Recurrent Neural Networks

Lesson 3

Tiago Cabo

Lesson Plan

- What are RNNs?
- Why RNNs are such big thing now?
- How to work with text
- Preprocessing
- Word-to-vec
- RNN architectures
- Apply trained CNNs

Lesson goals

- Be able to explain what RNN are
- Be able to understand main building blocks and apply a simple RNNs

RNN Applications

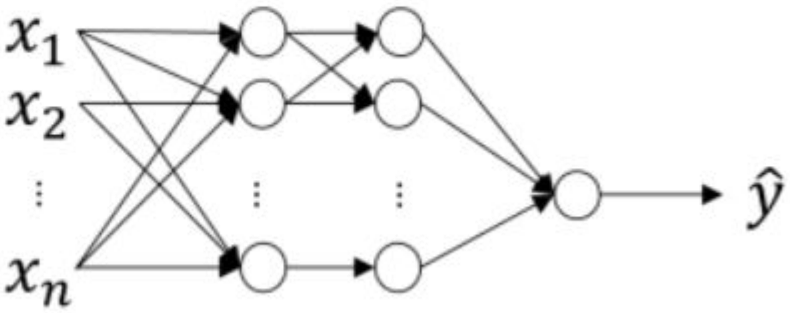
- Music generations
- Sentiment classification
- DNA sequence analysis
- Machine translation
- Video activity recognition
- Named entity recognition

Why not DNN?

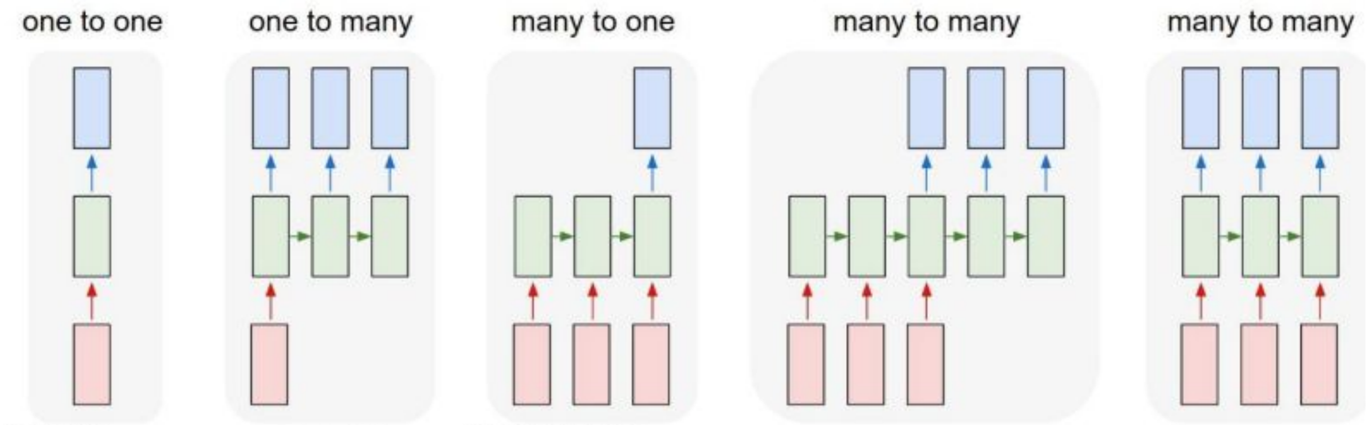
DNN simply cannot capture sequence by nature, unless some kind of feature engineering. This is due to:

- Fixed-sized inputs and outputs
- No temporal structure

They have a pure feed-forward processing, there are such memory or feedback process.



Configurations Scenarios



Input: No sequence
Output: No sequence
Example: standard classification / regression problems

Input: No sequence
Output: Sequence
Example: Image to caption, music generation

Input: Sequence
Output: No sequence
Example: sentence classification (sentiment classification), multiple-choice question answering

Input: Sequence
Output: Sequence
Example: machine translation, video classification, video captioning, open-ended question answering, named-entity recognition

What are RNNs?

The main idea behind RNN is to take the previous output or state as new input in the network. THe input has some historical information about the past. This time can be set. So it is possible to look to more or less past data. Or give different weights to past or more recent data.

This intermediate states are not predefined in the beginning so they are updated while learning

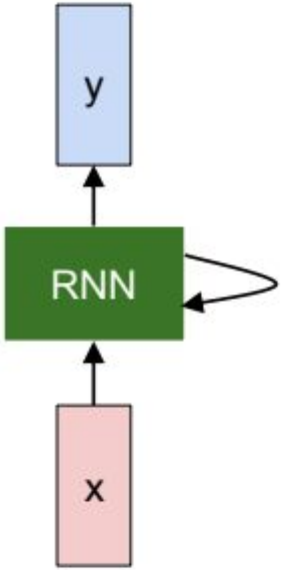
$$\boxed{h_t} = \boxed{f_W}(\boxed{h_{t-1}}, \boxed{x_t})$$

new state

old state

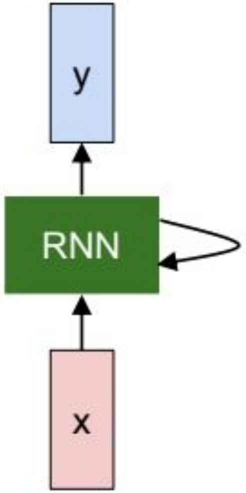
input vector at some time step

some function with parameters W



RNN

The state consists of a single “hidden” vector **h**:



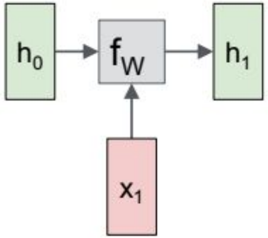
$$y_t = W_{hy}h_t + b_y$$

$$h_t = f_W(h_{t-1}, x_t)$$

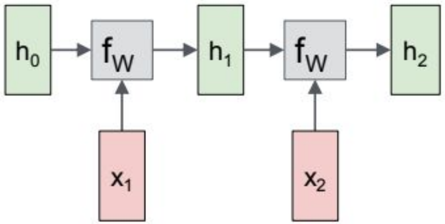
$$\downarrow$$

$$h_t = \tanh(W_{hh}h_{t-1} + W_{xh}x_t + b_h)$$

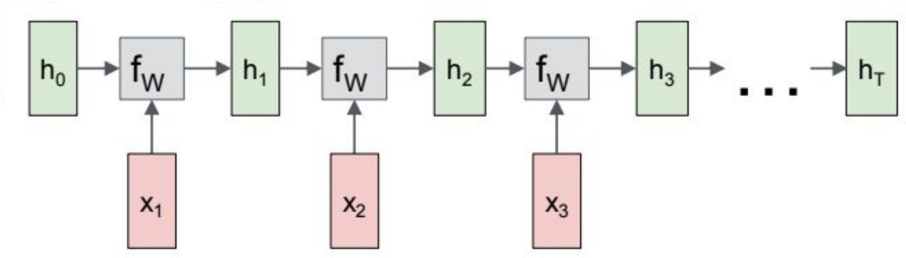
RNN Flow



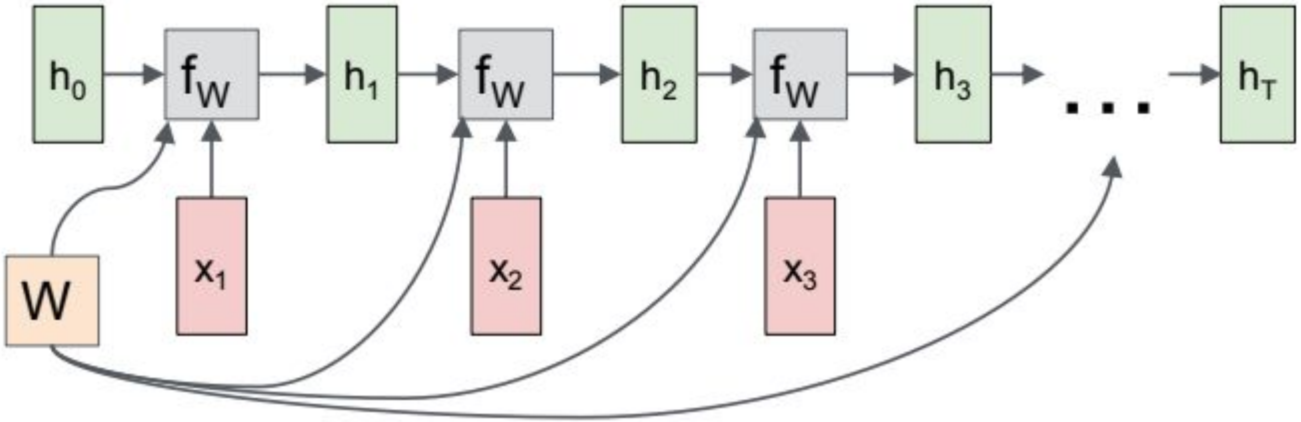
RNN Flow



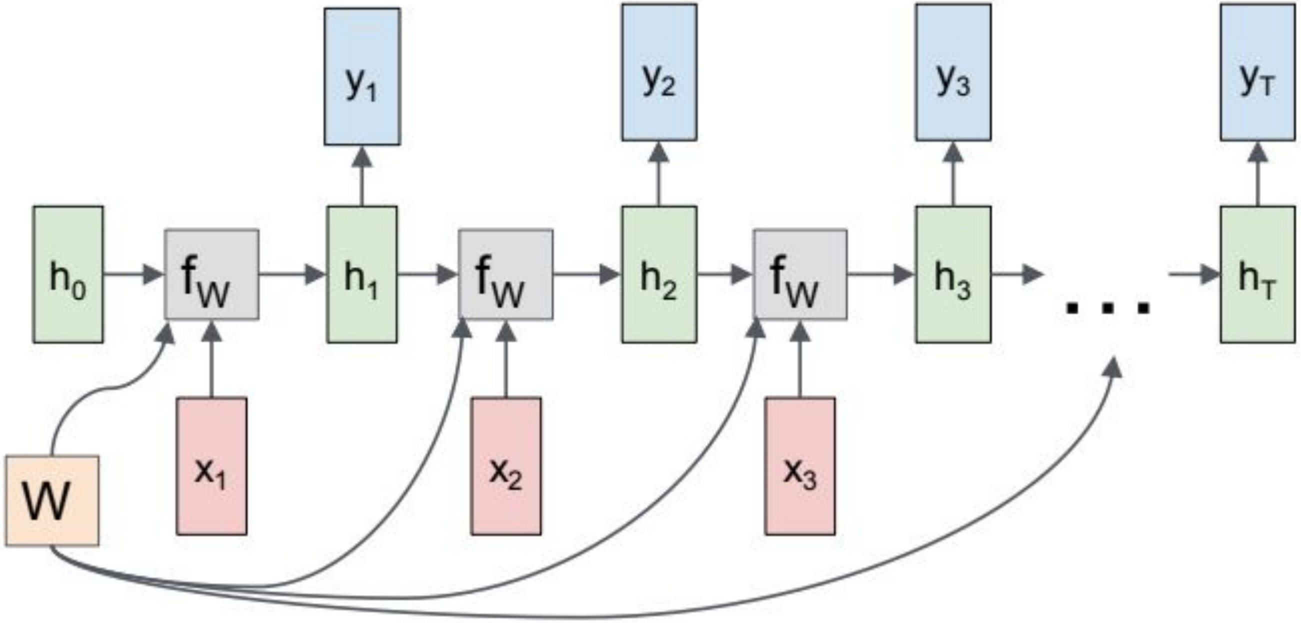
RNN Flow



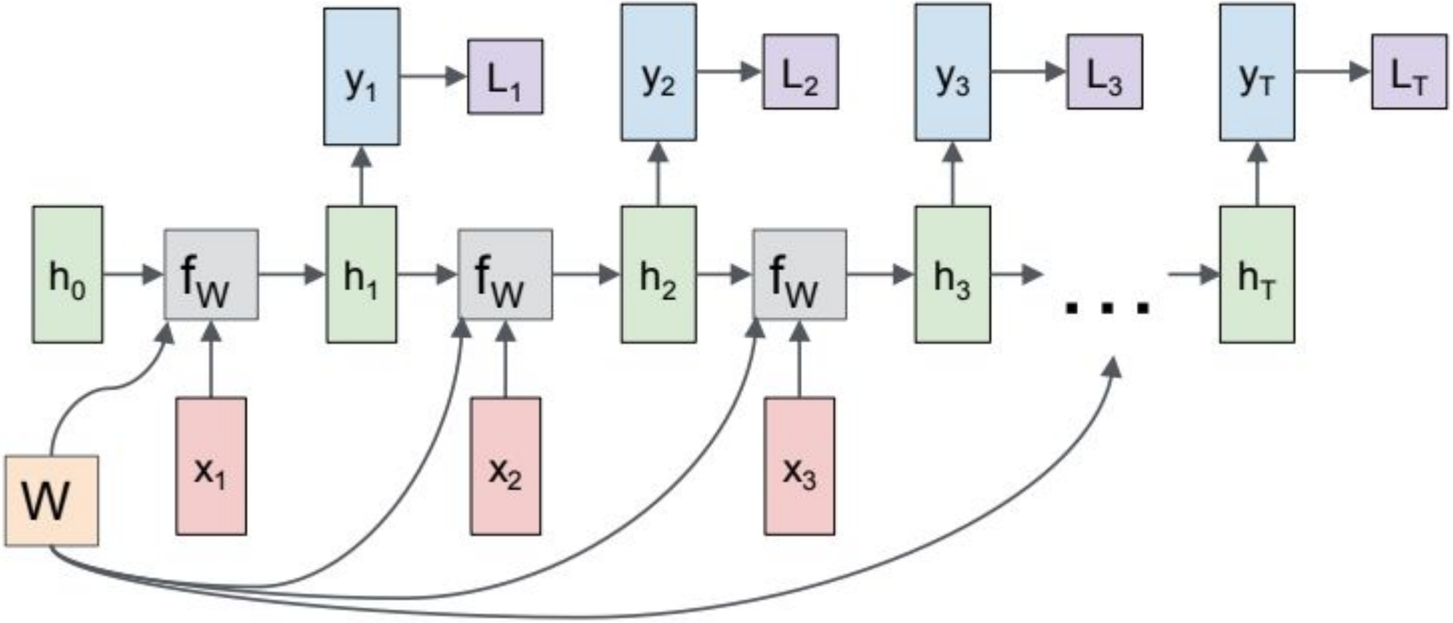
RNN Flow



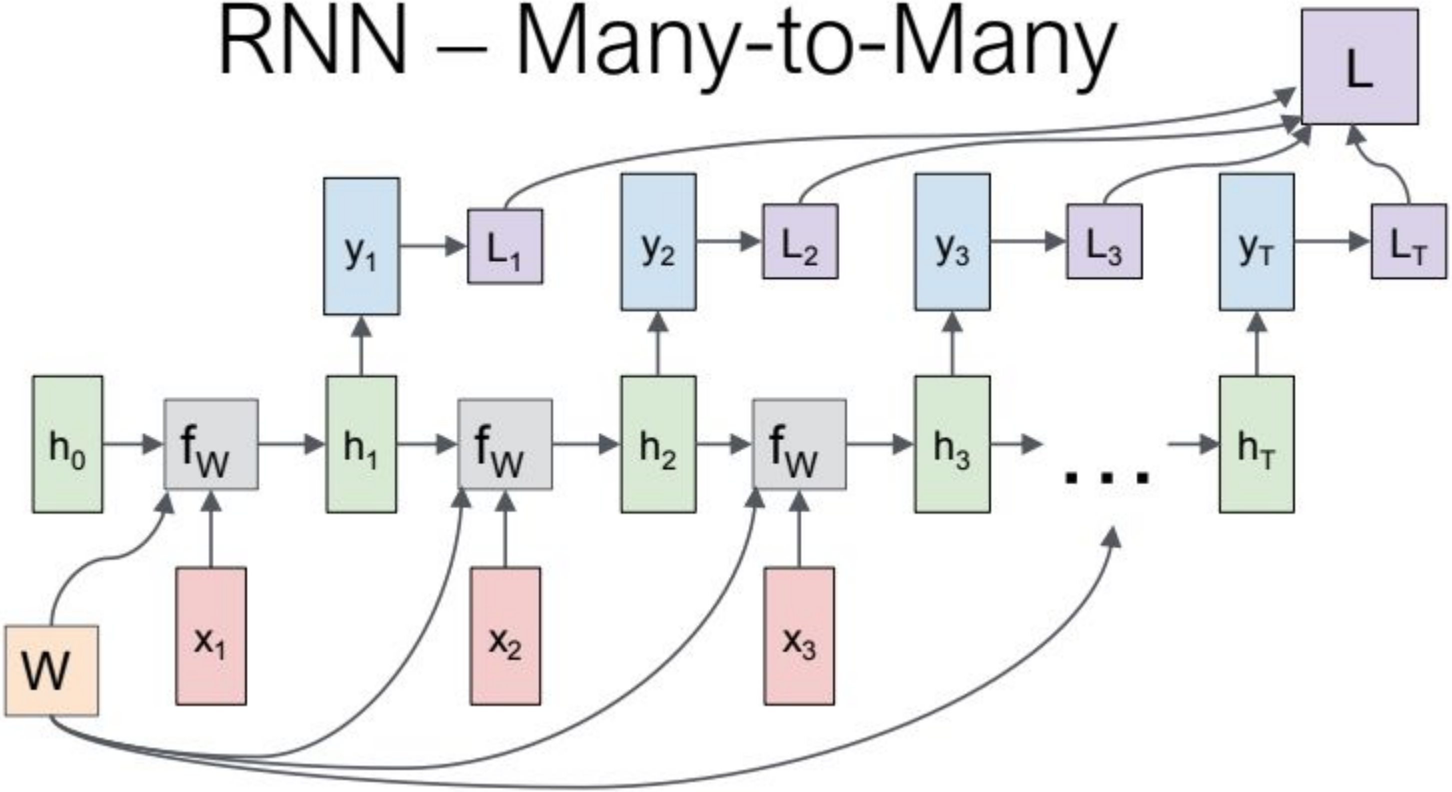
RNN - Many-to-many



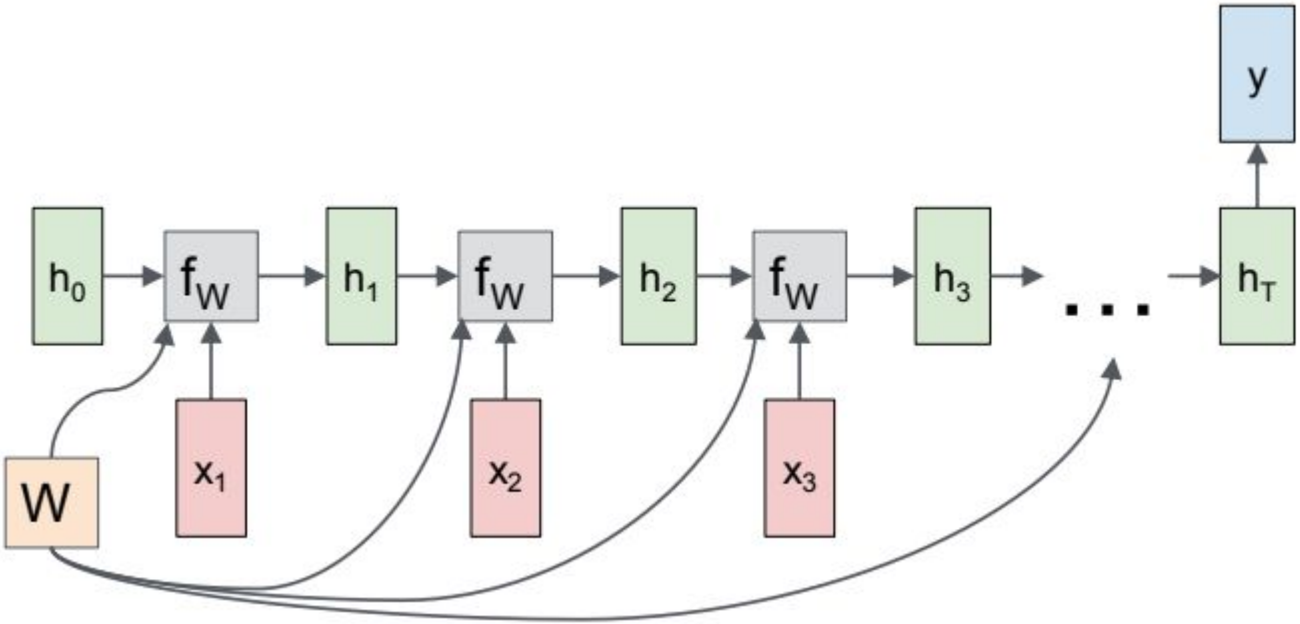
RNN - Many-to-many



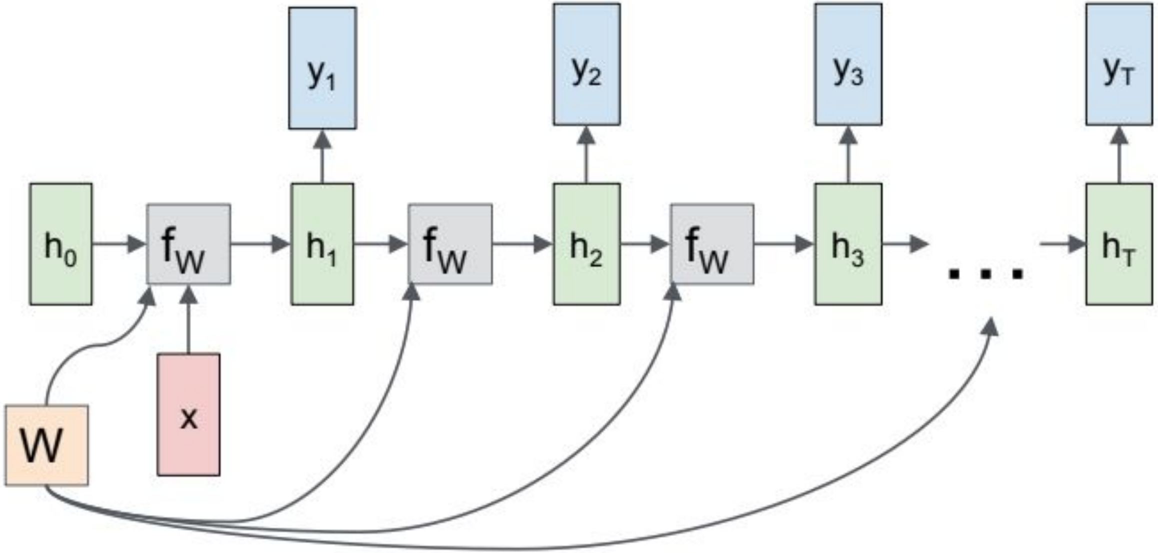
