Deep Learning for Named Entity Recognition

S. Do

June - October 2018

Introduction

This short document is extracted from the report that I wrote for a summer internship during my first year of master's degree in ENSAE Paristech in 2018.

It consists of a short introduction to basic concepts in Deep Learning applied to Natural Language Processing. It thus aims at recalling Deep Learning basics, introducing word embeddings, describing some popular NLP networks as LSTMs, and finally describing a 2016 NER architecture using all the aforementioned concepts.

Future versions of this report might include more recent concepts as attention learning, question answering systems, etc.

Contents

1 Deep Learning Basics

We begin by first laying the basis of Deep Learning, as it is the starting point of modern NLP techniques. To do so, we will first introduce the neuron concept and modelisation. Then, we will describe feed-forward neural networks (FFNNs), and finally we will show how to train FFNNs using backpropagation.

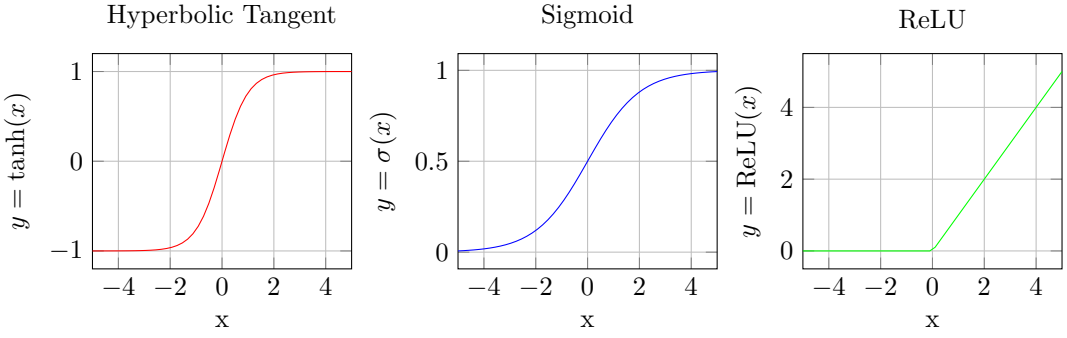
1.1 Neuron

The idea of copying and simplifying the most important cell in the human brain was first explored by Rosenblatt's Perceptron, described in 1957 in [12]. Some changes have been made since this first modelisation, but it stills somewhat ressembles the biological model of a neuron : a biological neuron receives signals from other neurons and outputs a signal in response, depending on some activation rules. An artificial neuron does approximately the same thing : it takes n signal inputs $(x_1, x_2, ..., x_n)$, and outputs the signal $y = f(w_1x_1 + ... + w_nx_n)$, where f is called the *activation function*, and $w_1, ..., w_n$ are called the weights of the neuron.

Basically, if $f = Id$, the neuron performs a linear regression of y on the $(x_i)_{i=1,...,n}$ by fitting the weights on sample data.
Usually, chosen activation functions are the sigmoid (σ), the hyperbolic tangent (tanh) or the ReLU functions:

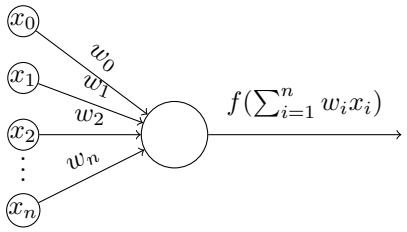
$$\begin{aligned} \forall x \in \mathbb{R}, \sigma(x) &= \frac{1}{1+e^{-x}} \\ \forall x \in \mathbb{R}, \tanh(x) &= \frac{e^{2x}-1}{e^{2x}+1} \\ \forall x \in \mathbb{R}, \mathrm{ReLU}(x) &= \max(0, x) \end{aligned}$$

Figure 1: Most common activation functions plots



Functions such as sigmoid and hyperbolic tangent are chosen for their non-linear and bounded nature. Neurons are usually schematized as in Figure 2.

Figure 2: Representation of a neuron

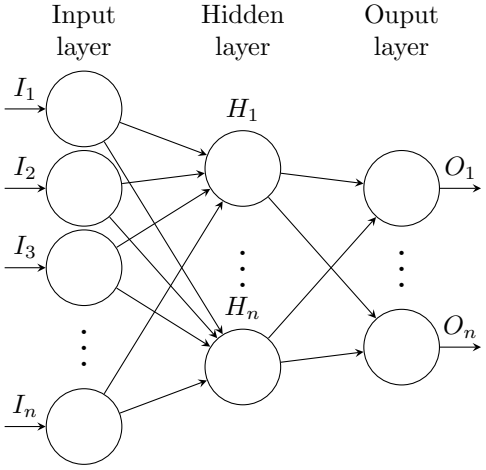


Though, modelling data with a single neuron doesn't bring any innovation : it is like doing linear, logistic, or censored regressions. Therefore, neurons are never used alone : the interesting thing with neurons is to stack them and turn this into a *network*. We thus explain the simplest model of neural network: the feed-forward neural network.

1.2 Feed-Forward Neural Networks

The feed-forward neural network (FFNN) stacks and parallelizes neurons, as schematized in Figure 3. In a network containing $L \in \mathbb{N}$ hidden layers, size $n_1, n_2, ..., n_L$ respectively, the inputs of the network are first fed to the first layer's neurons (n_1 neurons). Each neuron has its own weights and computes an output to the signal he receives from the inputs. The n_1 generated outputs are then fed to the n_2 neurons of the next layer, and so on.

Figure 3: Representation of a feed-forward neural net with a single hidden layer



One important thing about feed-forward neural networks is the word "forward" in their name. Indeed, FFNNs only pass the information they get forward to the next layer. There isn't any kind of recurrence in them, and that's what makes FFNN so simple. Their apparent simplicity is though capable of approximating any measurable function, even with a single hidden layer, if the dimensions of the layer(s) are large enough, as proven in Kurt Hornik, 1989 [5]. Not all networks keep their "forward" character : for instance, neural nets such as RNNs and LSTMs introduce recurrence. Their are developped in section 3.2.

For further developments, let's adopt the following notations : let L be the number of hidden layers, and n_l the number of neurons in the layer l. The i-th neuron in the l-th layer is noted $N_i^{(l)}$.

