Transfer Learning with Language Models for Classification Problems

Janne Nevalainen

Helsinki November 8, 2020

UNIVERSITY OF HELSINKI
Department of Computer Science

HELSINGIN YLIOPISTO — HELSINGFORS UNIVERSITET — UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty	Koulutusohjelma — Studieprogram — Study Programme
Faculty of Science	Computer Science

Tekijä — Författare — Author
Janne Nevalainen

Työn nimi — Arbetets titel — Title

Transfer Learning with Language Models for Classification Problems

Ohjaajat — Handledare — Supervisors

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
	November 8, 2020	93 pages + 2 appendices

Tiivistelmä — Referat — Abstract

Neural network based modern language models can reach state of the art performance on wide range of natural language tasks. Their success is based on capability to learn from large unlabeled data by pretraining, using transfer learning to learn strong representations for the language and transferring the learned into new domains and tasks.

I look at how language models produce transfer learning for NLP. Especially from the viewpoint of classification. How transfer learning can be formally defined? I compare different LM implementations in theory and also use two example data sets for empirically testing their performance on very small labeled training data.

ACM Computing Classification System (CCS):
General and reference → Document types → Surveys and overviews
Applied computing → Document management and text processing → Document management → Text editing

Avainsanat — Nyckelord — Keywords
Language model, transfer learning, NLP, word embedding, business news, hate speech

Säilytyspaikka — Förvaringsställe — Where deposited

Muita tietoja — övriga uppgifter — Additional information

Contents

1 Introduction

1.1 Pitch

Recently, language models have made big breakthroughs on NLP (Natural Language Processing) and reached SOTA (State Of The Art) results on several NLP tasks, making large improvements on the previous best models. They have also reached good results with lot less labeled data, than previously needed.

While some companies have obtained large amounts of annotated data, they often keep it private as a business asset. This raises the need for others to be able to use unlabeled data for learning.

Language models allow learning from large unlabeled datasets, such as Wikipedia and review databases, with billions of words and enormous amount of information. With transfer learning, this learned info can be used for advantage on new different target tasks and domains.

Many companies and application areas have hundreds of thousands or more text documents. **We can now use self supervised learning to learn from those.**

Tasks that previously required 10.000 or more labeled examples, making them in many cases totally undoable because cost of annotating, can now be done with even just 100-200 labeled examples, becoming easily doable in short time. **Language models enable whole new application areas for NLP!**

ULMfit [HR18] showed that their model can outperform previous state of the art accuracy by huge 20 percents! BERT [DCLT18] broke records and improved SOTA on 11 NLP tasks. On task of reading comprehension (SQUAD) they reached 93.2 % accuracy, outperforming human performance by 2%.

I will look at what language models are and how the transfer learning happens with them, using two concrete datasets and challenging tasks as real examples - those of hate speech detection and sentiment analysis of business news. I will also empirically compare several different language models and test how much labeled examples are needed in practise.

1.2 Background

Traditionally a limitation in training machine learning models has been the small amount of available labeled training data. In supervised learning, examples need to be labeled: this sample of text is positive, that one is negative, etc. Manually marking labels has been time consuming and expensive. In hard NLP tasks - the interesting ones, it is also common that not all human annotators agree on labels of samples, but may give single sample different labels.

Language models avoid this problem by being able to learn without labels. They can for example use whole English language Wikipedia as training data and instead of humans needing to read and label each article, the model is able to learn without labels (this is called unsupervised, or more recently - self supervised learning). The task given for the model to learn can be to predict the text itself. For example giving the model some amount of documents content, starting from beginning, but blocking each next word one at a time (and all text following that), letting the model predict probabilities for next word. This way the model can go through billions of words of text and learn from it.

The information learned by the model can then be used for domain and tasks different from the original. If original domain is Wikipedia texts and original task predicting next word, given the previous words, the new domain can be for example business news articles and and the new task can be predicting polarity - is the article positive or negative. This concept of learning from data in one domain and task, to be used for advantage in other domain and task is called transfer learning.

Embeddings

Context free word embeddings - generally referred just as **"word embeddings"**, such as Word2vec [MCCD13] and Glove [PSM14], released in 2013, learn and store single embedding per one word in vocabulary.

Contextual embeddings, which LMs mainly produce, can have different embedding for same word, depending on the context it appears in each time. Context being the words around it.

Token level

Models and embeddings can differ on what they use as the basic language unit - token, to work over and to create representation for. As an example word 'ungodly'. Word level model can treat each word as one smallest unit, mapping each word to

one integer: 'ungodly'. Character level model could treat each character of the word as own token: 'u-n-g-o-d-l-y.' Subword units can use different tokens, smaller than word, such as 'un-god-ly', learning representations for each subword units.

NN-Architecture

In Deep Learning for NLP and language models, there are two main architectures currently popular. One using LSTM (Long Short Term Memory) -architecture, feeding each iterations model output back to the model as input. And other one is Attention [VSU+17] based Transformer, that avoids the iteration, using direct pointers to areas in the text, assigning weight for how important it considers them.

Transfer Learning

Transfer Learning for NLP can be formally defined by concepts of Domain and Task, still separating these into Source and Target versions.

Source Domain $\mathcal{D}_S$ could represent training data set of English language Wikipedia articles and the text probabilities in it. Target domain $\mathcal{D}_T$ could be a set of business news articles and text probabilities of that.

The Source training task $\mathcal{T}_S$ can be that of hiding some words of the training text at time, and since the real value of hidden word is already known, using it as a training target. The Target task $\mathcal{T}_T$ could be a classification task, such as polarity of each input article in scale of how negative or positive it is - using human given labels.

1.3 Motivation

I will look into multiple modern Neural Network (NN) -based language models, such as ELMO [PNI+18], ULMFit, BERT, OpenAI GPT [RNSS18]. What are they and how do they reach high performance? What kind of design choices they use and how they differ from each other? (choice of training data, pretraining task, NN-architecture)? What do they have in common, how do they differ? How well they perform on different types of datasets?

What are important properties of LM? If considering new LMs several years in the future, can we already think now, what properties they are likely to have?

As a foundation and context to contrast them into, I will look at the core areas: transfer learning, representations/embeddings and language models.

Also if given a new dataset x, how much data can we expect to need as a) labeled

training data, b) unlabeled data. Can we estimate what kind of LM would be most suitable for specifics of certain data set and why?

Organizations often have large amounts of unlabeled textual data - such as hundreds of thousands of customer e-mails, large amounts of error reports etc. Being able to learn a model unsupervised (or semi-supervised) way from unlabeled data, perhaps adding only a small amount of manually annotated examples - a few hundred or one thousand perhaps - something that can be produced with couple person work days, provides a way of gaining valuable information and practical value from these data.

Desirable properties of the whole model building are minimum amount of required manual work for annotation and reasonable or small amount of processing power required for training.

1.4 Research questions

What is transfer learning for NLP? What is transfer learning for language in theory? How does the transfer happen with different systems: word embeddings and LMs?

What is a language model? How do LM:s implement the transfer?

What are a language models in theoretical sense? What are context free word embeddings and how do contextual embeddings produced by LMs differ from them? What kind of language models existed before modern NN:s? How do modern LMs produce the transfer learning?

How well do Language Models perform on text classification?

How much improvement do they bring against context free word vector based models? Assumption is, they can learn/capture more information about the text than static word vectors could.

What kind and how large differences are there between different language models? Such as depending on - architecture, source training data used , whether model is character level, subword level, or higher level.

How much labeled training data is needed? Can there be seen a difference that some type of language models perform better on small amount of labeled data and others require more? Does some amount of labeled training data seem to be enough?

I will look at these matters and questions also through two different data sets and their related classification tasks:
1) Business news and sentiment analysis
2) Twitter tweets and hate speech detection

1.5 Summary

Organizations have a lot of unlabeled textual data. With the viewpoint of taking advantage of this data with transfer learning:

I will look at different high performance and state-of-the-art models of their time: Bag Of Words and TF-IDF [MRS08], Word2vec, Glove, Fasttext [BGJM16] and [JGBM16], ULMFit, Bert, GPT etc. What implementation choices they made and how they differ from each other? What did they improve over previous models?

- Chapter 2: We will see how to represent textual data with numerical feature vectors. We will also look at traditional count based representations: Bag of Words, bag of N-grams [MRS08], TF-IDF.

Then we move to word embeddings produced by Neural Network and prediction: Word2vec, Glove, Fasttext.

- Chapter 3: We will look at Language models. Chapter 2 looked at **context free** word embeddings. Each word has one embedding, that is used everywhere through the text. With LM:s we will move to **contextual word embeddings** - a word is given different embedding through the text, depending on the words around it.

- Chapter 4 – Evolution of Language models. We look at real implementations of language models: ULMFit, BERT, GPT etc. Their design choices: pretraining data, pretraining task, architecture etc.

- Chapter 5: We will look at transfer learning. What is transfer learning? How it can be formally defined and represented? How and where the transfer happens?

- Chapter 6: We will introduce two example data sets and tasks: Sentiment analysis of Business news and hate speech detection of Twitter tweets.

We will see how to formally represent the example data sets as transfer learning. Especially how to define the Target Domain and Target Task.

- Chapter 7: I will empirically compare multiple different pretrained models for our two example data sets and their tasks. How do they perform in practice with low

amount of labeled target data: from as low as 100 annotated samples up to a few thousands.

- Chapter 8: Conclusions.

2 Representations and Embeddings

In representation learning the goal is learning such representation for data, that captures important information, and helps to succeed in required task or tasks. The learned representation follows a distribution fort the data used in pretraining - for example Wikipedia articles.

Sometimes the representation is used only for one kind of data and one kind of task, in which case, it is enough to perform well in those. In this case an overfit to training data and task is ok and can be even favorable.

In case of Transfer Learning, the model / representation is wished to also work for other kinds of data and other kinds of tasks. The goal in the pretraining phase already, is to learn more generalized distribution. Later the model weights can in addition be fine-tuned from the learned distribution more towards the distribution of the target data. This could be prelearning from Wikipedia articles, then if target data is business news articles, the language model (its weights) can be finetuned closer to the real distribution of the target domain.

2.1 Definitions

Here I will look at some terms, that are commonly used in NLP.

Word

Word in linguistics is defined as the smallest sequence of phonemes that can be uttered in isolation, and that have objective or practical meaning.

In NLP "word" can have different meanings in different contexts. On one hand, single word can have different inflections: eat, eating. On the other hand, very commonly, such as in word embeddings, different inflections of a word can be considered a different word and given different embedding. Reader is good to be aware of this distinction, and its possible change in different contexts.

Morpheme

Morphemes are the minimal units of words that have a meaning and cannot be subdivided further. An example of a free morpheme is "bad", and an example of a bound morpheme is "ly." [1]

Word	morphemes
badly	bad + ly
rocks	rock + s
cannot	can + not

Polysemy - Different senses of word

Same word can have multiple meanings. For example: "I was walking on a river bank." "I took money to the bank."

Another example is word "interest". The meaning and location of word embedding will be located somewhere between "interest for loan" and "interest in soccer",

Units larger than word

Words can also be combined to form other elements of language, such as phrases, clauses and sentences.

A clause is a group of words that contains subject and verb. A clause can alone form a sentence.

The sentences "I looked everywhere, but the cat was gone." includes two clauses: "I looked everywhere" and "the cat was gone".

A phrase is a group of words in a sentence that does not contain a subject and a verb. As such a phrase can not stand alone as a sentence.

"On the wall, in the water, over the horizon." are three clauses. We will later see, how in word embeddings, it is helpful addition to not handle phrases like "New York times" as separate words, but as a one phrase consisting of multiple words, but having one meaning.

2.2 Count based representations

For representing text and giving it as input to machine learning models, it needs to be changed into numerical form. There are multiple kinds of count based feature

[1] https://ielanguages.com/morphology.html

vectors, such as BOW and TF-IDF. These provide a feature vectors that tell how much each feature is present. Then a classifier such as Support Vector Machine can be trained on top of these features.

Historically, early common representations for textual data were numerical info about the document. One can for example count the times each words appear in texts with different label, documents length in characters or words etc. One advantage of numerical information is that it is often very fast to process.

2.2.1 Bag Of Words - BOW

Bag Of Words - BOW [MRS08] is a simple representation for text, where number of occurrences of each word in a document or dataset is counted. Similarity of two documents could then be compared by how many of the same words they contain.

BOW is a **multiset** - a set where each item can have a count higher than one. As a "bag" / set, information about location or ordering of the words in text is discarded and only count of occurrences is kept. This count is usually normalized by total number of words in the document.

If one were to build a classifier for review texts, then words ['love', 'great', 'fantastic', 'nice'] might appear more often in positive texts with positive class, while words ['bad', 'horrible', 'weak'] might appear more often in texts with negative class.

For counting words, it is not necessary to understand anything about the words or their meaning. Just to go through a large enough amount of texts that have been separated into groups by label and what ever words occur more commonly in one class, will suggest that class, and words occurring more common in other class, will suggest that class.

Lets see an example of how to count BOW:
"John likes to watch movies. Mary likes movies too."[2]

This responds to a list of words:

$$"John","likes","to","watch","movies","Mary","likes","movies","too"$$

BOW as JSON-object.
BOW1 = {"John":1,"likes":2,"to":1,"watch":1,"movies":2,"Mary":1,"too":1};
as non ordered, this is identical to alphabetical order:

[2] `https://en.wikipedia.org/wiki/Bag-of-words_model`

BOW2 = {"John":1,"likes":2,,"Mary":1, "movies":2,"to":1,"too":1,"watch":1,};

Example document can be represented in shorter form - a document vector, where each index responds to a specific word in a fixed dictionary:

$$1, 2, 1, 1, 2, 1, 1, 0, 0$$

When documents are represented as the counts of words, a machine learning model can be trained over these representations. Here we can also see, that the word "likes" is turned into just a "word with index 1". (starting from index 0, which is "John"). The word with index 1 occurs 2 time in document x, is all that the trained model will see. It does not anymore see which word it was.

2.2.2 Bag of N-grams

In n-gram [MRS08], count of n consecutive words or tokens in specific order is recorded. Bag of Words can be considered an n-gram where n is 1, that is uni-gram. In bi-gram word pairs are counted. In 3-gram triplets of words etc. Using N-grams higher than single word, allows recognizing possible phrases of for example 2 or 3 word, or other words appearing commonly together in certain kind of text.

As in BOW, in n-gram more generally also, a maximum amount is usually defined for m most common n-grams to appear in the text. This will allow to reduce the memory requirement, complexity and reduce possible problem with curse of dimensionality.

2.2.3 Term Frequency - Inverse Document Frequency (TF-IDF)

From the viewpoint of a single document, a term (word) that appears in this document, but also appears very commonly in almost all of the documents, gives little information about this specific document. A word that is rare in the documents generally, but is present in this document, on the other hand, can tell more, how this document differs from the others.

To increase the value of rare words and decrease value of common words, a technique called **Term frequency - inverse document frequency or TF-IDF** [MRS08] can be used. TF-IDF will give an importance weight for each term t in each document d.

Term frequency in its simplest form is a raw count of given word in given document. It gets higher value, the more often term appears in the given document.

TF is counted for all terms t and all documents d:

$d \in D, tf(t, d)$

If word "spectacular" occurs 2 times in document i, then tf("spectacular", i) = 2

In practise the id:s of the words will be used: if in the vocabulary, index 1047 corresponds word "spectacular", then tf(1047, i) = 2.

Inverse document frequency for a word is a measure going lower, when more of all documents contain that word.
Inverse Document Frequency - IDF, for term t in documents D is calculated as:

IDF(t, D) = log(nr of all docs / nr of docs containing this word)

If there are total of 1000 documents and 905 off them contain word "we", its IDF in all documents D will be: $idf(\text{"}we\text{"}, D) = log(1000/905) \approx 0.043$.

If the word "spectacular" only appears in 32 documents, its IDF will be:
$idf(\text{"}spectacular\text{"}, D) = log(1000/32) \approx 1.49$.

IDF is a global value - IDF is counted for all terms in all documents D: $idf(t, D)$. When processing the documents, for each document, the IDF of a word will always remain same. How often word "spectacular" appears in all documents D, is one number.

TF on the other hand is a local value - when going through documents, word "spectacular" can appear 2 times in document i and possibly 0 times in the document i+1.

Putting them together: TF-IDF

When TF and IDF are counted, their combining, TF-IDF is counted as TF * IDF.

When term "spectacular" appears 2 times in document i, its TF=2 and IDF of term "spectacular" was 1.49, the TF * IDF, tf-idf("spectacular", D)= 2 * 1.49 = 2.98

TF-IDF weight for the term "spectacular" in document i, will be 2.98.

There are variants of TF, such as Normalized TF, where count of the term is divided by total number of terms in the document.

2.3 Representing document as a vector

Here we can see, that these counts based models: BOW, N-gram count, TF-IDF produce a document vector for each document as a representation of it. The resulting

vector represents whole document, where as with word embeddings, each word vector represents only single word. As such, representing a document with word vectors still requires extra step, such as concatenating all word vectors of a document together, or taking a sum or average over those vectors. In each of these, the result will be a vector representation for a document. This vector can be considered a feature vector. Then each end task, such as classification still requires building a classifier on top of those features.

2.4 Preprocessing text

Count based representations

Traditionally for count based representations, such as BOW, n-gram, TF-IDF and word vectors, it is usual to preformat text with goal of getting the vocabulary smaller and occurrence of vocabulary items higher.

These methods include stemming or lemmatization to convert words to basic form, converting capital letters to lower case. For pure count based methods this is important, because they don't take notice of words meaning or internal structure, but only count its appearances. So that they can count different inflections: "eat" and "ate" or different casings "Eat" as same word, instead of totally different and non related words.

Also numbers can be converted into few combined once, to mark eg that 12 and 15 have something similar. These combinations could include numbers of length 1-4 as NUM, numbers longer than 5 figure as NUMLONG. Special characters such as "euro", "$", "£", can be all converted to same token, such as "$" to allow recognising that one sentence talking about "$" and another about "£" are referring to roughly same thing - monetary value.

Word embeddings

Word embeddings produce continuous embedding with word similarity. Due to this, they have less need for words to be in exactly same form - two different words with similar meaning will still locate in vector space near each other.

While purely count based methods count 'eat' and 'ate' as two different words with no relation, word vectors are more capable of capturing different forms and representing the similarity of the words.

For words, such as "eat" vs "ate", "drive" vs "drove". Word embeddings have

capability to map these into vector space such, that subtracting vectors v("eat") - v("ate") from each other can give as a result roughly the vector of imperfect. So that reducing the resulting imperfect vector from drive: v("drive") - (v("eat") - v("ate")), can result a vector close to vector of "drove".

In other words, Word embeddings have capability to store and represent that: when "ate" is to "eat" as x is to "drive", x is approximately 'drove'.

Due to this, with word embeddings there are less need to heavily preformat the data.

The challenge with representing digits in a meaningful ways still remains, and often it can be favorable in preprocessing turn digits into common tokens, similarly as in count based.

Language models

With Language models and their still larger representation power, the trend is towards keeping the input even more closely to original form. Still though for example with numbers, the LM:s still are fairly weak recognizing the numbers, so converting them into combined tokens might help.

In ULMfit, capitalization was treated with a solution to remove it from main word, but add a special tag marking capitalization before the word. And as such reaching both, making word recognizable by vocabulary + keeping info about capitalization.

BERT on the other hand offers separately uncased and cased versions of pretrained model, offering choice to make distinction or not.

For English language, Spacy tokenizer is commonly used for tokenization. For agglomerative languages like Finnish, Sentence-Piece tokenizer works better. This is due to Sentence-Piece not requiring text to be first tokenized into words.

2.5 Word embeddings and word similarity

While BOW and building over it, TF-IDF, give quite good results with low computational cost, their weakness is not being able to detect two similar, but not same words having similarity, such as "strawberry" and "raspberry". This is due to curse of dimensionality - if training data had a word "strawberry" and new data has "raspberry" they are treated as totally different words. One effect of this is also need for larger amount of training data.

Word embeddings such as Word2vec gave improvement on this. They take notice of

the nearby context of the word - which words it appears near to. Word embeddings will learn that "strawberry" and "raspberry" have a meaning close to each other, due to them appearing often together with same context words, such as "eat" and "pick". As such, their learned word vector will also be very close to each other and also near to vector of the word "berry".

Word similarity

Word embeddings embed a word into a continuous vector space, each word being represented as a dot in the vector space of n dimensions. This embedding is done with structure-preserving mapping, maintaining words relative location versus others.

Especially also, word embeddings are mapped into considerably lower dimensions than BOW. Where BOW might had counts for 30.000 or 100.000 different words - and have as many dimensions, word embeddings generally have for example 50 or 300 dimensions.

With word embeddings, when new unseen word is mapped into this vector space - using the context words around it in new text, it also will have meaningful location in relation to other words or tokens.

Context free and Contextual embeddings

Before going into Language Models, I will look at Word embeddings. In a larger picture, we can consider normal word embeddings, such as Word2vec as context free embeddings - same word having same identical representation as vector in different sentences. Language Models extend this by producing contextual embeddings - same word having different representation in different sentences, taking into account the context.

Goal here is to give reader an understanding with deep enough level, about what word embeddings are and how a context free word embedding can be trained from larger corpus of unlabeled data. This will work as a foundation to later understanding contextual word embeddings, built with Language Models.

"Word embedding is name for a set of language modeling and feature learning techniques where words or phrases from the vocabulary are mapped to vectors of real numbers." (Wikipedia)

In process of word embedding, words are embedded into a Vector Space Model. Large amount of words are put through dimensionality reduction and given representation with n features, ie in n-dimensional continuous vector space. Often this is

for example 300 or 50 features.

So word embeddings for 1 million words with 300 dimensions, in practise can be a matrix of 1 million x 300. Each row corresponding to a single word in vocabulary (a word to integer mapping is created, to connect each row of the matrix i, into a specific word) and each row has a 300 features - 300 float numbers, telling how much that specific word has that specific feature. In practical use, "word embedding" often refers to this kind of matrix of pretrained word vectors - and the weights of the matrix.

