

Noé Malais
noe.malais@ens-rennes.fr

Under the supervision of
Thierry Poibeau, KyungTae Lim, Serge Sharoff

Lattice Laboratory
(CNRS & ENS / PSL & Université Sorbonne nouvelle / USPC)
1 Rue Maurice Arnoux 92120 Montrouge

Paris
May - July 2019

Contents

Introduction

In this report, I present the subjects I had the opportunity to work on in the Lattice Laboratory during my internship. The report is divided into five sections. The first two sections go over the machine learning concepts that I had to understand before being able to work with the Lattice researchers. The third section presents some linguistics and Natural Language Processing concepts that are necessary to get what the machine learning algorithms were actually used for. The last two sections are focused on what I produced during the Internship. Section 4 is a coding project I worked on at the beginning of the internship to get used to the tools used in this field, while in section 5 I present the work I did together with the researchers, which included some contributions (and implementation thereof) in improving an existing deep-learning model for cross-lingual dependency parsing. I would like to heartily thank Thierry Poibeau who accepted me as an intern, for his kind attention and supervision of my internship, as well as KyungTae Lim and Serge Sharoff for their patient and careful explanations, and for the trust they gave me to work with them. I would also like to thank the whole Lattice team for their warm welcome and for having introduced me to linguistic curiosities and fields of research that were new to me.

The Lattice laboratory

The Lattice laboratory hosts researchers and Ph.D students from various fields ranging from linguistics and cognitive sciences to computer and data science. Its funding comes from several public instutions including CNRS and ENS/PSL and it is located in Montrouge just south of Paris. The main research themes at Lattice are the study of syntactic and semantic analysis to understand the dynamical constructions underlying speech that give rise to meaning, language evolution (e.g the study medieval french), and automated analysis of large corpora. Teams working on the latter develop and implement deep learning models that extract information from raw or annotated textual data to then perform various tasks, many of which were until recently thought to be unsolvable by machines due to the incredible complexity of language. However new deep learning structures have shown to achieve surprisingly decent results, thereby making Natural Language Processing (NLP) a very active and flourishing research field in recent years.

Internship chronological overview

The internship took place from May 15^{th} to July 5^{th}. The first three weeks I mainly learned how to use the PyTorch python library, followed the Stanford C224 class on Deep Learning for Natural Language Processing and read a number of blog posts and research articles about the latest achievements in the field. Meanwhile I had discussions with my three supervisors (two researchers from the Lattice, and one who was invited for a month, and came from the University of Leeds), who explained to me what they were working on and how they could include me in it. For the remaining four weeks, i mainly worked on adapting an existing dependency parser to incorporate the work of the researcher from Leeds in it and implementing an idea of "confidence score", to set up some experiments. During the internship I attended several conferences at the ENS and in the Lattice premises, about deep learning but also some about linguistics, some given by rare or near-extinct languages native speakers from african and asian countries.

1 Machine Learning Building Blocks

Deep Learning is a subfield of machine learning, therefore it heavily relies on basic machine learning concepts, which will be described in this section, with a strong focus on neural networks. Neural networks are algorithms that loosely mimic some inner workings of the human brain and how neurons process inputs and fire electrical signals to communicate information.

1.1 Artificial neurons

An artificial neuron is a function $\mathbb{R}^n \to \mathbb{R}$ where $n \in \mathbb{N}$. What the neuron does is first apply a weighted sum to its inputs $x_1, \ldots, x_n$ according to weights $w_1, \ldots, w_n \in \mathbb{R}$, and then apply an activation function $\varphi : \mathbb{R} \to \mathbb{R}$. Hence the output y of the neuron can be expressed as follows

$$y = \varphi\left(\sum_{i=1}^{n} w_i x_i\right)$$

In practice, the inputs of the neuron are either the inputs of the whole network, or outputs of precedents neurons. They represent action potentials (electrical signals) flowing through synapses (connections between biological neurons). The weights represent the strengh of the connections. They are typically randomly

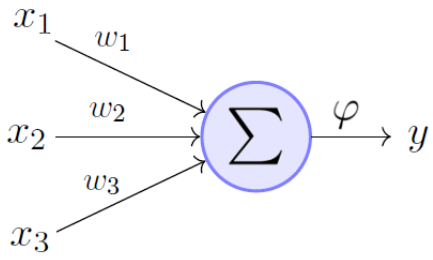


Figure 1: a schematic artificial neuron

initialized and then learned according to a process we present in the next sections. The activation function φ can in theory be any function, depending on the task we want to achieve. Some activation functions are more appropriate than others, and we go over the most used ones in section 1.4.

1.2 Neural Networks

Neural networks are structures made of artificial neurons, where outputs of previous neurons are fed as inputs to other neurons. The structure is divided into several layers, the input layer, one or more hidden layers, and the output layer. So a neural network is a function $\mathbb{R}^n \to \mathbb{R}^p$ where $n, p \in \mathbb{N}$. It is defined as follows. The inputs $x_1, \ldots, x_n$ are given to the neurons $N_1, \ldots, N_k$ of the first hidden layer. Then the outputs of the first hidden layer $h_1, \ldots h_k$ are fed to each neuron in the second hidden layer and so on. The last output is called the output layer. Each time we feed the output of a layer into a neuron of the next layer, we do it according to weights $w_1^{(i)}, \ldots, w_p^{(i)}$ and an activation function φ where i is the index of the neuron in the layer (we suppose the activation function is the same for every neuron in a layer). This process is called forward propagation. Since the process of taking weighted sums for each neuron is a linear operation, we can describe it using matrix multiplication :

$$\begin{aligned} h_1 &= \varphi^{(1)}\left(W_1 x + b_1\right) \\ h_i &= \varphi^{(i)}\left(W_i h_{i-1} + b_i\right), 1 < i < q \\ \hat{y} &= \varphi^{(q)}\left(W_{q-1} h_{q-1} + b_q\right) \end{aligned}$$

where q is the number of layers in the network, and $x = (x_1, \ldots, x_p)^\top$ is a vector of inputs. The weight matrices are of shape (n, p) where n is the number of outputs of the layer and p is the number of inputs of the layer. To the output of each neuron we also add a bias b_i.

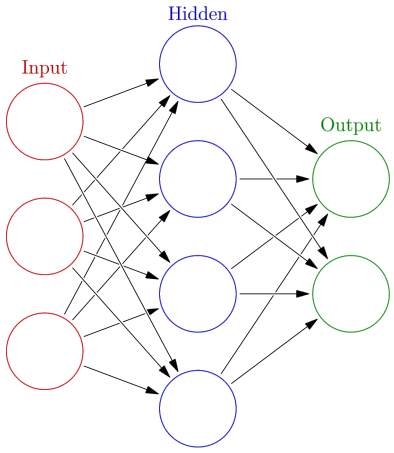


Figure 2: a schematic neural network

1.3 Loss Function and Backpropagation

We want a neural network to be able to achieve a specific task, for example given characteristics about a flower (height, width of the petals, etc...) decide if it is more of a daisy or a daffodil. Such a task is called classification. To get the network to perform well on this task, we use a process called backpropagation to train it (i.e. to find the right weights). First we define a function called the loss function that measures how different (how "wrong") the output of the network $\hat{y}$ is compared to the right answer y. For example in the case of flower classification, the network would have two outputs (representing probability of being a daisy and probability of being a daffodil) and the loss function could be computed as follows :

$$L(y, \hat{y}) = (y_0 - \hat{y}_0)^2 + (y_1 - \hat{y}_1)^2$$

In the case of probabilities, y_0 and y_1 are 0 or 1, and we see that the further $\hat{y}$ is from y, the greater the value of the loss function. The goal is then to minimize (or to maximize in some cases) the value of the loss function by adjusting the weights. With this information in hand, we then use gradient descent to train the network. The partial differential of the loss function with respect to one of the weight matrix gives us the direction of steepest ascent, and taking a small step in the opposite direction gives a weight matrix that tends to minimize the loss function. More formally, we can write the loss function as depending on updatable parameters

$$L(y, \hat{y}) = L(y(\theta), \hat{y})$$

where θ is a vector containing all the updatable parameters (weights and biases) of the network. Performing gradient descent consists of updating θ :

$$\theta_{new} = \theta - \lambda \nabla L(y(\theta), \hat{y})$$

Where $\lambda \in \mathbb{R}$ is called the learning rate. The network output y described above represents all the predictions of the network across a whole dataset. In practice when dealing with large datasets, we don't compute the gradient in this way, rather we approximate it by differentiating the loss function over one example at a time, a process called stochastic gradient descent. Empirically, the technique that gives the best results is batch gradient descent, where we differentiate the loss function after a 'batch' of training data items (typically 32 of them). Batch gradient descent reduces approximation noise (by averaging the gradient over a few example) while offering a quicker, more reliable and more precise way of training a network. There exists even better ways of performing gradient descent such as momentum and learning rate decay that we don't describe here.

