ML Theory Journal

Aritz Bercher

February 5, 2021

Abstract

In this journal, I will try to gather good references (mainly webpages) about some Machine Learning (and in particular Deep Learning and Natural Language Processing) algorithms as well as some personal thoughts on the subject. The aim of this journal is to keep gather in a same place the *theory* of Machine Learning

Contents

1 TODO

1.

2 Useful Resources

- I found this **blog** by Tobias Sterbak, which seem to cover a lot of **advanced topic in machine learning/deep learning, natural language processing, and computer vision** with both theory and implementation aspects treated: `https://www.depends-on-the-definition.com/classify-toxic-comments-on-wikipedia/`
- There is this **coursera mook about NLP**: `https://www.coursera.org/learn/language-processing`

- There is this **repo** from Sebastian Ruder which keeps track of the **best models for many NLP tasks**:
  `https://github.com/sebastianruder/NLP-progress`
- There is this **review** by two guys from Microsoft and one from Google about the **state of the art in Conversational AI** (as of 13.12.18):
  `https://arxiv.org/pdf/1809.08267.pdf`
- (17.09.19) I found this blog which provides both **theoretical explanations and implementation tutorials for NLP models**:
  `https://mlexplained.com/2019/`
- (17.09.19) Here is a useful website presenting **papers with their implementation**:
  `https://paperswithcode.com/`
- (09.10.19) **Latest arxiv publications** in specific domains, like for instance:
  `https://arxiv.org/list/cs.CL/recent`
- Machine Learning Explained: on this website, some important papers are explained in details, and implementation are presented. The focus is on Deep Learning and NLP. For instance XLNet is presented and implemented:
  `https://mlexplained.com/`
- This blog gives some excellent explanations of how complex models like transformers, BERT, GPT3 are working with very good **illustrations**:
  `http://jalammar.github.io/`

3 Specific technical questions which should be answered

1. (17.09.19) I should understand exactly how BERT can generate an answer. This paper touches the topic (but not in detail):
   `https://arxiv.org/pdf/1906.05416.pdf`

4 Journal

04.11.17 ~ Principal Component Analysis (PCA) and Singular Value Decomposition

PCA comes again and again, so I think I should try to learn it properly. I quickly reviewed what I had already seen in the first lecture of CIL. This being said I'm not use that it is exactly the PCA, since it's called SVD (Singular Value Decomposition). I read this very interesting page which gives a good explanation of the two different quantities and their relations:
Stack Exchange: PCA and SVD

21.02.18 ∼ **Skip-gram model for words embedding**
I read these two pages of a same tutorial on the **Skip-gram** model:
http://mccormickml.com/2016/04/19/word2vec-tutorial-the-skip-gram-model/
http://mccormickml.com/2017/01/11/word2vec-tutorial-part-2-negative-sampling/
Edit (27.07.18): The skip-gram model is also presented in the coursera mook on NLP, in a quite different way.
Edit (21.10.18): Basically, the skip-gram model does the following: it builds a neural network to predict for a given word, its context. The input is a one-hot encoded vector representing a word and the output is a vector of probability of size equal to the size of the vocabulary (let's call it n) indicating the probability of having any word as neighbor. The embeddings are obtained by taking the intermediary layer (which has no activation function and can be seen as matrix of size $n \times 300$ (if the embedding dimension is 300)).
Edit (19.01.20): This is the technique used by **Word2Vec**.
Edit (05.05.20): This article explains how to use Word2Vec to **build a recommendation System** and provides a detailed python implementation of it:
https://www.analyticsvidhya.com/blog/2019/07/how-to-build-recommendation-system-word2vec

14.03.18 ∼ **Recurrent Neural Networks (RNN), Long Short Term Memory (LSTM)**
I read this introduction to **Recurrent Neural Networks in NLP**, and their applications to **Language Modeling and Generating Text**, **Machine Translation**, **Speech Recognition**, **Generating Image Descriptions** (when coupled with some CNN):
http://www.wildml.com/2015/09/recurrent-neural-networks-tutorial-part-1-introduction-to-
I also read this page on **Long Short Term Memory (LSTM)** networks:
https://colah.github.io/posts/2015-08-Understanding-LSTMs/
which is a very successful special case of RNN.

17.03.18 ∼ **Bag of words model**
I read an introduction to the **Bag of word model (BOW)** here:
https://machinelearningmastery.com/gentle-introduction-bag-words-model/
I also read the beginning of the "Example: Logistic Regression Bag-of-Words classifier" here:
http://pytorch.org/tutorials/beginner/nlp/deep_learning_tutorial.html

21.03.18 ∼ **Character embedding vs Word embedding**
I read this post on Quora which compares word embedding and **character embedding**:
Quora: How does character embedding work in comparison to word embedding?

22.03.18 ∼ **More on character embedding, GloVe**
I read the beginning of this **article on character embedding**:

`http://minimaxir.com/2017/04/char-embeddings/`
but I didn't find it so good when it comes to explaining what character embedding is. But the little implementation with Keras seems like a good exercise. Maybe I could reproduce something like this with the dataset of fast ai.
I read this page about *GloVe*:
`https://nlp.stanford.edu/projects/glove/`
I think it is just a library doing this word embedding. It looks like the fundamental ideas are the same as for Word2Vec, i.e. train a model to predict co-occurrence of words (and then use the weights obtained as your embeddings).
Edit (19.01.2020): Actually even if the idea is a bit the same, and in both cases we build word vectors using co-occurances, the technique is different. In Word2Vec one uses a neural network to predict one word randomly selected in a window around a given word, with a neural network having only one hidden layer. The hidden layer is used as the embedding matrix. In GloVe one uses a matrix of co-occurence of the words in the corpus and use matrix factorization and dimensionality reduction techniques to produce embeddings. If I got it right, let say that we have

$$M = A\Sigma B$$

and can approximate it keeping only the first 300 features in the diagonal of Σ, by taking

$$M' = A'\Sigma' B'$$

then we can use the A' as our embedding matrix.
For GloVe, this page seems to give a nice explanation (but I haven't read it yet):
`https://blog.acolyer.org/2016/04/22/glove-global-vectors-for-word-representation/`
The I read this page:
analyticsvidhya.com: Word representations-text classification using Fasttext (an NLP library from facebook)
The introduction explains quite well what **character embedding** is and what are its advantages compared to classical (i.e. Word2Vec already mentioned above) word embedding techniques. Then it dives into how to use this library. Maybe I should try to use it on the data set from fastai.

23.03.18 ∼ Continuous Bag of Words
I learned a bit about the **continuous bag of word** model is used which differs from the skip-gram model (presented in this page that I had already read) as explained in this page from Quora:
Quora: What are the continous bag of words and skip-gram architectures?