Sometimes though term "word embedding" may also be used for the process of creating the embedding, ie transforming a word into vector space.

Distributional hypothesis - words are known by their company

Earlier methods required large amounts of work in manually defining language structures, morphemes, etc. and manually labeling texts with POS tags, trees etc. Due to large amount of manual work required, it remained limited - mostly to English language and inside that into specific fields of texts.

One key feature to overcome this was being able to create word embeddings **without annotated data.** This allowed using as training data very large corpora, such as whole English language Wikipedia - and also to build embeddings for other languages as well.

Underlying idea is based on distributional hypothesis. "You shall know a word by the company it keeps!" - (Firth 1957). That words that tend to appear in similar contexts, have similar meanings and that the other words that commonly appear together with a specific word, will give info about the meaning of that word.

2.6 Word embedding without NN : Co-occurrence matrix

One simple way to produce word embedding, is by counting context words. We could choose vocabulary size, eg. 30.000 words, create a large sparse matrix of vocab size * vocab size, where number of features is as large as the vocab size.

Then go through training data texts, and always when word i appears within chosen context length (such as within 3 words distance) of word j, a count in matrix for row i, column j is increased. Once matrix is formed this way, dimensionality reduction can be used to reduce amount of features, which still is 30.000, to turn it smaller, such as 300 or 50 features. Highly correlating columns (features) can also be collapsed

together, to remove redundancy in columns.

The problems with this kind of simple method include large memory requirement (1 million words vocabulary would require matrix of size 1 million * 1 million), very large amount of dimensions to be produced and curse of dimensionality. Also when the embeddings are produced this way, if new words are added, and the dimensionality reduction is already performed, there is no easy way to add new word, without doing large amount of computation again.

For these reasons, an improved way to create word embeddings, instead of counting each word pairs occurrence individually, will be setting a chosen smaller number of features, such as 50, predicting the the current word by its context (or opposite) by tuning these features weights and then using this produced weight matrix as the embeddings.

2.7 NN-based Word embeddings

Previously we saw how to form word embeddings with counting nearby context words into sparse matrix and then using dimensionality reduction to turning this into sparse matrix. This based on counting the words.

With neural networks, we iteratively **predict context words** and update model weights accordingly with gradient descent.

2.7.1 Word2vec

Big breakthrough of Deep Learning in NLP happened in 2013 when Mikolev et al. introduced Word2vec word vectors. [MCCD13]

By creating a vector representation for words, they could encode and save information about their similarity and meaning. They compute continuous vector representations of words. Each word found in the dictionary is represented by a word-vector, which commonly has for example 50 or 300 dimensions.

Mikolev et al. presented two architectures to produce word vectors. Continuous Bag of Words - CBOW, which doesn't use info about ordering of words and Skip-gram-model, which does.

Continuous Bag-of-Words (CBOW)

In CBOW the algorithms task is to predict current word in the middle, using the

context - info about n preceding and n following words, n being for example 4. The surrounding words are summed together to form one vector, which is used to predict the current word, by updating weights with backpropagation.

As the Bag of Words in CBOW implies, ordering or distance of context words does not matter and info of that is discarded.

Simplified example - training with single context word.

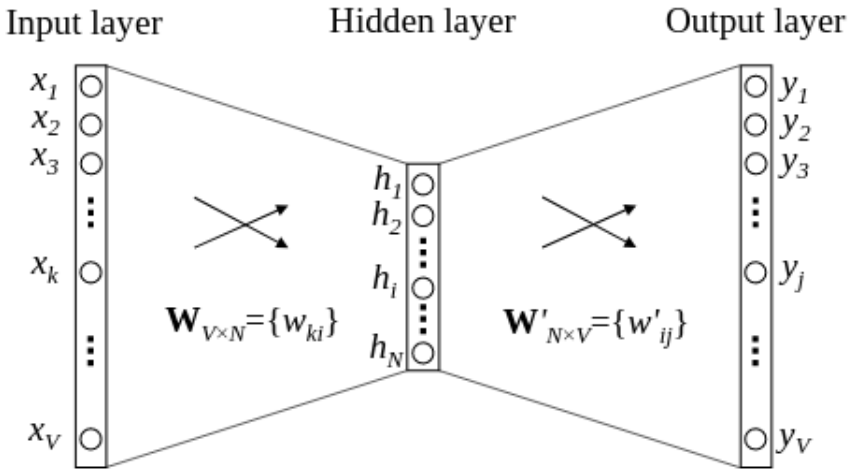

Figure 1: CBOW Xin Rong - Word2vec Parameter Learning Explained [Ron14]. Input layer is size V - vocabulary size and Hidden layer is size of dimensions in the vectors - N. For example 10.000 vocabulary and 300 dimensional vectors.

In Figure 1 is a simplified version of CBOW [Ron14], with only 1 context word as input. The input layer is a 1-hot vector of size V x 1, where only one of the V values is 1 and all others are 0 - marking the input word.

Hidden layer (weight vector W) is size of V x N - size of Vocabulary x Dimensions of vectors. For example 10.000 x 300. It is initialized to random values.

As we know, in neural networks, between fully connected layers of size 10.000 and 300 units, the weight matrix W will be of size 10.000 x 300. This weight matrix W will be the word vectors, where each row corresponds to one word in vocabulary and the 300 values in the row are values for the 300 dimensions of its word vector.

If we have vocabulary of 10.000 and have chosen 300 dimensional word vectors, the weight matrix W will be of size 10.000 x 300. Notice that when multiplying a matrix of 10.000 x 300 with an input of 1-hot vector, in the result all rows but one will have zero values. So in a sense the 1-hot vector where one word is 1, is just selecting that words row: 1x300 features from the weight matrix W of "word vectors".

Hidden layer in the model will be the context words from input in embedded form.

Output layer is again vector of size Vocabulary size V x 1, to represent the current word to be predicted by model. For the final layer a softmax distribution is produced over all words in vocabulary.

Training with multiple context words

Figure 2 expands simplified CBOW of one word context in figure 1, into one with multiple context words. Similarly as before, each input word coming in as 1-hot vector will be turned into its vector representation of 1x300. Then these vectors will be summed together to form the context.

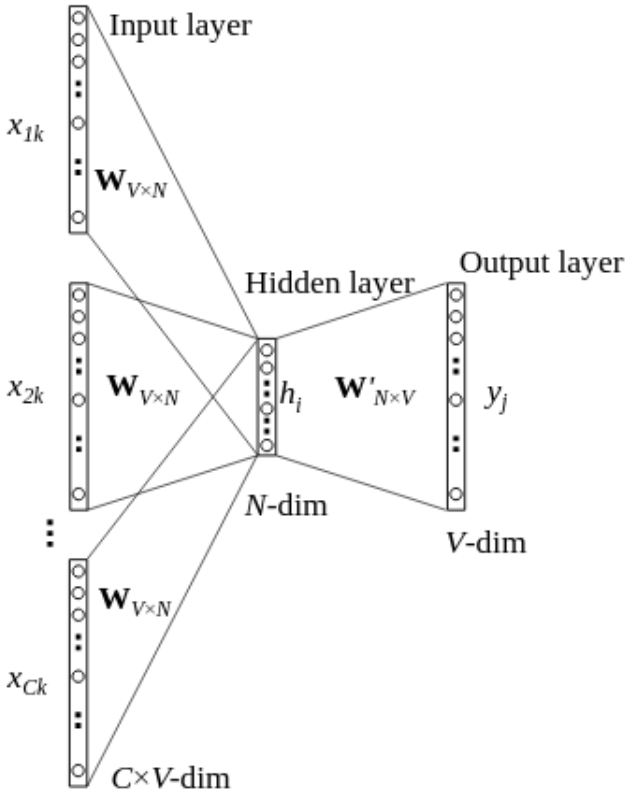

Figure 2: CBOW Xin Rong - Word2vec Parameter Learning Explained.

Now when context is more than 1 words and represented as sum of their word-vectors, we see why another weight matrix W' is needed. It is used to compare the middle layer representation into Vocabulary, trying to predict which of them is the correct middle word for the context.

In each training step, context to word, the weight matrices are updated with back propagation to slightly more accurate values.

Later the vector W' is discarded and only W kept.

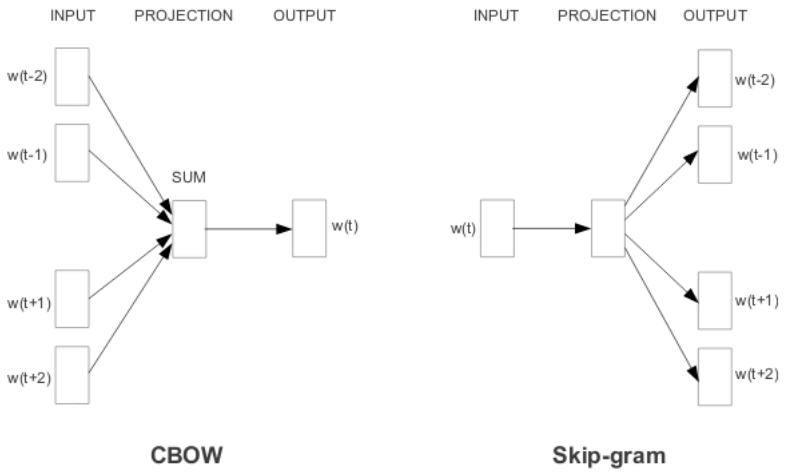

Figure 3: The CBOW architecture predicts the current word based on the context, and the Skip-gram predicts surrounding words given the current word. Image/text: Mikolov et al. 2013 [MCCD13].

Continuous Skip-gram

While CBOW uses context words to predict current word, skip-gram [3] instead uses current word as input to predict the context words. Authors state that increasing the range - how far previous and following words are taken into account, increases accuracy, but also increases computational complexity. They set the max range C=10, that is 10 words context on both sides of current word, giving a max context windows of 20.

Window size is defined by C, a maximum distance of words from current word. If C is selected as C=5, for each training a random number R will be selected from [1..5] and R words from history and future are used as target to be predicted. So if random R gives 2, the context will be 2 previous and 2 following word, that is 4 words context. If R gives 3, then context will be 6 words. As minimum R is 1, the context will always include the very previous and very next word. The further the word while still in context, the less often it will be included. This way the algorithm will weight nearby words higher.

The clever thing in skip-gram is, when weighting with this kind of random sampling, it will in effect choose nearby words more often and feed them more often as training samples, but as the sampling is done outside NN, the actual training step with NN

[3]Name skip-gram refers to n-gram, skip-gram being extension of it.

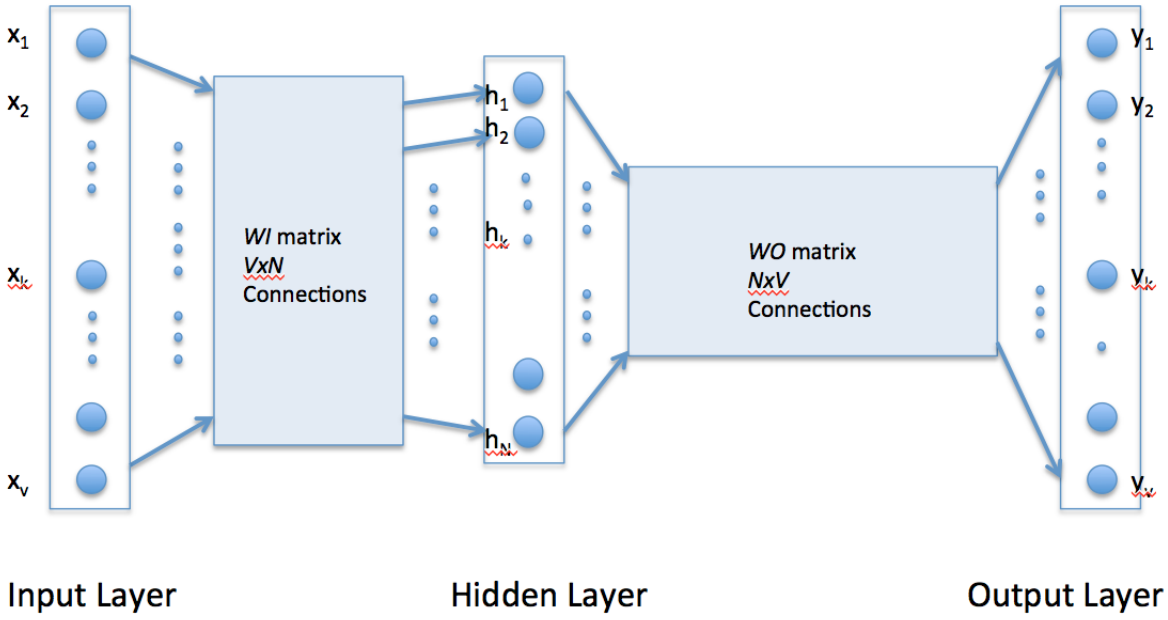

Figure 4: There are 2 weight matrices. One for hidden layer, size VxD: words x dimension of embedding. 2nd for output layer, size DxV. Image: Krishan [Kri15] https://iksinc.online/tag/skip-gram-model/

is done without any info about location of words - avoiding the high computational cost.

Comparison of CBOW and Skipgram

On surface, Word2vec uses context to predict middle word, and skipgram uses the middle word to predict context.

One difference is that Word2vec sums the context vectors first together and only uses this combined vector for prediction. While Skipgram in concrete level makes the predictions as 1-1 word pairs [gh].

If we consider how Skip-gram works, sliding a current word, and then picking a context around it, turning it into 1-to-1 word pairs for prediction, we could actually change the direction here - instead of using current word to predict context word, we could as well use the context word to predict current word - similarly to CBOW, and this would still be skip-gram.

While often the difference of CBOW and skip-gram is given as different direction in

prediction - current words vs context, it seems the important differences are, that skip-gram is using random sampling to give weighting to words by their distance while CBOW is not. And that while CBOW is combining the whole context into one vector before predicting - and with the sum operation, discarding some information, skip-gram is doing predicting on 1-to-1 word pairs without combining.

One effect of this difference - skip-gram feeding each context word individually and CBOW combining several words together, is that skip-gram may reach better performance for rare words, as CBOW might never train with the word itself only, but always as combined with vectors of other words.

Extensions to skip-gram

Also in 2013, Mikolov et al. represented some extensions [MSC+13] to improve efficiency and accuracy of produced Word2vec vectors.

These improvements were subsampling, negative sampling and phrase detection.

Subsampling of frequent words

Subsampling of the frequent words brings a significant speedup. In this, most frequent words in the training set are discarded with high probability during training. The reasoning is that very frequent words such as "the" appear very often around almost all the words, while bringing little information about those words. So after using them enough times for training the words own vector, eg the vector for word "the", after millions of trainings, it is no more useful to increase the training times.

In practise they used a formula that uses preset threshold t for frequency (t being around 10^{-5}), so that words being of higher frequency than the threshold, are subsampled aggressively, while words of frequency of below the threshold are mostly used.

Negative sampling

Where subsampling reduced amount of input samples to the NN, negative sampling is used to reduce computation in the end side of the NN.

In basic version the weights are tuned to slightly increase the softmax value for the correct predicted word. And weights for all other words (ie "wrong" words) in vocabulary are tuned to slightly reduce their softmax value. When the vocabulary size V can be 60.000 or in practise even over 1 million, this is a heavy computation on each training step.

In negative sampling not every wrong word in vocabulary is updated per training

step, but only small sample of them. The real predicted word, ie positive sample is always kept. In addition a small amount of negative samples (wrong predictions) are sampled randomly to be updated.

Phrase detection

Another extension was, that phrases of two or more words can be represented with single token. As an example 'Pittsburgh Penguins' or 'New York Times'. The phrases are discovered before the actual training of vectors, by using word counts - finding words that appear frequently together, but infrequently in other contexts.

2.7.2 Glove - global vectors

After Word2vec in 2013, another kind of embedding method was introduced by Pennington et al. in 2014: Glove - Global Vectors [PSM14].

While Word2vec trains vectors by trying to predict them and algorithm works by sliding a window and only working locally, within that window, Glove is based on counting statistics. Authors state that Skip-gram, used in Word2vec has a weakness due to using local context. In Glove they instead use global word to word co-occurrence probabilities.

Intuition behind Glove - Example

Pennington et al. give an example how some aspects of meaning can be extracted directly from co-occurrence probabilities:

Consider words "ice" and "steam". The relationship of these words can be studied by comparing the ratio of their co-occurrence probabilities with various other words, such as "solid". We know that ice is solid, but steam is not. When comparing the ratio of co-occurrence probabilities, the ratio P(ice | solid) / P(steam | solid), the result is large - larger than 1. This suggests that "ice and solid" are more related than "steam and solid".

For the word "gas" the ratio would be other way - P(ice | gas) is low, while P(steam | gas) is high, so ratio P(ice | gas) / P(steam | gas) goes smaller than one, toward zero - suggesting that steam and gas are related more than ice and gas.

Still they mention, that for words such as "water" - related to both words, or "fashion" related to neither of the words, the ratio should be close to one. [PSM14]

Practical usage

In practise, pretrained Word2vec and Glove vectors often show performance very similar to each other. More difference coming from the chosen dimensionality, than between the two models. The performance can also depend on specific field of text used on, so testing both for a specific purpose can be favorable.

It is also good to notice that the format of trained vectors of same dimension is identical between word2vec and Glove, so using either one or changing between them is quite straightforward. One do need to notice though, that the vocabulary is likely to differ. One way to increase covered vocabulary is to combine word2vec and glove vectors by summing the word vectors together and joining their vocabularies.

2.7.3 Paragraph Vectors - Doc2vec

Le and Mikolov represented also an extension of Word2vec, that is called Paragraph Vectors [LM14] (also called Doc2vec). Name emphasises that it can transform any length text into a fixed size feature vector - a phrase, a sentence, a document.

Paragraph Vectors, ie Doc2vec trains both, word vectors and Paragraph Vectors, with two algorithms that are modifications from CBOW and skip-gram. Similarly as in Word2vec each word is still mapped into a unique word vector, represented by a column in matrix W. In addition, every paragraph is mapped into a unique vector, represented by a column in a new matrix D. For predicting next word, the paragraph vector and word vectors are then combined with either averaging or concatenation.

2.7.4 Fasttext

Wordvectors that we previously saw - Word2vec and Glove start by mapping each word into unique integer, then continue to build the vectors. They don't care what letters the word includes or how it is written - they purely rely on context - other words occurring with a specific word.

In 2015 was introduced Fasttext [BGJM16], where viewpoint is to take into account the **morphology** of words. That is, words tend to be formed from meaningful smaller parts, that convey information. Fasttext enriches word embeddings with subword information by additionally learning embeddings for character n-grams.

By using a distinct vector representation for each word, original skip-gram ignores

the internal structure of the words. Fasttext presented a new approach based on the skipgram model, where **each word is presented as a bag of its character n-grams**. Each n-gram (subword unit) is represented as a vector and the whole word is the sum of its character n-gram vectors.

They add special boundary symbols "<" and ">" to beginning and end of sentence, to distinguish prefixes and suffixes. They also include word itself in the set of its n-grams, to learn a representation for each word also.

Bojanovski et al. offer an example with word "where" and n-gram of n=3. The word "where" will be represented by the character 3-grams:

$<wh, whe, her, ere, re>$ and a special sequence $<where>$.

Word is represented by the sum of the vector representations of its n-grams. This means above summing together representations of each 3-gram.

The sequence "<her>" corresponding to the word her is different from the trigram "her" in the word where. Here the boundary symbols come important.

In practise they extract all n-grams $n \geq 3$ and $n \leq 6$. Alternative could be to take all prefixes and suffixes for example.

This learning of representations for n-grams allows sharing the representations across words. This enables Fasttext to give embedding also for new, unseen words. If word is unseen before, its subword units will be used to create an embedding.

Vector representation for sentence

Same way as character n-gram vectors of a word can be summed together to get a vector for the word, also **vectors of all words in a sentence can be averaged together, to form a vector representation for the whole sentence** [JGBM16].

In addition to Fasttext algorithm, Facebook also released in 2015 Fasttext library that offers pre-trained models for 294 languages.

2.8 Combining multiple word embeddings

One limitation of word vectors is the **Out Of Vocabulary - OOV**. Facing words that were not included in the dictionary while training. One way to raise the word coverage is to combine more than one word embeddings. As an example one can take 300-dimensional Glove and 300-dimensional Paragram vectors and for each word in own dictionary, assign its word-vector the average (or weighted average) of the Glove

and Paragram vector for that word.

In practise this means that if a word is found even only in one of the used models, it will still be assigned a reasonable value (that is the found vector itself). As such, the word coverage available will be the union of the combined word vectors.

As a result this will give word vectors with similar dimensionality to original, with roughly same computational complexity in usage.

In Kaggles 2019 "Quora Insincere Questions Classification" competition, with external data forbidden and 4 different types of word-embeddings provided, the winner solution included combined (by sum) Glove and Paragram embeddings with weights of 0.7 x Glove + 0.3 x Paragram [Sin19]. (Which they fed into LSTM and concatenated with statistical features).

Above competition had GPU time limited to couple hours. With less limitations on processing time, one high accuracy option is to use multiple embeddings, giving each embedding its own LSTM-line that works as a feature extractor and then concatenating these features from different embeddings as one input to NN.

Chapter summary

Above we saw how to do represent documents as a vector, counting words or n-grams in them or using TF-IDF. Also we saw how to deal with curse of dimensionality, by using preprocessing: turning words into common basic forms with stemming or lemmatization and reducing vocabulary size.

Then we saw how to represent words with continuous word embeddings, that produce a representation for words by the context words they appear with. And that word embeddings measure word similarity, allowing us to possibility to give new unseen word a meaningful representation.

We saw different methods how to produce word vectors: Word2vecs alternative algorithms: Continuous BOW and Continuous Skip-gram and their differences. How to reduce computational cost in training by subsampling frequent words, negative sampling only small amount of wrong target words, and how to also create vectors for phrases by detecting them before the creation of embedding.

