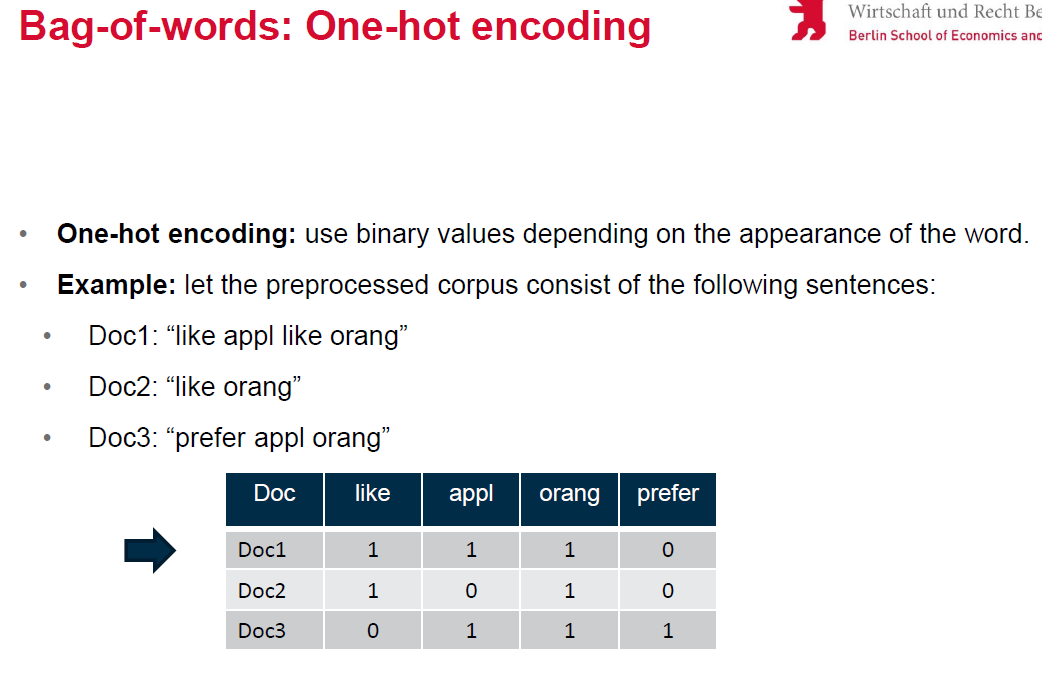
2nd Lesson

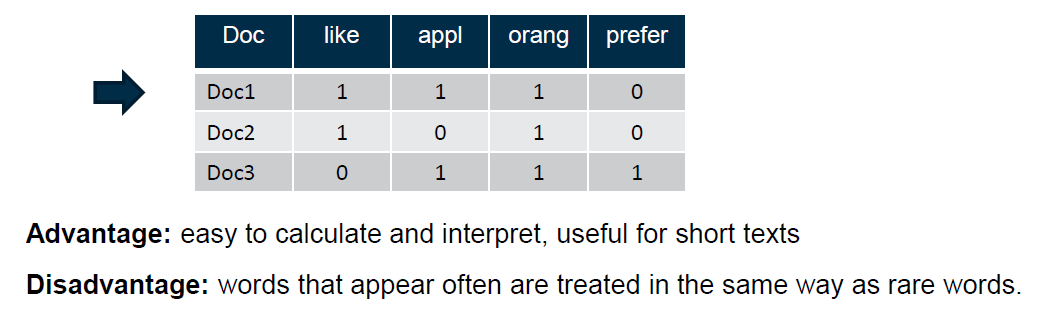
Weakness of Sentiment analysis: negations

* One hot encoding
* Absolut Term Frequencies
* Relative Term Frequencies
* Weighted Term Frequencies

TF 🡪 Document Frequencies

# Bag-of-words: One-hot encoding

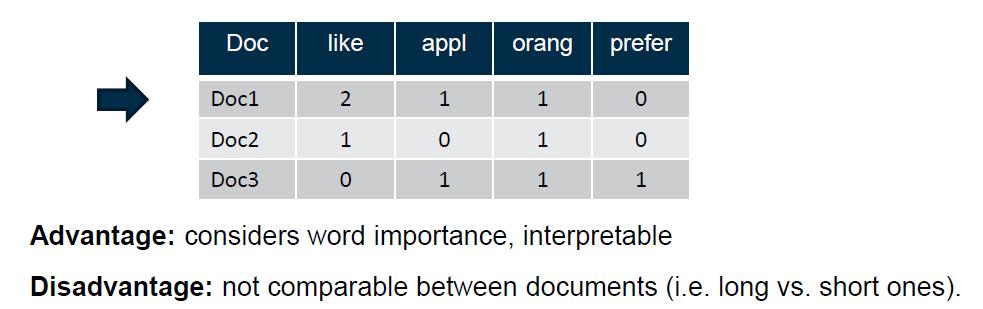




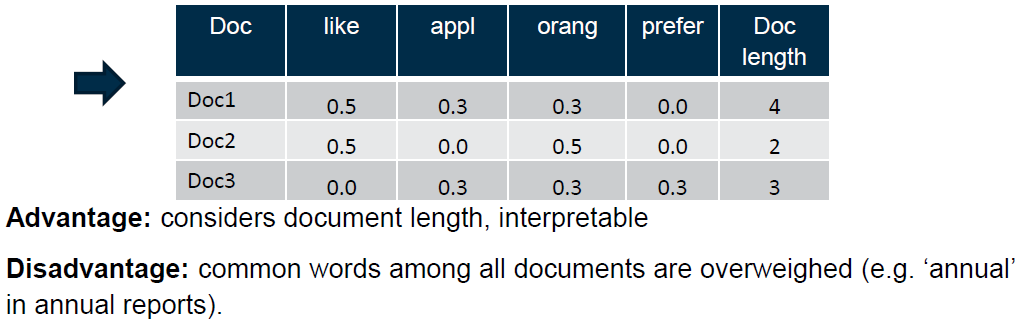
But one-hot-encoding has **disadvantages**:

1. Not possible to compare documents of different length
2. All words are considered equally important
3. Not useful for short texts