02.04.18 ∼ Constituency parser, Conditional Random Fields (CRF), Viterbi algorithm
I learned a bit about **constituency parser** here:

Stackoverflow: Difference between constituency parser and dependency parser
I read this introduction to **Conditional Random Fields (CRF)**:
`http://blog.echen.me/2012/01/03/introduction-to-conditional-random-fields/`
But this page doesn't mention any neural network.
Edit (28.10.19): The first thing to understand is that the goal is to assign a probability to a sequence of tags for a sequence of items (typically a sequence of words), and its strength comes from the fact that it tries to compute the **joined** probability of these items (in the specific given order).
I read the description of the **Viterbi algorithm** on wikipedia:
Wiki: Viterbi algorithm
From what I understood, it is just a kind of "dynamic programming" algorithm where instead of computing the probability of every possible sequence, we first find for each possible stae the most likely sequence of length 2 finishing by this state, then use it in order to find the most likely sequence of length 3 finishing by every possible state and so on.
Edit (30.04.18): In this paper it is explained (unfortunately not in a detail fashion) how to combine a (Bi)-LSTM network with a CRF:
`https://arxiv.org/pdf/1508.01991.pdf`
I also read this page which gives **an intuition of the Viterbi algorithm**:
Quora: What is an intuitive explanation of the Viterbi algorithm?
Edit (02.05.18): I think I understood how one can mix the LSTMs and the CRFs. It is kind of explained in this tutorial:
`https://pytorch.org/tutorials/beginner/nlp/advanced_tutorial.html`
Here are the main points: let say we want to use LSTMs without anything else to predict what should be the labels/tags of each word in a sentence. For each input (each word) the LSTM would output a distribution over the possible labels, and we would take its argmax as the predicted label (for a given word). Let say the words are $x_1, x_2, \dots$ and the labels can be taken from a set $S = \{s_1, \dots, s_N\}$. Now, we can see each entry of this vector of probability (of length N) as one of the weighted "feature function" of this explanation of CRF. For an word/input x_i, entry 2 of the "hidden state" (I don't really like this terminology because with LSTM, we never know if we are talking of h_i or C_i (following the terminology introduced in this tutorial) but here I mean h_i) outputed by the LSTM is seen as a feature function giving the likelihood of seeing x_i if the tag/label is s_2. Then we add some "transition scores" which are additional feature functions independent from the LSTM and which allows to use the power of usual CRF models.
Edit (03.05.18): Actually, the idea goes beyond superposing the two processes. As I realized in the pytorch tutorial mentioned above, the transition scores/probabilities, are themselves some tensor which are optimized over, when doing backpropagation.
Edit (12.10.18): I guess that both the hidden states produced by the lstm and the transition functions/scores/probabilities can be seen as feature functions, but (and this is where my guess is) the hidden states make use of the information regarding the position in the sentence, whereas the transition functions look at the neighbouring words of the given word. But since the transition functions are also learned, I guess that these

special "CRF" are merely an LSTM with an additional layer of weights on top of it.
Exit (22.11.2020): This paper explains one way of using BiLSTM to build a CRF:
`https://arxiv.org/pdf/1603.01360.pdf`
(Neural Architectures for Named Entity Recognition)

30.04.18 ∼ Chat bots: Tai, Xiaoice, Named entity recognition, A nice blog about NLP
It seems that some fairly advanced chatbots already exist:
Wiki: Xiaoice
Wiki: Tay
I learned a bit about **Named Entity Recognition (NER)** here:
Wiki: Named Entity Recognition
I found this **nice blog (mainly) about NLP**:
`https://www.depends-on-the-definition.com/about/`

01.05.18 ∼ TF-IBF, and Latent-Dirichelet-Allocation
Roman mentioned **tf-idf**, **Latent-Dirichelet-Allocation**, HMM, as a classical alternative to NN models, which should be used as benchmarks to measure the success of the NN models.
From the wikipédia page for tf-idf:
Wiki: tf-idf
this tf-idf seems to be used in order to find among a given set of documents, which ones are relevant for a specific query ("the brown cow") for instance. It gives a score to each document.
Edit (25.05.18): Actually TF-IDF is quite well explained in the first week of the NLP coursera course. It's simply a way to improve a bag-of-word model, where instead of entering just 0 or 1 in each column to indicate if a word is present or not, one puts a score which indicates both how much frequent the word is in the given document/sample, how rare the word is in the total corpus of documents/samples.

The **Latent Dirichelet Allocation** is a bit more complicated. From the wikipedia page:
"In natural language processing, latent Dirichlet allocation (LDA) is a generative statistical model that allows sets of observations to be explained by unobserved groups that explain why some parts of the data are similar. For example, if observations are words collected into documents, it posits that each document is a mixture of a small number of topics and that each word's creation is attributable to one of the document's topics."
The math behind it is explained on the wikipedia page. From what I saw, it seems to be a bit similar to the bayesian nets studied in the course of Krauser in the sense that we make some (smart) assumptions on the distributions of the words in a corpus of documents, and then using algorithm to estimate the parameters of the distributions. But the wikipedia page is very long and technical. Maybe the coursera course on NLP

would be better.

06.05.18 ~ Statistical Parsing
I read this wiki article about **Statistical Parsing**:
Wiki: Statistical Parsing

25.05.18 ~ Useful metrics for binary and multi-label classification, ROC curve
In the notebook of the exercise of the first week, I (re)discovered the notion of **F1-score**:
Wiki: F1 score which is a useful metric for binary or multi-label classification.
I also came back on the notion of **ROC curve**. I was confuse at first because, I had the impression that if we were doing binary classification (I guess we can generalize this discussion easily to multi-label classification) and we have just the predicted class of a model for a given set of data, there is only one True Positive Rate (TPR) and one False Positive Rate (FPR) to compute. But actually, we don't use the predicted classes but the *scores* output by the model, i.e. for each sample the probability to belong to each one of the two classes. Then we can consider for each $c \in [0, 1]$, the samples for which the probability to be in the second class is above c, to be "positive" and the other samples to be "negative", and then compare to the true labels. And this way we can compute this ROC curve.
Edit (18.02.20): I read this page which explains how to generalize metrics such as **precision** or **recall** to **multi-label** classification:
`https://stackoverflow.com/questions/9004172/precision-recall-for-multiclass-multilabel-c`

29.05.18 ~ More about NLP
In the NLP coursera course, I was introduced to various models, some using Neural networks, some using classical algorithms.
In particular, I learned that the way to evaluate language models is based on the notion of **Perplexity** (c.f. journal NLP coursera for more infos).

30.05.18 ~ Dropouts for regularization of LSTMs
I read this article (I have it on my computer):
`https://arxiv.org/pdf/1409.2329.pdf`
explaining how one can use dropouts (i.e. randomly put some coefficients of the NN to 0 during the different training phases) to **avoid over-fitting**. In the case of RNN, one has to be careful not to affect the coefficients which manage the transmission of the memory.

31.05.18 ~ Evolutionary learning: an alternative to gradient descent
I read this article that Mike recommended to me:
`http://togelius.blogspot.com/2018/05/empiricism-and-limits-of-gradient.html`

It presents some alternative technique to gradient descent to come up with a model achieving good performance. The basic idea is to start with a bunch of different models, test them, throw away the one performing the worst, modify randomly the others and combine them, and then repeat the procedure. The advantages are that you don't need differentiability of the loss function, and according to the author it is more likely to learn "something which isn't in the data" like a mathematical formula. At the end of the article there is a list of recent paper on the topic by some AI labs like Uber AI, Sentient Technologies, DeepMind, and OpenAI.

01.06.18 ~ Gradient clipping
I learned about **Gradient Clipping** there:
hackernoon: Gradient clipping
It seems to come down to putting a bound on the value of the norm of the gradient to avoid NaN appearing. It seems to be useful for RNN:
Stack Overflow: Why do we clip by global norm to obtain gradients while performing RNN?