1.4 Activation Functions

As we have seen, we need to be able to differentiate a neural network to perform backpropagation. To achieve this, every function used must be differentiable. It is the case for the weight matrices multiplications because they only consist of linear operations. The loss function seen above $y, \hat{y} \mapsto \sum (y_i - \hat{y}_i)^2$ also has a derivative at every point. It is also necessary that the artificial neurons have differentiable activation functions. Another characteristic activation functions are bound to have is that they must be non-linear. If they were to be linear functions, then the whole network would only be a composition of many linear functions, which means it would itself be a linear function. Learning the right weights for such a network would then be equivalent to performing linear regression on a function, which limits the complexity of the patterns that a neural network can reproduce. With non-linear activation functions however, neural networks become universal function approximators, which means that for any given continuous function, there exists a neural network that can approximate the function as well as we want. More formally this result can be expressed as follows :

$$\forall f \in C^0(\mathbb{R}^d, \mathbb{R}^n), \forall K \subset \mathbb{R}^d compact, \forall \varepsilon > 0, \exists N : \mathbb{R}^d \mapsto \mathbb{R}^n : \forall x \in K, \|N(x) - f(x)\|_2 < \varepsilon$$

Where N is a neural network. A proof of this result can be found in [1], at least in the case where all neural activation functions are sigmoids, and K is the hypercube of $\mathbb{R}^d$. Interestingly, the neural network N only needs to have a single hidden layer for this result to be true.

2 Deep Learning Structures

Neural networks as described in the previous section, often called 'vanilla' or 'feed-forward' neural networks, perform well on simple tasks, but they are incapable of learning complex features, working with sequences, or remembering data as they train. More advanced models were invented to address these problems, as we describe in this section. These models constitute what is often called 'deep learning' because they resemble simple neural networks, but have more layers of neurons, that are connected in more sophisticated ways.

2.1 Recurrent Neural Networks (RNNs)

One problem of simple neural networks is that they lack a kind of "memory" : Once they are trained, they can only try and perform their task on single data points taken separately. When dealing with text or video for example, the interpretation of a word or an image heavily depends on the context, e.g in the sentence : "We want to book a Hotel room" and "I read a nice book", the word "book" has different meanings, and even different syntactical uses in the sentence (VERB and NOUN, respectively). In this situation, a simple neural network encountering the word "book" would always produce the same output. Recurrent Neural Networks (RNNs) are structures that can deal with sequences of inputs and take into account previous inputs to produce context-dependent outputs.

2.1.1 Simple RNNs

A Recurrent Neural Network takes as input a sequence of arbitrary length $(x_i)_{i \in \{1,\dots,m\}} \in (\mathbb{R}^d)^m$ and outputs a sequence of vectors $(y_i)_{i \in \{1,\dots,m\}} \in \mathbb{R}^n$. It consists of two simple feed-forward neural networks $N : \mathbb{R}^{d+h} \mapsto \mathbb{R}^h$ (where $h \in \mathbb{N}$ is an integer representing the size of what is called the hidden internal state of the network) and $O : \mathbb{R}^{d+h} \mapsto \mathbb{R}^n$ which produces the final output. The way RNNs process inputs goes as follows : its first input is the vector x_1 (first vector in the sequence of inputs) concatenated with a vector $h_0 \in \mathbb{R}^h$ of random numbers. Its simple feed-forward neural network produces an output h_1 called the hidden internal state of the network (which can be seen as the network's memory of previous sequence items). Then, the neural network takes as input the concatenation of x_2 and h_1 to produce the output h_2 and so on. At each step i, the internal state h_{i-1} can be fed together with x_i to the output network O to get

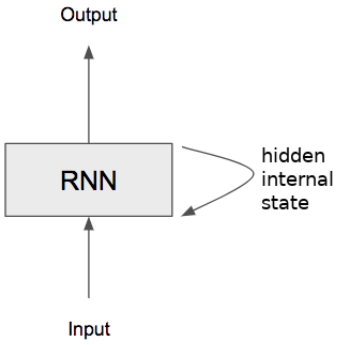


Figure 3: a recurrent neural network

an output y_i, that can be seen as the output of the network for the sequence $\{x_1, \ldots, x_i\}$. The final output of the network is y_m.

Schematically, we represent in Figure 3 a recurrent neural network with a loop, showing that the output at each step is fed back to the same feed-forward neural network (the weights are the same at each step). To better see the flow of the internal states, we can also represent the network in an "unfolded" fashion as shown in Figure 4

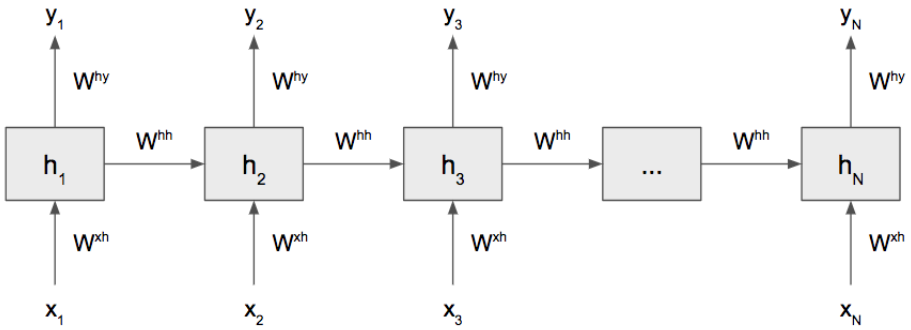


Figure 4: a RNN (unfolded representation)

We can write the equations governing each time step of a RNN as follows :

$$h_i = \varphi(W^{xh} x_i + W^{hh} h_{i-1} + b_h)$$
$$y_i = \varphi(W^{hy} + b_y)$$

As we see in Figure 4, it is possible to compute the output y_i at every time step so we can write the entire network as function :

$$f(\theta, x_1, \ldots, x_m) = (y_1, \ldots, y_m)$$

Where θ represents all the updatable parameters of the network (weight matrices and biases).

2.1.2 Backpropagation through time and computational graphs

Just like simple neural networks, RNNs are differentiable structures, and therefore they can be trained using backpropagation. However, since RNNs have multiple inputs and outputs, and since the same weight matrices are used several times to make one prediction, it isn't obvious how one should update the parameters. In fact, to compute a single loss function, we first compute separate losses for each output y_i and then sum all the loss functions together. The final loss function depends on the updatable parameters and can be differentiated. We don't derive here the computations by hand since in practice deep-learning platforms come equipped with automated differentiation algorithms that rely on the concept of computational graphs. Computational graphs are directed acyclic graphs that represent a sequence of computations.

A computational graph is a 6-tuple

$$(n, l, E, u, d, f)$$

where :

- $n \in \mathbb{N}$ is the number of vertices in the graph
- $l \in 1, \dots, n$ is the number of leaves in the graph
- $E \in \mathbb{N}^2$ is the set of edges, i.e. pairs (i, j) where $1 \leq i \leq n-1$ and $l+1 \leq j \leq n$. We say that the graph is topologically ordered.
- $u = (u^1, \dots, u^n)$ is a n-tuple of variables associated with each vertex.
- $d = (d^1, \dots, d^n) \in \mathbb{N}^n$ is a n-tuple of dimensionalities of the variables. That is, $\forall i \in \{1, \dots, n\}, u^i \in \mathbb{R}^{d^i}$
- $f = (f^{l+1}, \dots, f^n)$ is a n-tuple of local functions for each vertex of the graph that is not a leaf.

For a given computational graph, we define $\alpha = (\alpha^{l+1}, \dots, \alpha^n)$ where $\alpha^i = \{u^j | (j, i) \in E\}$ contains all the inputs for vertex i.