# Bag-of-words: Absolute frequency

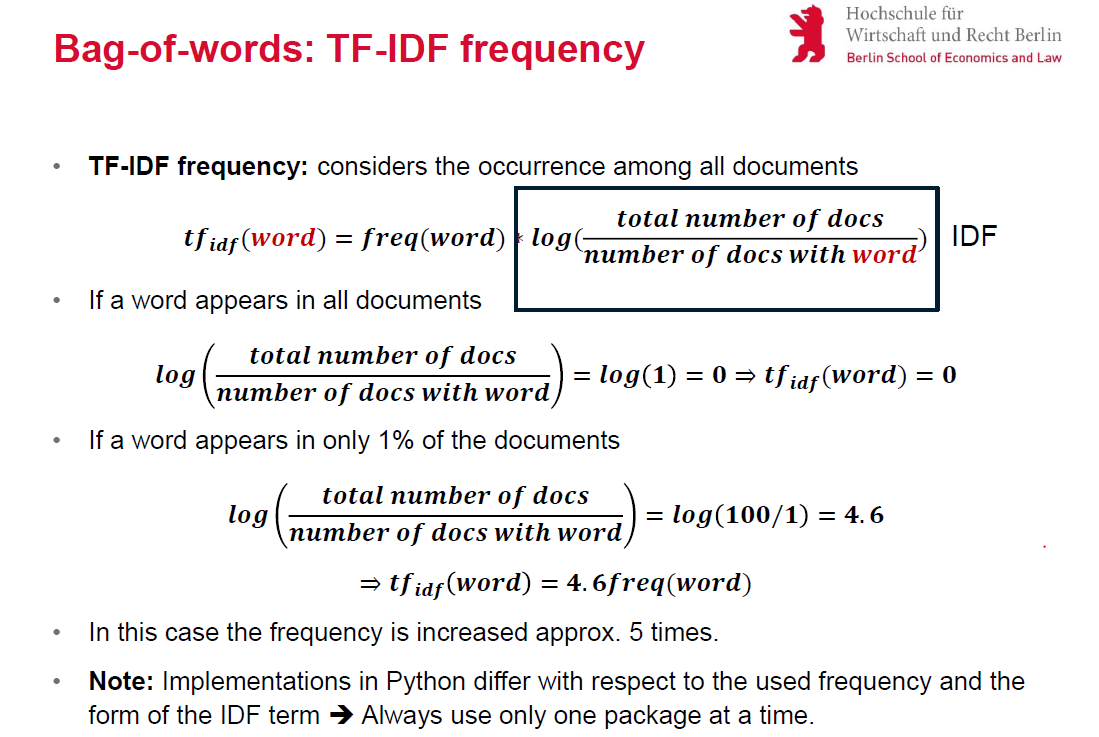


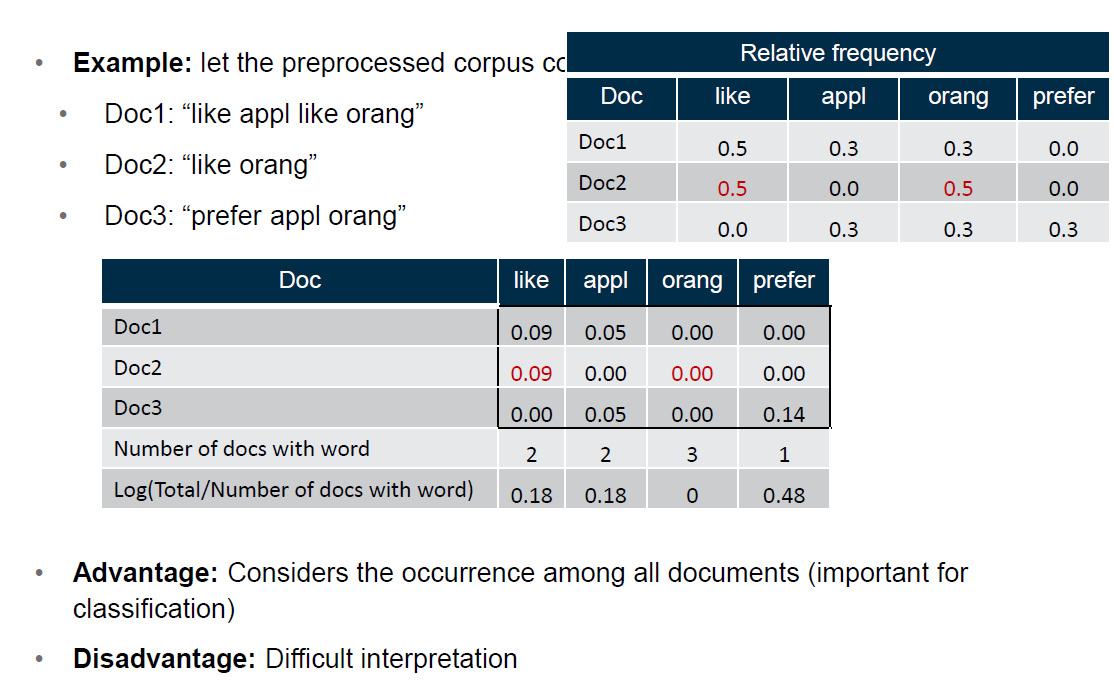
# Bag-of-words: Relative frequency



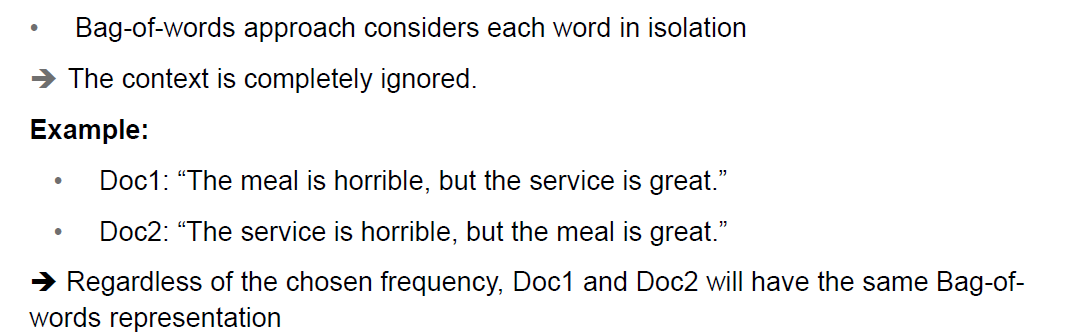
It’s important to not overweight common words:

# Bag-of-words: TF-IDF frequency

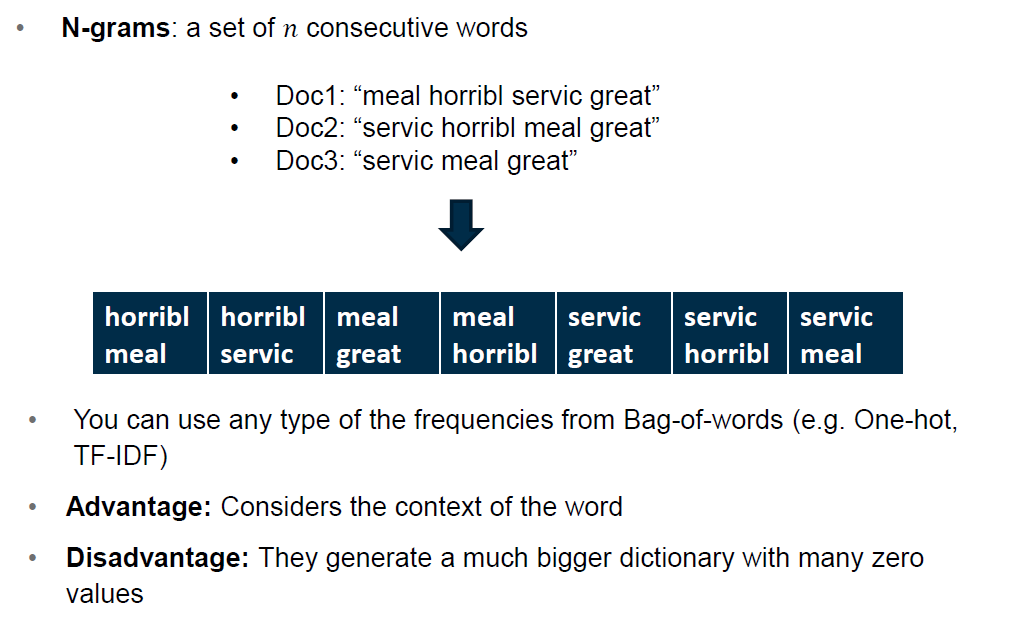




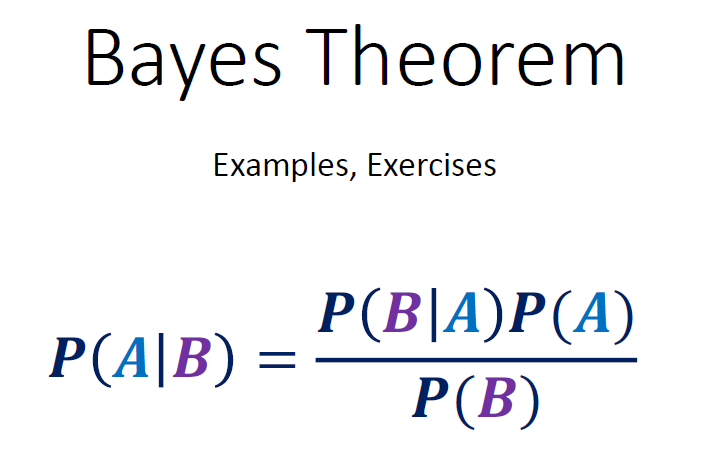
# Disadvantages of Bag-of-words:



# N-grams



3rd Lesson



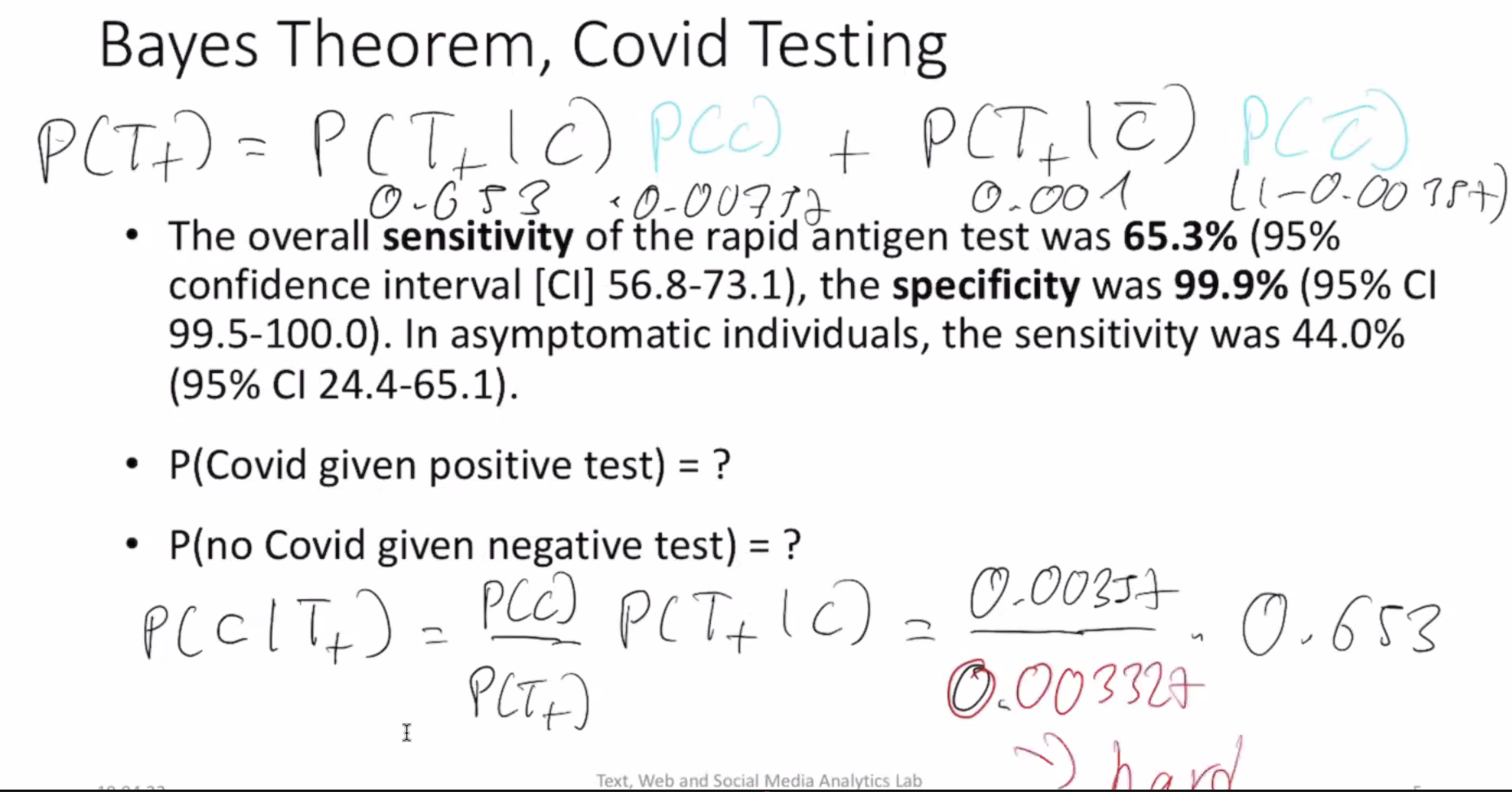
Baseline fallacy:

P(G|A) = 0.001

P(A|G) = P(A) / 0.001 x 0.001 = P(A)

Law of total probability:



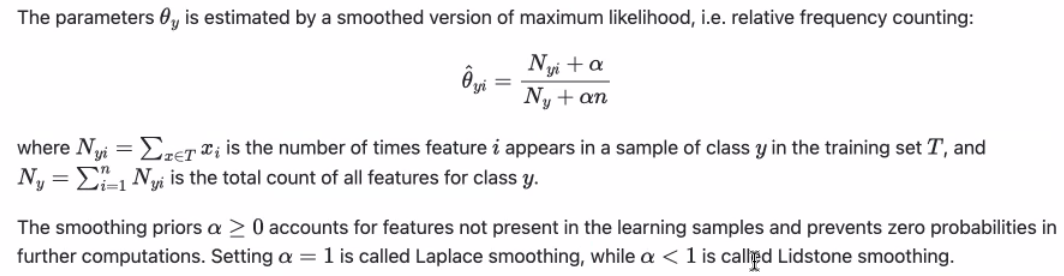


**Naïve bayes**: you never get the true probability out.

Simplifications:

1. It’s useful to leave the denominator outside because it’s shared between all choices.
2. You have multiple features.

Multinomial Naïve Bayes:



**Low count smoothers:** prior knowledge



**Neural Networks**

Regression = only 1 output node

Multinomial = different output nodes, one for each class

It’s done with the **keras** library

Number of weights 1st step: 10.000 x 16 features + 16 bias

Number of weights 2nd step: 16 x (16+1)

4th Lesson. Lesson\_LatentFactors.pdf

Objective: **reduce dimensions**.

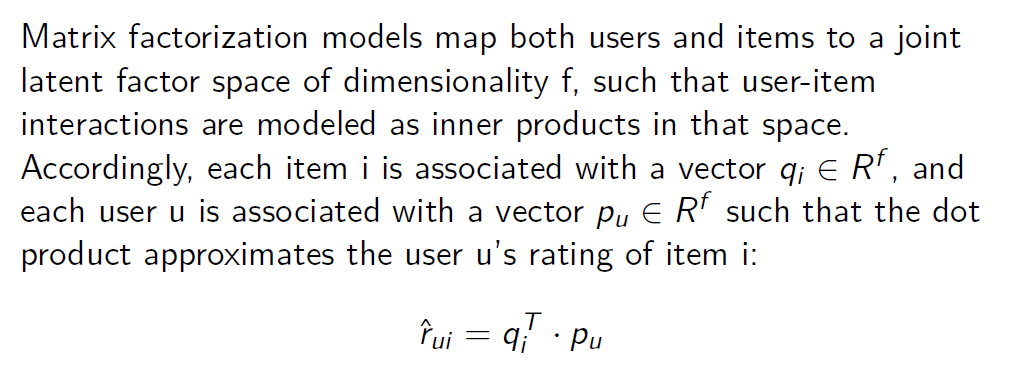
Unsupervised methods (no Y). Ex: k-means.

