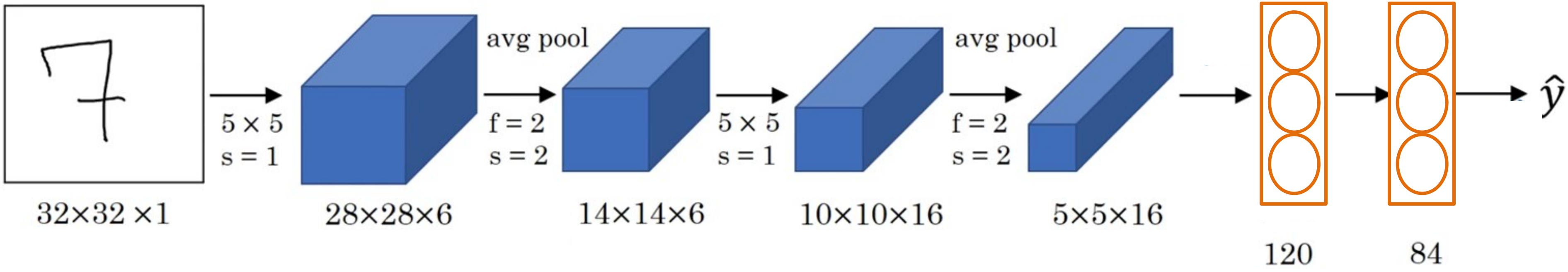
Lecture 10 Recap

LeNet

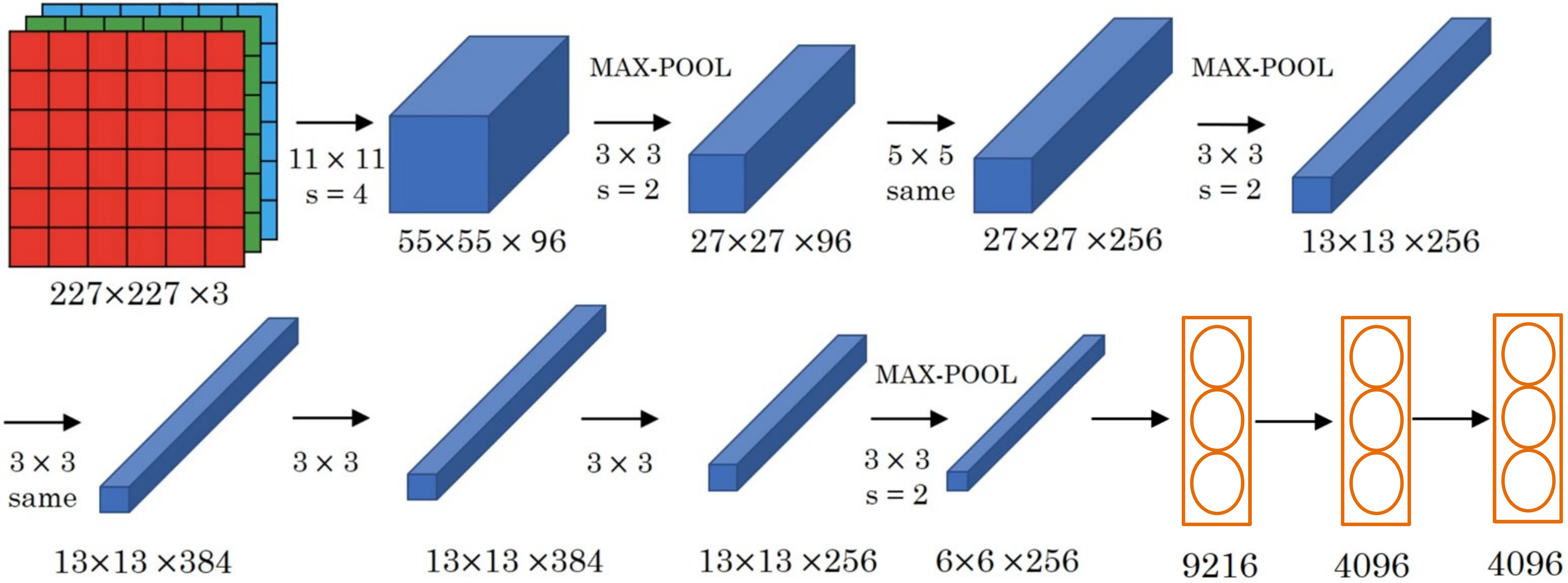
- Digit recognition: 10 classes

60k parameters



- Conv -> Pool -> Conv -> Pool -> Conv -> FC
- As we go deeper: Width, Height ↓ Number of Filters ↑

AlexNet



- Softmax for 1000 classes

[Krizhevsky et al., ANIPS'12] AlexNet

VGGNet

- Striving for **simplicity**
  - Conv -> Pool -> Conv -> Pool -> Conv -> FC
  - Conv=3x3, s=1, same; Maxpool=2x2, s=2
- As we go deeper: Width, Height ↓ Number of Filters ↑
- Called VGG-16: 16 layers that have weights