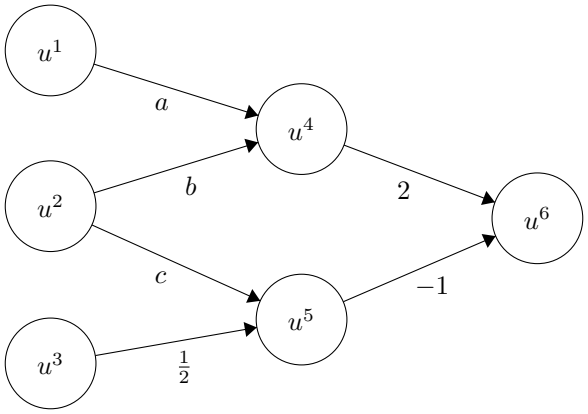
To differentiate this structure, we define a local Jacobian function for each edge (j, i) in the graph :

$$J^{j \to i}(\alpha^i) = \frac{\partial f^i(\alpha^i)}{\partial u^j}$$

We then apply the backward algorithm : given the values $u^1, \dots, u^n$, (computed using the forward algorithm), we set $P^n = I_{d^n}$. Then for $j = n-1, \dots, j = 1$,

$$P^j = \sum_{i > j} P^i J^{j \to j}(\alpha^i)$$

All is left to do is to return $P^1, \dots, P^l$ which are the partial derivatives of the graph output u^n with respect to leaf values $u^1, \dots, u^l$. Let's apply the algorithm on a simple example



Here, u^1, u^2, u^3 are the leaf values. We have

$$f^4 : x, y \mapsto x + y$$
$$f^5 : x \mapsto x^2 + \frac{y}{2}$$
$$f^6 : x, y \mapsto 2x - y$$

Suppose we have $u^1 = 1, u^2 = 4, u^3 = -1$ First we get by applying the forward algorithm :

$$u^4 = 1 + 4 = 5$$
$$u^5 = (-1)^2 + \frac{4}{2} = 3$$
$$u^6 = 2 \times 5 - 3 = 7$$

We set $P^6 = 1$, and we compute :

$$P^5 = 1 \times (-1) = -1$$
$$P^4 = 1 \times 1 = 1$$
$$P^3 = -1 \times \frac{1}{2} = -\frac{1}{2}$$
$$P^2 = 1 \times 1 + (-1) \times 2 \times 4 = -7$$
$$P^1 = 1 \times 1 = 1$$

In this case, applying the algorithm is just like applying the chain rule to get the partial derivatives needed for backpropagation. Those graphs are heavily used by the Pytorch platform which is the one I mainly worked with during the internship, and they enable stacking several RNNs layers, and developing more advanced structures such as LSTMs, without worrying about implementing differentials and manual parameters updates.

2.1.3 Long Short-Term Memory

A problem one encounters in practice with simple RNNs is that of the vanishing gradient. When dealing with long sequences, the network has trouble remembering old information because the gradient decays exponentially and becomes mixed with that of more recent elements in the sequence. To overcome this problem researchers have come up with better structures such as GRU and LSTMs. LSTM (Long Short-Term Memory) networks are Recurrent Neural Networks that compute their hidden state in a more elaborated way than just applying a feed-forward neural network to the input. They use gating mechanisms to ignore some inputs or strengthen retained information. They are described by the following equations :

$$\begin{aligned}
i_t &= \sigma(W_i x_t + U_i h_{t-1} + b_i) \\
\bar{C}_t &= tanh(W_c x_t + U_c h_{t-1} + b_c) \\
f_t &= \sigma(W_f x_t + U_c h_{t-1} + b_c) \\
C_t &= i_t \times \bar{C}_t + f_t \times C_{t-1} \\
o_t &= \sigma(W_o x_t + U_o h_{t-1} + V_o C_t + b_o) \\
h_t &= o_t \times tanh(C_t)
\end{aligned}$$

The network still has a internal hidden state h_t at each time step, but it now also has a "cell state" C_t. Both are vectors that are propagated through the network, the cell state represent the information retained from the sequence, and it is updated as it flows through the network with the new input.

In the equations above, we have :

$$
\begin{aligned}
i_t &: \textit{represents the input gate} \\
f_t &: \textit{represents the forget gate} \\
o_t &: \textit{represents the output gate} \\
\sigma &: \textit{the sigmoid function} \\
W_\cdot, U_\cdot, V_\cdot &: \textit{weight matrices} \\
b_\cdot &: \textit{biases} \\
x_t &: \textit{input at current time step} \\
C_t &: \textit{cell state} \\
h_t &: \textit{internal hidden state}
\end{aligned}
$$

When the cell state flows through the network, the input control the forget and input gate, which decide how much information the cell state will lose about previous inputs, and how much it will gain information from the new input. The input, the hidden state, and the cell state combined control the output gate, to decide what information to output. This structure is quite complex, but the network is able to learn how to manage the gates when being presented enough data, and better deals with longer sequences than a simple RNN.

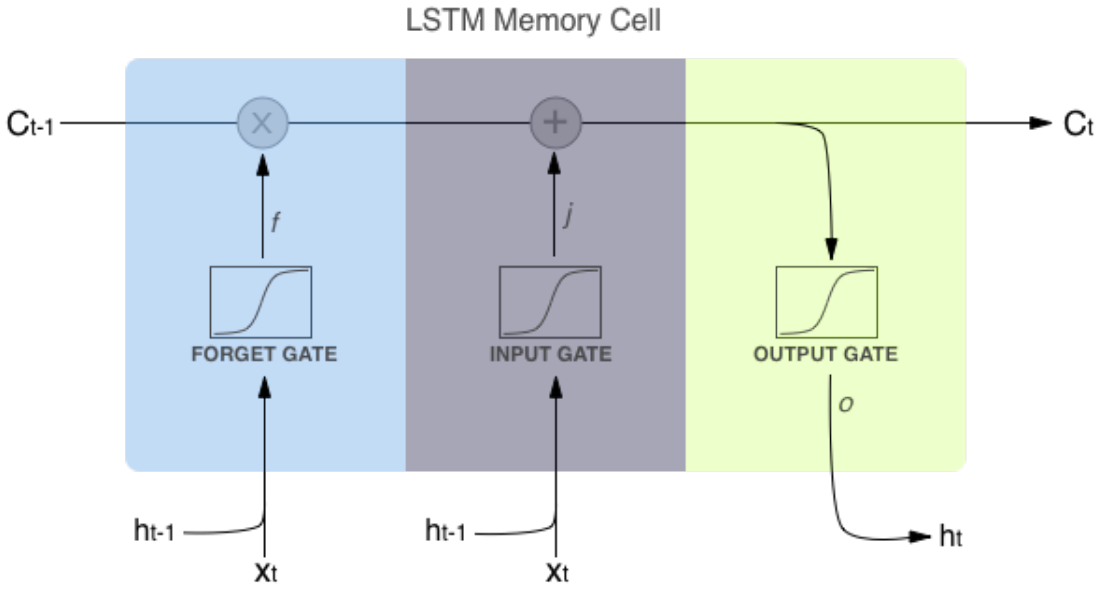


Figure 5: The structure of a LSTM, highlighting the three gates

2.2 Multitask Learning

Multitask learning is a deep-learning technique in which a model is trained to solve several different tasks at once. For example, in the case of image recognition, the tasks could be detecting cars and trucks on

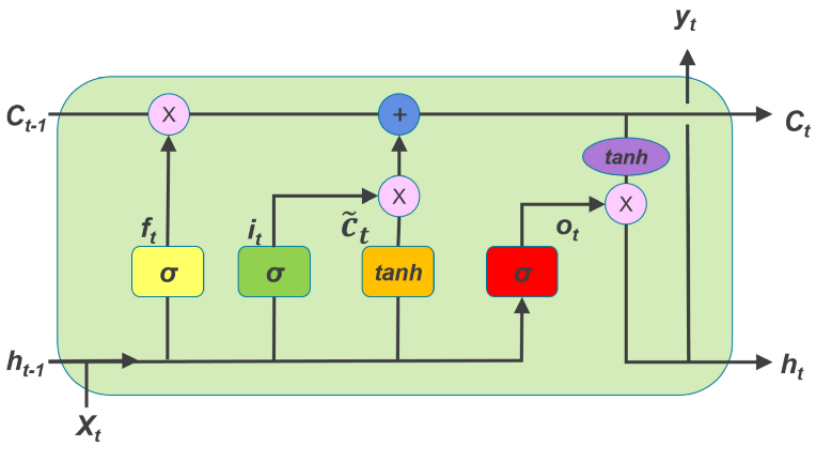


Figure 6: A time step in a LSTM Network

images. Surprisingly, multitask learning has shown to improve training : a model that was trained only to detect cars on images will not perform as good as another one that was trained on both tasks. In the case of Natural Language Processing, we observe the same property, i.e. multilingual models perform better on a given language than monolingual models trained specifically on that language.