RNN – Many-to-Many



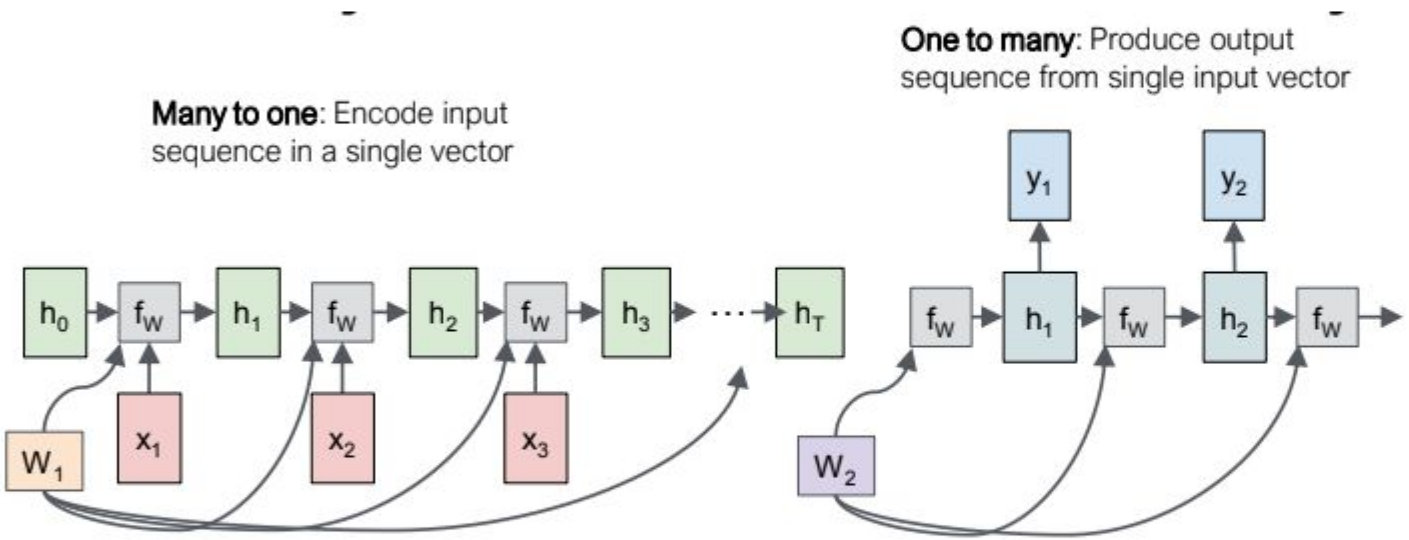
RNN - Many-to-One



RNN - One-to-many

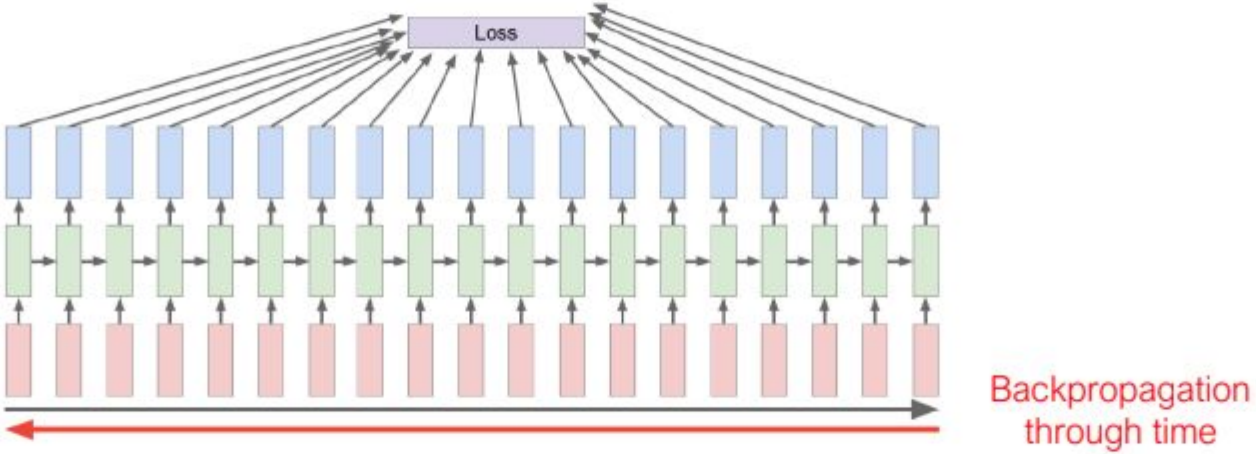


Sequence to Sequence



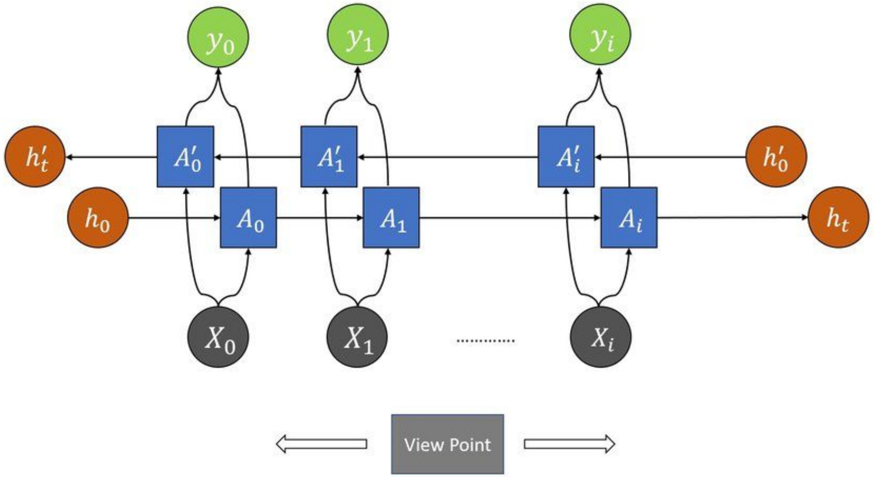
Forward and Backward propagations

Forward through entire sequence to compute loss, then backward through entire sequence to compute gradient



Bidirectional RNN

- Used in cases where the prediction not only depends on the past past alse on the future. For example is machine translation examples.
- Performance improvement