03.06.18 ~ Truncated Back Propagation Throw Time (TBPTT) for RNN
I read this article which explains some tricks used to change a bit the update scheme of the parameters when using RNN:
`https://machinelearningmastery.com/gentle-introduction-backpropagation-time/`
It is used in the RNN TF tutorial.

10.06.18 ~ permutation importance: a criterion to assess importance of features
I read this article about **permutation importance**:
`http://parrt.cs.usfca.edu/doc/rf-importance/index.html`
which explains that there is a better criterion that the "impurity decrease" called "permutation importance" to **judge how much a feature is useful**. This is particularly easy to use in the random forest case (because we have some OOB samples), but it can always be used if one has a validation set. I think it is implemented in scikit learn random forest estimator.

14.06.18 ~ Encoder-decoder

I read this Quora page about **encoder decoder Network**:
`https://www.quora.com/What-is-an-Encoder-Decoder-in-Deep-Learning`
because I wanted to understand something in the "lesson4-imdb" notebook of fastai (see entry of the 15.06.18 of the fastai journal, for details).
Edit (12.10.18): From what I understand an encoder-decoder is just a way to go from one initial representation which is the input of the encoder (for instance a sequence of bag of word vectors) to an intermediate one which is both the output of the encoder and

the input of the decoder (for instance a vector) and then to a third one (which is for instance a sequence of words). Autoencoders are just NN which are typically stacks fully connected layers which have to predict their (cathegorical) input. By keeping only the first layers, we build a useful way to represent the data in a low dimensional space (and we build this way an encoder). One can have a CNN as encoder and RNN as decoder as explained in the link above.

15.06.18 ~ Simple but efficient models for sentiment analysis, and more complex architectures
I read this article:
`https://www.aclweb.org/anthology/P12-2018`
(I have it on my computer) which shows that some simple methods can be used efficiently for sentiment analysis on texts. It provides comparison tables for different common data set. I discovered it in the "nlp" dataset from fastai.

Edit (14.01.20): I found this page which lists different kinds of Neural Network architecture (and some also non deep learning ones) used for sentiment analysis:
`https://blog.paralleldots.com/data-science/breakthrough-research-papers-and-models-for-s`

16.06.18 ~ Stop words
Stop words are words which are removed from a text before it is further processed. There are some lists of them (for instance in the "english" file of my NLTK package, or on this page) but they are not the same. Plus sometimes, it could actually harm the model to remove some common words (typically negations) which have a strong impact on the meaning of a sentence, if one wants to perform sentiment analysis. Some people suggest not to remove them at all:
Why sentiment words are in stopwords list?
Is there a stopword list specifically designed for sentiment analysis?

27.06.18 ~ Hyperparameter tuning for RNN models, measures against overfitting
There is an interesting section in this tutorial:
`https://www.oreilly.com/learning/perform-sentiment-analysis-with-lstms-using-tensorflow`
about how to **tune your hyperparameters** for a RNN.

In the same tutorial, the author explains how one should fight against overfitting: "The basic idea is that we train the model on our training set, while also measuring its performance on the test set every now and again. Once the test error stops its steady decrease and begins to increase instead, you'll know to stop training, since this is a sign that the network has begun to overfit."

14.07.18 ~ Convolutional Neural Network for text classification
I read this paper about the use of **CNN for text classification**:
`https://arxiv.org/abs/1412.1058`
There are two different ways to do it. Either do the convolution directly on texts converted to sequence of one-hot-encoded vectors representing the words, or something a bit more subtle where each local region where a convolutional kernel would be applied is encoded as a bag of words.

16.04.18 ~ Difference Valdiation and Test set
I read this page which clarified what was the distinction between the **test set and the validation set**:
Stack Exchange: What is the difference between test and validation set

06.08.18 ~ Multi-label and multiclass classification
I learned a bit about multi-label and multiclass classification:
`https://en.wikipedia.org/wiki/Multiclass_classification#One-vs.-rest`
`https://en.wikipedia.org/wiki/Multi-label_classification`
which are not exactly the same thing. But the approaches are similar. Interestingly, the canonical methods for each one (One-vs-rest and binary relevance methods) are implemented in the same python class in Scikit-Learn: sklearn.multiclass.OneVsRestClassifier.

21.08.18 ~ Tuning a model
In order to tune a deep learning model, here are the recommendations from Roman:
"I start with a model config that I expect to overfit and slowly reduce nn complexity until validation performance stops improving. As for non nn parameters as learning rate, I pick them after I settle on network topology, I also pick them manually with some sort of coordinate gradient descent (picking them on by one) or using the defaults if I think the parameter is not important".

This link also gives some instructions to tune an XGBoost model:
`https://machinelearningmastery.com/xgboost-python-mini-course/`

12.10.2018 ~ StarSpace

This paper explains how **StarSpace** is working:
`https://arxiv.org/abs/1709.03856`
I had encountered it already when doing duplicate detection in the NLP coursera course. It seems to be a way to build some custom embeddings. I found again a reference to it on this page explaining how the TensorFlow embedding pipeline of RasaNLU works.
Edit (08.10.19): I looked at the paper again. The main idea is that depending on the task, we define a set of (discrete) features (for text classification, the features would

be all words into a fixed English dictionary, and document labels), and then we train embedding for these features. Then we define "entities" as bag of features. A text to classify would have as features the words it contains, and the document labels would have only one feature each, the feature corresponding to the given labels. Then one trains the features embeddings using a special loss function based on similarity, and positive/negative examples. Later we can try for a new text to classify it by looking at the labels which have an embedding similar to the embedding of the new document (obtained by summing the embedings of the words it contains). One other application is to try to **predict relations in a graph**. I was thinking that it's kind of pointless since one can query the graph, but then I realized that it could be used to **guess relations which aren't explicitly encoded in the graph**, if the graph is big enough and contains some recurring structure.

Edit (11.08.2020): Nikos made me realize that using this similarity approach and choosing the class having the highest similarity with the embedding of the input is the same as passing the input through a last linear layer to reshape it to a vector having the length equal to the number of classes and selecting (with softmax for instance) the component having the highest value. It is simply a different way of seeing the same thing.

21.10.18 ~ Parallel SGD computation with Hogwild and others

I read the beginning of this page concerning **asynchronous stochastic gradient descent algorithm** performed with **Hogwild**:
`https://srome.github.io/Async-SGD-in-Python-Implementing-Hogwild!/`
which offers also some sample code in python to implement it. At the end of the article, the author points out to other methods.

01.11.18 ~ Neural Turing Machines

I read this page about **Neural Turing Machines (NTM)**:
`https://blog.acolyer.org/2016/03/09/neural-turing-machines/`
and even if I didn't understand every detail, it was interesting. One has to see how far these Machines can learn. The article focuses on some very basic tasks like copying. This topic was also mentioned in the course about Synthesis and other topics of ETH. This page:
`https://medium.com/snips-ai/ntm-lasagne-a-library-for-neural-turing-machines-in-lasagne-`
gives much more insight.

