

RNNs and Transformers

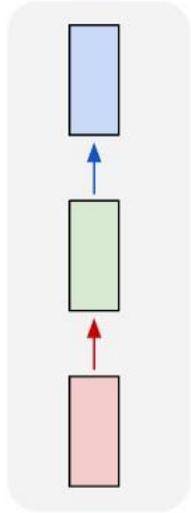
Recurrent Neural Networks

Processing Sequences

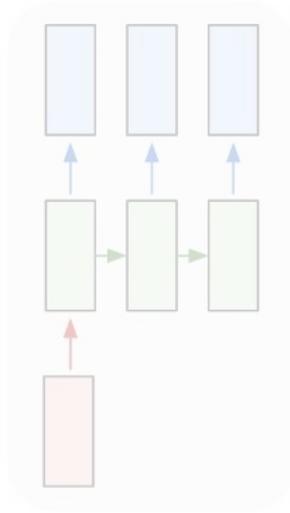
- Recurrent neural networks process sequence data
- Input/output can be sequences

RNNs are Flexible

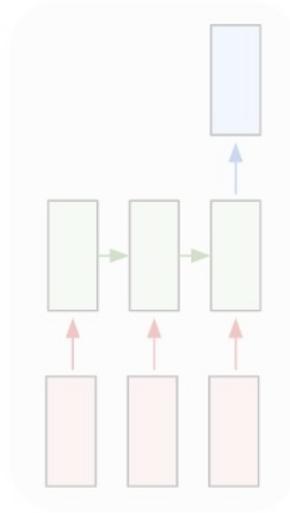
one to one



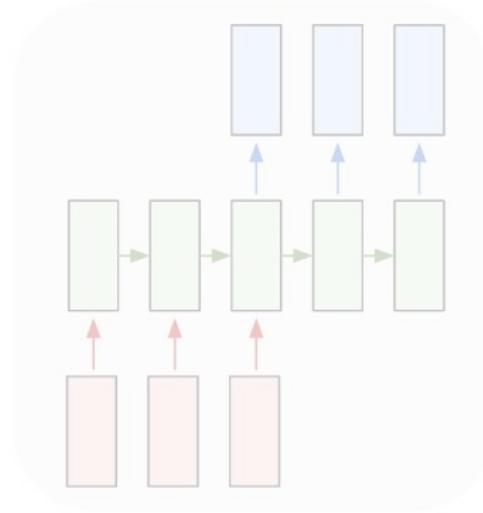
one to many



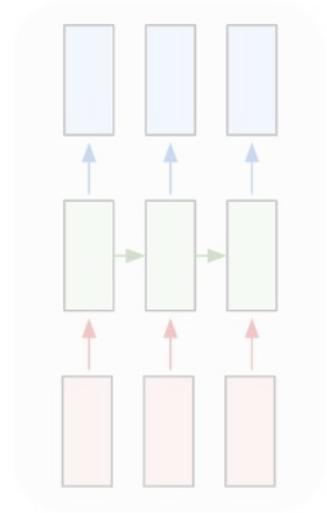
many to one



many to many



many to many

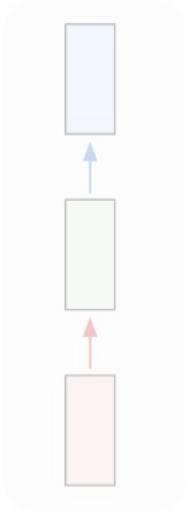


Classical neural networks for image classification

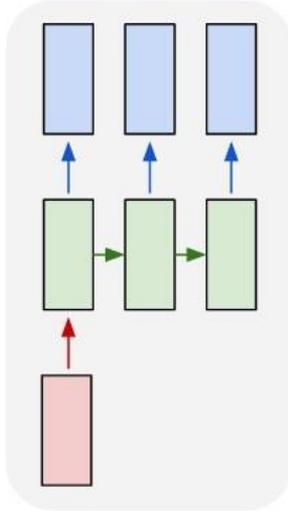
Source: <http://karpathy.github.io/2015/05/21/rnn-effectiveness/>

RNNs are Flexible

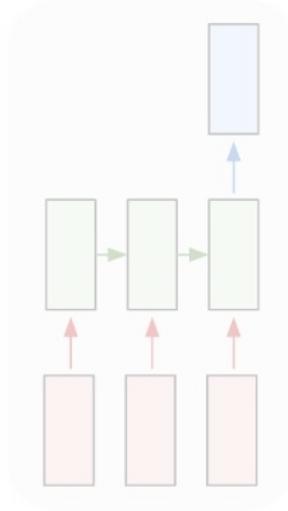
one to one



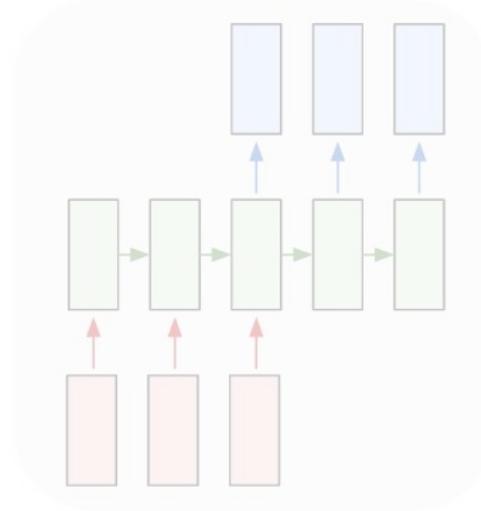
one to many



many to one



many to many



many to many

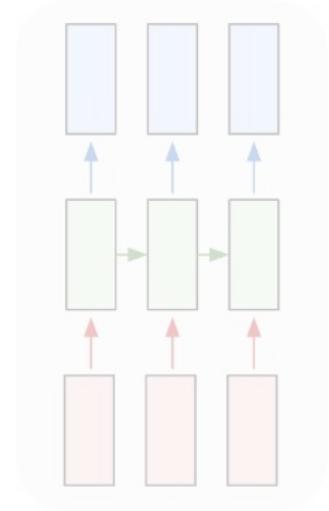
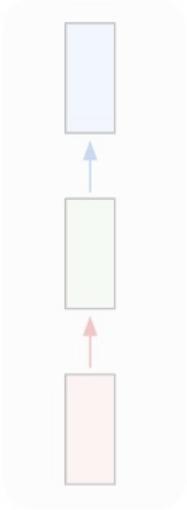


Image captioning

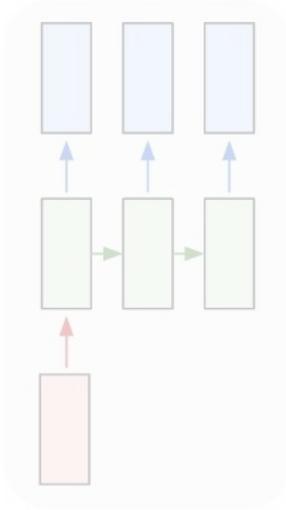
Source: <http://karpathy.github.io/2015/05/21/rnn-effectiveness/>

RNNs are Flexible

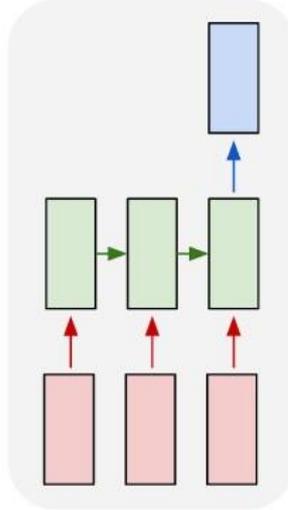
one to one



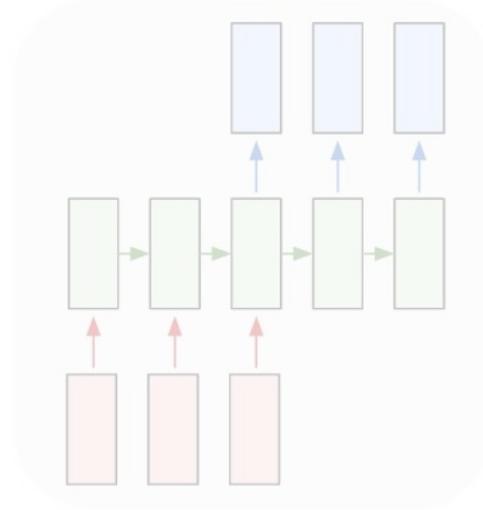
one to many



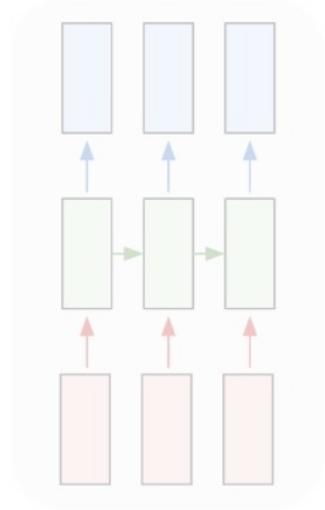
many to one



many to many



many to many

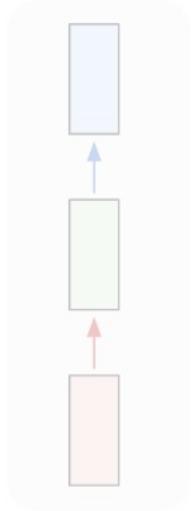


Language recognition

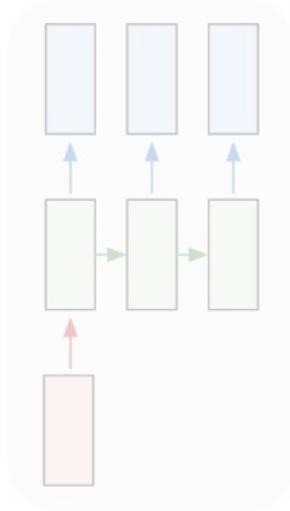
Source: <http://karpathy.github.io/2015/05/21/rnn-effectiveness/>

RNNs are Flexible

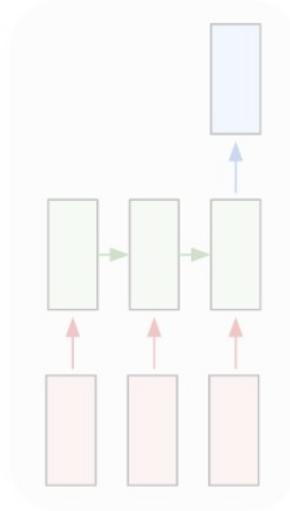
one to one



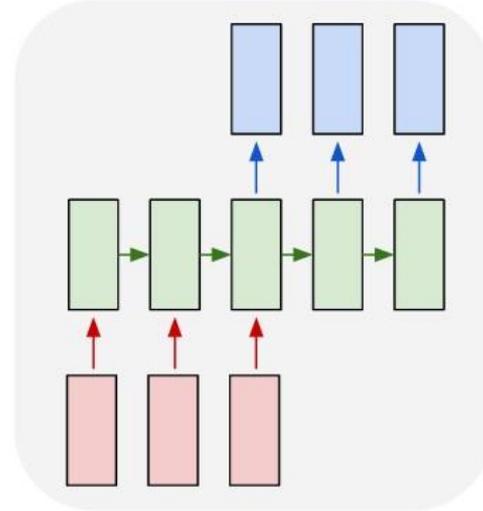
one to many



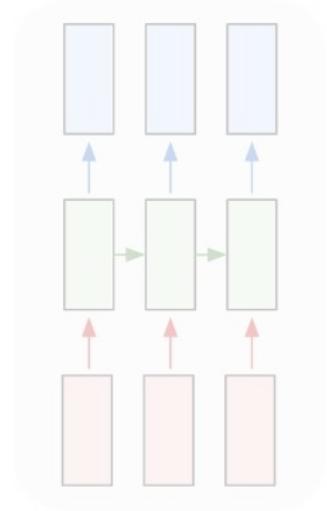
many to one



many to many



many to many

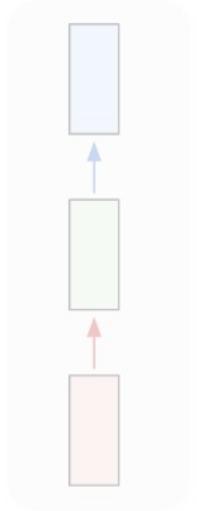


Machine translation

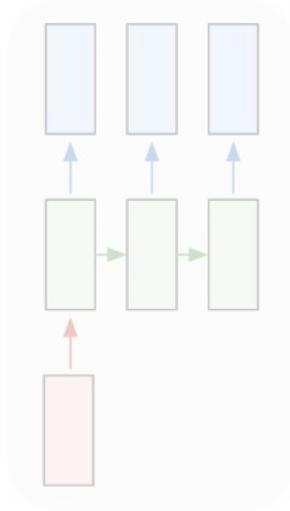
Source: <http://karpathy.github.io/2015/05/21/rnn-effectiveness/>

RNNs are Flexible

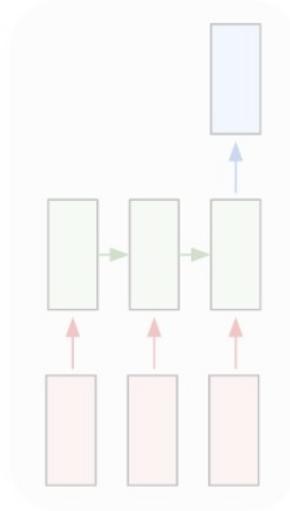
one to one



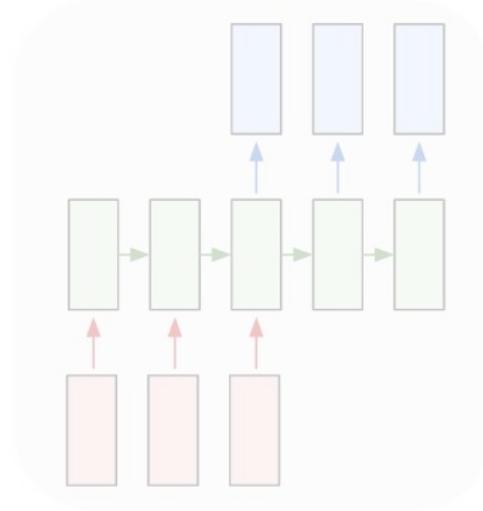
one to many



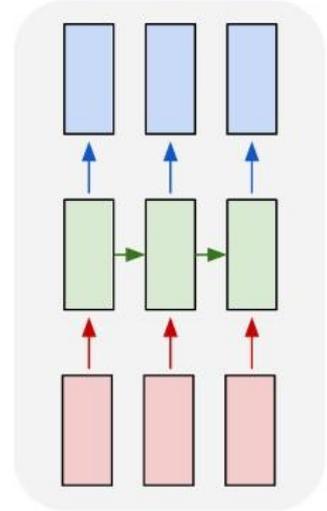
many to one



many to many



many to many

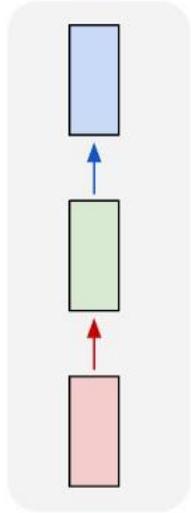


Event classification

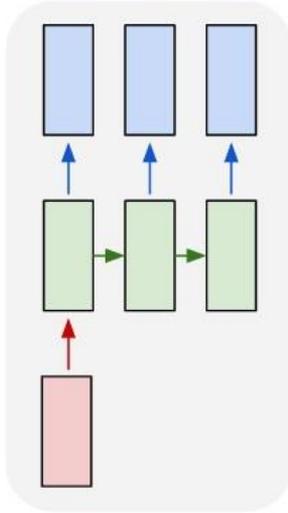
Source: <http://karpathy.github.io/2015/05/21/rnn-effectiveness/>

