Text Classification

Língua Natural e Sistemas Conversacionais

Luiz Faria

Adapted from Natural Language Processing course from HSE University

Text Classification Tasks

- Predict some tags or categories
- Predict sentiment for a review
- Filter spam e-mail

Text Classification Tasks

Example: Sentiment Analysis

- Input: text of review
- Output: class of sentiment
  - e.g. 2 classes: positive vs negative
- Positive example:
  - The hotel is really beautiful. Very nice and helpful service at the front desk.
- Negative example:
  - We had problems to get the Wi-Fi working. The pool area was occupied with young party animals. So the area wasn't fun for us.

Text Preprocessing

What is Text?

We can think of text as a sequence of

- Characters
- **Words**
- Phrases and named entities
- Sentences
- Paragraphs

What is a Word?

It seems natural to think of a text as a sequence of words

- A word is a meaningful sequence of characters

How to find the boundaries of words?

- In Portuguese or English we can split a sentence by spaces or punctuation

---

Input: Friends, Romans, Countrymen, lend me your ears
Output: Friends | Romans | Countrymen | lend | me | your | ears

---

- In German there are compound words which are written without spaces
  - "Rechtsschutzversicherungsgesellschaften" stands for "insurance companies which provide legal protection"
- In Japanese there are no spaces at all!
  - Butyoucanstillreaditright?

Tokenization

Tokenization is a process that splits an input sequence into so-called tokens

- You can think of a token as a useful unit for semantic processing
- Can be a word, sentence, paragraph, etc.

An example of simple whitespace tokenizer

- nltk.tokenize.WhitespaceTokenizer
  This | is | Andrew's | text, | isn't | it?
- Problem: "it" and "it?" are different tokens with same meaning

> We might want to merge "it" and "it?" because they have essentially the same meaning. The same for "text,", as it is the same token as "text"

Tokenization

Let's try to also split by punctuation

- nltk.tokenize.WordPunctTokenizer
  This | is | Andrew | ' | s | text | , | isn | ' | t | it | ?
- Problem: "s", "isn", "t" are not very meaningful

We can come up with a set of heuristic rules

- nltk.tokenize.TreebankWordTokenizer
  This | is | Andrew | 's | text | , | is | n't | it | ?
- "'s" and "n't" are more meaningful for processing

Python tokenization example

```
import nltk
text = "This is Andrew's text, isn't it?"
```

```
tokenizer = nltk.tokenize.WhitespaceTokenizer()
tokenizer.tokenize(text)
```

```
['This', 'is', "Andrew's", 'text,', "isn't", 'it?']
```

```
tokenizer = nltk.tokenize.TreebankWordTokenizer()
tokenizer.tokenize(text)
```

```
['This', 'is', 'Andrew', "'s", 'text', ',', 'is', "n't", 'it', '?']
```

```
tokenizer = nltk.tokenize.WordPunctTokenizer()
tokenizer.tokenize(text)
```

```
['This', 'is', 'Andrew', "'", 's', 'text', ',', 'isn', "'", 't', 'it',
 '?']
```

http://text-processing.com/demo/tokenize/

Token Normalization

We may want the same token for different forms of the word

- wolf, wolves $\rightarrow$ wolf
- talk, talks $\rightarrow$ talk

Stemming

- A process of removing and replacing suffixes to get to the root form of the word, which is called the **stem**
- Usually refers to heuristics that chop off suffixes

Lemmatization

- Usually refers to doing things properly with the use of a vocabulary and morphological analysis
- Returns the base or dictionary form of a word, which is known as the **lemma**

Stemming Example

Porter's stemmer

- 5 heuristic phases of word reductions, applied sequentially
- Example of phase 1 rules:

Rule		Example	
SSES	$\rightarrow$ *SS*	*caresses*	$\rightarrow$ *caress*
IES	$\rightarrow$ *I*	*ponies*	$\rightarrow$ *poni*
SS	$\rightarrow$ *SS*	*caress*	$\rightarrow$ *caress*
S	$\rightarrow$	*cats*	$\rightarrow$ *cat*

- nltk.stem.PorterStemmer
- Examples:

feet $\rightarrow$ *feet* *cats* $\rightarrow$ *cat*

wolves $\rightarrow$ *wolv* *talked* $\rightarrow$ *talk*

- Problem: fails on irregular forms, produces non-words

Lemmatization Example

WordNet lemmatizer

- Uses the WordNet Database to lookup lemmas
- nltk.stem.WordNetLemmatizer
- Examples:

$$feet \rightarrow foot \qquad cats \rightarrow cat$$

$$wolves \rightarrow wolf \qquad talked \rightarrow talked$$

- Problems: not all forms are reduced
- Takeaway: we need to try stemming or lemmatization and choose best for our task

Python Stemming Example

```
import nltk
text = "feet cats wolves talked"
tokenizer = nltk.tokenize.TreebankWordTokenizer()
tokens = tokenizer.tokenize(text)
```

```
stemmer = nltk.stem.PorterStemmer()
" ".join(stemmer.stem(token) for token in tokens)
```

```
u'feet cat wolv talk'
```

```
stemmer = nltk.stem.WordNetLemmatizer()
" ".join(stemmer.lemmatize(token) for token in tokens)
```

```
u'foot cat wolf talked'
```

Further Normalization

Normalizing capital letters

- Us, us → us (if both are pronoun)
- us, US (could be pronoun and country)
- We can use heuristics:
  - lowercasing the beginning of the sentence
  - lowercasing words in titles
  - leave mid-sentence words as they are
- Or we can use machine learning to retrieve true casing → hard

Acronyms

- eta, e.t.a., E.T.A. → E.T.A.
- We can write a large set of regular expressions → hard

Summary

- We can think of text as a sequence of tokens
- Tokenization is a process of extracting those tokens
- We can normalize tokens using stemming or lemmatization
- We can also normalize casing and acronyms
- Next steps: we will transform extracted tokens into features for our model of text classification

Transforming tokens into features

Bag of Words (BOW)

Let's count occurrences of a particular token in our text

- Motivation: we're looking for marker words like "excellent" or "disappointed", and make decisions based on absence or presence of that particular word
- For each token we will have a feature column, this is called **text vectorization**

good movie
not a good movie
did not like

good	movie	not	a	did	like
1	**1**	0	0	0	0
1	**1**	**1**	**1**	0	0
0	0	**1**	0	**1**	**1**

- Problems:
  - we loose word order, hence the name "bag of words"
  - counters are not normalized
  - huge sparse vectors

Preserving Tokens' Ordering

We can count token pairs, triplets, etc.

- Also known as n-grams
  - 1-grams for tokens
  - 2-grams for token pairs
  - ...
- Now we don't only have columns that correspond to tokens, but we have also columns that correspond to token pairs:

good movie
not a good movie
did not like

good movie	**movie**	**did not**	**a**	**...**
1	**1**	0	0	...
1	**1**	0	**1**	...
0	0	**1**	0	...

