Machine Learning for Graphs and Sequential Data

Sequential Data – Neural Network Approaches

lecturer: Prof. Dr. Stephan Günnemann
www.daml.in.tum.de

Summer Term 2020

Data Analytics and
Machine Learning

Roadmap

- Chapter: Temporal Data / Sequential Data
  1. Autoregressive Models
  2. Markov Chains
  3. Hidden Markov Models
  4. **Neural Network Approaches**
     a) **Word Vectors**
     b) RNNs
     c) Non-Recurrent Models (ConvNets, Transformer)
  5. Temporal Point Processes

Introduction

- Text is everywhere
- Applying machine learning to textual data to solve machine translation, question answering, sentiment analysis etc.
- Example:

  It's a brilliant, honest performance by Nicholson,
  but the film is an agonizing bore except when the
  fantastic Kathy Bates turns up.

- Goal: given text predict whether it is positive or negative
- Problem: how to represent words to input them into a subsequent model
- One solution: one-hot encoding
  - High dimensional
  - Too sparse
  - Assumes the words are **independent** of each other

Introduction

- Words as vectors while keeping the underlying language properties
- E.g. similar words should have vectors near each other
- **Distributional hypothesis** – words that appear in similar contexts have similar meanings

You shall know a word by the company it keeps.
J. R. Firth

- Example: *hotel* and *motel*
  - Can be used interchangeably in many sentences while remaining meaningful
- However: *duck* – an animal vs. *duck* – to lower head quickly

Introduction

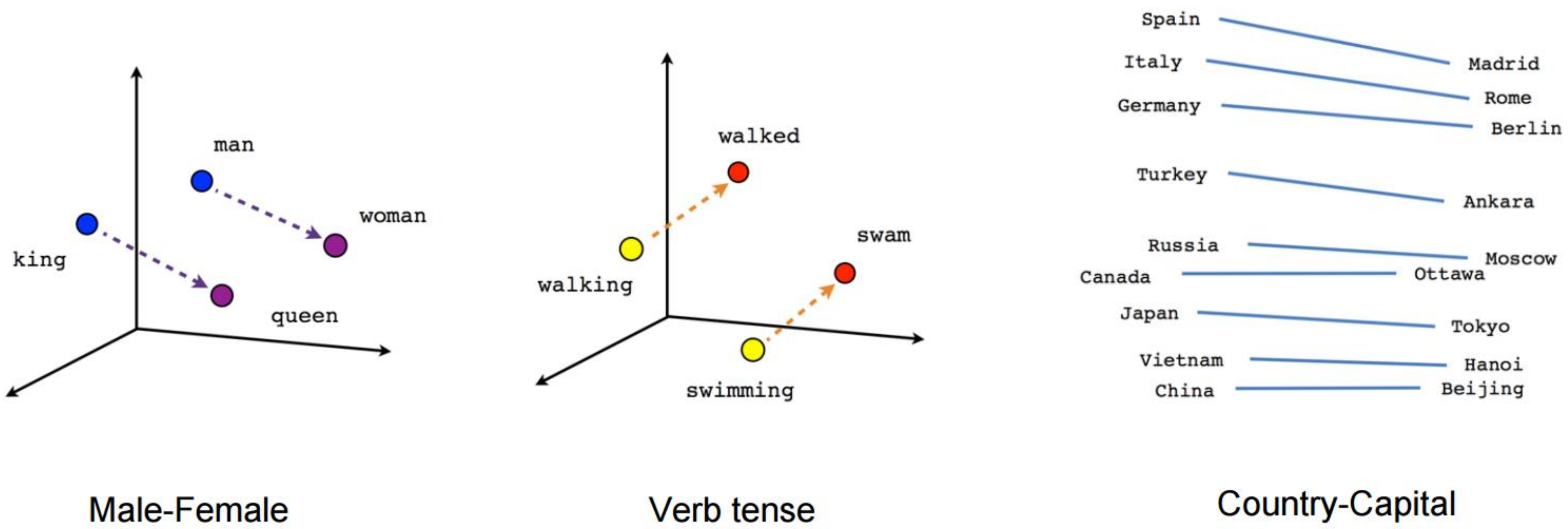


Illustration of how vectors can represent linguistic concepts
Figure from https://www.tensorflow.org/tutorials/representation/word2vec

Co-occurence Matrix

- To be aware of **context** we can simply count how many times each word appeared beside other words
- If the text is given with words $\{x_1, \dots, x_N\}$, then a window of size l around a word x_i is $\{x_{i-l}, \dots, x_{i-1}, x_{i+1}, \dots, x_{i+l}\}$
- We slide this window over sentences and count the co-occurences
- Example:

I like dogs. I like cats too. They hate each other.

- For the first sentence the windows ($l = 1$) are:
  – (Ø, like) (I, dogs) (like, .) (dogs, Ø)

Co-occurence Matrix

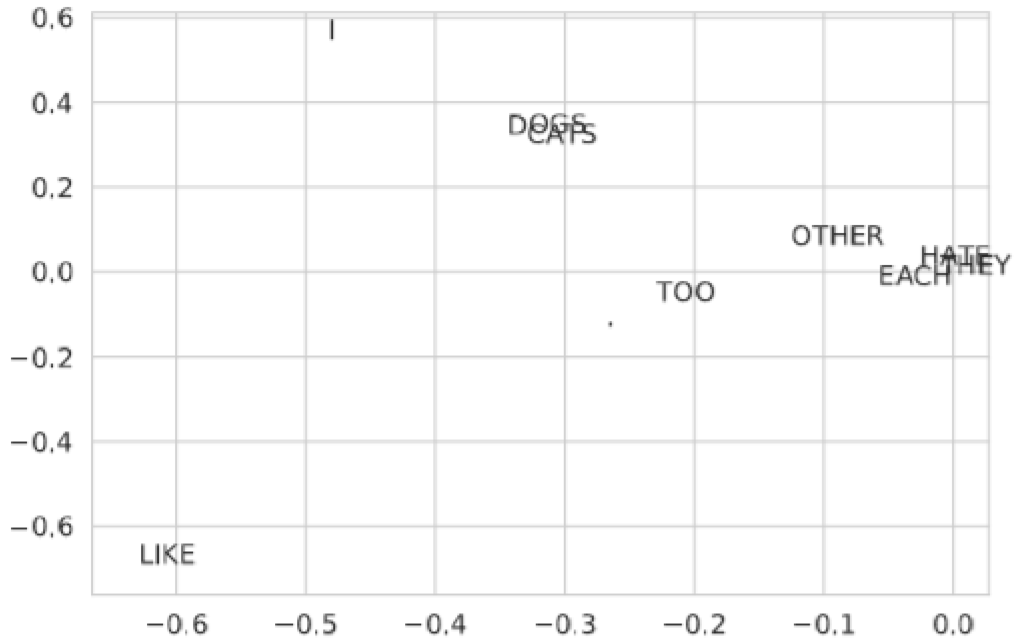
- After counting all the pairs we get a co-occurence matrix $\boldsymbol{M}$:

	.	I	cats	dogs	each	hate	like	other	they	too
.	0	0	0	1	0	0	0	1	0	1
I	0	0	0	0	0	0	2	0	0	0
cats	0	0	0	0	0	0	1	0	0	1
dogs	1	0	0	0	0	0	1	0	0	0
each	0	0	0	0	0	1	0	1	0	0
hate	0	0	0	0	1	0	0	0	1	0
like	0	2	1	1	0	0	0	0	0	0
other	1	0	0	0	1	0	0	0	0	0
they	0	0	0	0	0	1	0	0	0	0
too	1	0	1	0	0	0	0	0	0	0

- Pros: similar words have similar vectors
- Cons: still high dimensional and sparse
- Solution: reduce the dimension to get dense vector of fixed dimension

SVD

- We can reduce the dimension with an SVD decomposition: $\boldsymbol{M} = \boldsymbol{U\Sigma V}^T$
- If we take the first D columns of $\boldsymbol{U}$ we get D-dimensional word vectors
- Applied to the previous example ($D = 2$):



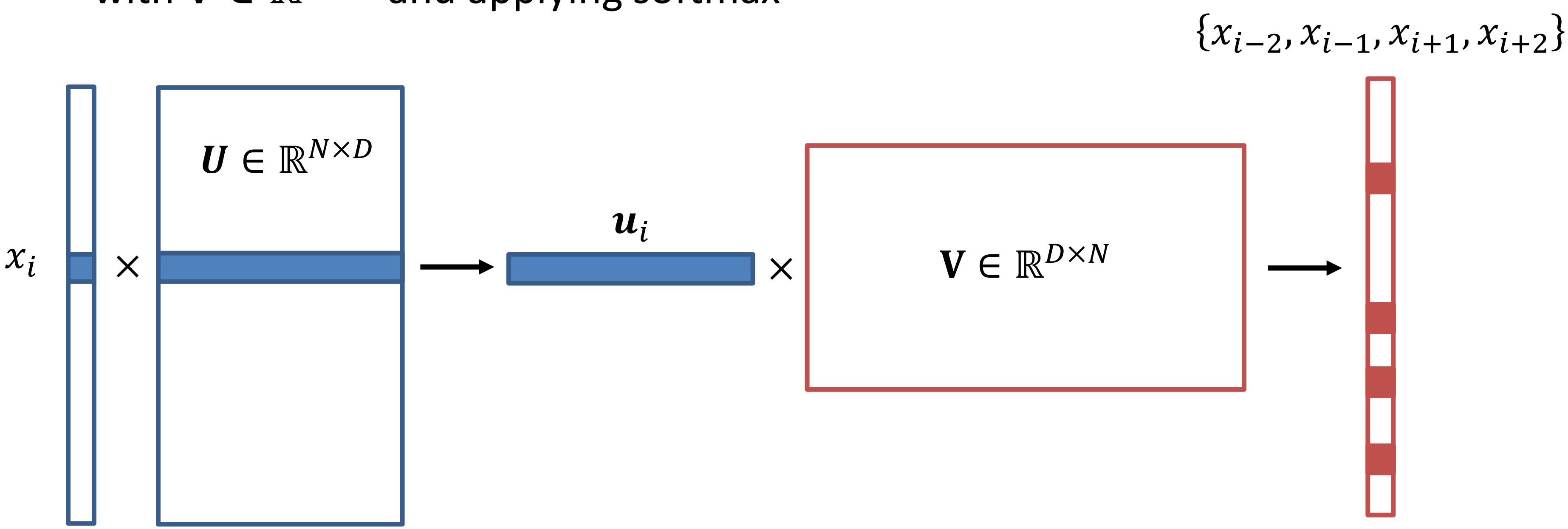
- Problems: slow computation and hard to add new words

Word2Vec

- A different way to get word vectors is with a neural network
- Task: prediction of words based on context
- Two approaches:
  - **Continuous bag-of-words (CBOW)**
    - Predicts a word from the words surrounding it (window)
    - Not good for rare words because the model might not predict them from context
  - **Skip-gram** [1]
    - Predicts the surrounding context from the current word
    - Given a rare word it must *understand* it to predict the context
    - Slower to train but can work well with smaller amounts of data and with rare words

Skip-gram

- Input: one-hot vector with dimension N
- Embedding: project the word to D-dimensional space with $\boldsymbol{U} \in \mathbb{R}^{N \times D}$
  - Since input has zeros everywhere except on ith position, multiplication is equivalent to taking ith row of $\boldsymbol{U}$
- Prediction: get probabilities of context words by multiplying embedding with $\mathbf{V} \in \mathbb{R}^{D \times N}$ and applying softmax



Skip-gram

- Formally: if S $= \{x_{i-l}, \dots, x_{i-1}, x_{i+1}, \dots, x_{i+l}\}$ is a window of size l around the word x_i, and $\boldsymbol{\theta}$ denotes model parameters, the objective is

$$\max_{\boldsymbol{\theta}} \mathbb{E}[P(S|x_i, \boldsymbol{\theta})] = \min_{\boldsymbol{\theta}}(-\mathbb{E}[P(S|x_i, \boldsymbol{\theta})])$$

$$\text{where } P(S|x_i, \boldsymbol{\theta}) = \sum_{x_k \in S} P(x_k|x_i, \boldsymbol{\theta})$$

$$\text{and } P(x_k|x_i, \boldsymbol{\theta}) = \text{softmax}(\boldsymbol{u}_i \boldsymbol{V})$$

- The vector $\boldsymbol{u}_i$ is the corresponding embedding
- We can choose to set $\boldsymbol{U} = \boldsymbol{V}$, giving less parameters to optimize but also less expressiveness

References

- [1] Mikolov, Tomas et al. (2013). "Efficient estimation of word representations in vector space". In: arXiv preprint arXiv:1301.3781.
- [2] Morin, Frederic and Yoshua Bengio (2005). "Hierarchical probabilistic neural network language model." In: Aistats. Vol. 5. Citeseer, pp. 246–252.