$N_i^{(l)}$ has n_{l-1} weights : let $w_{k,i}^{(l)}$ be the weight of the edge linking $N_k^{(l-1)}$ to $N_i^{(l)}$. If we denote $z_i^{(l)}$ the output of $N_i^{(l)}$, then it can be computed with:

$$z_i^{(l)} = f\left(\sum_{k=1}^{n_{l-1}} w_{k,i}^{(l)} z_k^{(l-1)}\right)$$

with f the activation function of $N_i^{(l)}$ (to simplify notations, we put the same activation to all neurons and all layers).

If we now denote $Z^{(l)} = (z_i^{(l)})_{i=1,...,n}$, we then have the following matrix notation for layer l:

$$Z^{(l)} = f\left(W^{(l)} Z^{(l-1)}\right)$$

where f is of course applied to each coordinate of $W^{(l)}Z^{(l-1)}$, and with $W^{(l)} \in \mathbb{R}^{n_l \times n_{l-1}}$:

$$W^{(l)} = \begin{pmatrix} w_{1,1}^{(l)} & \dots & w_{n_{l-1},1}^{(l)} \\ w_{1,2}^{(l)} & \dots & w_{n_{l-1},2}^{(l)} \\ \vdots & & \vdots \\ w_{1,n_l}^{(l)} & \dots & w_{n_{l-1},n_l}^{(l)} \end{pmatrix}$$

The set containing all the weights matrices will be denoted $\mathcal{W} := {W^{(l)}}_{l=1,...L}$ for further use.

A question remains for the output layer. If the aim is to do a regression, then classic activation functions are used. However, when trying to do classification, the ouput layer is a little bit modified. For a problem with K classes, the size of the ouput layer is set to $N^{(l)} = K$, and the chosen "activation" function is the softmax function :

$$\forall i = 1, ..., K, z_i^{(L)} := \frac{e^{a_i^{(L)}}}{\sum_{k=1}^{K} e^{a_k^{(L)}}}$$

where $\forall i = 1, ..., K, a_i^{(L)} := \sum_{k=1}^{n_{l-1}} w_{k,i}^{(l)} z_k^{(l-1)}$
This function represents the probability for an input x to belong to the i-th class.

The weights, the architecture, and the activation functions thus entirely define the network. Still, weights are an unknown parameter that has to be estimated with data. Let's go back to the classic statistical learning paradigm and let $(x_i, y_i)_{i=1,...,n}$ be our independant observations, drawn from an unknown probability distribution function called P. When a given x_i is fed to the network, weighted sums are computed in all layers, activation functions are activated, and finally the network outputs a $\hat{y}_i = \text{FFNN}(x_i|\mathcal{W})$ response. Our aim is to learn the weights $\mathcal{W}$ from the data by minimizing the error between y_i and $\hat{y}_i$. We thus have to define a loss functions, which quantifies this error. A standard expression of our minimization problem that sums up all this could be the following: find $\mathcal{W}*$ so that:

$$\mathcal{W}* \in \arg\min_{\mathcal{W}} \mathcal{L}(\mathcal{W}) + \lambda\Phi(\mathcal{W})$$

With $\mathcal{L}$ being the empirical risk term, or the **loss function**, and $\lambda\Phi$ being a regularization term. Regarding regression, usual losses apply : mean squared error (MSE), $\mathcal{L}_2$ loss function, mean absolute error, $\mathcal{L}_1$ loss function, etc... Still, some other useful losses are used in deep learning, as the Kullback Leibler divergence. Regarding classification, usual loss functions are the following : cross-entropy, negative logarithmic likelihood, hinge loss, etc. We recall the expression of the loss function that is going to be useful for the following : the cross-entropy.

Cross-entropy loss is adapted to networks in which the output layer function is a softmax, or more generally to a function that returns the probability to belong to each class. This loss comes from the negative logarithmic likelihood :

$$\mathcal{L}((x_i, y_i)_{i=1,\dots,n}) = -\sum_{i=1}^{n} \ln p(y_i|x_i, \mathcal{W})$$

We need to define more precisely what $p(y_i|x_i, \mathcal{W})$ really is. As we are in a classification context, we could write the $(y_i)_i$ as one-hot vectors: if a given example y_i belongs to the j-th class, then y_i is a vector of zeros excepted on its j-th coordinate, which is a one. By denoting C_k the k-th class, we have:

$$p(y_i|x_i, \mathcal{W}) = \prod_{k=1}^{K} p(C_k|x_i)^{y_{i,k}}$$

where $y_{i,k}$ is the k-th coordinate of y_i sample. As $p(C_k|x_i)$ is the result $\hat{y}_{i,k}$ of our softmax function, we can finally re-write :

$$\mathcal{L}(x_i, y_i) = -\sum_{k=1}^{K} y_{i,k} \ln(\hat{y}_{i,k})$$

1.3 Backpropagation Algorithm

Now that we have chosen a loss function to quantify the estimation error made by the network, we can write the gradient that we would like to compute in order to minimize the loss over all observations contained by our dataset:

$$\frac{\partial \mathcal{L}((x_i, y_i)_{i=1,\dots,n})}{\partial \mathcal{W}} = \sum_{i=1}^{n} \frac{\partial \mathcal{L}(x_i, y_i)}{\partial \mathcal{W}}$$

Unfortunately, the loss $\mathcal{L}(x_i, y_i)$ is difficult to differentiate with respect to $\mathcal{W}$, as $\mathcal{L}(x_i, y_i) = -\sum_{k=1}^{K} y_{i,k} \ln(\hat{y}_{i,k})$ and $\hat{y}_{i,k} = z_{i,k}^{(l)}$ results from a complex computation regarding the weights of the networks. The **backpropagation** algorithm tries to solve this problem by doing the following :

- It starts first by computing, for a given x, the output $\hat{y} = \text{FFNN}(x)$. This is called the forward pass.