3 Natural Language Processing Tasks

NLP (Natural Language Processing) is a subfield of computer science focused on processing human written (or spoken) language and extract meaning from it to perform various tasks. In this section we present some of those tasks, that are studied at the Lattice. Many of them were previously thought to be unsolvable by computers, but new machine learning techniques have achieved remarkable results.

3.1 Part-Of-Speech (POS) Tagging

3.1.1 What is POS-Tagging

During the first two weeks of the internship, I implemented a Part-of-Speech (POS) tagger. A POS tagger is a program that takes as input a sentence, and outputs a list of strings called "POS tags". These tags indicate the syntactical role of the words in the sentence. For example given the sentence "The cat eats.", the algorithm would return '[DET, NOUN, VERB, PUNCT]'. As we can see from this example, the program does not only read words, but also punctuation symbols such as the dot '.' . In general it processes every entity that has grammatical meaning, and we refer to those entities as "tokens".

3.1.2 The Universal Dependencies Project

As we have seen in section 2.2, learning on several languages simultaneously can improve the performance of deep-learning algorithms. To be able to learn on several languages at the same time, it is necessary that learning data comes in an unified format. This is not trivial since different languages might have different grammars, and therefore not use the same Part-of-speech tags. To circumvent this problem, linguists have come up with several systems. The data used during my internship mainly originated from the Universal Dependency Corpora, in which we limit ourselves to 17 main Part-of-Speech tags :

ADJ	adjective
ADP	adposition
ADV	adverb
AUX	auxiliary
CCONJ	coordinating conjonction
DET	determiner
INTJ	interjection
NOUN	noun
NUM	numeral
PART	particle
PRON	pronoun
PROPN	proper noun
PUNCT	punctuation
SCONJ	subordinating conjunction
SYM	symbol
VERB	verb
X	other

The Unversal Dependency Project uses these tags (and many others that express dependency parsing) across over 70 languages and provide annotated corpora that were used extensively during this internship to train deep learning models.

3.1.3 The challenge of POS Tagging

At first sight, part-of-speech tagging can seem to be a very easy task, because one could imagine an algorithm based on a dictionary that stores for each word its tag, and directly annotates the corpus just by assigning the right tag to each word. In fact, there are several problems with this approach because a word can have different POS tags depending on the context in which it is found, and two words that have different meanings can have the same writing. For example in the two sentences :

- Let's go for a run
- I run every day

The same word "run" serves both as a NOUN and as a VERB, and in the sentences

- The dogs are playing
- The sailor dogs the hatch

The words "dogs" and "dogs" have different meanings and different tags but are written in the same way, which makes the task hard for a computer to tell which meaning is being used.

3.2 Dependency Parsing

Dependency parsing is the task of extracting a "dependency graph" from a sentence. A dependency graph is an oriented acyclic connected graph that captures the syntactic relationships between tokens. Starting with an arbitrary root, the vertices are the token in the sentence and the edges and their labels describe the syntactical relationships between them. This is not an easy task as there can be ambiguities as we can see in Figure 8

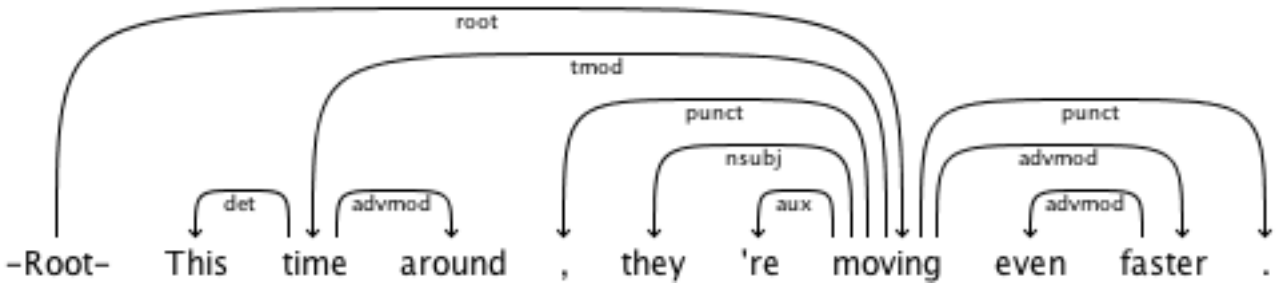


Figure 7: an example of a dependency graph

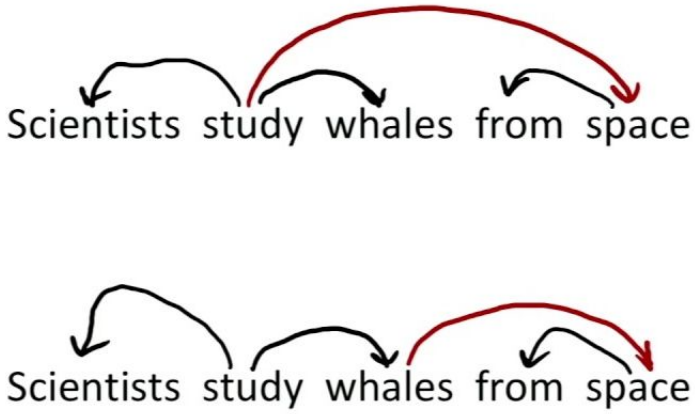


Figure 8: an ambiguous sentence

3.3 Word Embedding

We have seen that neural networks in general take as input vectors that live in $\mathbb{R}^d$, but we were saying just before that we can use deep learning to improve tasks where the input is a sentence, i.e. a list of words. So we have to find a way to express sentences using vectors. This can be done at a sentence level (a vector for each sentence), at a word level (a vector for each word) or at a character level (a vector for each character). Each approach has up and downsides. At a character level, we only need to store the same number of vectors as there are characters, but in practice it is difficult to get networks to capture long term dependencies and the concept of word entities using this approach. On the contrary the sentence level requires almost infinite memory to store vectors for every possible sentence. This is why most of the time we use word level representation, which necessitates to store a few hundred thousand vectors (one for each word in a language). But how to choose what vector to assign to each word, so that the numbers can actually be used ? It appears that to build vectors with meaning, one good approach is to already use machine learning. In the next subsection we present Word2Vec, which is an algorithm specifically designed to make a correspondence between a word and its associate vector, in such a way that vectors capture some kind of "meaning".

3.3.1 Word2Vec

The Word2Vec algorithm is a two-layer neural network that has as many inputs neurons as there are words in the dictionary. There are different model for Word2Vec, and here we present the skip-gram model : the

network is given a word from the dictionary (represented with a one-hot vector, i.e. a vector with only 0s and one 1 at the place where the word is in the dictionary), and it tries to predict the words that surrounds it. So its output is a vector that contains probabilities of appearance of other words in the context of the input word. This means that the model learns to predict a word's company. Once the model is trained, what we keep is not the outputs of the network, but the first weight matrix, because this matrix gives us one vector for each word in the dictionary. These vectors can then be used to perform other tasks.