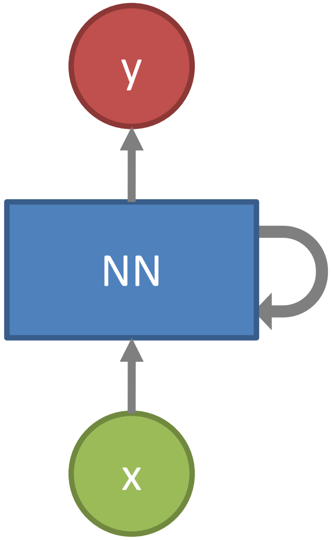
Roadmap

- Chapter: Temporal Data / Sequential Data
  1. Autoregressive Models
  2. Markov Chains
  3. Hidden Markov Models
  4. **Neural Network Approaches**
     a) Word Vectors
     b) RNNs
     c) Non-Recurrent Models (ConvNets, Transformer)
  5. Temporal Point Processes

Introduction

- In word embeddings, we learn a representation for every individual word
- How to process an entire sequence with neural networks?
  - In particular if the sequences have varying length?

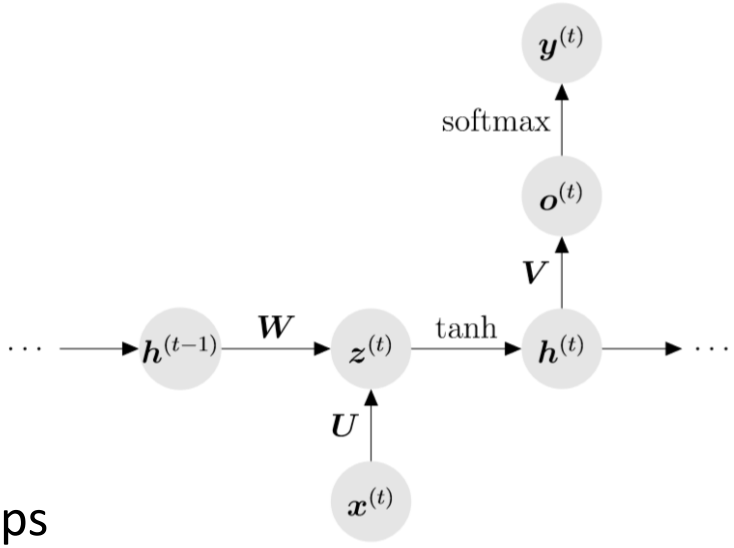
- We can use **Recurrent Neural Networks (RNNs)**



Definition

- Given a sequence of inputs $\{\boldsymbol{x}^{(1)}, \dots, \boldsymbol{x}^{(N)}\}$ and outputs $\{\boldsymbol{y}^{(1)}, \dots, \boldsymbol{y}^{(N)}\}$ we want to know the probability $P(\boldsymbol{y}^{(t)} | \boldsymbol{x}^{(1)}, \dots, \boldsymbol{x}^{(t)})$
- Represent a sequence $\{\boldsymbol{x}^{(1)}, \dots, \boldsymbol{x}^{(t-1)}\}$ with a hidden state $\boldsymbol{h}^{(t-1)}$
- Neural network takes $\boldsymbol{h}^{(t-1)}$ and current input and maps them to a new hidden state $\boldsymbol{h}^{(t)}$ from which we can predict the output at step t
  - Also use $\boldsymbol{h}^{(t)}$ in the next step
- The update equations are

$$
\begin{aligned}
\boldsymbol{z}^{(t)} &= \boldsymbol{W}\boldsymbol{h}^{(t-1)} + \mathbf{U}\boldsymbol{x}^{(t)} + \boldsymbol{b} \\
\boldsymbol{h}^{(t)} &= \tanh\left(\boldsymbol{z}^{(t)}\right) \\
\boldsymbol{o}^{(t)} &= \mathbf{V}\boldsymbol{h}^{(t)} \\
\hat{\boldsymbol{y}}^{(t)} &= \operatorname{softmax}\left(\boldsymbol{o}^{(t)}\right)
\end{aligned}
$$



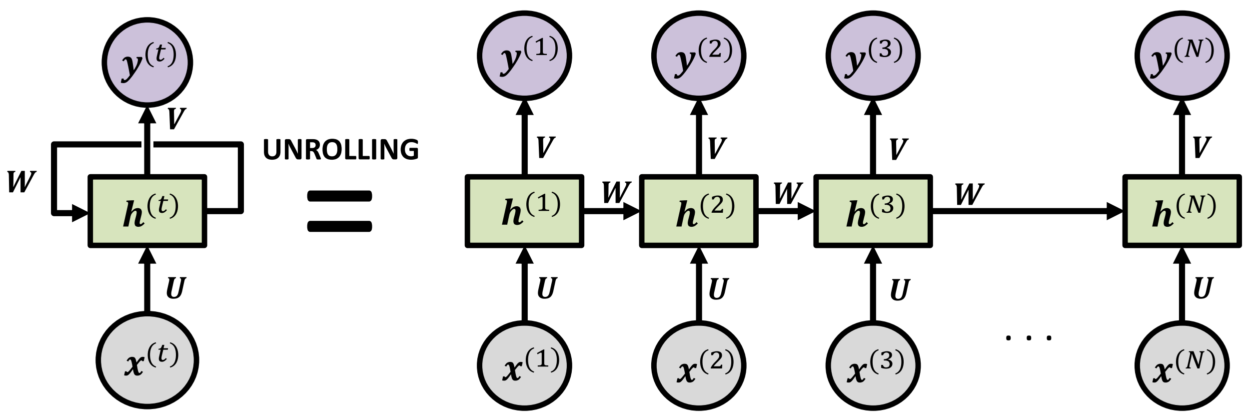
- The weights are shared over all time steps

Objective

- The negative log-likelihood is

$$L = -\log \prod_t p_{\text{model}}\left(\boldsymbol{y}^{(t)} \middle| \boldsymbol{x}^{(1)}, \dots, \boldsymbol{x}^{(t)}\right)$$
$$= -\sum_t \log p_{\text{model}}\left(\boldsymbol{y}^{(t)} \middle| \boldsymbol{x}^{(1)}, \dots, \boldsymbol{x}^{(t)}\right) = -\sum_t L^{(t)}$$

- Network fully differentiable – train with a gradient based method
- Unrolling of the RNN graph