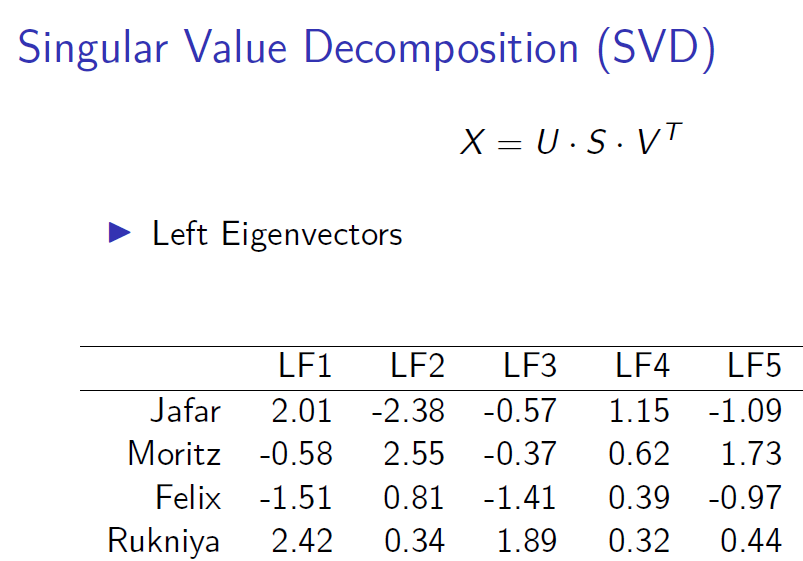
Combine vectors:



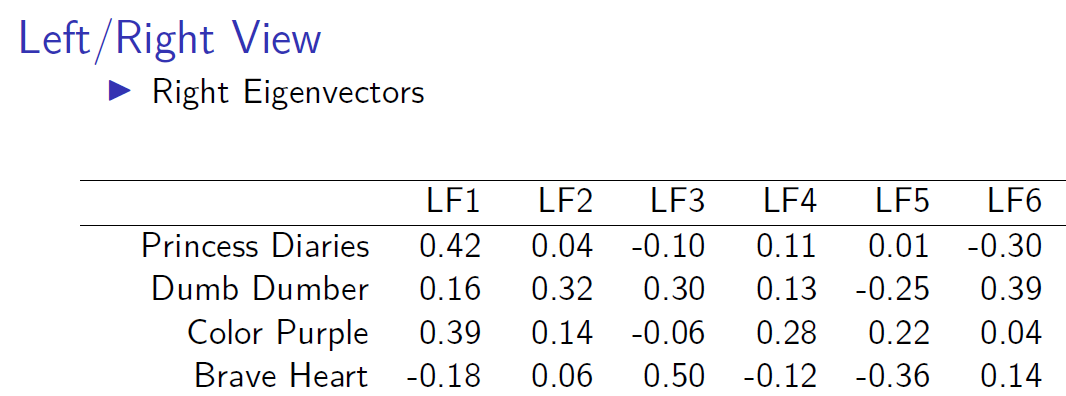
Critique: why only two vectors? Who chooses the features and why?

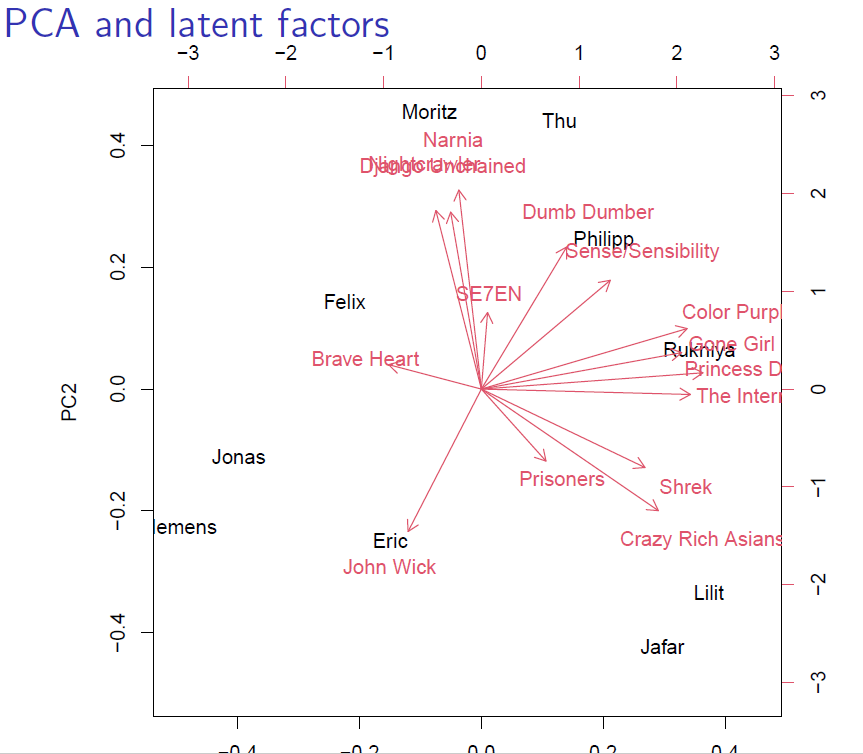
**Matrix factorization**





LF1 is a feature and the numbers define how affine are the persons to it.





Recommendation engines:

* Content filtering
* Collaborative filtering – relies on past behaviors

Text Representation 2. Embedding-based approaches

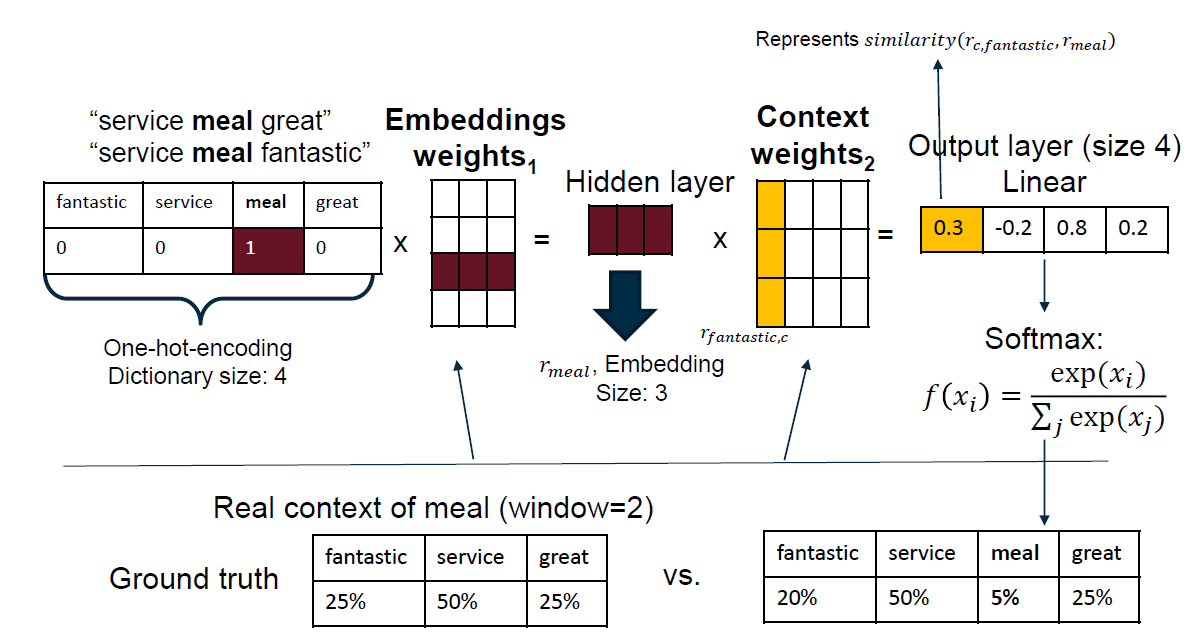
Bag-of-words approaches don’t consider the general meaning of the word in relation to others and thus do not allow for the comparison of documents that do not share words.