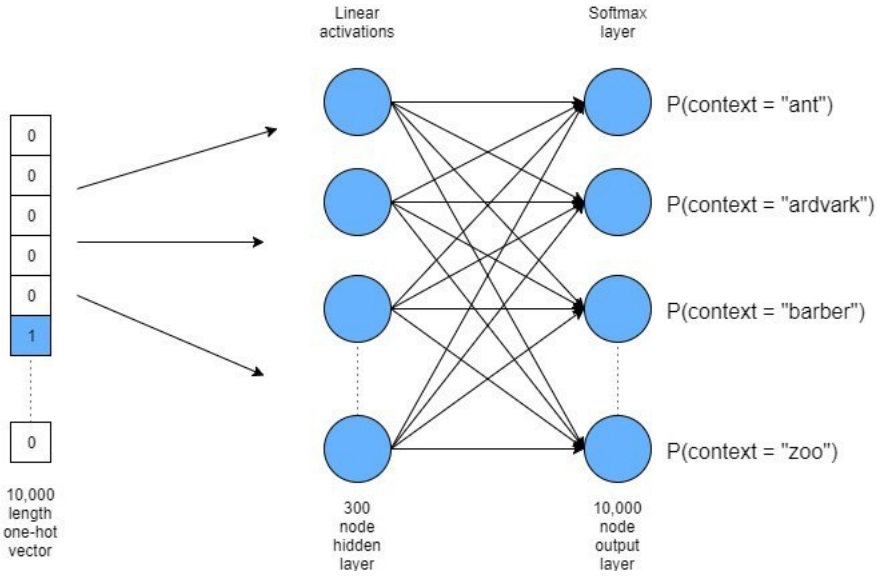


Figure 9: The Word2Vec skip-gram model

The loss function used to train this network is called the "negative log-likelihood"

$$J(\theta) = -\frac{1}{T}\sum_{t=1}^{T}\sum_{-m\leq j\leq m} log(p(w_{i+j}|w_i;\theta))$$

Where T is the number of word in the input text, m is the size of the context window, and $p(w_{i+j}|w_i;\theta)$ is the predicted conditional probability of the word w_{i+1} appearing near word w_i when the network has weights and biases θ. What this loss function does is incite the model to give higher probabilities to words that actually appear near the target word.

3.3.2 Interpreting word embeddings

The vector space produced by algorithms such as Word2Vec is called a word embedding space and the vectors are called the embeddings of the words. They have proven to produce better results when fed to machine learning algorithms, but just by looking at the embedding space it is possible to notice interesting qualitative properties. For example words that have similar meaning or similar role in a sentence are close to each other in the vector space (for example the vector corresponding to "cat" and the one corresponding to "dog"). Also pair of words that share similar relationships are located in the same relative places, and we find equations such as

$$king - man \approx queen - women$$

$$China - Beijing \approx France - Paris$$

3.4 Translation

Translation is one of the hardest tasks in NLP, because of course translating a sentence from a language to another is much more than simply translating one word at a time using a bilingual dictionary. Order of the words may vary, languages can have very different grammatical structures, use auxiliaries, implicit pronouns, prepositions, or even idiomatic expressions, which become entirely different sentences in the other language if we want to stay as close as possible to the original meaning. Another reason why it is a difficult task is that it is not clear how one should evaluate a translation : How can we say that a translation is a good translation ? How can we systematically compare two translations and say that one is better than the other ? Even professional translators often disagree when it comes to comparing translations. These questions remain mostly unanswered or partially answered. Deep learning algorithms still brought some breakthroughs in this field with so-called "sequence-to-sequence models". As their name indicate, these models take a sequence as input just like RNNs, and they also output a sequence after processing the entire input. They are sometimes called generative neural networks because they are able to generate a sequence of variable length (which is what we want in the case of translation). Loosely, the network reads the input sequence (the sentence we want to translate) and produces a vector that captures the meaning of the sentence. Then, it uses this vector to predict the first word of the translation only. After that, it uses both the vector and the predicted word to predict the second word, and so on, until it predicts the end of the sentence. These models, also called "encoder-decoder models" because they encode the first sentence into a vector space and then "decode" it in the target language, have proven very effective, especially since attention mechanisms were invented (giving the network the ability to focus on certain parts of a sentence while generating its translation). We don't describe these algorithms in details as they wasn't the subject of this internship.

3.5 Other tasks

There are many other tasks in NLP for which neural networks have proven useful. We briefly describe some of them here

- Sentiment analysis : given a text (usually a web comment), determine the emotional state the writer was in when writing this text
- Genre identification : given a text, determine if we are dealing with a news article, a scientific paper, a science-fiction novel...
- Named Entity Recognition : given a text, find portions that function together to form a single entity (e.g. Apple Inc. is a corporation and not a fruit, 5 th June 2019 is a date)

4 Designing and Implementing POS Tagger models

The main project I worked on during the first two weeks on my internship was to implement my own POS Tagger using deep-learning techniques. The models were programmed in Python using the PyTorch library. The PyTorch library is a deep-learning framework that provide convenient classes that one can inherit to produce new neural networks, while automating differentiation of all the computations (using computational graphs as described above). PyTorch also provides a way of communicating with GPUs, to train models in a parallelized way, which dramatically speeds up computations.

4.1 Linear Classifier

The first, 'naive' model is a multi-layer perceptron, i.e. a feed-forward neural network that takes as input words of a sentence one at a time, make them go through two layers (the embedding layer and the classification layer), and outputs the corresponding POS tag. Word embeddings are learned on the fly with the rest of the structure. All weights are initialized randomly. Of course the results were not very satisfying, and this

kind of structure has no sense of context, it processes every word in parallel with no interaction, and always assigns the same POS tag to a word regardless of where it stands in the sentence.

The training was done by feeding data to the network in the form of batches. Each batch contains a few sentences (from 4 to 32 for example) and the gradient is computed as an average over the batch. This way of training is called batch gradient descent as we have seen in section 1.3.

All experiments and their results are described at the end of this section.

4.2 LSTMs

The next step in implementing the POS tagger was to use LSTMs instead of a simple neural network. The PyTorch library saves the hassle of implementing all the control gates and provides us with a python class that is able to take as input batches of sentences and treat them sequentially.

4.3 biLSTMs

LSTMs enable us to achieve better tagging, because they are able to take into account previous words to determine the tag of the next word. However for the first words of the sentence, There are no or few words to take into account, and at this point there is no reason why LSTMs should perform better than simple neural networks. A simple way to correct this is to reverse the order of the words in the sentence before feeding it to the LSTM network. This way, the network tags words while taking into account the words that come after in the sentence. But then we get the opposite problem : words at the end of the sentence are tagged without context information. This where BiLSTM networks come into play : in a BiLSTM network, we give the sentence as input for a normal LSTM layer, and in parallel we give the reversed sentence to another LSTM layer. We then get two output instead of one, and we just concatenate them into a single vector that captures context information both from previous and following words in the sentence. Then we just need a classifier layer to output the probability of a POS tag. Here again, PyTorch comes with an implemented version of BiLSTMs, but I coded this model myself as an exercice. The two gave the same results, but the one from the library was a bit faster, because of the fact that PyTorch functions are implemented directly in C (as extensions of Python's built-ins) which is a faster language than Python.

4.4 Multi-Layers BiLSTM

To improve the model even further, the next step was to add layers. The BiLSTM networks described above can be stacked in several layers, in the same ways neurons are organized in layers in a feed-forward neural network. The output of the first BiLSTM is fed into a second BiLSTM, and so on. This allows the network to capture more complex and intricate features to make better sense of the input sentences.

4.5 Incorporating XLM pretrained Embedding

The last step to improve the POS tagger was to use pre-trained embeddings. In all models described before, the embedding layer (that maps a word to a vector in a vector space) was randomly initialized and trained with the network. However it is possible to train embeddings beforehand (using algorithms such as Word2Vec) and to include these embeddings in the model. In this case, the first layer is "frozen" which means its weights are not updated as the networks trains. The embeddings used came from a state-of-the art deep learning model called XLM. XLM is a cross-lingual language model, which means it learns from many languages at the same time and does "generative pretraining", i.e. it learns to predict missing words in sentences and builds embeddings based on this task. Its structure is based on the transformer model, which is a recent model that uses attention mechanisms without recurrent neural networks, allowing it to train faster on lots of GPUs because transformers are more convenient for parallelism. Incorporating pretrained embeddings just consists of loading the weights of the first layer (instead of initializing them randomly) and freezing them. The network is then supposed to have a beforehand understanding of relations between

words. The embeddings are even contextualized, which means the vector for a word is not fixed, but varies according to the context words in the sentence. Indeed the results improved when using these embeddings.