Gloves method based of counting global co-occurrence matrix.

We also saw how with Doc2vec it is possible to transform longer texts into document vector, by summing or averaging together the vectors of individual words in the text.

We saw how Fasttext takes advantage of subword units and character n-grams con-

tained in the words to learn representation and meaning for units smaller than word and as such, allowing transfer of information on a smaller level.

2.9 Limitations of Context free Word embeddings

Word embeddings brought a big improvement. Yet some limitations remained with word embeddings.

Same word can have multiple senses - Polysemy A word can have different senses, ie. meanings. In the following 2 sentences, word "bank" is written same, but has different meanings. "He walked at the river bank", "I took the money to the bank". The meaning is defined by context, but as word vectors create only one embedding per word - one dot in vector space, the word "bank" also will have just one embedding. In practise this embedding will become sort of weighted average over different senses of that word. If we would form one embedding by its context just for the "river bank", and separate for "bank" as financial institution, the embedding for "bank" will be placed between these 2 meanings in vector spaces. (and possible other meanings of "bank" could move the point to yet another direction.

Another example is word "interest". The meaning and location of word embedding will be located somewhere between "interest for loan" and "interest in soccer",

There is also an extension to word embeddings for dealing with polysemy - Multi-sense embedding. They use WSD - Word Sense Disambiguation, before training. But we don't go further into that here.

Meaning of word can change over time Sometimes even roughly same sense/meaning of a word can change to some extent over time. The meaning of airplane at 1890 and airplane at 2010 can be quite different. Perhaps latter is used commonly for commercial passenger transportation, taking possibly large amounts of passengers - while the previous did not. Often there can be information in the context - the older text might have different words also as context, that might contain information to separate between different shades of the word. Again - context free word vectors

These limitations come back to the cause, that word embeddings are **context free.** They only store one embedding per word, and this one embedding has to represent possible multiple senses of the word.

For completeness it is good to mention that there do exists versions of word embeddings, that can have more than one representation for same word in different

contexts. Yet these brought more complexity and problems and did not become popular in practise.

Context sensitive embeddings - with language models

As we will see, a way to overcome this limitation, of word having only one combined representation for multiple senses, turned out to be **Language Models.** Because the language models can learn to represent a word not context free, but within context. So same word can have different embedding in its different senses, or different time in history, due to context words being different.

In the next, we will see how to overcome these limitations and still improve word embeddings by creating contextual embeddings. That is - giving words a different representation depending on each specific context. That is done using Language Models.

3 Language Models

3.1 Language Model - a probability distribution over text

On word level, language model can be defined as a **probability distribution** over sequences of words. (Usually over limited size, predefined vocabulary.) In other words, the language model assigns a probability to every document in the language. Following equations and definitions are based on Jurafsky et al. [JM19].

Given a sequence of words as a start, it can give conditional probability distribution of the next word, over some predefined vocabulary. For example with vocabulary of size 60.000, given start "It was", the LM probability distribution can give probability for each of the 60.000 words for being the next. Probability for each of 60.000 words being between [0..1] and their sum totalling 1. (Special token UNK for unknown words - those not in vocabulary is often included in the vocabulary with some probability. Also end of document -token can be added, with also having some probability)

The probability of a specific sequence can be estimated as a product of conditional probabilities of the words:

$$P(sequence) = P(w_1, w_2, ..., w_m) = P(w_1)P(w_2|w_1)...P(w_m|w_1, ..., w_{m-1}) \quad (1)$$

In addition to word level, language model can also work on character or subword

unit level. A character level LM for example predicts next character in the sentence, instead of next word.

More generally a language model can be defined as a probability distribution over linguistic units, where these units can commonly be for example words, characters, subword units or character n-grams or combinations of more than one type of units.

3.2 N-gram based language models

N-gram language models

N-grams can be used as a simple language model.

One simple N-gram LM is bigram, where n=2. We can train a bigram LM by going through and counting all bigrams - the 2 word pairs, in the training data. Given a word x, its following word is given probability based on how often words followed word x in training data.

N-grams language models are based on Markov assumption - that probability of a word at t+1 is based only on the preceding n-1 words.

With n-gram, the probability is conditioned only on the previous n-1 words

$$\begin{aligned} P(sequence) &= P(w_1, w_2, ..., w_m) \\ &= P(w_1)P(w_2|w_1)P(w_3|w_1, w_2)...P(w_m|w_{m-(n-1)}, ..., w_{m-1}) \end{aligned} \tag{2}$$

equaling

$$P(w_1, w_2, ..., w_m) = \prod_{i=1}^{m} P(w_i|w_{i-(n-1)}, ..., w_{i-1}) \tag{3}$$

Example of bigrams probability

'Tomorrow is going to be a fine day.'

bigrams: (tomorrow, is), (is, going), (going, to)...

We can estimate probability probabilities by using frequency counts. To get probability of next word 'is' given current word 'tomorrow', eg. P(is | tomorrow) we count number of the bigram (tomorrow, is) in the training data. This count we divide by all instances of tomorrow followed by any token ("word") (tomorrow, x).

$P(is|tomorrow) = Count((tomorrow, is))/Count((tomorrow, x))$

If all instances of word 'tomorrow' are followed by a token (such as . or !), ie they are 1st word in a bigram, we can simplify the formula into

$P(is|tomorrow) = Count((tomorrow, is))/Count(tomorrow)$

If there would be 10.000 instances of 'tomorrow' followed by any token, in the training data, and 120 of these would be (tomorrow, is), the probability of P(is | tomorrow) would be 120 / 10.000 = 0.012

Smoothing and backoff

Often new text is such that exactly same n-grams of words have not been in training data - especially with higher order n-grams. In such case, n-gram language model would give zero probability for large parts of new text, which is not favorable. One solution to this is to using **smoothing**, by giving all combinations of n-grams of given length at least a small probability. Same goes for new words, mapped to $<UNK>$ -token.

While adding probabilities > 0 for new event, total probability mass would increase bigger than 1. To fix this, we can keep old probabilities as they are, add the new ones and just **normalize** by dividing all probabilities with the hew total probability mass, setting total back to 1.

Another method is **backoff**. When for example specific 3-gram was not in training data: (the word given 2 preceding word) back to 2-gram, the word given only 1 preceding word. Would it still not exist, can back to 1-gram - individual word probability [JM19].

3.3 Generating new text using n-gram LM

Language model can also be seen as a generative model.

It is possible to generate text with LM by giving starting words, or empty start, pulling the probability distribution for next word, then choosing the word with highest probability - or often, picking a random word weighted by their probability. This way you don't always receive exactly same result, but different words and texts with variation.

We can also consider probability of whole sentences. Then probability of a specific sentence is a product of conditional probabilities of the words. $P(sentence) = P(w_1, w_2, w_3, ...) = P(w_1)P(w_2|w_1)P(w_3|w_1, w_2)$ etc.

(LMs probability distribution is an estimation of the real distribution. Or the joint probability distribution over **training text**.)

Beam search

Producing texts from LM by picking **greedy way** always the next word with highest probability, is not optimal and does not guarantee to produce the sentence with highest probability. **Beam Search** is one algorithm to search for highest probability sentence by choosing Beam width, such as B=3, then saving B highest probability suggestions for 1st word, finding B highest Prob 2. words conditioned on each suggested 1st word, then from these 3*3 2 word sentences, keeping 3 highest probability, and again for each, choosing B most probable 3rd word etc [Ng17].

3.4 Continuous language models

While in n-gram language models the words are represented in a discrete space, continuous space language models use continuous representations of words.

We saw earlier, in chapter 4, how to train context free word vectors with Word2Vec and its algorithms CBOW or Skipgram. Here the NN language model can use such embedding to encode input words into continuous vectors of dimension d. For example into a word vector of 300 float values. Compared to N-gram LM, this will help the model to recognize context, even when context words are similar, but not fully same as in training data.

Where discrete N-gram model has hard to learn generalization, with continuous valued vectors of words, the functions in NN can learn to generalize even when word vector is slightly different - to give good output even when input word vectors are not seen in training data.

In figure 5 is an example of simple continuous language model by Jurafsky et al. [JM19], which they call Neural Language Model - due to being trained by NN. The model uses feed forward NN.

In the example model, on each step through the text, context of 3 previous words is used and entered as input. If word vectors of dimension d of 300 is used, then each word is converted into 300 dimensional vector - using the weight matrix produced by Word2vec for example.

The word vectors of 3 context words (3 times 1x300) are concatenated together, to a 1xNd layer, that is 1x 3 x 300, ie 1x900 layer. As in training word vectors, here

also output layer is again softmax for 1 x Vocabulary size, giving a probability for each word in vocabulary for being the next word.

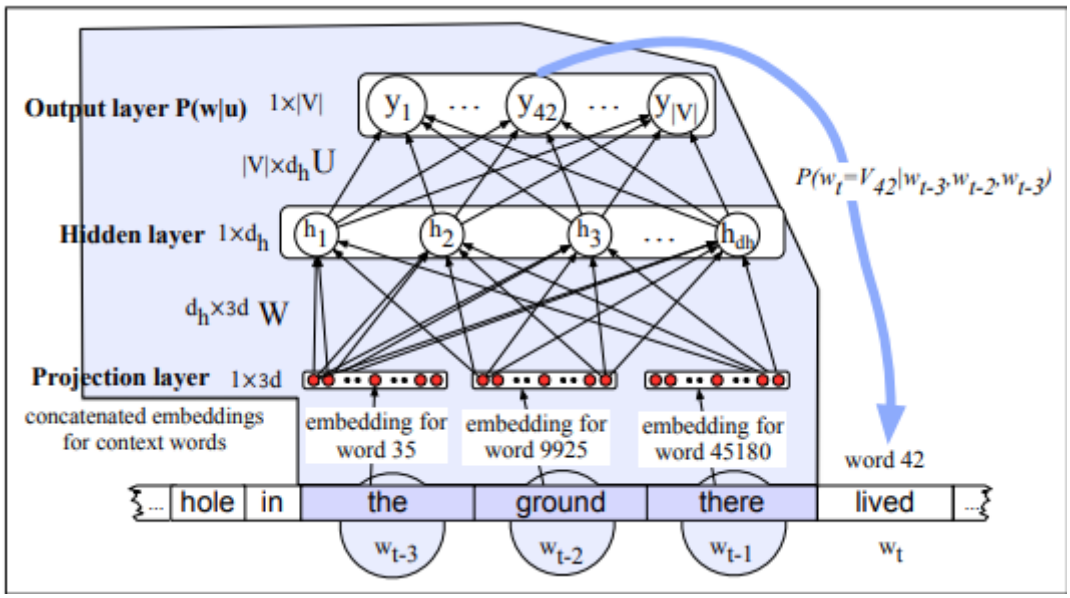

Figure 5: Simplified Neural Language Model (feed forward). Picture Jurafsky et al. [JM19].

Neural model as a function

Bengio et al. formalize Neural Model as task of learning a good model f, $f(w_t, ..., w_{t-n+1}) = \hat{P}(w_t|w_1^{t-1})$, so that it gives high out-of-sample likelihood [BDVJ03].

So we can consider that Neural Network implements this function f. Inputs being the context of n words, represented as integers in vocabulary V. Outputs being softmax over vocabulary V, giving a probability for each word in V being the next word, given the context of input.

They decompose this function f into two parts:

1. A mapping C from any element i of V to a real vector $C(i) \in R^m$. Representing the distributed feature vectors associated with each word in vocabulary.

This step 1 and mapping C is same mapping of each word (by index) in vocabulary into its word vector, that we saw above and what word2vec does.

2. The probability function g over words, expressed with a context C. A function g maps an input sequence of feature vectors (word vectors) in context, to a conditional probability distribution over words in V for the next word w_t.

Along this line they represent joint probability, decomposed as conditional proba-

bilities, using the function g(.) - represented by a NN.

$$\hat{P}(w_1, w_2, ..., w_m) = \prod_{i=1}^{m} \hat{P}(w_i | g(w_{i-(n-1)}, ..., w_{i-1})) \tag{4}$$

They mention that this implements "sharing parameters across time" in the way of same function g_i being used across time steps. Also C is shared across the words in the context [BDVJ03].

3.5 Evaluating LMs

Traditional ways to estimate LM include perplexity, cross entropy and bits pre character - BPC. Also LMs performance in downstream NLP tasks are increasingly used - such as GLUE.

3.5.1 Perplexity

One of the most common ways to estimate LM is perplexity. Perplexity does not involved separate end task, but is measured by predicting the words in real text, conditioned by beginning of it.

One big advantage of using Perplexity to compare LMs is that it gives a solid score without bias of end task. One source of possible bias being end task itself and possibly extra from errors and mislabeling in annotation.

For counting perplexity, a test sequence can be given to the language model, letting it predict each next word or token, given previous ones. That means, the language model will produce a probability distribution for each next word, and then can be seen, how high probability it predicted for the actual word that came next.

Perplexity can be thought as **how surprised the model is on average, on the words it sees.** This means, that a lower value of perplexity is better - the model is less surprised.

Perplexity: simple example of 6-sided dice.
If rolling a normal 6-sided fair dice (uniform distribution), a roll gives one number out of 6, with equal probability. In this case the probability-distribution as outcome is 1/6 or 0.166 for each number. One roll of a fair 6-sided dice has perplexity of 6.

Entropy is expected measure of bits required to represent info or to encode the

outcome of random variable, at smallest. One throw of 6 sided dice can be encoded with x bits, $2^x = 6$. Here x being 2.58. $2^{2.58} = 6$

On above equation, 2.58 is **number of bits required** to encode the information, ie 2.58 is **entropy**. **Perplexity is the exponentiation of the entropy.**. Perplexity above is 6.

Similarly tossing a coin for heads or tails, with 2 sides, will give a result 1 out of 2 with equal probability and have a perplexity of 2 per throw.

In a special case where event p models fair k-sided dice (a uniform distribution over k discrete events) its perplexity is k.

Perplexity of a sequence

We saw perplexity of single throw of dice or coin. How about perplexity of repeated throws?

$$PP(W) = P(w_1, w_2, ..., w_N)^{-1/N} \tag{5}$$

In equation 5 - Perplexity of a sequence of words, is the product of probability of each word, raised to potens of -1/N. N being size of vocabulary?. Because the formula is taken as inverse, the higher the conditional probability of the word sequence, the lover the perplexity.

We know that perplexity of a 6-sided dice is 6 per throw. If we throw the dice 3 times, we will get

$$PP(W) = P(w_1, w_2, w_3)^{-1/N} = P(6 * 6 * 6)^{-1/N} \tag{6}$$

Normalizing perplexity by number of words

Since longer sequences naturally produce higher perplexity, to make perplexity comparable, it is common to normalize the perplexity by number of words. If the target text has 34 words, one would divide total perplexity by 34 to get **perplexity per word**.

Example of text perplexity: Brown corpus

Brown corpus, compiled in the 1960s, contains about 500 English text samples, totalling around 1 million words. The lowest perplexity published for Brown corpus was 7.95 at 1992. (cite Wiki)

If text of 1000 words could be coded with total of 7.95 bits per word, that would

give model perplexity of $2^{7.95} = 247$ per word [4]. This means model is as confused on each word as if it would have to choose uniformly and independently from 247 possibilities for each word.

With more modern models, likely a lower perplexity could be reached.

N-Gram language models

Perplexity for a test set $W = w_1 w_2 w_N$ can be counted as shown in equations 7, 8 and 9 [JM19].

$$PP(W) = P(w_1, w_2, ..., w_N)^{-\frac{1}{N}} \tag{7}$$

$$PP(W) = \sqrt[N]{\frac{1}{P(w_1, w_2, ..., w_N)}} \tag{8}$$

$$PP(W) = \sqrt[N]{\prod_{i=1}^{N} \frac{1}{P(w_i|w_1, ..., w_{i-1})}} \tag{9}$$

Differences in test setting

When comparing different LMs with each other, it is important to notice differences in the setting: preformatting, vocabulary size etc. If one LM has bigger vocabulary than other, it is predicting form larger set on each step.

Also a perplexity for character level LM can be lot smaller than for a word based LM, if the character based model is guessing mostly from 26 letters of English alphabet plus few special characters it will get a lot smaller average perplexity than a LM guessing from word level vocabulary of 60.000 - or even over 1 million words. Subword units still give a different setting.

Measuring the UNK-token. If one model has smaller vocabulary and more unknown tokens, its predicted probability for UNK will be higher, and it will also more often face UNK. The combined probability for UNK may well be higher than other LMs probability of a rare real word.

A perplexity can and likely is different for different kind of texts. Knowing what would be a "good" perplexity for a certain text is not easy.

If comparing perplexity of LM with test set for example from Wikipedia, it is good

[4] `https://en.wikipedia.org/wiki/Perplexity`

to be aware if a LM has had pretraining data from Wikipedia - possibly including some of the test set.

3.5.2 GLUE benchmark - General Language Understanding Evaluation

While perplexity measures the language model itself - how well it predicts the text, there is also a need for evaluating LM:s performance in downstream tasks.

GLUE is a benchmark for evaluating model performance on NLU tasks [WAS+18] and includes 9 tasks of natural language understanding, with data sets, and the GLUE-score will be average score on those 9 tasks. They include for example Named entity recognition, Textual entailment (which of 2 sentences follows the other), Coreference resolution (eg. to which object in the sentence "it" refers to) and Binary classification of movie reviews: positive or negative.

While Perplexity is a measurement on LM with the data itself, without need for label, GLUE is a measurement for 9 end tasks, some of them with labels and some without.

In addition to these there is also a qualitative evaluation, where a human could read for example text produced by a LM, and evaluate how good it is.

3.5.3 Natural Language Decathlon

In 2018 Socher et al. from Salesforce introduced Natural Language Decathlon (decaNLP), a benchmark for general NLP, consisting of 10 different NLP tasks. The score from the benchmark is combination of scores from all 10 tasks [MKXS18].

They also introduced their system for the tasks: Multitask Question Answering Network (MQAN). For the interface their idea is to frame all tasks to input as Question Answering (question, context, answer).

"What is the summary?" "Text" "Answer"
"What is the translation from English to Germany" "Text" "Answer"
"Is this sentence positive or negative" "Text" "Answer"

While their scores in tasks don't match the SOTA of systems made specially for a single task, the capability to perform multiple different tasks seems promising.

(*The 10 tasks are question answering, machine translation, summarization, natural language inference, sentiment analysis, semantic role labeling, zero-shot relation

extraction, goal-oriented dialogue, semantic parsing, and commonsense pronoun resolution.)

As we will later see, language model BERT uses two tasks for pretraining: predicting masked word and also predicting if, from given two sentences, the one appears in the text right after the other or not.

3.6 Different Token levels

3.6.1 Word level Tokens

Vocabulary size In Word level LMs the computational complexity of softmax function grows along with the vocabulary size. Softmax is used for counting probability distribution, giving probability for every word in vocabulary. Because of this, the size of vocabulary is usually limited to something like 30.000 most common words.

Down side of this is that while using the trained model, rare words and mistyped words will not be found in vocabulary, but instead replaced with OOV token. (out of vocabulary). A problem that character-level LMs don't have, since number of different characters is considerably smaller than number of different words.

There have been suggestions to lessen the computational cost of softmax by using versions that instead of counting softmax for whole vocabulary, only count an estimate. Such as AdaptiveLogSoftmaxWithLoss in Pytorch.

3.6.2 Character level Tokens

Character level embedding is more capable to handle unseen words, mistyped words and twisted forms as word. Downside is its requirement of lot more computation steps for NN model, since there are lot more characters in a given text, than words.

3.6.3 Subword level Tokens

With word level embedding remains the problem that new unseen words or misspelled words will be out of vocabulary - OOV, and as such they won't have own embedding.

Character level models solve the OOV problem to some extent - every new word will have embedding, since it can be built from combining embeddings of individual

characters. Yet the number of tokens in a character level model gets large and as such, the processing can be computationally expensive.

In addition to this, also the "sub parts" of a word often contain information that will be missed by word and character level models.

To address these problems, between the granularity of single characters and whole words, there are subword units.

"Subword-based models are a promising approach, since morphology often reveals the semantic category of unknown words: The suffix-shire in Melfordshire indicates a location or city, and the suffix-osis in Myxomatosis a sickness. Subword methods aim to allow this kind of inference by learning representations of subword units, such as character ngrams, morphemes, or byte pairs. [HS18]"

BPE - Byte Pair Embedding

Byte pair embedding was introduced in 1994 by Gage [Gag94]. It was originally for compression, merging frequent pairs of bytes. In 2016 it re-emerged, but this time for merging characters of character sequences [SHB15].

BPE Algorithm, simplified

1) Define wanted subword vocabulary size
2) split text to individual characters plus end of word token </w>. Count their frequencies.
3) always select most common subword pair and generate a new subword as combination of those two.
4) repeat step 3 until max subword voc size is reached

So the first subwords are single characters and first combined subword will consist of the 2 characters appearing most often together.

WordPiece

WordPiece is a subword algorithm, similar to BPE. Difference being that where BPE chooses always to combine two subwords with highest frequency in the data, the WordPiece instead builds a language model and when choosing what subwords to combine, chooses those that most increase the likelihood to generate the training data.

Unigram language model

Unigram language model is based on a probabilistic language model. Unigram language model makes an assumption that each unigram is independent of those

around it. Probability of a subword sequence is counted as the product of the subword occurrence probabilities.

SentencePiece

While many subword segmentation tools assume that the input is pre-tokenized into word sequences, SentencePiece [KR18] can train subword models directly from raw sentences, which allows to make purely end-to-end language independent system.

SentencePiece is also a tokenizer that implements subword units, BPE and unigram language model [5].

Common subword based models are **Byte Pair Embedding** and **Unigram Language Model** These are implemented for example in the SentencePiece tokenizer.

Possible pre-tokenization

There is a difference between segmentation on whether text is split into tokens before subword creation or not. This will make a difference on how spaces and split between words are handled.