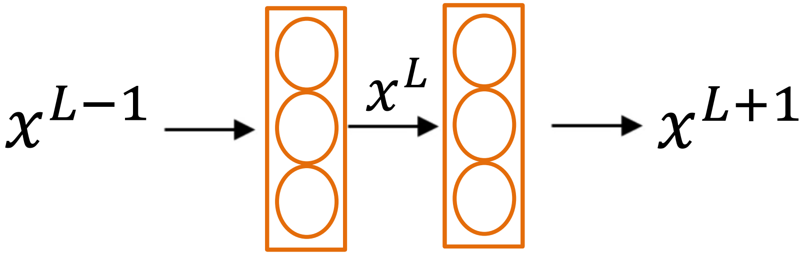
$\boxed{\text{138M parameters}}$

- Large but simplicity makes it appealing

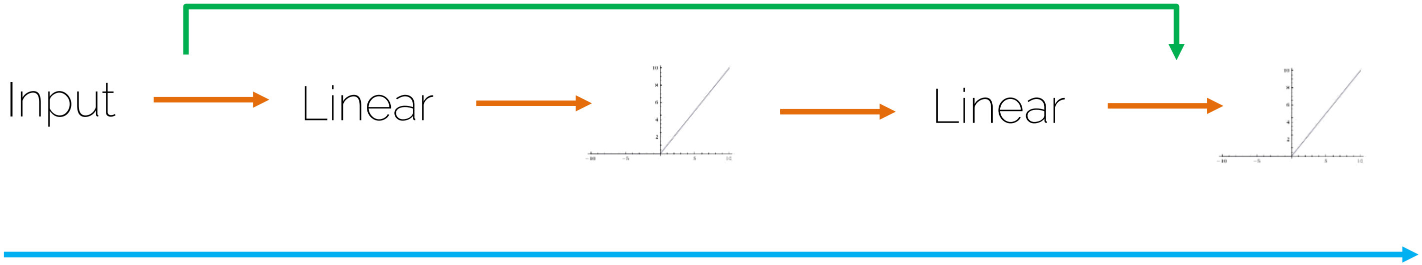
[Simonyan et al., ICLR'15] VGGNet

Residual Block

- Two layers

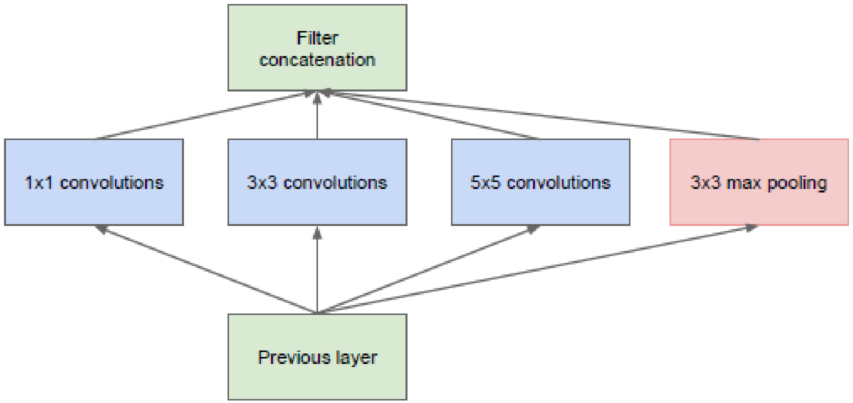


$$x^{L+1} = f(W^{L+1}x^{L} + b^{L+1} + x^{L-1})$$

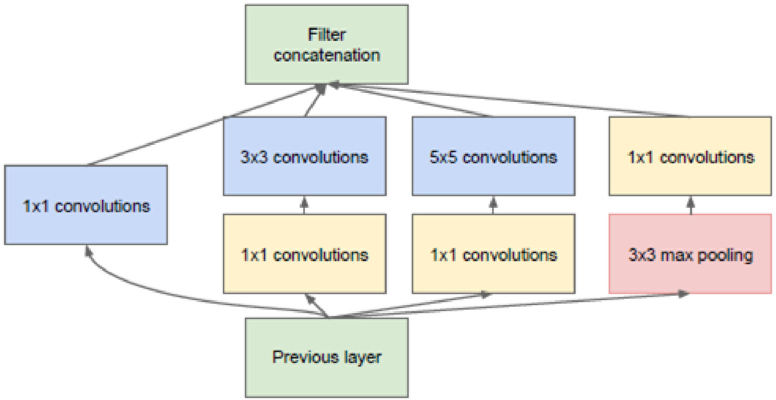