- Then, we are interested in quantifying the variation of the error when weights from the output layer change. In other words, we are interested in computing:

$$\frac{\partial \mathcal{L}(x,y)}{\partial w_{i,j}^{(L)}}$$

Using chain rule, we have:

$$\frac{\partial \mathcal{L}(x,y)}{\partial w_{i,j}^{(L)}} = \sum_{k=1}^{K} \frac{\partial \mathcal{L}(x,y)}{\partial a_k^{(L)}} \frac{\partial a_k^{(L)}}{\partial w_{i,j}^{(L)}} \tag{1}$$

$$= \frac{\partial \mathcal{L}(x,y)}{\partial a_j^{(L)}} \frac{\partial a_j^{(L)}}{\partial w_{i,j}^{(L)}} \tag{2}$$

$$= \delta_j^{(L)} \, z_i^{(L-1)} \tag{3}$$

(2) is obtained from the fact that if $k \neq j$, then $\frac{\partial a_k^{(L)}}{\partial w_{i,j}^{(L)}} = 0$ and (3) is obtained from the definition of $a_j^{(L)} = \sum_{k=1}^{K} w_{k,j} z_j^{(L)}$ which gives: $\frac{\partial a_j^{(L)}}{\partial w_{i,j}^{(L)}} = z_j^{(L-1)}$ Finally, $\delta_j^{(L)}$ is defined as $\delta_j^{(L)} = \frac{\partial \mathcal{L}(x,y)}{\partial a_j^{(L)}}$

- Let's now interest ourselves in $\delta_j^{(l)}$, for any $l = 1, ..., L-1$. Using chain rule,

$$\delta_j^{(l)} = \sum_{k=1}^{n_{l+1}} \underbrace{\frac{\partial \mathcal{L}(x,y)}{\partial a_k^{(l+1)}}}_{=\delta_k^{(l+1)}} \frac{\partial a_k^{(l+1)}}{\partial a_j^{(l)}} \tag{4}$$

And we also have by definition:

$$a_k^{(l+1)} = \sum_{k'=1}^{n_l} w_{k',k}^{(l+1)} f(a_{k'}^{(l)}) \tag{5}$$

$$\Leftrightarrow \frac{\partial a_k^{(l+1)}}{\partial a_j^{(l)}} = w_{j,k}^{(l+1)} f'(a_j^{(l)}) \tag{6}$$

Thus, (4) can be re-written as follows:

$$\delta_j^{(l)} = f'(a_j^{(l)}) \sum_{k=1}^{n_{l+1}} \delta_k^{(l+1)} w_{j,k}^{(l+1)} \tag{7}$$

which defines some kind of inverse recurrence relationship.

- Finally, the three following equations resolve our problem :

$$\begin{cases} \frac{\partial \mathcal{L}(x,y)}{\partial w_{i,j}^{(l)}} &= \delta_j^{(l)} \ z_i^{(l-1)} & \forall l = 1, ..., L-1 \\ \delta_j^{(l)} &= f'(a_j^{(l)}) \sum_{k=1}^{n_{l+1}} \delta_k^{(l+1)} w_{j,k}^{(l+1)} & \forall l = 1, ..., L-1 \\ \frac{\partial \mathcal{L}(x,y)}{\partial w_{i,j}^{(L)}} &= \delta_j^{(L)} \ z_i^{(L-1)} \end{cases}$$

  Those three equations mean that if we can compute the gradients with respect to the weights of the last layer, then, with the second equation of the system, we are able to go a step backward, and compute the gradients with respect to the weights of the previous layer. By backward induction, all gradients with respect to all weights can be computed.

The backpropagation algorithm is at the heart of deep learning. The process, proposed in 1986 indepently by the Rumelhart, Hinton and William team and Yann LeCun, gave a new breath to neural network theory, allowing to train efficiently FFNNs on more and more powerful computers. Even in more advanced architectures as LSTMs, the backpropagation algorithm holds with small modifications.

Once gradients can be computed thanks to the backpropagation algorithm, various optimizers can be used to minimize the loss function. In deep learning, most common optimizers are Adam optimizer, or the stochastic gradient descent (SGD). During the learning process, the training dataset is divided into batches, passed one by one to the optimizer. The dataset is generally seen many times by the network, with different batches division, or with batches presented in different order. The process of seeing one time the whole dataset is called an epoch. Training a NN on 100 epochs means seeing 100 times the whole dataset.

Now that we know how to train a FFNN, we can introduce a simple, but very effective FFNN that learns continuous and dense word representations : the Word2Vec.

2 Continuous Representation of Words

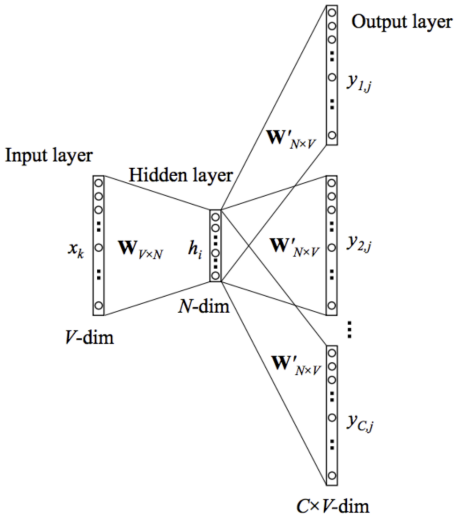
Representing words as vectors is a difficult task, and the idea has been explored since at least the 1970s, when word vector space models were created. There are many reasons for representing words as vectors: float vectors are a good input to other models, the kind of inputs learning theory has been designed for. For numerous reasons, many simple models provide a sparse and high-dimensionnal representation of words : for instance, the simplest model, one-hot encoding, causes sparsity and high-dimension as in this model, given a vocabulary V of size $|V|$, each word $w \in V$ is represented by a one-hot vector $x_w \in \mathbb{R}^{|V|}$.

Trying to address this issue in word representation, people have been searching for dense and continuous representation of words, leading to a groundbreaking article by Mikolov and al., 2013 [9], [8]. Their approach uses simple feed-forward neural networks. In this section, we will detail Mikolov's Word2Vec architecture, and learning techniques. Word2Vec is a generic word for two neural networks architectures, Skip-Gram and Continuous Bag-of-Words (CBOW), proposed by Mikolov et al., 2013, [8]. These networks aims both at a *fake task*: predicting the neighbourgh words surrounding a given word for Skip-Gram, predicting a word from its context for CBOW. In this section, we are going to focus on the Skip-Gram model. We will first describe the architecture, as in the first paper [8] , and then we will discuss further methods, described in [9], adopted to learn the representations more efficiently.