RNNs are Flexible

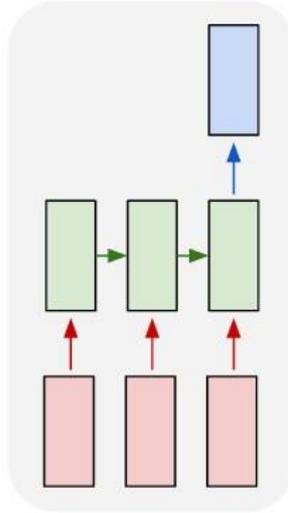
one to one



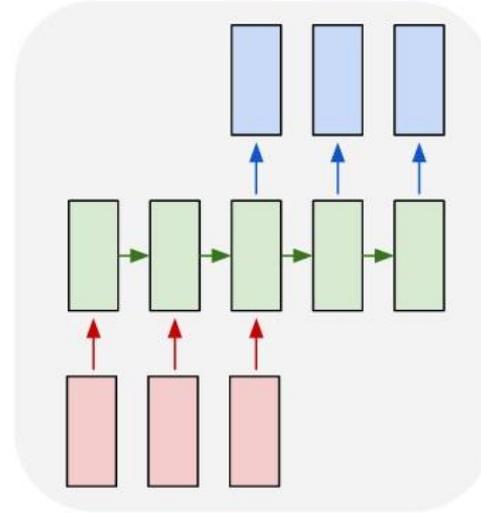
one to many



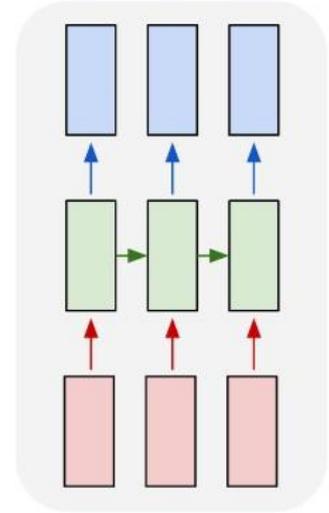
many to one



many to many



many to many

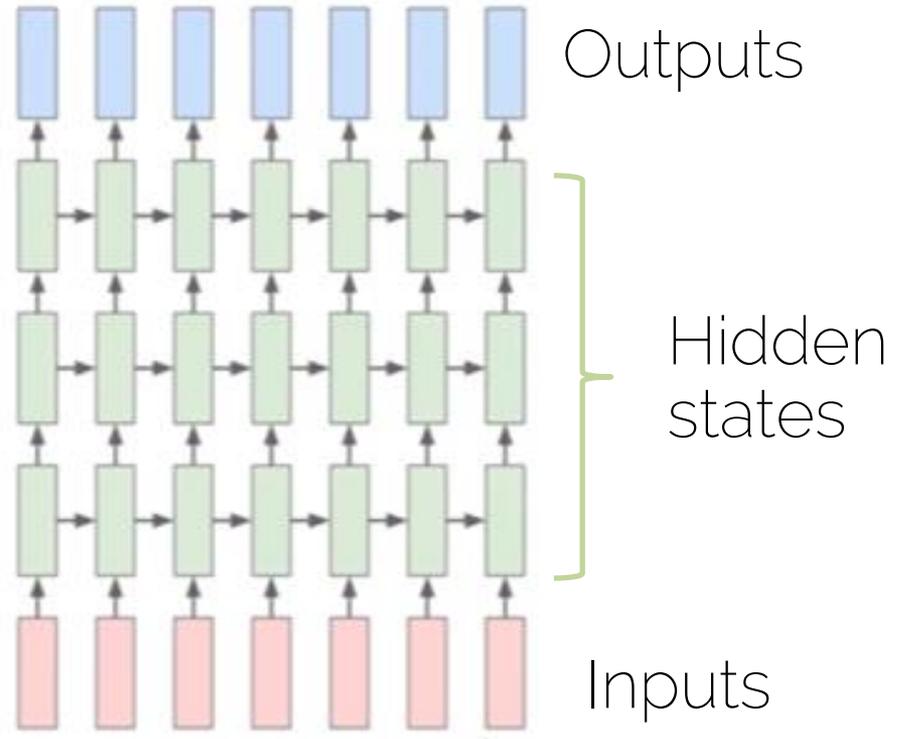


Event classification

Source: <http://karpathy.github.io/2015/05/21/rnn-effectiveness/>

Basic Structure of an RNN

- Multi-layer RNN



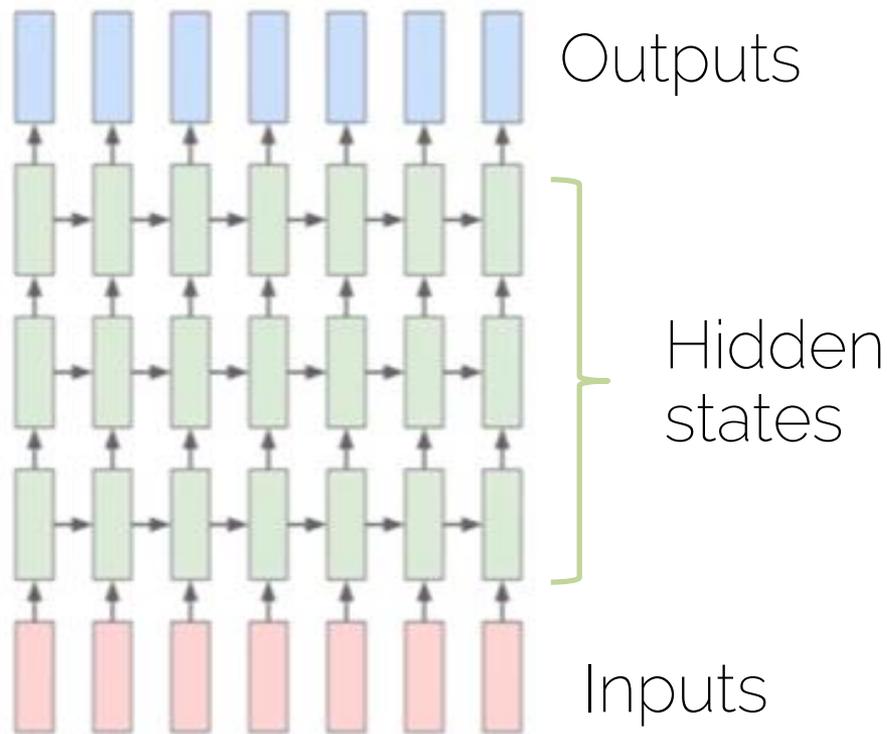
Basic Structure of an RNN

- Multi-layer RNN

The hidden state will have its own internal dynamics

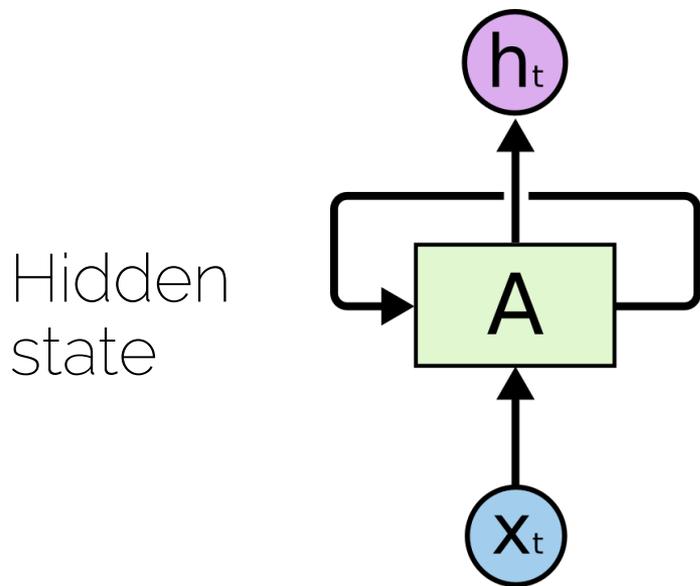


More expressive model!



Basic Structure of an RNN

- We want to have notion of “time” or “sequence”



Hidden
state

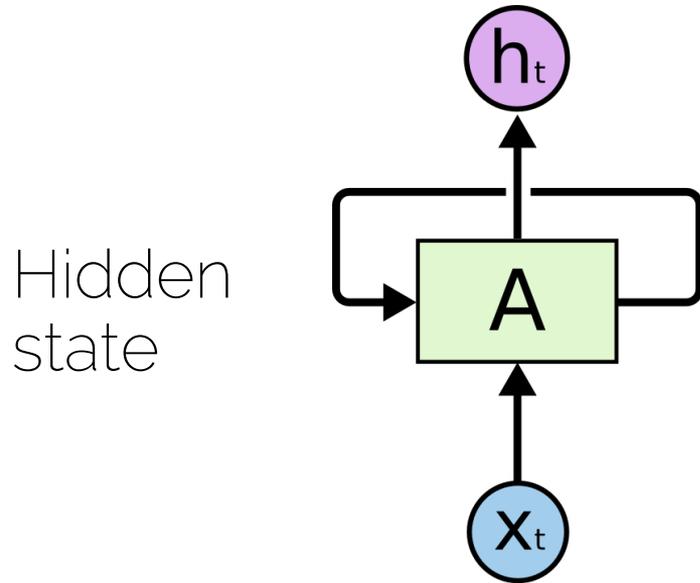
$$A_t = \theta_c A_{t-1} + \theta_x x_t$$

Previous
hidden
state

input

Basic Structure of an RNN

- We want to have notion of “time” or “sequence”

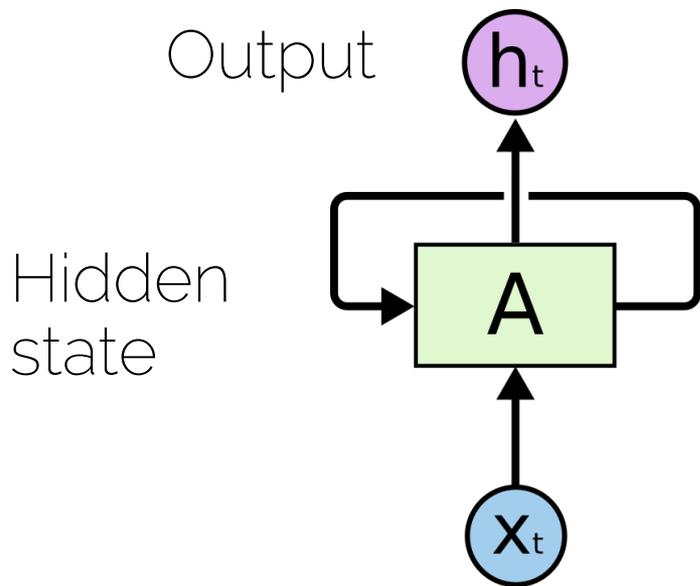


$$A_t = \theta_c A_{t-1} + \theta_x x_t$$

Parameters to be learned

Basic Structure of an RNN

- We want to have notion of “time” or “sequence”



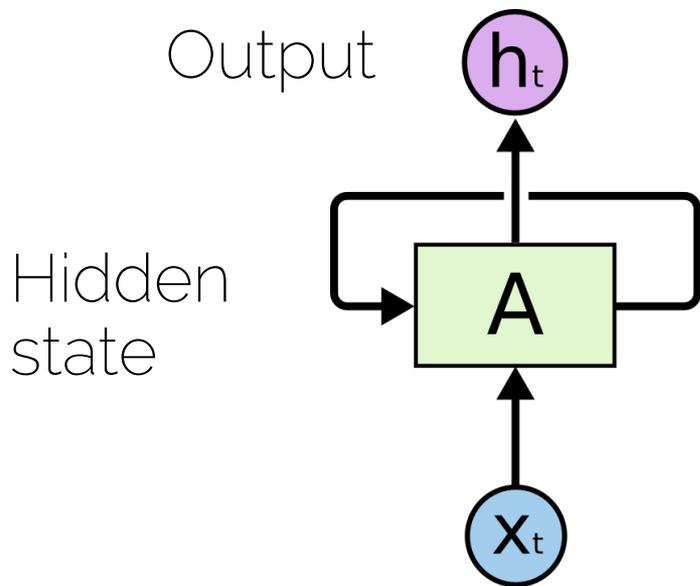
$$A_t = \theta_c A_{t-1} + \theta_x x_t$$

$$h_t = \theta_h A_t$$

Note: non-linearities
ignored for now

Basic Structure of an RNN

- We want to have notion of “time” or “sequence”



$$A_t = \theta_c A_{t-1} + \theta_x x_t$$

$$h_t = \theta_h A_t$$

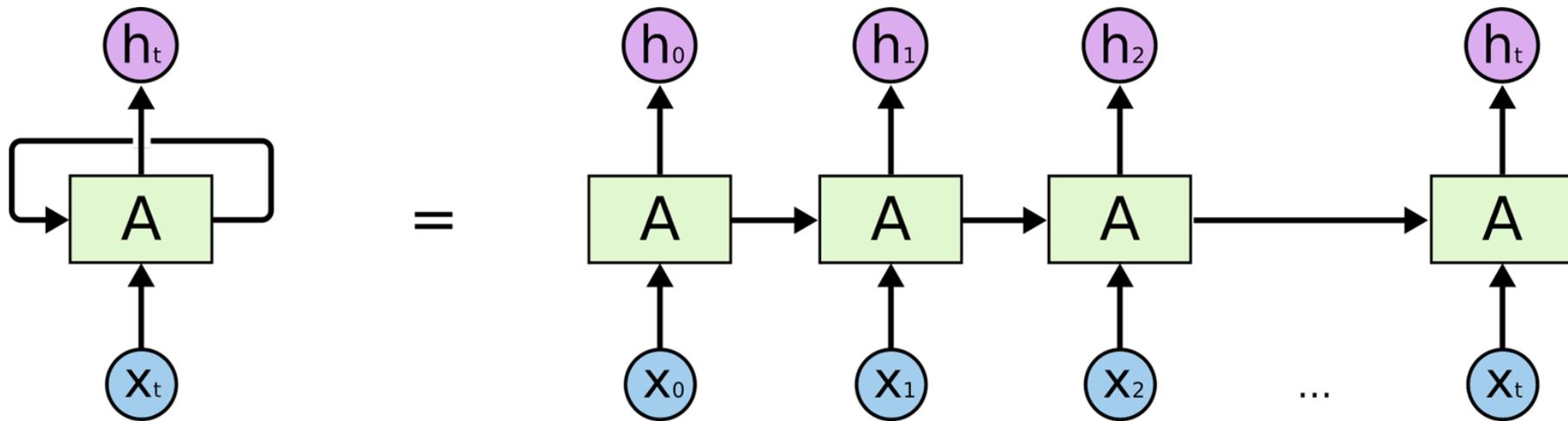
Same parameters for each time step = generalization!

[Olah, <https://colah.github.io> '15] Understanding LSTMs

Basic Structure of an RNN

- Unrolling RNNs

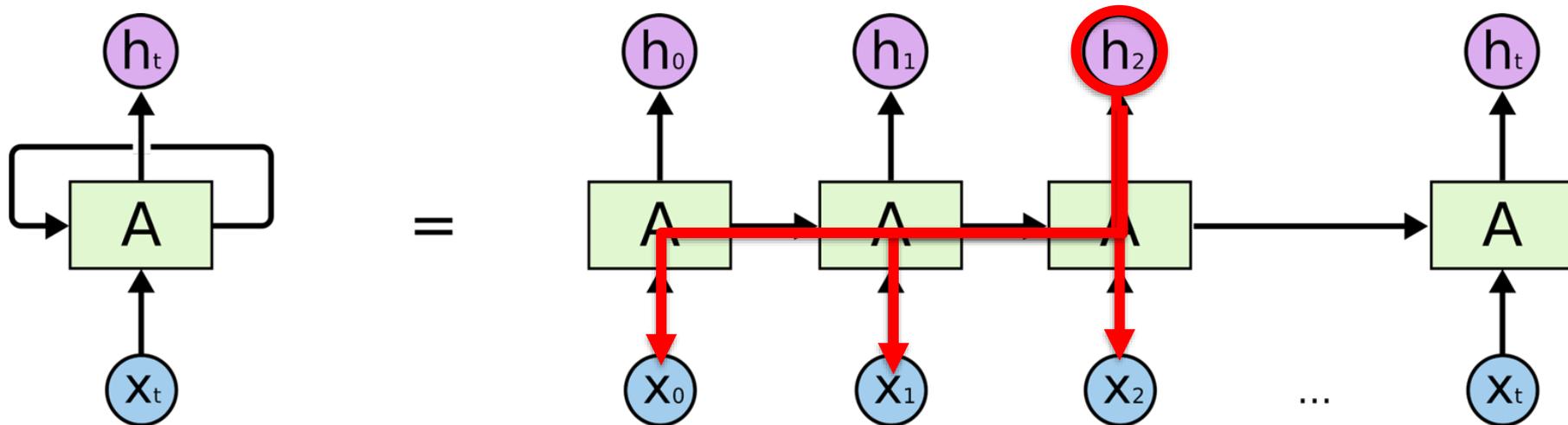
Same function for the hidden layers



[Olah, <https://colah.github.io> '15] Understanding LSTMs

Basic Structure of an RNN

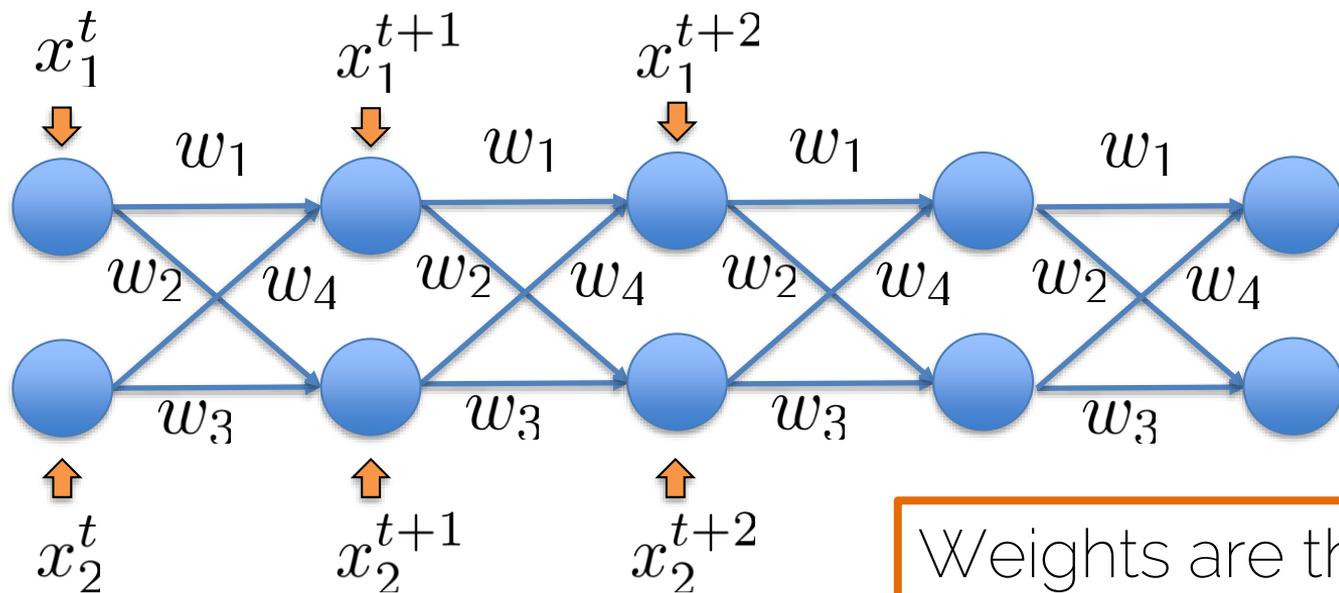
- Unrolling RNNs



[Olah, <https://colah.github.io> '15] Understanding LSTMs

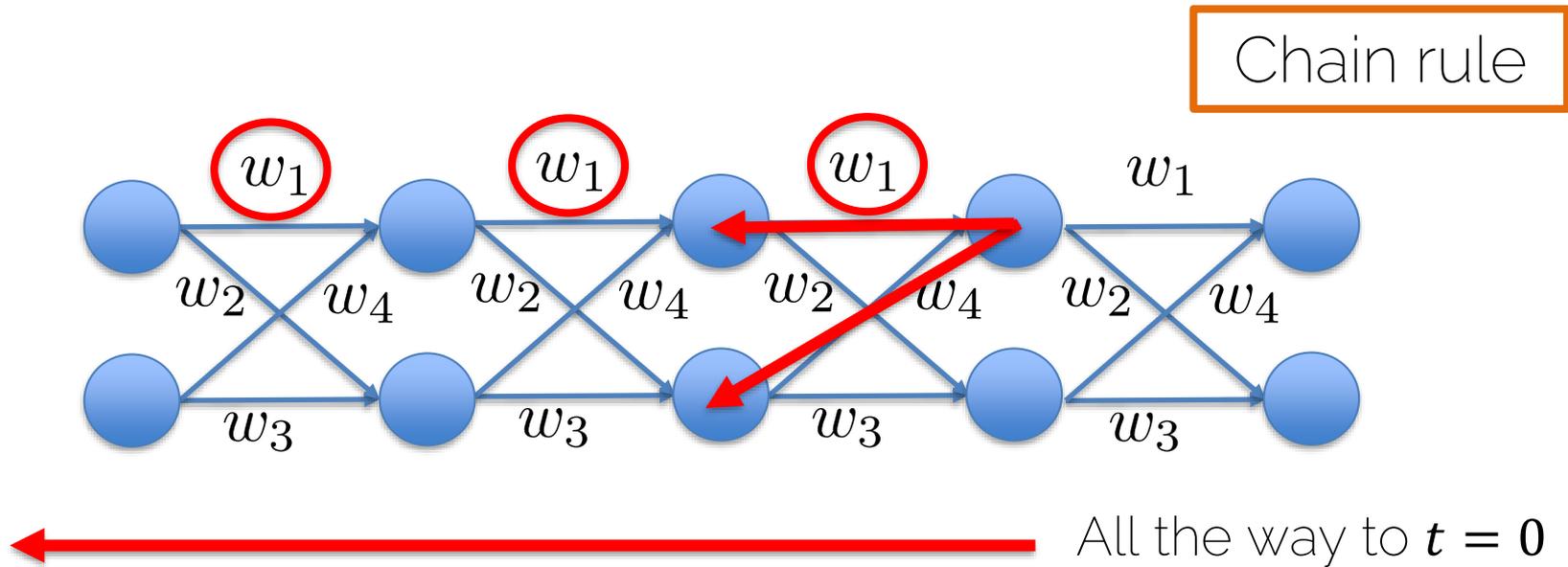
Basic Structure of an RNN

- Unrolling RNNs as feedforward nets



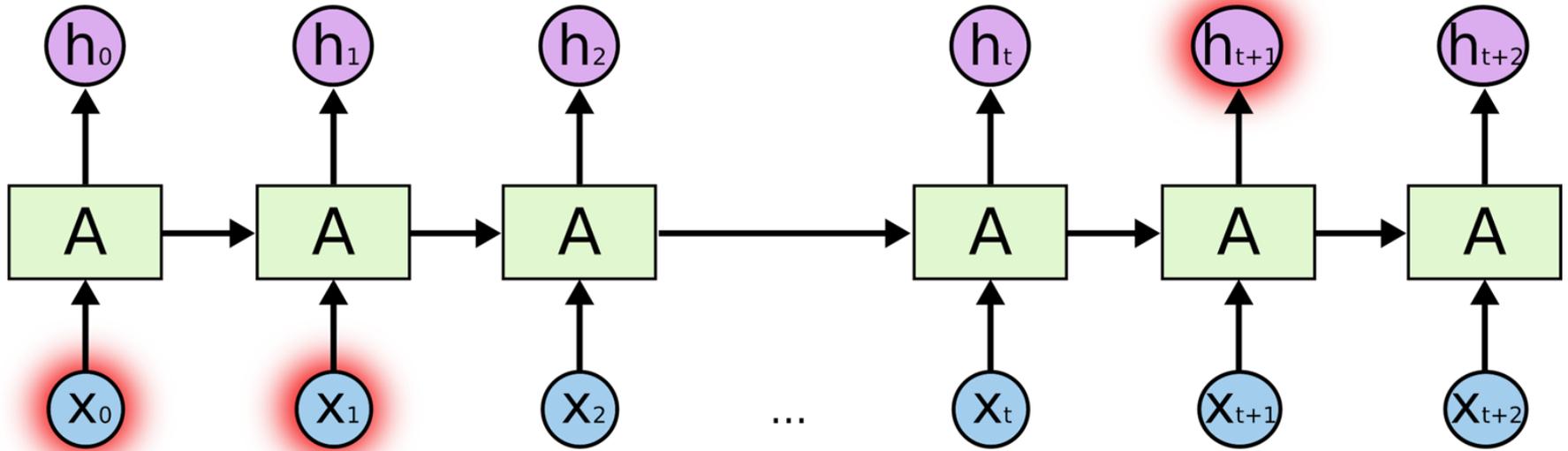
Backprop through an RNN

- Unrolling RNNs as feedforward nets



Add the derivatives at different times for each weight

Long-term Dependencies



I moved to Germany ...

so I speak German fluently.