- However we need all the sequence to make predictions. For example in real-time speech applications this is not interesting
- Bigger computation costs



Problems with standard RNN

As we saw, the weights are updated through time.This means that the gradient is propagated through time. This has the same implication of very deep networks, if nothing done we may end up with issues with vanishing gradient.

This means that for example in RNN, the information further back in time do not mean a lot for the prediction. This may be an issue in use cases like text generation. Because you may need to search for the subject in the beginning of the text.

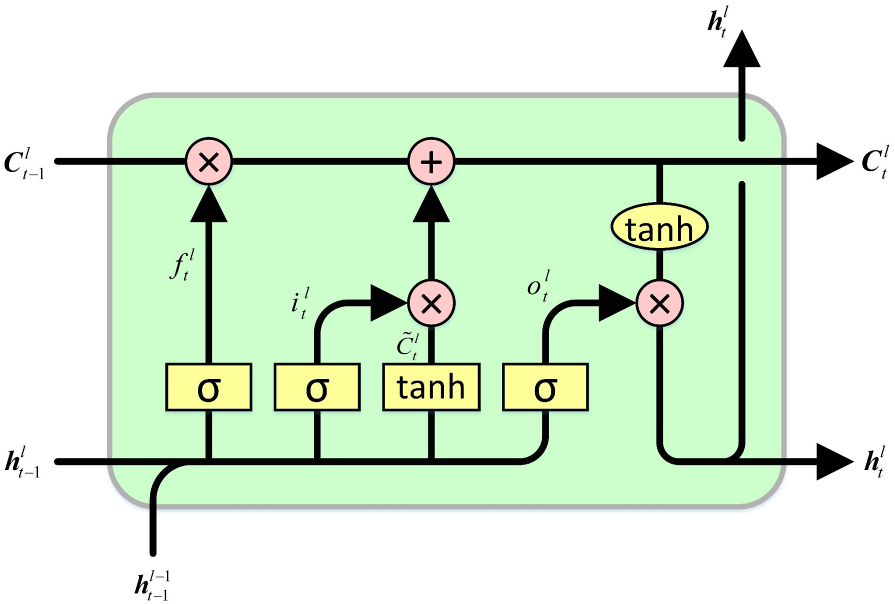
LSTM (Long short-term memory)

LSTM networks were developed in 1997, and add additional gates in each memory cell that consist in:

- Forget state
- Input state
- Output state

WIth this we prevent the vanishing/exploding gradient problems. Also enables the network to retain state information for longer.

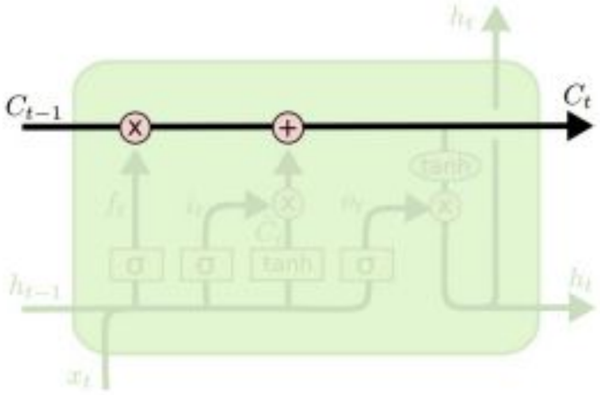
Interesting paper here: https://arxiv.org/pdf/1909.09586.pdf



LSTM cell state / Memory

A vector Ct, is maintained with the same dimensionality as the hidden state, ht.