- This way, we preserve some local word order
- Problems:
  - Too many features → to solve this we can remove some n-grams ...

Remove Some N-grams

Let's remove some n-grams from features based on their occurrence frequency in documents of our corpus

- **High frequency n-grams**:
  - Articles, prepositions, etc. (example: and, a, the)
  - They are called **stop-words**, they won't help us to discriminate texts → remove them
- **Low frequency n-grams**:
  - Typos, rare n-grams
  - We don't need them either, otherwise we will likely overfit
- **Medium frequency n-grams**:
  - Those are good n-grams

Medium Frequency N-grams Are Very Frequent

- It proved to be useful to look at n-gram frequency in our corpus for filtering out bad n-grams
- What if we use it for ranking of medium frequency n-grams?
- **Idea**: the n-gram with smaller frequency can be more discriminating because it can capture a specific issue in the review

> Suppose that somebody is not happy with the Wi-Fi and it says, "Wi-Fi breaks often", and that n-gram, "Wi-Fi breaks", it is not very frequent in our corpus, but it can actually highlight a specific issue with important meaning.

- To use that idea, we must introduce the notion of **Term Frequency**

TF-IDF

Term frequency (TF)

- $tf(t, d)$ – frequency for term (token or n-gram) t in document d
- Variants:

weighting scheme	TF weight
binary	$0, 1$
raw count	$f_{t,d}$
term frequency	$f_{t,d} / \sum_{t' \in d} f_{t',d}$
log normalization	$1 + \log(f_{t,d})$

Binary: 0 or 1 when the token is absent or present in the text

Raw count: count of how many times the term is present in the document

Term frequency: normalized counting such as the sum of the counts of all terms is equal to one

Logarithmic normalization: apply the logarithm to those counts, introducing a logarithmic scale for the counters, which may help to solve the task better

https://en.wikipedia.org/wiki/Tf–idf

TF-IDF

Inverse document frequency (IDF)

- $N = |D|$ - N is the total number of documents in corpus D
- $|\{d \in D : t \in d\}|$ - number of documents em D where the term t appears
- $idf(t, D) = \log \frac{N}{|\{d \in D : t \in d\}|}$

document frequency where term t appears: $\frac{|\{d \in D : t \in d\}|}{N}$

TF-IDF

- $tfidf(t, d, D) = tf(t, d) \cdot idf(t, D)$
- A high weight in TF-IDF is reached by a high term frequency (in the given document) and a low document frequency of the term in the whole collection of documents

That is precisely the idea that we wanted to follow. We wanted to find frequent issues in the reviews that are not so frequent in the whole data-set. So these are specific issues and we want to highlight them. TF-IDF allow to downscale weights for words that occur in many documents in the corpus and are therefore less informative than those that occur only in a smaller portion of the corpus.

Improved BOW

- Replace counters with TF-IDF
- Normalize the result row-wise (divide by $L_2 norm$)

$L_2 norm$: square root of the sum of the squared row values

good movie
not a good movie
did not like

good movie	movie	did not	...
0,17	**0,17**	0	...
0,17	**0,17**	0	...
0	0	**0,47**	...

The two-gram "good movie" appears in two documents. So in this collection it is a very frequent two-gram. That's why the value 0.17 is actually lower than 0.47 and we get 0.47 for "did not" two-gram because it happened only in one review and that could be a specific issue and we want to highlight, meaning that we want to have a bigger value for that feature.

Python TF-IDF Example

```
from sklearn.feature_extraction.text import TfidfVectorizer
import pandas as pd
texts = [
    "good movie", "not a good movie", "did not like",
    "i like it", "good one"
]
tfidf = TfidfVectorizer(min_df=2, max_df=0.5, ngram_range=(1, 2))
features = tfidf.fit_transform(texts)
pd.DataFrame(
    features.todense(),
    columns=tfidf.get_feature_names()
)
```

	good movie	like	movie	not
0	0.707107	0.000000	0.707107	0.000000
1	0.577350	0.000000	0.577350	0.577350
2	0.000000	0.707107	0.000000	0.707107
3	0.000000	1.000000	0.000000	0.000000
4	0.000000	0.000000	0.000000	0.000000

TfidfVectorizer parameters:

min_df: ignore terms that have a document frequency lower than the given threshold [(1)]

max_df: ignore terms that have a document frequency strictly higher than the given threshold (corpus-specific stop words) [(1)]

ngram_range: what n-grams should be used

[(1)] If float in range [0.0, 1.0], the parameter represents a proportion of documents

The less frequent one-grams and two-grams were filtered. Each row is normalized to have a norm of one.

Summary

- We've made simple counter features in bag of words manner
- We can add n-grams to try to preserve some local ordering
- We replaced each text by a huge vector of counters
- We can replace counters with TF-IDF values
- Next we will train a first model on top of these features

Linear Models for Sentiment Analysis

Sentiment Classification

IMDB movie reviews dataset

- `http://ai.stanford.edu/~amaas/data/sentiment/`
- Contains 25000 positive and 25000 negative reviews

- Contains at most 30 reviews per movie
- At least 7 stars out of 10 $\rightarrow$ positive (label $= 1$)
- At most 4 stars out of 10 $\rightarrow$ negative (label $= 0$)
- 50/50 train/test split: future researchers can use the same split and reproduce their results to compare models
- Evaluation: accuracy

Accuracy is appropriated because we have the same number of positive and negative reviews; so the dataset is balanced in terms of the size of the classes

Sentiment Classification

Let's take bag of 1-grams with TF-IDF values

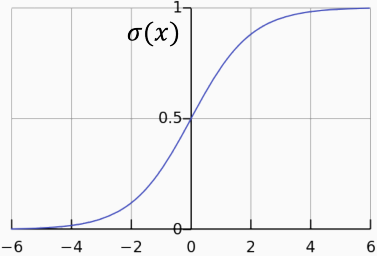
- 25000 rows, 74849 columns for training
- We get a extremely sparse feature matrix – 99.8% are zeros - This sparcity applies some restrictions on the models that we can use with these features
  - One model that is usable for these features is **logistic regression**

acting	actingjob	actings	actingwise
0.000000	0.0	0.0	0.0
0.000000	0.0	0.0	0.0
0.053504	0.0	0.0	0.0
0.033293	0.0	0.0	0.0
0.000000	0.0	0.0	0.0

Sentiment Classification

Model: Logistic regression

- $p(y = 1|x) = \sigma(\omega^T x)$
- Linear classification model
- Can handle sparse data
- Fast to train
- Weights can be interpreted



The model tries to predict the probability of a review being positive given the features that we gave to that model for that particular review. This features are the vector of TF-IDF values. The training process find the weight for every feature of that bag of words. Then, each TF-IDF value is multiplied by that weight, and the sum all of that products pass through a sigmoid activation function. Being a linear classification model, it can handle sparse data. If we get a linear combination that is close to 0, that means that sigmoid will output 0.5. So the probability of a review being positive is 0.5.

