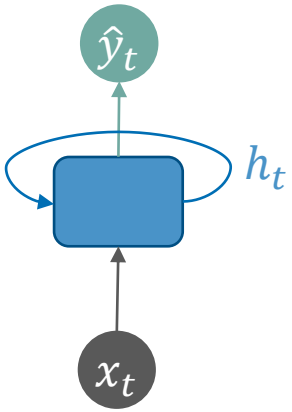
Data Science & Artificial Intelligence

Summer Term 2022

L11 Introduction to Sequence Modeling

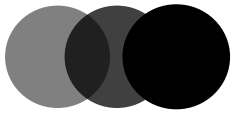
B. Stuhr, J. Haselberger, D. Schneider



L11.1 Sequence Modeling Basics

Why to we need Sequences?

vs.



In which directions does the ball move?

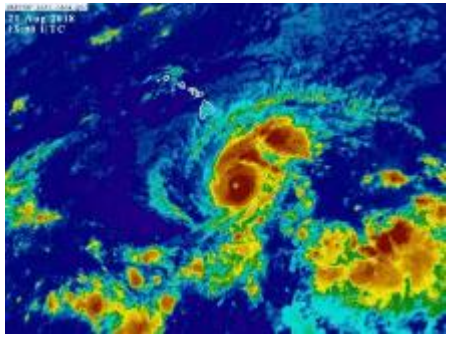
What is Sequence Modeling?

Videos/GIFs

I am a sequence

Text

Audio



Weather

! **Sequence models** interpret, learn from, predict on or generate sequences of data

- Instead of single inputs, a sequence of related inputs is given
  - Sequential data includes text, audio, video, in general time-series data, …

Motivation: Some Applications of Sequence Modeling

SYSTEM PROMPT (HUMAN-WRITTEN)

In a shocking finding, scientist discovered a herd of unicorns living in a remote, previously unexplored valley, in the Andes Mountains. Even more surprising to the researchers was the fact that the unicorns spoke perfect English.

MODEL COMPLETION (MACHINE-WRITTEN, 10 TRIES)

The scientist named the population, after their distinctive horn, Ovid's Unicorn. These four-horned, silver-white unicorns were previously unknown to science.

Now, after almost two centuries, the mystery of what sparked this odd phenomenon is finally solved.

Dr. Jorge Pérez, an evolutionary biologist from the University of La Paz, and several companions, were exploring the Andes Mountains when they found a small valley, with no other animals or humans. Pérez noticed that the valley had what appeared to be a natural fountain, surrounded by two peaks of rock and silver snow.

Pérez and the others then ventured further into the valley. "By the time we reached the top of one peak, the water looked blue, with some crystals on top," said Pérez.

Pérez and his friends were astonished to see the unicorn herd. These creatures could be seen from the air without having to move too much to see them – they were so close they could touch their horns.

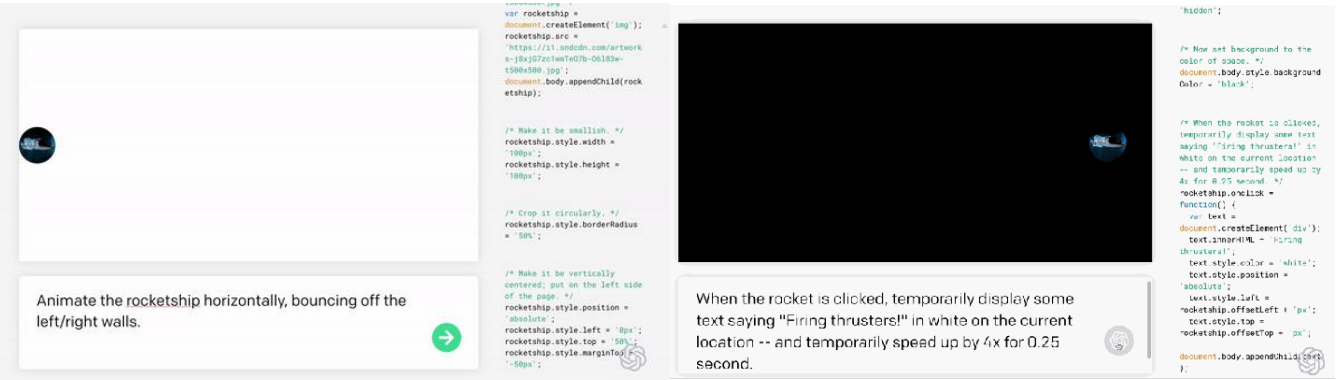
While examining these bizarre creatures the scientists discovered that the creatures also spoke some fairly regular English. Pérez stated, "We can see, for example, that they have a common 'language,' something like a dialect or dialectic."

Dr. Pérez believes that the unicorns may have originated in Argentina, where the animals were believed to be descendants of a lost race of people who lived there before the arrival of humans in those parts of South America.

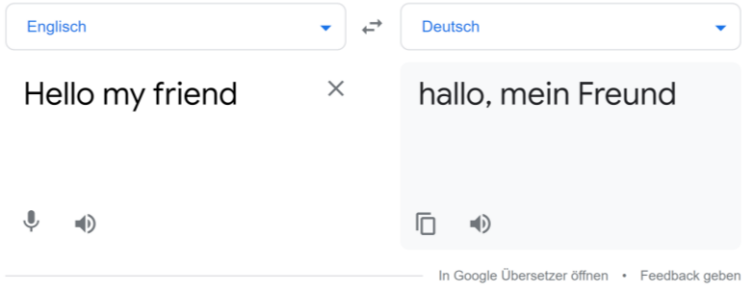
While their origins are still unclear, some believe that perhaps the creatures were created when a human and a unicorn met each other in a time before human civilization. According to Pérez, "In South America, such incidents seem to be quite common."

However, Pérez also pointed out that it is likely that the only way of knowing for sure if unicorns are indeed the descendants of a lost alien race is through DNA. "But they seem to be able to communicate in English quite well, which I believe is a sign of evolution, or at least a change in social organization," said the scientist.

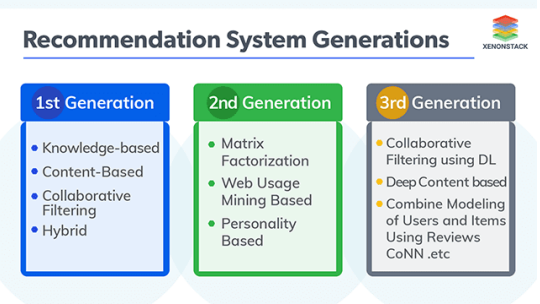
Transformers (GPT2 in this case) create realistic text in a certain style [Link]



OpenAI Codex generates Code from a Description [Link]



Translation [Link]

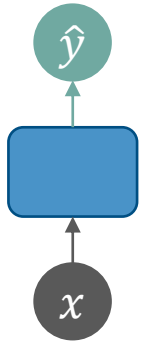


Recommender Systems [Link]

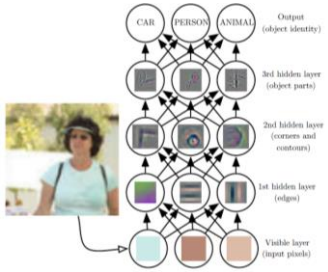
- And other slides from L07!
- And way more: Basically, every application that requires understanding of sequential inputs
- Sequence modeling is widely used
- Methods are transferred to areas, which are not sequence-based

Sequence Modeling: Application Types

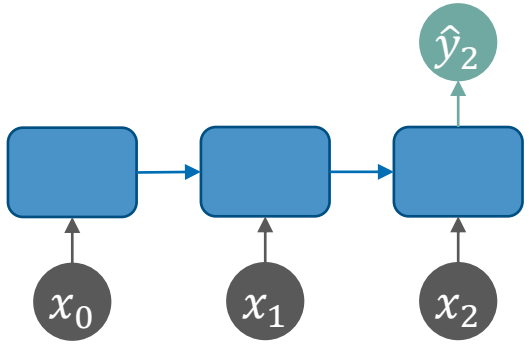
One to One



E.g., Classification



Many to One



E.g., Sentiment Analysis

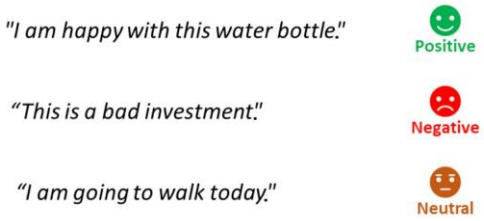
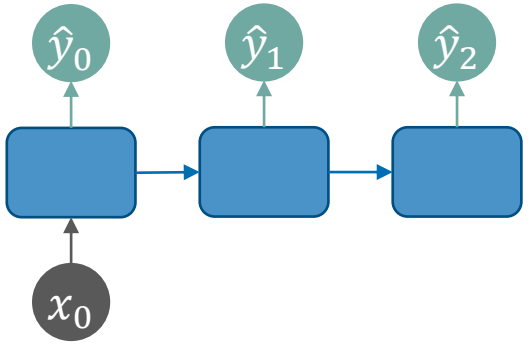


Image from [Link]

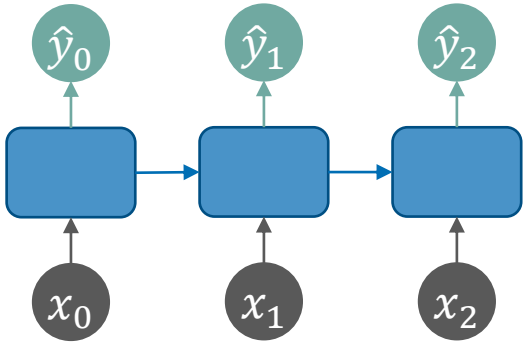
One to Many



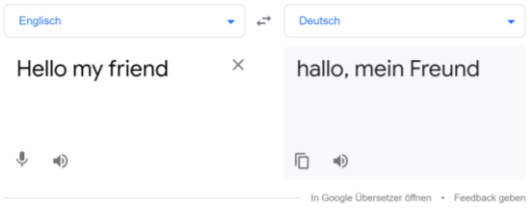
E.g., Image Captioning

Image from [Link]

Many to Many

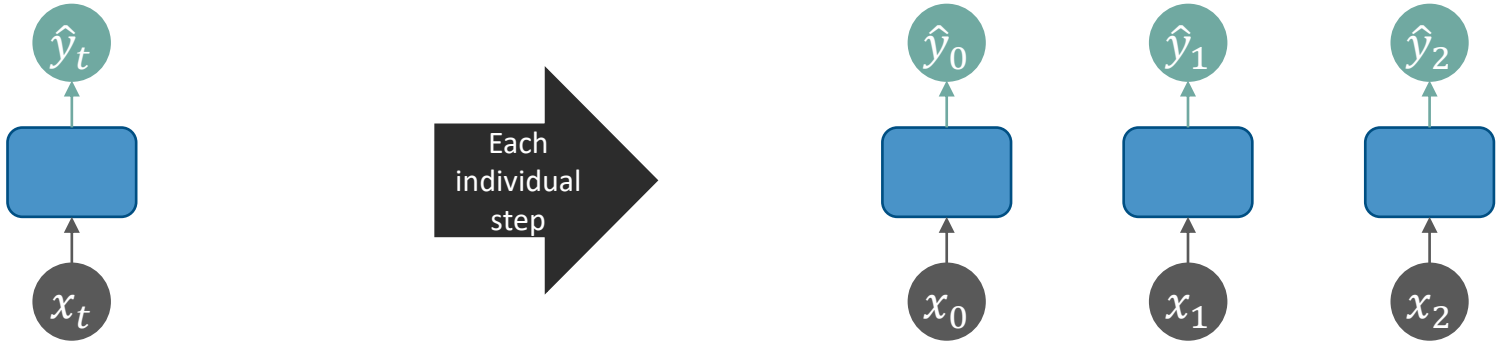


E.g., Translation



From Feed Forward Models to Models with Recurrent Relations

Output at step t,
e.g., “word” of a translated sentence

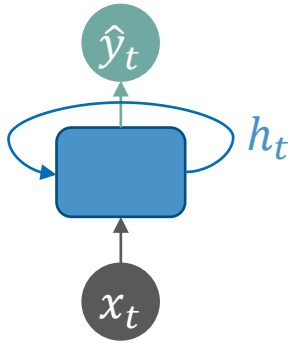


Input at step t,
e.g., “word” of a sentence

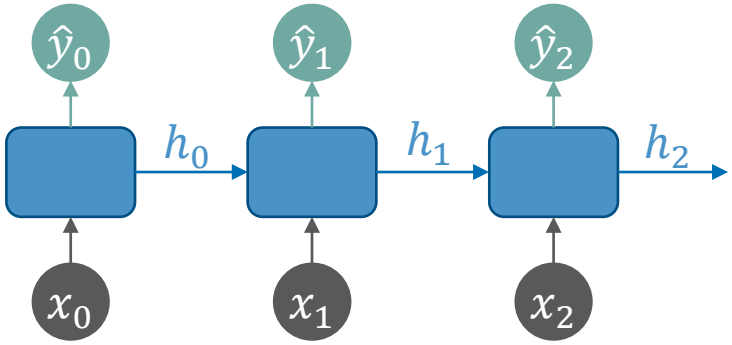
Each step has no information about the **previous** input our output: $\hat{y}_t = \mathrm{f}(x_t)$