$$x^{L+1} = f(W^{L+1}x^{L} + b^{L+1})$$

Inception Layer



(a) Inception module, naïve version



(b) Inception module with dimensionality reduction

[Szegedy et al., CVPR'15] GoogleNet

Lecture 11

Transfer Learning

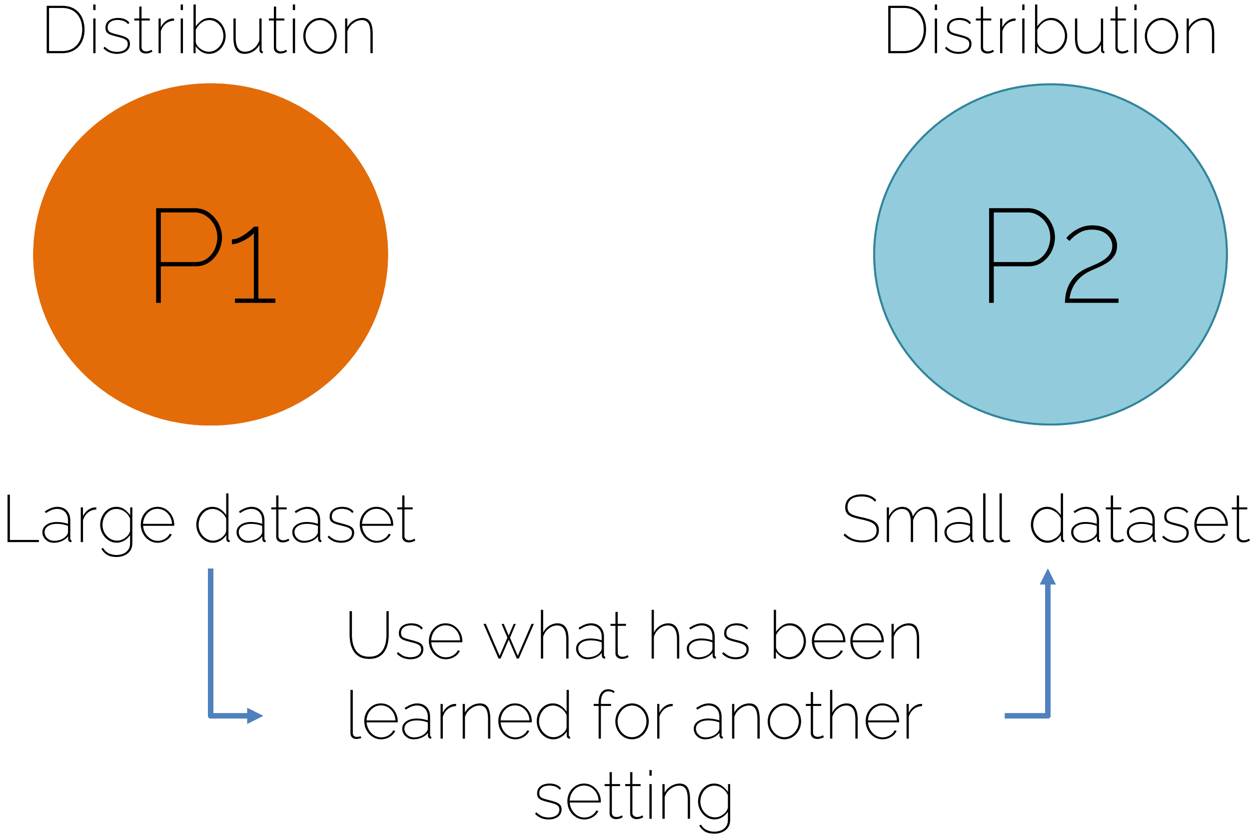
Transfer Learning

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

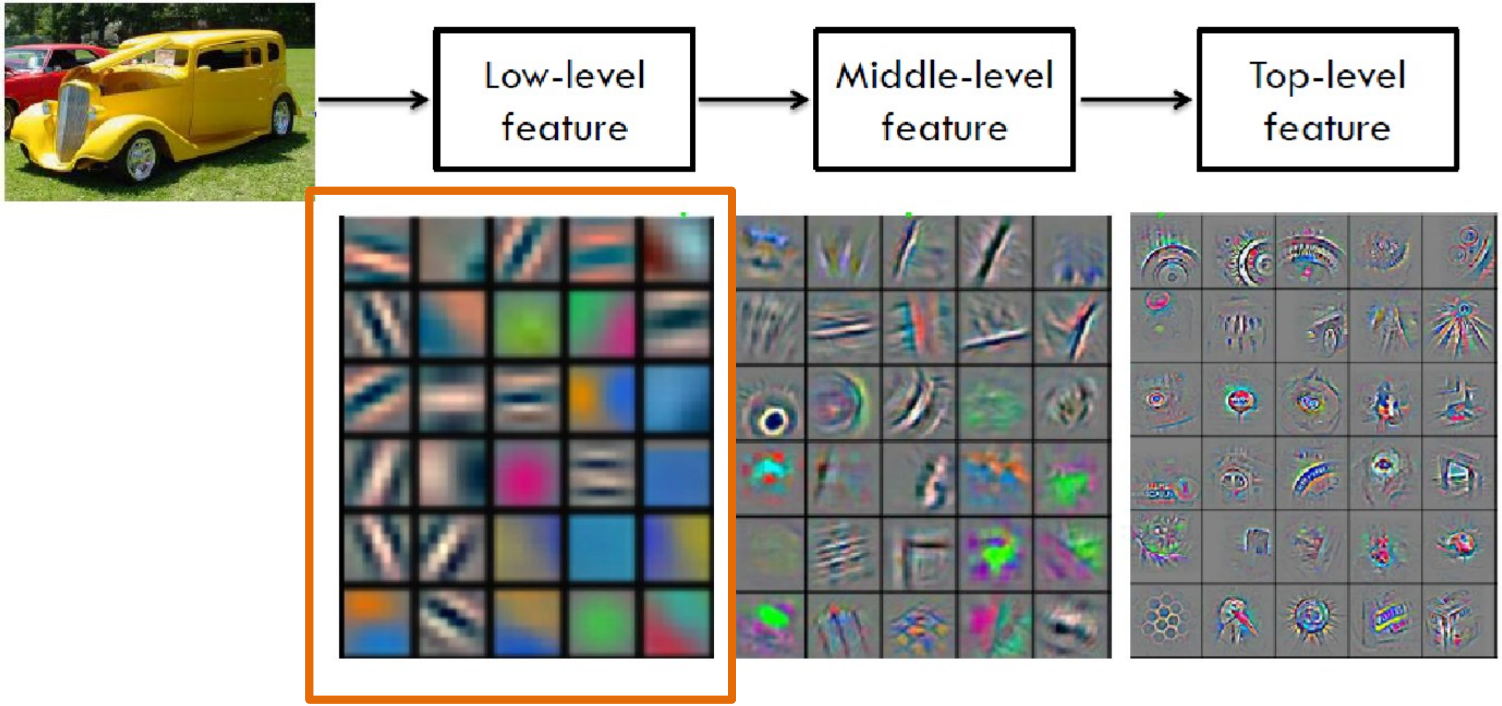