(In some of the methods, the process of transforming text into subwords (or words, characters alternatively) **happens in preprocessing**- before feeding the training data to the model.)

In normal BPE the text is tokenized first and then fed to subword model. Another way that SentencePiece does is to avoid the tokenization step and process the raw text directly.

Especially the tokenization can be language specific, requiring different ways for different languages. In case of some languages with not easy tokenization, training sub word model directly from raw sentences can simplify one step.

With language models, for English language, Spacy tokenizer is commonly used for tokenizing text. For agglomerative languages like Finnish, Sentence-Piece tokenizer works better.

Text preprosessing for LM:s

With language models the direction is towards keeping text in its original form, with argument that different shapes of the word, capitalization etc contain information, which is lost if removed. LMs often have shown to have capability to use this information and still manage recognizing similarity of same word in slightly different forms.

[5] `https://github.com/google/sentencepiece`

In ULMfit, capitalization was treated with a solution to remove it from main word, but add a special tag marking capitalization before the word. And as such reaching both, making word recognizable by vocabulary + keeping info about capitalization.

Summary

In this chapter we saw how to define a LM as a probability distribution over language units.
We saw different token levels: words, characters and subwords, used as language units to build LM over.
We saw how to generate new text using LM.
We saw how LM can be implemented without neural networks, based on N-grams.
We also saw ways for evaluating LM:s, perplexity and GLUE benchmark.

4 NN-based Language Models - Evolution

In this chapter I will look at specific language models, their technical choices and how they differ from each other.

Table 1 list evolution of specific language models: time of publication, architecture they were based on and training data used.

Published	Model	Architecture	Trained on	Params	Ref
Aug 2017	CoVe	LSTM + Attn	ger to eng[6]	-	[MBXS17]
Oct 2017	ELMo	Bi-LSTM	1B Word Benchm	-	[PNI+18]
Mar 2018	Univ Sent Enc	Transformer	-	-	[CYK+18]
Mar 2018	Univ Sent Enc	Deep Avg Network	-	-	[CYK+18]
1-5 2018	ULMFit	AWD-LSTM	Wikitext-103	-	[HR18]
Jun 2018	OpenAI GPT-1	Transf-decoder	BookCorpus	-	[RNSS18]
Nov 2018	Google AI BERT	Transformer	BookCorp + Wiki	340 M	[DCLT18]
Feb 2019	GPT-2 Small	Transformer	WebText 40GB	117 M	[RWC+19]
Feb 2019	GPT-2 Large	Transformer	WebText 40GB	1.5 B	[RWC+19]

Table 1: Evolution of language models.

[6]* Cove did not actually use language modeling as a task, but machine translation.

Model	Architecture	Trained on	Parameters	Fine-tune
ULMFit	AWD-LSTM	Wikitext-103	-	yes
OpenAI GPT-1	Transf.-decoder	BookCorpus	-	yes
Google AI BERT	Transformer?	cWikipedia	340 M	yes
GPT-2 Large	Transformer?	WebText 40GB	1.5 B	no
GPT-2 Small	Transformer?	WebText 40GB	117 M	no

Table 2: Main Models

4.1 Background - Attention & Transformer

Attention and Transformer are core pieces of many LM architectures. Attention used in modern NN:s was presented by [VSU+17].

Attention can be split into 2 classes - soft and hard. In soft attention all target values are processed with some weight. In hard attention some target values are left without attention. In practise the attention used is soft one - all input is processed and given attention weight for. Figure 6 shows attention for each input word, when encoding the word 'it'. Darker color means higher attention. In multihead attention there are multiple parallel attention heads. In this case picture shows the attention weights for attention head #5.

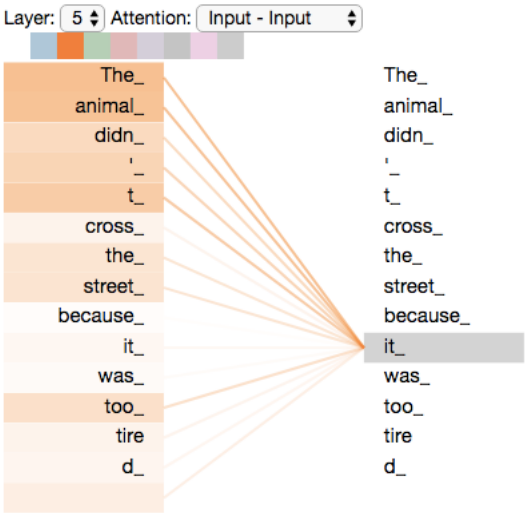

Figure 6: Attention for input words, when encoding word 'it' Image: Illustrated Transformer [Ala18] http://jalammar.github.io/illustrated-transformer/

Weng [Wen18] describes Attention as following:

In BiRNN encoder,if hidden state at step i: h_i is concatenation of forward hidden state i and backward hidden state i, $[forw\ h_i, backw\ h_i], i = 1...n$

Then the decoder has hidden states $s_t = f(s_{t-1}, y_{t-i}, \mathbf{c}_t), t = 1...m$

Where context vector $\mathbf{c}_t$ is a weighted average of all source hidden states h_i: a sum of the hidden states of input sequence weighted by ("alignment") attention scores.

Attention can be additive or multiplicative.

Key parts of attention: Query, Key and Value are all **abstractions.** These are pictured in Figure 7.

Input words embeddings are multiplied by Weight matrix. All of the weight matrices are of same form.

The score is calculated by taking the dot product of the query vector with the key vector of the respective word we're scoring. So if we're processing the self-attention for the word in position #1, the first score would be the dot product of q1 and k1. The second score would be the dot product of q1 and k2 [Ala18].

For computing these we have Weight matrices WQ, WK and WV.

As an example, sentence starting with "Thinking machines..." Embedded version of each word is used as input Xn.

Embedding of word "Thinking". Dimension 512 This embedding X1 is multiplied by 3 Weight Matrices, WQ, WK and WV, to produce 3 outputs: Q, K and V. All of the Q,K and V are of same dimension.

Output/Value of Query, Key and Value are dimension 64 of this case, which are smaller than the original 512-embedding of a word!

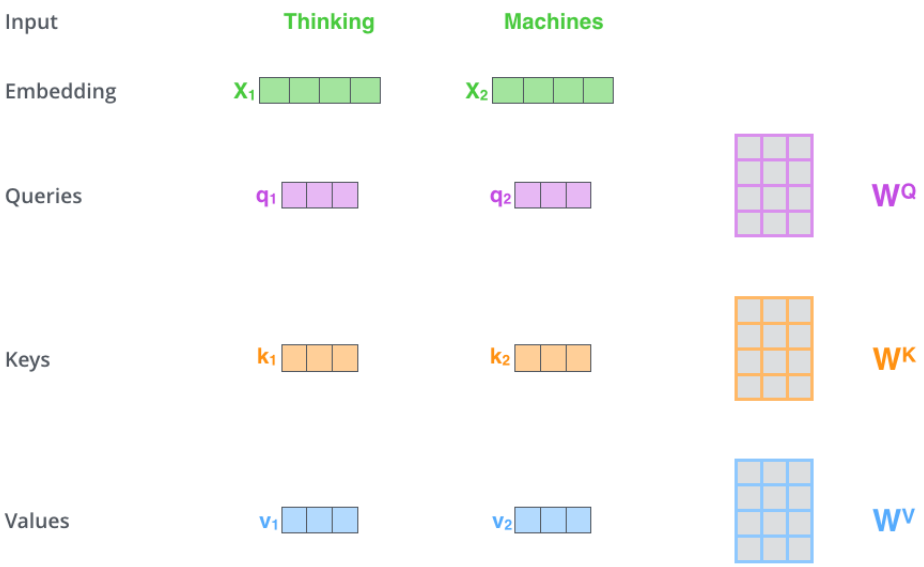

Figure 7: Vectors for Query, Key, Value. Image: Illustrated Transformer http://jalammar.github.io/illustrated-transformer/

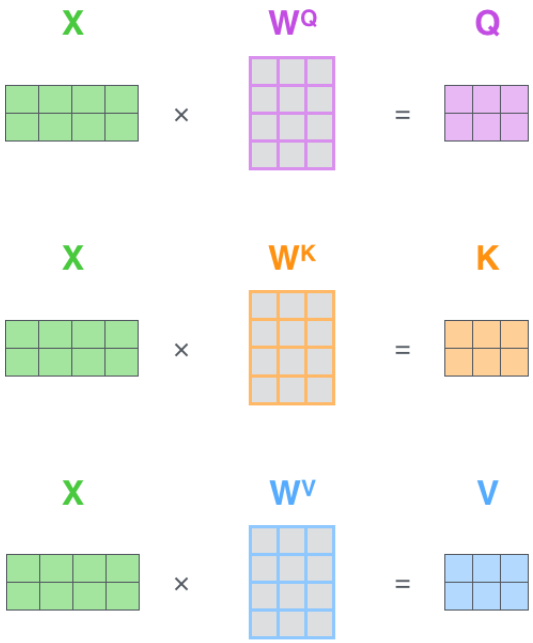

Figure 8: Calculating K,V,Q. "Every row in the X matrix corresponds to a word in the input sentence. We again see the difference in size of the embedding vector (512, or 4 boxes in the figure), and the q/k/v vectors (64, or 3 boxes in the figure)" Image: Illustrated Transformer [Ala18] http://jalammar.github.io/illustrated-transformer/

Multi-Head Attention

A set of Q,K, V is one Self-attention. This can be expanded by using multiple Attention-heads, for example 8 units.

This allows each attention-head to focus on different feature.

Figure 9 shows how attention is computed with softmax from K,V,Q.

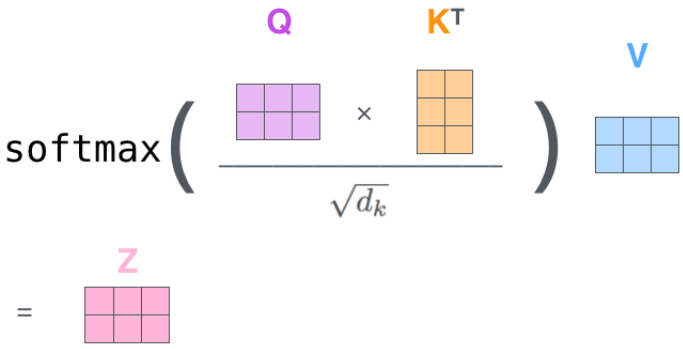

Figure 9: Calculating attention. Image: Illustrated Transformer http://jalammar.github.io/illustrated-transformer/

Transformer

Transformer is a sequence2sequence based model, consisting of encoder and decoder. It takes an input sequence and maps it into higher dimensional space. This vector is then given to decoder, to produce and output sequence.

Transformer - as name suggests, transforms input sequence into output sequence, for this, using the attention mechanism.

4.2 CoVe - Contextualized word vectors

While previously common was to use context free word vectors such as Glove, in 2017 McCann et al. [MBXS17] used the sequence of Glove-embedding for words as input to Bi-LSTM to produce **contextual word vectors**.

If w is a sequence of words and GloVe(w) the corresponding sequence of word vectors produced by the GloVe model, then sequence of the context vectors is:

$$CoVe(w) = MT - LSTM(Glove(w))^7 \tag{10}$$

[7]MT in MT-LSTM refers to Machine Translation, that CoVe used as pretraining task

Then for task such as classification and question answering **the context free vector and contextual vector are concatenated**. To get embedding for each word the word vector Glove(w) is concatenated with its corresponding contextual vector CoVe(w).

$$\tilde{w} = [GloVe(w); CoVe(w)] \tag{11}$$

The LSTM-model used by CoVe was an attentional sequence-to-sequence model trained for machine translation from english to germany. 2-layer BiLSTM as encoder, initialized with Glove vectos. They called this model the MT-LSTM, MT from Machine Translation as pretraining task. And 2-layer LSTM with attention as decoder.

Outputs of pre-trained encoder are used as context vectors.

(Machine Translation needs to use context of the words and for example disambiguate between different meanings of same word, by the context. It also needs to understand some long range connections. As such, these features are encoded by the MT-LM.)

Their largest data set though is 7 million sentence pairs. With no continuity between sentences.

4.3 ELMo - Deep contextualized word representations

Later in 2017 were introduced the ELMO-embeddings: ELMo - Embeddings from Language Model [PNI+18]

ELMos idea was to extract embeddings from multiple layers of the LSTM-model. They show that different layers capture different information and using more than final layer gives higher performance.

"Character based: ELMo representations are purely character based, allowing the network to use morphological clues to form robust representations for out-of-vocabulary tokens unseen in training."

"ELMo word representations are functions of the entire input sentence. They are computed on top of two-layer biLMs with character convolutions"

ELMo embedding
Elmo produces embedding for each word, using whole input sentence as a context.

1. concatenate representation for a word from different hidden layer
2. multiply by weights based on the task
3. Sum the now weighted vectors

Output is an embedding for the input word in given context.

Dataset
They train model on the 1B Word Benchmark corpus. One limitation of this data set is that sentence order has been randomized. As such, the learning of long distance relations within text is limited inside single sentence. Previous or next sentence in training data are not related to current one.

Pre-trained vectors are concatenated with (task specific) ELMo vectors from the BiLM

$[x_k; ELMo_k^{task}]$

Different layers of BiLM have different information of a word. Cove trains on supervised language translation data, ELMo trains on unsupervised LM data.

Authors stated in 2017 that: "Adding ELMo to existing NLP systems significantly improves the state-of-the-art for every considered task. In most cases, they can be simply swapped for pre-trained GloVe or other word."

4.4 Universal Sentence Encoder

Universal Sentence Encoder [CYK+18] was published in 2018.

Cer et al. provide two pre-trained models for sentence encoding. One is trained with transformer model and aimed for higher accuracy. Other is aimed for faster computation and trained with Deep Averaging Network (DAN). DAN-versions compute time is linear to the input length. Both pre-trained models are made available in TensorFlow hub.

The encoder takes as input a lower cased string, tokenized by PTBTokenizer and outputs a 512 dimensional vector as the sentence embedding. The dimension of output is fixed to 512.

Unsupervised training data for the sentence encoding models is from multiple sources: Wikipedia, web news, web question-answer pages and discussion forums. They augment unsupervised learning with training on supervised data from the Stanford Natural Language Inference (SNLI) corpus.

4.5 ULMFit - Universal Language Model Fine Tuning

Before ULMFit [HR18] in 2018, inductive transfer had been used with high success on computer vision. Models such as ImageNet were pretrained on large amount of image data and then used with success on different target domains.

Yet attempts to similar **finetuning** on NLP area to gain inductive transfer (Mou et al. 2016, Dai and Le 2015) had failed [HR18].

Howard and Ruder state that (Dai and Lee 2015) first proposed finetuning a LM, but required millions of documents in target domain to reach good score. And that Mou et al. 2016 were unsuccessful with inductive transfer in NLP.

"but has been shown to fail between unrelated ones (Mou et al., 2016). Dai and Le (2015) also fine-tune a language model, but overfit with 10k labeled examples and require millions of in-domain documents for good performance. [HR18]"

ULMFit

Here two important things contributed to success of UMLFit: Regularizing and finetuning.

Regularizing - AWD-LSTM

One key factor was AWD-LSTM in 2017. Merity et al. in "Regularizing and Optimizing LSTM Language Models" [MKS17] showed how to more effectively regularize the training of LSTM - allowing to learn over longer distance in the text. Especially with two techniques: **Weight Dropping** and **Averaged-SGD** (Stochastic Gradient Descend) - Thus the name AWD-LSTM. Averaged, Weight Dropped LSTM.

Weight Dropped LSTM In weight dropping some of the recurrent hidden to hidden weight matrices are dropped, helping to prevent overfitting on the recurrent connections.

Merity et al. used 3-layers in their AWD-LSTM and Howard used same in ULMFit.

Finetuning

Another factor for sucess of ULMFit being that the author of ULMFit – Jeremy Howard had been rank #1 grandmaster in Kaggle competitions and also acted as CEO of Kaggle. He had a strong knowledge on how to optimize and finetune models for high performance became one key factor for ULMFits success.

Three new techniques he presented **were discriminative fine-tuning**, **slanted triangular learning rates**, and **gradual unfreezing**.

Three stages of ULMFit

Transfer with ULMFit consists of the following three steps: (See figure 10.)
1) The LM is pretraining on general domain Source Data (Eng Wikipedia [8]) to capture general features of the language in different layers.
2) The LM is finetuned on Target Data (for example business news)
3) Classifier finetuning on Target Task ; classifying the news articles

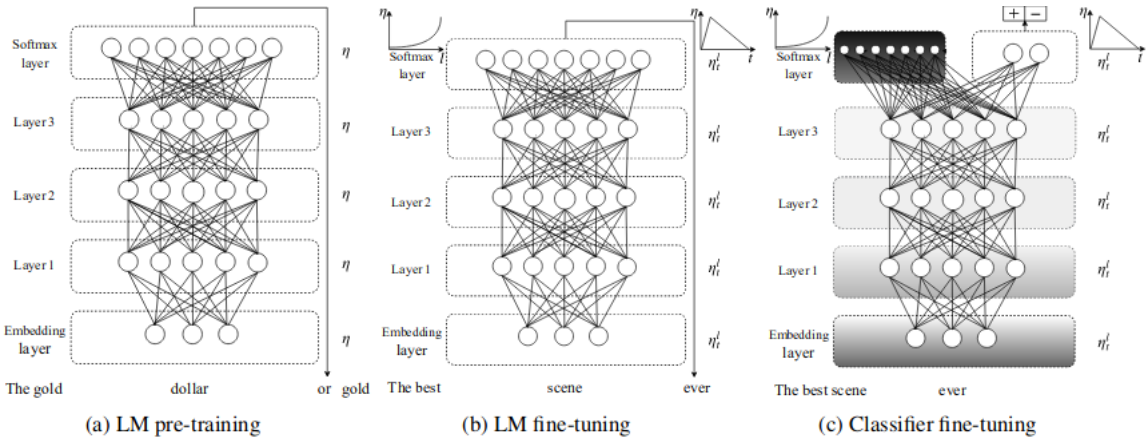

Figure 10: Three stages of ULMFit (shaded: unfreezing stages, black: frozen). Image/text by ULMFit [HR18]

a) The LM is trained on general domain corpus (Eng Wikipedia), to capture general features of the language in different layers.

b) The full LM is finetuned on target domain data using **discriminative finetuning** and **slanted triangular learning rates** to learn task specific features.

c) The classifier part is finetuned on the target task using **gradual unfreezing**.

Let's see what these mean in practise.

Discriminative finetuning

Usually each layer of weights in LSTM is updated with same learning rate .

Idea behind Discriminative finetuning is that as different layers capture different type of information, they should also be finetuned to different extent. This is reached by giving different LR to different layers.

For finding optimal learning rates, the ULMFit implementation time to time in training uses method "findLR" to compare different LR:s and find a good one.

[8] For pretraining data they use Wikitext-103 dataset, consisting of 28,595 Wikipedia articles and 103 million words.

Slanted triangular learning rates

Slanted triangular learning rates is a variation of Cyclical learning rates (CLR).

Reasoning behind CLR [Smi15] is, that diminishing learning rate leads to a local optima, which is hard to escape from. Cyclically raisin LR higher again, allows the learning to "jump" further and escape the local optima, into a better area.

"The essence of this learning rate policy comes from the observation that increasing the learning rate might have a short term negative effect and yet achieve a longer term beneficial effect." Smith L. [Smi15]

Smith mentions that experiments with different functional forms, such as linear, parabolic and sinusoidal produce equivalent results. The reason to prefer linear is it being simplest of the functions. Thus Smith ends up to triangular learning rate.

While triangular learning rate will increase and decrease with equal angle, reaching highest LR at half of iteration, as name suggests, in **slanted** version the triangle is not equal, but the highest point in LR is reached lot sooner. Reasoning being for model more quickly to converge to a suitable region. Howard et al. (2018) back this with practical testing.

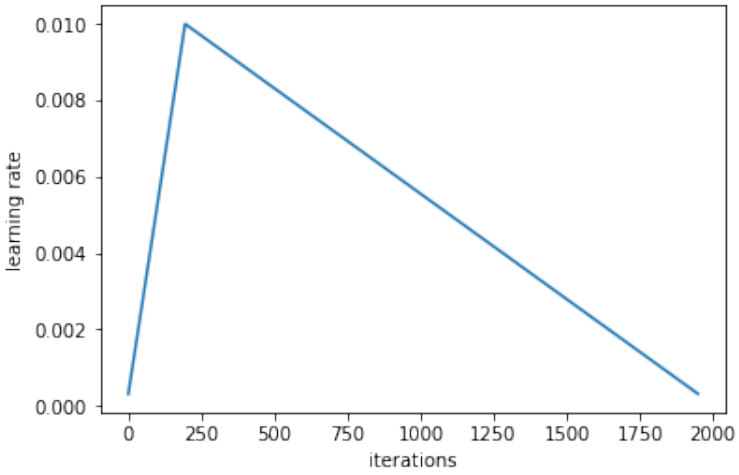

Figure 11: Cyclical learning rate in practise. clr(32,10) parameter 32 defines **ratio** of learning rate, highest LR being 32 times the smallest. Learning is started with small LR, increasing it and parameter 10 defines that the LR is raised to maximum at point of 10% of total iterations. Here total iterations is 2000 so max LR is reached at iteration 200. Rest 90% of iterations LR is decreased again.

Gradual unfreezing

When chancing a totally new task for pretrained model, such as giving a new classification task for LM that has been pretrained with task of predicting words, the model has no idea yet how to do the new task. Early samples of new task can bring largely random updates.

A lot of time and computing has been used for pretraining, going over perhaps a billion words to find a strong representation that captures the knowledge in the text.

Now if we upgrade all the weight layers in the network on each training sample - with largely random weight updates, this can lead to catastrophic forgetting and weaken the representations.

So the idea is - when learning a new target task - eg. classification, lets not touch our representations in the beginning, but freeze those weight layers of the NN. Only update one or few last layers of the network, where the classification - the new task, happens. Only when the network has learned to do the new classification task well enough (the layers of new task have reasonable weights), then we can start unfreezing the layers of actual representation, and updating the representation towards better fitting the new target domain and task.