From Neurons to Neurons with Recurrent Relations

Output at step t,
e.g., “word” of a translated sentence



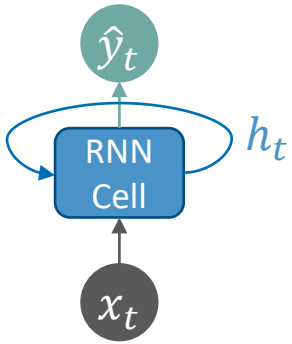
Input at step t,
e.g., “word” of a sentence



We can add a connection to inform each step about the previous step:
$\hat{y}_t = \mathrm{f}(x_t, h_{t-1})$

Recurrent Neural Network (RNNs)

RNN



$$y_t = \mathrm{f}(x_t, h_{t-1})$$

RNN State Update and Output

3. *Output Vector*

$$\hat{y}_t = \boldsymbol{W}_{hy}^T \boldsymbol{h}_t$$

2. *Update Hidden State*

$$\mathbf{h}_t = \tanh(\boldsymbol{W}_{hh}^T \boldsymbol{h}_{t-1} + \boldsymbol{W}_{xh}^T \boldsymbol{x}_t)$$

„fully-connected weights of a dense layer“

activation function

1. *Input Vector*

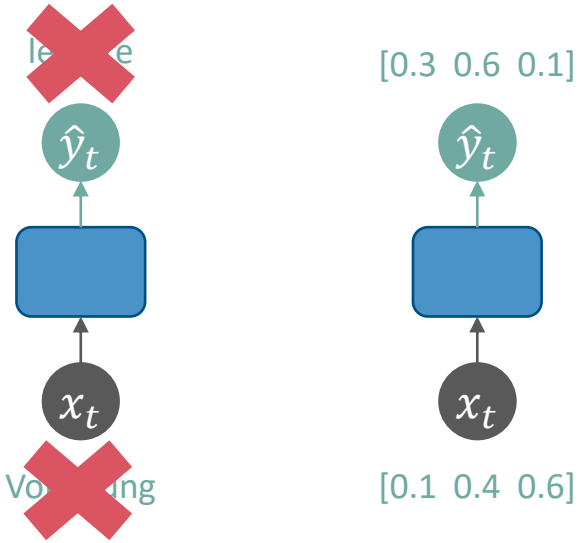
$$\boldsymbol{x}_t$$

Recurrent Neural Networks (RNNs) apply a recurrent relation at every step to process a sequence

Since we use the same network for each step, the weights are shared across these steps

```
torch.nn.RNN(*args, **kwargs)
```

Sequence Embeddings (e.g., Text)



! When we represent non-numerical sequences (e.g., text) to a Neural Network we need to use an **embedding**, to transfer the input into a numerical input

Embedding

1. Vocabulary: corpus of words

hallo, Vorlesung, wie, ..., juhu, wo

2. Indexing: give each word an index

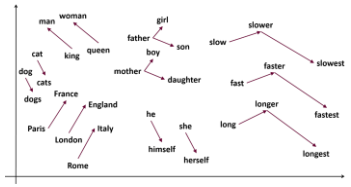
Word	Index
hallo	0
wie	1
juhu	3
Vorlesung	4
...	...

3. Embedding: index to fixed-size vector

„Vorlesung“ = [0 0 0 1 0 0]

i-th index

One-hot Encoding



Learned (e.g., word2vec)

Image from [Link]

RNN Example: Code

```python
import torch.nn as nn

# We define an RNN class that is a subclass of nn.Module.
class RNN(nn.Module):
    def __init__(self, vocab_size, output_dim):
        super(RNN, self).__init__()

        self.embedding_dim = 64
        self.hidden_dim = 128

        # A simple lookup table that stores embeddings of a fixed dictionary and size.
        # This module is often used to store word embeddings and retrieve them using indices. The input
        # to the module is a list of indices, and the output is the corresponding word embeddings.
        # With this layer we map the indices from the vocabulary to embeddings.
        # embedding_dim defines the size of the embedding vectors.
        # Embedding vectors are learnable.
        self.embedding = nn.Embedding(vocab_size, self.embedding_dim)

        # The RNN takes word embeddings or features as inputs, and outputs hidden states with
        # dimensionality hidden_dim.
        self.rnn = nn.RNN(self.embedding_dim, self.hidden_dim)

        # The linear layer that maps from hidden state space to tag space.
        self.fc = nn.Linear(self.hidden_dim, output_dim)


    def forward(self, text, text_len):

        # text Shape: [sequenze_lenght, batch size]
        embeds = self.embedding(text)

        # embeds Shape: [sequenze_lenght, batch size, embedding_dim]
        rnn_feats, final_hidden_states = self.rnn(embeds)

        # Select the outpuf of the rnn for the last input (e.g. word) in the sequence.
        # rnn_feats Shape: [sequenze_lenght, batch size, hidden_dim]
        out = self.fc(rnn_feats[-1])

        # out Shape: [batch size, 2]
        return out
```

What are the benefits of RNNs in general

✓ Handel variable sequence lengths (step by step):

	I am a sentence	I am a longer sentence
Sequence length:	4	5

✓ Capture differences in sequence order:

The food was good, not bad at all.

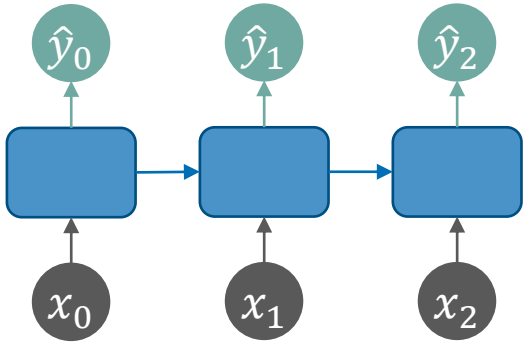
vs.

The food was bad, not good at all.

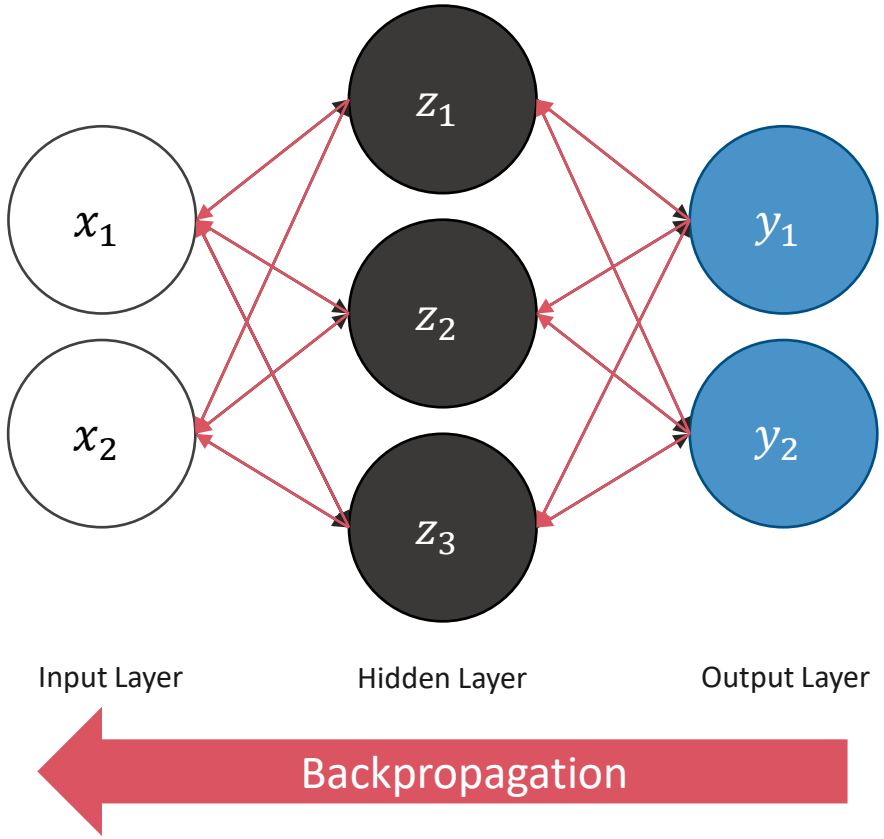
✓Model long-term dependencies:

I grew up in Germany when and I still speak fluent ___

✓Share weighs across the sequence:



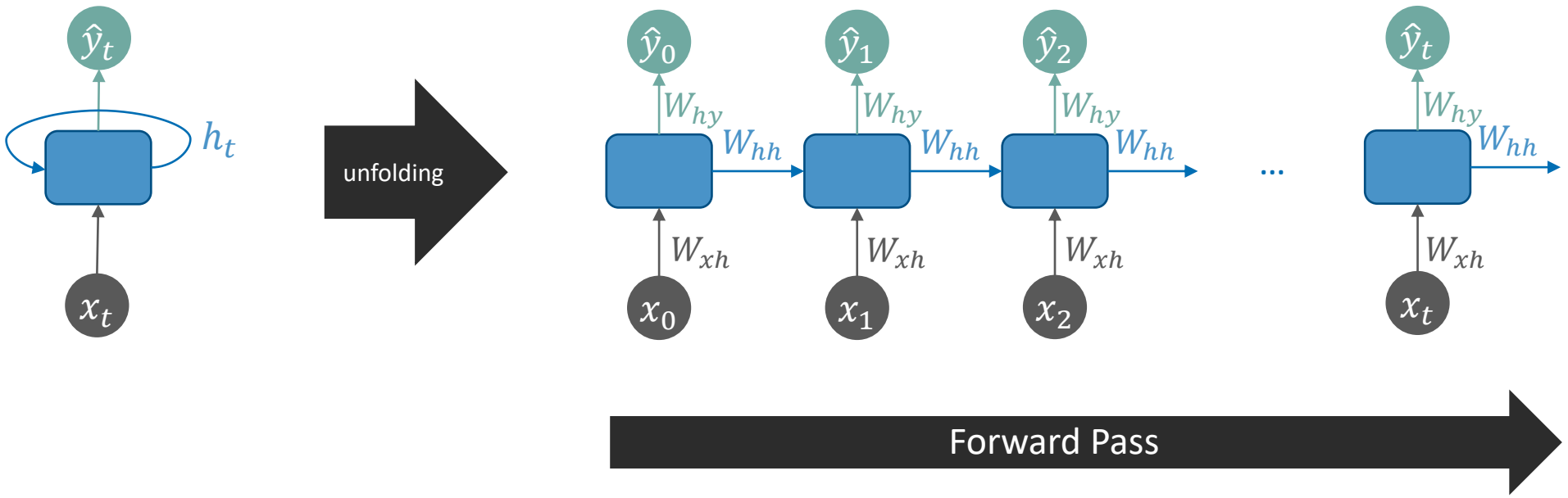
Backpropagation in Feed Forward Models: Quick Recap



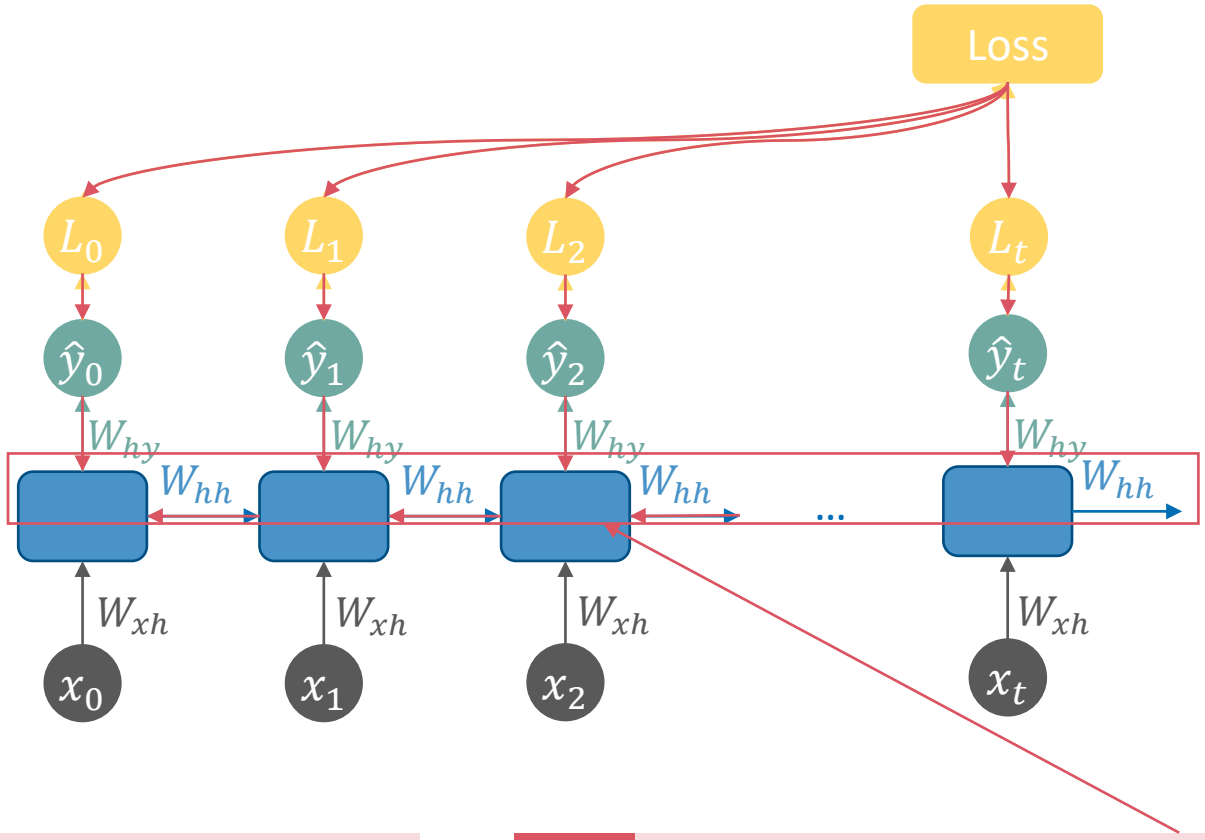
Backpropagation:

1. Calculate the gradient (derivative) of the loss with respect to the weights of the network
2. Update the weights to minimize the loss

Backpropagation in RNNS: Unfolding the model



RNNs: Backpropagation through Time (BPTT)



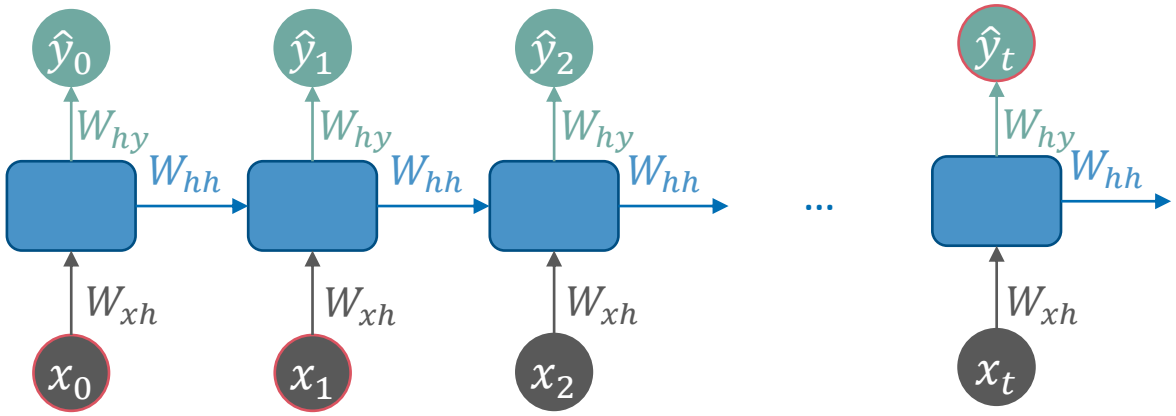
! **Backpropagation through Time:**
Unfolds the RNN "in time" and backpropagates through the unfolded model

! Computing the gradient with respect to h_0 involves many factors of W_{hh} and its computation of the gradient is repeated multiple times

- **Vanishing gradients** when many values < 1
- **Exploding gradients** when many values > 1

Problem of Vanishing Gradients: Long-term Dependencies in the Sequence

- Multiplying small weights of the hidden state together leads to smaller and smaller values
- This is a problem if the model make a mistake early in the sequence
- When we compute the gradient with this small values the gradients become small (they are vanishing), and it becomes hard for the model to learn from its early mistake
- The model will be biased for short-term dependencies instead of long-term dependencies

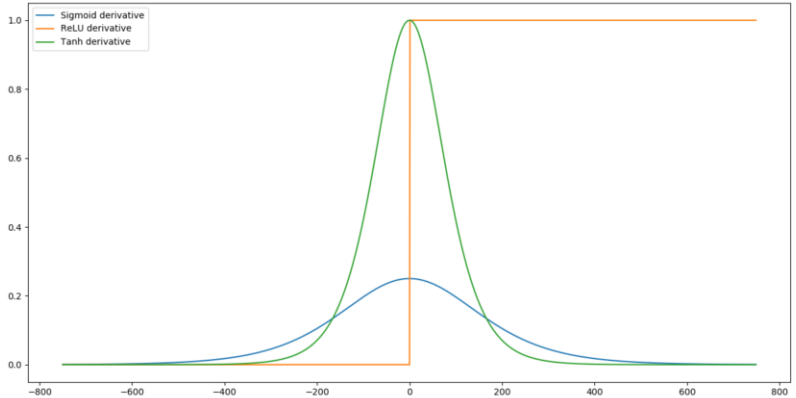


I grew up in Germany when and I still speak fluent ___

Exploding Gradients and Vanishing Gradients Solutions

Solution for Vanishing Gradients

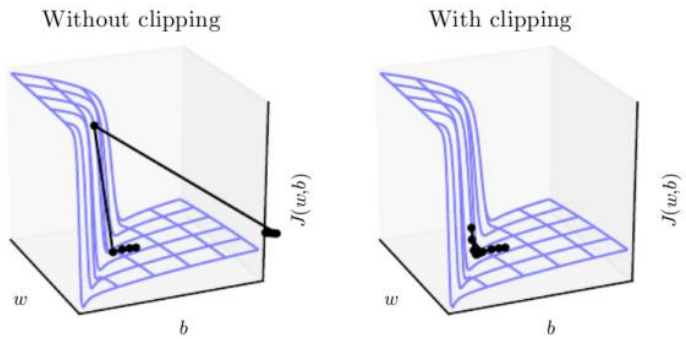
- Change Activation Function: Use ReLu
  - Prevents vanishing gradients, since its derivative is not saturating
- Parameter Initialization
  - ➢ Initialize weights to identity matrix
  - ➢ Initialize biases to zero
  - ➢ This prevents the weights from shrinking to zero
- **Gated Cells: LSTM, GRUs, …**



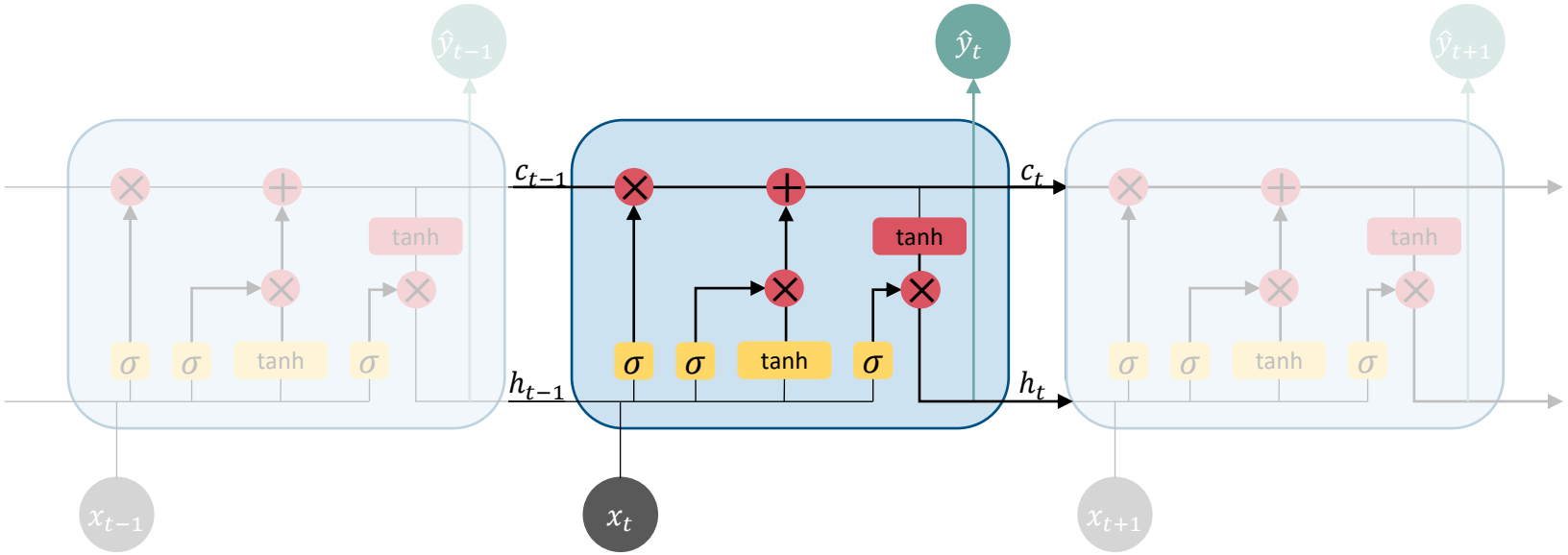
Derivatives of some Activation Functions. Image from [Link]

Solution for Exploding Gradients

- Gradient Clipping
  - Clip to large gradients to a maximal value
- **Gated Cells: LSTM, GRUs, …**

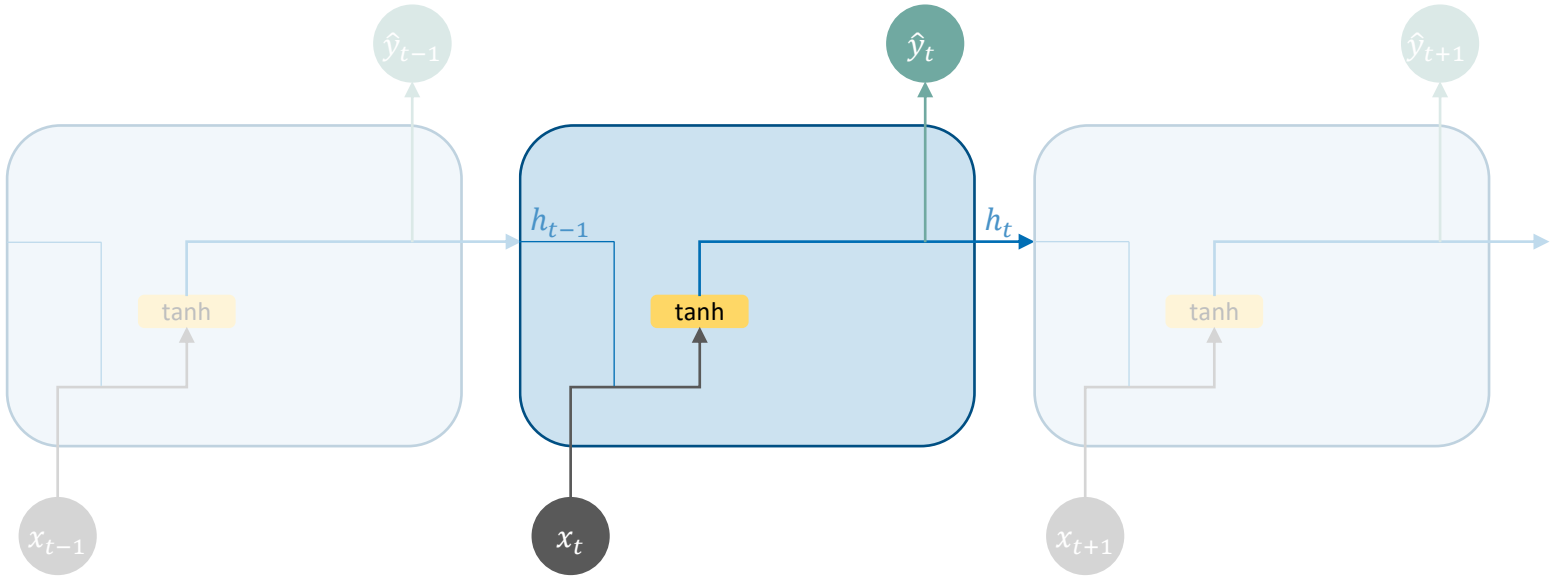


Effect of Gradient Clipping for RNNs [Pascanu2013]