2.1 Skip-Gram Architecture

The Skip-Gram architecture is the one of a single hidden-layer neural network, as it can be seen in Figure 4. Learning pairs proposed to the network are a word (w_t) as the input and its context with respect to a certain window size C : $(w_{t-k}, ..., w_{t-1}, w_{t+1}, ...w_{t+k})$ as the ouput, so as to the network can learn to predict a context by its center words. In order words, the network performs a classification : each word being a class, the network tries to link a class to a set of C (the window size) classes (= the C surrounding words). As the networks performs a classification, the ouput layer's activation function is a softmax.

Figure 4: Representation of skip-gram



Inputs and outputs are clear, but the projection layer is still a little bit mysterious. It in fact consists of $N \in \mathbb{N}$ neurons, where N is the size of the embeddings that we want. The activation function is the identity for all N neurons. If the vocabulary size is $V \in \mathbb{N}$, the number of classes is also V (each unique word is a class). Thus, the dimension of the weight matrix $W = W^{(1)}$, linking the inputs to the projection layer, is of size (V, N).

We then define for a given one-hot encoded word X its embedding, called X_{emb} by the following:

$$X_{\text{emb}} = X^T W \tag{8}$$

$$X = \begin{pmatrix} 0 \\ \vdots \\ 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{pmatrix} \in \{0,1\}^V, \quad W = \begin{pmatrix} w_{1,1} & \dots & w_{1,N} \\ \vdots & & \vdots \\ w_{V,1} & \dots & w_{V,N} \end{pmatrix} \in \mathbb{R}^{(V,N)}, \quad X_{\text{emb}} \in \mathbb{R}^N$$

The question of the chosen training loss is left for section 2.3, where learning process and improvements are described more completely.

An important thing to note is that the coherence of Skip-Gram embeddings is underlayed by a fundamental linguistic hypothesis, called the "distributional hypothesis". According to this hypothesis, words who appear in similar contexts as more likely to have the same meaning. This idea is summed up by a very popular quote from Firth, 1957 : *"A word is characterized by the company it keeps"*. In Skip-Gram, words who appear in the same context are more likely to have similar vector representations, and then if the ditributional hypothesis is to be true, words with close representation are likely to have the same meaning. That is what empirical properties of the Word2Vec embeddings tend to show.

2.2 Word Embeddings Properties

As word embeddings are dense vectors in $\mathbb{R}^N$, we can compute distances between them. Usually, it's the cosine similarity that is chosen, instead of the usual euclidian distance : for two word embeddings $X_{1,\text{emb}}, X_{2,\text{emb}}$, we have:

$$\text{Cosine similarity}(X_{1,\text{emb}}, X_{2,\text{emb}}) = \frac{X_{1,\text{emb}} \cdot X_{2,\text{emb}}}{||X_{1,\text{emb}}\ ||\ \cdot ||X_{2,\text{emb}}||}$$

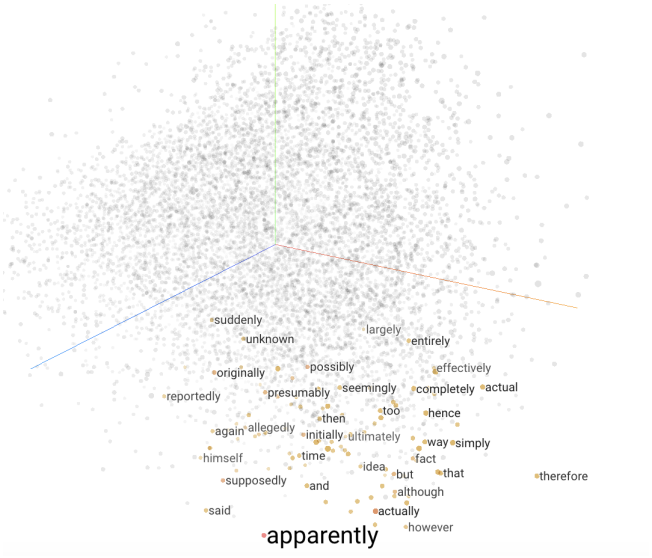
Using this, we can evaluate similarity between word embeddings. As expected, words with similar meanings are close. Moreover, it is very naturally to try to visualize the word embedding space. Using classic PCA or t-SNE, dimension can be reduced to a 3D or 2D space, creating interesting maps [1]. Using PCA, and following the most similar words in original space, we provide following examples:

Table 1: 3 Most similar words to four usual words

[1] Tensorflow provides a useful tool for this : https://projector.tensorflow.org/

simple	university	topological	application
complex	college	hausdorff	applications
useful	school	topology	interface
complicated	harvard	spaces	software

Figure 5: PCA projection of 200-D Word Embeddings using Tensorflow Projector

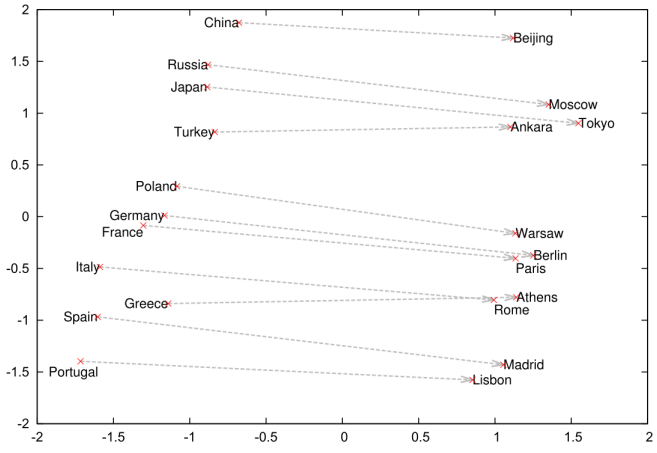


A more impressive thing is the algebraic relationship that can be explored with the word embeddings. For instance, the vector resulting of the following operations:

$$X = \text{king}_{\text{embedded}} + \text{woman}_{\text{embedded}} - \text{man}_{\text{embedded}}$$

has the vectors representing the words "queen", "princess" as nearest vectors. Thus, calculating all the differences between capitals and their countries, as done in [9] gives colinear vectors, when projected in 2D with a PCA. This can be seen in Figure 6

Figure 6: 2D PCA representing words vectors differences, from [9]



2.3 Negative sampling, Word Phrases Recognition and Subsampling

Up to now, we haven't talked about the caveats of the training. One of the main difficult points in Word2Vec training is the size of vocabularies. Large vocabularies imply large matrices, but also a high computational cost for the softmax computing. Thus, in [9], Mikolov develops techniques to address those issues. The first technique, hierarchical softmax focuses on softmax's computational cost reduction. The second technique, called negative sampling, tries to choose a relevant loss function. Finally, the vocabulary is downsampled in two steps : first, "word phrases" (tokens that contain more than a unigram, like "New York City" for instance) are detected and gathered as a single token. Then, a subsampling is done with regards to words frequency.