Backpropagation through time

- All functions used in the update equations are differentiable (linear, tanh, softmax) → We can compute the derivative w.r.t the parameters:

$$\frac{\partial L}{\partial \boldsymbol{V}} = \sum_t \left(\hat{\boldsymbol{y}}^{(t)} - \boldsymbol{y}^{(t)}\right)\left(\boldsymbol{h}^{(t)}\right)^T$$

$$\frac{\partial L}{\partial \boldsymbol{W}} = \sum_t \operatorname{diag}\left(1 - \left(\boldsymbol{h}^{(t)}\right)^2\right)\frac{\partial L}{\partial \boldsymbol{h}^{(t)}}\left(\boldsymbol{h}^{(t-1)}\right)^T$$

$$\frac{\partial L}{\partial \boldsymbol{U}} = \sum_t \operatorname{diag}\left(1 - \left(\boldsymbol{h}^{(t)}\right)^2\right)\frac{\partial L}{\partial \boldsymbol{h}^{(t)}}\left(\boldsymbol{x}^{(t)}\right)^T$$

$$\boldsymbol{z}^{(t)} = \boldsymbol{W}\boldsymbol{h}^{(t-1)} + \mathbf{U}\boldsymbol{x}^{(t)} + \boldsymbol{b}$$

$$\boldsymbol{h}^{(t)} = \tanh\left(\boldsymbol{z}^{(t)}\right)$$

$$\boldsymbol{o}^{(t)} = \mathbf{V}\boldsymbol{h}^{(t)}$$

$$\hat{\boldsymbol{y}}^{(t)} = \operatorname{softmax}\left(\boldsymbol{o}^{(t)}\right) \text{ or } \boldsymbol{o}^{(t)}$$

- Since parameters are shared over the steps, final derivative are a sum of all the contributions at every step t.

Backpropagation through time

- The hidden state $\boldsymbol{h}^{(t)}$ recursively depends on all previous hidden states $\boldsymbol{h}^{(t-1)}, \dots, \boldsymbol{h}^{(0)}$ i.e.

$$\boldsymbol{h}^{(t)} = \tanh\left(\boldsymbol{W}\boldsymbol{h}^{(t-1)} + \mathbf{U}\boldsymbol{x}^{(t)} + \boldsymbol{b}\right)$$

- The gradient $\frac{\partial L}{\partial \boldsymbol{h}^{(t)}}$ depends on future times

$$\frac{\partial L}{\partial \boldsymbol{h}^{(t)}} = \boldsymbol{V}^T\left(\widehat{\boldsymbol{y}}^{(t)} - \boldsymbol{y}^{(t)}\right) + \boldsymbol{W}^T \operatorname{diag}\left(1 - \left(\boldsymbol{h}^{(t+1)}\right)^2\right)\frac{\partial L}{\partial \boldsymbol{h}^{(t+1)}}$$

- The impact of future times might **vanish** or **explode** (e.g. 1-D example: $W > 1$ or $W < 1$) → RNN cannot retain information for many steps.

$$\frac{\partial L}{\partial h^{(t)}} = \sum_{s=t}^{N} \frac{\partial L}{\partial h^{(s)}} \frac{\partial h^{(s)}}{\partial h^{(t)}} = \sum_{s=t}^{N} \frac{\partial L}{\partial h^{(s)}} \prod_{t+1 \le k \le s} \frac{\partial h^{(k)}}{\partial h^{(k-1)}} = \sum_{s=t}^{N} \frac{\partial L}{\partial h^{(s)}} \prod_{t \le k \le s} W\left(1 - \left(h^{(k)}\right)^2\right)$$

GRU

- Solution: change the RNN architecture so it can keep information longer
- Main idea: not every input should be fully taken into account when updating the hidden state – update partially with a *gating* mechanism
- **Gated Recurrent Unit (GRU)** [2]

$$\boldsymbol{z}^{(t)} = \sigma\left(\boldsymbol{W}_z\left[\boldsymbol{h}^{(t-1)}, \boldsymbol{x}^{(t)}\right]\right)$$
$$\boldsymbol{r}^{(t)} = \sigma\left(\boldsymbol{W}_r\left[\boldsymbol{h}^{(t-1)}, \boldsymbol{x}^{(t)}\right]\right)$$

Gates

$$\tilde{\boldsymbol{h}}^{(t)} = \boxed{\tanh\left(\boldsymbol{W}\left[\boldsymbol{r}^{(t)} \odot \boldsymbol{h}^{(t-1)}, \boldsymbol{x}^{(t)}\right]\right)}$$

Simple RNN update – gives candidate state

$$\boldsymbol{h}^{(t)} = \boxed{\left(1 - \boldsymbol{z}^{(t)}\right)} \odot \boldsymbol{h}^{(t-1)} + \boxed{\boldsymbol{z}^{(t)}} \odot \tilde{\boldsymbol{h}}^{(t)}$$

How much to take from previous state vs. candidate state

LSTM

- More powerful architecture: **Long Short-Term Memory (LSTM)** [3]
- Introduces a cell state $\boldsymbol{c}^{(t)}$ in addition to $\boldsymbol{h}^{(t)}$ – we have two states

$$\boldsymbol{f}^{(t)} = \sigma\left(\boldsymbol{W}_f\left[\boldsymbol{h}^{(t-1)}, \boldsymbol{x}^{(t)}\right]\right)$$

Forget gate

$$\boldsymbol{i}^{(t)} = \sigma\left(\boldsymbol{W}_i\left[\boldsymbol{h}^{(t-1)}, \boldsymbol{x}^{(t)}\right]\right)$$

Input gate

$$\boldsymbol{o}^{(t)} = \sigma\left(\boldsymbol{W}_o\left[\boldsymbol{h}^{(t-1)}, \boldsymbol{x}^{(t)}\right]\right)$$

Output gate

$$\boldsymbol{c}^{(t)} = \boldsymbol{f}^{(t)} \odot \boxed{\boldsymbol{c}^{(t-1)}} + \boldsymbol{i}^{(t)} \odot \boxed{\tanh\left(\boldsymbol{W}\left[\boldsymbol{h}^{(t-1)}, \boldsymbol{x}^{(t)}\right]\right)}$$

$$\boldsymbol{h}^{(t)} = \boldsymbol{o}^{(t)} \odot \boxed{\tanh(\boldsymbol{c}^{(t)})}$$