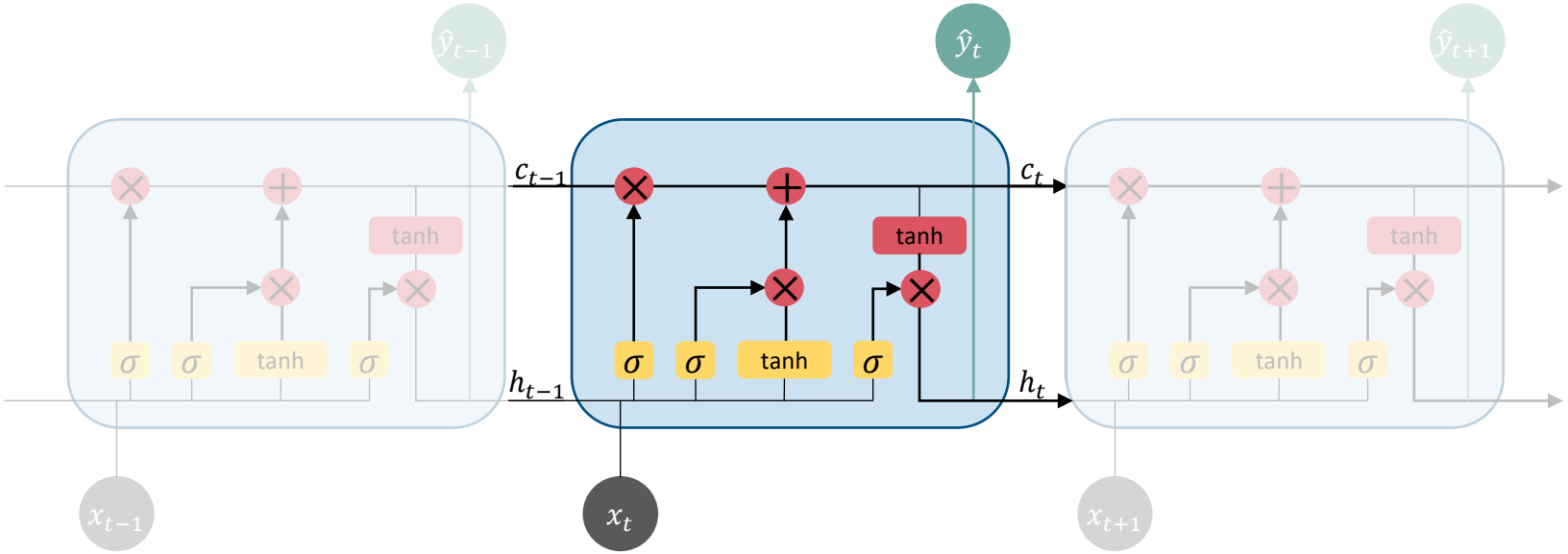


L11.2 Long short-term Memory (LSTM)

More detailed RNN Cell



LSTM (Long short-term Memory) Cell



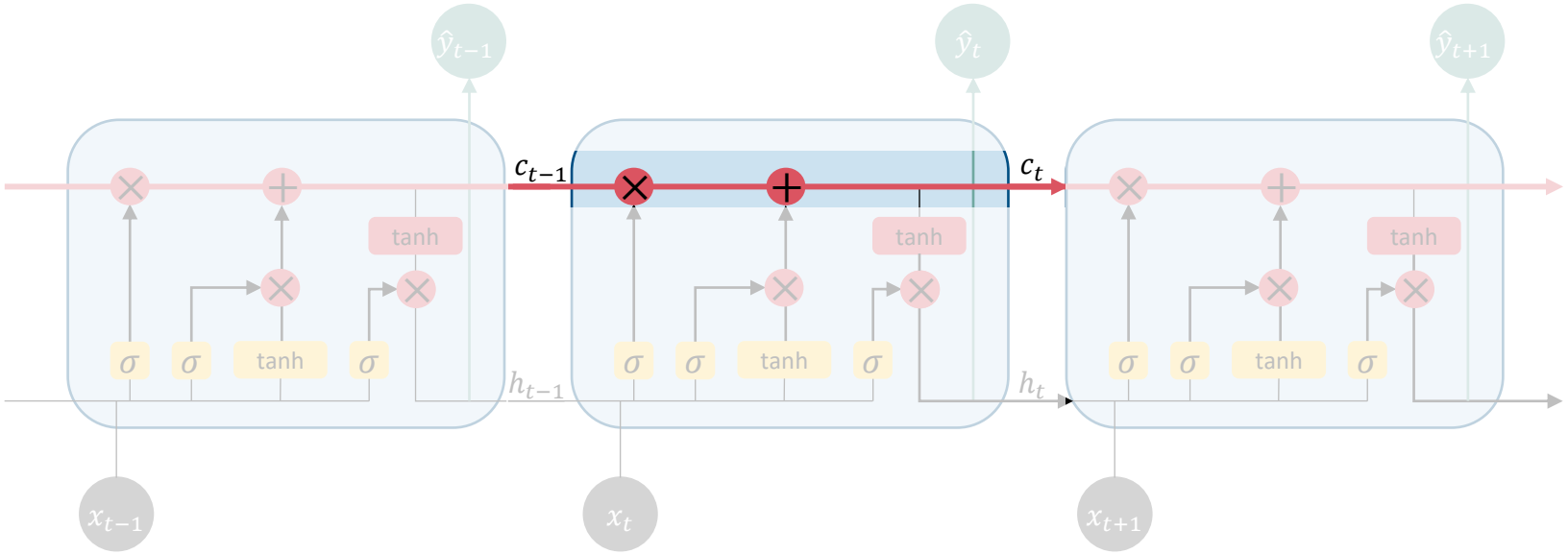
!

LSTM:
A complex recurrent cell with gated cells to control what information is pass trough the cell

- Capable of learning long-term dependencies

```
torch.nn.LSTM(*args, **kwargs)
```

LSTM: Cell State (Core Idea)



Cell State:
Runs straight down the entire chain, with only some minor linear interactions.

- Easy for information to flow along it unchanged
- Gates are use by LSTM to carefully remove or add information to the cell state

LSTM: Gates

Gate:

Gates:
Gates are a way to optionally let information through.

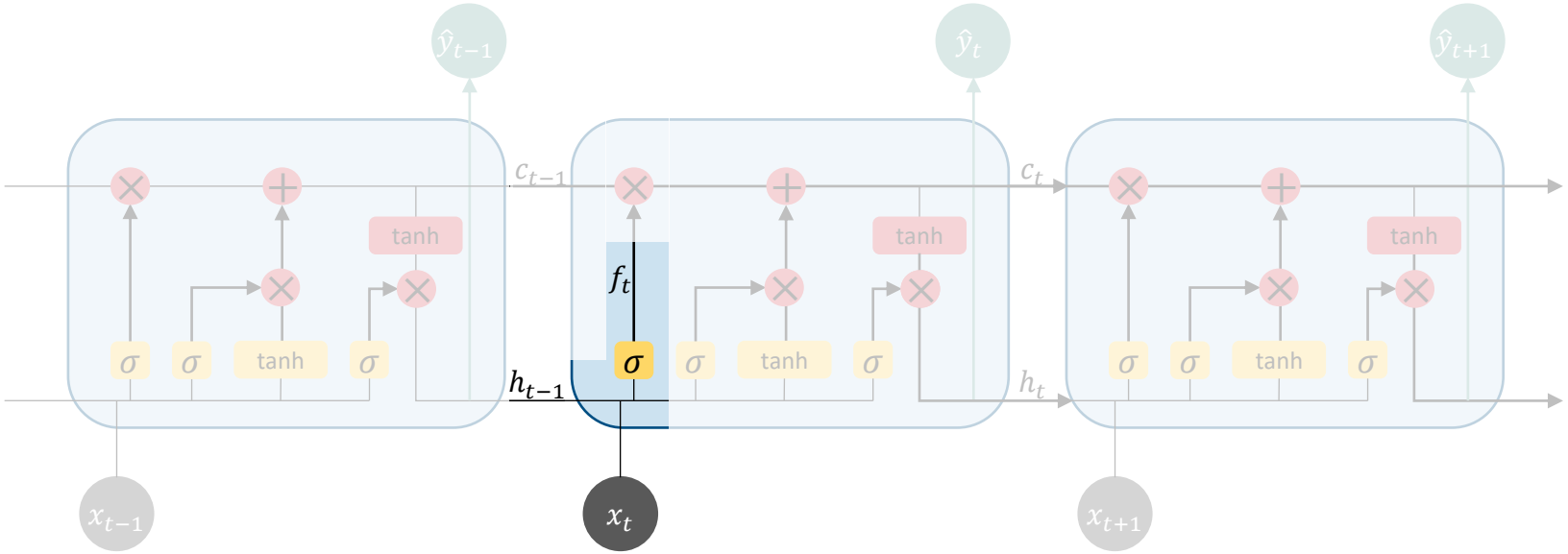
- An LSTM has three of these gates, to protect and control the cell state

Gates: Sigmoid neural net layer + pointwise multiplication

- Sigmoid Activation: Outputs numbers between zero and one to control how much information to let through
- ➢ Zero: let nothing through, One: let everything through

LSTM: Forget

1. Forget 2. Store 3. Update 4. Output

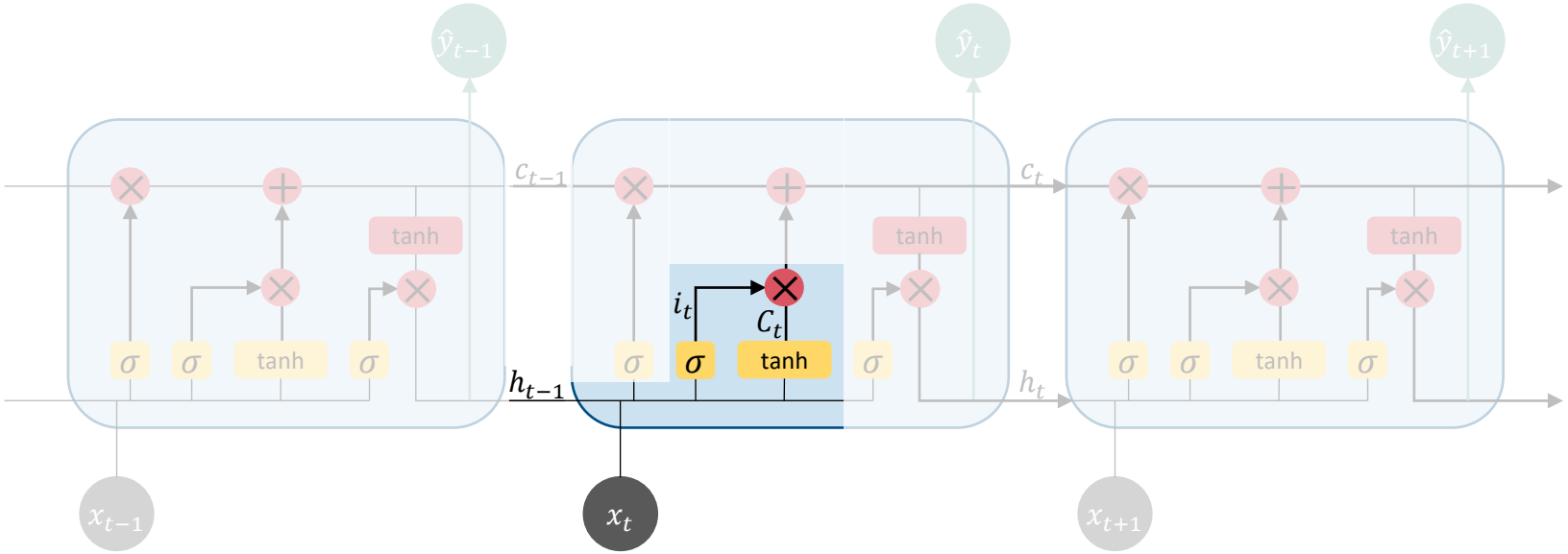


! **Forget:** Decide what information to forget from the cell state

$$\boldsymbol{f}_t = \sigma(\boldsymbol{W}_f \cdot [\boldsymbol{h}_{t-1}, \boldsymbol{x}_t] + \boldsymbol{b}_F)$$

LSTM: Store

1. Forget **2. Store** 3. Update 4. Output

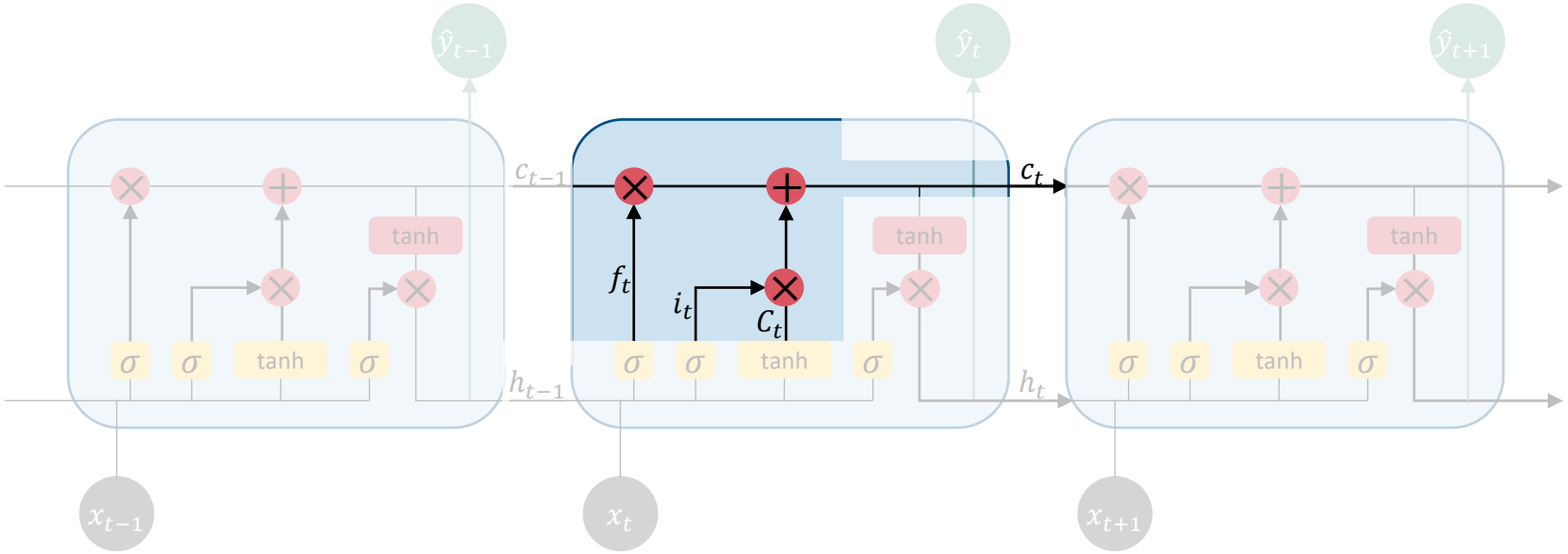


Store: Create new information for the cell state ($\widetilde{\boldsymbol{C}}_t$).
Decide what new information to store in the cell state ($\boldsymbol{i}_t$).

$$\boldsymbol{i}_t = \sigma(\boldsymbol{W}_i \cdot [\boldsymbol{h}_{t-1}, \boldsymbol{x}_t] + \boldsymbol{b}_i)$$
$$\widetilde{\boldsymbol{C}}_t = tanh(\boldsymbol{W}_C \cdot [\boldsymbol{h}_{t-1}, \boldsymbol{x}_t] + \boldsymbol{b}_C)$$

LSTM: Update

1. Forget 2. Store **3. Update** 4. Output



!