We only describe the techniques used in the NER architecture : negative sampling, word phrases recognition, and frequency subsampling. Let $\mathcal{W} = \{w_1, ..., w_T\}$ be our word training sequence. Let c be the parameter of our context window size. As a starting point, we could simply define the loss as the cross-entropy loss (Section 1.2):

$$\mathcal{L}(\mathcal{W}) = \frac{1}{T}\sum_{t=1}^{T}\sum_{-c\leq j\leq c}\log p(w_{t+j}|w_t)$$

We also have from the fact that our output layer is a softmax layer:

$$p(w_{t+j}|w_t) = \frac{\exp(e_{t+j}e_t)}{\sum_{v=1}^{V}\exp(e_v e_t)}$$

Where e_t is the vector representation of the word w_t and V the vocabulary size. This expression is computationally expensive, and is thus not very relevant. Among the proposed solution, we adopted the Negative Sampling solution. It consists in re-defining the loss by replacing $\log p(w_{t+j|w_t})$ by the following :

$$\log\sigma(e_{t+j}e_t) + \sum_{i=1}^{k}\mathbb{E}_{w_i\sim P_n(w)}[\log\sigma(-e_i e_t)]$$

This loss defines a new task : being able to distinguish the context word (that we want to predict) from k noises words from our dataset, drawn from a distribution $P_n(w)$. Thus, the training process is less expensive : we choose P_n as a parameter (the paper recommends unigram distribution with power 3/4, chosen empirically), draw k noises words ('negative samples') for each word w_t, and then compute the gradient of the new loss with respect to the weights. An interested reader could find a more detailed explanation in Goldberg et al., 2014 [2].

Now that we have made some optimization on the loss, we are interested in vocabulary subsampling, as computations on large matrices are somehow expensive. In large corpus, vocabulary size generally explodes to several millions of unique tokens, even with cleaning. Downsampling vocabulary is thus absolutely necessary. First thing for tackling this issue: it can be remarked that sometimes a word is formed of two tokens : 'New York City' designs a single concept, but is composed of three tokens. 'Autrian Airline' designs a concept and is made of two tokens. These words are called word phrases. A first idea to downsample vocabulary is to gather word phrases tokens in one single token, for instance 'Austrian_Airlines'. Thus, instead of adding two words to the vocabulary ('Austrian', 'Airlines'), we just add one word. However, this technique is

effective only for words that occur almost only together. We thus define a score for each bigram encoutered in the learning sentences set :

$$\text{score}(w_i, w_j) = \frac{\text{count}(w_i, w_j) - \delta}{\text{count}(w_i) \times \text{count}(w - j)}$$

Where δ is a parameter. We decide to gather tokens of the bigram if the score is above a certain threshold, which is a parameter. The process can be done more than one time to find words phrases composed by more than two tokens.

Finally, we downsample the vocabulary with a last process : frequency subsampling. The process is simple : we discard words if they are too rare. To do so, we discard a word w_i with probability:

$$P(w_i) = 1 - \sqrt{\frac{t}{f(w_i)}}$$

Where t is a parameter threshold and $f(w_i)$ is w_i's frequency over all the corpus.

With these advanced features, Word2Vec is more efficient and less computationally expensive, and that is the reason why the second paper, Mikolov et al. 2013 [9] has so much democratized Word2Vec. After having seen basic neural networks structure, we are going to explain Lample's NER architecture, through the introduction of slightly more complex networks : recurrent neural networks, and their effective counterpart LSTMs.

3 Recurrent neural networks and LSTMs

An important question motivating innovation to move beyond FFNNs is the following : why would aforeseen FFNNs only pass the information *forward*? It is in fact natural to think to one of the most universal neurological function of the brain of living being, that lacks FFNNs : memory. Recurrent neural networks and LSTMs are made to remedy to this lack, by trying to model memory.

3.1 Recurrent neural networks

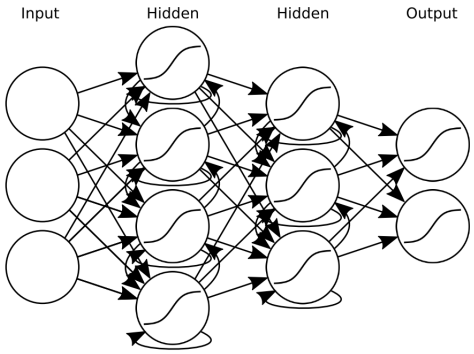
Recurrent neural networks (RNN) are very similar to FFNNs in their structure. The difference between RNN and FFNN lies in the fact that inputs become sequential, and the elements of the sequence are passed consecutively to the network. The network has now a 'time' dimension. When a sequence is successively fed to the RNN, the output of a neuron at layer l is not only passed to the neurons of next layer $l+1$, but will also be passed to the neurons of the layer l at next time step . This creates a recurrent connection, hence RNN's name. A schematic vision of this network can be seen in Figure 7.

This recurrence and this memory state are a major change between FFNNs and RNNs. It imposes to clarify this important time dimension asepect, by changing notations for RNNs. A sample input x is no more a 'simple input', but is now a 'time' sequence of length T, so that :

$$x = \{x^{(1)}, ..., x^{(T)}\}$$

This kind of sequential data could for instance correspond to a sequence of words in a sentence.