Previous cell state

Simple RNN update – LSTM treats it as an input

Update hidden state (now the output) using a cell state

Summary

- LSTM and GRU are two examples of improvements to the basic RNN
- Gating enables *skipping* some inputs to capture long-term dependencies
  - Actually, since it uses an element-wise product, it can *remember* or *forget* per individual dimension of a hidden state
  - Avoids gradient problems that RNN has
- It is fully differentiable so we can derive gradients for all the parameters as in the RNN and train it with, e.g., gradient descent
- Many variations on LSTM architecture
  - E.g. *peephole LSTM* – replaces $\boldsymbol{h}^{(t)}$ with $\boldsymbol{c}^{(t)}$ in all the equations

References

- [1] Cho, Kyunghyun et al. (2014). “Learning phrase representations using RNN encoder-decoder for statistical machine translation”. In: arXiv preprint arXiv:1406.1078.
- [2] Pascanu, Razvan, Tomas Mikolov, and Yoshua Bengio (2013). “On the difficulty of training recurrent neural networks”. In: International conference on machine learning, pp. 1310–1318.
- [3] Hochreiter, Sepp and Jürgen Schmidhuber (1997). “Long short-term memory”. In: Neural computation 9.8, pp. 1735–1780.
- [4] Peters, Matthew E., Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer. "Deep contextualized word representations." *arXiv preprint arXiv:1802.05365* (2018).

Roadmap

- Chapter: Temporal Data / Sequential Data
  1. Autoregressive Models
  2. Markov Chains
  3. Hidden Markov Models
  4. **Neural Network Approaches**
     a) Word Vectors
     b) RNNs
     c) Non-Recurrent Models (ConvNets, Transformer)
  5. Temporal Point Processes

Introduction

- Sometimes when modeling a sequence we do not need the complete history to produce the output
- Example: **generating speech**
  - Raw audio has many data points (16000 per second)
  - Important relations on many time scales
- Recall an autoregressive model:

$$X_t = c + \sum_{i=1}^{p} \varphi_i X_{t-i} + \varepsilon_t$$

- Uses a fixed window of p previous inputs and performs regression
- Can we use neural networks to capture more complex behavior?
  - RNNs share the parameters across time steps, but **depend on full history**
  - We can instead use **Convolutional Neural Networks (ConvNets)**

Recap: Definition

- The convolution $f * g$ of functions f and g is

$$(f * g)(\mathrm{t}) = \int_{-\infty}^{+\infty} f(\tau) g(t-\tau) d\tau$$

- In image processing, given an image I and a kernel K, both 2-D matrices, the convolution can be writen as:

$$(K * I)(i,j) = \sum_m \sum_n I(i-m, j-n) K(m,n)$$

- Output is again a 2-D matrix (transformed image), where an element (pixel) is a sum of its neighbors, weighted by a kernel

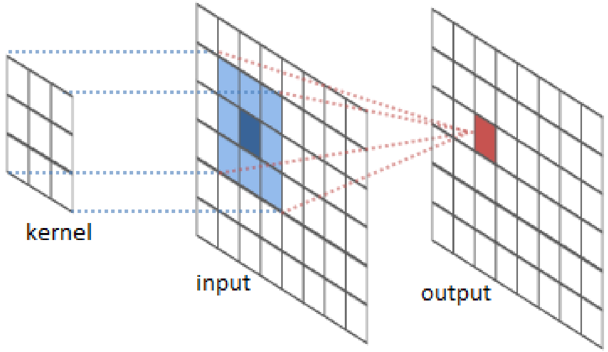


Figure from https://intellabs.github.io/RiverTrail/tutorial/

Recap: Examples

Kernel

Output

$$\begin{matrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{matrix}$$

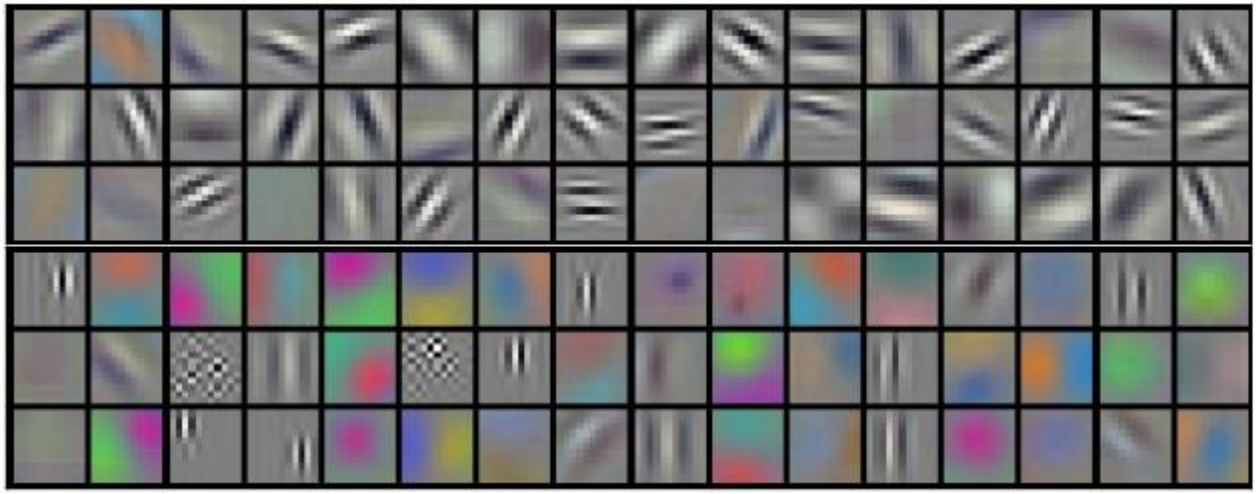
$$\begin{matrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{matrix}$$

$$\begin{matrix} 1/9 & 1/9 & 1/9 \\ 1/9 & 1/9 & 1/9 \\ 1/9 & 1/9 & 1/9 \end{matrix}$$

Source: https://en.wikipedia.org/wiki/Kernel_(image_processing)