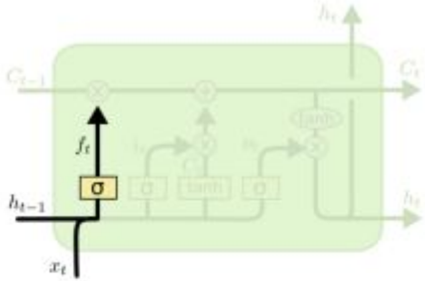
Additional information can be added or deleted from this vector via the forget and input gates.



Forget Gate

Forget gates computes a 0 or 1 value using a sigmoid output function given the input, xt and the previous state, ht-1

Instead of the sigmoid, tanh activation function can be used instead.

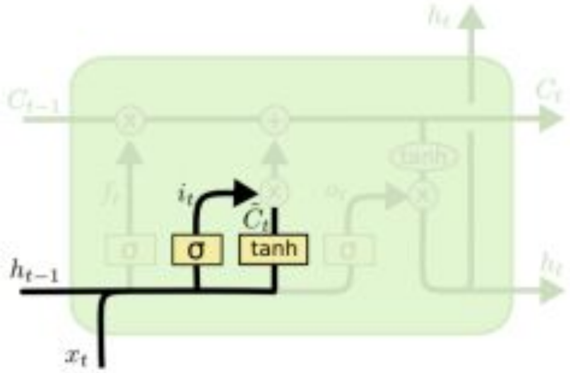


$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$$

Input Gate

In this gate two operations are performed:

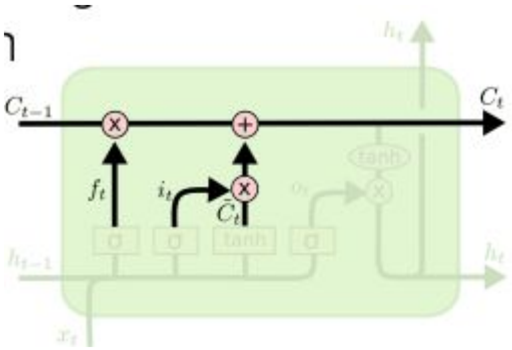
- First, we need to determine which entries in the cell state to update by computing a sigmoid output.
- Tens, we need to determine what amount to be subtracted from the entries by computing a tanh activation output function given the previous state and the input.



$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \ + \ b_i\right)$$
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \ + \ b_C)$$

Memory update

Cell state is computed by adding the forget and the new information.

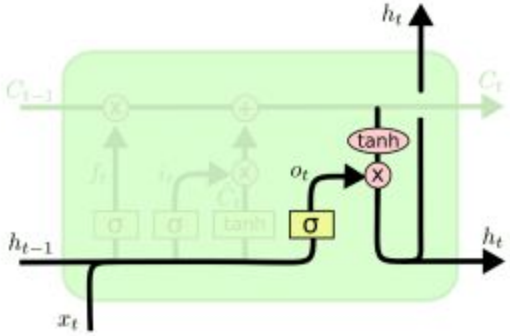


$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

Output Gate

This gate is responsible for computing the overall state to be transmitted to the next cell.

This involves a sigmoid function that computes ot. However the final ht is calculated given the Ct and ot using a tanh activation function