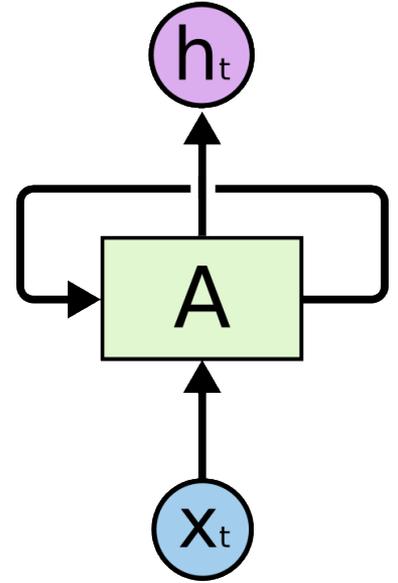
[Olah, <https://colah.github.io> '15] Understanding LSTMs

Long-term Dependencies

- Simple recurrence $\mathbf{A}_t = \boldsymbol{\theta}_c \mathbf{A}_{t-1} + \boldsymbol{\theta}_x \mathbf{x}_t$

- Let us forget the input $\mathbf{A}_t = \boldsymbol{\theta}_c^t \mathbf{A}_0$

Same weights are multiplied over and over again



Long-term Dependencies

- Simple recurrence $\mathbf{A}_t = \boldsymbol{\theta}_c^t \mathbf{A}_0$

What happens to small weights?

Vanishing gradient

What happens to large weights?

Exploding gradient

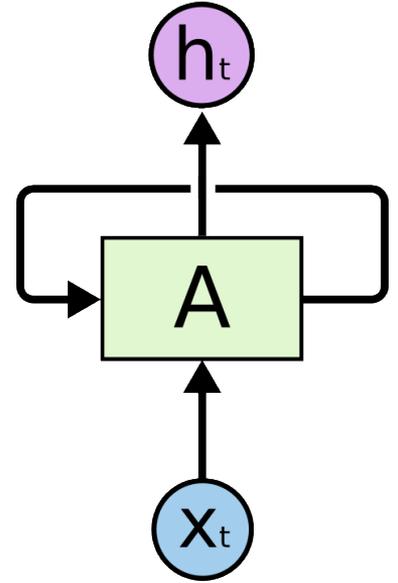
Long-term Dependencies

- Simple recurrence $\mathbf{A}_t = \boldsymbol{\theta}_c^t \mathbf{A}_0$
- If $\boldsymbol{\theta}$ admits eigendecomposition

$$\boldsymbol{\theta} = \mathbf{Q}\boldsymbol{\Lambda}\mathbf{Q}^T$$

Matrix of
eigenvectors

Diagonal of this
matrix are the
eigenvalues



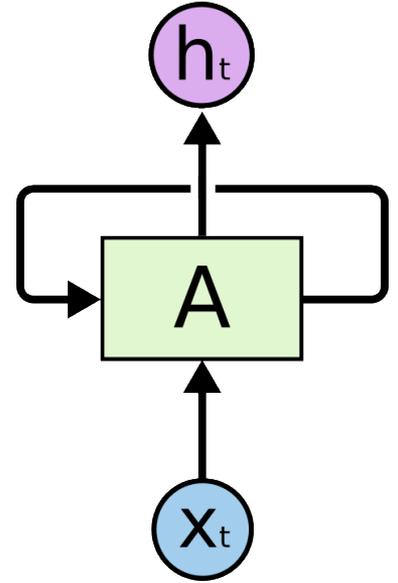
Long-term Dependencies

- Simple recurrence $\mathbf{A}_t = \boldsymbol{\theta}^t \mathbf{A}_0$
- If $\boldsymbol{\theta}$ admits eigendecomposition

$$\boldsymbol{\theta} = \mathbf{Q}\boldsymbol{\Lambda}\mathbf{Q}^T$$

- Orthogonal $\boldsymbol{\theta}$ allows us to simplify the recurrence

$$\mathbf{A}_t = \mathbf{Q}\boldsymbol{\Lambda}^t \mathbf{Q}^T \mathbf{A}_0$$



Long-term Dependencies

- Simple recurrence $\mathbf{A}_t = \mathbf{Q}\mathbf{\Lambda}^t\mathbf{Q}^T\mathbf{A}_0$

What happens to eigenvalues with magnitude less than one?

Vanishing gradient

What happens to eigenvalues with magnitude larger than one?

Exploding gradient



Gradient clipping

Long-term Dependencies

- Simple recurrence $\mathbf{A}_t = \boldsymbol{\theta}_c^t \mathbf{A}_0$

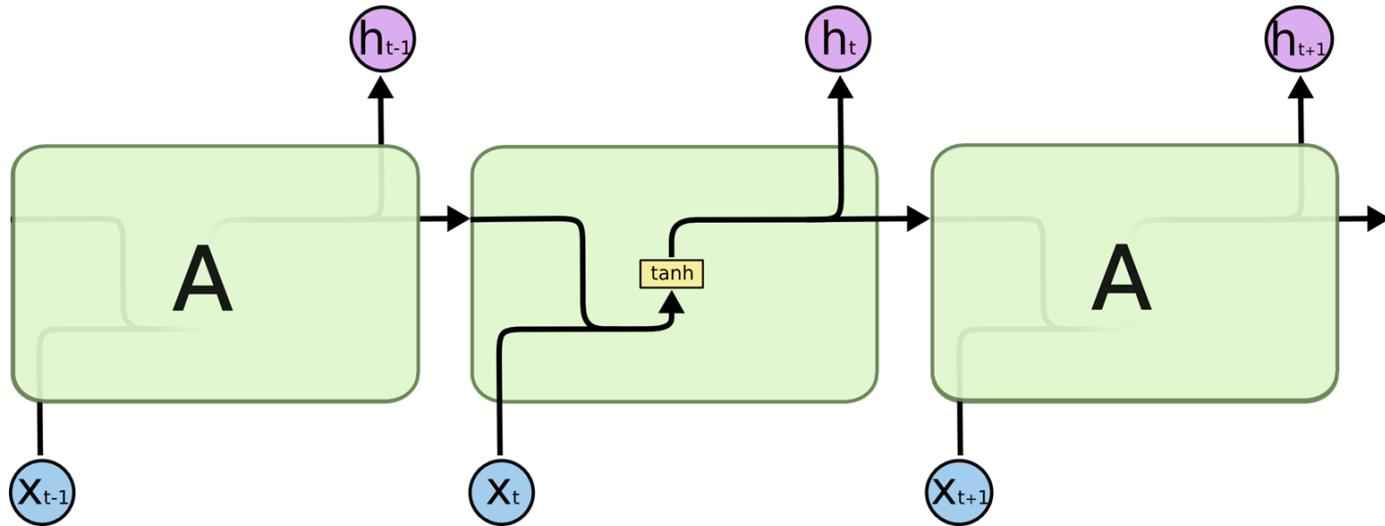
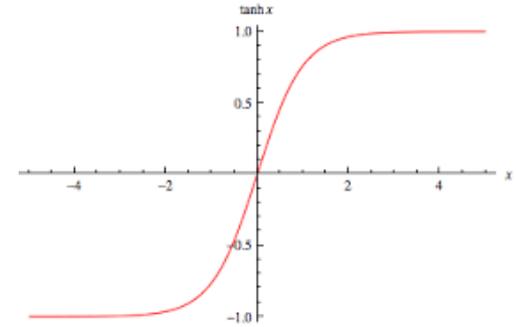


Let us just make a matrix with eigenvalues = 1

Allow the **cell** to maintain its "*state*"

Vanishing Gradient

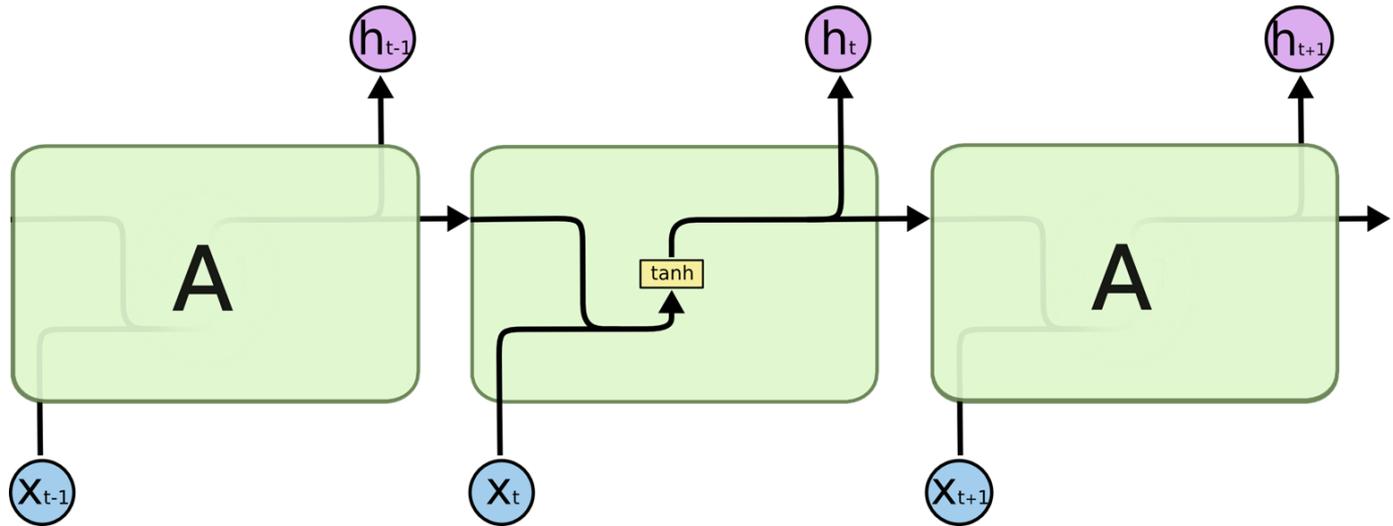
- 1. From the weights $\mathbf{A}_t = \boldsymbol{\theta}_c^t \mathbf{A}_0$
- 2. From the activation functions (*tanh*)



[Olah, <https://colah.github.io> '15] Understanding LSTMs

Vanishing Gradient

- 1. From the weights $\mathbf{A}_t = \theta^t \mathbf{A}_0$
1
- 2. From the activation functions (*tanh*) ?



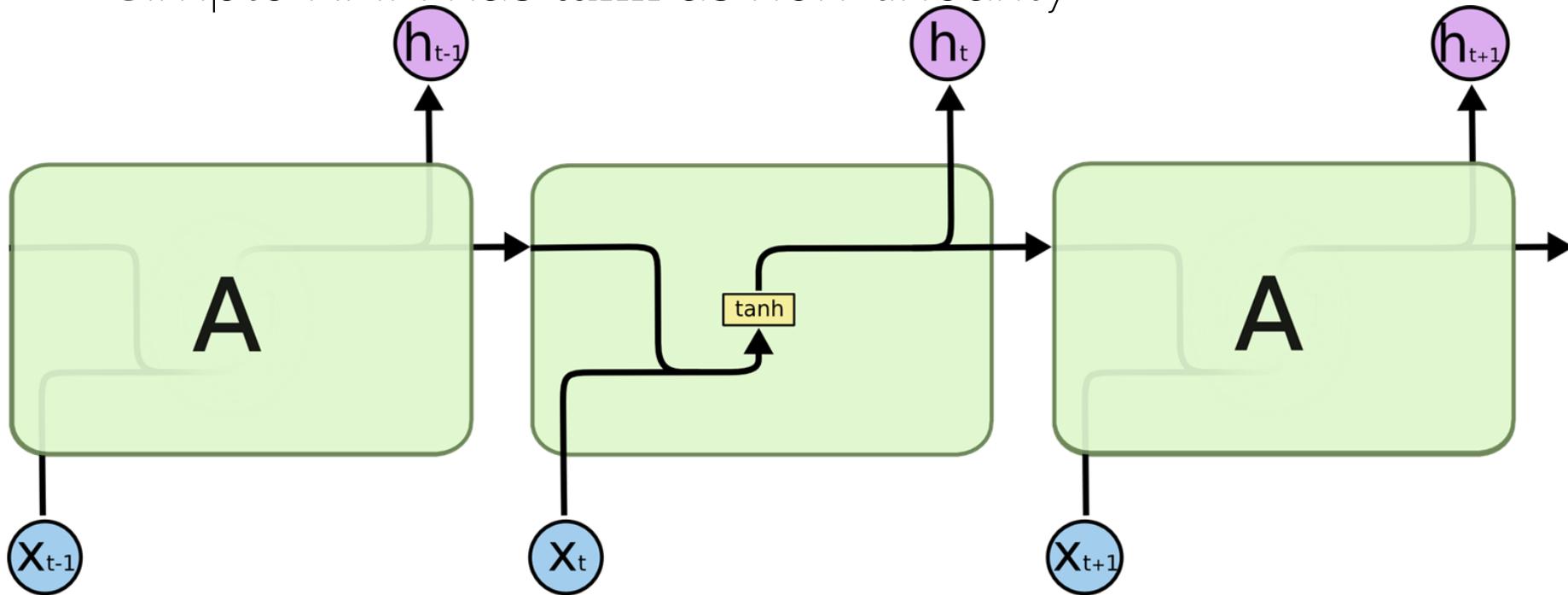
[Olah, <https://colah.github.io> '15] Understanding LSTMs

Long Short Term Memory

[Hochreiter et al., Neural Computation'97] Long Short-Term Memory

Long-Short Term Memory Units

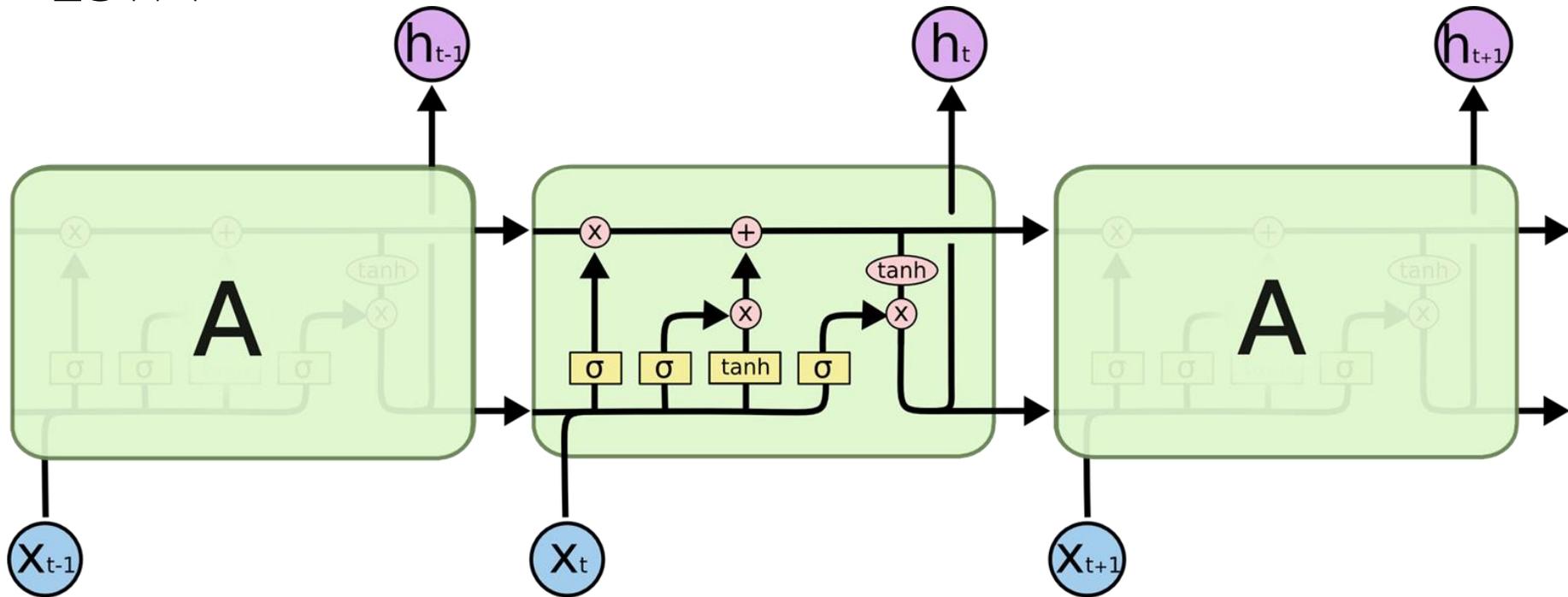
- Simple RNN has **tanh** as non-linearity