Recap: Definition

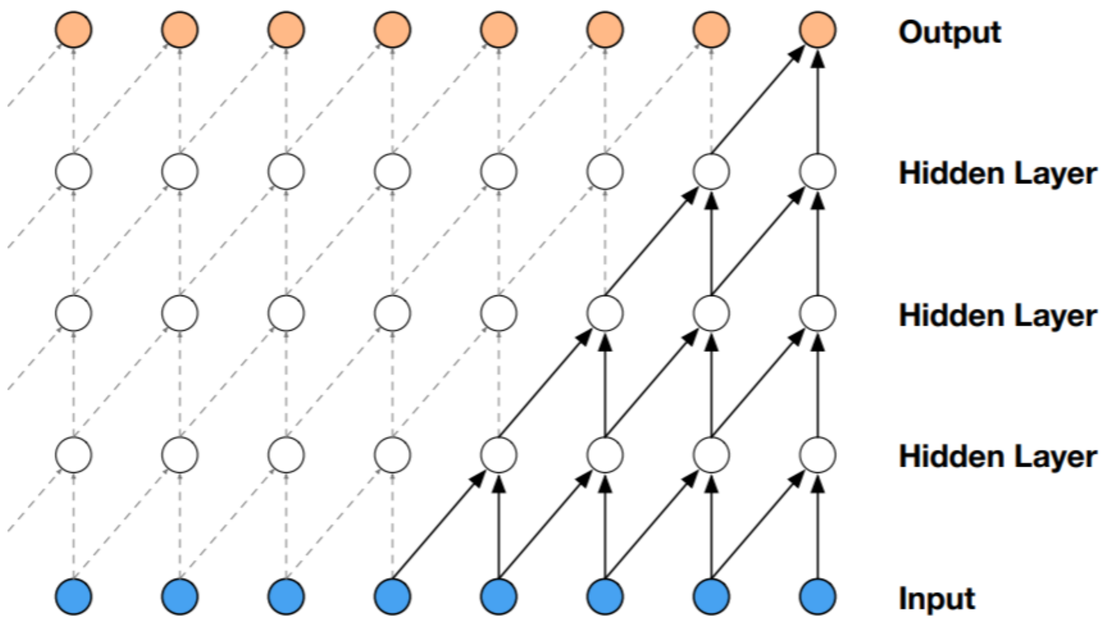
- A **Convolutional Neural Network** contains convolutions as its layers
  - **Kernel** is often called **filter**
  - The parameters/weights of the filter are learned



Example of learned filters [1]

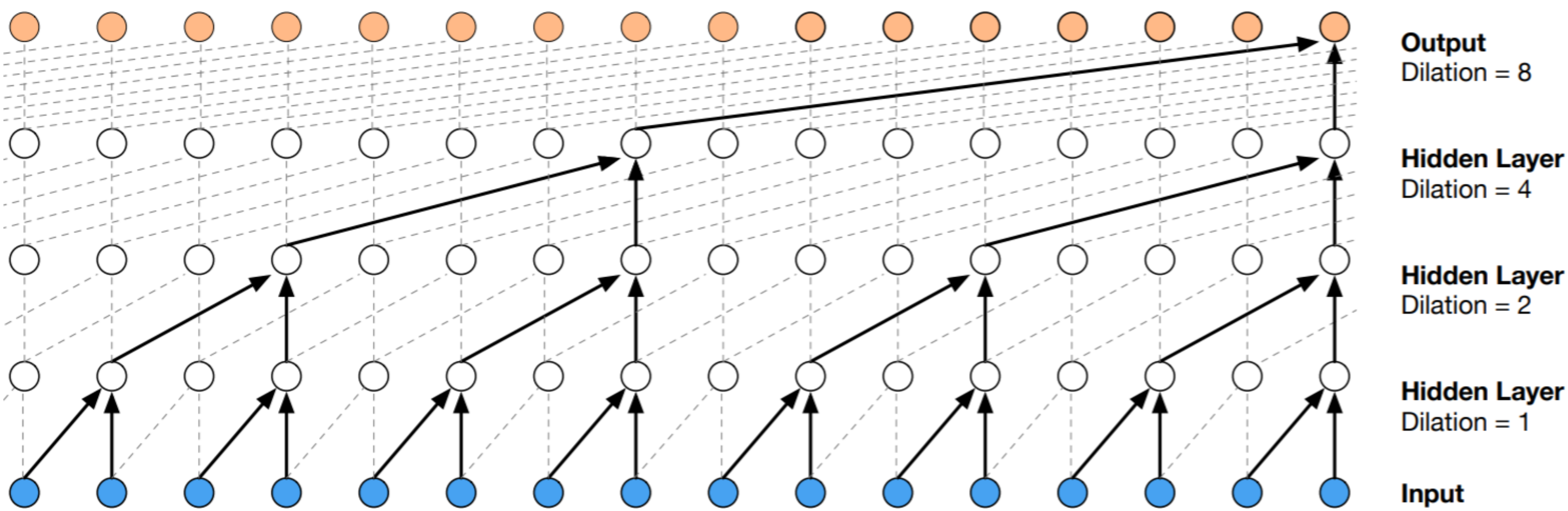
WaveNet

- Sequences are 1-D so we can use a 1-D version of ConvNets
- **WaveNet** [2] is an architecture that uses 1-D ConvNets to model speech
  - In addition, it uses special convolutions to ensure causality and increase receptive field
- **Causal convolutions** – ensure that the output only depends on the past



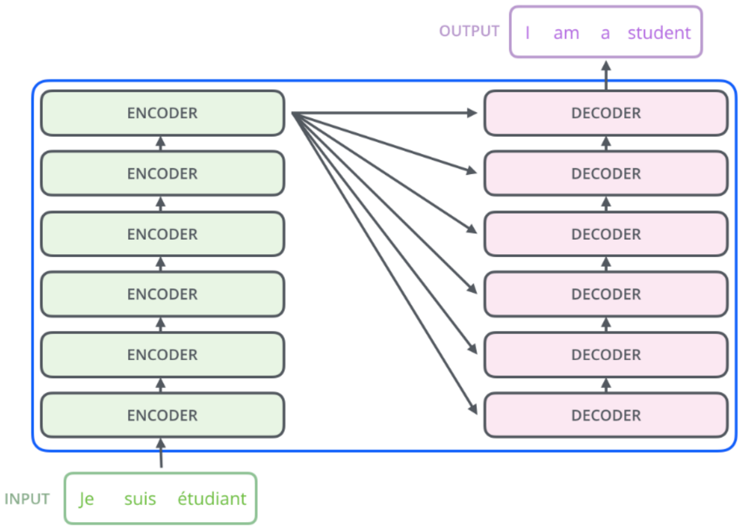
WaveNet

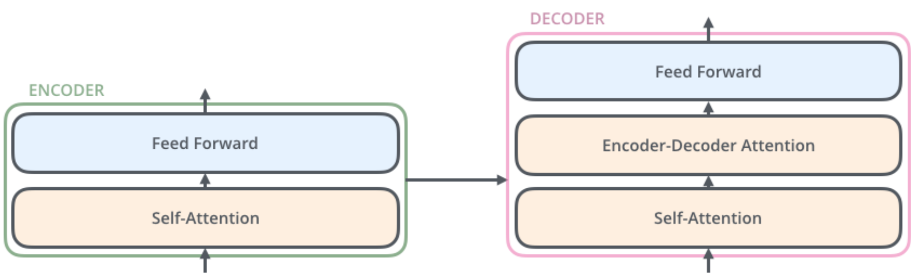
- **Dilated convolutions** – skip some inputs to increase the receptive field
  - Dilation of 1 gives standard convolution
  - If we start with dilation of 1 in the first layer and double it with every layer (2,4,8...) the receptive field will be the exponential of the number of layers
  - E.g. with 4 layers we use 16 inputs in the first layer