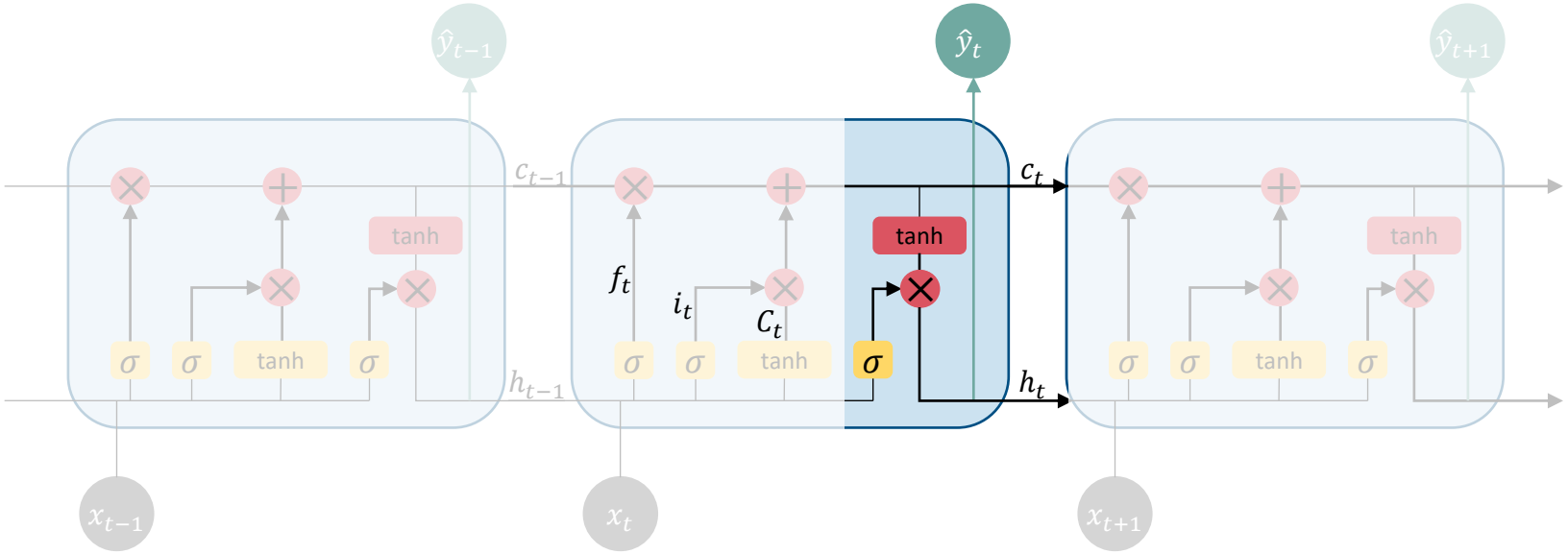
Update: Update the old cell state $\boldsymbol{C}_t$

- Multiply the old state by $\boldsymbol{f}_t$, forgetting the information the cell decided to forget earlier
- Add $\boldsymbol{i}_t * \widetilde{\boldsymbol{C}}_t$ to the old state, storing the information the cell decided to store

$$\boldsymbol{C}_t = \boldsymbol{f}_t * \boldsymbol{C}_{t-1} + \boldsymbol{i}_t * \widetilde{\boldsymbol{C}}_t$$

LSTM: Output

1. Forget 2. Store 3. Update **4. Output**



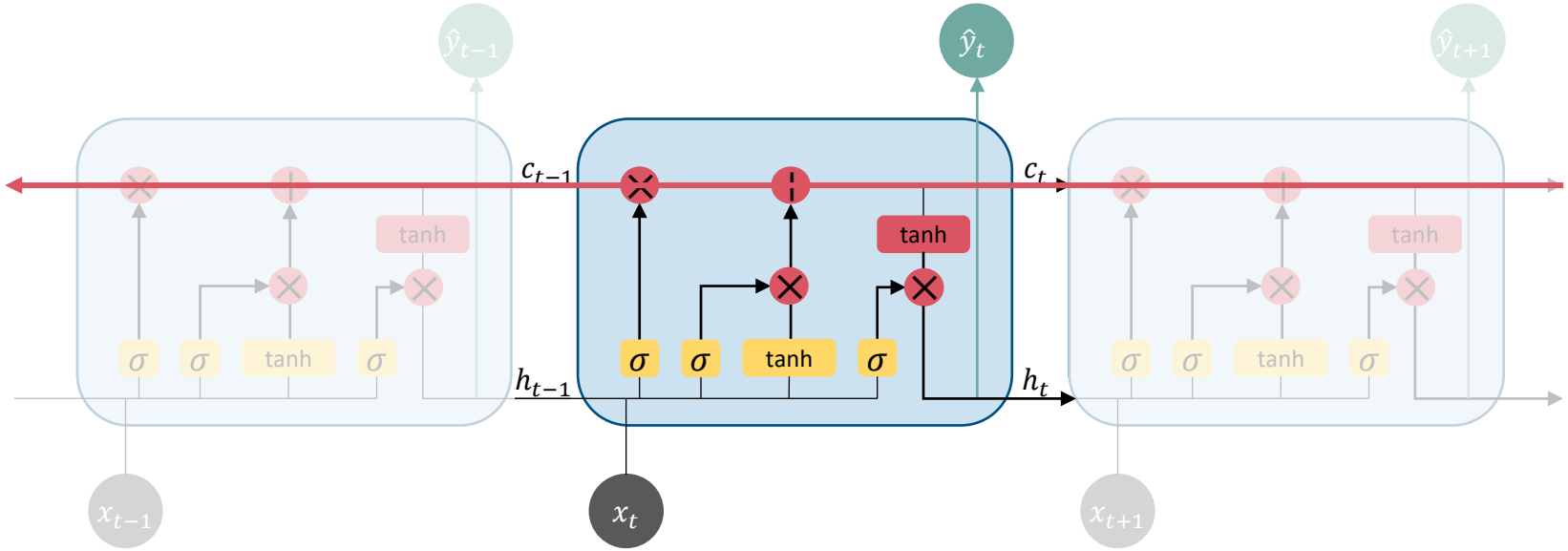
!

Output: decide what to output. Output is a filtered (gated) version of the cell state

1. Activation Function (tanh, values to be between −1 and 1)
2. Sigmoid layer: Decides what to output

$$\boldsymbol{o}_t = \sigma(\boldsymbol{W}_o \cdot [\boldsymbol{h}_{t-1}, \boldsymbol{x}_t] + \boldsymbol{b}_o)$$
$$\boldsymbol{h}_t = \boldsymbol{o}_t * tanh(\boldsymbol{C}_t)$$
$$\hat{\boldsymbol{y}}_t = \boldsymbol{h}_t$$

LSTM: Uninterrupted Gradient Flow



!

Uninterrupted Gradient Flow:
Because the cell state is only modified by minor linear interactions the gradients can flow uninterrupted across this “highway”. This **mitigates** the **Vanishing Gradients** Problem and allows the LSTM to **capture Long-term Dependencies**

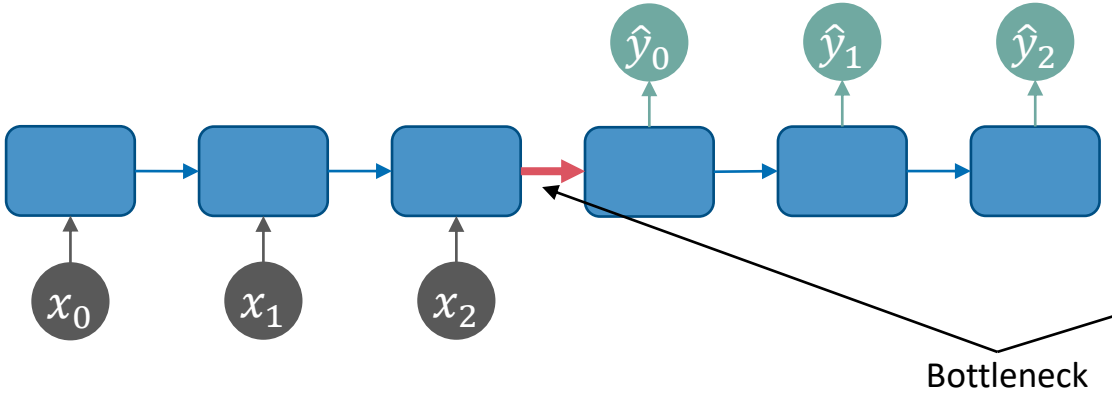
LSTMs: Key Concepts

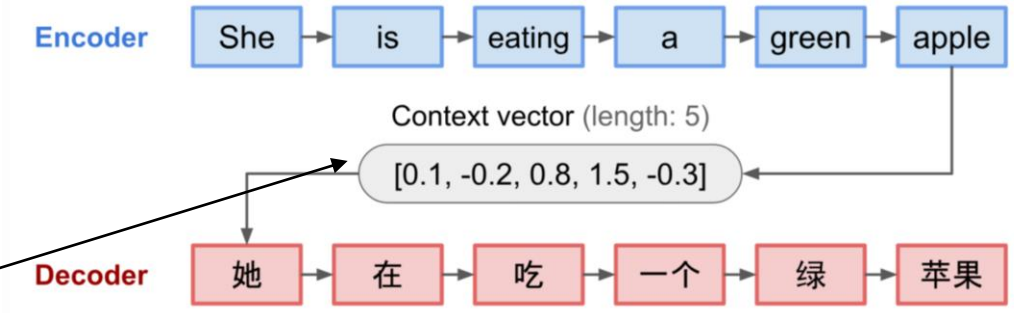
1. **Cell State:** Use a separate state with minor linear interactions for uninterrupted gradient flow
   - Mitigates Vanishing Gradient Problem
   - Allows the LSTM to capture Long-Term Dependencies (to a degree)
2. **Gates:** Use gate to control the information flow
   1. **Forget**: Get rid of irrelevant information
   2. **Store**: Store relevant information from the current input
   3. **Update**: Selectively update the cell state
   4. **Output**: Return a filtered output of the cell state

LSTM Example: Code

```python
import torch.nn as nn

# We define an LSTM class that is a subclass of nn.Module.
class LSTM(nn.Module):
    def __init__(self, vocab_size, output_dim):
        super(LSTM, self).__init__()

        self.embedding_dim = 64
        self.hidden_dim = 128

        # A simple lookup table that stores embeddings of a fixed dictionary and size.
        # This module is often used to store word embeddings and retrieve them using indices. The input
        # to the module is a list of indices, and the output is the corresponding word embeddings.
        # With this layer we map the indices from the vocabulary to embeddings.
        # embedding_dim defines the size of the embedding vectors.
        # Embedding vectors are learnable.
        self.embedding = nn.Embedding(vocab_size, self.embedding_dim)

        # The LSTM takes word embeddings or features as inputs, and outputs hidden states with
        # dimensionality hidden_dim.
        self.lstm = nn.LSTM(self.embedding_dim, self.hidden_dim)

        # The linear layer that maps from hidden state space to tag space.
        self.fc = nn.Linear(self.hidden_dim, output_dim)


    def forward(self, text, text_len):

        # text Shape: [sequenze_lenght, batch size]
        embeds = self.embedding(text)

        # embeds Shape: [batch size, sequenze_lenght, embedding_dim]
        lstm_feats, (final_hidden_states, final_cell_states) = self.lstm(embeds)

        # Select the outpuf of the rnn for the last input (e.g. word) in the sequence.
        # lstm_feats Shape: [batch size, sequenze_lenght, hidden_dim]
        out = self.fc(lstm_feats[-1])

        # out Shape: [batch size, 2]
        return out
```