[Olah, <https://colah.github.io> '15] Understanding LSTMs

Long-Short Term Memory Units

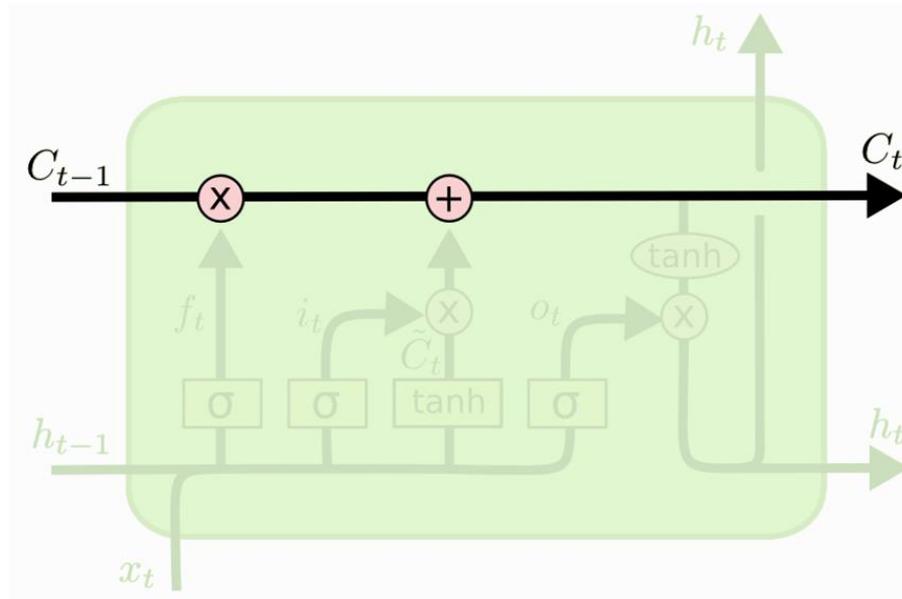
LSTM



[Olah, <https://colah.github.io> '15] Understanding LSTMs

Long-Short Term Memory Units

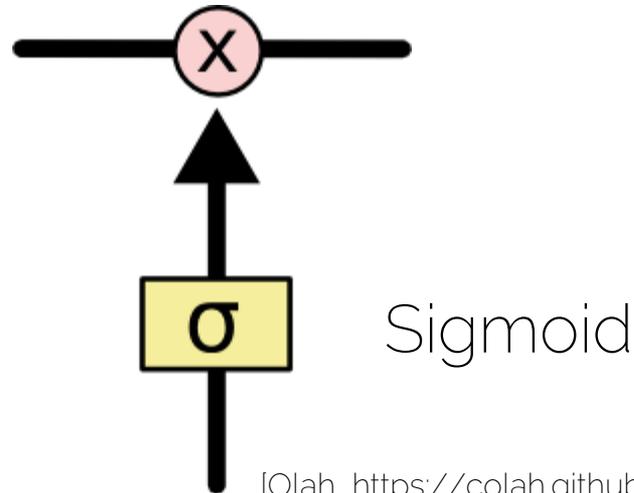
- Key ingredients
- Cell = transports the information through the unit



[Olah, <https://colah.github.io> '15] Understanding LSTMs

Long-Short Term Memory Units

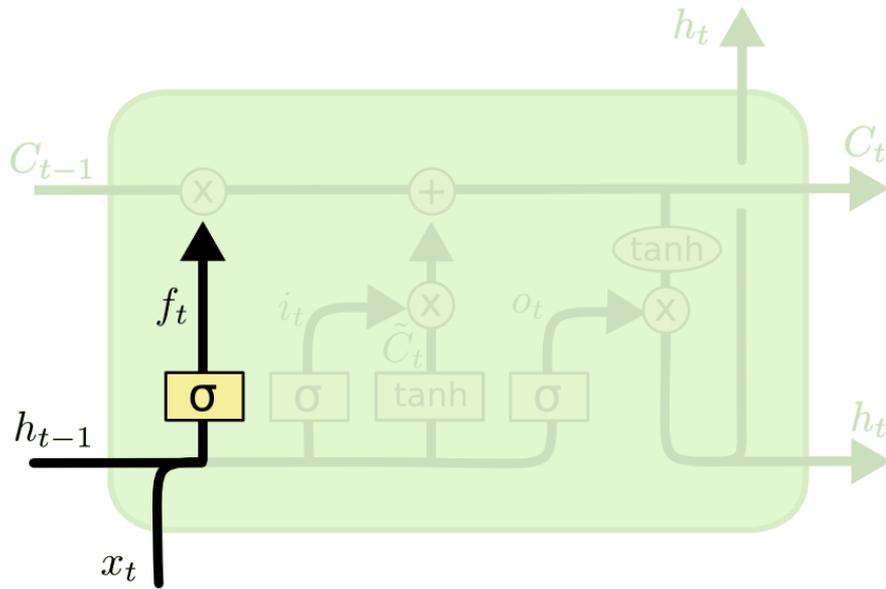
- Key ingredients
- Cell = transports the information through the unit
- Gate = remove or add information to the cell state



[Olah, <https://colah.github.io> '15] Understanding LSTMs

LSTM: Step by Step

- Forget gate $f_t = \text{sigm}(\theta_{xf}x_t + \theta_{hf}h_{t-1} + b_f)$

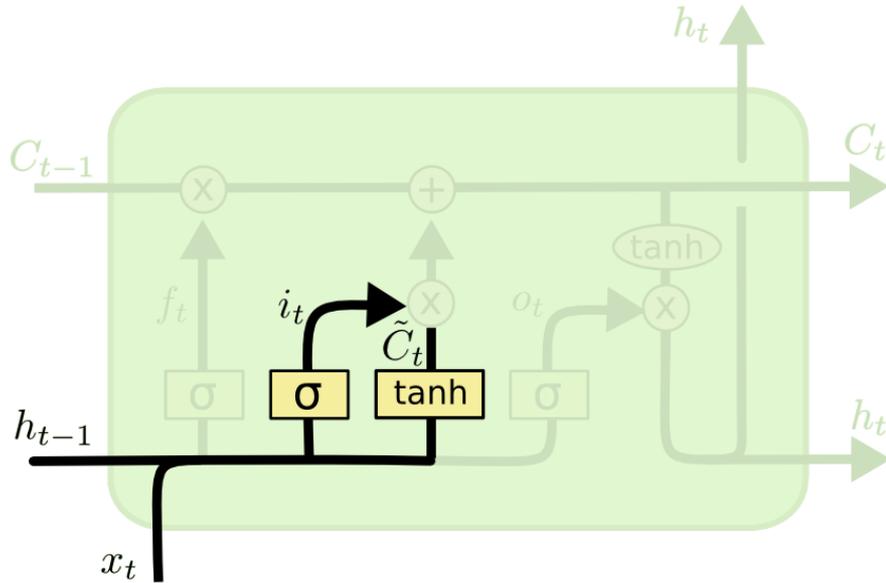


Decides when to erase the cell state

Sigmoid = output between **0** (forget) and **1** (keep)

LSTM: Step by Step

- Input gate $\mathbf{i}_t = \text{sigm}(\boldsymbol{\theta}_{xi}\mathbf{x}_t + \boldsymbol{\theta}_{hi}\mathbf{h}_{t-1} + \mathbf{b}_i)$

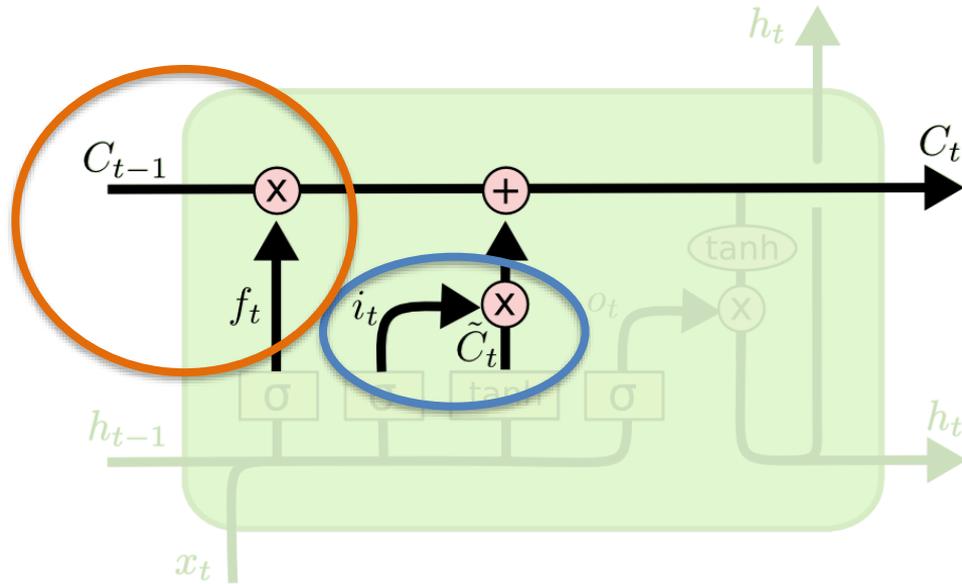


Decides which values will be updated

New cell state, output from a **$\tanh(-1,1)$**

LSTM: Step by Step

- Element-wise operations



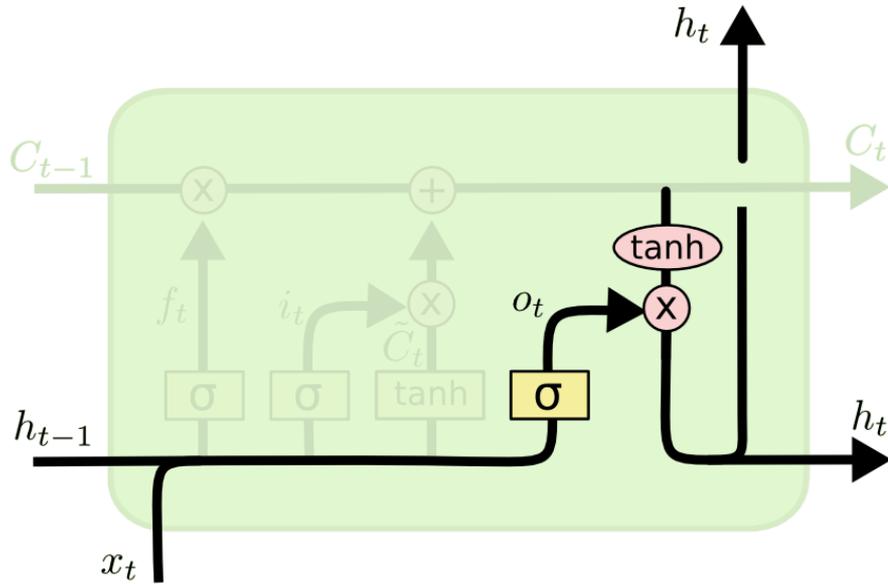
$$c_t = f_t \odot c_{t-1} + i_t \odot g_t$$

Previous
states

Current
state

LSTM: Step by Step

- Output gate $\mathbf{h}_t = \mathbf{o}_t \odot \tanh(\mathbf{C}_t)$



Decides which values will be outputted

Output from a $\tanh (-1, 1)$

LSTM: Step by Step

- Forget gate $\mathbf{f}_t = \text{sigm}(\boldsymbol{\theta}_{xf}\mathbf{x}_t + \boldsymbol{\theta}_{hf}\mathbf{h}_{t-1} + \mathbf{b}_f)$
- Input gate $\mathbf{i}_t = \text{sigm}(\boldsymbol{\theta}_{xi}\mathbf{x}_t + \boldsymbol{\theta}_{hi}\mathbf{h}_{t-1} + \mathbf{b}_i)$
- Output gate $\mathbf{o}_t = \text{sigm}(\boldsymbol{\theta}_{xo}\mathbf{x}_t + \boldsymbol{\theta}_{ho}\mathbf{h}_{t-1} + \mathbf{b}_o)$
- Cell update $\mathbf{g}_t = \text{tanh}(\boldsymbol{\theta}_{xg}\mathbf{x}_t + \boldsymbol{\theta}_{hg}\mathbf{h}_{t-1} + \mathbf{b}_g)$
- Cell $\mathbf{C}_t = \mathbf{f}_t \odot \mathbf{C}_{t-1} + \mathbf{i}_t \odot \mathbf{g}_t$
- Output $\mathbf{h}_t = \mathbf{o}_t \odot \text{tanh}(\mathbf{C}_t)$

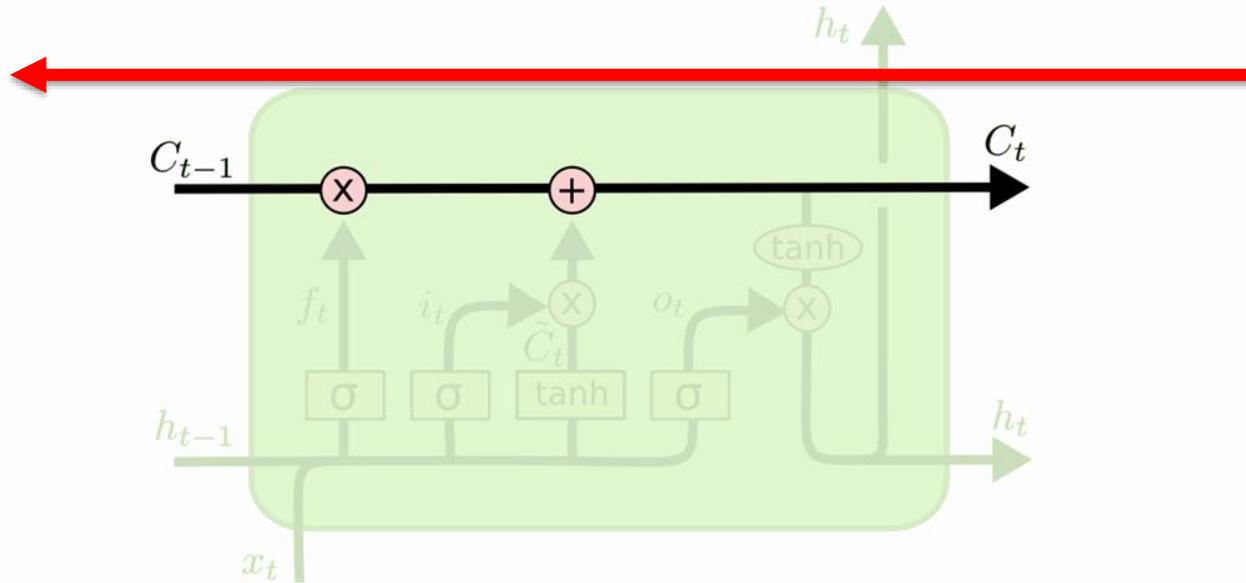
LSTM: Step by Step

- Forget gate $\mathbf{f}_t = \text{sigm}(\boldsymbol{\theta}_{x_f} \mathbf{x}_t + \boldsymbol{\theta}_{h_f} \mathbf{h}_{t-1} + \mathbf{b}_f)$
- Input gate $\mathbf{i}_t = \text{sigm}(\boldsymbol{\theta}_{x_i} \mathbf{x}_t + \boldsymbol{\theta}_{h_i} \mathbf{h}_{t-1} + \mathbf{b}_i)$
- Output gate $\mathbf{o}_t = \text{sigm}(\boldsymbol{\theta}_{x_o} \mathbf{x}_t + \boldsymbol{\theta}_{h_o} \mathbf{h}_{t-1} + \mathbf{b}_o)$
- Cell update $\mathbf{g}_t = \text{tanh}(\boldsymbol{\theta}_{x_g} \mathbf{x}_t + \boldsymbol{\theta}_{h_g} \mathbf{h}_{t-1} + \mathbf{b}_g)$
- Cell $\mathbf{C}_t = \mathbf{f}_t \odot \mathbf{C}_{t-1} + \mathbf{i}_t \odot \mathbf{g}_t$
- Output $\mathbf{h}_t = \mathbf{o}_t \odot \text{tanh}(\mathbf{C}_t)$

Learned through
backpropagation

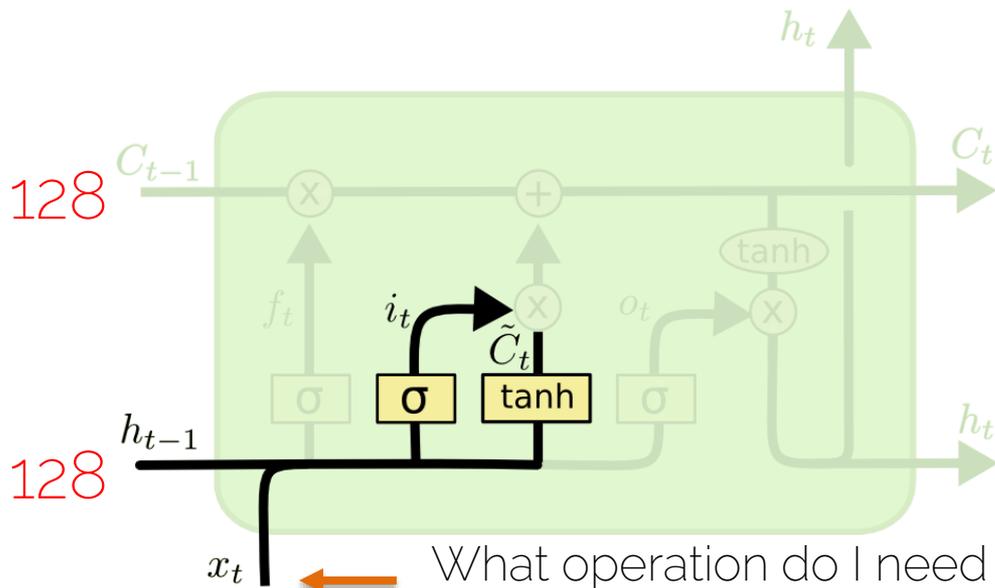
LSTM

- Highway for the gradient to flow



LSTM: Dimensions

- Cell update $\mathbf{g}_t = \tanh(\boldsymbol{\theta}_{xg}\mathbf{x}_t + \boldsymbol{\theta}_{hg}\mathbf{h}_{t-1} + \mathbf{b}_g)$



When coding an LSTM, we have to define the size of the hidden state

Dimensions need to match

What operation do I need to do to my input to get a 128 vector representation?