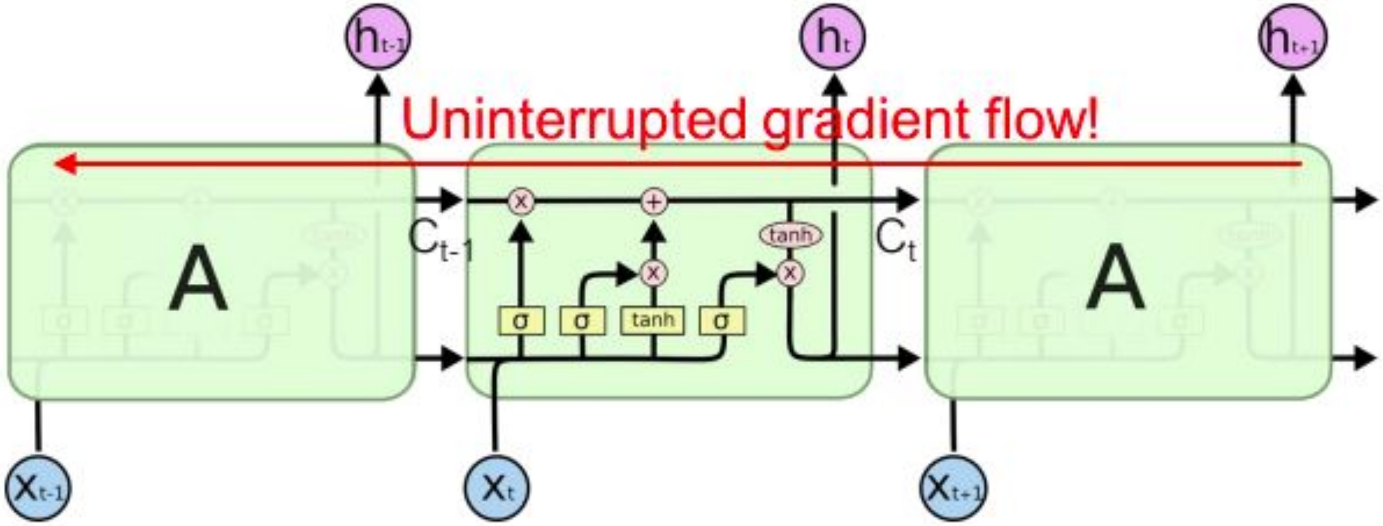


$$o_t = \sigma\left(W_o \left[h_{t-1}, x_t\right] + b_o\right)$$

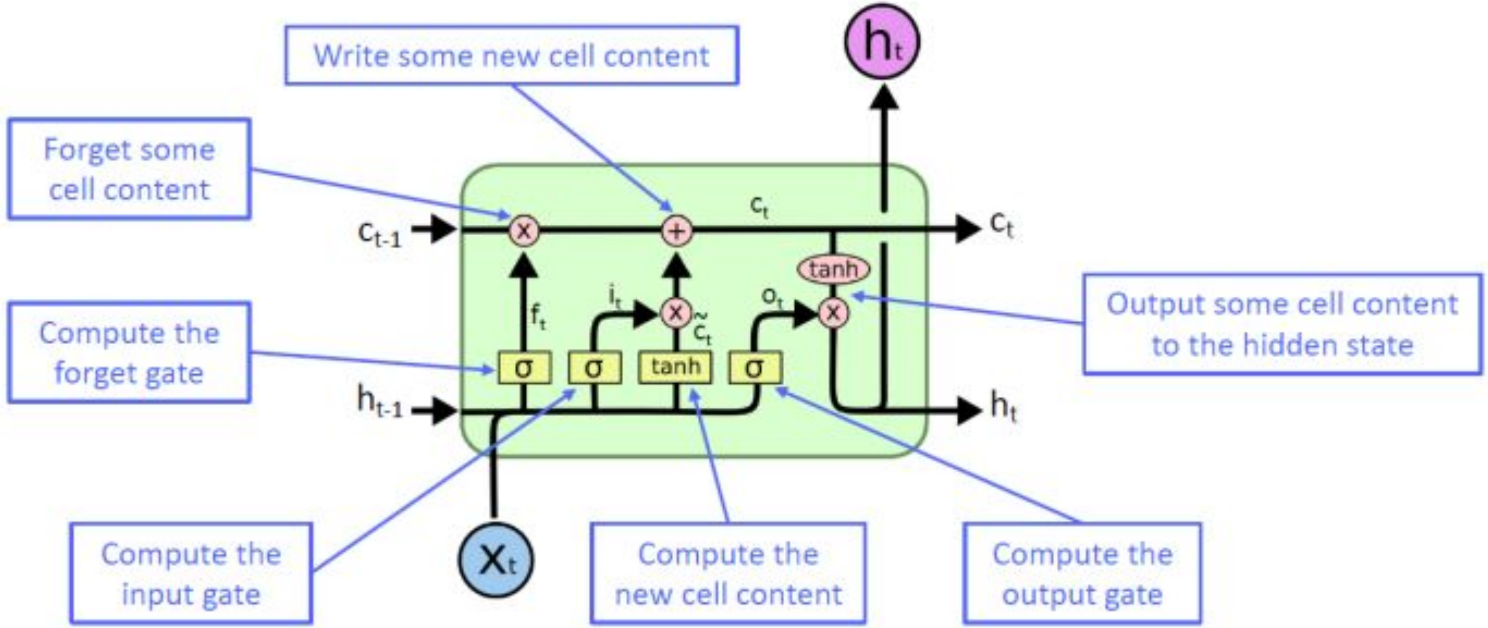
$$h_t = o_t * \tanh\left(C_t\right)$$

Addictive layers

Case multiple inputs the cells are concatenated.



Summary


Write some new cell content
Forget some cell content
Compute the forget gate
Compute the input gate
Compute the new cell content
Compute the output gate
Output some cell content to the hidden state
h_t
c_{t-1}
c_t
h_{t-1}
f_t
i_t
$\tilde{c}_t$
o_t
σ
tanh
x_t

LSTM Training

LSTM training is very similar to the other DNN. It works using backpropagation derivatives. The algorithms used are:

- Gradient Descent
- SGD
- GD with momentum
- ADAM

Usually for LSTM to work well a lot of data is required. Also, due to the complex and multiple operations that are required take a while to train. This can be improved with the GPU usage.

Gated Recurrent Unit (GRU)

GRUs are an alternative to LSTM that use fewer gates and therefore have performance improvements.

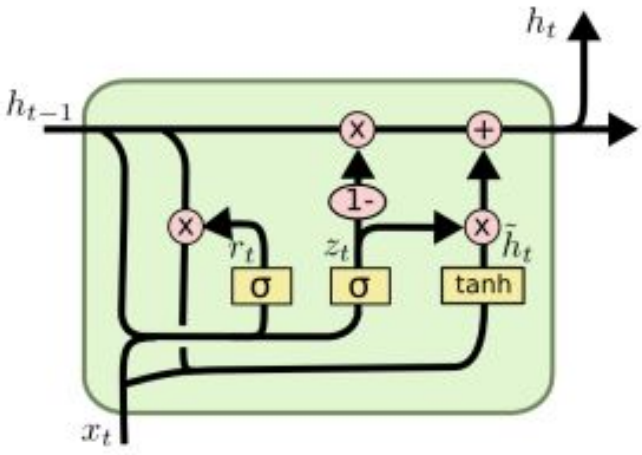
The cell state vector is removed.

$$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t]\right)$$

$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$

$$\tilde{h}_t = \tanh\left(W \cdot [r_t * h_{t-1}, x_t]\right)$$

$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$



GRU vs LSTM

Because the cell state is removed and the forget and input gate are combined the network has fewer parameters which makes it a lot easier to train.

In terms of raw performance, bot perform similarly on a wide range of problems. However, depending on the problem one may be preferred to the other.

As in most ML problems several algorithms need to be tested in order to evaluate the best.

Other algorithms

Limitations of RNN based algorithms:

- Sequential computation inhibits parallelization
- No explicit modeling of long and short-range dependencies
- Distance between positions is linear

Attention networks for the rescue:

- Allow networks to focus on particular parts of the input, at different time stamps, during processing

Transformers

- Use **attention + CNN**
- Solves **parallelization problem**
- **Dramatically reduces** training time

Text processing

Text processing

As you can see from the example on the right we can observe that text can be a bit confusing because:

- Expressions like ooooh
- ….. Special signals
- Typos
- Etc

So normally the first steps, as in any ML project is to preprocess the data.

```
ooooh.... LOL that
leslie.... and ok I
won't do it again so
leslie won't get mad
again
```

```
@cocomix04 ill tell
ya the story later
not a good day and
ill be workin for
like three more
hours.....
```

Let’s load data

1. Please load into a jupyter notebook the following dataset from drive “twitter_sentiment.csv.zip”

Typical preprocessing steps - Tokenizing

This is about transforming a sentence in individual tokens or words.

```
from nltk.tokenize import word_tokenize

sentence = "Runners have planned for 20km run. Previously, they ran a 15km run up." #sentence to be stemmed

words = word_tokenize(sentence) #tokenizing the words of a sentence

#printing the results of stemming the words of a sentence
for x in words:
   print(x, " : ", ps.stem(x))
```

Typical preprocessing steps - remove stop words

This is a process that has the goal to remove stop words in text. Words like “the”, “a”, “etc”. In most problems this is applicable.