Sentiment Classification

Logistic regression over bag of 1-grams with TF-IDF

- Accuracy on test set: 88.5%
- Let's look at learnt weights:

ngram	weight
great	9.042803
excellent	8.487379
perfect	6.907277
best	6.440972
wonderful	6.237365

Top positive

VS

ngram	weight
worst	-12.748257
awful	-9.150810
bad	-8.974974
waste	-8.944854
boring	-8.340877

Top negative

Improving Sentiment Classification

Let's try to add 2-grams (1 and 2-ngrams)

- Throw away n-grams seen less than 5 times (e.g. typos)
- 25000 rows, 156821 columns for training (matrix of TF-IDF values)

and am	and amanda	and amateur	and amateurish	and amazing
0.068255	0.0	0.0	0.0	0.0
0.000000	0.0	0.0	0.0	0.0
0.000000	0.0	0.0	0.0	0.0
0.000000	0.0	0.0	0.0	0.0
0.000000	0.0	0.0	0.0	0.0

Improving Sentiment Classification

Logistic regression over bag of 1,2-grams with TF-IDF

- Accuracy on test set: 89.9% (+1.5%)
- Let's look at learnt weights:

Introducing 2-ngrams made the model consider 2-ngrams as top positive (such as "well worth")

well worth	13.788515		bad	-24.467648
best	13.633200		poor	-24.319746
rare	13.570259	VS	the worst	-23.773352
better than	13.500025		waste	-22.880340
Top positive			Top negative	

How to Improve it Better

Play around with tokenization

- Special tokens like emoji, ":)" and "!!!" can help

Try to normalize tokens

- Adding stemming or lemmatization

Try different models

- SVM, Naïve Bayes, ..., that can handle sparse features

Throw BOW away and use Deep Learning

- `https://arxiv.org/pdf/1512.08183.pdf`
- Accuracy on this test set in 2016: 92.14% (+2.5%)

Summary

- Bag of words and simple linear models actually work for texts
- The accuracy gain from deep learning models is not mind blowing for sentiment classification
- Next we'll look at spam filtering task

Spam Filtering Task

Mapping n-grams to Feature Indices

If your dataset is small you can store {n-gram $\rightarrow$ feature index} in hash map or Python dictionary

But if you have a huge dataset that can be a problem

- Let's say we have 1 TB of texts distributed on 10 computers
- You need to vectorize each text
- You will have to maintain {n-gram $\rightarrow$ feature index} mapping
  - May not fit in memory on one machine
  - Hard to synchronize
- An easier way is **hashing**: {n-gram $\rightarrow$ hash(n-gram)%2^{20}}
  - Has collisions but works in practice
  - sklearn.feature_extraction.text.**HashingVectorizer**
  - Implemented in **vowpal wabbit library**

Increase the power of 2 until collisions can be neglected

Spam Filtering: A Huge Task

Spam filtering proprietary dataset

- `https://arxiv.org/pdf/0902.2206.pdf`
- 0.4 million users
- 3.2 million letters
- 40 million unique words

Let's say we map each token x to index using hash function ϕ

- $\phi(x) = hash(x)\%2^b$
- For $b = 22$ we have 4 million features
- That is a huge improvement over 40 million features (we mapped 40 millions of features into 4 $\rightarrow$ 10 times less)
- It turns out it doesn't hurt the quality of the model

As hash collisions are very unlikely, we can actually train the model on top of that formula and features and still get the same pretty decent result

Hashing Example

- $\phi(good) = 0$
- $\phi(movie) = 1$
- $\phi(not) = 2$
- $\phi(a) = 3$
- $\phi(did) = 3$
- $\phi(like) = 4$

Hash collision
(with a small b)

$$hash(s) = s[0] + s[1]p^1 + \ldots + s[n]p^n$$

s – string

p – fixed prime number

$s[i]$ – character code

good movie
not a good movie
did not like

0	**1**	**2**	**3**	**4**
1	**1**	0	0	0
1	**1**	**1**	**1**	0
0	0	**1**	**1**	**1**

Trillion of Features with Hashing

- We actually proposed a way allowing as to squeeze the number of features that we originally had
- So, in a bag-of-words manner, we had 40 million features and if we hash them, then we have four million features
- This way, we can control the number of features that we have in the output by adjusting that b parameter
- This means is that now we can introduce a lot of tokens, a lot of features, trillion features - if we hash them, we still have the fixed number, 2^b of features that we can analyze

Personalized Tokens

Until now the features indices was obtained by: $\phi_o(token) = hash(token)\%2^b$

Now lets consider personalized tokens:

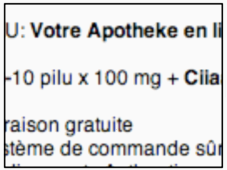
Suppose we have thousands of users U and want to perform related but not identical classification tasks for each of the them. Users provide labeled data by marking emails as spam or not-spam. Ideally, for each user $u \in U$, we want to learn a predictor ω_u based on the data of that user solely. However, webmail users are notoriously lazy in labelling emails and even those that do not contribute to the training data expect a working spam filter. Therefore, we also need to learn an additional global predictor ω_o to allow data sharing amongst all users.

Personalized Tokens

- That means that we want to have a feature that says that for that particular user u, and that particular token, if you see that user and that token in the email, that means that we want to learn some personalized preference of that user in spam or non-spam e-mails
- Let's personalize $u, token$ pairs: we concatenate user id, an underscore and token, and get a new token
- If we take all pairs of $u, word$, we get 16 trillion pairs $\rightarrow$ not possible to look at those features as a bag-of-words representation because it takes 16 terabytes of data
- But we can take the hash of those features and get a fixed number of features: $\phi_u(token) = hash(u + "\_" + token)\%2^b$
- We obtain 16 trillion pairs (user, word) but still 2^b features

Personalized Tokens

text document (email)



bag of words

NEU
Votre
Apotheke
...

bag of words (personalized)

NEU
USER123_NEU
Votre
USER123_Votre
Apotheke
USER123_Apotheke
...