4.6 Hardware used

To train the different models, I used different platforms. At first, I trained them on my own laptop, which has the following specifications :

- CPU : Intel ^{R} Core ^{TM}
  i7-7500 U, 2,70GHz
  2 Cores, 2 threads per core
- GPU : GeForce 940MX,
  2GB of memory
- RAM : 8GB

This setup was sufficient to train the linear classifier and the one-layer LSTM network, however as there began to be several layers, the time it took to go over all the training data was too long to do many experiments, and I had to resort to the Lattice's server. The lab server had the following specifications :

- CPU : Intel ^{R} Core ^{TM}
  i7 U, 2,60GHz
  6 Cores, 12 threads per core
- GPU : 2 GeForce 1080Ti
- RAM : 64GB

With this setup, it was possible to train two models in parallel, because the CPU charge is still low enough that it can handle two models, one on each GPU. This sped up the time it took me to conduct experiments, once I figured out how to use a server through the SSH protocol. However, 2 GPUs is still not an incredible amount of computing resources, and Serge Sharoff, who initially wanted to conduct an experiment where we would have trained a replicated version of Google's BERT, noticed it would take about 99 days to properly train the model on our setup, which is more time than it usually takes for the state of the art to be surpassed in this field ! In research centers of large companies, they use tens or even hundreds of top-quality GPUs and TPUs (tensor processing units), but having access to such huge infrastructure is rarely accessible in public research laboratories due to lack of funding.

4.7 Results

For each model, time of training and accuracy was computed, and the results are listed in the tables below. The tables display the accuracy for a few relevant numbers of epochs. And epoch is a full round of training over the whole dataset. We observe experimentally that deep-learning models improve when training several times on the same dataset. The "no training" line corresponds to the random initialization of the model's weights. At this stage the network acts as a uniformly distributed random output generator. Since there are 17 POS tags, a model that would produce random outputs is expected to have an accuracy of about $\frac{1}{17} \approx 5.89\%$ on average.

4.7.1 State Of The Art

The POS-tagger was trained on the French-GSD corpus from the Universal Dependencies project. The best POS-taggers for this corpus reach about 97% (The 1st place of the 2018 Conll shared task reaches 96.97%, and [3] reaches 97.11%)

4.7.2 Linear Classifier

Training times on the laptop

Number of epochs	CUDA	CPU
1	1'30"	1'38"
2	2'19"	5'47"
3	3'21"	8'10"
4	4'26"	not tested
10	11'09"	not tested

Accuracies with an embedding space of dimension 100 :

Number of epochs	Accuracy
no training	2-6 %
1	57 %
2	60 %
3	58 %
4	57 %
10	57 %

Accuracies with an embedding space of dimension 50 :

Number of epochs	Accuracy
no training	2-11 %
1	65 %
2	66 %
3	66 %
4	67 %
10	66 %

After the tenth epoch, the linear classifier did not improve. We notice that the embedding space significantly impacts the accuracy of the model. It is a "hyperparameter" of the model, which means a value that can be modified but that is not trained with the network weights. When trying to design the best model possible, choosing the right set of hyperparameters is a difficult task, because it requires to train models over and over again, which is very time-consuming. As my goal with this tagger was to get familiar with deep learning implementation techniques, I did not spend time on adjusting the hyperparameters in the best way possible.

4.7.3 LSTM network

Using LSTM networks, the network could gain context information and produced better results, however the training times were higher.

Accuracies with an embedding dimension of 50

Number of epochs	Accuracy	Time
no training	3-8 %	0
1	84 %	8'40 (cpu)
2	87.7 %	17'43" (cpu)
3	88.6 %	25'21" (cpu)
4	88.8 %	33'17" (cpu)
15	88.7 %	18'43" (cuda)
50	88.7 %	46'19" (cuda)

Several experiments were trained in parallel on my laptop to gain time (one on CPU and one on cuda GPU), and this is why some of them took longer. Accuracies with an embedding dimension of 100

Number of epochs	Accuracy	Time
no training	2-7 %	0
1	87 %	1'42 (cuda)
2	89 %	3'24" (cuda)
3	90 %	5'06" (cuda)
4	90.2 %	6'48" (cuda)
15	90.2 %	25'30" (cuda)

As we can see, in this setup an embedding dimension of 100 produces better results. An interpretation of this result could be that LSTMs produce embeddings that have more meaning and need higher dimensionality to be stored. With the linear classifier, higher dimensions only confused the network because it did not have that much information to create its embeddings. As it is often the case in machine learning, these results interpretation of "why the network works" are only guesses and are hard to verify, and scientists who use machine learning, but do not do fundamental research on it, just try to get intuition on what works and why, to be able to apply it in the right cases.

4.7.4 Multi-Layer BiLSTM

As the training times got much higher for more complex architectures, I had to reduce the number of experiments. I kept training on 5 epochs as the accuracy often did not increase very much after this number.

Number of epochs	Accuracy	Time
5	90.6 %	24'22 (cuda)

The result above was from the Multi-Layer BiLSTM that I coded myself, and I also did experiments with the already existing PyTorch class, which yielded slightly better results :

Number of epochs	Accuracy	Time
5	91.6 %	22'13" (cuda)

4.7.5 with XLM weights

Adding the contextualized embeddings from the state-of-the-art model XLM into the tagger significantly increased the results (by 3 percentage points).

Number of epochs	Accuracy	Time
5	94 %	58'38 (cuda)

However the model was beginning to take unreasonable amounts of time to train, so I did not test up to 10 or 50 epochs as I initially planned. At this point I was not familiar with how the server worked so all the experiments were done on my laptop but they would have been doable on the lab's server, which I used for the second part of the internship. I did not want to loose to much time experimenting since the tagger was only a way to get me to learn about deep learning.

5 Improving a Cross-Lingual Dependency Parser in Low-Resource Scenarios

This is a quote from the introduction of [4] :

> The total number of living languages in the world is estimated at more than 7,000 (Simonsand Fennig, 2017). If we only include the top 100 languages with the largest number of native speakers, they cover about 85 % of the world population. Many languages do not have sufficient NLP resources, such as annotated word lists, syntactic parsers or Named Entity Recognition (NER) tools. For example, Balochi, Belarusian and Konkani share the rank of 98–100 in this list with $\approx$ 8M speakers each, which is more than the number of speakers of much better resourced languages such as Danish or Finnish, while they have almost no resources. Similarly, Ukrainian with its 30M native speakers occupies the 40th position in this list (the 8th position in Europe), while having very minimal NLP resources.

In NLP, most research is currently focused on improving deep-learning model for high-resource languages, because these models work best when they have plenty of data to learn from. However, it is still possible to train NLP deep learning models when not so much data is at hand. What we worked on during the second part of my internship was to search for techniques and ideas to have deep learning model produce better results in low-resource scenarios. We considered a dependency parser designed by KyungTae Lim that has won 3rd place in the 2018 Conll shared task, and tried to improve it for low-resource languages. We focused on two ideas :

- Embedding space alignment using Levenshtein distance : Serge Sharoff created a better way of aligning word embedding spaces for several languages using sub-word information, to produce pretrained embeddings that could easily be included into a deep-learning model.
- Confidence score : To improve transfer learning, we experimented with an idea of confidence score, which consists of weighing data during learning so that the network would take more into account data that is more relevant (according to pre-computed characteristics).

Most experiments were conducted on English, French and German, as they are the languages I'm most familiar with, so that I could have better insight to debug models and see how promising they were, simulating low-resource scenarios by reducing the amount of data. We also did experiments on Finnish and Northern Sami, two closely related languages but with very few resources for Northern Sami.

5.1 The SEx-BiST Parser

The Dependency Parser we wanted to improve is the SEx-BiST Parser, of which a description can be found in [2]. It takes as input batches of sentences (represented as dependency graphs) and outputs POS, LAS and UAS tags (that describe functions for the token in the sentences as well relations between them. Its inner workings include character-level attention mechanisms which I didn't have to study in detail to make the wanted modifications.

5.2 Cross-Lingual Word Embeddings

The SEx-BiST uses cross-lingual embeddings. We have seen in the previous sections that in NLP, words are coded by vectors in a vector space of typically 100 dimensions called the "embedding space". However when we train a model on several languages, each word has a label depending on which language the word is from. Therefore there are as many vector spaces as there are languages. In a cross lingual embedding approach, we train the model using one single vector space in which we encode all words regardless of from which language they are from. We then hope to better capture meaning across languages and to have words that are translations of each other have special relations within the cross-lingual vector space. When training the

parser, as we have seen with the POS tagger, we might want to use pre-trained embeddings to improve our results. However, provided several embedding spaces (one for each language), how can we code the word information within a single vector space, in a way that makes sense considering the relations between the languages ? One technique to do this is embedding space alignment which is described in the next section.

5.3 Embedding Spaces Alignment

One way to encode token information for several languages in a single embedding space is to do embedding space alignment [TODO: add ref]. First, we obtain word embeddings for each language using a technique such as FastText or Word2Vec, then we align the spaces using bilingual dictionaries.