However, please take attention, because in some unique cases these words may be important

```
import nltk
nltk.download('stopwords')
```

```
from nltk.corpus import stopwords
stopwords.words('english')
print stopwords.words() [620:680]
```

```
from nltk.corpus import stopwords
print stopwords.fileids()
```

```
from nltk.corpus import stopwords
en_stops = set(stopwords.words('english'))
```


Tiago Cabo

Typical preprocessing steps - Stemming

Stemming is about replacing words by it base. This means for example replace words like:

- Going -> go
- Walking -> walk

Python implementation

```
from nltk.stem import PorterStemmer

ps = PorterStemmer() #creating an instance of the class

ps.stem(x)
```

Typical preprocessing steps - Lemmatization

This is very similar to stemming, but in this case it fixes the word to words in the dictionary and adding more natural words. This methods es more recommended in production

This means for example replace words like:

- Natur -> naturally

```
from nltk.stem import WordNetLemmatizer

lemmatizer = WordNetLemmatizer()

def lemmatize_words(text):

    return " ".join([lemmatizer.lemmatize(word) for word in text.split()])

df["text"] = df["text"].apply(lambda text: lemmatize_words(text))
```

Typical preprocessing steps - Other techniques

Other techniques may involve:

- Removing extra spaces
- Removing numbers of special characters not relevant to the problem
- Remove punctuation
- Lowercase if not relevant to the problem

Exercise

- Remove stop words from the twitter example
- Apply stemming in the tweets

Now we have the clean words, then what?
Can I pass words directly to the algorithms?

NO, we need to transform them into numbers

https://towardsdatascience.com/introduction-to-word-embeddings-4cf857b12edc

Word Representation - one hot encoding

There are several techniques. One of the simplest is the one-hot encoder that we already used in the categorical variable example in M8.

One-hot encoder with a lot of token as several problems, being:

- Curse of dimensionality
- Very sparse matrix
- No relation between words. For example, queen and king are related

	1	2	3	4	5	6	7	8	9
man	1	0	0	0	0	0	0	0	0
woman	0	1	0	0	0	0	0	0	0
boy	0	0	1	0	0	0	0	0	0
girl	0	0	0	1	0	0	0	0	0
prince	0	0	0	0	1	0	0	0	0
princess	0	0	0	0	0	1	0	0	0
queen	0	0	0	0	0	0	1	0	0
king	0	0	0	0	0	0	0	1	0
monarch	0	0	0	0	0	0	0	0	1

tf-idf

TF-IDF stands for Term Frequency — Inverse Document Frequency and is a statistic that aims to better define how important a word is for a document, while also taking into account the relation to other documents from the same corpus.

This is performed by looking at how many times a word appears into a document while also paying attention to how many times the same word appears in other documents in the corpus.

$$w_{i,j} = tf_{i,j} \times \log\left(\frac{N}{df_i}\right)$$

$tf_{i,j}$ = number of occurrences of i in j
df_i = number of documents containing i
N = total number of documents

$$TF(i,j) = \frac{\text{Term i frequency in document j}}{\text{Total words in document j}}$$

$$IDF(i) = \log_2\left(\frac{\text{Total documents}}{\text{documents with term i}}\right)$$

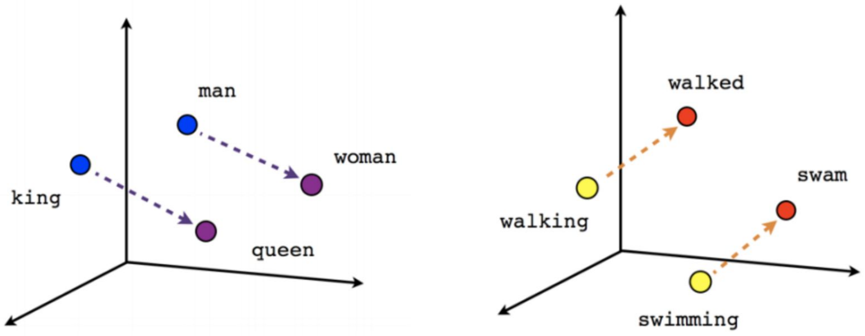
Word Representation - embeddings

Using embeddings we are able to have relations between words represented in the embeddings. The idea is to be able to find related words by searching similarity between vectors in the word representation space.

This enables things like:

King - Man + Women = Queen

Paris - France + Italy = Rome



Male-Female

Verb tense

Word-to-vec

Word-to-vec is an approach that uses shallow neural network algorithms that is able to understand the relationship between words and represent them in a n dimensional vector.

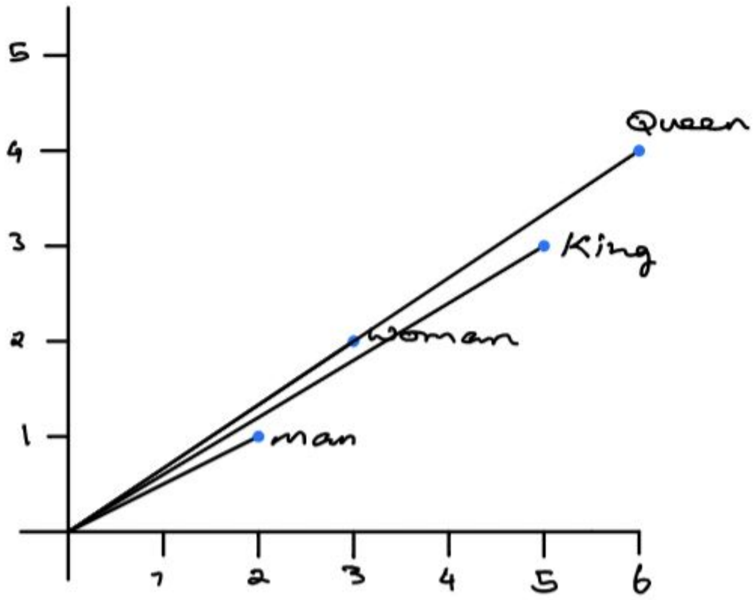
Those techniques are:

- CBOW (Continuous bag of words)

In this the model is trained using the surrounding context words

- Skip-grams

In this approach the model is trained using a word and trying to predict the context words.



https://arxiv.org/pdf/1301.3781.pdf

Word-to-vec example

```
from gensim.models import Word2Vec

# Create CBOW model

model = gensim.models.Word2Vec(data, min_count = 1, vector_size = 100, window = 5)

# Print results

print("Cosine similarity between 'alice' " +  "and 'wonderland' - CBOW : ",model.wv.similarity('alice', 
'wonderland'))
```

Let’s apply in our use case

1. Train word-to-vec in a matrix shape
2. Build a LSTM network like
3. Compute some predictions

https://www.kaggle.com/code/paoloripamonti/twitter-sentiment-analysis

```
model = Sequential()
model.add(embedding_layer)
model.add(Dropout(0.5))
model.add(LSTM(100, dropout=0.2, recurrent_dropout=0.2))
model.add(Dense(1, activation='sigmoid'))

model.summary()
```