Figure 7: Schematic visualization of a 3-layer recurrent neural network, from [10]



Let's recall that with FFNNs, we denoted $z_i^{(l)}$ the output of the neuron i of layer l, and we had :

$$z_i^{(l)} = f\left(\sum_{k=1}^{n_{l-1}} w_{k,i}^{(l)} z_k^{(l-1)}\right)$$

Now, the output of the neuron i of layer l must be taken at a time t, thus we denote it now $z_i^{(l,t)}$ to show the time dependence. Let $v_{i,k}^{(l)}$ be the weight for the connection from neuron $N_i^{(l)}$

to neuron $N_k^{(l)}$. It must be specified that weights (forward or recurrent) have indeed **no** time dependence. The output equation is now :

$$z_i^{(l,t)} = f\left(\sum_{k=1}^{n_{l-1}} w_{k,i}^{(l)} z_k^{(l-1,t)} + \sum_{k'=1}^{n_l} v_{k',i}^{(l)} z_{k'}^{(l,t-1)}\right) \tag{9}$$

This time form with the last recurrent sum term brings a new issue : is it still possible to compute the gradient for optimization? Trying to compute the new gradient $\frac{\partial \mathcal{L}(x,y)}{\partial w_{i,j}^{(l)}}$ leads to a computable expression that enables to optimize the weights with respect to the loss function. This process is called the backpropagation through time (BBTT). As it is not our core subject here, we refer the interested reader to this note [3].

Still, even if the gradient is computable, it has some problems, called the problem of the vanishing (or exploding gradient). Bengio et al. [1] first identified this problem in 1994, and it has since been further explored by Pascanu, Bengio and Mikolov [11]. In order to tackle this issue, a full redefinition of memory has been given by Hochreiter et al. [4], leading to the invention of LSTMs.

3.2 Simulating Long and Short Term Memory with LTSM Cells

Instead of having a neuron which receives recurrent informations from its own layer, as in RNNs, Hochreiter et al. [4] create a complex computation cell, which is going to replace the neuron. As with RNN, we have a sequential data :

$$x = \{x_1, ..., x_T\}$$

A given LSTM cell A 'evolves' with the time, and is caracterized in time by a *hidden state* h_t and a *memory* c_t. We denote $A^{(t)}$ the state of cell A at time t. $A^{(t)}$ has three inputs :

- h_{t-1}, the hidden state transmitted by the previous cell state $A^{(t-1)}$.
- c_{t-1}, the memory transmitted by the previous cell state $A^{(t-1)}$.
- x_t, the 'real' sample input

And two outputs :

- h_t, the updated hidden state transmitted to $A^{(t+1)}$.
- c_t, the updated memory transmitted to $A^{(t+1)}$.

These inputs and outputs are schematized in Figure 8 [2]. The core question of LSTMs is to define update rules for h_t and c_t. First, we want to know how to update memory, or in other words, which informations to forget and which informations to remember. To do so, we define :

- $i_t = \sigma\left(x_t U^i + h_{t-1} X^i\right) \in [0, 1]$, where U^i, W^i are weights to learn. This function is called the *input gate*. The input gate is a weighted sum of the sample data at time t and the previous hidden state of the cell, regularized by the sigmoid function. The inpute gate decides, what new information we are going to store in the updated memory, and in which proportion, depending on the precedent data.

[2] The figure is extracted from Christopher Olah's (Google Brain) excellent blog : http://colah.github.io/

- $\tilde{c}_t = \tanh\left(x_t U^c + h_{t-1} X^c\right) \in [-1, 1]$, where U^c, W^c are weights to learn. $\tilde{c}_t$ is a candidate for creating 'new c_t values'.
- $f_t = \sigma\left(x_t U^f + h_{t-1} X^f\right) \in [0, 1]$, where U^f, W^f are weights to learn. This is called the *forget gate*. The forget gate decides the proportion at which each element of the memory is going to be forgotten.

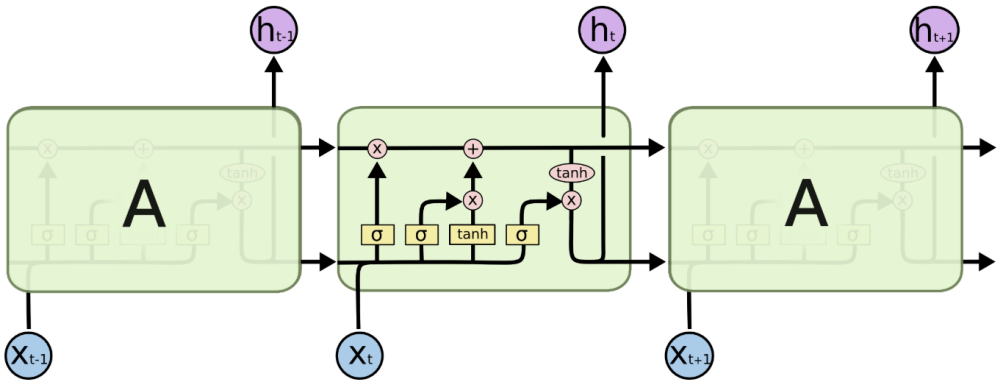
The memory can then be updated, with respect to the following rule:

$$c_t = f_t * c_{t-1} + i_t * \tilde{c}_t$$

Finally, to update the hidden state, we define the *output gate* in the same fashion, and we update h_t :

$$\begin{aligned} o_t &= \sigma\left(x_t U^o + h_{t-1} X^o\right) && \in [0, 1] \\ h_t &= \tanh(c_t) * o_t && \in [-1, 1] \end{aligned}$$

Figure 8: A representation of a LSTM cell in time



Some variants of LSTM cells may define input, output and forget gates differently. For instance, the peephole version of LSTM implements the natural idea that the previous memory state c_{t-1} has to be considered when deciding the input, output and forget rates. GRU, Gated Recurrent Units are other variants of LSTMs.

One question remains : what exactly is the hidden state h? h can directly be the output to learn : in our NER task, for instance, a sample sentence of length T would be the sequence of words $x = \{x_1\ , ..., x_T\ \}$. We want to assing a tag h_t to each word x_t of our sentence. In this case, h is the output. But nothing prevents us from putting another layer, after h. For instance, we could chain two LSTMs by deciding that the hidden state h_t of the first LSTM is, for each t, the input of the second LSTM.

Appart from having a very flexible and modular structure, LSTMs cells do not suffer of the same vanishing gradient problem as RNNs in practice, and are thus more effective in modelling sequential data. They are particularly adapted to NLP problems because of their sequential nature. We will see in the next section one of their most well-known implementation.