[Olah, <https://colah.github.io> '15] Understanding LSTMs

LSTM in code

```
def lstm_step_forward(x, prev_h, prev_c, Wx, Wh, b):
    """
    Forward pass for a single timestep of an LSTM.

    The input data has dimension D, the hidden state has dimension H, and we use
    a minibatch size of N.

    Inputs:
    - x: Input data, of shape (N, D)
    - prev_h: Previous hidden state, of shape (N, H)
    - prev_c: previous cell state, of shape (N, H)
    - Wx: Input-to-hidden weights, of shape (D, 4H)
    - Wh: Hidden-to-hidden weights, of shape (H, 4H)
    - b: Biases, of shape (4H,)

    Returns a tuple of:
    - next_h: Next hidden state, of shape (N, H)
    - next_c: Next cell state, of shape (N, H)
    - cache: Tuple of values needed for backward pass.
    """
    next_h, next_c, cache = None, None, None

    N, H = prev_h.shape
    # 1
    a = np.dot(x, Wx) + np.dot(prev_h, Wh) + b

    # 2
    ai = a[:, :H]
    af = a[:, H:2*H]
    ao = a[:, 2*H:3*H]
    ag = a[:, 3*H:]

    # 3
    i = sigmoid(ai)
    f = sigmoid(af)
    o = sigmoid(ao)
    g = np.tanh(ag)

    # 4
    next_c = f * prev_c + i * g

    # 5
    next_h = o * np.tanh(next_c)

    cache = i, f, o, g, a, ai, af, ao, ag, Wx, Wh, b, prev_h, prev_c, x, next_c, next_h

    return next_h, next_c, cache
```

```
def lstm_step_backward(dnext_h, dnext_c, cache):
    """
    Backward pass for a single timestep of an LSTM.

    Inputs:
    - dnext_h: Gradients of next hidden state, of shape (N, H)
    - dnext_c: Gradients of next cell state, of shape (N, H)
    - cache: Values from the forward pass

    Returns a tuple of:
    - dx: Gradient of input data, of shape (N, D)
    - dprev_h: Gradient of previous hidden state, of shape (N, H)
    - dprev_c: Gradient of previous cell state, of shape (N, H)
    - dWx: Gradient of input-to-hidden weights, of shape (D, 4H)
    - dWh: Gradient of hidden-to-hidden weights, of shape (H, 4H)
    - db: Gradient of biases, of shape (4H,)
    """
    dx, dh, dc, dWx, dWh, db = None, None, None, None, None, None

    i, f, o, g, a, ai, af, ao, ag, Wx, Wh, b, prev_h, prev_c, x, next_c, next_h = cache

    # backprop into step 5
    do = np.tanh(next_c) * dnext_h
    dnext_c += o * (1 - np.tanh(next_c) ** 2) * dnext_h

    # backprop into 4
    df = prev_c * dnext_c
    dprev_c = f * dnext_c
    di = g * dnext_c
    dg = i * dnext_c

    # backprop into 3
    dai = sigmoid(ai) * (1 - sigmoid(ai)) * di
    daf = sigmoid(af) * (1 - sigmoid(af)) * df
    dao = sigmoid(ao) * (1 - sigmoid(ao)) * do
    dag = (1 - np.tanh(ag) ** 2) * dg

    # backprop into 2
    da = np.hstack((dai, daf, dao, dag))

    # backprop into 1
    db = np.sum(da, axis = 0)
    dprev_h = np.dot(Wh, da.T).T
    dwh = np.dot(prev_h.T, da)
    dx = np.dot(da, Wx.T)
    dWx = np.dot(x.T, da)

    return dx, dprev_h, dprev_c, dWx, dWh, db
```

Attention

Attention is all you need

Attention Is All You Need

Ashish Vaswani*
Google Brain
avaswani@google.com

Noam Shazeer*
Google Brain
noam@google.com

Niki Parmar*
Google Research
nikip@google.com

Jakob Uszkoreit*
Google Research
usz@google.com

Llion Jones*
Google Research
llion@google.com

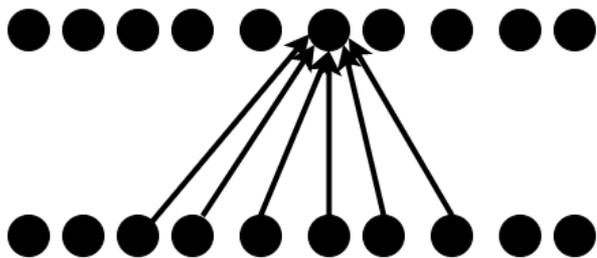
Aidan N. Gomez* †
University of Toronto
aidan@cs.toronto.edu

Lukasz Kaiser*
Google Brain
lukaszkaizer@google.com

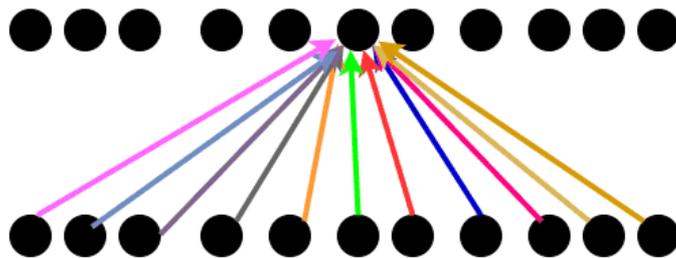
Illia Polosukhin* ‡
illia.polosukhin@gmail.com

Attention vs convolution

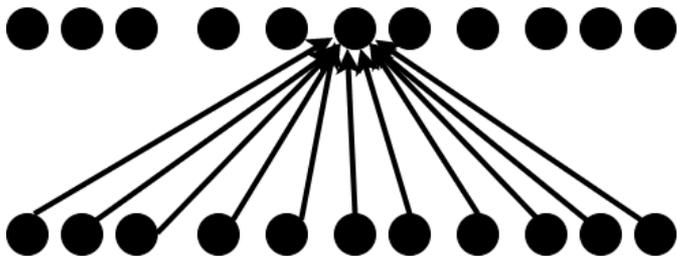
Convolution



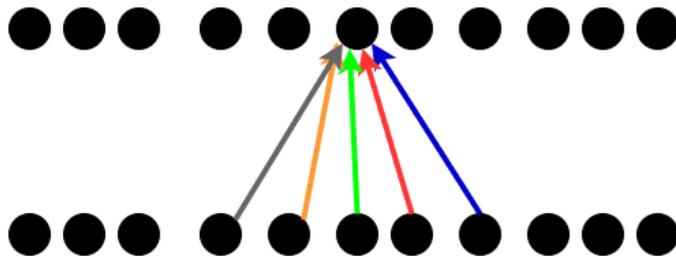
Global attention



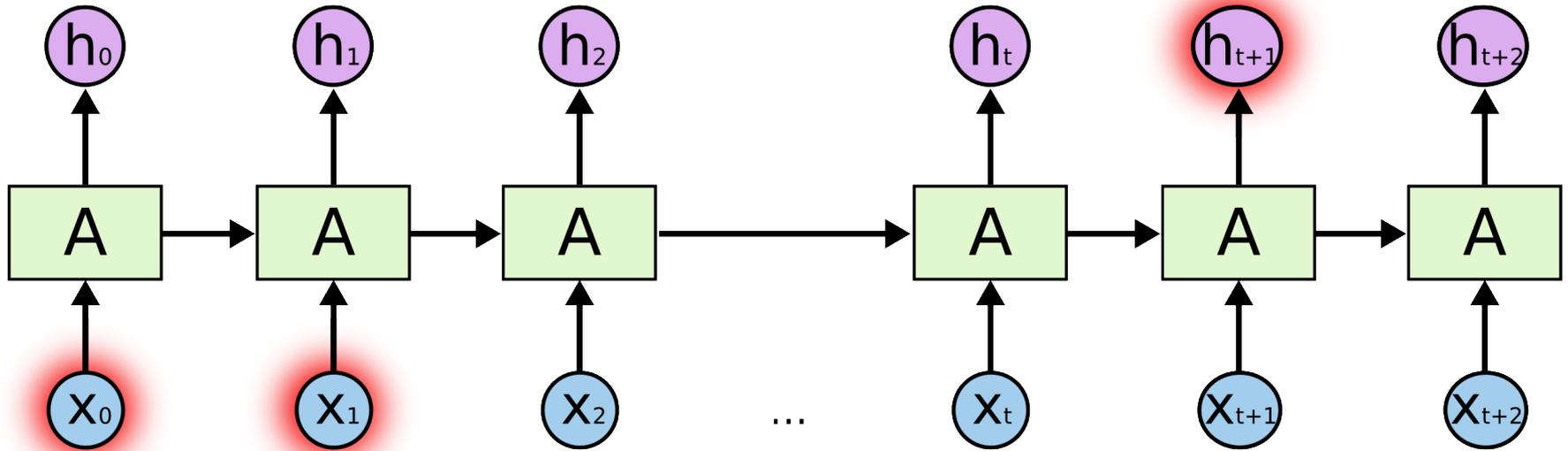
Fully Connected layer



Local attention



Long-Term Dependencies

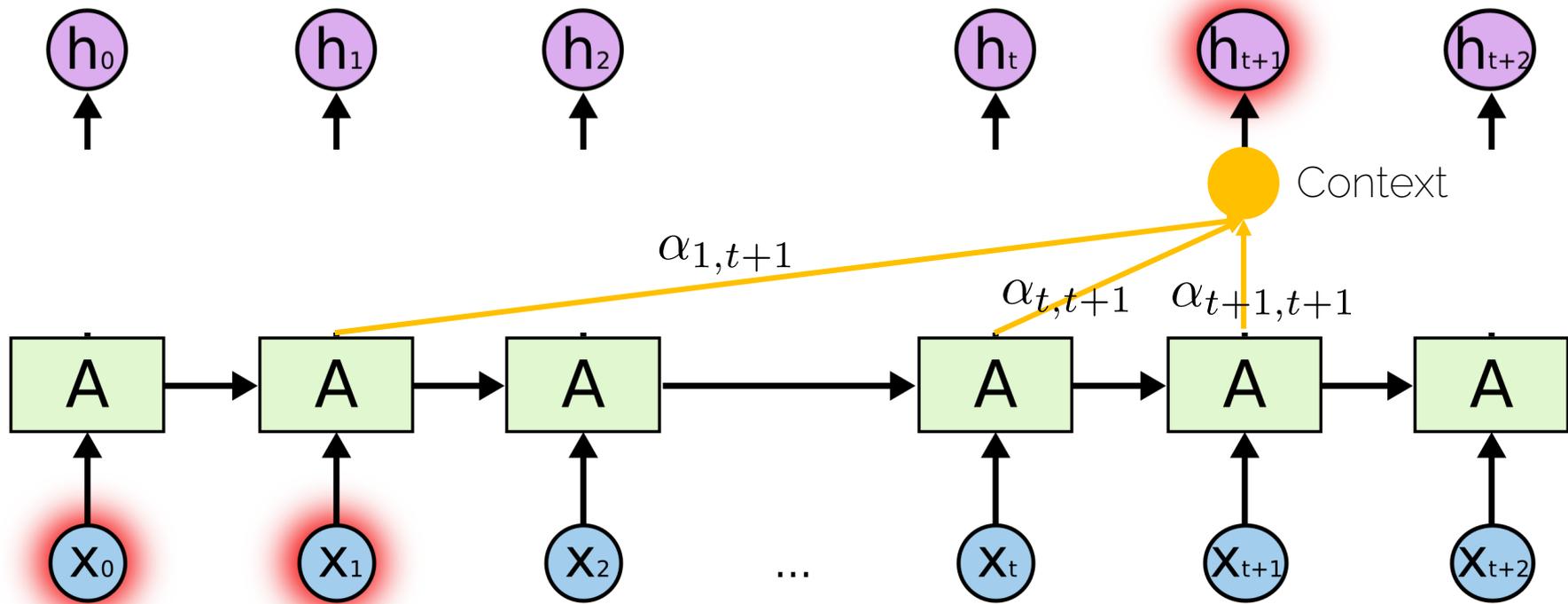


I moved to Germany ...

so I speak German fluently.

Source: <https://colah.github.io/posts/2015-08-Understanding-LSTMs/>

Attention: Intuition

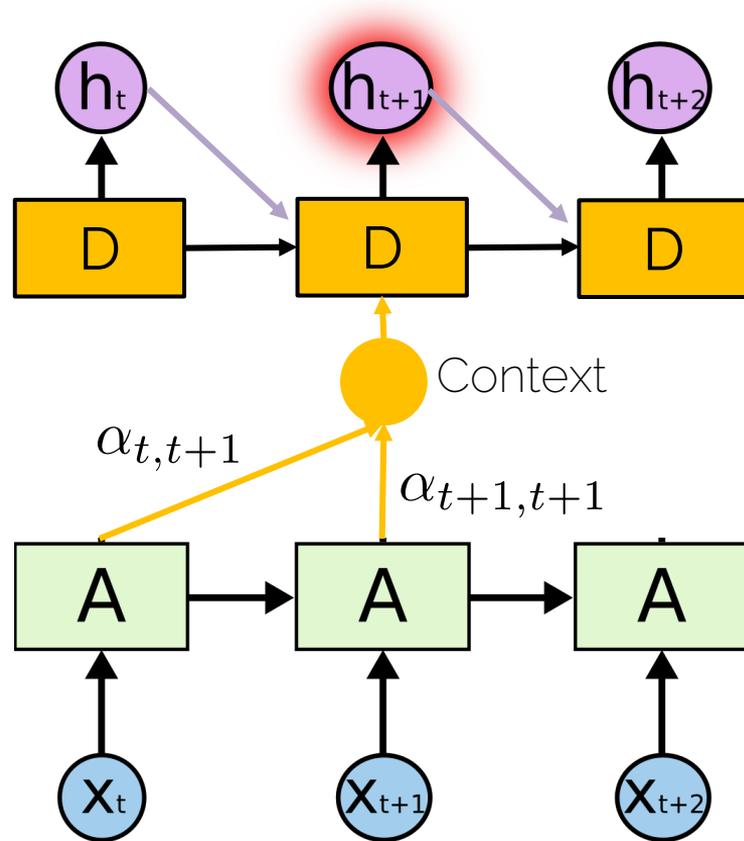


I moved to Germany ...

so I speak German fluently

Attention: Architecture

- A decoder processes the information
- Decoders take as input:
 - Previous decoder hidden state
 - Previous output
 - Attention



Transformers

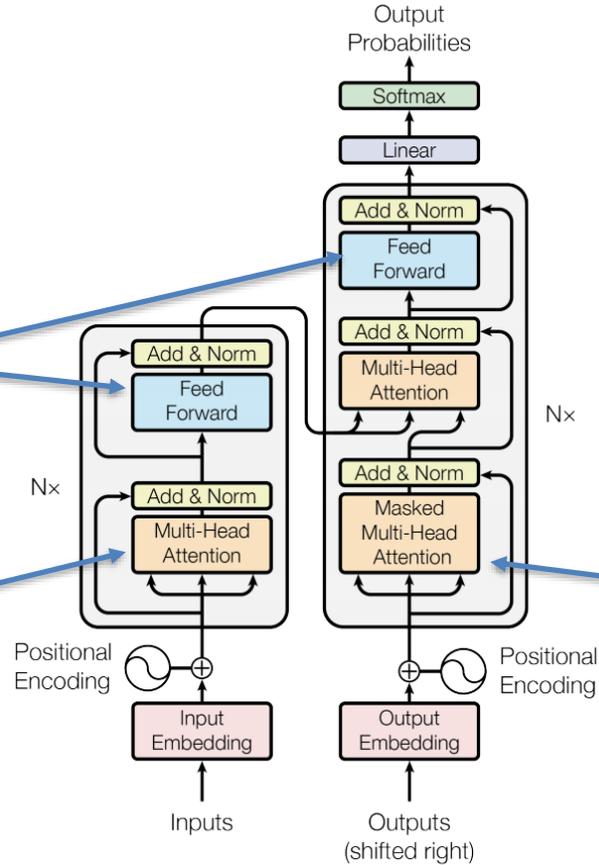
Deep Learning Revolution

	Deep Learning	Deep Learning 2.0
Main idea	Convolution	Attention
Field invented	Computer vision	NLP
Started	NeurIPS 2012	NeurIPS 2017
Paper	AlexNet	Transformers
Conquered vision	Around 2014-2015	Around 2020-2021
Replaced (Augmented)	Traditional ML/CV	CNNs, RNNs

Transformers

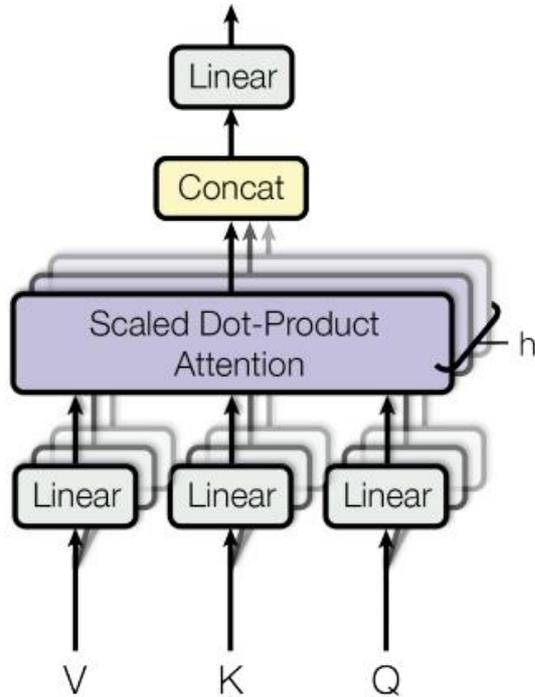
Fully connected layer

Multi-Head Attention on the "encoder"



Masked Multi-Head Attention on the "decoder"

Multi-Head Attention



Intuition: Take the query Q, find the most similar key K, and then find the value V that corresponds to the key.

In other words, learn V, K, Q where:
V – here is a bunch of interesting things.
K – here is how we can index some things.
Q – I would like to know this interesting thing.

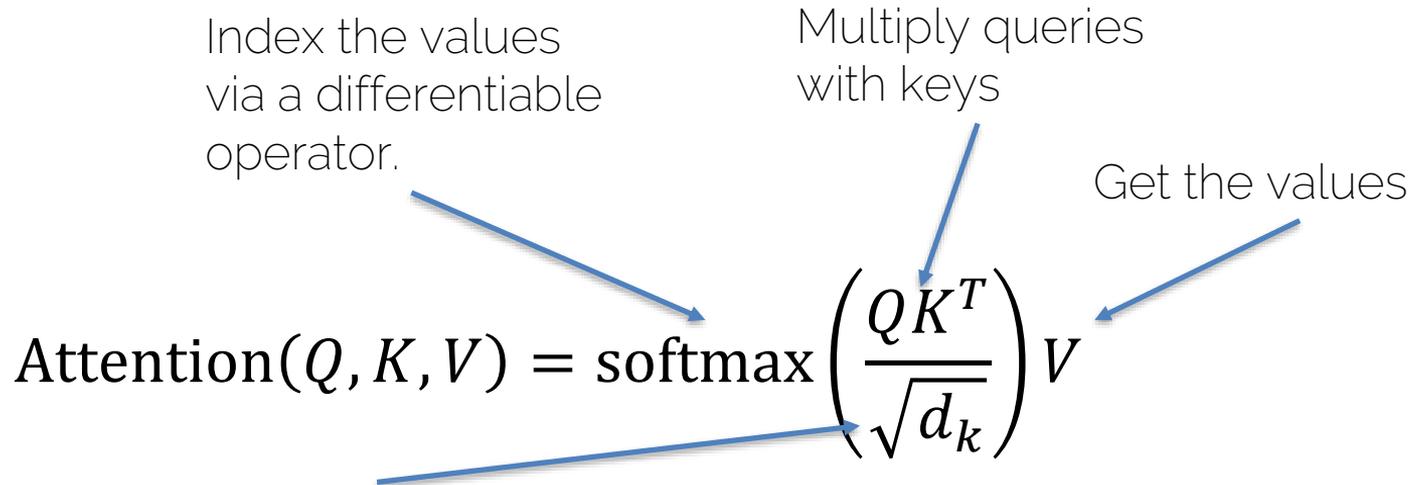
Loosely connected to Neural Turing Machines (Graves et al.).

Multi-Head Attention

Index the values
via a differentiable
operator.