e.g.,

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

Transfer Learning



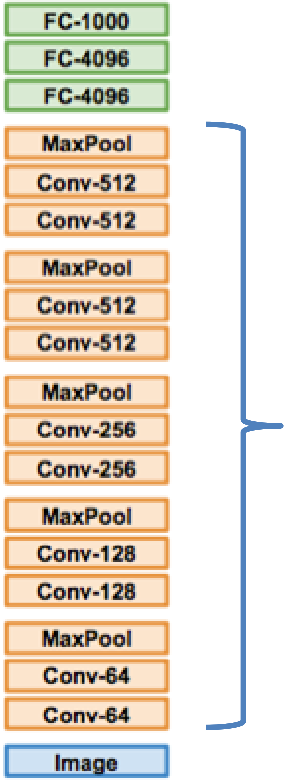
Transfer Learning for Images



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

Transfer Learning

Trained on ImageNet

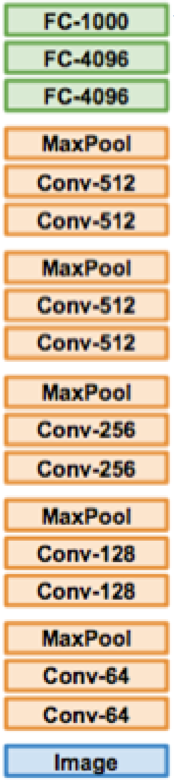


Feature extraction

[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)