Transformers

- **Transfomers** [3] are fast models using attention mechanisms
  - Like WaveNet, it is not a recurrent neural network → parallelizable and fast
  - It is composed of a stack of encoders and decoders using self-attention
  - Achieves high performances in NLP translations

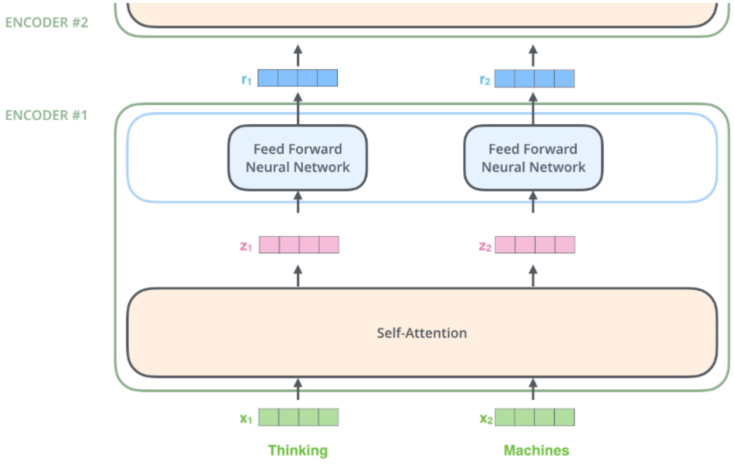




The following images are taken from [4] Jalammar blog

Transformers

- Words are represented with **embeddings** and flow all at once during training.
- Below illustrated by an encoder block
  - The self-attention layer “couples” the embeddings
  - The rest handles the embeddings independently



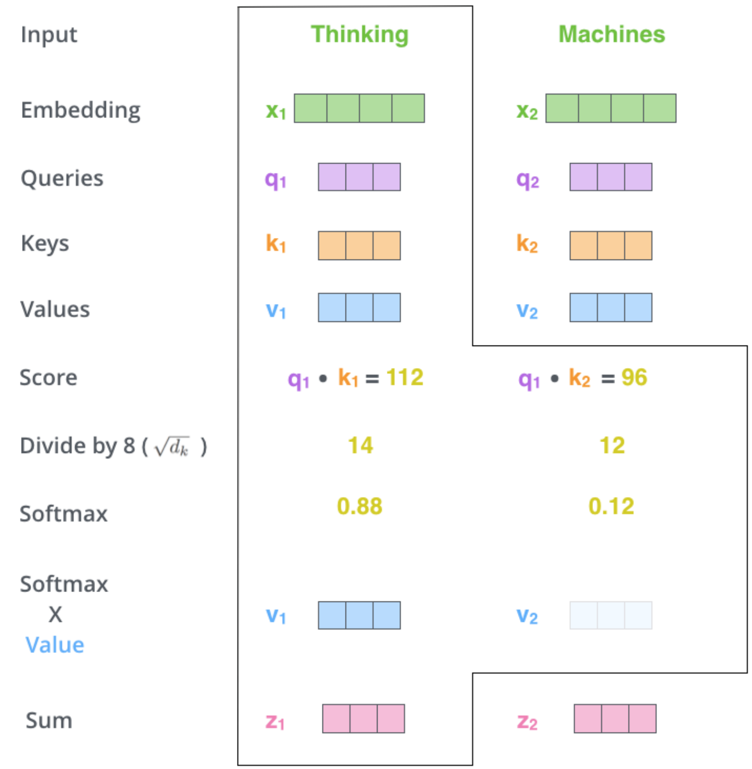
Transformers – Attention

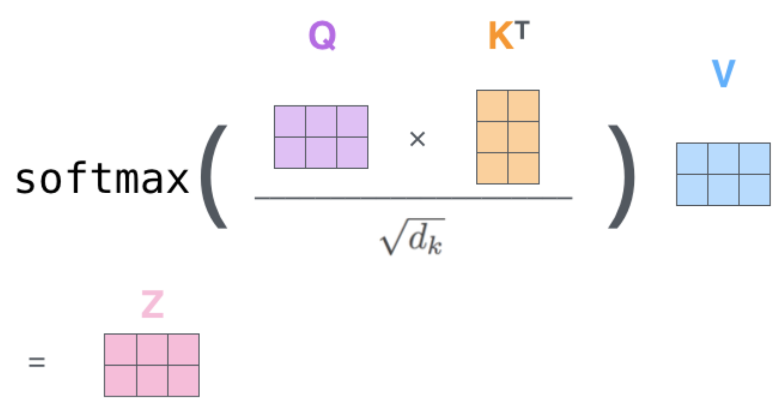
- Attention is a learned weighting over the elements x_j (given element x_i)
  - The weighting is computed by applying softmax to query/key scores
  - Query depends on x_i; key on x_j
  - The weight indicates how much of v_j we use (the "value" of x_j)

- Self-attention: the attention is on the input signal itself

- It is easy computable in a matrix formulation.