13.02.19 ~ Levenshtein distance

I watched this video introducing the **Levenshtein distance** which gives a **distance** between two finite **sequences**:
`https://stackabuse.com/levenshtein-distance-and-text-similarity-in-python/`

17.03.19 ~ Tree LSTM

I read the beginning of this article about **tree LSTM**:
`https://arxiv.org/pdf/1503.00075.pdf`
which are a bit like LSTM, but the memory/state cell associated to element/word i of the sentence is computed using the memory/state cell and hidden states of the elements/words which "children" of i ($C(i)$). The set $C(i)$ is computed using a tree which is also given as input to the model (typically a consitutency or dependency tree). This seems to be used in the paper that Aurélien used for his master thesis:
`https://arxiv.org/pdf/1609.06038.pdf`

23.03.19 ~ Capsule networks

This blog gives a good introduction to **capsule networks**:
`https://medium.com/ai%C2%B3-theory-practice-business/understanding-hintons-capsule-netwo`
Maybe I could use them to generate animated characters from static picture (front, side, back).

26.03.19 ~ Multi-Task Deep Neural Network for NLP

Rolland presented a paper called **Multi-Task Deep Neural Network for NLP**:
`https://arxiv.org/pdf/1901.11504.pdf`

31.05.19 ~ Sequence to sequence translation, Attention, Transformer, and Bert

I started reading this very well illustrated tutorials:

- **Neural Machine Translation Model (Mechanics of Seq2seq Models With Attention):** `https://jalammar.github.io/visualizing-neural-machine-translation-mechan`
- **Transformer:** `http://jalammar.github.io/illustrated-transformer/`
- **BERT:** `http://jalammar.github.io/illustrated-bert/`

Since there is a lot to say abou each model, I will split this entry into sections that I will complete little by little.

4.1 Neural Machine Translation Model (Mechanics of Seq2seq Models With Attention)

- In the sequence to sequence translation with attention model, the input is a sequence (of words in English for instance) and the output is another sequence (of words in French for instance), not necessarily of the same length.
- In the tutorial about sequence to sequence translation with attention model, in the illustration called Neural Machine Translation, the pink output of the Dense layer after the first unit of the RNN (1) is not exactly the same as the input of the second unit of the RNN (2) even if they are depicted in the same way. Most likely (1) is a vector of the same dimension as the encoder hidden space, whereas (2) is something like the sum of the scalar product of (1) with every encoder hidden space (it must be something like this as the number of such encoder hidden space vectors will depend on the input).
- In the tutorial about sequence to sequence translation with attention model, I guess that the attention coefficients are probably computed in a similar way as in the Transformer tutorial.
- **Edit (06.08.19):** I found this other page which explains with more details the attention mechanism:
`https://distill.pub/2016/augmented-rnns/#attentional-interfaces`

4.2 The Transformer

- In the tutorial about the Transformer, the **input** is a sequence and the **output** is a sequence (like in the sequence to sequence translation with attention tutorial). The task presented was **translation** from French to English.
- Unlike in the attention tutorial, the model of the Transformer, doesn't seem to use LSTMs. Instead the encoder layers receive a list of vectors of fixed length (this is a hyper parameter and I guess that padding is used) containing all the input as once, and outputs a list of vectors containing all its output at once. It has a component called **self-attention layer**, which doesn't seem (a priori) to process input with an order. The function which takes this list as input takes the whole list, i.e. it is not repeatedly applied to every vector in the list individually. Indeed this self-attention mechanism uses every element of the input list to compute each element of the output list. In order to help the model better learning to make use of the relative positioning of the elements of the input sequence, each initial input word/object vector is concatenating with another vector encoding only the position of the word/object.
- Looking at this Transformer encoder, one thing which wasn't clear to me at first was how we can take a sentence of length n and output a sentence of size different from n. When we use an LSTM like in the sequence to sequence translation with

attention model, it is clear since we can let run the decoder as long as it hasn't predicted the "stop" character (which is added to the end of each sentence of the training data). In the case of the Transformer, I guess that **a sequence cannot be longer than the predefined fixed size** of the list of vector given as input to the self-attention layer, but that discarding the elements of the output list which will be predicted as padding elements, we can have shorter sentences.

- I think that the reason why some layers are considered to be part of the Encoder and some of the Decoder is because the one in the Decoder only receive the output of the last layer of the Encoder.

- Both the Encoder and the Decoder receive as initial inputs, (non-contextual) embeddings of the words of the input (in the example of the tutorial, it could be GloVe embeddings for French). The Decoder receives additionally as input the output of the Encoder.

- In the Transfomer model, the Decoder had an architecture different from the Encoder. At the **decoder** level, there are **two attention mechanism** used: the first one is called **(masked) self-attention layer** where "all of the keys, values and queries come from the same place, in this case, the output of the previous layer in the (decoder)" (p.5 of "Attention is all you need"), and the second one **encoder-decoder attention layer**, where "the queries come from the previous decoder layer, and the memory keys and values come from the output of the encoder" (p.5 of "Attention is all you need"). For the encoder-decoder attention layer, I guess that the memory key, value, and query matrices are specific to the given layer (not shared with any other) but that the vectors used to generate the memory key and value vectors are the output of the encoder, whereas the vectors used to generate the query vectors are the output of the previous decoder layer. For the self-attention layer, they explain that the attention at position i is restricted to positions 1 to $i-1$ (to prevent the mechanism to simply learn to copy rather than predict). From what understand, if I use the notation of the tutorial $q_m * k_n$ is set to $-\infty$ if $n > m$. This implies than the the weight for this position is 0 after we take the soft-max. My question is: is that differentiable? But for the encoder-decoder attention layer, in order to compute the new output vector at position i, the output vectors of the previous layer for all positions can be used. I understand that if we train a model to do language modeling (guessing what is word at position i looking only at the outputs until position $i-1$), where we try to rebuild the input sentence, the model could otherwise simply learn to copy the input embeddings. But if we try to translate English to French, the reason is less obvious. The authors of the paper justify it by stating: "We need to prevent leftwardinformation flow in the decoder to preserve the auto-regressive property." I guess that it somehow forces the model to make smart embeddings.

- In the paper , there is a very intresting remark about **long-term dependency**: "The third is the **path length between long-range dependencies in the**

network. Learning long-range dependencies is a key challenge in many sequence transduction tasks. One key factor affecting the ability to learn such dependencies is the length of the paths forward and backward signals have to traverse in the network. The shorter these paths between any combination of positions in the input and output sequences, the easier it is to learn long-range dependencies. Hence we also compare the maximum path length between any two input and output positions in networks composed of the different layer types."

- The Transformer model presented in the tutorial mentioned above appeared in the paper *Attention is all you need* here:
`https://arxiv.org/pdf/1706.03762.pdf`
where there is no language modeling task and where the goal is to do English-to-French translation, but the idea of using it as a basis for leveraging an **unsupervised-learning task (language modeling)** and then fine-tuning it for other specific tasks appear in a paper called *Improving language understanding by Generative pre-training* (**GPT**) from OpenAI, here:
`https://s3-us-west-2.amazonaws.com/openai-assets/research-covers/language-unsupervi`
`language_understanding_paper.pdf`
They actually explain in this paper, that their approach (training in an unsupervised manner a model for language modeling and fine-tuning in a supervised manner for a specific NLP task) is very close to ULMFit presented by Howard and Ruder, here:
`https://arxiv.org/pdf/1801.06146.pdf`
with the difference that they use the Transformer architecture instead of the LSTM. In the *Improving language understanding by Generative pre-training* article, the authors explain that they didn't use the original architecture of the transformer but a modified version called `Transformer Decoder` presented in the paper *Generating Wikipedia by summarizing long sequences* here: `https://arxiv.org/pdf/1801.10198.pdf`
(c.f. p.5). They explain there that they use only the decoder part. This makes me wonder what happens with the encoder-decoder attention sub-layer. Where are the queries computed from? Do we use the vectors of the previous decoder layer? Or do we remove the encoder-decoder attention sub-layers? If we remove them, why do we say that this is the decoder part and not the encoder part? Is it because we keep the mask thing (which is actually modified in this paper *Generating Wikipedia by summarizing long sequences* and replaced by two alternatives)? But coming back to using the transformer to get good embeddings, I don't understand why we would keep the masks if we want to generate embeddings. On the opposite, we would like that every token gets enriched by the whole context. Personally I would train the encoder together with the decoder, and then use only the encoder... It isn't clear at all in the *Improving language understanding by Generative pre-training* article where the explanation is very shallow. I guess I should look at the implementation. I could look at it on the website paper with code:
`https://github.com/openai/finetune-transformer-lm`