RNN Problems (Again)



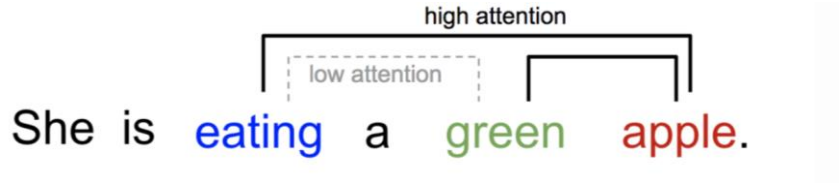


Bottleneck of a LSTM or GRU for Machine Translation (seq2seq). Image from [Link]

- Encoding bottleneck (e.g., Machine Translation)
- Slow:
  - Sequential processing (step-by-step): inefficient one modern GPUs (no parallelization within a squeeze possible)
- BBPT is expensive because it is calculated over multiple steps
- Limited long-term memory capacity
  - Better then basic RNNs but there is still room for improvement

L11.3 Attention and Transformers

Attention



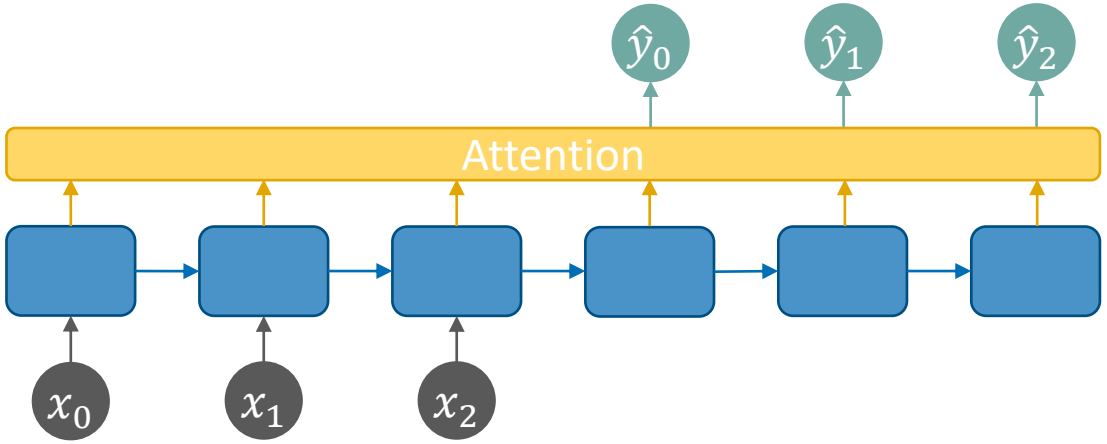
We can look at all the different words at the same time and learn to “pay attention“ to the correct ones depending on the task. Image from [Link]

Visual Attention woman is throwing a frisbee in a park [Xu2015]

! **Attention:** Mechanism that give the model the ability to focus on parts of the input or certain features.

- Is a ”memory”, gained from attending at multiple inputs through time
- Biologically Inspired: Mimics cognitive attention

Attention (Early Steps)



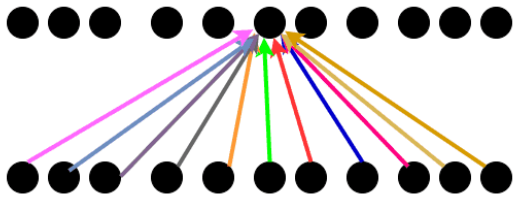
Use Attention to relieve the encoder from the burden of having to encode all information into a fixed-length vector

Solves two Problems:
- Encoding Bottleneck
- limited long term memory to memorize long sentences

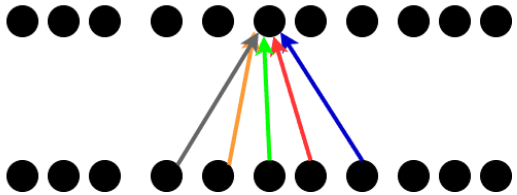
- Bahdanau first proposed attention in 2015 for NLP (Natural Language Processing) [Bhandanau2015]
- Since then, various type of attentions have been proposed.

Global vs. Local Attention

Global attention



Local attention



Images from [Link]

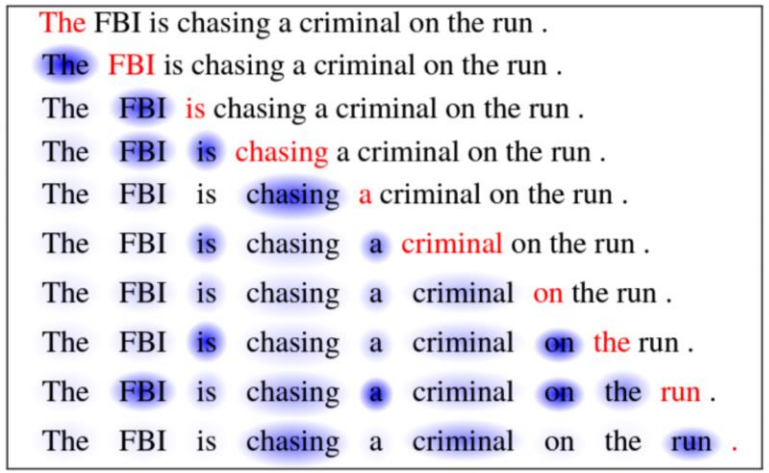
Global Attention: Computing Attention over the entire input

- Can be very costly (often quadratic complexity)

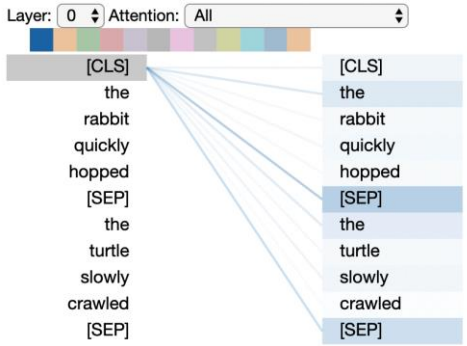
Local Attention: Computing Attention over a subset of the input

- More efficient

Self-Attention



[Cheng2016]

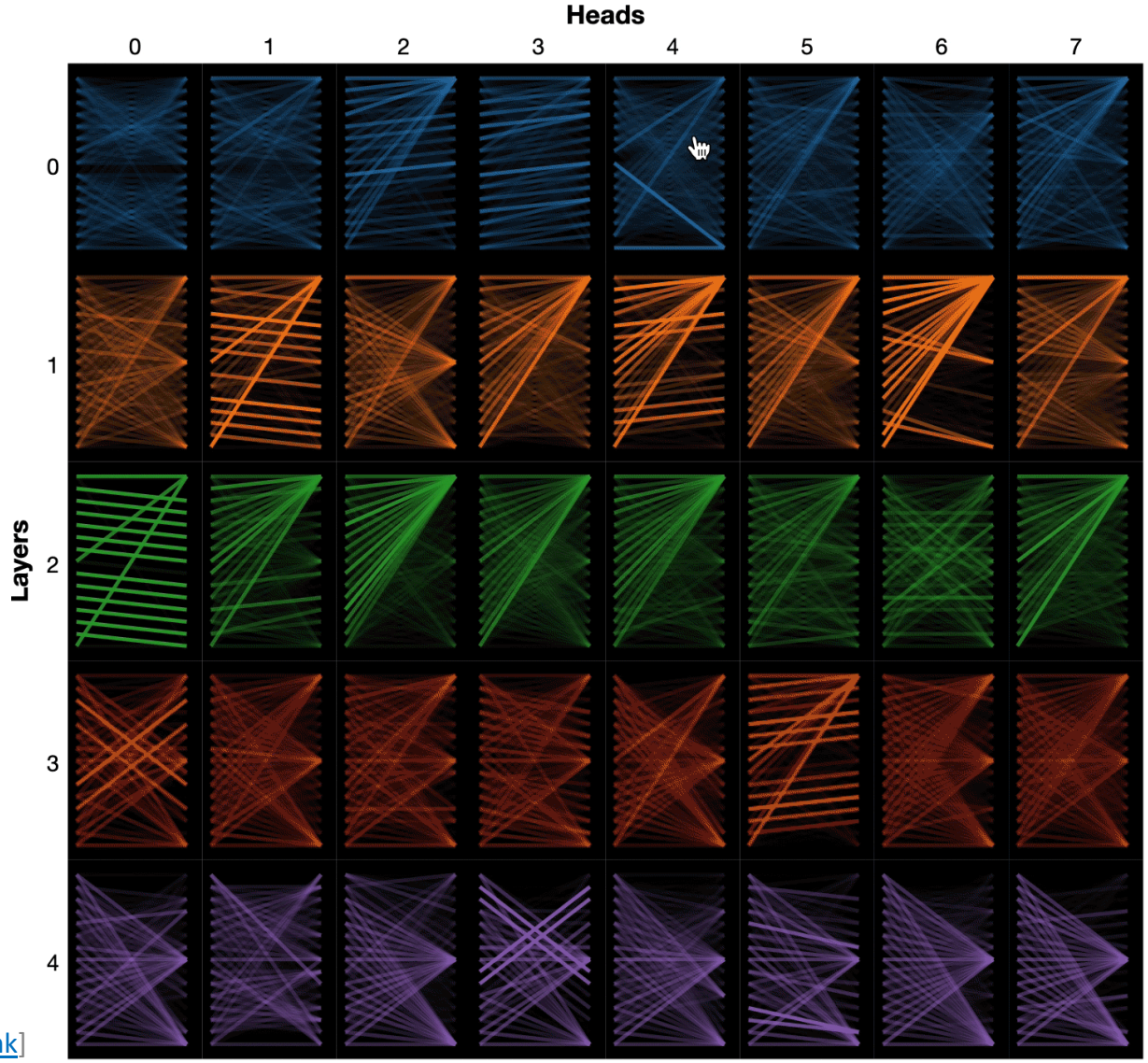


Gif from [Link]

! **Self-Attention**: Instead of looking for an input-output association/alignment, look for scores between the elements of a sequence itself.

Advantages of Attention

- ✓ Mitigates the Bottleneck Problem
- ✓ Allows for long term memory
- ✓ Eliminates the Vanishing Gradient Problem
  - ➢ Provides direct connections between encoder and decoder
  - ➢ Conceptually similar to skip connections in a CNN
- ✓ **Explainability**
  - ✓ By inspecting the distribution of attention weights, we can gain insights into the behavior of the model and understand its limitations



Inspecting attention weights. Gif from [Link]

Transformer: Key Concepts

Transformers use Attention “everywhere” in the encoder and decoder.

- The original paper is called “Attention is all you need”

Parallelized Encoding

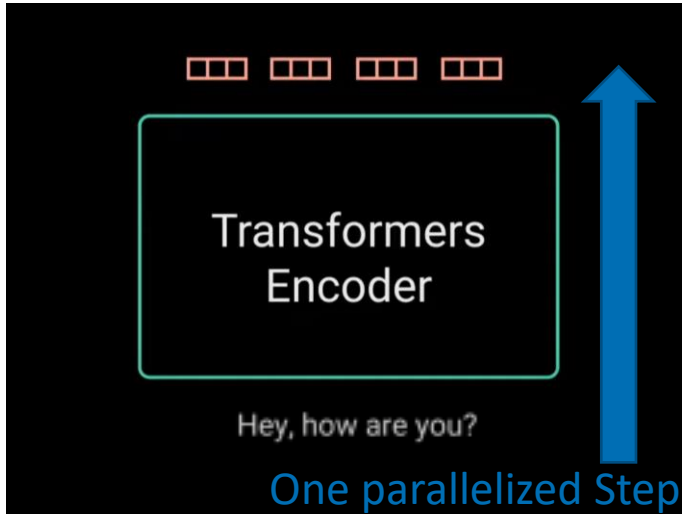
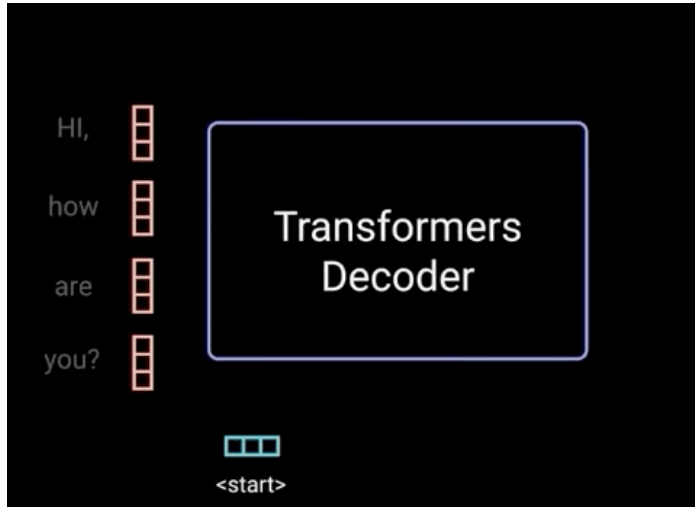


Image from [Link]

By using Attention the Transformers encodes a sequence in parallel.