5.3.1 Basic Idea

Say we have two embedding vector spaces E and F both of dimension n. "Aligning" E on F is defined as finding a linear transformation $W : E \to F$ that minimizes the following objective function ψ :

$$\psi(W) = \sum_{i=1}^{N} \|We_i - f_i\|$$

Where $N \in \mathbb{N}$ is the size of a seed bilingual dictionary that contains word vectors from E and their translations in F, $e_i \in E$ and $f_i \in F$ are the pairs of word vectors in this bilingual dictionary. Intuitively, minimizing this function ensures that words that are known to be translations from each other do end up relatively close to each other.

5.3.2 Sub-Word Information using Weighted Levenshtein Distance

The first idea to improve the SEx-BiST parser was to give it better pre-trained cross-lingual embeddings to train upon. As it is more extensively described in [TODO: add ref], an alignment method that takes into account sub-word information was used to produce cross-lingual embeddings. This alignment method is based on cognates, which are pair of words that come from different languages but have the same etymological root. For example, the words "totalement" in French and "totally" in English. The alignment algorithm works in an iterative way, and at each step it executes the following actions :

1. Align the two vector spaces using the method described in [TODO: add ref] using the words in the bilingual dictionary.
2. Identify words that have the same meaning in both languages using cosine similarity and Levenshtein distance.
3. Add these words to the bilingual dictionary

Let's see step 2 in more details. The cosine similarity of two vectors $u, v \in \mathbb{R}^d$ is simply defined as the cosine between the two vectors (close to 1 if the vectors are similar, else close to -1). The Weighted Levenshtein Distance (WLD) is a modified version of the Levenshtein Distance (which is the minimum number of insertions, deletions or substitutions required to change one word into the other). In the weighted version, words are aligned based on parts on the words that correspond to one another, but that can be longer than single characters. The frequency of the alignment is computed on a large corpora, and the formula for the WLD is given as :

$$WLD(s_e, s_f) = \frac{\sum_{(e,f) \in al(s_e, s_f)} (1 - p(f|e))}{max(len(s_e), len(s_f))}$$

Where s_e and s_f are two words, one from each language. $al(s_e, s_f)$ is the set of alignments of those two words (on the sub-word level) and $p(f|e)$ is the probability of the alignment (computed using an alignment model learned from the alignment frequencies).

As a result, we get a similarity score for pairs of words defined as

$$score(s_e, s_f) = \alpha cos(s_e, s_f) + (1 - \alpha) WLD(s_e, s_f)$$

Where α is an external parameter that can be adjusted. With this approach we see that the alignment takes into account a larger and larger seed dictionary of words that are supposed to be translations of each other (they have the same meaning in the previous cross-lingual space and they "look alike", i.e. they are close according to the WLD, orthographically speaking). However we have to be careful to keep α big enough so that the cosine similarity is still taken into account, because trying to identify word pairs using only WLD leads to false-friend pairs. False-friends are words that orthographically look similar but have very different meaning. For example "Éventuellement" in French and "Eventually" in English are close for the WLD but have very different meanings.

5.3.3 Results

The pretrained cross-lingual embedding space for Finnish and Northern Sami was computed and given to me by Serge Sharoff. I modified the parser under study to have train on those two languages and included into the model the pretrained embeddings. As preliminary experiments were looking promising, I launched the training over the entire UD corpus, which took about 20 hours to complete. Two training sessions were done in parallel, one without the pretrained embeddings, and one with them. We observed a clear improvement over the baseline model : 71.5% of accuracy with the pretrained cross-lingual embeddings versus 69.4% for the baseline model, i.e. an increase of 1.1 percentage points. The model was trained for 180 epochs using a batch size of 32 sentences per batch. A complete learning curve can be found in Fig 10

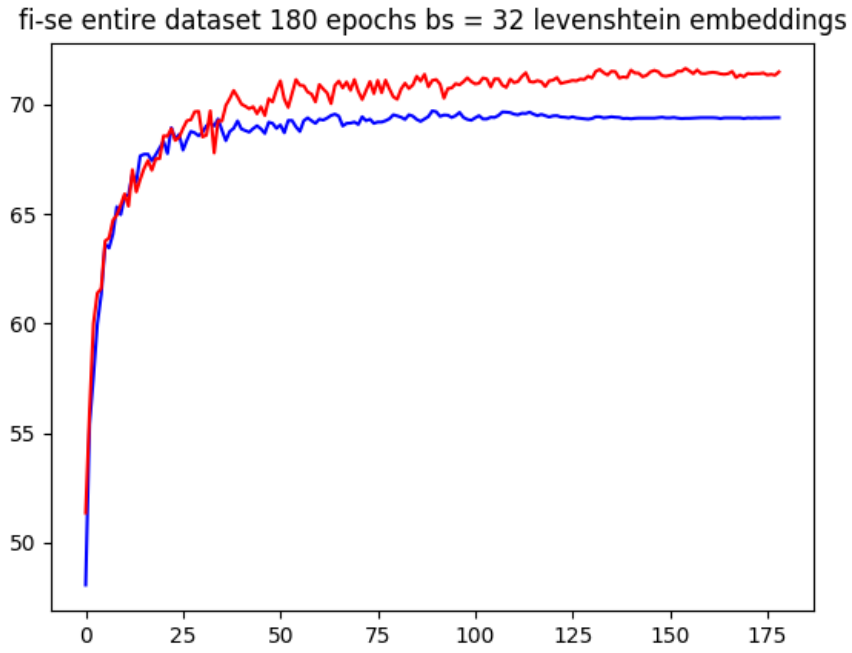


Figure 10: Learning curves for Finnish (high-resource language) and Northern-Sami (low-resource language). Baseline model in blue, Experimental model (using leveshtein embeddings) in red

5.4 Confidence Score

When using transfer learning to improve results on a target (low-resource) language, we hypothesized that there could be a danger of learning features (from the high-resource language) that are irrelevant for the task on the target language. This problem is minimized by choosing languages that are closely related (e.g Spanish and French), but even then some sentences in the high-resource language will sometimes have syntactical structures that are rarely found in the low-resource language. Intuitively, we would like the model to know which input sentences are relevant for training and which are not, and scale its gradient descent steps according to how "confident" it is that the knowledge acquired will transfer. To achieve this, we introduce a confidence score computed prior to learning for each sentence in the training corpus. When learning, we then multiply the loss by the confidence score. Since differentiation is a linear operation, scaling the loss function also scales the gradient for each input sentence. With this approach we hoped to obtain batch gradient descent steps more oriented towards learning relevant features for the low-resource language.

5.4.1 Computing confidence scores

To decide how to compute a confidence score for a given sentence, one must first have an idea of what are the relevant features in the data that will be useful for transfer learning. Let us consider the two following sentences in German and their translations in French (each word has been assigned a number to keep track of its translation):

Der	kleine	Vogel	Singt	in	der	Eichen	.
1	2	3	4	5	6	7	8
Le	petit	oiseau	chante	dans	le	chêne	.
1	2	3	4	5	6	7	8

Mit	so	einem	langweiligen	Spiel	kann	man	überhaupt	keinen	Spass	haben	.
1	2	3	4	5	6	7	8	9	10	11	12
On	ne	peut	pas	du tout	s'amuser	avec	un	jeu	si	ennuyeux	.
7	_	6	9	8	10-11	1	3	5	2	4	12

If we want to transfer some knowledge from a German corpus in order to apply it to French, and since the task we are trying to improve is part-of-speech tagging and dependency parsing, it is clear that the first sentence should weigh more in the process, since its structure is much more similar to that of a typical French sentence.

To formalize this idea we proposed a simple statistical approach based on part-of-speech n-grams frequency.

Given a sentence from the high-resource language $S_{hr} = u_1, u_2, ..., u_p$ (where u_i are POS tags), the confidence score is computed using the following formula :

$$\frac{1}{p} \sum_{i=1}^{p} \sum_{k=1}^{p-i+1} occ(u_k, ..., u_{k+i})$$

Where the *occ* function maps a POS sequence to its number of occurences in the low-resource corpus. This formula captures the fact that sentences containing POS sequences that are more frequent in the low-resource language should have a higher confidence score. We divide by the length of the sentence so that longer sentences don't systematically get a higher score than short ones.