but unfortunately it is in tensorflow which is a pain in the ass.
Edit (25.09.19): Joram explained me that actually, what is used is indeed just the decoder part but the layers don't have the encoder-decoder attention sublayers. They keep the mask because it forces the model to have the "auto-regressive" property, so in a sense, to predict one word after the other.
Exit (03.02.2021): This page about **GPT-2**:
`http://jalammar.github.io/illustrated-gpt2/`
confirms what I suspected above: in the *Improving language understanding by Generative pre-training* article, only the decoder part of the transformer is used and the encoder-decoder attention sub-layer is removed.

- In the paper *Improving Language Understanding by Generative Pre-Training* (**GPT**), they present the idea of pretraining a transformer-like model on language modeling. I guess that for an input sequence there is an output sequence where output with label k is the predicted $k+1$ input token, and since there are mask layers, input $k+1$ and following cannot be used to compute the value of output k. But all tokens are predicted at the same time.

- From what I understand, unlike in ELMo, only the output of the last transfromer layer is used as input for the downstream task.

- One of the strength of the Transformer, is that the predictions for the different masks tokens can be done in **parallel**.

4.3 ELMo

ELMo is one of the ancestors of BERT. From what I understood, BERT is a kind of ELMo where the LSTM has been replaced by a Transformer. The paper is here:
`https://arxiv.org/pdf/1802.05365.pdf`
Edit (10.10.19): I read it again and the idea is simple: create contextual embeddings for each input token of a sentence by taking linear combination of the output of the different layers of a multi-layer bi-lstm pretrained on language modeling task.
The language modeling task goes like this: there are two LSTMs, a forward one and a backward one. In order to train the model to produce a vector embedding for token i, the first $i-1$ tokens are passed recursively to the L-layers LSTM, and parallely, tokens $i+1$, $i+2$, ..., n are passed recursively (starting by the last) to the L-layer backward LSTM. The output vecors of both are then fed to a feed forward network with last layer having output size equal to the size of the dictionary. Finally a soft-max layer is applied, and Cross-Entropy loss is used as a loss function to determine if the prediction was good or not.
The input is a sentence of token and the output is a sequence of vectors (embeddings). Actually all intermediary layers of the bi-lstm can be used to create the final embeddings using a learned linear combination of them. The weights of the linear combination depend on the downstream task and are chosen during fine-tuning.
I think that there won't be any problem of model learning to copy the input because

the forward lstm and the backward lstm stay separate, and receive different inputs. The second layer of the forward lstm doesn't see the out put of the first layer of backward lstm.

4.4 BERT

- BERT seems to a priori be a sequence to sequence translation model in the sense that it takes a sequence as input and output a sequence of vectors. I guess that the length of both input and output sequence is fixed and that padding is used. In order to turn it into a classifier, the authors add a first [CLS] symbol at the beginning of the input sequence. The first vector in the output sequence is then used to predict the class (with an additional simple feed-forward network with soft-max).
- I think that like GPT, BERT uses only the output of its last layer for the downstream task (I think that this is what is meant by "BERT and OpenAI GPT are fine-tuning approaches, while ELMo is a feature-based approach".
- In the way it is used, BERT is similar to ELMo and GPT, but tries to take strengths from both models: it uses (decoder) transformer blocks like GPT, but doesn't stick to forward language modeling like GPT (so is closer to ELMo in this sense). In GPT, the self-attention layers can use only input at positions $1, \dots, i-1$ to predict the output at position i. If the self-attention layer was allowed to use the inputs of position $1, \dots, i-1$ and $i+1, \dots, n$ in order to predict the output for position i, then the next layer could indirectly "see" the input of the first layer at position i by looking at the output of the first layer at positions different from i. In order to avoid this problem, 15% of the input tokens are randomly replaced by a special MASK token, and the loss is only computed for the prediction of these tokens (the prediction should match the original token).
- Given the fact that BERT uses a different masking mechanism than the transformer decoder (the transformer encoder doesn't have any), I'm not sure that it is correct to say that BERT stacks up transformers layers, but in this webpage:
  `http://jalammar.github.io/illustrated-gpt2/`
  it is written that BERT relies on a stack of **transformer encoders**.
- Additionally to language modeling, BERT also has a "binarized next sentence prediction task" as pretraining task. There, it has to find if two sentences given in inputs and separated by a special token, are consecutive or not.

04.06.19 ∼ Batch Normalization, Weight Normalization, and Layer Normalization

I read the two following posts about different types of normalization which are supposed to speed up the training of NN:

http://mlexplained.com/2018/01/10/an-intuitive-explanation-of-why-batch-normalization-re
https://mlexplained.com/2018/01/13/weight-normalization-and-layer-normalization-explaine
It seems to be based on the following papers:
http://proceedings.mlr.press/v37/ioffe15.pdf
https://arxiv.org/pdf/1607.06450.pdf

06.06.19 ~ Resiudal Connection

Here is a page explaining what **Residual Connections** are in NN architecture. From what I understand, it just means adding the input of a layer to its output (a bit like in a lstm):
https://www.quora.com/How-does-deep-residual-learning-work

28.06.19 ~ Inductive vs Transductive learning

I learned here the difference between **inductive learning** and **transductive learning**. It seems to me that inductive learning is the usual approach where we try to build a classification/regression function f using only the training set, whereas, for transductive learning, we already have the full test set and might use it (as a whole) to build an algorithm (which will of course also use the training set) which will give the right labels to the inputs of the test set.

19.07.19 ~ Ressources for semantic parsing

This paper (that I haven't read yet) gives a **good historical overview of semantic parsing**:
https://arxiv.org/pdf/1812.10037.pdf
Edit (06.11.19): I read the section 2, "Related Work". It's indeed quite detailed but I would need to read the papers mentioned there to really have an overview. and this paper gives a good introduction to **logic based formalism**, **Graph based formalism**, which can both be used for semantic parsing:
https://openreview.net/pdf?id=HylaEWcTT7

21.07.19 ~ Common sense with Concept NET

I read in the description of the COIN (EMNLP 19) workshop that **Concept NET** can be used to model **common sense**:
http://conceptnet.io/
the Read The Web project is also mentioned:
http://rtw.ml.cmu.edu/rtw/

30.07.19 ~ Lambda Calculus, Haskell and Category, Homotopy Type Theory

I read part of this wikipedia page on **Lambda Calculus**:
`https://en.wikipedia.org/wiki/Lambda_calculus`
what I get out of it is that you can write a sequence of operation on a given type of input using this formalism. The specific low-level operations that one performs can be arbitrary (I guess) as long as there are preliminary set. I think that this formalism can be used to write trees or grammar trees (Bilyana had explained me something like this with map-reduced).