**Embeddings** (distributional representation) represent words in a continuous vector space such that similar words are close together. They also allow for negative values.

**Embeddings: Word2vec**

**Word2vec** is a model for embeddings. Idea: **word that appear in similar context must have similar embedding representation**. It’s based on a **two-layer neural network**.

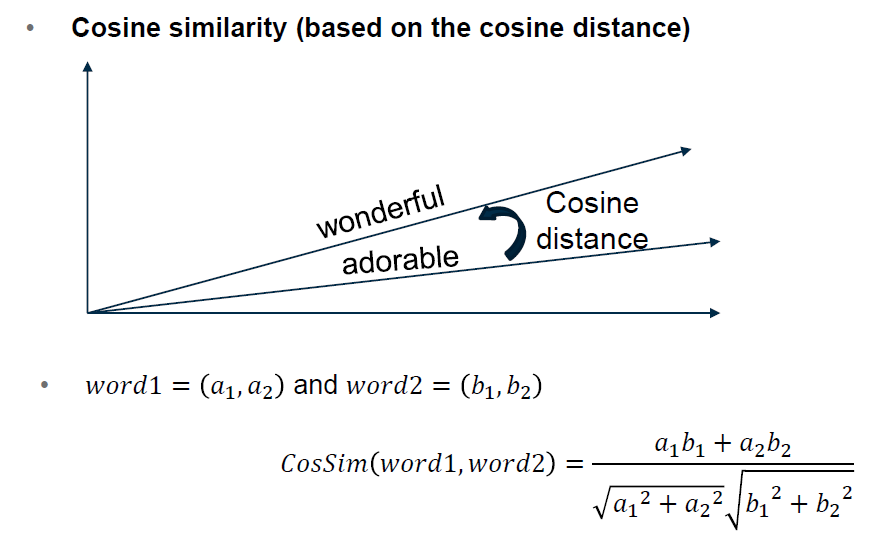
Two approaches: continues-bag-of-words and skip-gram



The resulting embeddings:

* Are derived based on the similarity between words
* Allow for vector arithmetic that mimic the semantic meaning of a word
* Are the same for all usages of the word

Calculate **similarity between two representations**:



Ex. Slide 17

**Advantages Word2vec**: considers relationship of the word meaning to other words

**Disadvantage Word2vec**: vector representation is always the same regardless of particular use (bank office vs river bank vs blood bank). Solution: Contextual Embeddings

Recorded Lesson. Transformers

Problem of N-grams: high dimensionality

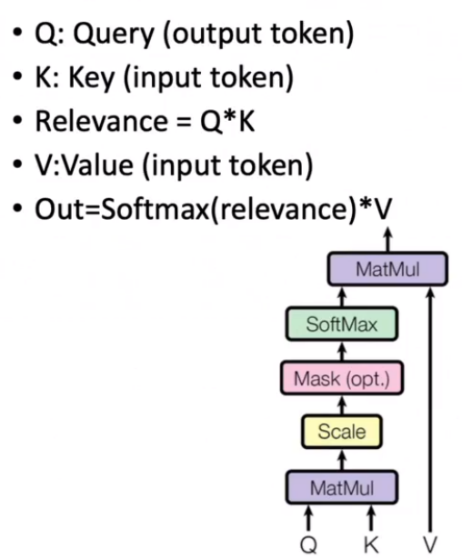
**RNN**: it’s like a *for loop*. class of **artificial neural networks** where connections between nodes form a directed or undirected graph along a temporal sequence. This allows it to exhibit temporal dynamic behavior. They work really well **only for short sequences**.

The scaler number of the matrixes tells how much the matrix grows (exploding gradient) or shrinks (vanishing gradients).

**LSTM**: difficult to train, long gradient paths, transfer learning never worked, needs specific labeled data sets for each new task.

**Transformers: Main Architecture**

Transformers follow an encoder-decoder transformer architecture stemming from translation systems. The original transformer paper uses six encoders and six decoders. Order matters.

Muppet models = Elmo, Bert…

**Attention** mechanism:

* All-to-all comparison for doc length N
* Every output is a weight sum of every input
* The weighting is a learned function

**Transformers** are very suitable for **translation tasks**. **But** many **NLP** tasks aim at classification and require embeddings (not translated ones) as inputs to a classification model.

***How do we apply transformers to classification tasks?***

* Use only decoders, but then a word can only see the previous words and not the next one (loss of info.).
* Use only encoders, but then you can’t train on next word prediction.

**BERT (Bidirectional encoder representations from transformers)**

Trained on Wikipedia and BookCorpus data. The training was done by 1) masking randomly words in the text and predicting them and 2) next sentence prediction. The models can be adapted for other tasks like sentiment analysis.

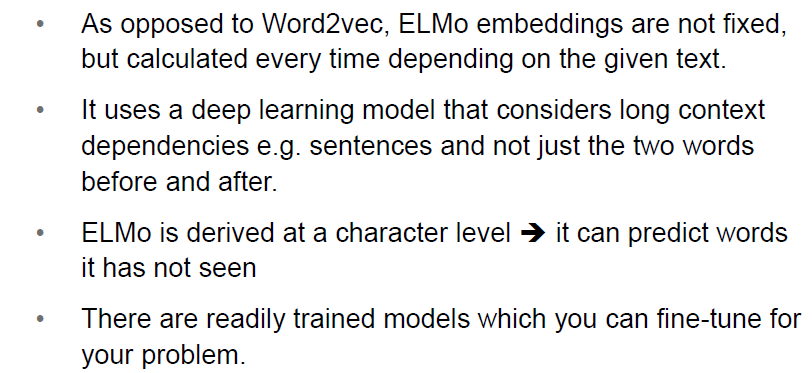
BERT uses as input **WordPiece** embeddings, where words like “playing” are split into “Play” and “#ing”.

**Token embeddings** are the dictionary IDs for each token in a text

**Sentence embeddings** distinguish between sentencers (relevant only for training on sentences).

**Positional embeddings** indicate the position of each word.

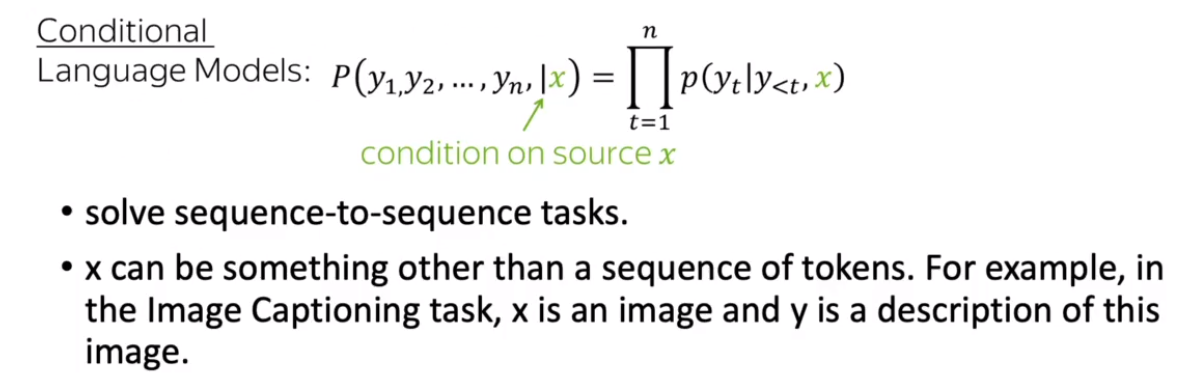
**ELMO (Embedding for Language Models)**