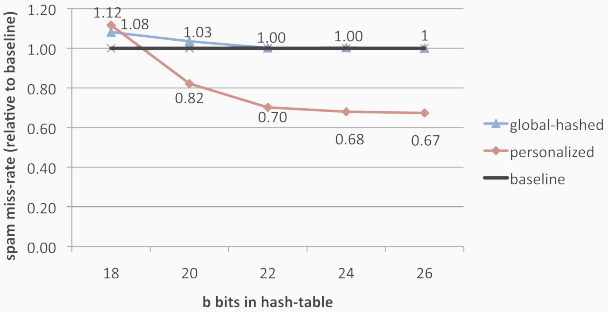
$\phi_o(x) + \phi_u(x)$
hashed, sparse vector

Each token is duplicated and one copy is individualized (e.g. by concatenating each token with a unique user identifier). Then, the global hash function maps all tokens into a low dimensional feature space where the document is classified.

https://arxiv.org/pdf/0902.2206.pdf

Personalized Tokens
Performance

- For $b = 22$ it performs just like a linear model on original tokens
- We observe that personalized tokens give a huge improvement in miss-rate



As baseline, it was chosen a global classifier trained over all users and hashed into 2^{26} dimensional space. As 2^{26} far exceeds the total number of unique words we can regard the baseline to be representative for the classification without hashing.

https://arxiv.org/pdf/0902.2206.pdf

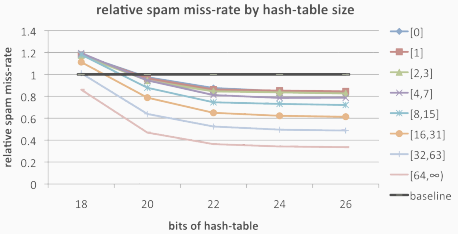
Why Personalized Features Work

Personalized features capture "local" user-specific preference

- Some users might consider newsletters a spam but for the majority of the people they are fine

It turns out we learn better "global" preference having personalized features which learn "local" user preference

- You can think of it as a more universal definition of spam



Number of emails in training

For example, the bucket $[8, 15]$ consists of all users with 8 to 15 training emails. Although users in buckets with large amounts of training data do benefit more from the personalized classifier (up to 65% reduction in spam), even users that did not contribute to the training corpus at all obtain almost 20% spam-reduction.

Why Dataset Size Matters

Why do we need such huge datasets?

- It turns out you can learn better models using the same simple linear classifier

Techniques do deal with trillions of features, billions of training examples:

- `https://arxiv.org/pdf/1110.4198.pdf`

Data sampling: noticeable drop in accuracy after subsampling

Vowpal Wabbit

- A popular machine learning library for training linear models
- Uses feature hashing internally
- Has lots of features
- Really fast and scales well

VOWPAL WABBIT

https://github.com/JohnLangford/vowpal_wabbit/wiki

Summary

- We've taken a look on applications of feature hashing
- Personalized features is a nice trick
- Linear models over bag of words scale well for production
- Next we'll take a look at text classification problem using deep learning

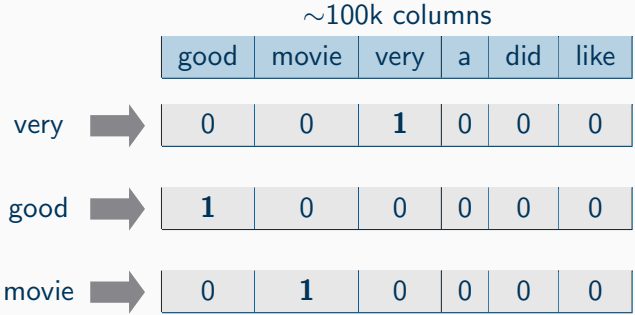
Neural Networks for Text

What is Text

We can think of text as a sequence of

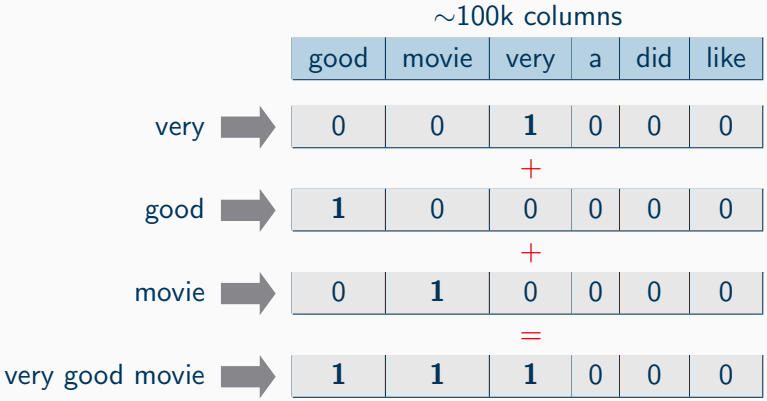
- Characters
- **Words** or **tokens**
- Phrases and named entities
- Sentences
- Paragraphs
- ...

Remembering Bag of Words Way (Sparse)



For each word we have a feature column. Each word is vectorized with one-hot-encoded vector that is a huge vector of zeros that has only one non-zero value which is in the column corresponding to that particular word. So in this example, we have very, good, and movie, and all of them are vectorized independently.

Remembering Bag of Words Way (Sparse)

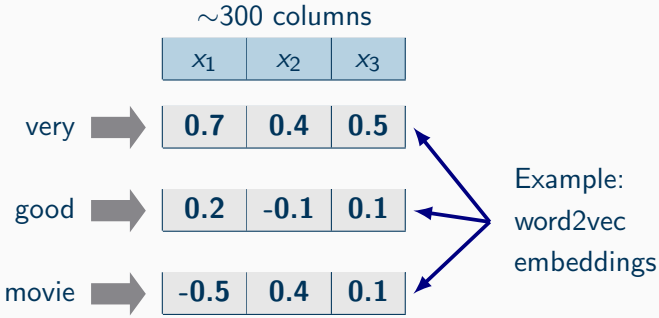


Bag of words representation
is a sum of sparse one-hot-encoded vectors

For real world problems, we can get hundreds of thousands of columns. To get a BOW representation we can sum all those vectors, obtaining a BOW vectorization that corresponds to "very good movie". So, the BOW representation can represent a sum of sparse one-hot-encoded vectors corresponding to each particular word.

Neural Way (Dense)

Opposite to a sparse representation like in BOW, now we have a dense representation - each word is replaced by a shorter real dense vector. word2vec is an example of such vectors embeddings, that are pre-trained in an unsupervised manner (details further ahead). Words that have similar context in terms of neighboring words, tend to have vectors that are collinear.