5.4.2 Normalization

Before incorporating them into the model, it is necessary to normalize the confidence scores, so that the gradient descent takes steps of similar sizes compared to the baseline model. To do this we used a divide-by-mean approach :

$$N(c_i) = \frac{c_i}{\bar{c}}$$

where the c_i are the confidences for the n sentences of the corpus, $\bar{c}$ is the arithmetic mean of $(c_1, \ldots, c_n)$ and N is the normalization function

This normalization technique ensures that confidences scores stay relatively close to one (they typically ranged from 0.2 to 3.0) and that their sum was equal to the number of sentences in the corpus. These two considerations matter since multiplying the loss by a very large (or very small) number causes unwanted huge gradient descent steps (or too tiny steps) preventing convergence.

Of course, sentences from the low-resource language were assigned maximum confidence, since learning from them is sure to positively impact the results when testing on the same language.

5.4.3 Submitting the idea to EurNLP conference

After conducting many preliminary experiments on very small datasets (so that the tests would run in a reasonable amount of time), we concluded that the approach seemed promising in a scenario of zero-shot transfer learning, i.e. when the model is only given data from the high-resource language, and no data at all from the low-resource language. In other scenarios however, adding a confidence score provided no significant improvement over the baseline model. We did not have time to conduct enough experiments, but as some results showed hints that the idea of a confidence score could be useful, we made a one-page submission to present it at the EurNLP summit. Our submission was not selected.

Cross-Lingual Dependency Parsing based on Model Transfer Learning with Sequential Similarity of POS

Anonymous ACL submission

1 Introduction

Because there are lots of languages in the world with little resources, the ability to transfer relevant linguistic information from one language to another is nowadays a key issue in NLP. This approach has been successfully explored in the context of dependency parsing (McDonald et al., 2013; Guo et al., 2015; Ammar et al., 2016; Lim et al., 2018). However these methods do not directly consider syntactic similarity during training. To solve this problem, we propose here a new method that takes into account N-gram based POS similarity on top of features traditionally used for this task. We show that our simple POS sequence similarity feature boosts dependency parsing.

2 Proposed Approach

(Baseline Model) The goal of the dependency parsing is to produce a tree structure for each sentence $x = (w_1, w_2 ...w_m)$ (Kiperwasser and Goldberg, 2016). In general, the parser is represented by learning network parameters θ that maximize the probability $P(y_j|x_j, \theta)$ based on the conditional negative log-likelihood loss (θ):

$$loss = \sum_{(x_j,y_j)\in T} -\log P(y_j|x_j,\theta) \qquad (1)$$

$$\hat{y} = \arg\max_{y} P(y|x_j,\theta)$$

where $(x_j, y_j) \in T$ denotes an input data from training set T, y is a set of gold labels (l^{Head}, l^{Dep}) and $\hat{y}$ is a set of predicted labels. In transfer learning, T is divided into high-resource and low-resource languages as T^h and T^l (Lim et al., 2018).

(N-gram based POS similarity) According to formula (1), the model learns from the high-resource language T^h whatever the sentence structure is, which is not optimal. To solve this problem, we penalize our $loss$ as $\alpha * loss$ by taking into account an N-gram based syntactic similarity score between the two languages. Given a sentence from the high-resource language $x^h = u_1, u_2, ..., u_p$ (where u_i is a POS tag), the confidence score α is computed as:

$$\alpha = \frac{1}{p}\sum_{N=1}^{p}\sum_{k=1}^{p-N+1} occ(u_k, ..., u_{k+N}) \qquad (2)$$

where the occ function maps a POS N-gram to its number of occurrences in the low-resource corpus and N denotes the size of N-gram. Intuitively, sentences containing POS N-grams that are more frequent in the low-resource language should have a higher confidence score to learn from.

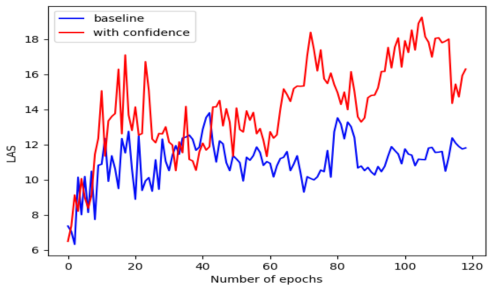


Figure 1: Parsing results on the French-GSD test set.

3 Results and Conclusion

Figure 1 shows the evaluation of our approach on the French UD 2.2 test corpus. To investigate the benefit for an extremely low-resource scenario, we trained our model on 1000 German sentences only (as T^h) and no French corpus. And we computed the confidence score using 1000 French training sentences to build the function occ in (2). We found a significant improvement of 4.7 LAS points of our approach over the baseline method that does not take into account POS sequences. In the future, we plan to test this approach on different languages, like Finnish vs Saami utilizing their genetic similarity (Sharoff, 2019).

Out of 179 valid submissions, the EurNLP team accepted 57 submissions, which yields an acceptance rate of 31.8 %.

Two reviews were given on the submission :

Review 1

SUBMISSION: 57
TITLE: Cross-Lingual Dependency Parsing based on Model Transfer Learningwith Sequential Similarity of POS
AUTHORS: Kyungtae Lim, Noé Malais, Serge Sharoff and Thierry Poibeau

——— Contributions, main strengths and weaknesses ———
The authors propose to weight the loss function for training a model for cross-lingual dependency parsing dependent on the POS tag similarity of the source and the target language.

Strengths:
- The approach is intuitive and well motivated.
- Improvements over the baseline.

Weakness:
This seems like a nice approach, but very similar to work by Dingquan Wang and Jason Eisner from the last years, where they reorder or synthetically generate parsed sentences in a "source language" for cross-lingual transfer of dependency parsing.

——— Overall recommendation ———
SCORE: 3 (Ambivalent: It has merits, but there are some weaknesses.)

——— Justification ———
Results seem convincing, so my biggest concern with this work is its novelty.

Review 2

SUBMISSION: 57
TITLE: Cross-Lingual Dependency Parsing based on Model Transfer Learningwith Sequential Similarity of POS
AUTHORS: Kyungtae Lim, Noé Malais, Serge Sharoff and Thierry Poibeau

——— Contributions, main strengths and weaknesses ———
The paper presents a simple technique for making cross-lingual parsers sensitive to word order similarities between source and target language by putting more weight on training sentences containing pos n-grams that are frequent in the target languages. Preliminary experiments show improvements in accuracy. The main strength of the paper is the simplicity of the technique together with the positive results. The main weakness is the limited evaluation (only one data set). In addition, the approach presupposes that pos-annotated data is available for the target language, which is not always realistic in low-resource scenarios.

——— Overall recommendation ———
SCORE: 4 (Strong: I would like to see it accepted.)

——— Justification ———
The proposed technique appears to be novel and effective enough to justify acceptance, although the experimental evaluation is quite limited. The originality is also somewhat limited, as this is basically a minor variation on known techniques for cross-lingual parsing.

Conclusion

This internship was a great opportunity for me to familiarize myself with machine learning and work on actual implementation of deep learning networks in a research context. I was also able to learn a lot about Natural Language Processing, attended interesting conferences, and got to experience the work ecosystem in a research laboratory, which was very enriching.

The results of the confidence score approach were not very satisfying (small improvement over the baseline and only in the very specific scenario of zero-shot learning). However, they still show that the approach has some potential, and we think it could have been improved with another technique to compute the confidences, for example using a simple language model instead of the POS n-grams statistical approach.

References

[1] George Cybenko. "Approximation by Superpositions of a Sigmoidal Function". In: *Mathematics of Control, Signals, and Systems* 2 (1989), pp. 303–314. DOI: `http://www.dartmouth.edu/~gvc/Cybenko_MCSS.pdf`.

[2] KyungTae Lim. "SEx BiST: A Multi-Source Trainable Parser with Deep Contextualized Lexical Representations". In: *Association for Computational Linguistics* (2018). DOI: `https://www.aclweb.org/anthology/K18-2014`.

[3] Dat Quoc Nguyen and Karin Verspoo. "An improved neural network model for joint POS tagging and dependency parsing". In: (2018). DOI: `https://arxiv.org/pdf/1807.03955.pdf`.

[4] Serge Sharoff. "Finding next of kin: Cross-lingual embedding spaces for related languages". In: *Natural Language Engineering* 1.1 (2019), pp. 1–23. DOI: `http://corpus.leeds.ac.uk/serge/2019-jnle.pdf`.