"Self-attention allows the model to look at other positions in the input sequence for clues that can help lead to a better encoding for this word" [4]

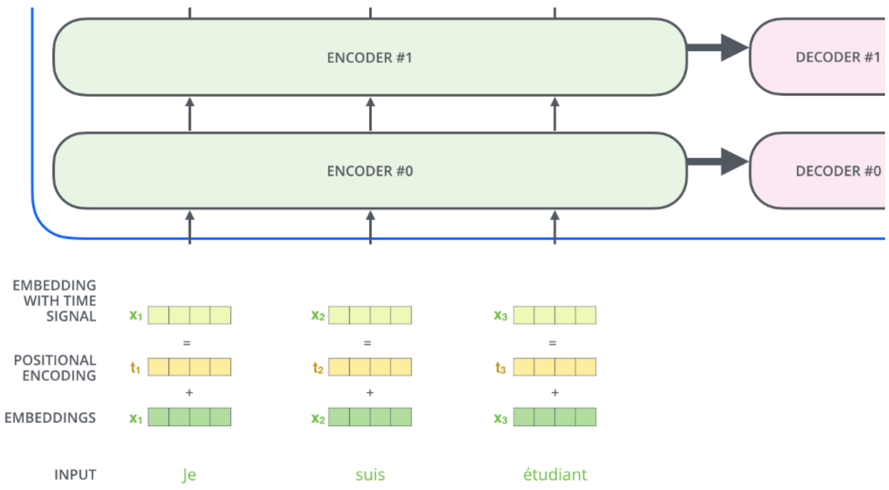


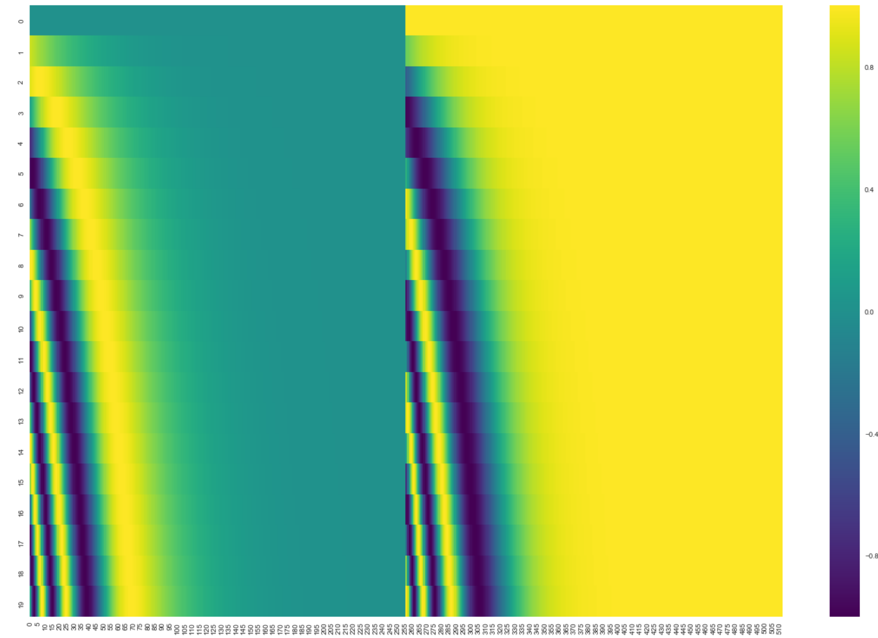


Transformers – Positional Encoding

- A “plain” transformer actually does not care about the order, i.e. the architecture itself is not aware of the non-i.i.d. nature

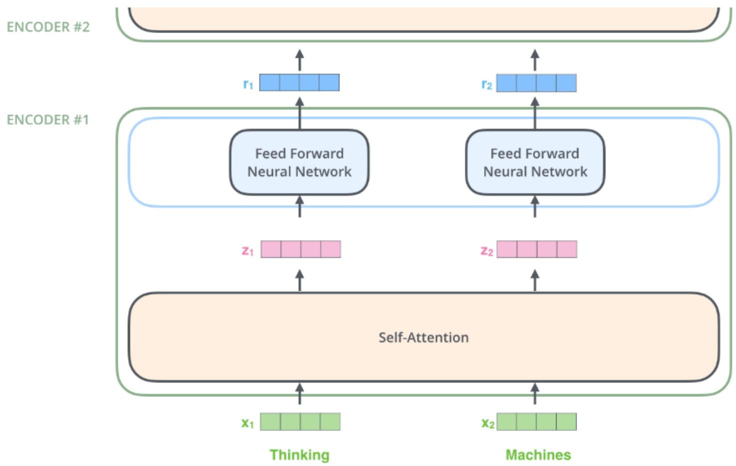
➢ Transformers use a **positional encoding** to represent the word order. Positional encoding: meaningful static vectors which are concatenated with the word embeddings.

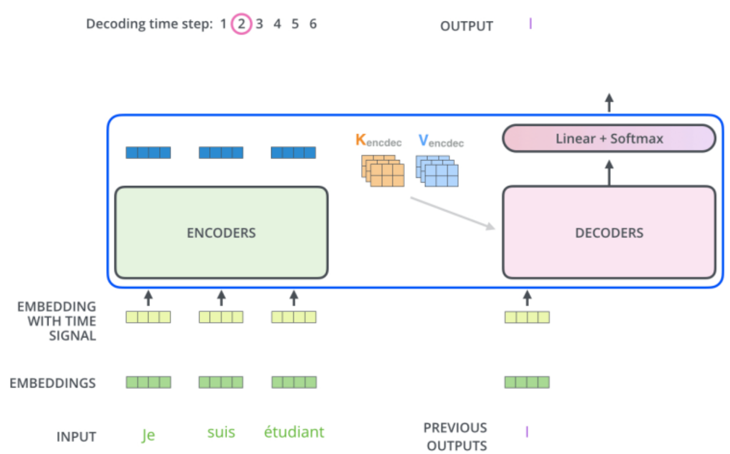




Transformers: Notes on Training and Inference

- During training, the embeddings flow all through the transformer at the same time/in parallel
  - This enables efficient training on very large datasets; crucial for the success of recent models
- At inference time, decoding is done one step after the other until the end of sentence symbol is reached.





Questions – NN

1. In an RNN, the hidden state at a given time influences all hidden states into the future. However, an RNN cannot model long-term dependencies. Why?

2. What is the receptive field of a causal convolution and dilated convolution with n layers ?

References

- [1] Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E. Hinton. "Imagenet classification with deep convolutional neural networks." In *Advances in NIPS*, pp. 1097-1105. 2012.
- [2] Van Den Oord, Aäron, Sander Dieleman, Heiga Zen, Karen Simonyan, Oriol Vinyals, Alex Graves, Nal Kalchbrenner, Andrew W. Senior, and Koray Kavukcuoglu. "WaveNet: A generative model for raw audio." *SSW* 125 (2016).
- [3] Ashish Vaswani et al., "Attention Is All You Need"   NIPS 2017
- [4] Jalammar blog, http://jalammar.github.io/illustrated-transformer/