For more details about finetuning see [9] and [10].

FastAI framework for deep learning

Such an access for giving different layers of NN different learning rates and to freezing some of them, were not readily available in deep learning frameworks.

To be able to implement ULMFit and to use these kind fine grained parameters for LSTM, Howard also authored FastAI deep learning framework, which works over Pytorch and extends it.

Conclusion

The ULMFit offered both a structure for the transfer, by using pretraining and then finetuning on target data. And they also offered efficient finetuning techniques of the LSTM model.

[9] `https://github.com/cedrickchee/knowledge/blob/master/courses/fast.ai/deep-learning-part-2/2018-edition/lesson-10-transfer-learning-nlp.md`

[10] `https://www.youtube.com/watch?v=h5Tz7gZT9Fo&feature=youtu.be&t=1h49m09s`

4.6 OpenAI Transformer - GPT-1

GPT-1 was presented in **Improving Language Understanding by Generative Pre-Training** [RNSS18].

The authors from OpenAI suggest and present a system where they use **generative pre-training** to create a LM, then **discriminative fine-tuning** on each task.

They present **Task-aware input transformations** to allow same model to take varying input from different kinds ot tasks in similar type of format.

For source to learn from, they use BookCorpus dataset, containing 7000 unpublished books and 800 M words. For model, where Howard et al. (2018) used AWD-LSTM, instead of RNN, Radford et al have chosen to use Attention and Transformer based architecture. Their reasoning is, that the dataset contains longer texts and as such, longer distance information, than for example Elmos 1Billion..., which consists of shuffled single sentences. And also the Attention-based system is more able to capture long distance connections, than RNN-based one.

They assume a large corpus of unlabeled text and several data sets of manually annotated training examples. They combine unsupervised pre-training and supervised fine-tuning. They mention goal being to "learn a universal representation that transfers with little adaptation to a wide range of tasks."

For GPT-1 they use a multi-layer Transformer decoder for the language model, which is a variant of the transformer.

Size Pretraining: 1 month on 8 GPUs. 37-layer (12 block) Transformer architecture, and trained on sequences of up to 512 tokens.

GPT-1 uses BPE with vocabulary of 40,000 merges. They used Spacy tokenizer.

4.7 Google AI BERT

"Bidirectional Embedding Representations from Transformer".
Pre-training of Deep Bidirectional Transformers for Language Understanding [DCLT18]

Bert advances the SOTA for 11 NLP tasks.

Key ideas:
- Transformer
- Deeply 2-way combined into single model.
- MASK:ing

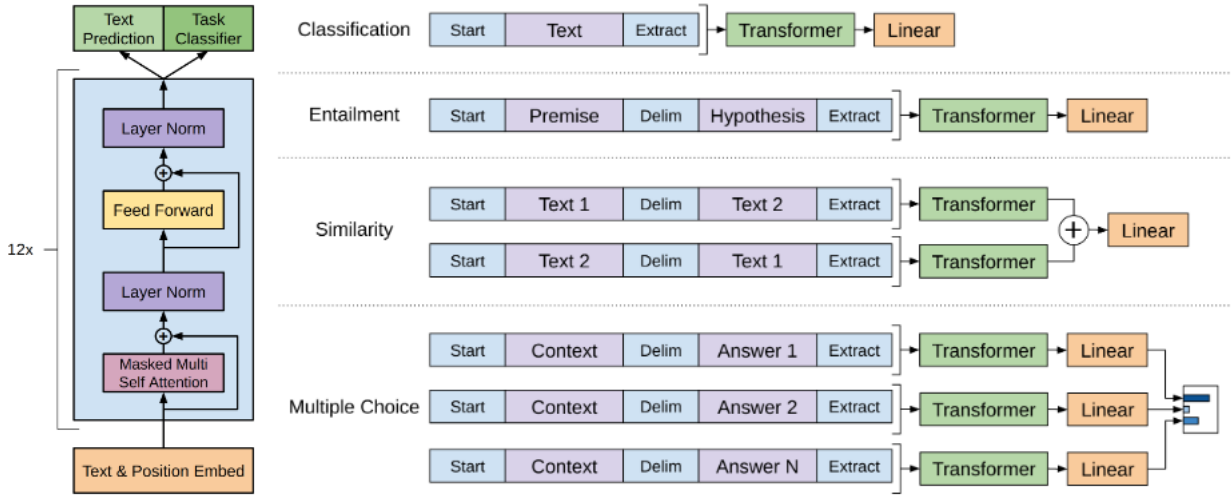

Figure 12: (left)Transformer architecture and training objectives used in GPT-1. (right)Input transformations for fine-tuning on different tasks. Image: OpenAI, Improving Language Understanding by Generative Pre-Training [RNSS18].

- 2 training tasks
- fine-tuning based representation
- bookCorpus + Eng Wikipedia
- multi-layer bidirectional

BERT is a Transformer based language model.

Genuine 2-directional model

Earlier models, such as ULMFit, and OpenAI-GPT were 1-way models. Only conditioning on left side context - the preceding text.

Limiting factor with earlier models wad been, when using pre-training task of predicting next word in the text, that deeply 2-way model predicting next and previous word, would also indirectly see these both words. ELMo and ULMfit used concatenation of independently trained forward and backward models.

Creators of BERT figured out how to make a truly 2-way model. This allows taking simultaneously advantage of context both before and after the word.

BERT learns what they call **deep bidirectional representation** by jointly conditioning on both left and right side context in all layers of the model. We can contrast this to **shallowly bidirectional** models such as ELMo, where left and right directions are trained each separately and then combined afterwards.

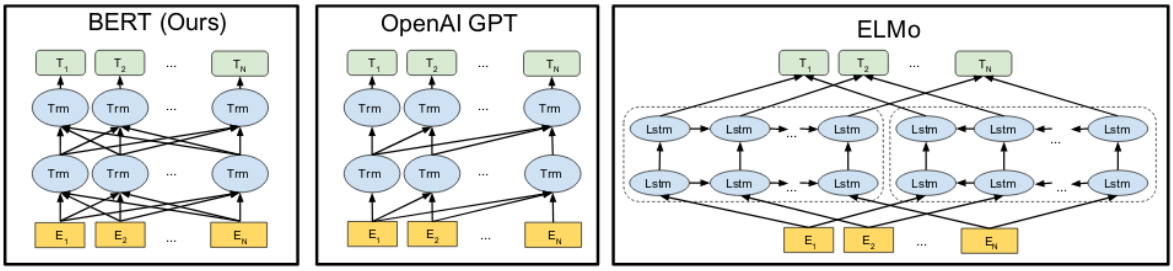

Figure 13: Differences in pre-training model architectures. BERT uses a bidirectional Transformer. OpenAI GPT-1 uses a left-to-right Transformer. ELMO uses the concatenation of independently trained left-to-right and right-to-left LSTM to generate features for downstream tasks. Among three, only BERT representations are jointly conditioned on both left and right context in all layers. The lowest target-task "Multiple Choice" is question answering, where one is the correct. Image/text by BERT [DCLT18].

Change of pre-training task to allow truly 2-way training

Reason why training 2-way model was not used before was that when training task was predicting next word, if given both ways, the model would see the future words it is tasked to predict.

Masked target word

To use genuine 2-way model, feeding preceding and following text, the authors of BERT had to come up with now kind of pretraining objective: "the masked language model"

Example of MASK from BERT:
Input: The man went to the [MASK]1 . He bought a [MASK]2 of milk .
Labels: [MASK]1 = store; [MASK]2 = gallon

BERT randomly mask 15 % of tokens. Token selected for masking is 80% of time replaced with [MASK], 10% of time replaced with random word and 10% of time kept unchanged. Purpose of the last one is to bias the representation towards the actual observed word.

BERT is trained to MASK only 15% of the text per run, while non-masking model that is predicting next word will predict 100 % the words on single run. Effect of this is that BERT requires more training epochs for reaching optimal state.

Adding second training task

In addition to masked language model, BERT uses "next sentence prediction" as additional task that jointly pretrains text-pair representations.

BERT combines 2 tasks both in pretraining and fine-tuning.
1. Predict masked word in the text.
2. Given sentences A and B, predict if sentence B is next sentence after A.

The 2nd task is enforcing model to learn relationship between sentences. Sentence B is chosen 50% time to be the next sentence to A in training data and 50% of time to be random, unrelated sentence.

Chosen maximum length of combined A + B is 512 tokens.

Architecture

BERTs model architecture is a multi-layer bidirectional Transformer encoder based on the original implementation described in Vaswani et al. (2017) and released in the tensor2tensor library.

Pre-training corpus is concatenation of BooksCorpus (800M words) and English Wikipedia (2500M words) for a total of 3,3 billion words.

BERT uses a WordPiece embeddings with 30,000 token vocabulary. Input representation used can represent both single text sentence or a pair of text sentences, allowing multiple types of tasks. Sentence pairs are separated with a special token "[SEP]".

Comparing performance of BERT to GPT-1

In table 3 are GLUE-scores compared to earlier models from BERT-paper. (Glue baseline = BiLSTM+ELMo+Attn. BERT and OpenAI GPT are single-model, single task)

$Bert_{Base}$ was chosen to have identical model size to OpenAI GPT1, for comparison reasons.
$Bert_{base}$ 12 Layers, hidden size 768, 12 self-attention heads. total parameters 110 M
$Bert_{Large}$ 24 Layers, hidden size 1024, 16 self-attention heads, total parameters 340 M

As we can see, BERT improves the GLUE-score on average of 4.5 percent points, while keeping similar size model as OpenAI GPT-1.

OpenAI GPT-1 itself brought 1.2 percent point improvement on average to previous SOTA. Bert Large still improves Bert Base with 2.3 percent point. The whole leaderboard can be found at `https://gluebenchmark.com/leaderboard`

System	MNLI	QQP	QNLI	SST-2	CoLA	STS-B	MRPC	RTE	Avg.
Glue Baseline	76.4/76.1	64.8	79.9	90.4	36.0	73.3	84.9	56.8	71.0
Pre-GPT sota	80.6/80.1	66.1	82.3	93.2	35.0	81.0	86.0	61.7	74.0
OpenAI GPT	82.1/81.4	70.3	88.1	91.3	45.4	80.0	82.3	56.0	75.2
BERT BASE	84.6/83.4	71.2	90.1	93.5	52.1	85.8	88.9	66.4	79.6
BERT LARG	86.7/85.9	72.1	91.1	94.9	60.5	86.5	89.3	70.1	81.9

Table 3: GLUE Test results from BERT [DCLT18].

Effect of Next Sentense Prediction and Masked Language Model.
In Bert paper they do ablation study by training similar model, but removing one part at a time.
a) They remove one of 2 tasks: Next Sentence Prediction (NSP)
b) They use only 1-way training, Left-to-Right, similar to GPT. Not the 2-way Masked LM.

While Bert base scores 84.4 in MNLI-m, identical model without Next Sentence Prediction, but only single training task (the masked word), scores 83.9 - that is 0.5% less. As a second test: Model "LTR & No NSP": with no Masked LM, but only Left-to-Right next word prediction as task scores 82.1, another 1.8% less.

As such, both 2-way training with masked model and 2nd training task of next sentence prediction, both seem to bring significant improvement.

Training time and cost

Authors mention training times to been 4 days for pre-training of each model.
$BERT_{BASE}$ using 4 Cloud TPU:s (16 TPU chips total) and
$BERT_{LARGE}$ using 16 Cloud TPU:s (64 TPU chips total).
With around $\$7,50$ per hour [11] in the cloud this comes to a total prices of a bit under $3000 for BASE version and 4 times that for LARGE version. Or one third of that if running in preemptible mode. Preemptive mode uses surplus capacity of the cloud - unused GPU:s. But if someone else would buy that capacity at normal price and cloud doesn't have surplus anymore, then your process can be shutdown within 1 minute notice.

Fine-tuning mode vs feature-based approach

To compare BERT in fine-tuning mode vs feature-based approach, they generate

[11] In 2019 price level.

ELMo-like pre-trained contextual embeddings and use them as input to 2-layer 768-dimensional BiLSTM before classification layer. They report that concatenating 4 last hidden layers gives a score 0.3 behind fine-tuned version, 96.1 F1 vs 96.4. The sum of last 4 hidden layers gives 95.9.

They report results Dev F1 as
Finetuned version 96.4
Embeddings layer 91.0
second-to-last 95.6
Last hidden 94.9
Sum Last 4 hidden 95.9
concat Last 4 hidden 96.1

4.8 OpenAI GPT-2

After GPT-1 and BERT being published in 2018, in 2019 OpenAI published a larger version of GPT-1, namely GPT-2 [RWC+19].

Main contribution of GPT-2 is showing that making the system considerably larger will also considerably increase the performance of the model. GPT-2 is larger version of GPT-1. Like its predecessor its pretraining task also is predicting next word in the text.

While training data on GPT-1 was BookCorpus, on GPT-2 they selected data by using Reddits outbound links, which had received at least 3 "karma points". Training data is larger than GPT-1. 8 million web pages - 40 GB of data.

"GPT-2 is a direct scale-up of GPT, with more than 10X the parameters and trained on more than 10X the amount of data. The diversity of the dataset causes this simple goal to contain naturally occurring demonstrations of many tasks across diverse domains (OpenAI blog [AR19])."

Model size in GPT-2 Large was increased considerably - to 1.5 Billion parameters. GPT-2 Small is 117 Million parameters and GPT-2 Large 1,5 Billion parameters.

Training time

Training was mentioned to be done on 32 Googles Cloud TPU v3s, each with 8 core, making total 256 cores. Training took a bit over a week.

As comparison, BERT-Large was 16 Cloud TPU with 4 cores each (64 cores) cores and 4 days. So OpenAI GPT-2 used roughly 4 x cores and 2x training time,

No fine-tuning

In addition to larger size and BPE, third big change in GPT-2 is that it doesn't use fine-tuning. Though this can be questionable on some viewpoints. If you have a own data set for the end-task, you would likely want to fine-tune the model to move its distribution closer to the end-task.

4.9 Zalando Flair

Zalando's Flair is name for both - and algorithm and a Framework.

Flair Framework Brings many other pre-trained embeddings to use by interface: Bert, ELMO. Flair framework also provides stacked embeddings - easy way to combine several embeddings, such as embeddings from language model and word embeddings. Provides optimization of hyperparameters by library hyperopt.

Flair algorithm and signature embedding

Contextualized character-level embeddings:
"Contextual String Embeddings for Sequence Labeling" [ABV18]

5 Transfer Learning

In transfer learning information learned in one task, is used for improved performance in another task.

Common situation is having data and task to be done, but no direct labeled training material. The idea of transfer learning is: Lets use different data and different task as source to train a model over, and then use that model to our own target data and task. Assumption being, that the model will learn to perform on the target data and task, even when trained on different data and/or different task. This learned knowledge can be for example a **good representation** that captures the important information about text.

Transfer learning is the ability of a learning algorithm to exploit commonalities between different learning tasks in order to share statistical strength, and transfer knowledge across tasks. [BCV13]

Transfer learning is learning on one domain and/or task, and then using learned model/embedding on another task.

5.1 Representation learning vs Transfer learning

Representation learning is learning such representation for data, that captures important information, and helps model to manage well in various kinds of tasks.

This representation can be learned unsupervisedly by pre-training with a general type of text data.

When contrasting transfer learning to representation learning, we can see that in transfer learning also,the result of pretraining, with Source Data and Source task, is a representation.

In representation learning, the goal is to learn such a representation that captures the important features of the data and/or task.

In transfer learning we especially want this representation to be such, that it doesn't represent only the distribution of the source data and task, but to be more general - so that the representation also works with the target data and target task.

In transfer learning, the common case is, that the Source Data and Target Data have a different distribution. We wish to learn such a distribution / representation, that it applies not only to the source data, but also to the target data.

In the process of fine tuning the pretrained model, the model can be tuned towards the distribution of the new data.

For example if the LM is trained with Wikipedia data, and target data is fiction literature, while lot of the language are similar, there are also differences in the kind of language - the data's have a different distribution. Here the pretrained LM (the pretrained weights of the LM) can be updated by training the model again by using training data from fiction literature. As a result the weights of the LM will be tuned more away from fact text and more towards fiction text - that is, the model is tuned closer to the distribution of the target data.

5.2 Notation / definition

For clarity, I choose one notation and use it throughout whole text. I adopt the notation from Sebastian Ruder (2019) [Rud19] (who got it from Pan et al. (2009) [PY10]). The two main concepts to define are Domain and Task, which still divide into Source Domain and Source task and Target domain and and Target task.

Domain

A domain $\mathcal{D}$ consists of a pair: feature space $\mathcal{X}$ and a marginal probability distribution P(X), where $X = \{x_1, ...x_n\} \in \mathcal{X}$

Marked by $\mathcal{D} = \{\mathcal{X}, P(X)\}$

In case of text documents, Feature space $\mathcal{X}$ is the set of all possible documents that could exist or that could be represented as a text within given limits - max length, vocabulary, possibly rules of language. x_i is a specific vector in feature space, corresponding to a concrete document. X is a particular sample of concrete documents.

One way to look at the matter is to consider $X = \{x_1, ...x_n\} \in \mathcal{X}$ as **feature vectors**. In this case what the vector represents depends on the specific LM and the units it is based on - character, subword unit, words.

It could be a vector of characters, vector of subword units or vector of words. The vocabulary (of chars, subwords or words) as well as the form of probability distribution also differs based on this.

Especially a probability distribution of a word level LM over a limited size vocabulary means that many words will be out of the distribution - represented with UNK-token.

Task

Given a specific domain $\mathcal{D} = \{\mathcal{X}, P(X)\}$, a task $\mathcal{T}$ consists of label space $\mathcal{Y}$, a prior distribution P(Y), and a conditional probability distribution P(Y|X).
Training data often consists of pairs $x_i \in X$ and $y_i \in \mathcal{Y}$

In case of classification of text documents, label space $\mathcal{Y}$ is the set of allowed labels.

Transfer learning

Ruder still defines that:
"Given a source domain $\mathcal{D}_\mathcal{S}$, a corresponding source task $\mathcal{T}_\mathcal{S}$, as well as a target domain $\mathcal{D}_\mathcal{T}$ and a target task $\mathcal{T}_\mathcal{T}$, the objective of transfer learning now is to learn the target conditional probability distribution $P_T(Y_T|X_T)$ in $\mathcal{D}_\mathcal{T}$ with the information gained from $\mathcal{D}_\mathcal{S}$ and $\mathcal{T}_\mathcal{S}$ where $\mathcal{D}_\mathcal{S} \neq \mathcal{D}_\mathcal{T}$ or $\mathcal{T}_\mathcal{S} \neq \mathcal{T}_\mathcal{T}$"
That is, objective is to learn target tasks in target domain, while either source domain or source task differs from target on.

Why does at least one of domain or task need to be different between source and target? If $D_S = D_T$ and $T_S = T_T$, ie. both domain and task are same in source and target, that is the case of traditional machine learning. So while it is common

setting, it does not qualify as transfer learning.

Definition by Pan

Pan defines Task a bit differently. Where Ruder uses prior distribution P(Y) and conditional probability distribution P(Y|X), Pan in place of this pair, uses objective predictive function f(.)

Pan: Task $\mathcal{T}$ is a pair of label space $\mathcal{Y}$ and objective predictive function f(.) denoted by $\mathcal{T} = \{\mathcal{Y}, f(.)\}$ And f(.) is not observed, but can be learned from the training data: feature vector and label pairs $\{x_i, y_i\}$, where $x_i \in X$ and $y_i \in \mathcal{Y}$.

Similarly, Pan defines transfer learning using the predictive function:

Given a source domain D_S and learning task T_S, transfer learning aims to help improve the learning of the target predictive function $f_T(.)$ in $\mathcal{D}_\mathcal{T}$ using the knowledge in $\mathcal{D}_\mathcal{S}$ and $\mathcal{T}_\mathcal{S}$ where $\mathcal{D}_\mathcal{S} \neq \mathcal{D}_\mathcal{T}$ or $\mathcal{T}_\mathcal{S} \neq \mathcal{T}_\mathcal{T}$

Here Ruder and Pan differ in definition, when Ruder defines task T consisting of 3 things, including Probability distribution. while Pan defines it by 2 - including the predictive function f(.).

5.3 Transfer Learning Taxonomy

In hes doctoral thesis, Sebastian Ruder represents a taxonomy of Transfer Learning [Rud19]. Ruder separates Transfer learning into two main categories depending if the Task before and after transfer are same or different.

In **Transductive transfer learning** the task is same before and after transfer, ie. source task and target task are same. In **Inductive transfer learning** the task is different before and after transfer, ie. source and target task are different (no matter whether the source and target domains are the same or not). The taxonomy is represented as a tree in Figure 14.