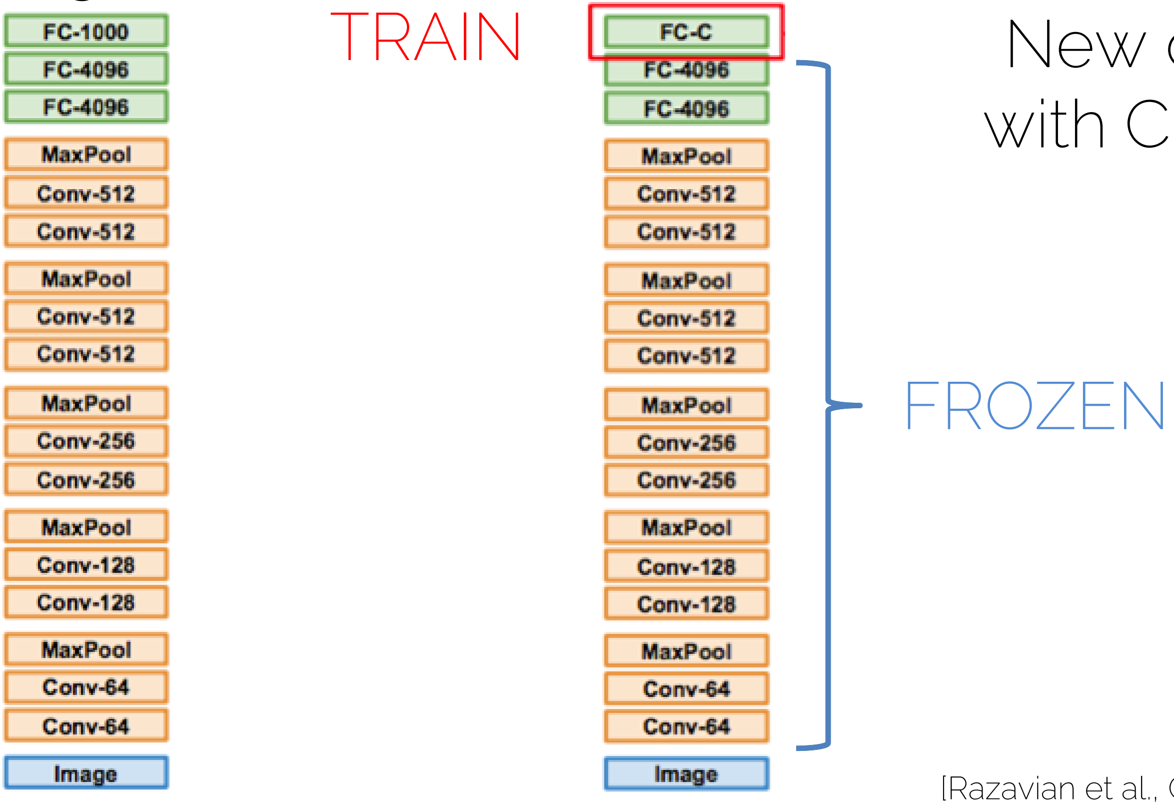
Edges

4

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

Transfer Learning

Trained on ImageNet

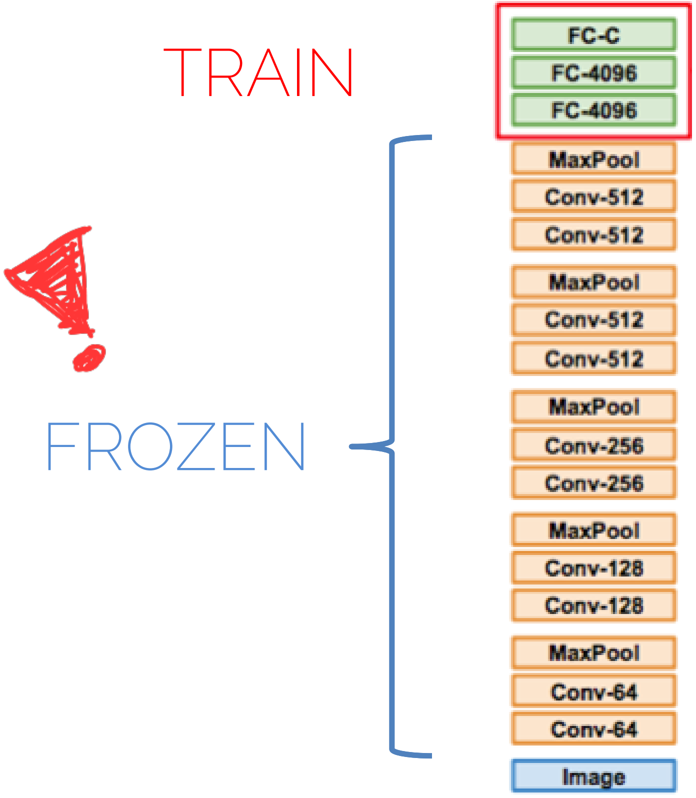


New dataset with C classes

[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 with a low learning rate



When Transfer Learning Makes Sense

- When task T1 and T2 have the same input (e.g. an RGB image)
- When you have more data for task T1 than for task T2
- When the low-level features for T1 could be useful to learn T2

Now you are:

- Ready to perform image classification on any dataset
- Ready to design your own architecture
- Ready to deal with other problems such as semantic segmentation (Fully Convolutional Network)