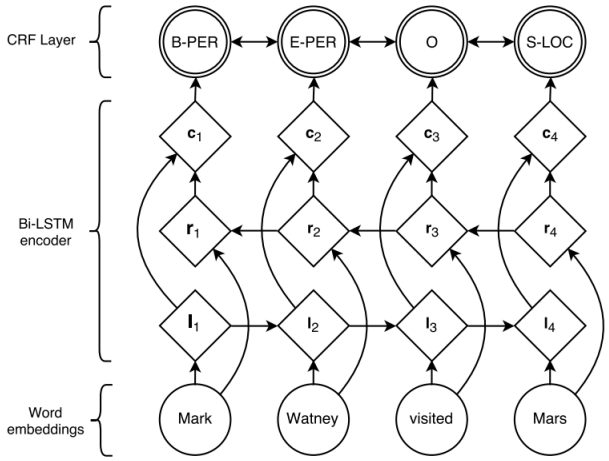
4 NER architecture

In this section, we describe Lample et al. (2016) [6] NER neural system. The task of a named entity recognition (NER) system is to find the entities in the sentence, and categorize them. For instance, in the sentence 'John is a computer scientist living in Paris', the NER system has to find is the two entities 'John' and 'Paris', and categorize them respectively as a person and as a location. The task can seem easy : if a word's first letter is uppercased, the word is a proper noun and thus is an entity. In facts, it is rarely the case. Groups, organizations names like 'United Nations' or 'World Health Organization' are groups of tokens that are difficult to recognize automatically. Thus, we define a little bit more precisely the task of the NER. If we decide to use k entities categories called $C_1, ..., C_k$, then a NER system is a classifier assigning to each word one of the following classes : $\{\text{B-}C_1, ..., \text{B-}C_k, \text{I-}C_1, ..., \text{I-}C_k, \text{O}\}$. This format is called the IOB format. A B preceding the category C_i means that the word is the beginning of an entity of category C_i. A I preceding an entity category means that the word is 'in' the entity, and the O tag corresponds to the 'outside' tag, which means that the word doesn't belong to any entity.

4.1 General architecture

In Lample et al. (2016) [6], the model lays on chaining two LSTMs with a conditionnal random field (CRF). The Figure 9 illustrates the architecture of the neural network.

Figure 9: General scheme of the NER model, from Lample et al. (2016) [6]



First, words are embedded with two different techniques : a Word2Vec embedding and a Character-based representation of the word. We defer to Section 4.3 the definition of the second embedding. Theses embeddings are concatenated, and the embedded sentences are passed to a Bi-LSTM encoder. A Bi-LSTM encoder consists of two independant LSTMs (i.e. they are not chained as described in Section 3.2). The first LSTM reads the words of the sentence in the right order (from left to right), and the second does the inverse (reads the words from right to left). This way, each embedded word x_t is associated by LSTMs to respective outputs h_t^L, h_t^R. These outputs are representations of the left and right context of x_t. They are concatenated ($[h_t^L, h_t^R]$), and then

fed to a CRF layer. The definition of the CRF layer is deferred to Section 4.2.

The most significant ideas here are first the use of the Bi-LSTM encoder, reading in both directions. This idea kind of imitates the 'global' reading of a human, when the eyes skim upon a text for instance. Second idea is the use of the conditional random field rather than an output layer which supposes that tags are independant from their context tags.

Unliked rule-based NER, or machine learning classifiers, this NER requires few training data, and does not require any feature construction, selection, etc. The algorithm has a sparing use of resources.

4.2 Conditionnal Random Fields

In NER classification, order counts. The location of the words in the sentence, their context, and the tags of the context words is likely to impact their NER tag. For instance, a tag `I-ORG` must be preceded by a tag `B-ORG`, a tag `I-ORG` cannot follow a `B-PER`, etc. Thus, we cannot use the raw outputs $[h_t^L, h_t^R]$ to decide, with a simple softmax for instance, the tag class of an embedded word x_t, as this implicitely assumes that the tagging decision for x_t is independent of the tagging decisions for $x_{t-1}, x_{t-2}, ...$

To model this context dependency, Lample et al. 2016 [6] procede in the following way. Let again be $x = \{x_1, ..., x_T\}$ our sentence of lenght T. The number of tag classes will be again denoted k. Let $h_t^i \in \mathbb{R}^d$ be the hidden state of LSTM $i \in \{L, R\}$ at time t. The dimension d of h_t^i is left as a parameter.
As our aim is to classify the words in k classes, it is natural to assume that the ouput of the architecture must be of dimension k, reflecting the probabilities for a word to belong to the k classes. To build such an output, we begin by adding a simple projection layer after Bi-LSTM, to 'reformat' them. Thus, Bi-LSTM outputs $[h_t^L, h_t^R]$ are transformed by the projection layer to an output $p_t \in \mathbb{R}^k$. Let's denote P the matrix of size $T \times k$ containing all coordinates of the Bi-LSTM at time t :

$$P := (p_{t,j})_{t=1,...,T, j=1,...k}$$

Let finally the sequence of correct known NER tags be $y = \{y_1, ..., y_n\}$. We consider the following score for a sequence :

$$s(x, y) = \sum_{t=0}^{T} A_{y_t, y_{t+1}} + \sum_{t=1}^{T} P_{t, y_t}$$

Where A is a square matrix of size $k + 2$, and A_{k_1,k_2} represents the transition score for passing from tag k_1 to tag k_2. This matrix is not known in advance, and will be learned during the training.

Finally, in order to have a readable output, i.e. a probability for x_t to belong to a class k_i, we compute a softmax over all possible tag sequences:

$$p(y|x) = \frac{e^{s(x,y)}}{\sum_{\tilde{y} \in Y_X} e^{s(x,\tilde{y})}}$$

Where Y_X is the set of all possible tag sequences (with no rule, so it includes sequences that do not verify the IOB format). We finally maximize $\log p(x|y)$ to train the network. To predict a

sequence after the network has been trained, we choose the sequence $y* = \arg\max_{\tilde{y} \in Y_X} s(x, \tilde{y})$ that can be obtained with dynamic programming.