I met a man called Declan, working for the UK government who told me that **Category Theory** can be used with Haskell a "purefly functional programming language". He explained me that a map function can be seen as a functor. Here is a stack overflow page about the link between category theory and haskell:
`https://stackoverflow.com/questions/57128213/what-concept-in-category-theory-can-be-used`
Declan also mentioned **homotopy type theory**:
`https://en.wikipedia.org/wiki/Homotopy_type_theory`
Edit (22.10.19): Sarvesh pointed me toward this book:
`https://github.com/hmemcpy/milewski-ctfp-pdf`

03.09.19 ~ Generating Logical Forms from Graph Representations of Text and Entities

Joram presented this paper:
`https://arxiv.org/pdf/1905.08407.pdf` which adapts the concept of Transformer to see it as a way to embed a fully connected graph and transforms, layer after layer, in the encoder the representation of it nodes. In the described case the inputs are the list of token in a natural language sentence, and a list of possible entities (with possible overlaps or wrong entities), and the output is a query for a graph data base (the graph is given). One of the key ideas is to compute differently the attention depending on the two types of nodes (for which the attention is computed).

17.09.19 ~ Introduction to XLNet

I started reading this article which introduces **XLNet**:
`https://mlexplained.com/2019/06/30/paper-dissected-xlnet-generalized-autoregressive-pret`
That made me ask myself the following question: how are exactly concatenated the intermediate hidden outputs of the previous layers with the hidden outputs of the current layers? I guess that instead of having n different (customized) word embeddings, we have 2n, but that one computes new vectors only for the n new (the n old are only used to compute keys and values).

The key points I take out of this are:

1. XLNet is based on **Transformer XL**. This uses a recurring structure to **get rid of the limitation on the size of the input** of the vanilla Transformer. It also replaces the mask mechanism by **Permutation Language Modeling**. The ability to parallelize the computation of the prediction for each input token is lost but some problems (masks do not appear at testing time, predictions for two masks tokens are independent) are resolved. It is not absolutely clear to me if the authors of XLNet are using Transformer XL as such or only take inspiration from it.

10.10.19 ∼ Loss function for classification: Cross Entropy

I came back to the topic of the loss function since it's a crucial point when it comes to the training of these NN (using SGD). I read this interesting post:
`https://rdipietro.github.io/friendly-intro-to-cross-entropy-loss/`
which explains that cross-entropy is related to the idea of **compressing information** and also to the idea of **likelihood**, which makes it a good loss function. If there are n examples the loss will be

$$-\sum_{i=1}^{n} \log(\mathbb{P}\left(\{\text{prediction for sample } i = \text{ground truth for sample } i\}\right)$$

15.10.19 ∼ ERNIE and ERNIE 2.0

I started reading the paper about **ERNIE 2.0** coming from Baidu:
`https://arxiv.org/abs/1907.12412`
For this I had to look at the first version of ERNIE:
`https://arxiv.org/pdf/1904.09223.pdf`
What I get from ERNIE is that replace the masking procedure of BERT by more sophisticated ones. First, there are **several pretraining-task**, all **similar to word prediction**, that they regroup under the name of **knowledge integration**. The first one (**Basic-Level Masking**) is similar to what's done in BERT. The second (**Phrase-Level Masking**) consists in masking (logically coherent) parts of sentences. And the third one (**Entity-Level Masking**) consists in masking entire entities. But then one obviously needs more than simple text. One needs tagged text for entities, and a tool to cut (in a good way) parts of sentences. Let's note that all these tasks were performed on English and **Chinese** as well.
ERNIE 2.0 generalize the idea by adding new pretraining tasks. They are split in 3 categories:

1. **Word-aware**
    - Knowledge Masking Task (this covers all 3 original tasks of ERNIE 1.0)

- Capitalization Prediction Task
- Token-Document Relation Prediction Task

2. **Structure-aware**
   - Sentence Reordering Task
   - Sentence Distance Task
3. **Semantic-aware**
   - Discourse Relation Task
   - Information Retrieval (IR) Relevance Task

The Information Retrieval Relevance Task is based on a dataset created by Baidu search engine.

21.11.19 ∼ Recurrent Neural Network Grammars, Building a Neural Semantic Parser from a Domain Ontology

I read this paper which introduces a new neural architecture called **Recurrent Neural Network Grammar (RNNG)**:
`https://arxiv.org/pdf/1602.07776.pdf`
It can be used for two tasks: random generation of English sentences with an associated parse tree, creation of parse tree for a sentence. Given an input sentence, the model recursively chooses a sequence of actions among "opening a non terminal node" (meaning creating a new subtree), **SHIFT**, and **REDUCE** (see p.2 of the paper for details). Recurrent Neural Network are used at each step to encode the current state of the algorithm, and the next action is predicted using this state (made of three vectors) with a simple one layer NN (see section 4.1).
The authors this paper of facebook:
`https://www.aclweb.org/anthology/D18-1300.pdf`
present an annotation system which goes beyond the simple intent and slots paradigm, with annotation being a kind of tree over the sentence. The tree structure comes with the benefit of clearly decomposing the general order to give to the computer into sub task, making the task of designing the DM somehow easier. The RNNG presented above perform very well to create these trees. facebook provides both an implementation as well as an annotated data set:
`https://pytext.readthedocs.io/en/master/hierarchical_intent_slot_tutorial.html`

Edit (09.01.20): I had a quick look at this paper:
`https://arxiv.org/pdf/1812.10037.pdf`
which intends to **build a neural semantic parser from a Domain Ontology**. I

have the impression that it bares some similarity with the above approach, but I didn't dig enough into the details to fully understand. It can generate some natural language queries data from an already existing ontology, and it can also create a tree representation of a query in natural language. I'm not sure to understand exactly what the different steps of the tree generation are, and I haven't read how they are chosen. I just know that it's using NN, so it is kind of similar to the above method.

14.01.20 ~ Text generation with specific topic

Joram presented this paper:
`https://arxiv.org/pdf/1912.02164.pdf`
which aims at doing text generation using a LM based on GPT2, with the possibility of deciding **which model topic is going to be the main theme of the generated text**.
According to the paper, this approach requires less training (maybe even no additional training) than the classical approach consisting in fine-tuning the language model on specific data associated to the given topic. It seems to be somehow related to what is done with style transfer in image processing.
The key idea was to use some implementation of hugging face for GPT2 (this implementation was introduced for optimization reasons), and before predicting what is the $k+1$ word, one update the activation values (and not the attention weight (this is a trick similar to the one used in transfer learning in image processing)) of the k first words by using back propagation with regards to another loss function, not related to linear modeling but to topic prediction)

29.01.20 ~ Neural Architecture Search

While reading this article:
`https://ai.googleblog.com/2020/01/towards-conversational-agent-that-can.html?m=1`
I stumbled upon the topic of **Neural Architecture Search** which seems to be a new branch of DL where the architecture of the neural network is automatically found. I looked at the wikipedia page:
`https://en.wikipedia.org/wiki/Neural_architecture_search`
but I didn't get much out of it.