Steps-by-Step Decoding

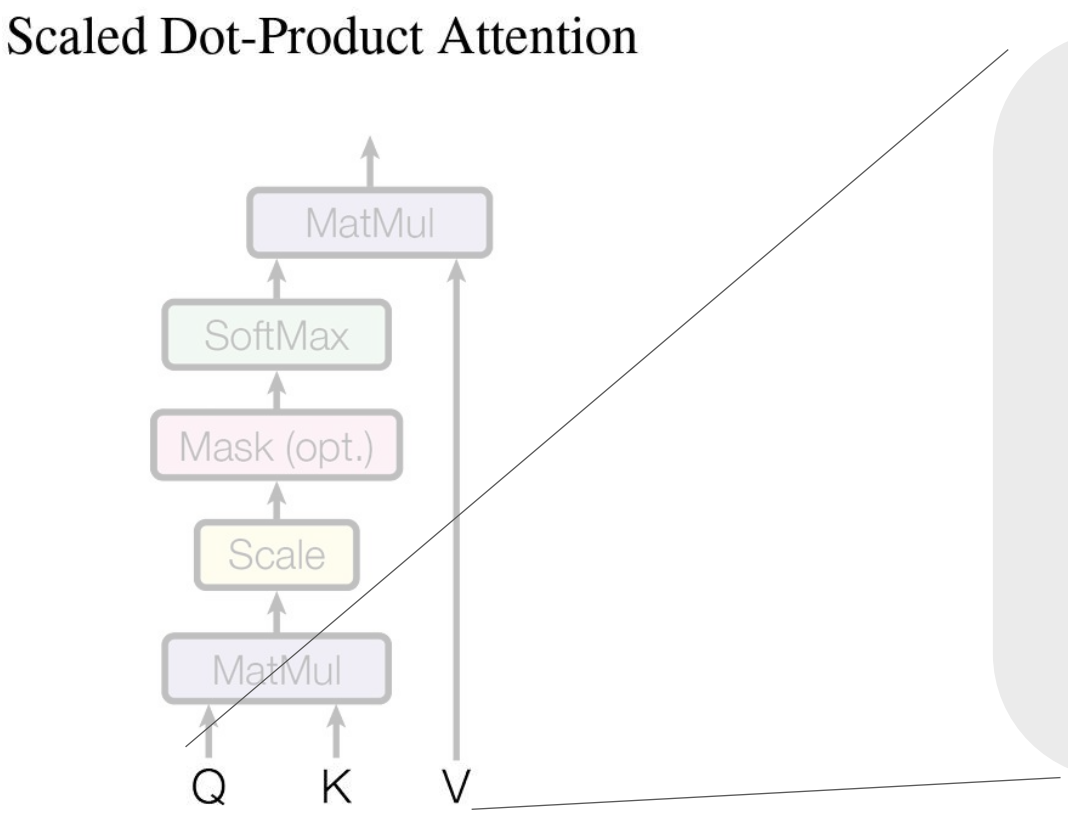


Gif from [Link]

Decoding a sequence (e.g., for translation) is still done step-by-step

- Each encoded word contains useful information and is proceeds by the decoder for each step

Transformer: Query, Key, Value



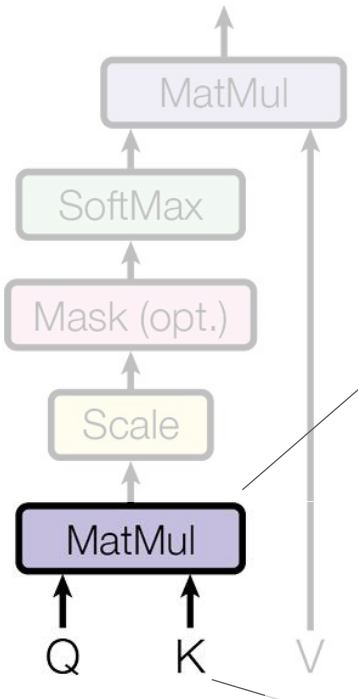
- Query/Key/Value concept is similar to retrieval systems.

- E.g., when you search for videos on Youtube:
  - Query: Text in the search bar
  - Key: Video title, description, etc.
  - Value: Best matched videos

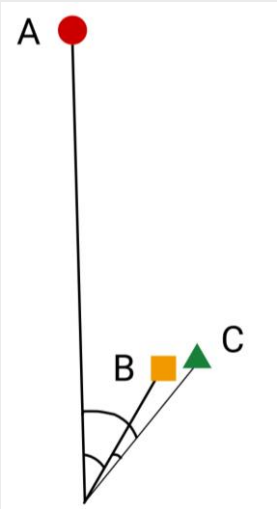
- **Query** (Q): Things the model want to attend to (or want to calculate similarity for)
- **Key** (K): Things the model can attend to (or can calculate similarity for)
- **Value** (V): The actual Things the model attends to

Transformer: Scaled Dot-Product Attention

Scaled Dot-Product Attention



1. **MatMul $Q \cdot K^T$**: Score Matrix
   - Dot product is proportional to the cosine and length differences between two vectors vectors
   - ➢ With the Dot product we can calculate similarity between two vectors
   - ➢ And between multiple "vectors" in a matrix
   - ➢ We check how similar the queries and keys are (more similar = more focus)



A and C are more similar, from [Link]

Hello	1.0	1.5	0.6	0.2
how	0.1	2.0	1.0	0.3
are	1.0	0.7	0.9	1.3

Querys

·

Hello	how	are
0.5	1.0	1.3
0.9	0.7	0.4
1.1	0.7	3.0
2.0	1.5	0.6

Transposed Keys

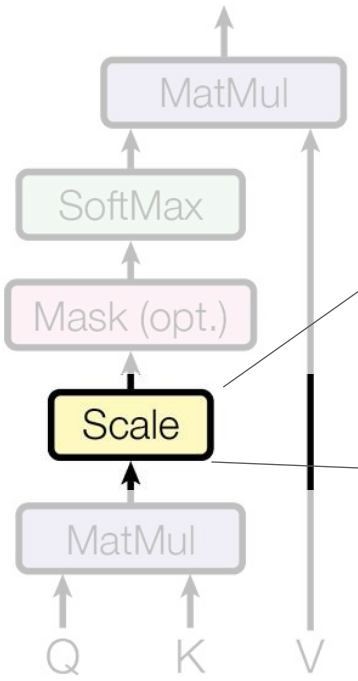
=

	Hello	how	are
Hello	2.9	2.7	3.8
how	3.5	2.6	4.1
are	4.7	4.0	5.0

Score Matrix

Transformer: Scaled Dot-Product Attention

Scaled Dot-Product Attention



2. Scale: Scaled down scores with $\sqrt{d_k}$ (d_k= key dimension)

- This mitigates exploding gradients

	Hello	how	are
Hello	2.9	2.7	3.8
how	3.5	2.6	4.1
are	4.7	4.0	5.0

Score Matrix

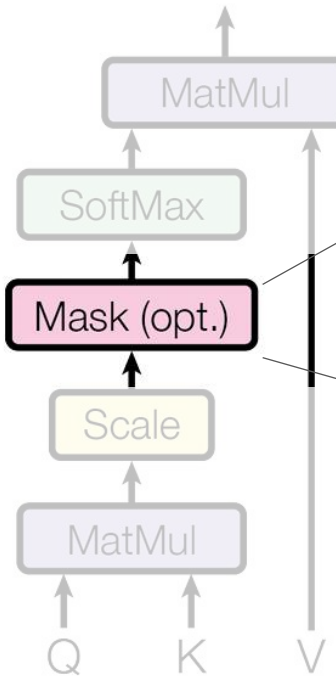
$\backslash\sqrt{d_k}$ =

	Hello	how	are
Hello	1.5	1.4	1.9
how	1.8	1.3	2.0
are	2.4	2.0	2.5

Scaled Score Matrix

Transformer: Scaled Dot-Product Attention

Scaled Dot-Product Attention



2.1 (Optional) Mask: Only for the decoder to mask future not predicted words.

- We do not want to attend to words in the future

	Hello	how	are
Hello	1.5	1.4	1.9
how	1.8	1.3	2.0
are	2.4	2.0	2.5

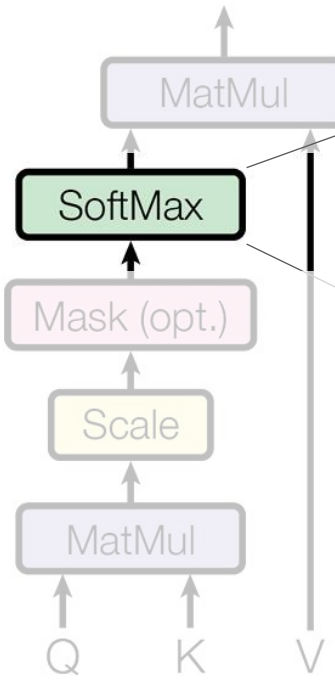
Unmasked

	Hello	how	are
Hello	1.5	-∞	-∞
how	1.8	1.3	-∞
are	2.4	2.0	2.5

Masked

Transformer: Scaled Dot-Product Attention

Scaled Dot-Product Attention



3. SoftMax leads to Attention Matrix:

- Apply SoftMax Activation Function elementwise on the matrix
- Probability values between 0 (less important) and 1 (more important)

	Hello	how	are
Hello	0.1	0.1	0.1
how	0.1	0.1	0.1
are	0.2	0.1	0.2

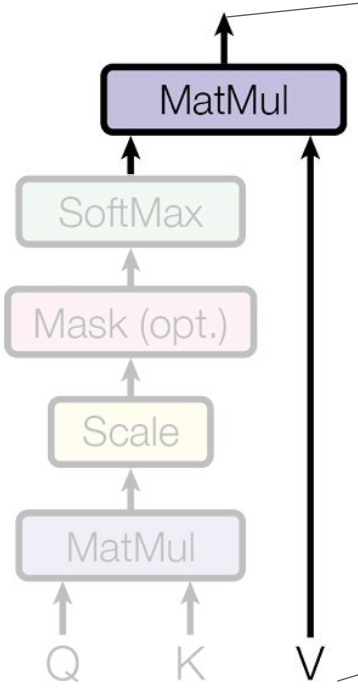
SoftMax Unmasked

	Hello	how	are
Hello	0.1	0	0
how	0.1	0.1	0
are	0.2	0.1	0.2

SoftMax Masked

Transformer: Scaled Dot-Product Attention

Scaled Dot-Product Attention



4. Multiply Attention Matrix with Values to **attend to the Values** („word features")

SoftMax

	Hello	how	are
Hello	0.1	0.1	0.1
how	0.1	0.1	0.1
are	0.2	0.1	0.2

·

Values

Hello	1.5	0.7	1.5	2.1
how	1.3	1.0	2.4	0.9
are	0.3	0.7	0.5	2.0

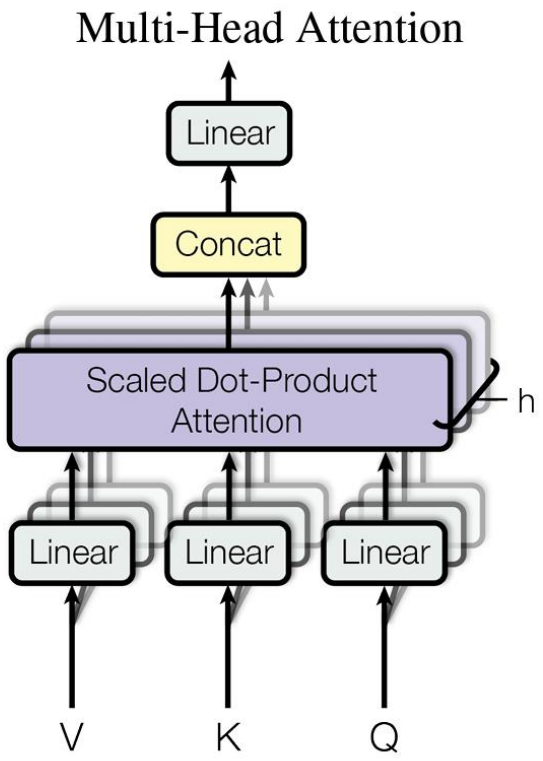
=

Outputs

Hello	0.2	0.2	0.3	0.4
how	0.3	0.2	0.3	0.5
are	0.5	0.4	0.6	0.9

$$Attention(\boldsymbol{Q},\boldsymbol{K},\boldsymbol{V}) = Softmax\left(\frac{\boldsymbol{Q}\boldsymbol{K}^T}{\sqrt{d_k}}\right)\boldsymbol{V}$$