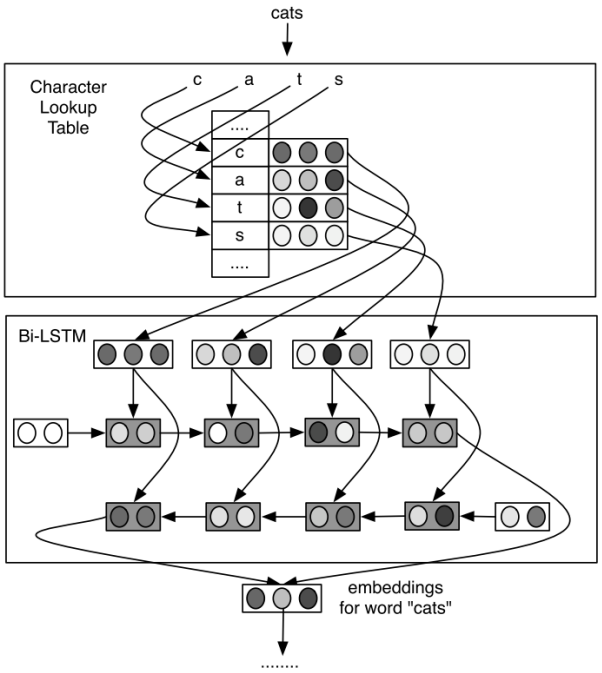
4.3 Character-level embedding

A last question about the Bi-LSTM model remains. How do we represent words? What is the real input of our network? We can't feed character strings into the network. We have seen in Section 2 that we have a first embedding, based on the distributional hypothesis, Word2Vec. However, it can be thought that a word is not only characterized by its company, but also by its form. As words which have ressembling forms generally derive from the same family, they tend to have close meanings. This is the main point of Ling et al. 2015 [7], in which authors are interested in the way a character sequence characterize a word.

This is in fact justified by an historic concept in linguistics, due to Ferdinand de Saussure who draws the distinction between *signified and signifier*. In linguistics, the *signified* is the concept, the idea behind the word, and the *signifier* is the way the word is represented (symbols, sounds). Word2Vec and other distributional embeddings thus focus on the *signified*, and character-level representation focuses on the *signifier*. Far from being opposed, these two points of view complement each other, and it is very common to use a combination of both embeddings.

In order to create word embeddings based on their form, Ling et al. 2015 [7] propose to use Bi-LSTMs. The inputs are the sequences of the characters of a word, and the outputs are the word embeddings. The architecture of the model, named 'C2W' by its authors, is showed in Figure 10.

Figure 10: C2W model architecture, from Ling et al. [7]



We assume that we have a word $x = \{c_1, ..., c_T\}$ where c_i is the i-th character of the word. Characters are first encoded as one-hot vectors of dimension C, which denotes the number of characters in the vocabulary. They are passed to a projection layer P, in order to become vectors e_c (embedding of character c) of size d_C, which is a parameter that can be chosen. This projection layer is made to "capture similarities" between characters (for instance, vowels, consonants). The characters embeddings e_{c_t} are then sequentially passed to the Bi-LSTM, which has an hidden state of dimension d_H (also a parameter). We only keep the final left and right outputs of the Bi-LSTM, namely h_T^L, h_0^R. These outputs are finally summed with weights (to learn) to give the word embedding. By denoting e_w the word character-level embedding, we finally have:

$$e_w = W^L h_T^L + W^R h_0^R v + b$$

Where b is a bias.

The main question in this model is : how can we train the network? Finding word embeddings is by definition an unsupervised task, so we don't have at our disposition embedding samples to learn from. A good idea is to chain this embedding to layers/full models which do other supervised tasks. Embeddings are thus learned while, for instance, classifying words, etc. In our NER example, embeddings weights are learnt while learning the whole NER model.

Conclusion

In this report, we described an end-to-end NER model with Deep Learning. To do so, words had first to be transformed into computer-intelligible data, namely numbers. The embeddings were learned in two ways : a distributional way, Word2Vec, underlayed by the hypothesis that words with same meanings occur in the same context, focussing on the signified rather than the signifer. The other way to learn the embeddings, C2W, has been focussing on word forms, on signifiers, in order to complete the first signified analysis. To do so, neural networks emulating long and short term memory, LSTMs, have been used. As they favor sequential data such as sentences, they have been of great help for the subsequent part of the task : NER classification. However, as NER tags cannot be assigned independently from their predecessors, we had to use a Conditional Random Field to finally classify our words.

Advanced models have been developped since Lample's NER in 2016. They generally rely on attention mechanisms, and have rated state-of-the art performances on NER tasks.

References

[1] Y. Bengio, P. Simard, and P. Frasconi. Learning long-term dependencies with gradient descent is difficult. IEEE Transactions on Neural Networks, page 157–166, 1994.

[2] Yoav Goldberg and Omer Levy. word2vec Explained: deriving Mikolov et al.'s negative-sampling word-embedding method. (2):1–5, 2014.

[3] Jiang Guo. BackPropagation Through Time. Manuscript, (1):1–6, 2013.

[4] Sepp Hochreiter and JÃŒrgen Schmidhuber. Long short-term memory. Neural Computation, 9(8):1735–1780, 1997.

[5] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. Neural Networks, 2(5):359–366, 1989.

[6] Guillaume Lample, Miguel Ballesteros, Sandeep Subramanian, Kazuya Kawakami, and Chris Dyer. Neural Architectures for Named Entity Recognition. 2016.

[7] Wang Ling, Tiago Luís, Luís Marujo, Ramón Fernandez Astudillo, Silvio Amir, Chris Dyer, Alan W. Black, and Isabel Trancoso. Finding Function in Form: Compositional Character Models for Open Vocabulary Word Representation. (September):1520–1530, 2015.

[8] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient Estimation of Word Representations in Vector Space. pages 1–12, 2013.

[9] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed Representations of Words and Phrases and their Compositionality. In C J C Burges, L Bottou, M Welling, Z Ghahramani, and K Q Weinberger, editors, Advances in Neural Information Processing Systems 26, pages 3111–3119. Curran Associates, Inc., 2013.

[10] Lasse Regin Nielsen. Language modelling using deep learning Generating answers to medical questions using recurrent neural networks. 2017.

[11] Ravzan Pascanu, Tomas Mikolov, and Yoshua Bengio. On the difficulty training of recurrent neural networks. 2013.

[12] Frank Rosenblatt. The perceptron, a perceiving and recognizing automaton. 1957.