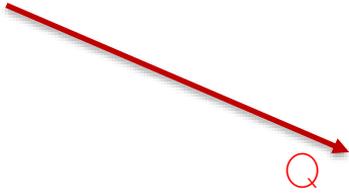
Multiply queries
with keys

Get the values

$$\text{Attention}(Q, K, V) = \text{softmax} \left(\frac{QK^T}{\sqrt{d_k}} \right) V$$
The diagram illustrates the attention mechanism equation. Three blue arrows point from descriptive text to parts of the equation: one from 'Index the values via a differentiable operator.' to the softmax function, one from 'Multiply queries with keys' to the QK^T term, and one from 'Get the values' to the V term. A fourth blue arrow points from the bottom left towards the denominator $\sqrt{d_k}$.

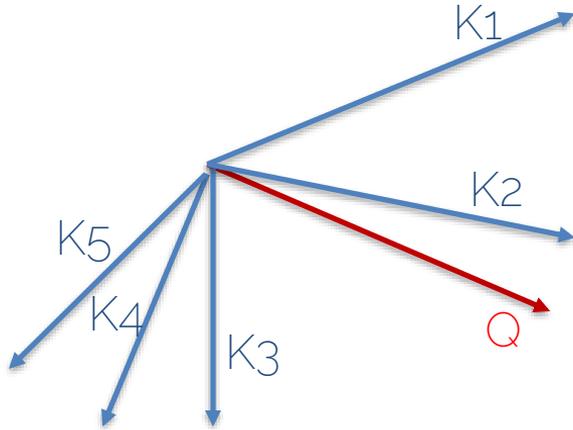
To train them well, divide by $\sqrt{d_k}$, "probably" because for large values of the key's dimension, the dot product grows large in magnitude, pushing the softmax function into regions where it has extremely small gradients.

Multi-Head Attention

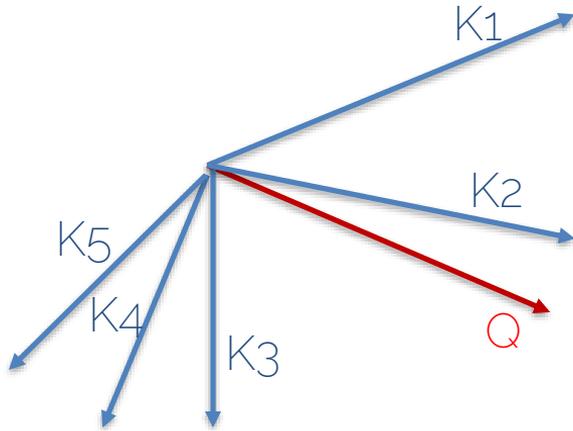


Adapted from Y. Kilcher

Multi-Head Attention

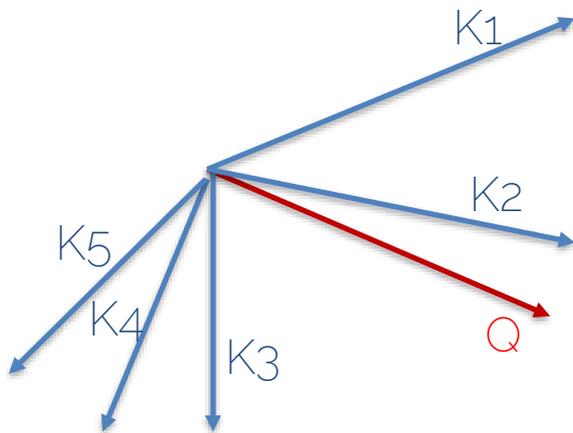


Multi-Head Attention



Values
V1
V2
V3
V4
V5

Multi-Head Attention

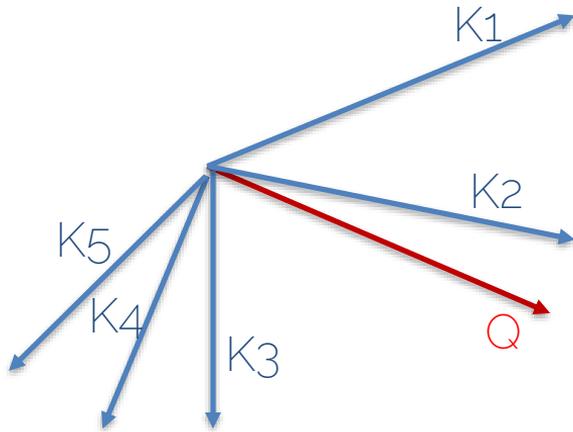


Values
V1
V2
V3
V4
V5

$$QK^T$$

Essentially, dot product between $\langle Q, K1 \rangle$, $\langle Q, K2 \rangle$, $\langle Q, K3 \rangle$, $\langle Q, K4 \rangle$, $\langle Q, K5 \rangle$.

Multi-Head Attention

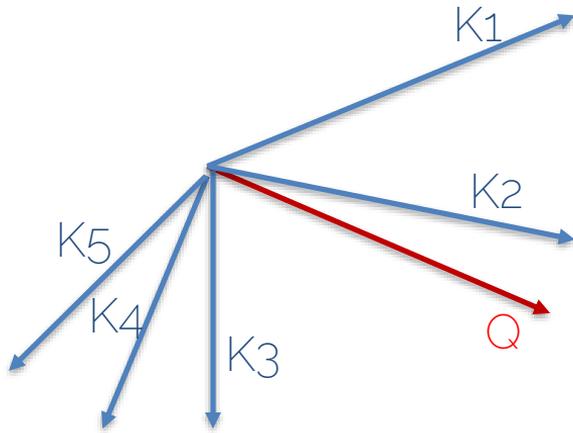


Values
V1
V2
V3
V4
V5

$$\text{softmax} \left(\frac{QK^T}{\sqrt{d_k}} \right)$$

Is simply inducing a distribution over the values.
The larger a value is, the higher is its softmax value.
Can be interpreted as a differentiable soft indexing.

Multi-Head Attention

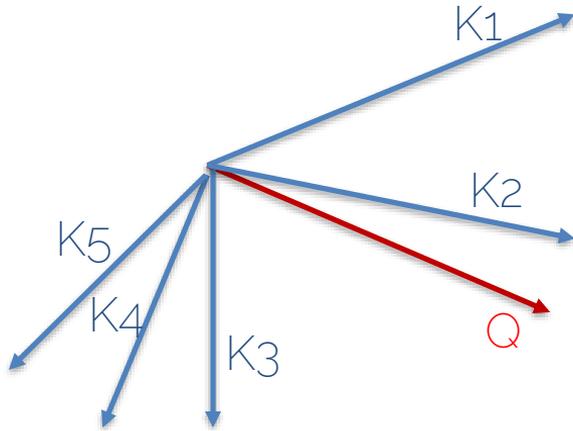


Values
V1
V2
V3
V4
V5

$$\text{softmax} \left(\frac{QK^T}{\sqrt{d_k}} \right)$$

Is simply inducing a distribution over the values.
The larger a value is, the higher is its softmax value.
Can be interpreted as a differentiable soft indexing.

Multi-Head Attention

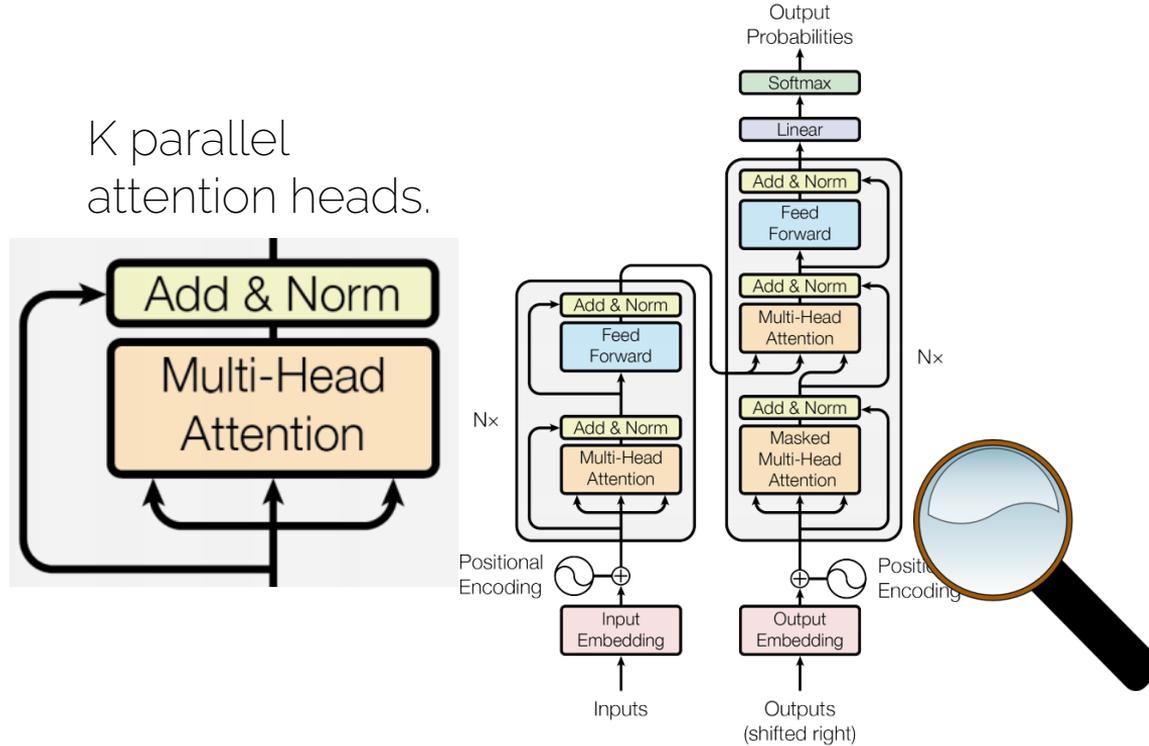


Values
V1
V2
V3
V4
V5

$$\text{softmax} \left(\frac{QK^T}{\sqrt{d_k}} \right)$$

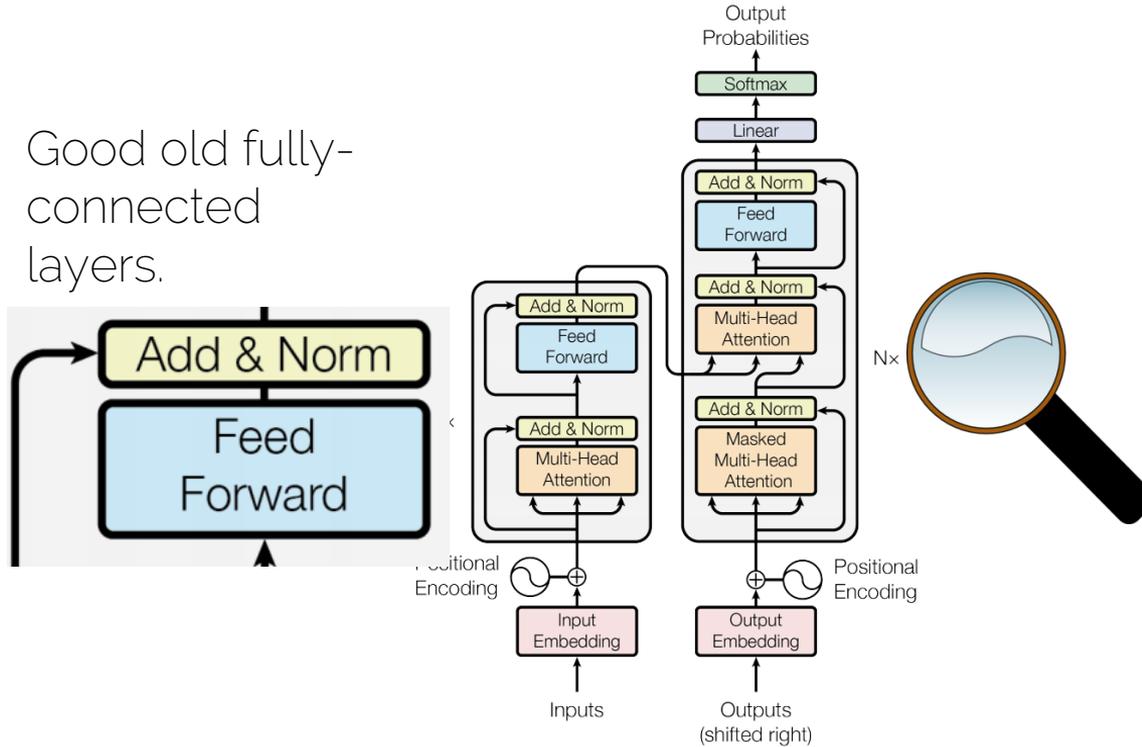
Selecting the value V where the network needs to attend..

Transformers – a closer look



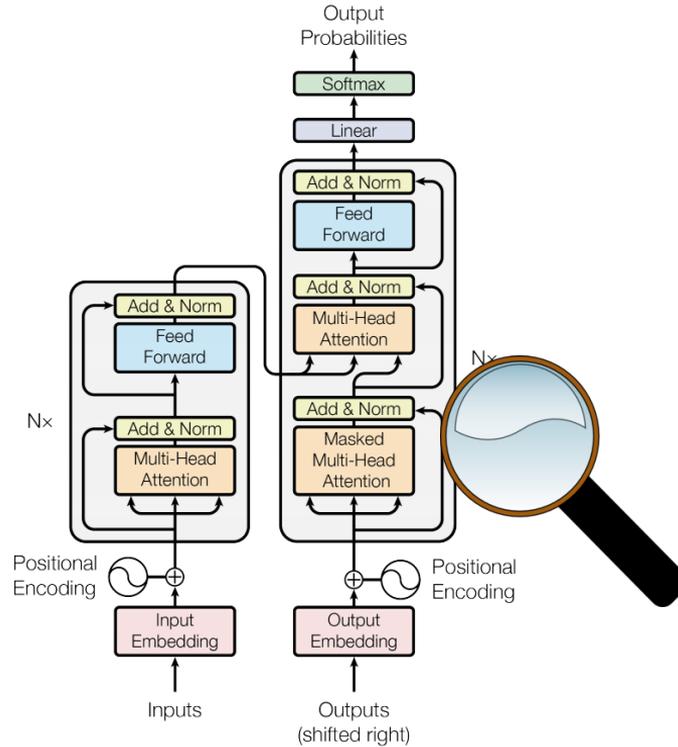
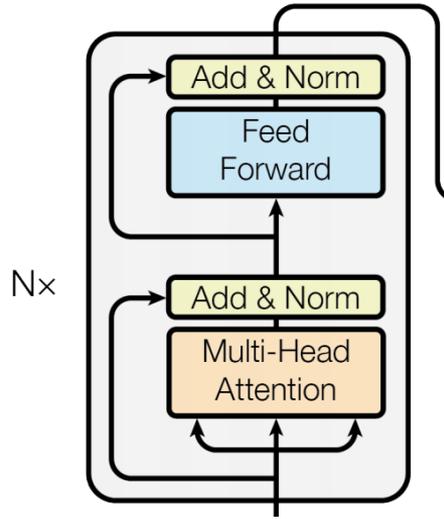
Transformers – a closer look

Good old fully-connected layers.



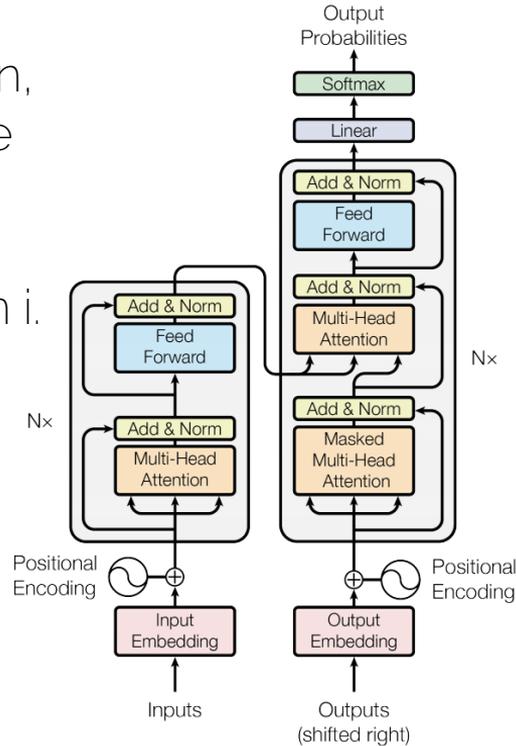
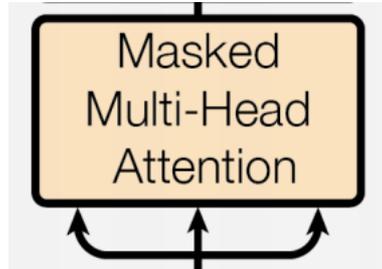
Transformers – a closer look

N layers of attention followed by FC



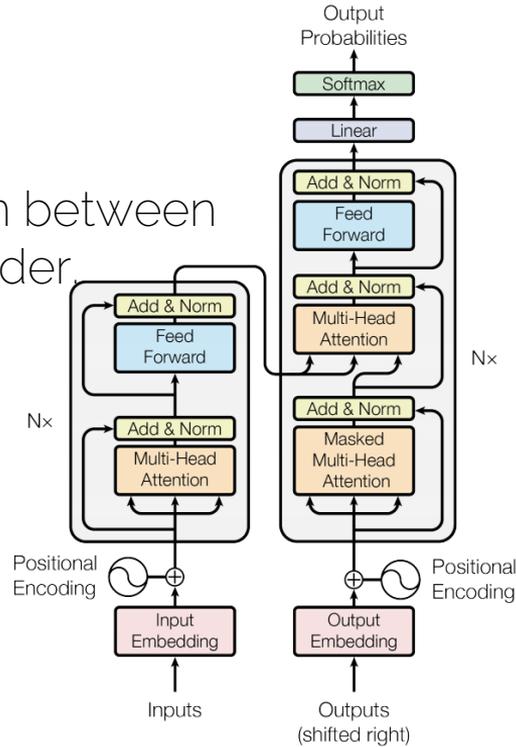
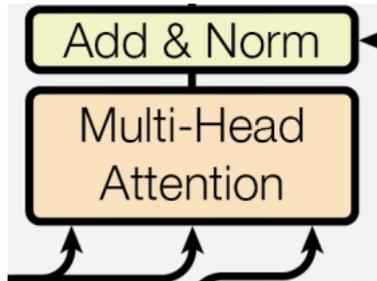
Transformers – a closer look

Same as multi-head attention, but masked. Ensures that the predictions for position i can depend only on the known outputs at positions less than i .