5.3.1 Inductive transfer learning - Different task

Inductive transfer learning still divides into two subcategories. If source and target tasks are learned simultaneously, it is called **Multi-task learning**. When tasks are learned sequentially, it is called **Sequential transfer learning.**

When doing transfer learning with pretrained word embeddings or pretrained language models, the training task in pretraining, ie. Source Task, has generally been

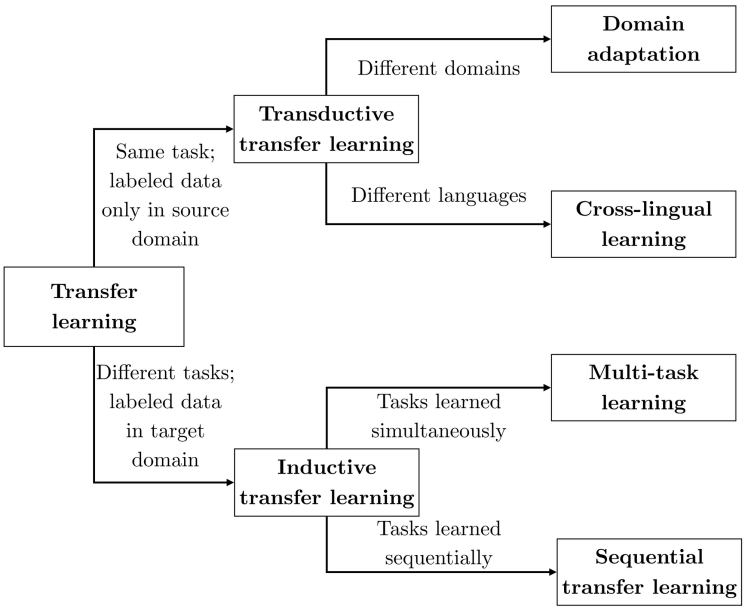

Figure 14: A taxonomy for transfer learning for NLP. Sebastian Ruder [Rud19]. General case for transfer learning with LM is located in bottom right - Sequential transfer learning.

to use the text, withhold some words of it at time and predict those withheld words - ie to **predict the source text itself**. The downstream NLP task after transfer usually is something quite different, as in our case - text classification. This means that when taking advantage of transfer learning with pretrained word embeddings or pretrained language models, due to different source and target tasks, the context is generally that of **Inductive Transfer learning**.

Sequential transfer learning

Also case of **sequential transfer learning** is common, since target task is learned after computationally heavy pretraining target task has already been completed.

As an example, pretraining a LM over Wikipedia data can take weeks of processing time when run on several dozens of GPU:s. Because of this, often it is not wished to be repeated. The common case, when using pretrained models for text classification, is that of **Inductive transfer learning with sequential transfer.**

In this taxonomy our example task and data for text classification, will locate in lower branch of "Inductive transfer earning" and under there in "Sequential transfer learning". Please see Figure 14.

In some settings though, it can be favorable to train own language model from

scratch, for example for a language different than English, or for a special task, the other areas of transfer learning might also come into use - such as multitask learning.

Multi-task learning

Multi-task learning overlaps between Multi-task learning as itself and multiple tasks used in pretraining stage of sequential transfer learning.

Ruder defines that "Generally, as soon an optimization problem involves more than one loss function, we are effectively doing multi-task learning (in contrast to single-task learning) [Rud19]."

Yet he places LMs with multiple pretraining tasks under sequential transfer learning.

Ruder mentions as advantage of multiple tasks that model learns a better generalization, as it is not able to overfit single task, but has to learn representation capturing relevant information for all of the tasks.

Also he mentions multiple training tasks working as a data augmentation, due to different tasks having different noise with same samples and being able to learn different patterns about the data samples [Rud19].

5.3.2 Transductive transfer learning - Same task

In transductive transfer learning the task is same before and after transfer. It is divided into 2 subcategories: domain adaptation and cross-lingual learning.

Domain adaptation

In domain adaptation task stays same, but domain changes. That is, target domain is different from source domain.

When **domains are different**, either (text classification) :
1) The feature spaces between domains are different.
2) Feature spaces between domains are same, but marginal probability distributions between domains are different.

Example of 1 are different languages - words are different between languages, creating a different features space.
Example of 2 is English language but different topics. Same language has same feature space (assuming here vocabulary is same), but the marginal probability distribution of those words is different between texts in different topics.

Literature Survey: Domain Adaptation Algorithms for Natural Language Processing [Li12].

defines:

Domain adaptation
- Adaptation between different corpora
- Adaptation from a general dataset to a specific dataset
- Adaptation between subtopics in the same corpus
- Cross-lingual adaptation:

Training and test data come from same distribution, same domain vs different.

** Defines domain
A domain $\mathcal{D}$ consists of two components: a feature space $\mathcal{X}$ and a marginal probability distribution P(X), where $X = x_1, ..., x_n \in X$

Task:
Given a specific domain, D = X, P(X), a task consists of two components: a label space Y and an objective predictive function f(.), denoted by T = Y, f(.), which is not observed, but can be learned from the training data.

task T = Y, P(Y|X)

When **Tasks are different**, then either
1) Label spaces between the domains are different, $Y_{source} \neq Y_{target}$, or
2) the conditional probability distributions between domains are different.

Example of 1 is binary labels vs multi-labels giving different label space.

Cross-lingual learning

Cross-lingual learning generally deals with learning a common representation for multiple languages. This can be especially helpful if target domain is in low resource language and English language has more relevant training data available.

5.3.3 Taxonomy and Ontology

While taxonomy defines relationships withing a category, ontology defines a larger structure, possibly containing multiple taxonomies.

"Taxonomy identifies hierarchical relationships within a category. Each taxonomy is designed to categorize items within just one dimension. For example, home im-

provement stores have taxonomies of their products.

But what if you want to connect certain specialty items into a list of potential clients in those areas? A taxonomy for tool types cannot do that on its own, but when connected with other taxonomies for "compatible materials" and "client needs" into a larger structure of an ontology, it becomes possible. Ontologies achieve a higher level of sophistication by providing richer information, including information about the relationships among entities. [Sch]"

It is possible that the taxonomy represented by Ruder would not be the whole picture, but ontologically there could be theoretically more of meaningful concepts. That said, the taxonomy of Ruder fits well the current modern language models and gives a good tool in understanding and formally presenting them.

5.4 Availability of unlabeled data while training

(Note below: Arnold et al. (2007) provide definitions for transductive learning, which do not match that of Ruders, but are a different view.)

Pan points the difference in defining Transductive transfer learning. Arnold et al. require, in addition to task being same, that **all unlabeled data in the target domain are available at training time** [ANC07].

Pan et al. (2009) self only require that **part of the unlabeled target data** is available at training time, in order to obtain the marginal probability for the target data [PY10].

Additional paradigms

Arnold et al. describe additional learning paradigms. According to Arnold et al, In the paradigm of **inductive learning** (X_{train}, Y_{train}) are known when training, while (X_{test}, Y_{test}) are completely hidden.

In case of **semi-supervised inductive learning** the model is also provided with auxiliary unlabeled data $X_{auxiliary}$, that is not part of test set. "It has been noted that such auxiliary data typically helps boost the performance of the classifier significantly"

Another setting closely related to semi-supervised learning is **transductive learning** in which X_{test}, but importantly not Y_{test} is seen at training time.

That is, the learning algorithm knows exactly which examples it will be evaluated

on after training. This can be a great asset to the algorithm, allowing it to shape its decision function to match and exploit the properties seen in X_{test}. One can think of transductive learning as a special case of semi-supervised learning in which $X_{auxiliary} = X_{test}$.

In practise

We can see this difference realizing concretely in ULMFit language model. While ULMFits pretraining data is from Wikipedia, when trained on business news articles as target domain, it will in one phase use available unlabeled data to fine tune the probability distribution of its language model closer to distribution of the Business news articles. Separately in later phase is a classifier trained.

So here comes the choices, if to feed the X_{train} only (and X_{dev}), or also X_{test}.

The more unlabeled target domain data the model gets, the closer its distribution is likely to become to that of target domain.

If an organization has a large amount of target domain documents they are training a language model for, they should take notice if the chosen language model is taking advantage of that unlabeled data for fine-tuning its probability distribution towards the target domain. This could make a noticeable difference in performance.

Same time some language model frameworks make a hard assumption in their interface, that data is separated into train, dev and test sets with labels, and unlabeled data outside train or dev set is not taken advantage of.

Example - Sets to use for training

If organization has 100.000 unlabeled text documents and they manually annotate for example 3000 of them for training the model and split that for 2000 of train set and 1000 test set. **inductive learning** would use those 2000 labeled examples for training a model. **Transductive learning** will also use the 1000 test examples without labels. **semi-supervised inductive learning** would use all 100.000 samples without labels, that is, also the remaining 97.000 non annotated samples. Adding $X_{auxiliary}$ data that is not part of labeled training data nor test data.

The case of ULMFit is that of semi-supervised inductive learning. It uses in 1st step all 100.000 samples, including unlabeled samples (and test set without labels, if given) to fine-tune the language model part, so that the distribution is updated closer to the distribution of the target data. Then in 2nd step, the 2000 labeled samples would be feed with labels, to train a classifier.

Letting model see the test set beforehand

So when would model be allowed to see the test set beforehand? If we consider a model been run once a week to produce reporting from latest news articles, producing predictions for a larger batch of new samples, then it is ok for model to see the whole batch of new samples, update its LM part first and then produce the predictions. This resembles human taking a test and being allowed to read through all the questions before answering the first question.

On something like hate speech detection on the other hand, it is not ideal to detect samples only once a week and remove or process detected possible ones with several days delay. Here more real time processing is favored, meaning that model don't have such a larger batch of new samples available.

We can see that from performance point, it is favorable to use also the unlabeled documents from target domain, to tune the LM closer to the distribution of the target domain. Yet many implementations do not do this in practise, but only use the labeled data. So in practise it is good to be aware of this distinction and if a specific LM is using the unlabeled data for fine-tuning or not.

5.5 Self Supervised Learning

One way to view the highly successful Language Models is that of Self Supervised Learning. Self supervised learning fits in between supervised and unsupervised learning. The bulk of what is learned consists of understanding the data itself and its regularities - through some prediction or reconstruction criterion (such as hiding a word in text and asking to predict it), rather than learning some particular end task with labels.

The NN is still trained supervised way, but the training target comes from the data itself and not from external label.

'I now call it "self-supervised learning"'
"Self-supervised learning uses way more supervisory signals than supervised learning, and enormously more than reinforcement learning. That's why calling it "unsupervised" is totally misleading. That's also why more knowledge about the structure of the world can be learned through self-supervised learning than from the other two paradigms: the data is unlimited, and amount of feedback provided by each example is huge [LeC19]." Yann LeCun

LeCun also mentions that self-supervised learning works especially well for NLP since **text is a discrete space** in which probability distributions are easy to represent. And that so far similar approaches have not worked as well for images or videos, because their distributions are of higher-dimensional continuous spaces.

5.6 Active Learning

When working with small amount of labeled target data, one related concept is that of Active Learning. Active learning is the task of reducing the amount of labeled data required to learn by querying the user for labels for the most informative examples, so that the concept is learnt with fewer examples.

With large unbalance between classes - such as only 5.9% of samples in hate speech data set being hate speech, if human annotators annotate 1000 samples of data, they will annotate expectation of 59 samples of hate speech. It is plausible that training a model could do considerably better with more balanced amount of samples.

In active learning one would annotate some data, train model over it and let the model predict / pick new samples to be annotated, that are more likely than 5.9% to be the minority class.

In addition to most likely label, the models can also return the softmax values for each label - ie. how certain the model is about its prediction. One strategy would be to pick for annotation the samples that model considers most likely to be the minority class. Another option is to look at the hardest decision border - for hate speech data set that is the border between aggressive speech and hate speech. So one could pick the samples that the model is most uncertain if they are aggressive speech or hate speech. The examples closer to the border of two classes might be more valuable training data.

5.7 Choice of source task and data for prelearning

5.7.1 Choice of Source task

Popular pretrained language models, such as ULMFit, BERT etc, not only differ with each other in structure of the neural network, but they also differ in the choice of Source Data (and with that, the Source Domain) and the choice of Source Task for pretraining.

In Cove*, authors use Machine Translation (MT) from English to Germany as the source task for Language Model. Reasoning being that to be able to accurately translate from language to another, the system has to learn to understand details of the language and its structure, such as singular/plural, long range connections etc.

In ULMFit, authors use Language Modeling - their chosen task is, given sequence of words, to predict the next work.

Both use similar arguments why MT / LM is a good choice for transfer learning - able to capture understanding of the language.

The big advantage of LM compared to MT is that it doesn't require language pairs of data, so data suitable for training is better available.

In table 4 are some LMs and their source data and Source Task.

Model	Source Data	Source task
CoVe	Eng and German lang pairs of text	Translating text
ULMFit	Wikipedia eng	Predicting next word in text.
BERT	English Wikipedia (2,500M words)	Predicting masked word
	and BooksCorpus (800M words)	and Textual entailment of 2 sentences.

Table 4: Example of Source Domain and Task.

Comparison of source tasks

Zhang et al. looked at what kind of effect the choice of pre-training task has on linguistic information that is learned [ZB18].

They compared four pre-tasks for learning a representation for transfer learning: language modeling, translation, skip-thought, and autoencoding, by using the learned representation for several kinds of related end tasks (POS-tagging and Combinatorial Categorical Grammar -supertagging). While skip-thought and autoencoding have limitations such as producing constant length encoding for all inputs, interesting comparison is the LM vs translation.

They found that representations from LM perform best.

"We find that representations from language models consistently perform best on our syntactic auxiliary prediction tasks, even when trained on relatively small amounts of data, which suggests that language modeling may be the best data-rich pretraining task for transfer learning applications requiring syntactic information [ZB18]."

LM as pre-training task offers:

1. Good testing results
2. Good practical results with multiple LM:s in end task performance
3. Avoiding possible unwanted bias from specialized pre-training task, and
4. Practical issue of large availability of suitable unsupervised training data

The reasons above together suggest that currently LMs seem to be the favorable pre-task for learning representation.

5.7.2 Choice of Source data

Modern language models have used varying source data sets.

ULMFit uses Wikitext-103. OpenAI uses BookCorpus dataset, containing 7000 unpublished books. Generally it would be good, if source data has a distribution not too different from the target data set.

Radford et al. [RJS17] mention that source data of Amazon product reviews might not have enough coverage to generalize well, to work well with data set of books. Ie. the distribution of source and target data set might be too different.

"Since we are interested in high-quality sentiment representations, we chose the Amazon product review dataset as a training corpus." - "But some of the classification tasks they are evaluated on, such as sentiment analysis of reviews of consumer goods, do not have much overlap with the text of novels. We propose this distributional issue, combined with the limited capacity of current models, results in representational underfitting [RJS17]."

One way to deal with this issue of distribution is to use multiple different data sets for pre-training, such as BERT using Wikipedia data and BookCorpus to train on both: non-fiction and fiction based distributions.

6 Example data sets + tasks for experiment

To make theory more concrete and to ground it, I will use two example data sets and their related classification tasks. Limiting into classification tasks helps reducing and limiting the theory into more reasonably definable.

I will use the data sets as practical examples and show how to formally present the transfer - with concepts of source domain, target domain, source task and target task. Applying it with real data sets shows how the formalism, while in itself clearly

defined, does not have a good one to one match into real data, but lot of choices need to be made while mapping it. Part of this due to Pan et al. defining the formalism with a running example of feature space containing binary feature vectors. Here the starting point of textual data is not in numerical form yet.

Second, for comparing the performance of different types of language models, I will also complete the attached classification tasks with different LM:s.

Data sets

The two data sets are quite different from each other.

One data set is business news for entity level sentiment analysis. It has long news articles, which as published text, have generally good and correct sentence structure and correctly typed words.

Other data set is Twitter tweets for hate speech detection. As text, tweets are lot shorter, less structured than published text and have misspelling, both with and without purpose, on meaning to make own versions of words. Also it might have misspelling with the purpose to deceive classifier ("f*ock"). Understanding misspelled or made-up words probably would gain from a model that can understand text on smaller than word level.

6.1 Hate Speech detection on Twitter tweets

The hate speech data set contains 24783 tweets from Twitter [DWMW17]. They are classified as hate speech, offensive language or neither - annotated by CrowdFlower (CF) users. By minimum of 3 users, but sometimes more, when judgements determined unreliable by CF.

Class	description	percent of data
0	hate speech	0.0577
1	offensive language	0.7743
2	neither	0.1679

Table 5: Classes and distribution in hate speech data set. ‘class‘ = class label for majority of annotators.

Description of the classes (labels) and their distribution can be found in table 5 and in figure 15. Examples of data can be found in Table 6.

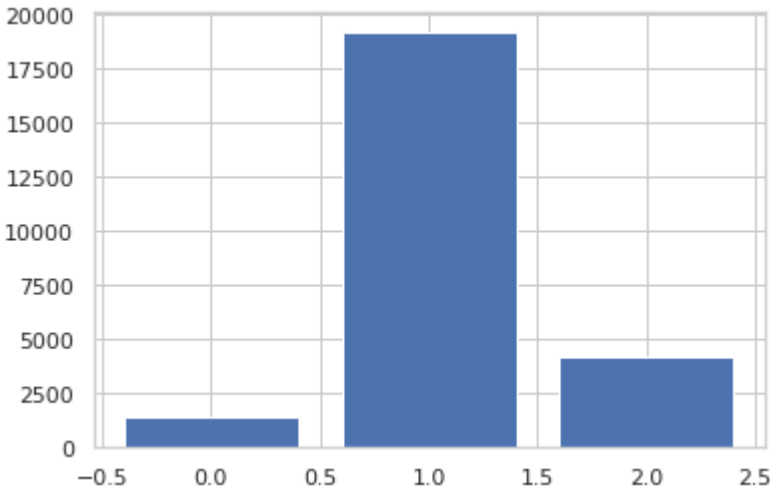

Figure 15: Class distribution of hate speech data set.

Authors self describe the data set

"Data has been selected from larger amount of tweets by looking for keywords present in Hatebase[12]. Because of that, data set is highly imbalanced: about 80% of tweets contain offensive language [DWMW17]."

They mention that "Only 5% of tweets were coded as hate speech by the majority of coders and only 1.3% were coded unanimously". This low unanimousity also tells on its part, how deciding if a text is hate speech, is hard for humans also.

Data source

Data itself is presented in [13].

They mention that a larger version of the dataset is available in Github [14]. This larger set is what I use.

What counts as hate speech?

They define hate speech as:

We define hate speech as language that is used to express hatred towards a targeted group or is intended to be derogatory, to humiliate, or to insult the members of the group.

And continue *In extreme cases this may also be language that threatens or incites violence, but limiting our definition only to such cases would exclude a large proportion*

[12] https://hatebase.org/

[13] `https://data.world/crowdflower/hate-speech-identification`

[14] `https://github.com/t-davidson/hate-speech-and-offensive-language/`

class	tweet
2	!!! RT @mayasolovely: As a woman you shouldn't complain about cleaning up your house. & as a man you should always take the trash out...
2	!!!!! RT @mleew17: boy dats cold...tyga dwn bad for cuffin dat hoe in the 1st place!!
1	i need a trippy bitch who fuck on Hennessy
1	i txt my old bitch my new bitch pussy wetter
0	@CB_Baby24: @white_thunduh alsarabsss" hes a beaner smh you can tell hes a mexican
0	These sour apple bitter bitches, I'm not fuckin wit em

Table 6: Example of tweets and their class.

of hate speech.

Importantly, definition does not include all instances of offensive language because people often use terms that are highly offensive to certain groups but in a qualitatively different manner.

For example some African Americans often use the term "nigga" in every-day language online, people use terms like "hoe" and "bitch" when quoting rap lyrics, and teenagers use homophobic slurs like "fag" as they play video games. Such language is prevalent on social media , making this boundary condition crucial for any usable hate speech detection system [DWMW17].

As we can see from above, the key border in the dataset and task is that of between offensive language and hate speech.

Context of social media posts

Detecting Hate speech in social media has raised up as a hard classification problem.

In addition to words misspelled by typo, bad language etc, hate speech also can contain words misspelled on purpose to avoid detection and whole sentences or texts modified to fool algorithm..

Handling these might need model to understand also subword level and never seen words that are made up, but which humans still recognize meaning another word. These might be different structure than words misspelled by accident or by typo. Tweets are also short, which brings its own challenges.

User posts in social media, twitter etc. can be less clean and less formal than more edited texts, such as Wikipedia or news articles. They have mistyped words, special niche words, faulty language, less structure etc.

The misspelled words in them self mean that used representation/model would gain from understanding also subword information. Either on themselves, or possibly with some kind of preprocessing. This both makes task harder for model, but also provides a harder and in some sense, more realistic challenge.

In Wired article, Matsakis mentions that researches have found hate speech detection models to be context dependent. Model trained on one data set does not work well on other. In this matter, fine-tuning pre-trained LM might have an advantage, due to their ability to learn different context from small amount of new target data [Mat18].

6.1.1 As Transfer learning - Hate speech

On higher level we can consider the transfer learning and transfer happening between each models source domain, such as Wikipedia and its distribution (ULMFit) or BookCorpus and its distribution (GPT-1) and the target domain - that of Tweets and their distribution.

While pretraining data actually could be whole English Wikipedia at some time point and in that sense represent the real distribution of Wikipedia, the target training data is not all existing tweets but only a small sample and as such is only an approximation of the distribution of tweets.

Source task T_S usually has been some version of predicting the source data itself. Target task T_T in this case is classifying tweets into 3 different labels.

We now have a source domain D_S of Wikipedia articles and source task T_S of predicting next word. We also have a Target domain D_T of tweets with maximum length of 140 characters [15] and its target task T_T of classification into 3 classes.

More detailed level

While the formal definition of transfer learning seems clear with Source domain etc., the real data sets are lot more messy and grey area.

The literature largely, and definitions such as by Pan et al. assume starting point of having source and target data, where there are already vectors of train samples and their responding labels: $train_X$, $train_y$, $test_X$ and $test_y$. Yet there is no clear

[15] Maximum length of tweet has since been raised by Twitter to 280.

mapping from real data set to the formal definition, but multiple choices to be made.

In more detailed level, even if each model would be given exactly same pretraining data (such as Wikipedia), depending on each models token level, they will represent it in a different kind of feature vectors (words, characters or subword units) and still in addition they will make several kinds of transformations to reduce computing required (cutting vocabulary of words from +1million to max 60.000, possibly truncating sample texts to shorter etc.)

Domain $\mathcal{D}$

We saw that a domain $\mathcal{D}$ consists of a pair: feature space $\mathcal{X}$ and a marginal probability distribution P(X), where $X = \{x_1, ...x_n\} \in \mathcal{X}$

In case of tweets, Feature space $\mathcal{X}$ is the set of all possible tweets that could exist or that could be represented as a text within given limits - max length, vocabulary.

X is a particular sample of tweets. That is, the tweets in the data set or smaller sample of it. Probability distribution P(X) for the words and text is defined by specific samples - tweets in data set, $X = \{x_1, ...x_n\} \in \mathcal{X}$.