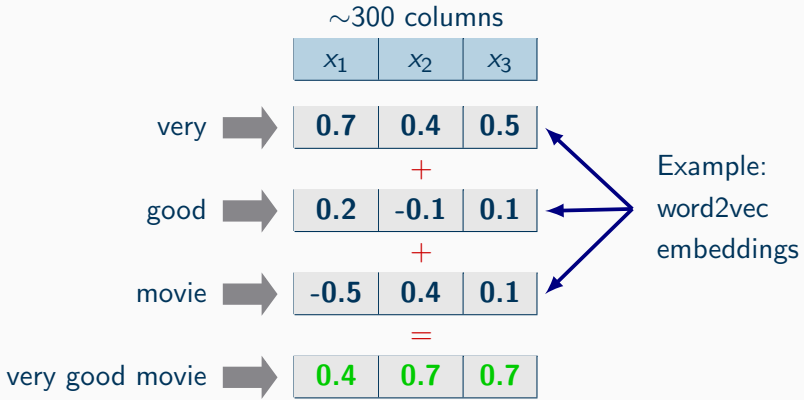


Word2vec property:

Words that have similar context tend to have collinear vectors

Neural Way (Dense)

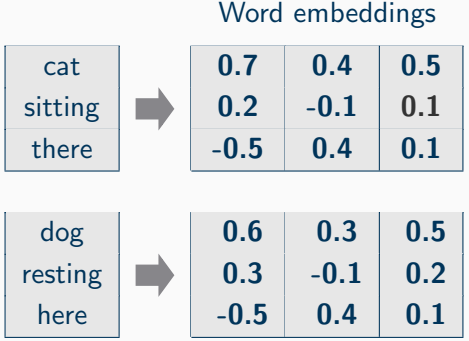
How to use these dense vectors of 300 real values to build a feature descriptor for the whole text? We can sum those vectors.



Sum of word2vec vectors
can be a good text descriptor already

1D Convolutions to Improve Performance

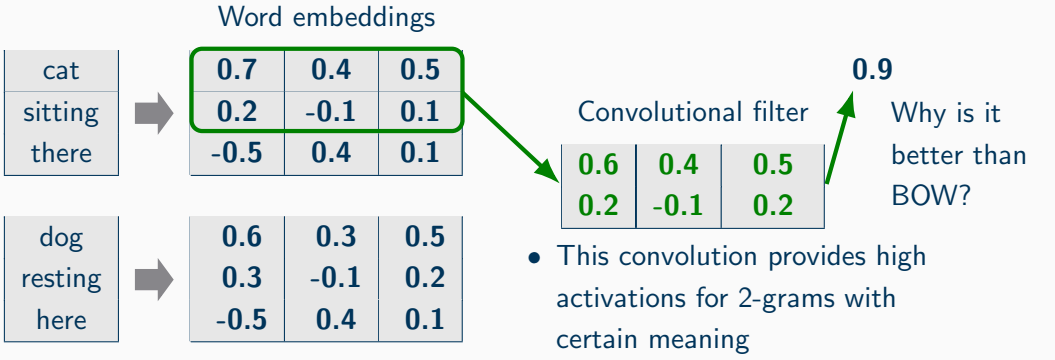
Another approach is applying a neural network over these embeddings. Let's look at two examples: "cat sitting there" and "dog resting here" - for each word we have a row representing a word2vec embedding of length 300.



How do we make 2-grams?

How do we make use of 2-grams using this representation? In BOW representation, for each particular 2-gram, we had a different column, originating a long sparse factor for all possible 2-grams. But here, we don't have word2vec embeddings for token pairs, we have word2vec embeddings only for each particular word. So, how can we analyze 2-grams here?

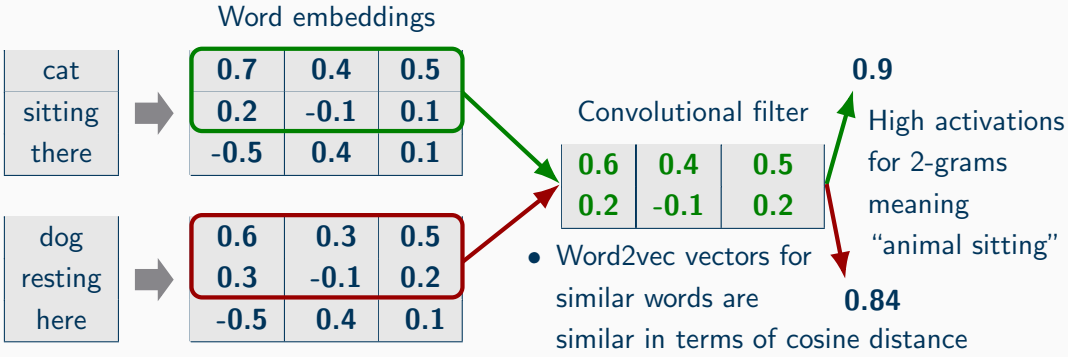
1D Convolutions to Improve Performance



For each pair of adjacent word embeddings (green border) we will apply a convolutional filter with the same size. The filter of the example has values that are close to the values that correspond to "cat sitting". So when we convolve that 2-gram with that filter we get a high activation because the convolutional filter is very similar to the word embeddings of these pair of words. In BOW for each particular 2-gram, we had a different column. Now we have to use a lot of convolutional filters that will learn that representation of 2-grams and we'll be able to analyze 2-grams as well. Why is it better than BOW?

http://bionlp-www.utu.fi/wv_demo/

1D Convolutions to Improve Performance



Word2vec vectors for similar words (in terms of the context that they are seen in) are similar in terms of cosine distance (similar to dot product - convolution). If we apply the same filter to two similar sentences we get high activations in both cases, meaning that the sentences are similar. Why this is better than BOW? If we have good embedding of all of vectors, then using convolutions, we can actually look at more high-level meaning of the two gram. It's not just "cat sitting", or "dog resting", or "cat resting", it's "animal sitting" - that is the meaning of that 2-gram that we can learn with the convolutional filter. We don't need a lot of columns for all possible 2-grams.

`http://bionlp-www.utu.fi/wv_demo/`

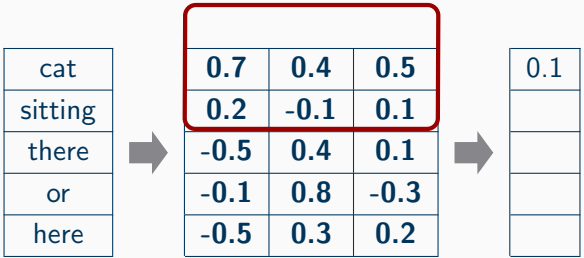
1D Convolutions

- Can be extended to 3-grams, 4-grams, etc.
- One filter is not enough, need to track many n-grams
- They are called 1D because we slide the window only in one direction

It can be extended to three-grams, three-grams and any other n-gram. And contrary to a BOW representation, the feature metrics won't explode, because the feature metrics is actually fixed. All we change is the size of the filter, with which we do convolution. However, one filter is not enough. We need to track many n-grams, many different meanings of those two, three grams and that's why we need a lot of convolutional filters. These filters are called 1D convolutions because we slide the window only in one direction.

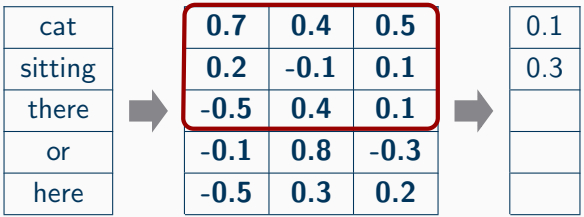
1D Convolutions

- How the sliding window works?