15.03.20 ~ Doc2Vec

In this post I sumbled upon **Doc2Vec**:
`https://medium.com/towards-artificial-intelligence/multi-label-text-classification-using`
I found the explanation given here:
`https://medium.com/wisio/a-gentle-introduction-to-doc2vec-db3e8c0cce5e`
not very satisfying because it's not clear at all how embeddings would be created for

new paragraphs. For Word2Vec the vocabulary is fixed before the training so how could we use the same method but allow having new paragraphs? The trick is that it's not the same kind of training. As explained in the paper:
`https://cs.stanford.edu/~quocle/paragraph_vector.pdf`
at inference (which is here means when we want to use our model to generate embeddings for new paragraphs), one actually does the same kind of back-propagation training as during the initial training, except that one can only update the weights of the vector of the paragraph (this is all we care about, and the word vectors have been trained in the initial phase of the training).

19.03.20 ~ AWD-LSTM

I read this article about **AWD LSTM**:
`https://yashuseth.blog/2018/09/12/awd-lstm-explanation-understanding-language-model/`

from what I understand, the big changes are: drop out is done at the level of the weights (the one connecting the hidden layers), and there is a variant of SGD used for optimization. The first thing prevents over-fitting (from what is explained, I understand that it's more appropriate than classical drop-out for RNN), and the second helps better optimizing the weights. The task described was language modeling.

16.07.20 ~ keyword extraction: RaKUn, RAKE

I finished reading the RaKUn paper:
`https://arxiv.org/pdf/1907.06458.pdf`

- An edge is built between two tokens if they appear one after the other in the same line (according to the implementation since the paper isn't clear)
- The method seems to work better for long documents (see p.10).
- The method allows to retrieve expressions made of 1, 2, or 3 tokens.
- If 3 tokens are extracted, it might be that they do not appear sequentially in the text (see page 5 and 6).
- By default, keywords extracted are lemmatized.
- There is an open source implementation.

Edit (06.08.2020): I read the paper explaining how **RAKE**, an other method for keyword/keyprhase extraction works:
`https://www.aclweb.org/anthology/W04-3252.pdf`
There is an open source python implementation:
`https://pypi.org/project/rake-nltk/`

The general idea is that it finds keyphrases (bunch of words next to each other) candidates by splitting the text into chunks using punctuations and stop words as seperator. Then there is are three methods to assign weights to these chunks. None of them are very subtle. The default method (using $deg(w)/freq(w)$) gives preferences to longer chunks. Since the chunks weight are computing by summing weights computed for each one of the words constituting them, one could even use this network load centrality computed in RaKUn.

27.07.20 ~ Adaptative Resonance Theory (ART) Neural Networks, Multi-label classifiers

While looking at the possible classifiers offered by the python library scikit multtilearn here (see MLARAM):
`http://scikit.ml/modelselection.html`
I stumbled upon **ART models**. It seems to be a family of neural networks. I found some brief overview in these pages:
`https://en.wikipedia.org/wiki/Adaptive_resonance_theory`
`https://www.geeksforgeeks.org/adaptive-resonance-theory-art/`
`https://www.javatpoint.com/artificial-neural-network-adaptive-resonance-theory`
from which I understood that they have the interesting capabilities of **forming clusters by themselves**, creating new ones if a sample in the training doesn't match any of the previous ones. One apparent weakness (maybe solved by improved versions called TopoART and Hypersphere TopoART according to the wikipedia article cited above) is that the clusters formed depend upon the order in which the samples are fed to the model during training.
I haven't completely understood the architectures of the neural network, and according to wikipedia, the training process is quite unusual (involving differential equations or algebraic equations). I currently don't have the time to dive into the details, but I found two resources which seem to provide detailed explanation. The article accessible here:
an introduction `https://www.researchgate.net/publication/247749347_Adaptive_Resonance_Theory_ART_An_Introduction`
(I have the pdf on my machine)
an overview of the different extensions and existing implementations:
`https://www.sciencedirect.com/science/article/pii/S0893608019302734`
(I have the pdf on my machine under art_models_survey.pdf)

This page presents plenty of models (and their implementation in the given library) for **multi-label classification**:
`http://scikit.ml/modelselection.html`

11.08.2020 ~ Rasa DIET classifier

I looked at this video which introduces quite well the Rasa DIET classifier which does simultaneously intent classification and entity extraction:
https://www.youtube.com/watch?v=vWStcJDuOUk

26.08.2020 ~ Padding and Stride for convolutional neural networks

I read this article which explains what is **padding** and **stride** for cnn:
https://machinelearningmastery.com/padding-and-stride-for-convolutional-neural-networks/

01.10.2020 ~ DistilBert

I read about DistilBert which uses this teacher student idea to make a light version of BERT:
https://kgptalkie.com/distilbert-smaller-faster-cheaper-lighter-and-ofcourse-distilled/
It seems to be accessible in Hugging Face.

01.11.2020 ~ Abstract Representation Meaning (AMR), locally sensitive hashcodes

I read the beginning of this article about a way to represent English as graphs (**ARM**):
https://www.aclweb.org/anthology/W13-2322.pdf
and
here is the **PENMAN** notation:
https://penman.readthedocs.io/en/latest/notation.html
One interesting aspect is that there is no pre defined way to parse an English sentence into one of these graphs, even if a priori, there should be only one graph fitting to a given sentence.

In the WeCNLP conference, I came across a talk about this paper:
http://ceur-ws.org/Vol-2202/paper4.pdf
The main tool that they use is called **locally sensitive hashcodes** or **kernel hasing**. If one has already a way to turn a text into a vector and a way to compute similarity between these vector, this is a technique to speed up the search for the nearest neighbor of a new input text. Instead of using the similarity on every vector, one represent all the vectors (including the new input vector) by a simpler representation (a binary vector of fixed length, built using these **hash functions**). For each simplified representation correspond several different vectors (it's not a bijection). Finding the simplified representation in the data the closest to the simplified representation of the incoming vector is computationally less expensive than computing the similarities to all original vectors. Then one computes the similarities using the original vectors corresponding to the closest simplified representation with the original vectors. This is simply a way to speed up the process, by putting vectors into "buckets" and reasoning first on the buckets to

discard a majority of them.
Section 3 of this paper:
Dialogue Modeling Via Hash Functions (I have it on my computer)
and section 5 of this paper:
Similarity Estimation Techniques from RoundingAlgorithms
give more details.
There are several ways to build the has functions. In the paper by this Irina Rish (the one of the presentation) they explain the following approach. If their training data size is M, they fix an integer parameter $1 \leq \alpha << M$ and then for each hash function (which will produce one bit), they sample at random a dataset of this size, split it in two randomly, and train a binary classifier to distinguish these two subsets. The binary classifier gives the hashing function.
Edit (04.11.2020): I started reading this paper presenting an **overview of hashing techniques**:
A survey on learning hash
Section 2 explains how one can use this to speed up **nearest neighbor** and how one can use it for **search** (which is not so different). For this second task, the authors distinguish between two techniques **hash table lookup** and **hash table ranking**. The first one aims at grouping similar items together into bucket (they get the same hash) and when a query enters, its hash is computed and the bucket with the given hash is used to find the likely neighbors. The second one is to try to use the has as a way to reduce the dimensionality of the original items. Each item is mapped to a different hash but since the dimension is smaller, it is easier to compute the distances.
Exit (08.11.2020): I progressed in the reading of the article presented in WeCNLP:
Dialogue modeling via hash functions
If I get it correctly, here is the **general idea**:

1. Using a special objective function (see section 4) they optimize the parameters of their hash functions using a special objective function based on **mutual information** in order to have two things at the same time: first, to have patient and therapist responses couples similar, second, to have the different hash functions producing independent bits. If one was only aiming for the first goal, having all hash functions always mapping to 0 would be the best, which would be poor. By adding this second constraint, we hope that the hash functions learn to represent different aspects of the responses.