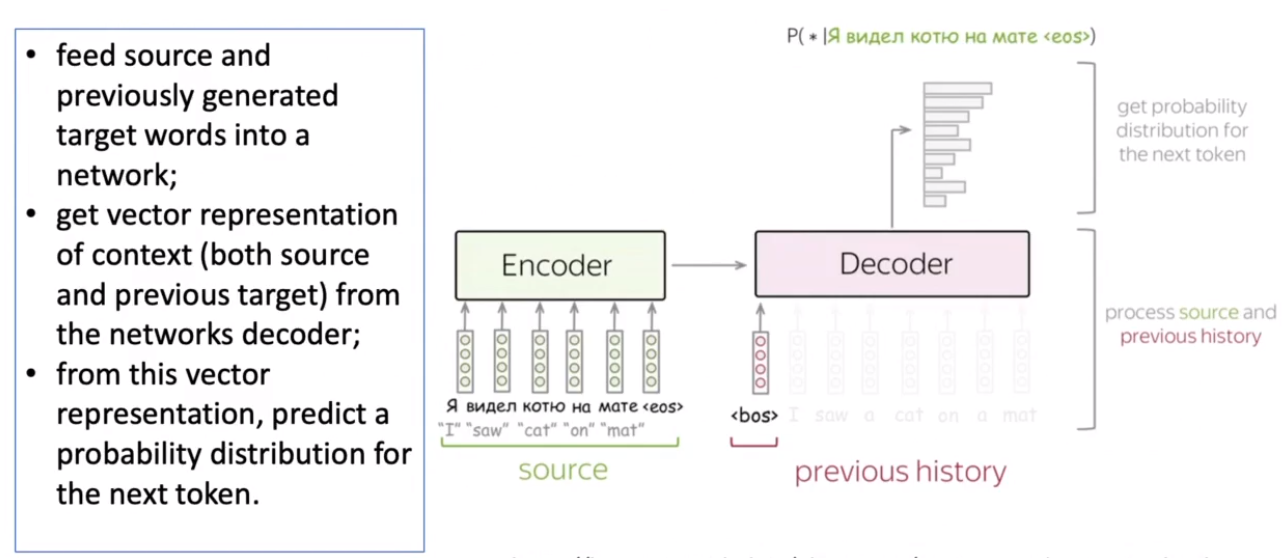
* ELMo uses a 2-layer-long short-term memory neural network: a NN capable of modelling long-term dependencies.
* The network is **bi-directional** such that both forward and backward language model are trained.
* The results from the last layer create **prediction** **probabilities**.

Disadvantage: training is difficult to parallelize due to recurrent nature.