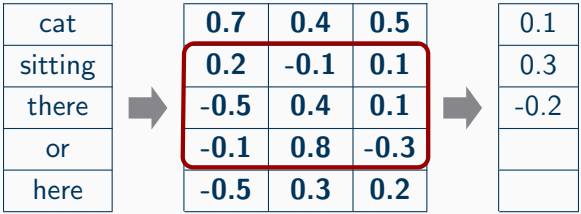


We have an input sequence, "cat sitting there or here", we have for each word a word2vec representation and we have a sliding window of size three. And let's add some padding so that the size of the output is the same as the size of the input. Let's convolve the first patch that we got from these metrics and let's say we get a 0.1, then 0.3, -0.2, 0.7 and -0.4. The sliding direction represents time.

1D Convolutions

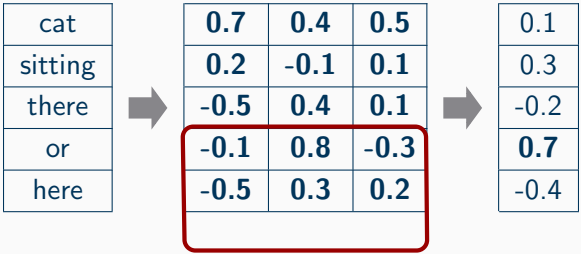


1D Convolutions



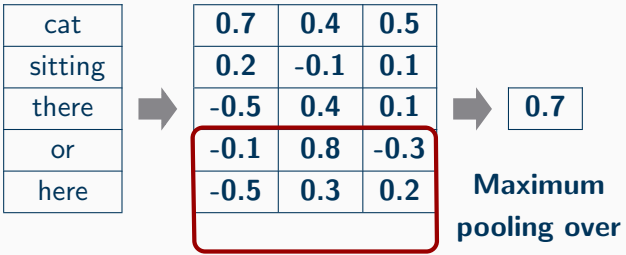
1D Convolutions

1D Convolutions



What do we do with this vector? Let's assume that just like in a bag-of-words manner, we can lose the ordering of the words. That means that we don't really care where a particular 2-gram occur, in the beginning or at the end of the sentence. The only thing we care is whether that 2-gram is in the text or not. Assuming this, we can apply the filter through the whole text and take the maximum activation that we got - Maximum puling over time.

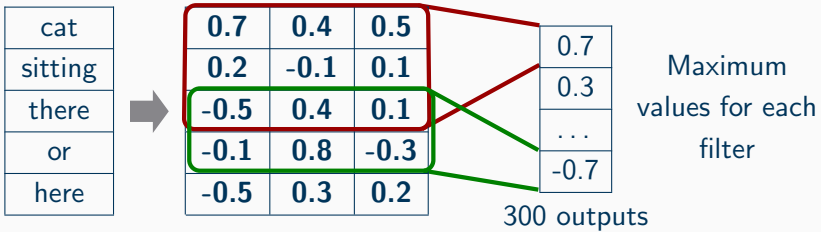
1D Convolutions



Training Several Filters

Final architecture

- 3,4,5-gram windows with 100 filters each
- MLP on top of these 300 features



We can use the filters of size 3, 4 and 5, so that we can capture the information about 3, 4 and 5-grams, and for each n-gram we will learn 100 filters. That means that we have 300 outputs. Applying a filter (red window) to an input sequence we get as maximum activation 0.7. For the green filter, for two-grams, if we convolve it throughout the whole sentence, then the maximum value that we've seen is -0.7 and we add it to the output. And this way using different filters of different size we have 300 outputs. This vector is a kind of embedding of the input sequence - we transformed the input sequence into a vector of fixed size. Next we can apply some more dense layers and we actually apply Multi-Layer Perceptron on top of those 300 features and train it for any task - classification, regression, ...

Training Several Filters

Quality comparison on customer reviews (CR)

- Naïve Bayes on top of 1,2-grams – 86.3
- 1D convolutions with MLP – 89.6

Comparing the quality of this model with classical BOW approach. The link above refers to these experiments. A customer reviews dataset was used to compare their model with Naive Bayes on top of 1,2-grams. And this classical model gave 86.3 accuracy. Applying 1D convolutions architecture with MLP on top of those features, the accuracy increase 3.8. Besides, applying convolutions is faster than BOW.

http://www.jmlr.org/papers/volume12/collobert11a/collobert11a.pdf

https://arxiv.org/pdf/1408.5882.pdf

Summary

- We can just average pre-trained word2vec vectors for our text
  - Split the text into tokens
  - For each token we take an embedding vector and then just sum them
  - This is a baseline model
- Another approach consist in using 1D convolutions that learn more complex features
  - This way we train neural network end to end - we have an input sequence and a result that we want to predict - use back propagation and train all those convolutions, to train the specific features that the neural network needs to classify the sentence.
- Next we'll continue to apply convolutions to text

Further Text Processing

What is Text

We can think of text as a sequence of:

- **Characters**
- Words
- Phrases and named entities
- Sentences
- Paragraphs
- ...

Text as a Sequence of Characters

One-hot encoded characters, length ~ 70

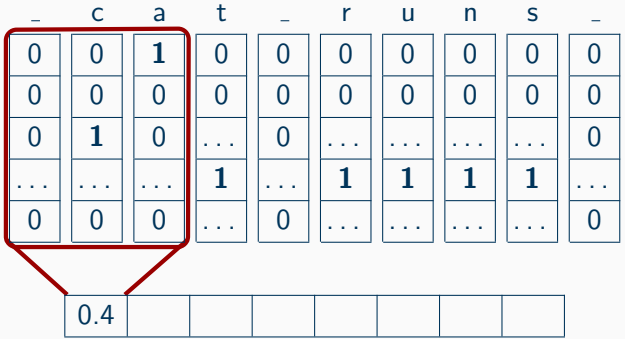
_	c	a	t	_	r	u	n	s	_
0	0	**1**	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	**1**	0	...	0	...	...	...	...	0
...	...	...	**1**	...	**1**	**1**	**1**	**1**	...
0	0	0	...	0	...	...	...	...	0

Let's start with character n-grams!

How can we treat a text as a sequence of characters? Lets take an example phrase "cat runs". Each character is embedded in an one hot-encoded vector of length 70. Let's start with character n-grams just like we did with words.

1D Convolutions on Characters

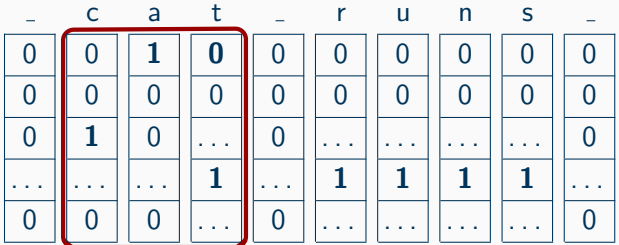
One-hot encoded characters, length ~ 70



We apply a convolution filter through the character sequence. Then, we can apply several filters to the same sequence, 1024 filters, for instance.