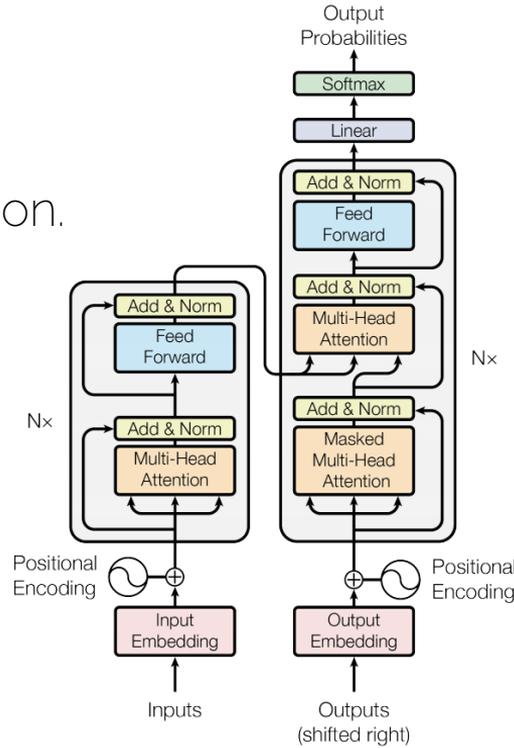
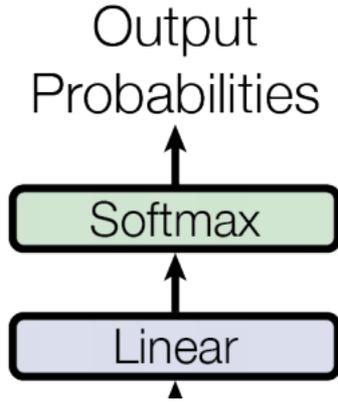
Transformers – a closer look

Multi-headed attention between encoder and the decoder



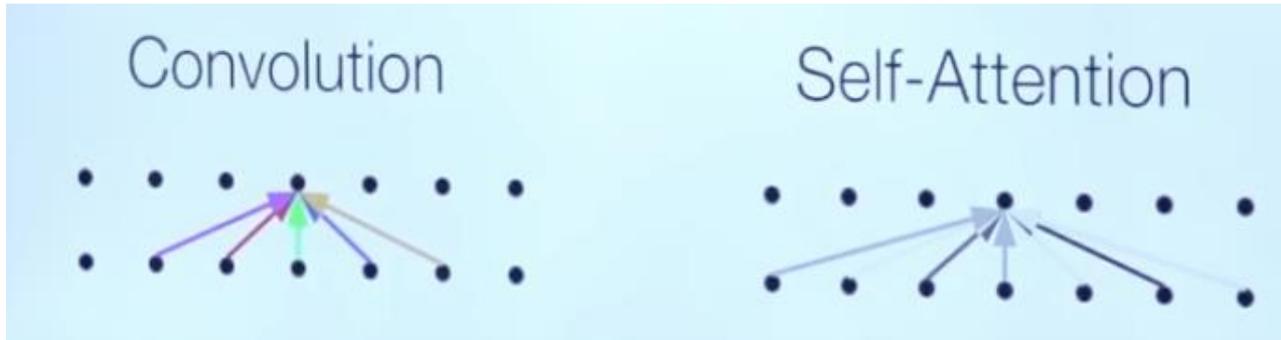
Transformers – a closer look

Projection and prediction.



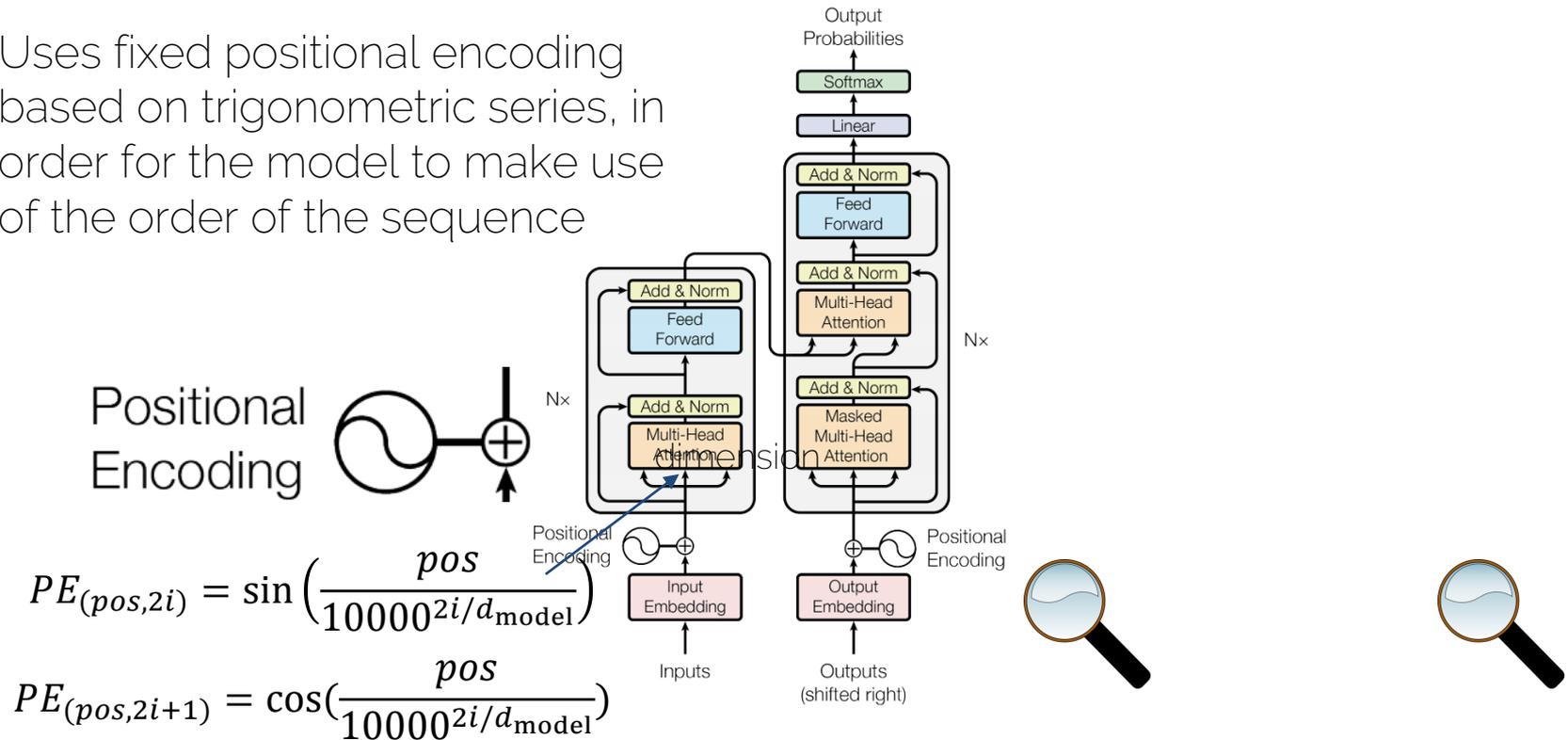
What is missing from self-attention?

- Convolution: a different linear transformation for each relative position. Allows you to distinguish what information came from where.
- Self-attention: a weighted average.

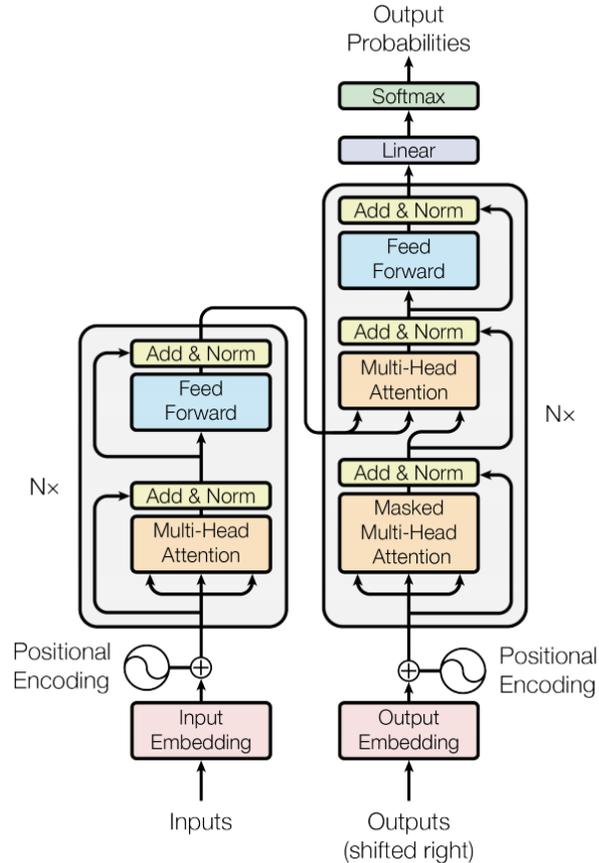


Transformers – a closer look

Uses fixed positional encoding based on trigonometric series, in order for the model to make use of the order of the sequence



Transformers – a final look



Self-attention: complexity

Layer Type	Complexity per Layer	Sequential Operations	Maximum Path Length
Self-Attention	$O(n^2 \cdot d)$	$O(1)$	$O(1)$
Recurrent	$O(n \cdot d^2)$	$O(n)$	$O(n)$
Convolutional	$O(k \cdot n \cdot d^2)$	$O(1)$	$O(\log_k(n))$
Self-Attention (restricted)	$O(r \cdot n \cdot d)$	$O(1)$	$O(n/r)$

where n is the sequence length, d is the representation dimension, k is the convolutional kernel size, and r is the size of the neighborhood.

Self-attention: complexity

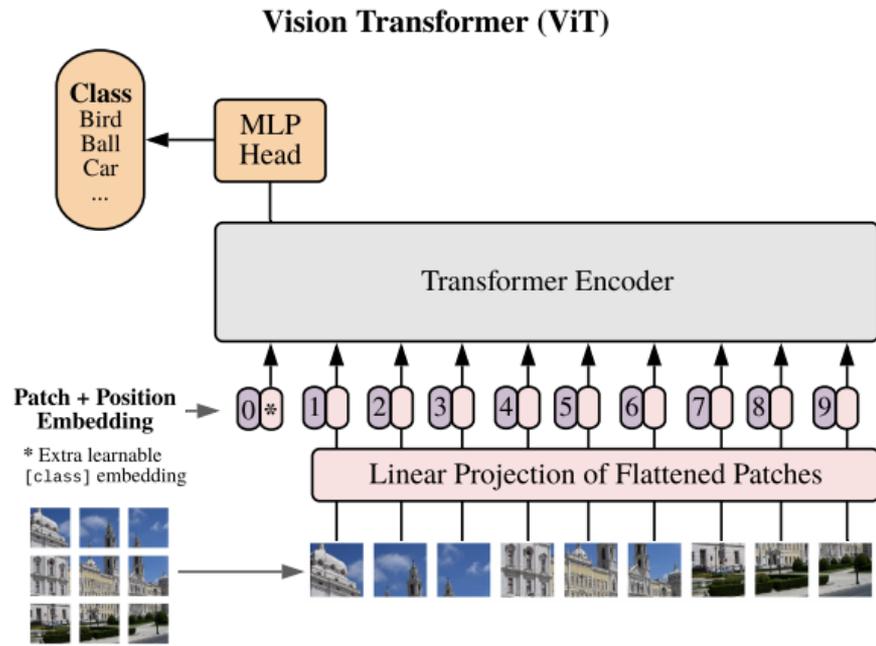
Layer Type	Complexity per Layer	Sequential Operations	Maximum Path Length
Self-Attention	$O(n^2 \cdot d)$	$O(1)$	$O(1)$
Recurrent	$O(n \cdot d^2)$	$O(n)$	$O(n)$
Convolutional	$O(k \cdot n \cdot d^2)$	$O(1)$	$O(\log_k(n))$
Self-Attention (restricted)	$O(r \cdot n \cdot d)$	$O(1)$	$O(n/r)$

where n is the sequence length, d is the representation dimension, k is the convolutional kernel size, and r is the size of the neighborhood.

Considering that most sentences have a smaller dimension than the representation dimension (in the paper, it is 512), self-attention is very efficient.

Transformers for Images

- Vision Transformers (ViTs)



*Dosovitskiy et al. "An image is worth 16x16 words: Transformers for image recognition at scale", ICLR 2020.

Transformers - summary

- Significantly improved SOTA in machine translation
- Launched a new deep-learning revolution
- Building block of NLP models like Gemini (Google) or GPT/ChatGPT (OpenAI)

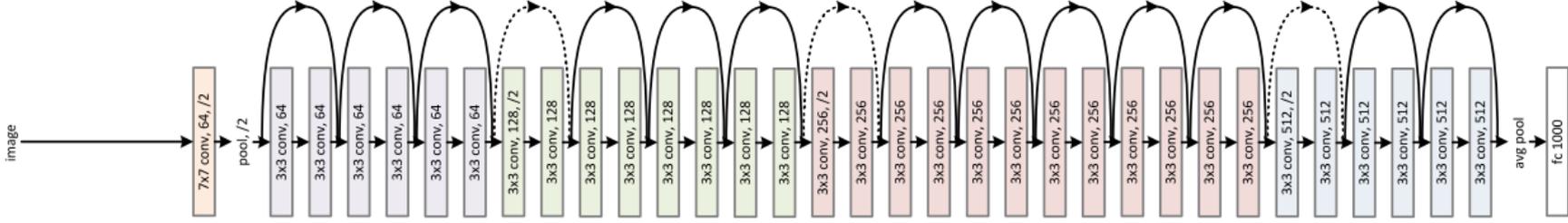
- And eventually made its way to computer vision (and other related fields)*

*Dosovitskiy et al. "An image is worth 16x16 words: Transformers for image recognition at scale", ICLR 2020.

Transfer Learning

ResNet

34-layer residual



ResNet-152:
60M parameters

Transfer Learning

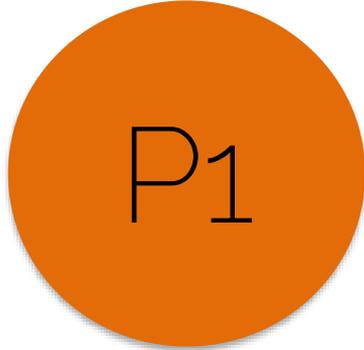
- Training your own model can be difficult with limited data and other resources

e.g.,

- It is a laborious task to manually annotate your own training dataset
- Why not reuse already pre-trained models?

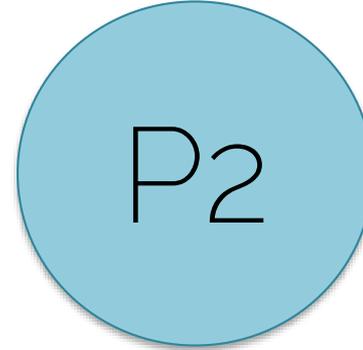
Transfer Learning

Distribution

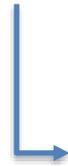


Large dataset

Distribution



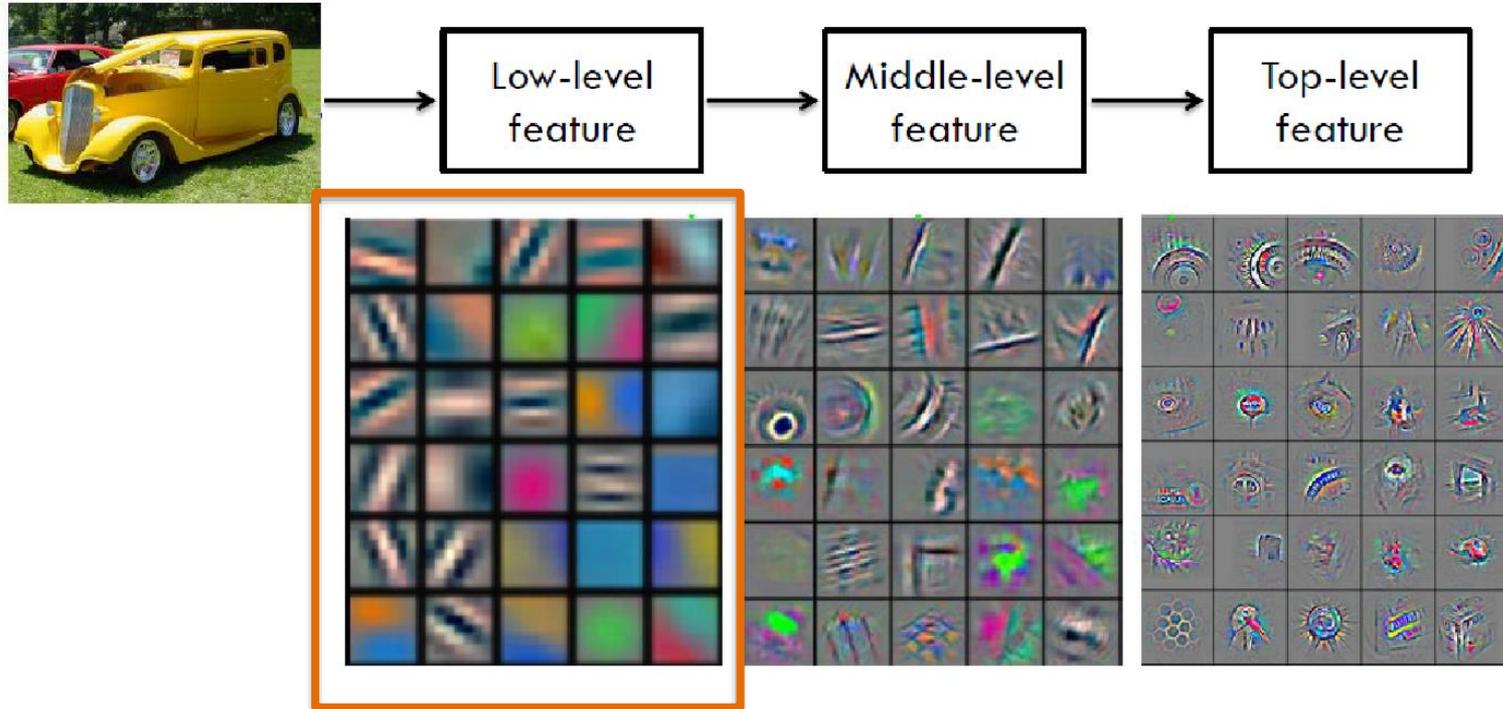
Small dataset



Use what has been
learned for another
setting



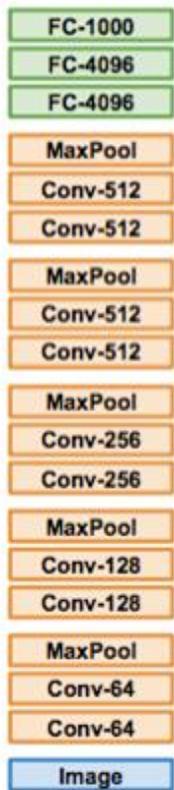
Transfer Learning for Images



[Zeiler et al., ECCV'14] Visualizing and Understanding Convolutional Networks

Trained on
ImageNet

Transfer Learning

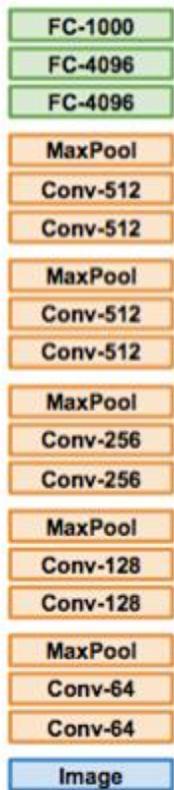


Feature
extraction

[Donahue et al., ICML'14] DeCAF,
[Razavian et al., CVPRW'14] CNN Features off-the-shelf

Transfer Learning

Trained on
ImageNet



Decision layers

Parts of an object (wheel, window)

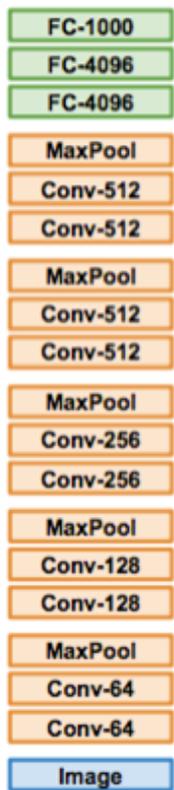
Simple geometrical shapes (circles, etc)