2. Once we have these hash functions optimized, we encode the whole training set using them, obtaining this way pairs (c_i^p, c_i^t), where c_i^p is the hash code of the ith response of the patient and c_i^t is the hash code of the ith response of the therapist. Then one trains a classifier to predict c_i^t from c_i^p.

3. At run time, we first compute the hash code of the response of the patient, then we predict the hash code of the therapist, and then we retrieve one of the text answers of the therapist associated with the predicted hash code.

4. The objective function used for training can also be used to evaluate the quality of the response prediction to compare different choice of hashing function types.

I was wondering why we need to predict the hash code of the therapist's response instead of just using the hash code of the patient as proxy, but since it is the same hash functions used to encode the patient and the therapist responses, and that the patient tends to talk more, it might very well be that some bits are often different in the patient and in the therapist response. The prediction could account for that (Figure 2 points in that direction).
From what they say in the paper, this prediction step is an innovation. So far, only vector similarity on the hash code was used.
They test several types of hash functions and compare their results.
One interesting thing with this approach is that all these hash functions are built using a subset of the training set. So one can choose this subset in order to push the model to learn some features.
At the end of the paper, they present some predicted answers, and these are really not convincing, so this approach has a nice theoretical side, but doesn't deliver perfect results. During the talk, the woman said that it still worked better than some NN based alternatives, but I don't remember them.

18.11.2020 ~ Contrastive loss

I learned about **contrastive loss** here:
`https://towardsdatascience.com/contrastive-loss-explaned-159f2d4a87ec`
It is a loss function which can be used to obtain embeddings such that the similarity between examples from the same class are high and similarity between examples from different classes are low.

30.11.2020 ~ Bias-variance trade-off, How to train a neural network

I found that these two pages explain well how the **bias-variance trade-off** is related to **underfitting** and **overfitting**:
`https://www.machinelearningplus.com/machine-learning/bias-variance-tradeoff/`
`https://towardsdatascience.com/is-your-model-overfitting-or-maybe-underfitting-an-exampl`

I also found this page which gives a lot of **concrete recommendations** on how to **tune a neural network**:
`https://www.wandb.com/articles/fundamentals-of-neural-networks`
In particular, it touches topics like:

- How to solve **vanishing gradients** and **exploding gradients**, and how this is related to activation functions, weight initialization, batch norm, gradient clipping, early stopping. See section 4.
- Different **activation functions** and when to use which. Plenty of variants of

activation functions are quickly introduced like logistic, tanh, ReLU, Leaky ReLU, ELU, SELU, GELU, PreLU. See section 4.

- Different types of **weight initialization methods** including **He initialization**, **LeCun initialization**, and **Glorot Initialization**. See section 4
- **Batch norm**
- **Dropouts:** Recommended values for different types of neural networks, variants (AlphaDropout)
- **Optimizer:** When to use SGD and when to use its cousins.
- **Learning Rate Scheduling**

The tutorial also points to plenty of additional material.

Edit (29.12.2020): I will try to gather here many useful resources and information concerning the training of neural network:

- **Sebastian Ruder** overview of **stochastic gradient descent and its variants**: `https://ruder.io/optimizing-gradient-descent/`
- A good simple practice is to have a **validation set** on which we measure the accuracy of the model after each epoch of training. Save the model when this accuracy reaches a maximum (with respect to previous epochs). In addition, one can let the model train as long as the results on the validation set improve, and stop the training when the model hasn't improved for let say 3 epochs (3 is the so called "patience" here). This method is called **early stopping**.
- One can also use the **cyclic learning rate** update scheme presented in this paper: `https://arxiv.org/pdf/1506.01186.pdf`

Edit (01.01.2021): I would like to have a clear **method to tune Neural networks**. I will try to describe it here.

1. Fix a type of model (BOW + logistic regression, GloVe + LSTM, BERT, etc...)
2. Fix the meta-parameters of the model (number of layers, size of the layers, etc...)
3. Fix the optimizer
4. Use the following training schedules:
    - For different learning rates (start high): Train the model with the same learning rate for a large number of epochs while monitoring
        - Loss on training set
        - Loss on validation set

  - Appropriate metric (MSE, Accuracy,...) on validation set

  Look for moment when the loss on validation goes high again as it means that the model is overfitting. Save model when max accuracy/mse on validation is reached.

- **Annealing learning rate**: Start with high learning rate. Train for a fixed number of epochs with it (5 to 10), then for each new epoch, if the accuracy/mse on validation is not improving divide learning rate by 10 (I add the condition not to update the learning rate twice in a row to avoid having the learning rate shrinking to fast).
- **Cyclical learning rate**: Have the learning rate linearly cycling from a min to a max, going in several steps from one to the other. Probably good to use a burning period with fix learning rate. On can choose the bound using a technique explained in this paper:

For each of these methods, one can save computation, one can use early stopping (with patience 3 or 4).
In order to see which learning rates range is good, one can use visual inspection with the first method.
If we have limited time for a specific configuration, and we don't want to do an exhaustive search, I would just use annealing with initial learning rate 1, training during 10 epochs, and early stopping.

Because of time and resource limitations, there is a trade-off between getting the most out of each model architecture (and hence trying plenty of learning rates and learning rate schedules) and trying out a big variety of different models (and hence multiplying the effort dedicated to each one by the number of architecture tried out).

08.01.2021 $\sim$ NN Weight initialization

I read this webpage explaining how **weight initialization** of neural network layers and the **activation function** used are **related**:
`https://www.machinecurve.com/index.php/2019/09/16/he-xavier-initialization-activation-fu`
More precisely, some activation functions like sigmoid or tanh have a very low derivative on values outside a quite small window. This means that the optimization process will be very slow at first. I guess that there are other considerations but I didn't delve into the details. The two main techniques seem to be:

1. **Xavier initialization**: weights of a layer are initialized randomly using a Normal distribution $\mathcal{N}(0, \sigma^2)$ where the variance σ^2 is set to be equal to $1/N$ where N is the number of incoming neurons (size of the output of the previous layer). It was first proposed in this paper:
   Understanding the difficulty of training deep feedforward neural networks
2. **He initialization**: from what I understand, it's the same but one uses $\sigma^2 = 2/N$. It was introduced in this paper:

Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification

Here is another paper doing a review of the topic:
On weight initialization in deep neural networks

Note that as explained in the third answer to this stack overflow post:
`https://stackoverflow.com/questions/49433936/how-to-initialize-weights-in-pytorch`
Pytorch has already some of these initialization techniques used by default by most of their layer types.

04.02.2021 ~ Improving ASR with GPT2 Language Model

This article: Joint contextual modeling for ASR correction and language understanding
a lot of approaches to **correct the ASR output** are presented. Either directly, or citing other papers. One simple trick presented in the section "Statistical and Neural LMs for Re-ranking/Re-scoring" (between p.2 and p.3), is to use a Language Model coming from GPT2 (by it's architecture it can easily be used to predict for each word in its vocabulary the probability that it is the next word) to compute a score for each ASR suggestion and rerank it.