1D Convolutions on Characters

One-hot encoded characters, length ~ 70



1D Convolutions on Characters

One-hot encoded characters, length ~ 70

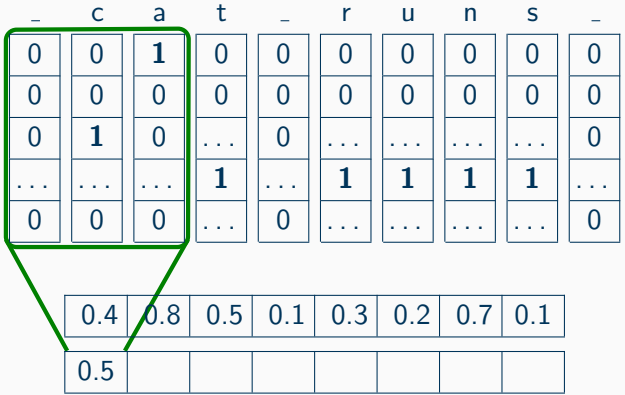
_	c	a	t	_	r	u	n	s	_
0	0	**1**	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	**1**	0	...	0	...	...	...	...	0
...	...	...	**1**	...	**1**	**1**	**1**	**1**	...
0	0	0	...	0	...	...	...	...	0

Filter #1

0.4	0.8	0.5	0.1	0.3	0.2	0.7	0.1

1D Convolutions on Characters

One-hot encoded characters, length ~ 70



1D Convolutions on Characters

One-hot encoded characters, length ~ 70

_	c	a	t	_	r	u	n	s	_
0	0	**1**	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	**1**	0	...	0	...	...	...	...	0
...	...	...	**1**	...	**1**	**1**	**1**	**1**	...
0	0	0	...	0	...	...	...	...	0

Filter #1	0.4	0.8	0.5	0.1	0.3	0.2	0.7	0.1
Filter #2	0.5	0.1	0.4	0.2	0.3	0.9	0.1	0.8

1D Convolutions on Characters

One-hot encoded characters, length ~ 70

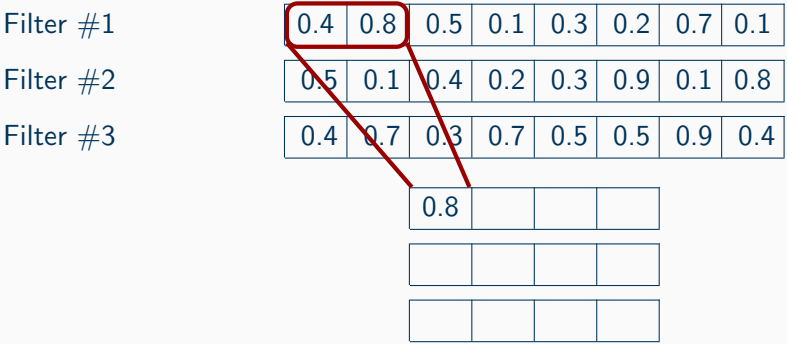
_	c	a	t	_	r	u	n	s	_
0	0	**1**	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	**1**	0	...	0	...	...	...	...	0
...	...	...	**1**	...	**1**	**1**	**1**	**1**	...
0	0	0	...	0	...	...	...	...	0

Filter #1	0.4	0.8	0.5	0.1	0.3	0.2	0.7	0.1
Filter #2	0.5	0.1	0.4	0.2	0.3	0.9	0.1	0.8
Filter #3	0.4	0.7	0.3	0.7	0.5	0.5	0.9	0.4

~ 1024 filters

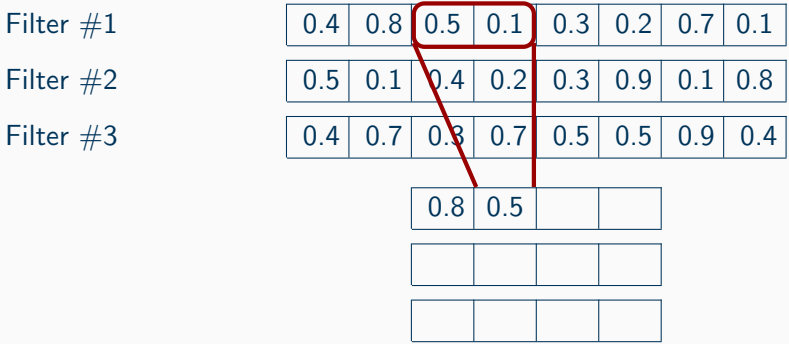
What's next? Let's add pooling!

Max Pooling

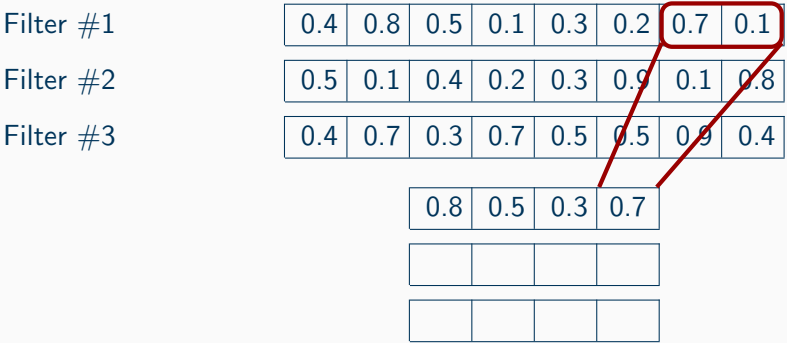


We usually add convolution followed by pooling, then again convolution and pooling and so forth. So let's add pooling. Let's see how pooling is applied. We take neighboring values, and we takes the maximum of them. This time, it's 0.8. Then we move that window with a stride of two, and we take the maximum of those values as well. And we do it all to the end, and we do that for all the filters we have, and this is our pooling output.