Edges

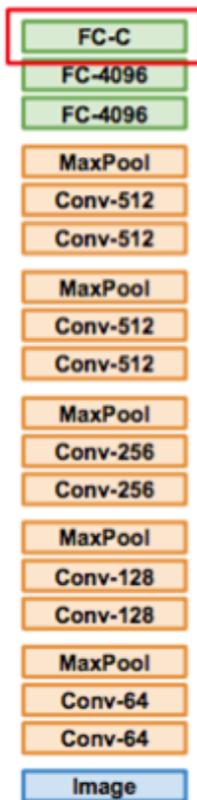
[Donahue et al., ICML'14] DeCAF,
[Razavian et al., CVPRW'14] CNN Features off-the-shelf

Trained on
ImageNet

Transfer Learning



TRAIN



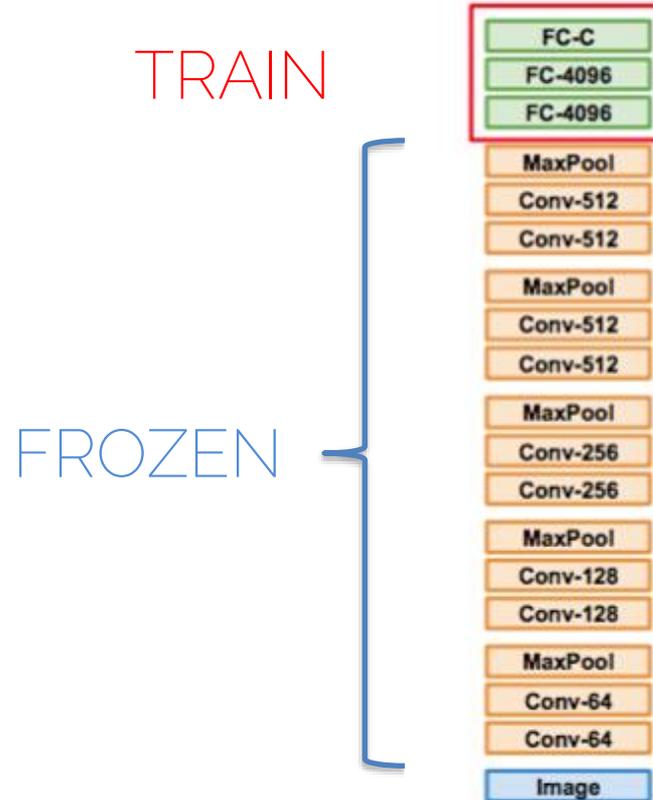
New dataset
with C classes

FROZEN

[Donahue et al., ICML'14] DeCAF,
[Razavian et al., CVPRW'14] CNN Features off-the-shelf

Transfer Learning

If the dataset is big
enough train more
layers
If not, train fewer
layers



When Transfer Learning Makes Sense

- When task T_1 and T_2 have the same input (e.g. an RGB image)
- When you have more data for task T_1 than for task T_2
- When the low-level features for T_1 could be useful to learn T_2

Representation Learning

Learning Good Features

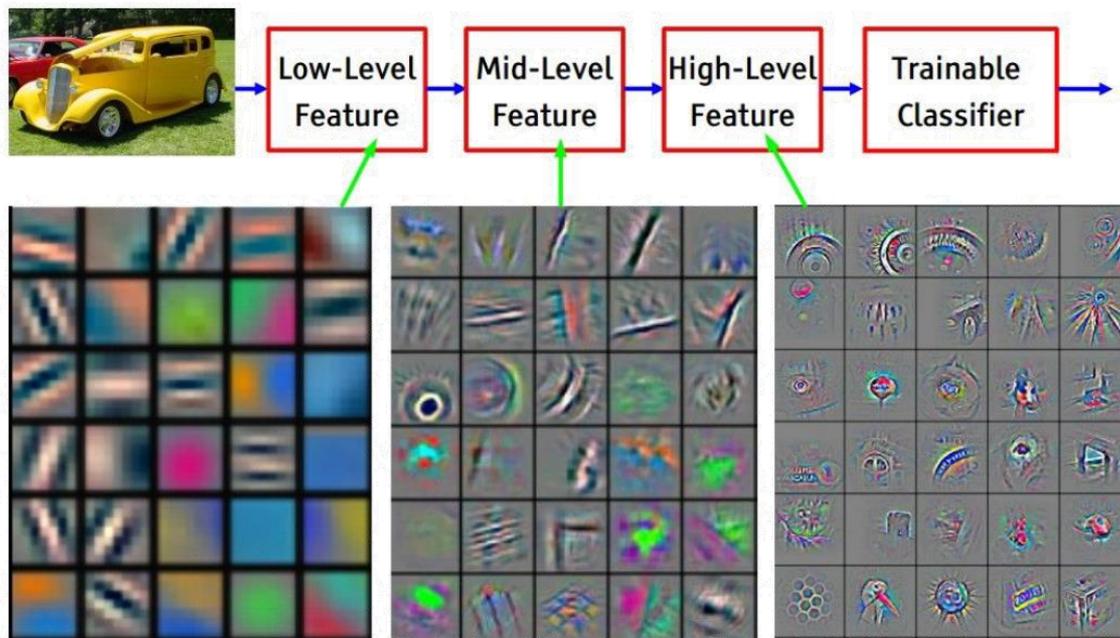
- Good features are essential for successful machine learning
- (Supervised) deep learning depends on training data used: input/target labels
- Change in inputs (noise, irregularities, etc.) can result in drastically different results

Representation Learning

- Allows for discovery of representations required for various tasks
- Deep representation learning: model maps input \mathbf{X} to output \mathbf{Y}

Deep Representation Learning

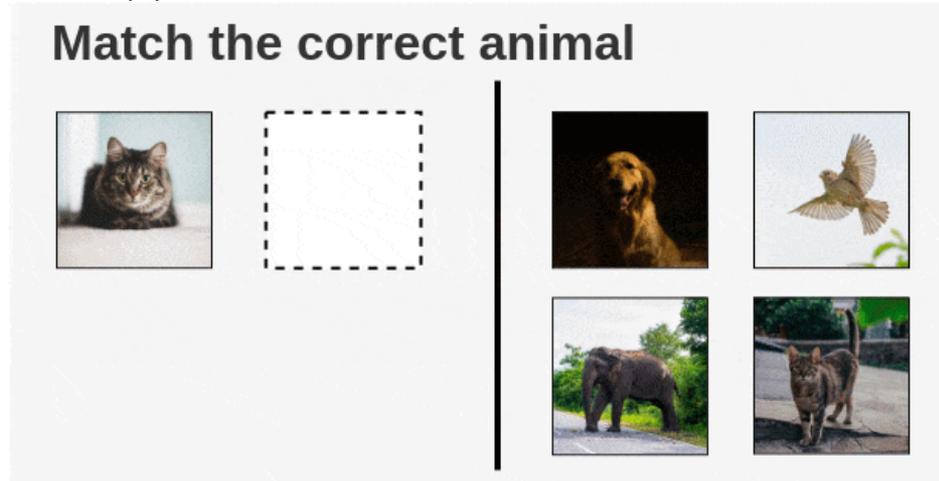
- Intuitively, deep networks learn multiple levels of abstraction



Feature visualization of convolutional net trained on ImageNet from [Zeiler & Fergus 2013]

How to Learn Good Features?

- Determine desired feature invariances
- Teach machines to distinguish between similar and dissimilar things

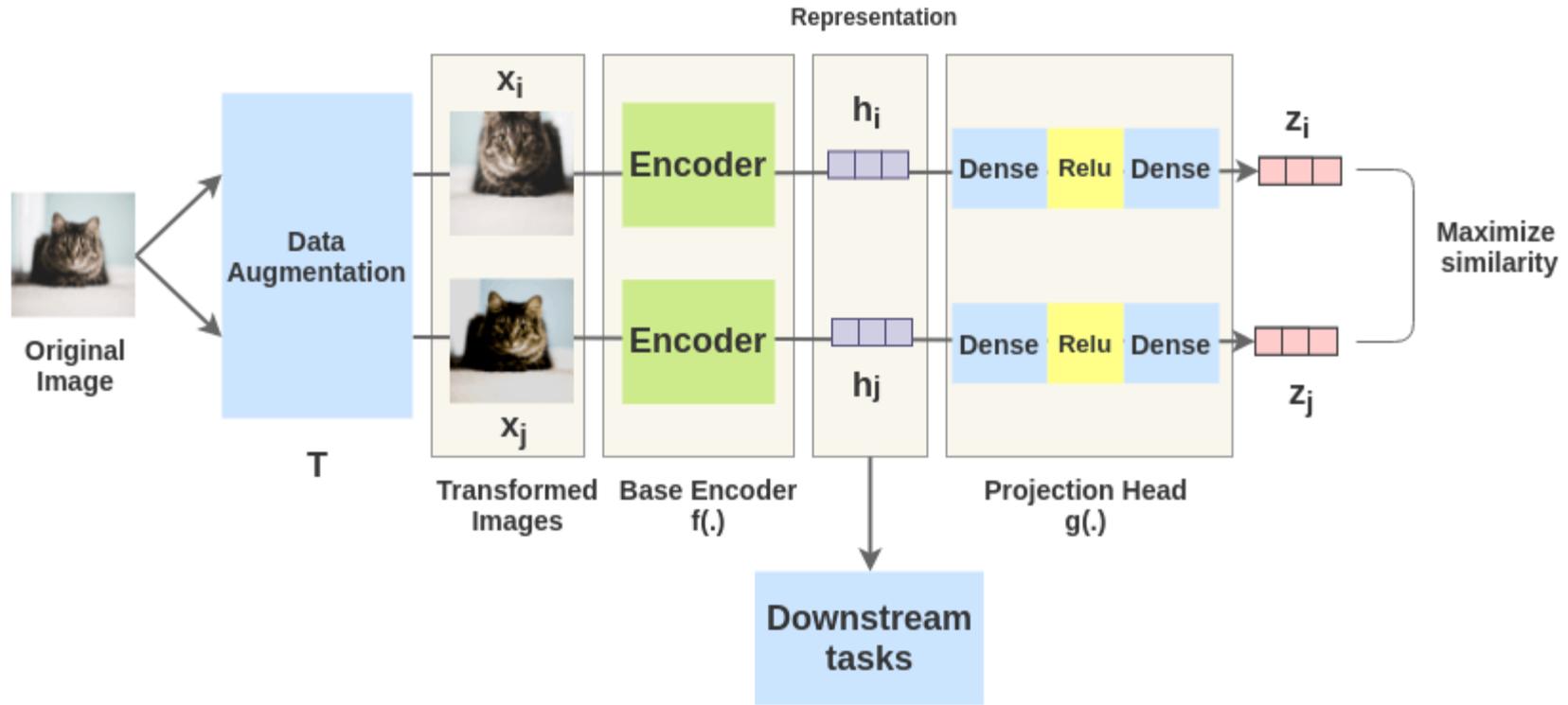


<https://amitness.com/2020/03/illustrated-simclr/>

Self-Supervised Representation Learning

- Why not just supervised learning?
 - Relies heavily on labeled data
 - Learns task-specific features
- Want general useful internal features that are transferable to many tasks

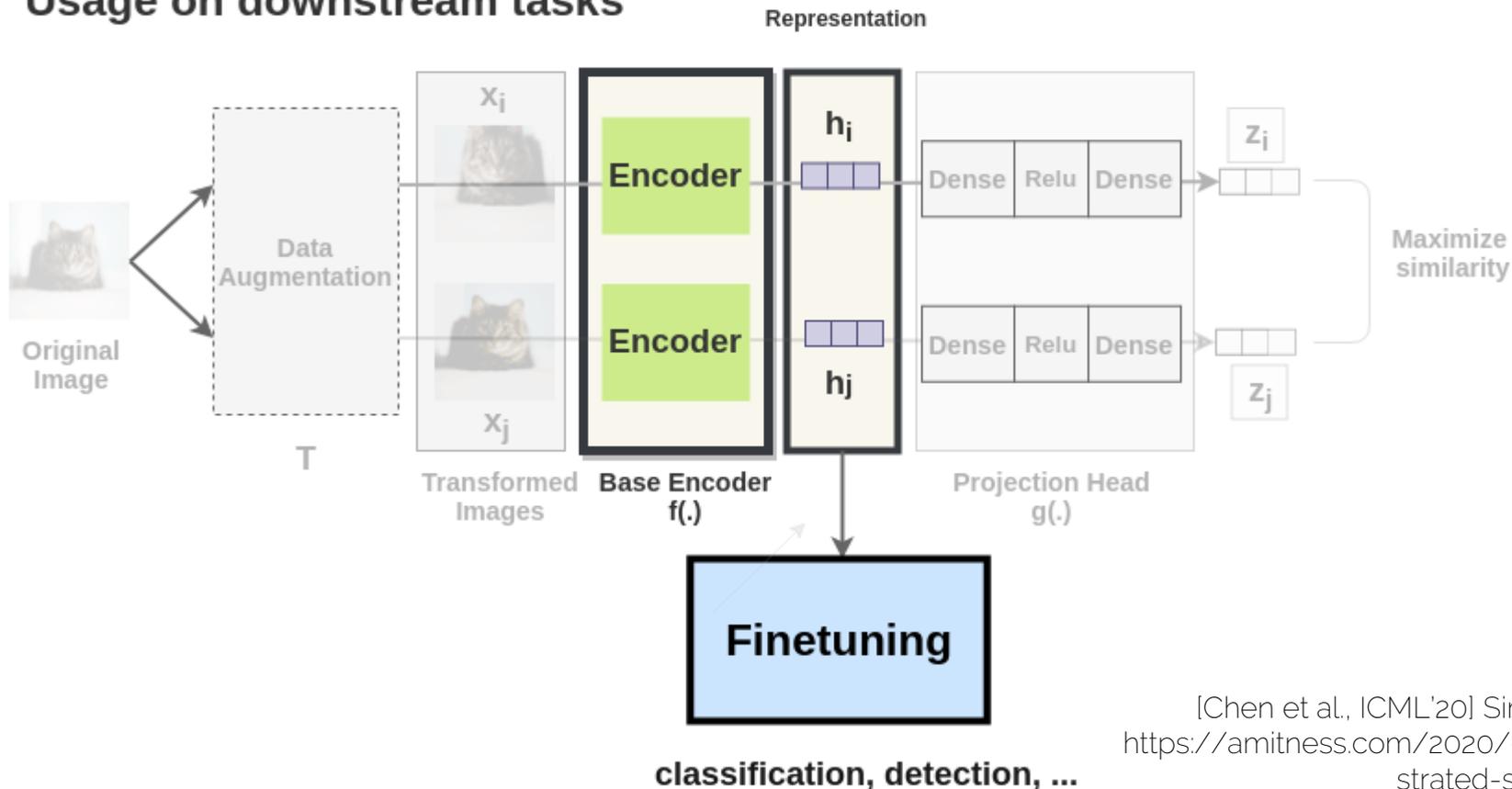
How to Learn Good Features?



[Chen et al., ICML'20] SimCLR,
<https://amitniss.com/2020/03/illustrated-simclr/>

Apply to Downstream Tasks

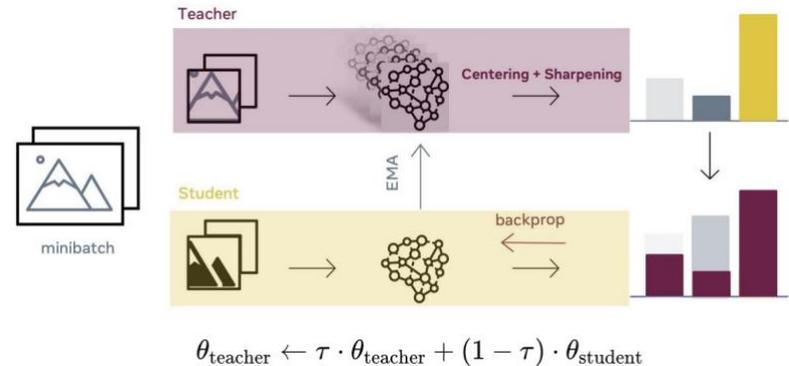
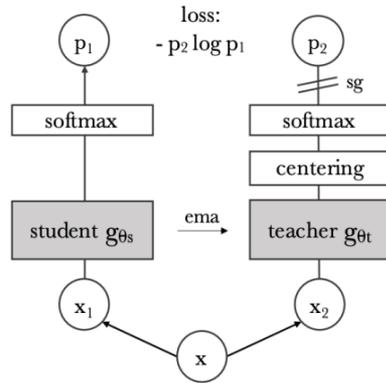
Usage on downstream tasks



[Chen et al., ICML'20] SimCLR,
<https://amitness.com/2020/03/illustrated-simclr/>

DINO

- Self-supervised learning
- Trains a vision transformer (ViT) to produce meaningful representations using self-distillation w/ no labels

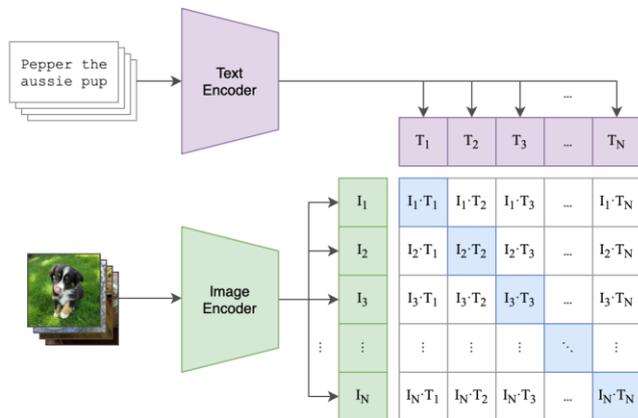


[Caron et al., ICCV'21] Emerging Properties in Self-Supervised Vision Transformers

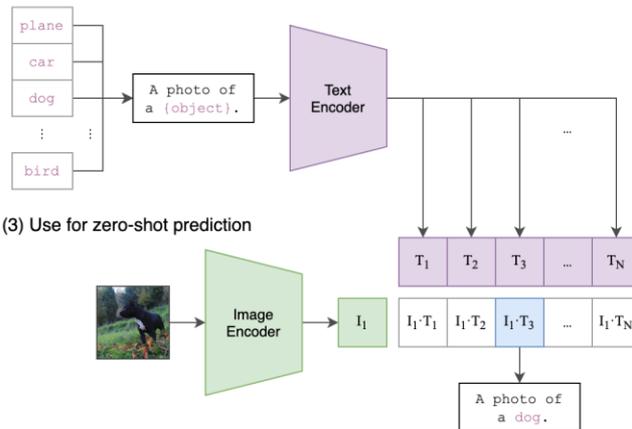
CLIP: Contrastive Language-Image Pre-training

- Connects images and text by training on massive dataset of (image, caption) pairs scraped from the internet

(1) Contrastive pre-training



(2) Create dataset classifier from label text



(3) Use for zero-shot prediction

Transfer & Representation Learning

- Transfer learning can be done via representation learning
- Effectiveness of representation learning often demonstrated by transfer learning performance (but also other factors, e.g., smoothness of the manifold)

See you next time!