Differences in Feature space and Target Domains

Feature space $\mathcal{X}$

Here we can see a difference between models. Depending on the specific language models, the original samples of tweets could be represented on different granularity level: as words, characters, subword units or some other. This will also give a bit different feature space and along it, different probability distribution.

Feature space and its feature vector x_i might be a list of integers, mapping to words, with vocabulary of 60.000 (ULMFit). Or it could be a list of subword units from vocabulary of 50.000 subword units (GPT-2).

Probability distribution P(X)

With different training sizes, also the target Domain will be a bit different - in each case the X, sample of particular tweets will be that specific sample and the transfer will happen from original distribution to the distribution of the training data sample. Which in itself is an approximation of the distribution of all tweets. With smaller training data it is likely a worse approximation.

The Feature space of target domain is already defined by the specific model or language model - since **feature space will be same between source and target**. The marginal probability distribution P(X) is related to specific sample of X in $\mathcal{X}$

and as such is different between source and target.

Here we can see, that between different training samples, such as trainsize 100 and trainsize 3000, the feature space remains same, but probability distribution P(X) is different - since train sample X itself is different.

Task T

Task T consists of Domain D + Label space $\mathcal{Y}$.

In case of hate speech - text classification, label space $\mathcal{Y}$ is simple: the set of all labels $\{0, 1, 2\}$.

Goal of transfer

After seeing above, now the goal in transfer learning is to learn the target conditional probability distribution $P_T(Y_T|X_T)$ in D_T with the information gained from D_S and T_S.

6.2 Sentiment analysis of Business news

The business news data set by Pivovarova et al. consists of 14,172 business news articles and 16840 chosen entities (in this case company names) in them [PKY18]. Articles have been human annotated for the chosen entities, for what is articles polarity towards the mentioned entity - a company name.

There are 5 classes of Polarity: Very negative, negative, neutral, positive, very positive. In addition there was a 6th class: "contradiction", where human annotators did not agree on the class. Samples with class contradiction I removed from the data. Classes and their distribution is presented in table 7

Class	description	percent of data
0	very negative	0.291
1	negative	0.1760
2	neutral	0.019
3	positive	0.317
4	very positive	0.197

Table 7: Polarity classes for entities in news articles and their class distribution.

There can be more than one different entities in the article, in which case the article comes more than once in the data set. Once for each entity. For the corpus, see[16].

[16]The corpus is available at `http://puls.cs.helsinki.fi/polarity`

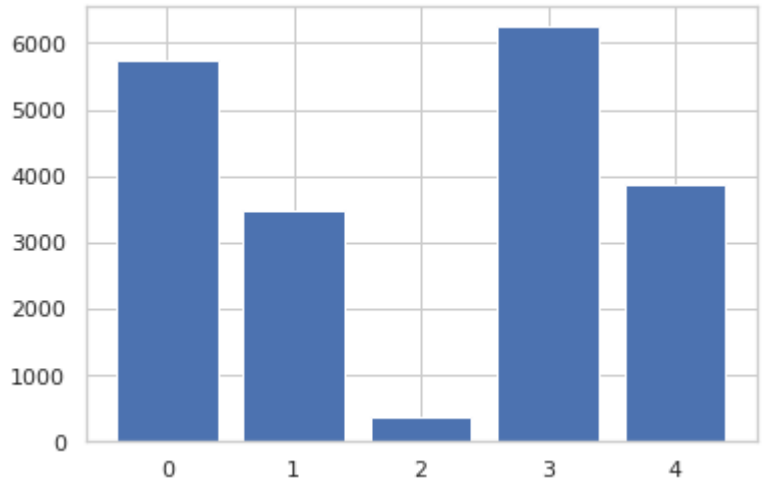

Figure 16: Class distribution of business news data set.

Example of data

Here is an example of the data. While this article is about Tesla, the target entity here is Facebook - with just one mention and in this case neutral polarity.

entity info

'entityId': 14123
'name': 'Facebook'
'offsets': ['end': 565, 'start': 557] (character locations)
'polarity': 2 (neutral)

content field

article date: '20170201
article id: 0A21D5765ED3B8F51065D3CF7339371E
article header: Tesla is Now Officially More Than Just About Cars
article main body: rest of content

text body

"The company has been working on home energy products for quite some time now, but it's the acquisition of SolarCity Corp. that solidified the name change. Now known as just Tesla, Inc., the company is moving beyond its original goal of manufacturing electric vehicles and focusing on its role as an energy company. The name change has been hinted over the last few months, with Tesla also dropping Motors from its ***Facebook*** *page as well as its official website's URL. SEE ALSO: Tesla Model S Falls Short of IIHS Top Safety Pick Rating Tesla acquired SolarCity late last year, a solar power company also founded by Elon Musk's cousins..."*

Article length

LSTM-based neural networks generally require each input sample to be identical length. If samples were to be used as they are, usual way is to extend all samples to same length as the maximum one, with extra EMPTY tokens. Then these also need to be processed by the NN with computational cost.

The length of business news articles in data set varies quite a lot. In fig 17 we can see distribution of the article length as characters. There are very few articles of length 9000 characters or more. Longest article being 13825 characters long. In practise we can see that truncating documents to at least around 5000 or 6000 characters will cut quite minimally information away, but reduce the size of input data to NN to almost one third. This will increase processing speed considerably and also reduce memory requirement, possibly allowing larger batch sizes if wished.

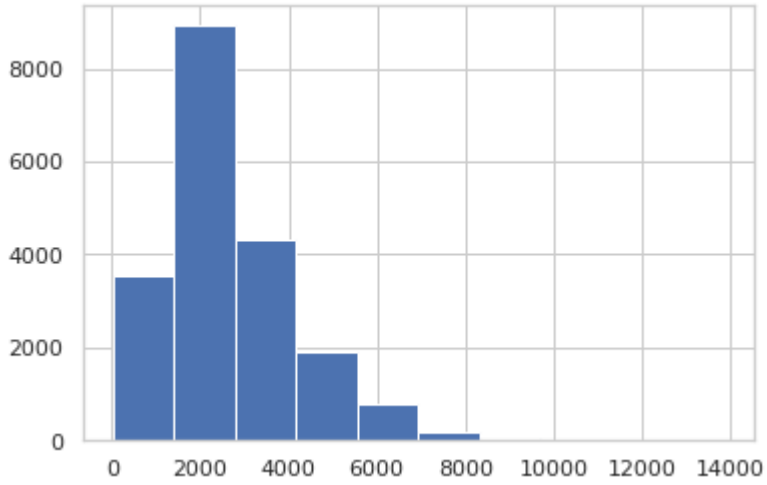

Figure 17: News articles distribution: length in characters. X-axis is document length. Y nr of articles per bin.

6.2.1 As Transfer Learning - Business news

We saw that the formalism defines Target Domain and Target Task. So, what do these mean in practise with our data set of Business news classification?

On higher level similarly to before, Source Domain is defined by each models pre-training data, such as Wikipedia. These are identical in the case of hate speech.

Instead Target domain is different here.

Now target domain is that of business news and Target Task is that of sentiment

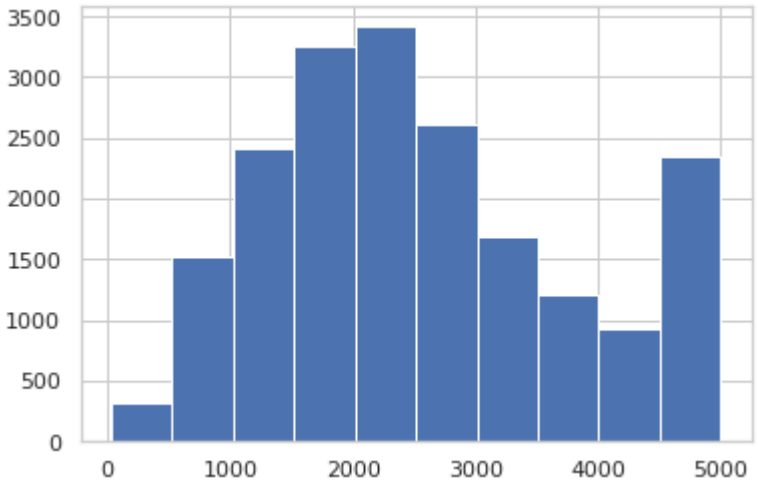

Figure 18: News articles distribution: length in characters. After truncating to 5000 characters.

analysis - defining if given news article is positive or negative towards given entity (company name) and on what scale.

More detailed level

Target Domain $\mathcal{D}_T$

Return in mind that a domain $\mathcal{D}$ consists of a pair: feature space $\mathcal{X}$ and a marginal probability distribution P(X), where $X = \{x_1, ...x_n\} \in \mathcal{X}$

In case of business news, Feature space $\mathcal{X}$ is the set of all possible documents that could exist or that could be represented as a text within given limits - max length, vocabulary.

X is a particular sample of the business news articles. That is, the articles in the data set. Probability distribution P(X) for the words and text is defined by specific samples (articles in data set) $X = \{x_1, ...x_n\} \in \mathcal{X}$.

The Target domain is that of business news articles in the data set.
$\mathcal{X}$ is the feature space of all possible business news articles.
Distribution for the words and text is defined by specific samples (articles in data set) $X = \{x_1, ...x_n\} \in \mathcal{X}$.

Feature space $\mathcal{X}$ and sample X

X is a particular sample of the business news articles. Now here this could mean different things depending on situation, such as some of train/dev/test sets and also different chosen sizes for them.

As an example in a train set of 3000 articles. $X_i \in \mathcal{X}$ is a single article with index i.

Marginal probability distribution P(X)

Marginal probability distribution is represented as P(X), where $X = \{x_1, ...x_n\} \in \mathcal{X}$. Here the marginal probability distribution P(X) defines a a probability for a specific news article.

One way to look at the matter is to consider $X = \{x_1, ...x_n\} \in \mathcal{X}$ as **feature vectors**. In this case what the vector represents depends on the specific LM and the units it is based on - character, subword unit, words.

So for the same data, depending on chosen LM, the feature vector can consist of characters, subword units, or words. The vocabulary will also differ based on this.

Target Task - $\mathcal{T}_\mathcal{T}$

Given a domain $\mathcal{D} = \{\mathcal{X}, P(X)\}$, a task $\mathcal{T}$ consists of label space $\mathcal{Y}$, a prior distribution P(Y), and a conditional probability distribution P(Y|X).

Training data consists of pairs $x_i \in X$ and $y_i \in \mathcal{Y}$

The task is to classify a business news article by how positive or negative it is toward given entity in the text. Our label space $\mathcal{Y}$ = [0, 1, 2, 3 , 4]

This is our goal, ie Target. As such, our Target Domain: D_T = X, P(X), where X is the space of all possible document representations. X_i is the representation of news article i.

As an example in ULMFit the source data is English Wikipedia articles. Source task is hiding the next word in the sentence, word by word and predicting it.

Goal of transfer

We now have a source domain D_S of Wikipedia articles and source task T_S of predicting next word. We have a Target domain D_T of Business news articles and its target task T_T of sentiment classification.

Now our goal in transfer learning is to learn the target conditional probability distribution $P_T(Y_T|X_T)$ in D_T with the information gained from D_S and T_S.

6.3 Reducing and transforming Feature space X

When implementing a LM in practise, a decision usually needs to be made on parameters, max length of the document. (in words, characters or bytepairs, depending

of type of a specific LM). Documents longer than this will be truncated to this. Documents shorter than this will usually be elongated by adding an extra filler token.

This is to fill the requirement of many NNs to have input of same length and to allow batch processing. Vocabulary (if word based model) is usually capped into n most common words, such as 30.000 or 60.000. just Here we can see, that the detailed definition of a target Domain can wary depending on the parameters: the max allowed length of the document, the size of vocabulary to keep etc.

A character or subword unit based LM might be able to represent all possible words in the data, while word based usually will have OOV for some words. As such, some rare words might belong into the feature space $\mathcal{X}$ in one LM, but not in the other (word based).

Simplified feature space after preprocessing

On abstract level we can here separate:
1) the real documents and their feature space. and
2) possible simplified feature space after preprocessing text (eg. shortening to max length, removing special characters), replacing numbers with a token
3) The feature space of individual LMs, where they might still limit the feature space more

When we are given a Task T, what ultimately counts is the 1). The reductions in 2) and 3) can be made if considered to improve the performance.

Limiting vocabulary size

Vocabulary can be cut smaller.

Truncating the text shorter

Truncating can be done to max amount of characters, or to max amount of words.

Transforming words or special characters

Special characters may be removed, or transformed to other, such as euro, pound and dollar all into dollar.

7 Empirical comparison - Experiments with hate speech and business news

I will compare how different context free models and language models perform in transfer learning and in text classification, with different small amounts of labeled training data.

All models are trained on same data. Final test set is 3k samples, and it will stay same samples during different sized training data.

Random sampling of data keeping class ratios

Usually train, dev and test sets are chosen large enough, that they statistically approach the distribution of the whole data set. Ending up far from the data sets distribution with sampled data is very unlikely. Here though, train set is on purpose set very small - as small as 100 training samples. For example the hate speech data set has only 5.9% of the samples from minority class. This means that randomness could get high effect on the results: producing 4 samples or 8 samples of the 100 being minority class. To counter this, random sampling of data is done **keeping the class ratios**. So train, dev and test sets will have same percentage of the classes, than the whole data set has. Another option would be to use cross validation, but as there are multiple models and multiple training sizes already, leading to three figure amount of trained NN:s, that was not used.

Framework

As a framework I will use Zalando Flair, which is based on Pytorch. Flair has implementations of several popular language models.

7.1 Models to compare

Baseline: Majority class

FLAIR:
- Glove
- en-twitter (static word embedding)
- Fasttext: en crawl
- Fasttext: news wiki
- ELMO
- Flair-embedding

- Bert base-cased
- BPE - Byte Pair Embedding
- GPT-1

Glove and en-twitter are static word embeddings, where specific word has only one (at most) embedding through text. Words outside max vocabulary are replaced with unknown-token. Fasttext has two versions pretrained on different data. In Fasttext for unknown words in target data, character n-grams of the unknown word are used to create embedding. Elmo, Bert, GPT-1 are language models - contextual word embeddings. Flair has implementation of BPE as Flair. Flairs signature embedding - Flair embedding, is a mix of multiple techniques.

Embedding based models: Embedding produced for text and then that is fed to LSTM. (Flair: DocumentRNNEmbeddings with 512 hidden units LSTM. Non-bi-directional. Reproject words with dimension=256. TextClassifier)

7.2 Tests to perform

In context of business news and hate speech, how do different LMs perform? What is their F1-score (micro)? How much they improve over word vectors.

A test set of 3000 items is separated and is same for all models. Different training data sizes are sampled in different sizes, from 100, up to 18.7k (tweets) and up to 13.7k (business news). All models are trained with same training data and then used to produce predictions over the same test set. These predictions are loaded and micro-F1 computed for them versus true labels.

Test setting

Number of samples to use in the test is:
Hate speech data, largest train set:18783 samples.
Business news data, largest train set: 13718 samples.

Comparing performance with different amounts of training data

Compare different sizes of training data - n labeled examples: 100, 200, 500, 1000, 3k, 7k, (18.7k / 13.7k) (for hate speech, largest train data set is 18783 samples, for business news largest train data is 13718)
- how does the amount effect models performance? Can there be seen a number or area / curve, that would be "enough" items of labeled data? - do some of the models perform considerably better or worse with low amount of labeled data?

Evaluating models

For better reliability, instead of using models own reported accuracy, each model is used to produce actual predictions for the test samples and those predictions are saved in a file. Then separately, predictions of all models are loaded and F1-score produced by comparing to real known labels of the test set. This way also provides possibilities for further analysis between predictions of different models.

Source code for experiments

The experiments with source code can be found under open source license:

`https://github.com/jannenev/compare-nlp-models`

7.3 Results - max training data and speed

Flair: F1-scores on hate speech data (15 epochs). Single embedding. 18.7K train data

Model	F1-score	training time s
glove	0.892	420
fasttext web-crawl	0.914	512
fasttext news/wiki	0.903	490
en-twitter	0.907	505
elmo	0.909	557
Flair	0.829	521
bert-base-cased	0.914	1232
BytePairEmbedding	0.899	638
GPT-1	**0.917**	**1123**

Table 8: F1 for max training data.

F1 scores for different models with full 18.7K training data are shown in Table 8. GPT-1 reaches highest accuracy with 0.917. Bert uncased comes near with 0.914. Noteworthy is how near also comes fasttext web-crawl to 0.914, with over half faster training time. Same data is represented as barplot in Figure 20.

Prediction time is in figure 19.

Analysis - high training data

With large 18K training data, the highest F1 is narrowly reached by GPT-1 at F1

Model	Traintime	Prediction time
glove	71.011	1.614
fasttext web-crawl	208.219	1.697
fasttext news/wiki	209.539	1.652
en-twitter	129.847	1.741
elmo	77.256	43.431
Flair	56.918	31.028
bert-base-cased	259.950	22.955
BytePairEmbedding	61.296	2.724
gpt-1	198.082	21.752

Figure 19: Model prediction time : seconds per 1000 samples. (traintime per 100 samples

of 0.917. BERT and Fasttext web-crawl getting very near at 0.03 smaller. Noticing both F1 and processing time, the Fasttext web-crawl is a high performer. Bert and GPT-1 are over 2 times slower in training and over 10 times slower in prediction.

7.4 Effect of training data size

Baseline

I will use predicting always majority class as a baseline, which models should improve over.

For hate speech data set, the majority class (1 - offensive language) represents 76 % of samples in test data (0.7596). Predicting always majority gives also f1-score of 0.76, which we will use as a lower bound.

For Business news data set, the majority class is 3 "positive" with 31.7 % of data. This gives f1 baseline of 0.317.

Having total of 5 classes in news data, compared to 3 of hate speech, the news data set gives harder problem to score numerically high.

Trainsize

In figure 21 and table 9 are F1-scores for hate speech data with different sizes of training data.

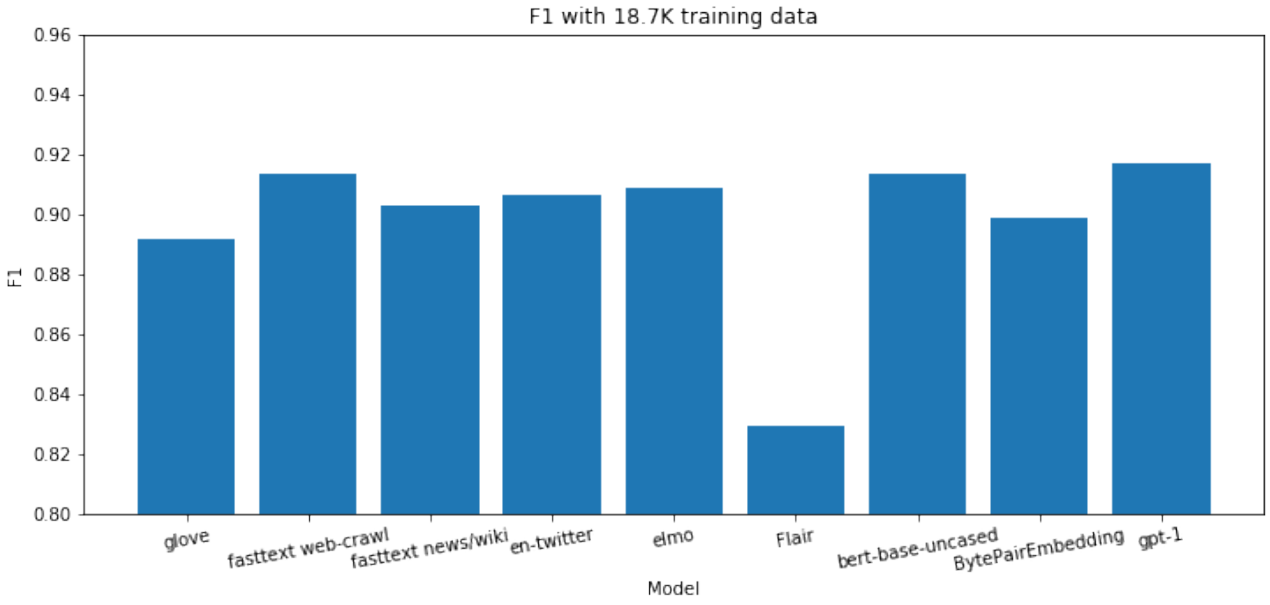

Figure 20: Flair:Test F1 for 18.7K training data

In figure 22 and table 10 are F1-scores for similar scores for Business news.

Hate speech
With max train data of 18.7 k, highest score is GPT-1 with f1 of 0.917 with max data of 18.7k, BERT and Fasttext web-crawl coming very near, 0.003 behind.

Fasttext web-crawl is a surpriser, losing only by slightly at max data and on 3000 train samples even reaching highest score. Though we can see that still at 500 train samples Fasttext is not able to converge, but at 1000 samples it reaches second place.

Bert descending in score: Bert is getting smaller score at 1000 than 500 samples. I sampled new random data and tried a bit different batch sizes (501 and 1001) but same phenomena remained - f1 goes down from 500 to 1000. Also in news dataset Bert is falling in score from 1000 to 3000 samples. Possible reasons could include a bug in algorithm or bug in Flair implementation of Bert (more likely). Despite of this, with max training data BERT is still reaching score very near the winner, GPT-1.

Business news
Fasttext (web-crawl) that uses subword units: character n-grams was very strong in Twitter data with unclean text and mistyped words. Here with well formed and edited text and correct spelling of words, it does weaker and is around bottom third of models. Fasttext (news/wiki) turned out to be among bottom ones in both data.

Surprisingly from two fasttexts, the news/wiki version is doing weaker than the web-crawl version, even when target data is that of business news!

Big winner in Business news is Elmo. One possible explanation being that similarly as Fasttext don't gain from its subword information.