**Conditional Language Models**

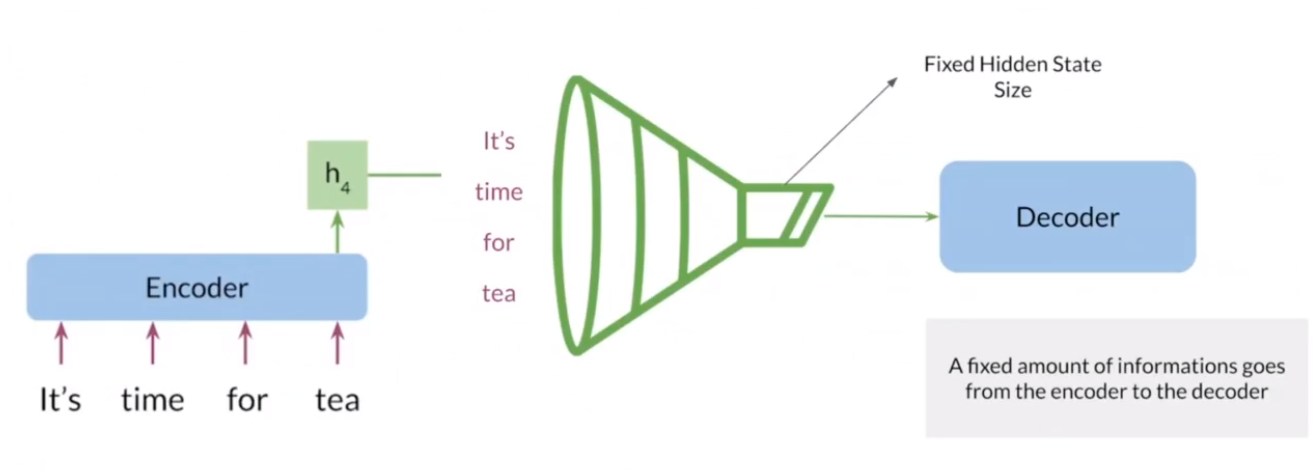


**Encoder-Decoder Framework**

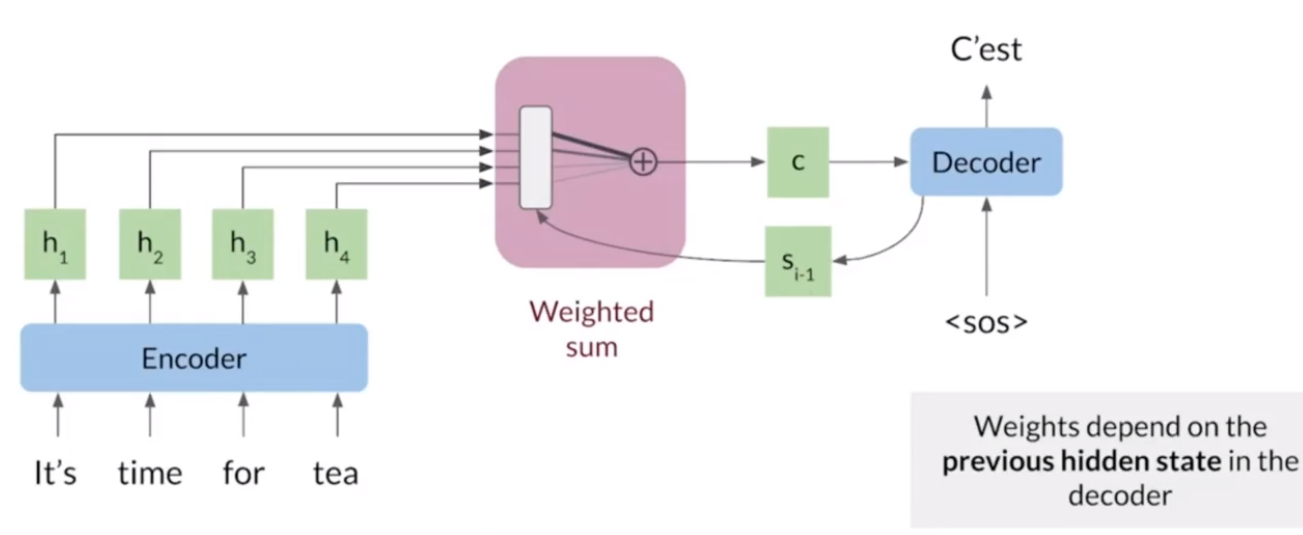


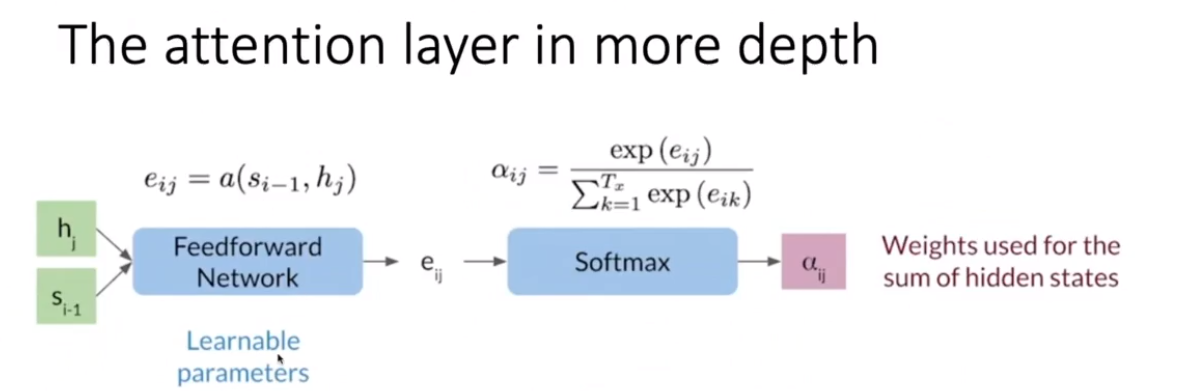
**Seq2seq**

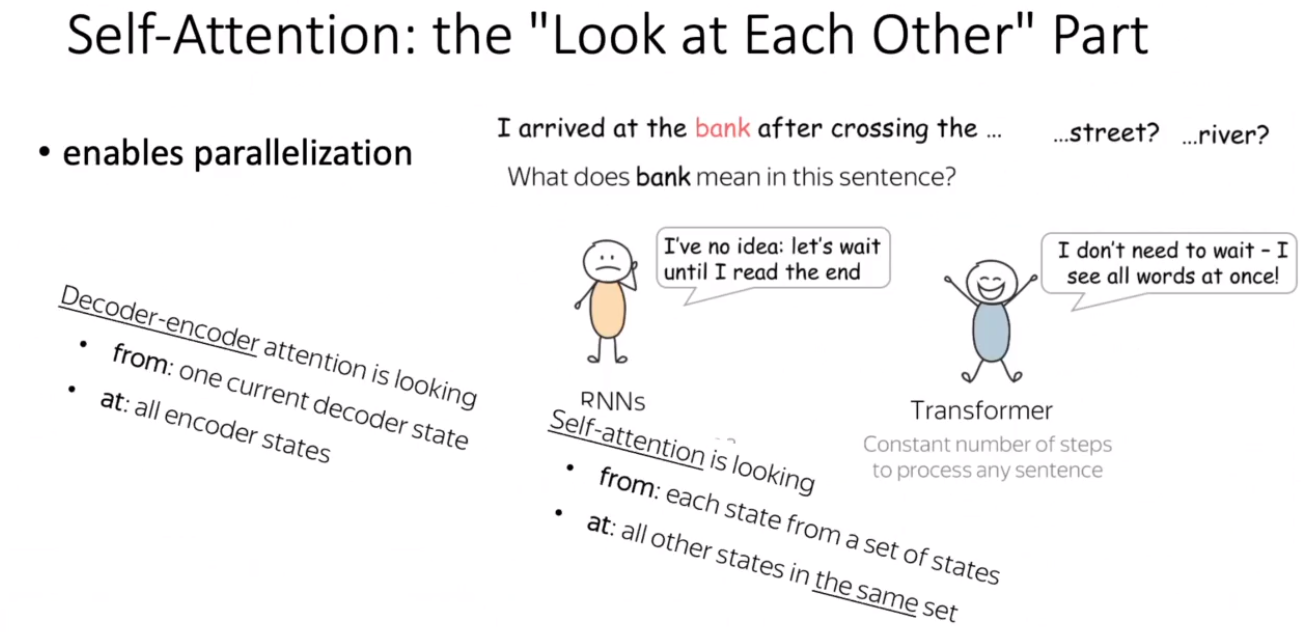
Information bottleneck: long sentences bring problems.

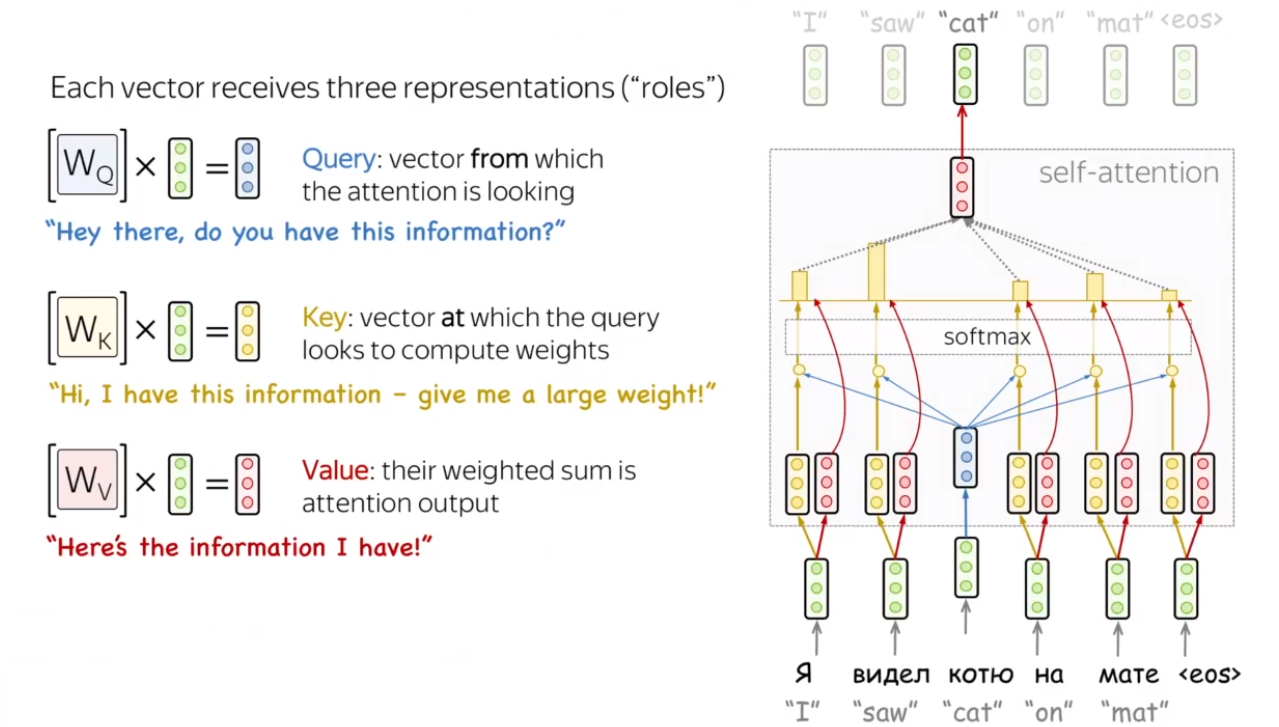


**Solution: attention**









Translation purposes. Uses Attention.

Attention reduces dimensionality to the important vectors. Attention tries to capture relation between words. It also tries to know which word is more related to the one being studied.

Multi-head Attention: Multiple Attention matrices that are learnt separately.

**Transformer’s Tokenizer**: 101 marks the beginning of a phrase and 102 the end.

Outputs.last\_hidden\_state.shape = (2, 13, 768)

13 is the longest token phrase length. 768 is the last

GPU

GPU works well with matrixes. CPU is for low latency, GPU for parallel processing.

Transfer Learning

Take a pretrained model and update the weights with the addition of your own data.

Neural Networks - Side Topic Overfitting

Controlling overfitting by:

* Capacity/Size
* Epochs
* Regularization (Weight decay)
* Dropout
* Data (Image) authentication

How to adjust your Neural Network to avoid overfitting:

* Bigger/Smaller set
* Weight regularization *from keras import regularizers*
* Adding dropout: dropping random neurons
* Data augmentation: really boosts accuracy

Using a pretrained convnet

Fine tuning: tuning one specific block

HuggingFace Briefing. Processing Data

**Natural Language Processing**: understand words and the context. There are different types of tasks for NLP:

* Classifying sentences
* Classifying words
* Generating text content
* Extracting an answer from text
* Generating a new sentence from input text

**Transformers**: the pipeline() function connects a model with its preprocessing and postprocessing steps, allowing to input any text.