Max Pooling


Filter #1
Filter #2
Filter #3
0.4 0.8 0.5 0.1 0.3 0.2 0.7 0.1
0.5 0.1 0.4 0.2 0.3 0.9 0.1 0.8
0.4 0.7 0.3 0.7 0.5 0.5 0.9 0.4
0.8 0.5

Max Pooling


Filter #1
Filter #2
Filter #3
0.4 0.8 0.5 0.1 0.3 0.2 0.7 0.1
0.5 0.1 0.4 0.2 0.3 0.9 0.1 0.8
0.4 0.7 0.3 0.7 0.5 0.5 0.9 0.4
0.8 0.5 0.3 0.7

Max Pooling

Filter #1	0.4	0.8	0.5	0.1	0.3	0.2	0.7	0.1
Filter #2	0.5	0.1	0.4	0.2	0.3	0.9	0.1	0.8
Filter #3	0.4	0.7	0.3	0.7	0.5	0.5	0.9	0.4

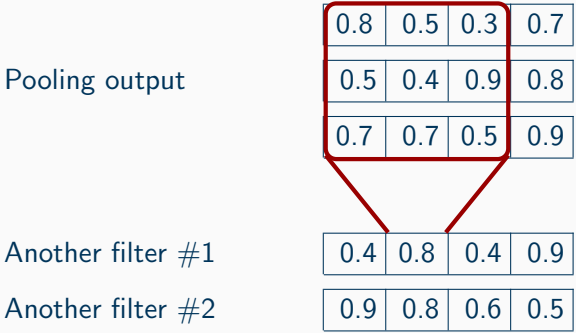
Pooling output

0.8	0.5	0.3	0.7
0.5	0.4	0.9	0.8
0.7	0.7	0.5	0.9

Pooling introduces position invariance to character n-grams. If a character n-gram slides one character to the left or to the right, there is a chance that thanks to pooling, the activation that we will have in that pooling output will stay the same.

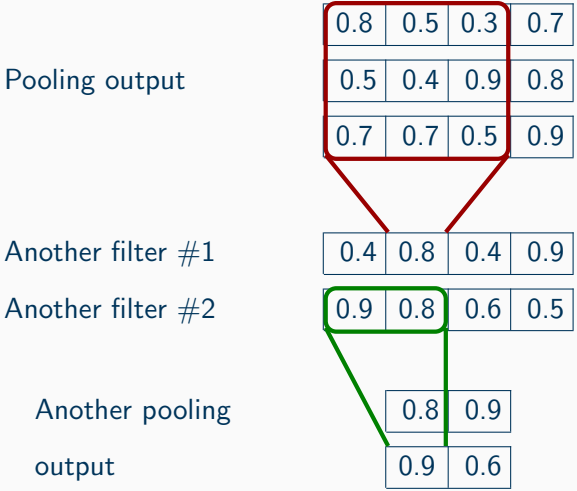
Provides a little bit of position invariance for character n-grams

Repeat 1D Convolution + Pooling



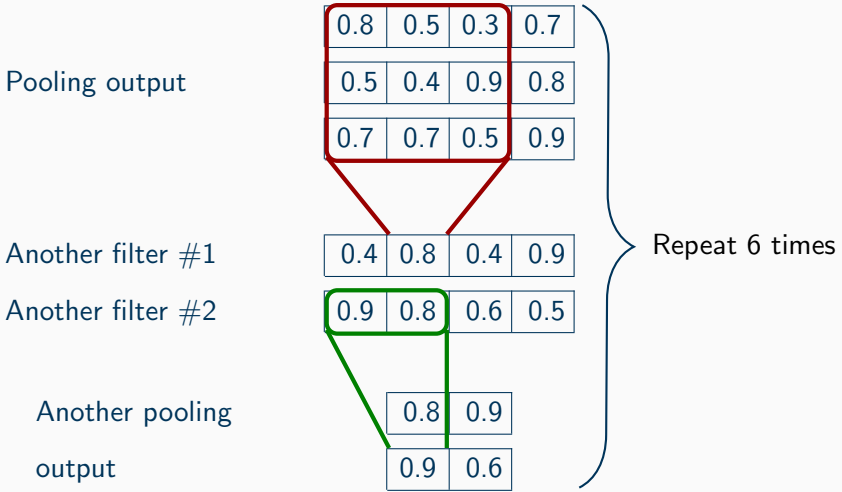
Repeat 1D Convolution + Pooling

We continue to apply convolutions followed by pooling and so forth. So let's take that previous pooling output, and let's apply 1D convolutions as well.



Repeat 1D Convolution + Pooling

Notice that the length of the feature representation decreases. That means that our receptive field actually increases and we look at more and more characters in our input when we make decision about activation on a deep level. We can repeat this procedure six times.



Final Architecture

- Let's take only first **1014** characters of text
- Apply 1D convolution + max pooling **6** times
  - Kernels widths: 7, 7, 3, 3, 3, 3
  - Filters at each step: 1024
- Apply MLP for your task

https://arxiv.org/pdf/1509.01626.pdf

Experimental Datasets

Categorization or sentiment analysis

	Dataset	Classes	Train Samples
Smaller	AG's News	4	120,000
	Sogou News	5	450,000
	DBPedia	14	560,000
	Yelp Review Polarity	2	560,000
Bigger	Yelp Review Full	5	650,000
	Yahoo! Answers	10	1,400,000
	Amazon Review Full	5	3,000,000
	Amazon Review Polarity	2	3,600,000

All these datasets are either a categorization, like news datasets, or a sentiment analysis like reviews. The red datasets are smaller, containing at most 600,000 training samples. The green ones are bigger datasets that contains millions of samples.

https://arxiv.org/pdf/1509.01626.pdf

Experimental Datasets

In the first table we have classical models with linear model on top of that. For small datasets (in red border), the error is the least when we use n-grams with TFIDF, most of the time. So for small training set it makes sense to use classical approaches.

For huge datasets, convolutional architecture beats LSTM.

Errors on test set for classical models:

Model	AG	Sogou	DBP.	Yelp P.	Yelp F.	Yah. A.	Amz. F.	Amz. P.
BoW	11.19	7.15	3.39	7.76	42.01	31.11	45.36	9.60
BoW TFIDF	10.36	6.55	2.63	6.34	40.14	28.96	44.74	9.00
ngrams	7.96	2.92	1.37	**4.36**	43.74	31.53	45.73	7.98
ngrams TFIDF	**7.64**	**2.81**	**1.31**	4.56	45.20	31.49	47.56	8.46

Errors on test set for deep models:

LSTM	13.94	4.82	1.45	5.26	41.83	29.16	40.57	6.10
Sm. Full Conv.	11.59	8.95	1.89	5.67	38.82	30.01	40.88	5.78
Lg. Full Conv. Th.	9.51	-	1.55	4.88	38.04	29.58	40.54	5.51
Sm. Full Conv. Th.	10.89	-	1.69	5.42	**37.95**	29.90	40.53	5.66
Lg. Conv.	12.82	4.88	1.73	5.89	39.62	29.55	41.31	5.51
Sm. Conv.	15.65	8.65	1.98	6.53	40.84	29.84	40.53	5.50
Lg. Conv. Th.	13.39	-	1.60	5.82	39.30	**28.80**	40.45	**4.93**
Sm. Conv. Th.	14.80	-	1.85	6.49	40.16	29.84	**40.43**	5.67

Deep models work better for large datasets!

https://arxiv.org/pdf/1509.01626.pdf

Summary

- We can use convolutional networks on top of characters (called learning from scratch)
- It works best for large datasets where it beats classical approaches (like BOW)
- Sometimes it even beats LSTM that works on word level
- We saw some deeper representations at character level, where we don't tell the system or the model where the words are and it works better