Also
Character based models - fasttext web-crawl, fasttext news/wiki and also en-twitter, which were doing strongly with tweets, are here doing weaker with well edited news-data.

Very small training data

With 100 labeled items many models are not able yet to increase almost at all from the majority class.

With small amount of labeled data, three models stand out: GPT-1 is a clear winner. Bert and Elmo coming behind. At 500 labeled samples most traditional models are not yet converging much.

Fasttext web-crawl is at 500 labeled data among weakest, but at 3000 it has reached the highest F1.

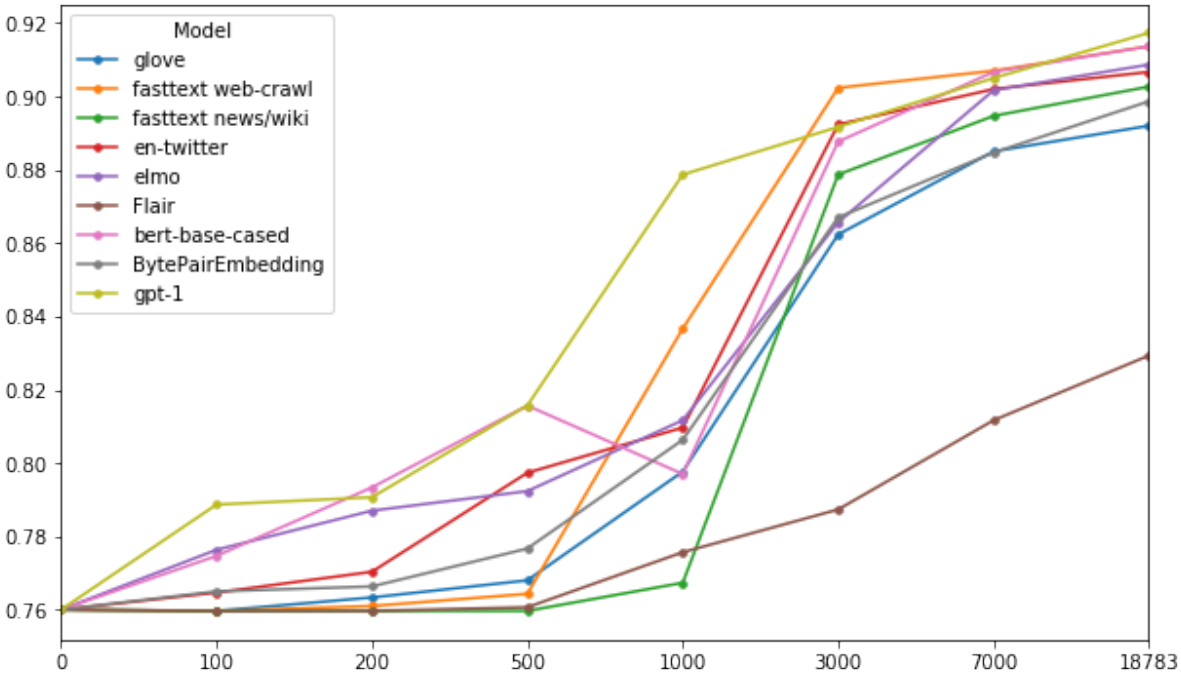

Figure 21: F1 per train samples: Hate speech / Tweets

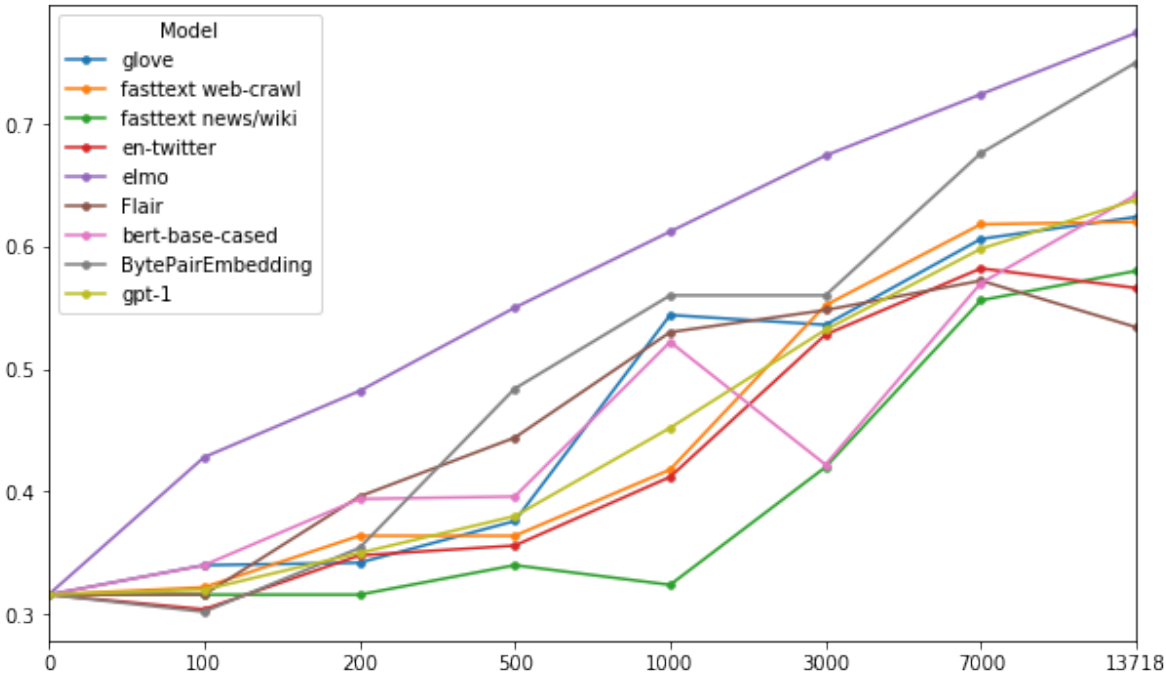

Figure 22: F1 per train samples: Business new

Model	0	100	200	500	1000	3000	7000	18783
glove	0.760	0.760	0.763	0.768	0.798	0.862	0.885	0.892
fasttext web-crawl	0.760	0.760	0.761	0.764	0.837	0.902	0.907	0.914
fasttext news/wiki	0.760	0.760	0.760	0.760	0.767	0.879	0.895	0.903
en-twitter	0.760	0.765	0.770	0.797	0.810	0.892	0.902	0.907
elmo	0.760	0.776	0.787	0.792	0.812	0.866	0.902	0.909
Flair	0.760	0.760	0.760	0.761	0.775	0.787	0.812	0.829
bert-base-cased	0.760	0.775	0.793	0.816	0.797	0.888	0.907	0.914
BytePairEmbedding	0.760	0.765	0.766	0.777	0.806	0.867	0.885	0.899
gpt-1	0.760	0.789	0.791	0.816	0.879	0.892	0.905	0.917

Table 9: F1 per amount of training data - hate speech. F1 for 0 is Majority class, without model, for a baseline.

Model	0	100	200	500	1000	3000	7000	13718
glove	0.316	0.340	0.342	0.376	0.544	0.536	0.606	0.624
fasttext web-crawl	0.316	0.322	0.364	0.364	0.418	0.552	0.618	0.620
fasttext news/wiki	0.316	0.316	0.316	0.340	0.324	0.420	0.556	0.580
en-twitter	0.316	0.304	0.348	0.356	0.412	0.528	0.582	0.566
elmo	0.316	0.428	0.482	0.550	0.612	0.674	0.724	0.774
Flair	0.316	0.316	0.396	0.444	0.530	0.548	0.572	0.534
bert-base-cased	0.316	0.340	0.394	0.396	0.522	0.422	0.570	0.642
BytePairEmbedding	0.316	0.302	0.354	0.484	0.560	0.560	0.676	0.750
gpt-1	0.316	0.320	0.350	0.380	0.452	0.532	0.598	0.638

Table 10: F1 per amount of training data - Business news. Col 0 is Majority class.

How much labeled training data is needed?

Hate speech

With these data sets, 100 or 200 samples of training data does not do much. But even there the LM:s, Elmo, Bert, GPT-1 do show noticeable improvement (from 0.76 to 0.78 f1) where earlier models do not. On 1000 samples GPT-1 reaches 0.88 which we can consider already a good result - compared to max f1 of 0.917 with 18k training data.

So here 1000 train samples of annotated data seems enough to produce a working model.

News data

If looking at 1000 samples "strong point" from hate speech, here in news also the 1000 training samples give 0.612 f1. From baseline of 0.32 to max 0.774 at 13.7k train data, we can see that 0.612 is already a pretty good score so we can conclude that with these data sets **1000 train samples of annotated data** could give a working model, useful in real task.

How do models perform compared to each other?

The more modern models - BERT and GPT-1, which have bigger model sizes and larger pretraining data, already with 100 samples of training data show increase in performance. Bert with increase of 0.015 from 0.76 to 0.775 and GPT-1 even more, 0.029 from 0.76 to 0.789. This is noticeably better than word embedding based ones. We can see that models based on context free embedding need higher amount of training data.

Elmo does strong with 100 training data, even outperforming slightly Bert, but with larger amounts of training data, Elmo falls behind. The Flair embedding from Flair framework turned out to be a low performer in this type of data.

7.5 Limitations on results

Different language models do gain from different preprocessing of data. While it is possible and I did fine tuning for several individual models, for a comparison of larger amount of models this would get too time consuming and add the size of the own framework.

If one were to choose individual model, experiment with different preprocessing for the data, different model parameters and different ways of training, one likely could improve F1 in several of these models.

To be noted, the goal in the comparisons is to compare models to each other - not to reach the maximum possible performance with the test data sets. The "maximum training data" in many tests is samples minus 6000: 3000 for dev and 3000 for test set.

So reader should not take F1 score of GPT-1 for example as a measure of maximum or top performance of GPT-1 can do in this data set (except as a lower bound). Using larger share of the data set for training data, tuning the parameters and size of the model, training larger amount of models and taking average, would bring

noticeable improvement in f1 score.

On the other hand, for either production or scientific use, if working with multiple language models, this also comes as practical issue - how well a specific model can do without too much extra effort put into fine tuning it in different ways from others. How well does it do out of the box?

8 Conclusions

We saw how LM can be defined. How Transfer learning can be defined and concrete data sets represented formally as transfer learning.

On testing models with small amounts of training data, modern language models such as Bert and GPT-1 do well already with small training data. Yet 100 labeled samples is not yet enough for this kind of data, also including imbalance between classes.

For language models, possible areas likely to see more improvement in the future include:
- different kind subword or language units used as basic token.
- different kind and multiple pretraining tasks
- bigger model sizes as well as bigger and more varied training data

References

ABV18 Akbik, A., Blythe, D. and Vollgraf, R., Contextual string embeddings for sequence labeling. *COLING 2018, 27th International Conference on Computational Linguistics*, 2018, pages 1638–1649.

Ala18 Alammar, J., The illustrated transformer, 2018. URL `http://jalammar.github.io/illustrated-transformer/`. visited on 2020-09-30.

ANC07 Arnold, A., Nallapati, R. and Cohen, W. W., A comparative study of methods for transductive transfer learning. *Seventh IEEE International Conference on Data Mining Workshops (ICDMW 2007)*, 2007, pages 77–82.

AR19 Alec Radford, Jeffrey Wu, D. A. D. A. J. C. M. B. I. S., Better language models and their implications, 2019. URL `https://openai.com/blog/better-language-models/`.

BCV13 Bengio, Y., Courville, A. and Vincent, P., Representation learning: A review and new perspectives. *IEEE transactions on pattern analysis and machine intelligence*, 35,8(2013), pages 1798–1828. URL `http://arxiv.org/abs/1206.5538`.

BDVJ03 Bengio, Y., Ducharme, R., Vincent, P. and Jauvin, C., A neural probabilistic language model. *JOURNAL OF MACHINE LEARNING RESEARCH*, 3, pages 1137–1155.

BGJM16 Bojanowski, P., Grave, E., Joulin, A. and Mikolov, T., Enriching word vectors with subword information. *arXiv preprint arXiv:1607.04606*.

CYK+18 Cer, D., Yang, Y., Kong, S., Hua, N., Limtiaco, N., St. John, R., Constant, N., Guajardo-Cespedes, M., Yuan, S., Tar, C., Sung, Y., Strope, B. and Kurzweil, R., Universal sentence encoder. *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, Brussels, Belgium, nov 2018, Association for Computational Linguistics, pages 169–174, URL `https://www.aclweb.org/anthology/D18-2029`.

DCLT18 Devlin, J., Chang, M.-W., Lee, K. and Toutanova, K., Bert: Pre-training of deep bidirectional transformers for language understanding, 2018. URL `http://arxiv.org/abs/1810.04805`.

DWMW17 Davidson, T., Warmsley, D., Macy, M. and Weber, I., Automated hate speech detection and the problem of offensive language. *Proceedings of the 11th International AAAI Conference on Web and Social Media*, ICWSM '17, 2017, pages 512–515, URL `https://aaai.org/ocs/index.php/ICWSM/ICWSM17/paper/view/15665`.

Gag94 Gage, P., A new algorithm for data compression. *The C Users Journal*, Volume 12, Issue 2.

gh gojomo (https://stats.stackexchange.com/users/119339/gojomo), Why the skip-gram model is called as predicting source context words from the target word?, Cross Validated. URL `https://stats.stackexchange.com/q/284860`.

HR18 Howard, J. and Ruder, S., Universal language model fine-tuning for text classification. *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Melbourne, Australia, jul 2018, Association for Computational Linguistics, pages 328–339.

HS18 Heinzerling, B. and Strube, M., Bpemb: Tokenization-free pre-trained subword embeddings in 275 languages. *Proceedings of the Eleventh International Conference on Language Resources and Evaluation (LREC 2018)*, Miyazaki, Japan, May 2018, European Language Resources Association (ELRA), URL https://www.aclweb.org/anthology/L18-1473.

JGBM16 Joulin, A., Grave, E., Bojanowski, P. and Mikolov, T., Bag of tricks for efficient text classification. *arXiv preprint arXiv:1607.01759*.

JM19 Jurafsky, D. and Martin, J. H., *Speech and language processing : an introduction to natural language processing, computational linguistics, and speech recognition (3rd ed draft)*. 2019.

KR18 Kudo, T. and Richardson, J., Sentencepiece: A simple and language independent subword tokenizer and detokenizer for neural text processing. *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, Brussels, Belgium, November 2018, Association for Computational Linguistics, pages 66–71, URL https://www.aclweb.org/anthology/D18-2012.

Kri15 Krishan, Words as vectors, 2015. URL https://iksinc.files.wordpress.com/2015/04/screen-shot-2015-04-10-at-4-16-00-pm.png. visited on 2020-10-28.

LeC19 LeCunn, Y., I now call it 'self-supervised learning', 2019. URL https://www.facebook.com/722677142/posts/10155934004262143/. visited on 2020-09-30.

Li12 Li, Q., Literature survey: Domain adaptation algorithms for natural language processing, 2012.

LM14 Le, Q. V. and Mikolov, T., Distributed representations of sentences and documents. *ICML*, volume 32 of *JMLR Workshop and Conference Proceedings*. JMLR.org, 2014, pages 1188–1196.

Mat18 Matsakis, L., To break a hate-speech detection algorithm, try 'love', 2018. URL https://www.wired.com/story/break-hate-speech-algorithm-try-love/. visited 2020-09-30.

MBXS17 McCann, B., Bradbury, J., Xiong, C. and Socher, R., Learned in translation: Contextualized word vectors. *Advances in Neural Information Processing Systems 30*. Curran Associates, Inc., 2017, pages 6294–6305.

MBXS18 McCann, B., Bradbury, J., Xiong, C. and Socher, R., Cove - learned in translation. *NIPS*.

MCCD13 Mikolov, T., Chen, K., Corrado, G. and Dean, J., Efficient estimation of word representations in vector space, 2013. URL `https://arxiv.org/pdf/1301.3781.pdf`.

MKS17 Merity, S., Keskar, S. and Socher, R., Regularizing and optimizing lstm language models. *CoRR*, abs/1708.02182.

MKXS18 McCann, B., Keskar, N. S., Xiong, C. and Socher, R., The natural language decathlon: Multitask learning as question answering, 2018. URL `http://arxiv.org/abs/1806.08730`.

MRS08 Manning, C. D., Raghavan, P. and Schütze, H., *Introduction to Information Retrieval*. Cambridge, UK, 2008. URL `http://nlp.stanford.edu/IR-book/information-retrieval-book.html`.

MSC+13 Mikolov, T., Sutskever, I., Chen, K., Corrado, G. S. and Dean, J., Distributed representations of words and phrases and their compositionality. In *NIPS*, Curran Associates, Inc., 2013, pages 3111–3119, URL `http://papers.nips.cc/paper/5021-distributed-representations-of-words-and-phrases-and-their-compositionality.pdf`.

Ng17 Ng, A., Deep learning specialization, 2017. URL `https://www.coursera.org/specializations/deep-learning`.

PKY18 Pivovarova, L., Klami, A. and Yangarber, R., Benchmarks and models for entity-oriented polarity detection. *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 3 (Industry Papers)*, New Orleans - Louisiana, June 2018, Association for Computational Linguistics, pages 129–136, URL `https://www.aclweb.org/anthology/N18-3016`.

PNI+18 Peters, M., Neumann, M., Iyyer, M., Gardner, M., Clark, C., Lee, K. and Zettlemoyer, L., Deep contextualized word representations, 2018.

PSM14 Pennington, J., Socher, R. and Manning, C. D., Glove: Global vectors for word representation. *In EMNLP*, 2014.

PY10 Pan, S. and Yang, Q., A Survey on Transfer Learning. *IEEE Transactions on Knowledge and Data Engineering*, 22,10(2010), pages 1345–1359.

RJS17 Radford, A., Jozefowicz, R. and Sutskever, I., Learning to generate reviews and discovering sentiment, 2017. URL `http://arxiv.org/abs/1704.01444`. cite arxiv:1704.01444.

RNSS18 Radford, A., Narasimhan, K., Salimans, T. and Sutskever, I., Improving language understanding by generative pre-training. *Preprint*.

Ron14 Rong, X., word2vec parameter learning explained. *CoRR*, abs/1708.02182.

Rud19 Ruder, S., Neural transfer learning for natural language processing, 2019.

RWC+19 Radford, A., Wu, J., Child, R., Luan, D., Amodei, D. and Sutskever, I., Language models are unsupervised multitask learners.

Sch Schweizer, C., What is the difference between taxonomy and ontology? URL `https://www.earley.com/blog/what-difference-between-taxonomy-and-ontology-it-matter-complexity`. visited on 2020-09-30.

SHB15 Sennrich, R., Haddow, B. and Birch, A., Neural machine translation of rare words with subword units. *CoRR*, abs/1508.07909.

Sin19 Singer, P., 1st place solution, 2019. URL `https://www.kaggle.com/c/quora-insincere-questions-classification/discussion/80568#472178`.

Smi15 Smith, L., Cyclical learning rates for training neural networks, 2015. URL `http://arxiv.org/abs/1506.01186`.

VSU+17 Vaswani, A., Shazeer, N.and Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L. and Polosukhin, I., Attention is all you need. *Advances in Neural Information Processing Systems*, 2017, pages 5998–6008.

WAS+18 Wang, Alex, Singh, A., Michael, J., Hill, F., Levy, O. and Bowman, S., Glue: A multi-task benchmark and analysis platform for natural language understanding. *Proceedings of the 2018 EMNLP Workshop BlackboxNLP: Analyzing and Interpreting Neural Networks for NLP*, Brussels, Belgium, November 2018, Association for Computational Linguistics, pages 353–355.

Wen18 Weng, L., Attention? attention! URL http://lilianweng.github.io/lil-log/2018/06/24/attention-attention.html.

ZB18 Zhang, K. and Bowman, S., Language modeling teaches you more than translation does: Lessons learned through auxiliary syntactic task analysis. *Proceedings of the 2018 EMNLP Workshop BlackboxNLP: Analyzing and Interpreting Neural Networks for NLP*, Brussels, Belgium, November 2018, Association for Computational Linguistics, pages 359–361, URL https://www.aclweb.org/anthology/W18-5448.

Appendix 1. Finnish language NLP resources

Many NLP resources are made primarily for English language. Things like tokenization and lemmatization that for English language work easily out of the box with ready libraries, require more work for Finnish language. Also for using pretrained models/weights, one needs model that is pretrained on Finnish language data or a multilingual model where Finnish is among the languages.

Here is NLP resources for Finnish language.

Tokenization

While Spacy is a commonly used tokenizer for English, for agglunative / agglomerative languages like Finnish or Chinese, Sentence-Piece tokenizer is preferred.

mpBPE has Pre-trained subword embeddings based on Byte-Pair Encoding (BPE) for 275 languages - including Finnish. `https://github.com/bheinzerling/bpemb`

Parsers

More traditional NLP for Finnish language include Turku-NLP Finnish dependency parser pipeline, which include tokenization, sentence splitting, lemmatization, morphological tagging and dependency parsing. `https://turkunlp.org/finnish_nlp.html`

Voikko has Finnish lemmatization. `https://voikko.puimula.org/`

Language models with pre-trained model for Finnish language

Zalandos Flair framework contains pre-trained Flair embeddings for multiple languages.
embedding-id 'fi-X' is for Finnish. (Trained on Wikipedia/Opus)
flair_fi = FlairEmbeddings(fi-X)
`https://github.com/zalandoresearch/flair/blob/master/resources/docs/TUTORIAL_4_ELMO_BERT_FLAIR_EMBEDDING.md`

BERT

BERTs multilingual version also includes Finnish. Trained on Wikipedia.
`https://github.com/google-research/bert/blob/master/multilingual.md`

TurkuNLP Bert
`https://github.com/TurkuNLP/FinBERT`

"Multilingual is not enough: BERT for Finnish"
`https://arxiv.org/abs/1912.07076`

Other

NLTK includes common stopwords for finnish language

$importnltk$

$nltk.download('stopwords')$

$from\ nltk.corpus\ import\ stopwords$

$stopwords = stopwords.words('finnish')$