Transformers are trained on large amounts of raw text in a **self-supervised** learning type = data not labeled. This model develops a statistical understanding on language, but not really practical. Then the model goes through **transfer learning**, so it’s supervised by human-annotated labels on a given task.

**Encoders**: at each stage the **attention** layers can access all the words in the initial sentence. These models have **bi-directional attention**, and are often called auto-encoding models. Pretraining of these is normally done by *corrupting* a given sentence and tasking the model.

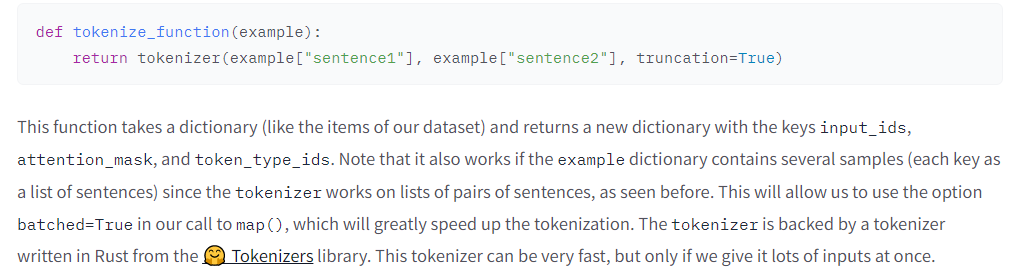
**Decoder models**: for a given word the **attention** layers can only access the words positioned **before** in the sentence. These models are called **auto-regressive models**. Their pretraining revolves around predicting the next word in the sentence. They are used for **text generation**.

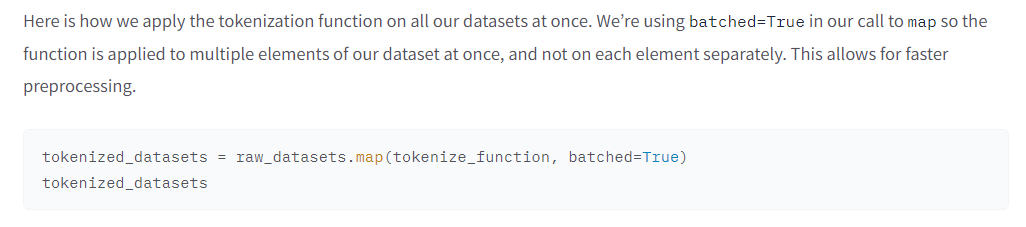
**Sequence-to-sequence models**: **encoder-decoder** models use both parts of the Transformer architecture. They are best suited for tasks like **generating new sentences depending on a given input**, such as summarization, translation or generative question answering.

**Tokenizers:** are one of the core components of the NLP pipeline. They **translate text into data that can be processed by the model**. Models can only process **numbers**.

**Processing the data**: in general you don’t need to worry about whether or not there are *token\_type\_ids* in your tokenized inputs: as long as you use the same checkpoint for the tokenizer and the model, everything will be fine as the tokenizer knows what to provide to its model.

Speed up tokenization:

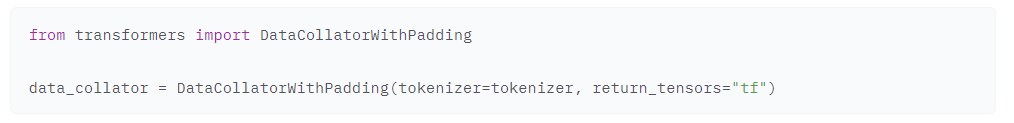




**Dynamic padding**: pad all the examples to the length of the longest element when we batch elements together. **Collate function**.

Speeding up in Padding:

We have deliberately postponed the padding, to only apply it as necessary on each batch and avoid having over-long inputs with a lot of padding. This will speed up training by quite a bit, but note that if you’re training on a TPU it can cause problems — TPUs prefer fixed shapes, even when that requires extra padding.

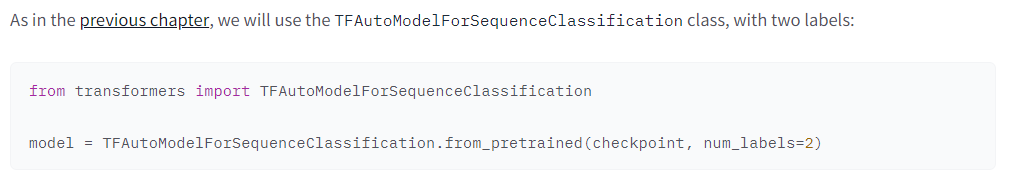


All preprocessing explained: <https://huggingface.co/course/chapter3/2?fw=tf>

HuggingFace Briefing. Fine-tuning a Keras model

Set **GPU** before fitting the model.

Preprocessing steps: <https://huggingface.co/course/chapter3/3?fw=tf#finetuning-a-model-with-keras>

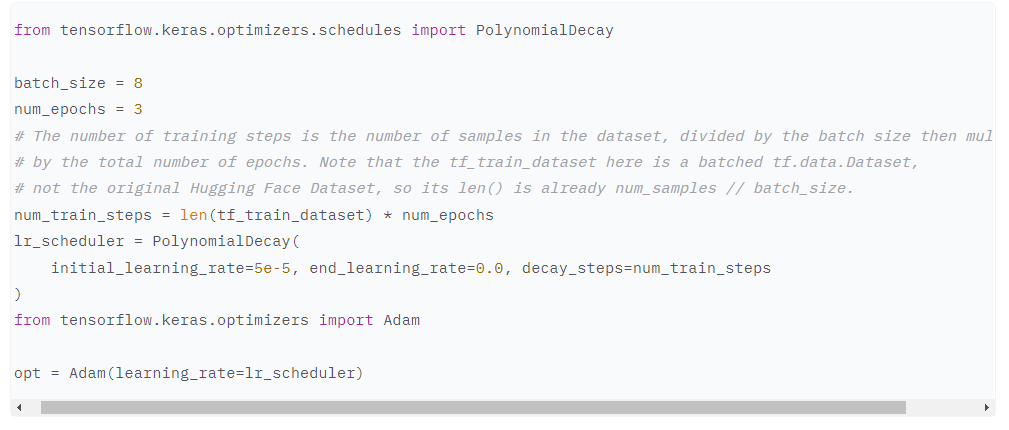


Transformers use a loss function automatically if you don’t set a loss argument in *compile()*. You will need to pass your labels as part of the input, not as a separate label (which is the way Keras works).



To **improve training performance**, we will lower the **learning rate**, as transformers benefit from this (5e-5, 0.00005).

We will also **reduce the learning rate over the course of training** with a **learning rate scheduler**, e.g. **PolynomialDecay**. For that we have to compute how long training is going to be with *num\_train\_steps*.

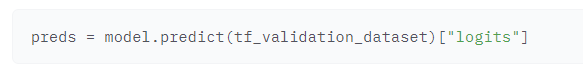


Transformers library has a **create\_optimizer()** that will create an AdamW optimizer with learning rate decay.

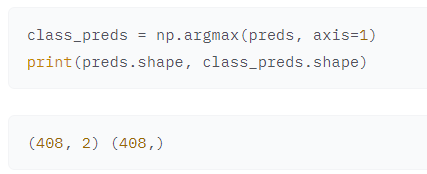


**Predictions**

We can use *predict()* which will return the output head of the model, one per class.



We can convert these logits into the model’s class predictions by using argmax to find the highest logit, which corresponds to the most likely class:



**Metrics**