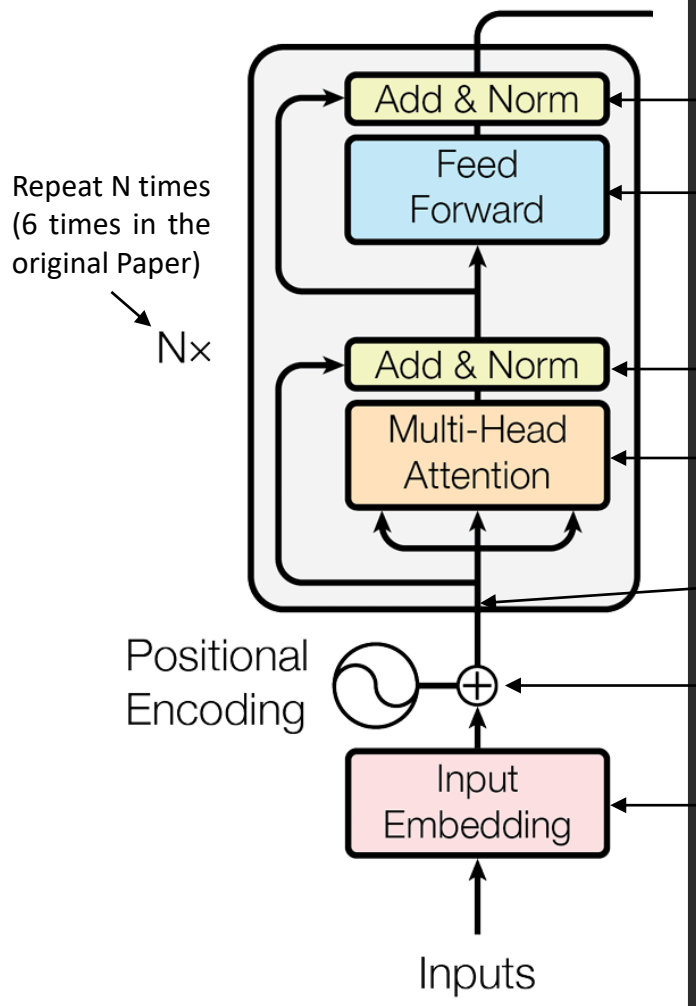
Transformer: Multi-Head Attention



Multi-Head Attention: Consist of multiple (h) Attention Heads

- Use linear layers to calculate Values Keys and Queries h-times
- Apply the Attention h-times
- Each head learns something different
- Concatenate the results of each head to one vector und fuse them via a linear layer
- Model gets more representation power

Transformer: Encoder



7. **Normalization Layer** and a **Residual Connection**

6. **Linear Layer** to project the output of self-attention in a higher dimensional space
   - Solves bad initializations and rank collapse (weights around zero without meaning)

5. **Normalization Layer** and a **Residual Connection**

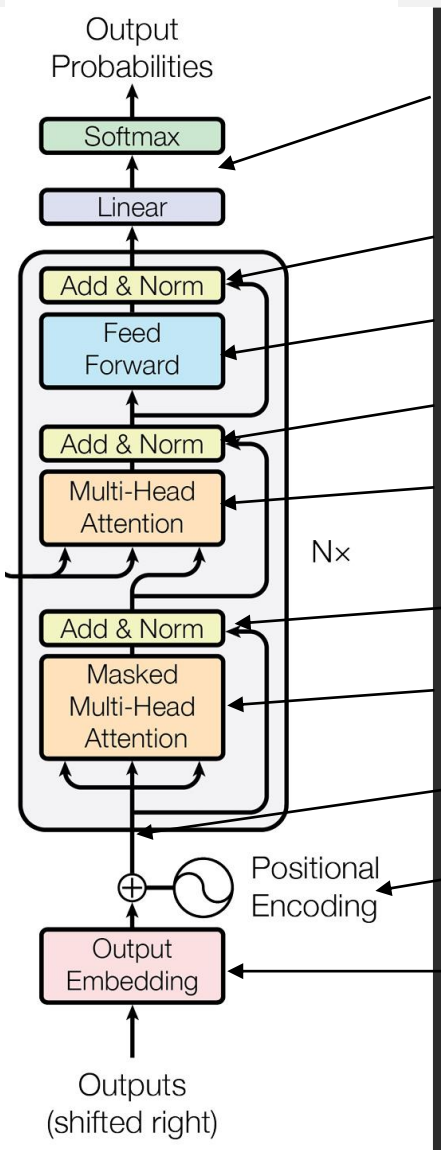
4. **Multi-head Self-Attention** layer to find correlations between each word

3. Word vectors are passed to the first **encoder block**

2. **Positional Encodings** are applied to each embedding resulting in word vectors with positional information

1. **Word embeddings**

Transformer: Decoder



10. **Softmax Linear Layer** to output probabilities to predict the next word in the output sentence
- assign a probability to each word in the French language and we simply keep the one with the highest score

9. **Normalization Layer** and a **Residual Connection**

8. **Linear Layer** to project the output of self-attention in a higher dimensional space
- Solves bad initializations and rank collapse (weights around zero without meaning)

7. **Normalization Layer** and a **Residual Connection**

6. **Encoder-Decoder Multi-head Self-Attention** layer to associates the input sentence from the decoder with the corresponding output word

5. **Normalization Layer** and a **Residual Connection**

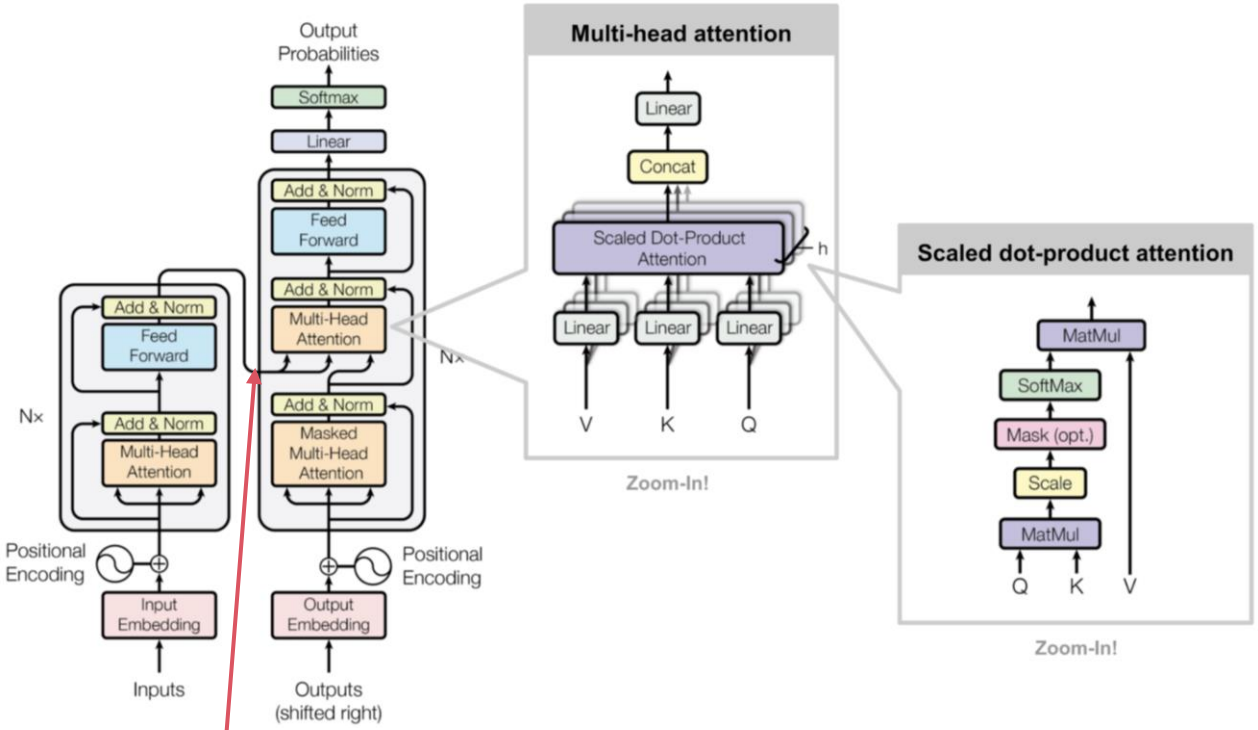
4. **Mask Multi-head Self-Attention** layer to find correlations between each (translated) word, but no future words

3. Word vectors are passed to the first **encoder block**

2. **Positional Encodings** are applied to each embedding resulting in word vectors with positional information

1. **Word embeddings**

Transformer. Full Model Architecture



Encoder-decoder attention associates the input sentence with the corresponding output word

E.g., It is possible for the model to determine how each English word is related to French words. This is where the mapping between English and French takes place.

```
torch.nn.Transformer(*args, **kwargs)
```

Intuition of why Transformer work well

- ✓ **Meaning heavily depends on the context**:
  - Self-attention weights associate relationships between word representations
  - There is no notion of locality, as the model naturally makes global associations through attention

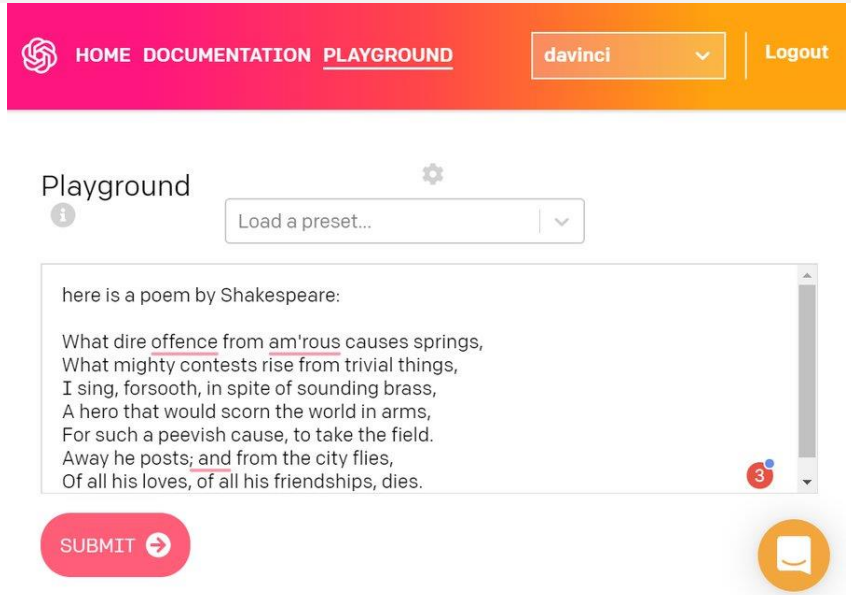
- ✓ **Hierarchical representations similar to a CNN**
  - With more layers, the model learns more abstract representations

- ✓ **Combination of high and low-level information with skip connections**

- ✓ **Robust learning of Attention weights with Multi-headed Attention**
  - Multiple Attention weights learn different ways to attend to representations in parallel
  - If some attention weights fail, other can still back them up

GPT1, GPT2, GPT3

- GPT-1, GPT-2, GPT-3 are huge pre-trained Transformer Models from OpenAI
  - GPT-1: **117 million parameters**
  - GPT-2: **1.5 billion parameters**
  - GPT-3: **175 billion parameters**
    - Human Brain 86 billion neurons and 85 billion nonneuronal cells
    - Roughly the same number of neurons as there are stars in the Milky Way
    - However, Brain has 10^{15} connections between neurons
  - Training Cost: **4.6-12 Million Dollar** for a single training run (cloud cost)
    - There are no official numbers given by OpenAI, these are estimates
  - GPT-4 will have **100 trillion** parameters (according to OpenAI)
- They Improve basic Transformers in various ways (more parameters, training strategies, model architecture, better datasets, ….)
- Text generated by GPT-3 can be so realistic that it is difficult to determine whether or not it was written by a human,
  - This has both benefits and risks
- GPT-3 is currently not public ally available and owned by Microsoft
- There are many demos of GPT-3 online: https://github.com/elyase/awesome-gpt3



Rewrite in clickbait style:
"A fox jumped over a lazy dog."
To:
"This Idiot Messed with the Wrong Fox!"

Rewrite in clickbait style:
"A fox jumped over a lazy dog."
To:
"This Fox Is So Hilarious It Can't Even Be Funny"

Rewrite in clickbait style:
"A fox jumped over a lazy dog."
To:
"Watch This Insanely Clever Fox Outwit a Dog 10 Times in a Row."

Further Reading

- Deep Learning Book: https://www.deeplearningbook.org/
- The newest Papers (there are way too many)
- YouTube Lectures of other Unis, e.g.:
  - MIT: https://www.youtube.com/watch?v=5tvmMX8r_OM&list=PLtBw6njQRU-rwp5__7C0oIVt26ZgjG9NI
  - Stanford: https://www.youtube.com/watch?v=vT1JzLTH4G4&list=PLC1qU-LWwrF64f4QKQT-Vg5Wr4qEE1Zxk
  - TUM: https://www.youtube.com/watch?v=QLOocPbztuc&list=PLQ8Y4kIIbzy_OaXv86lfbQwPHSomk2o2e
  - TUM (advanced): https://www.youtube.com/watch?v=Bt5O1HjT9cI&list=PLog3nOPCjKBkngkkF552-Hiwa5t_ZeDnh
  - Tübingen: https://www.youtube.com/watch?v=OCHbm88xUGU&list=PL05umP7R6ij3NTWIdtMbfvX7Z-4WEXRqD
  - ….

IFM
Institut für Fahrerassistenz
und vernetzte Mobilität

Hochschule
Kempten
University of Applied Sciences

www.hs-kempten.de/ifm