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

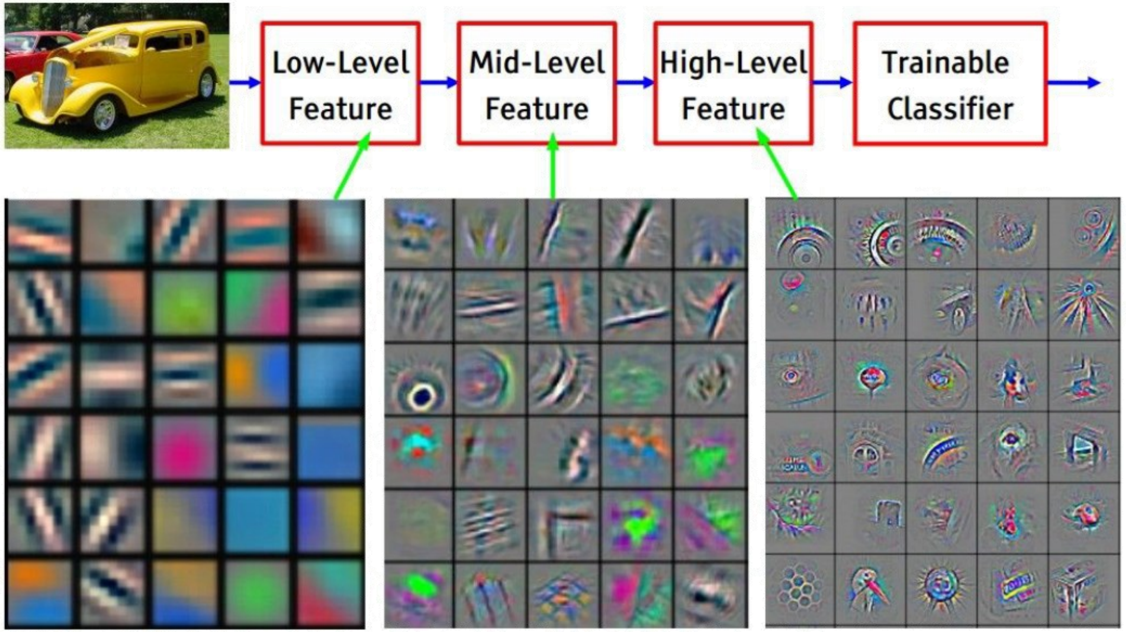
Representation Learning

- Allows for discovery of representations required for various tasks

- Deep representation learning: model maps input $\boldsymbol{X}$ to output $\boldsymbol{Y}$

Deep Representation Learning

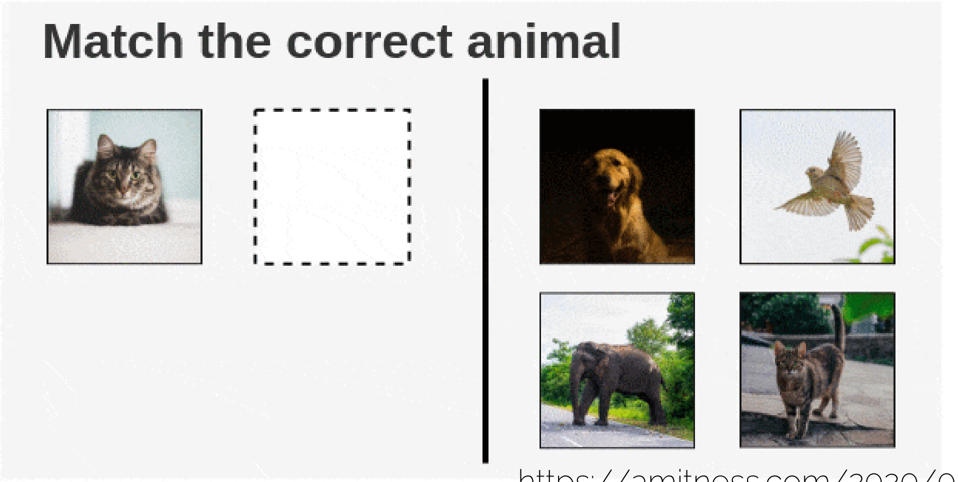
- Intuitively, deep networks learn multiple levels of abstraction



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

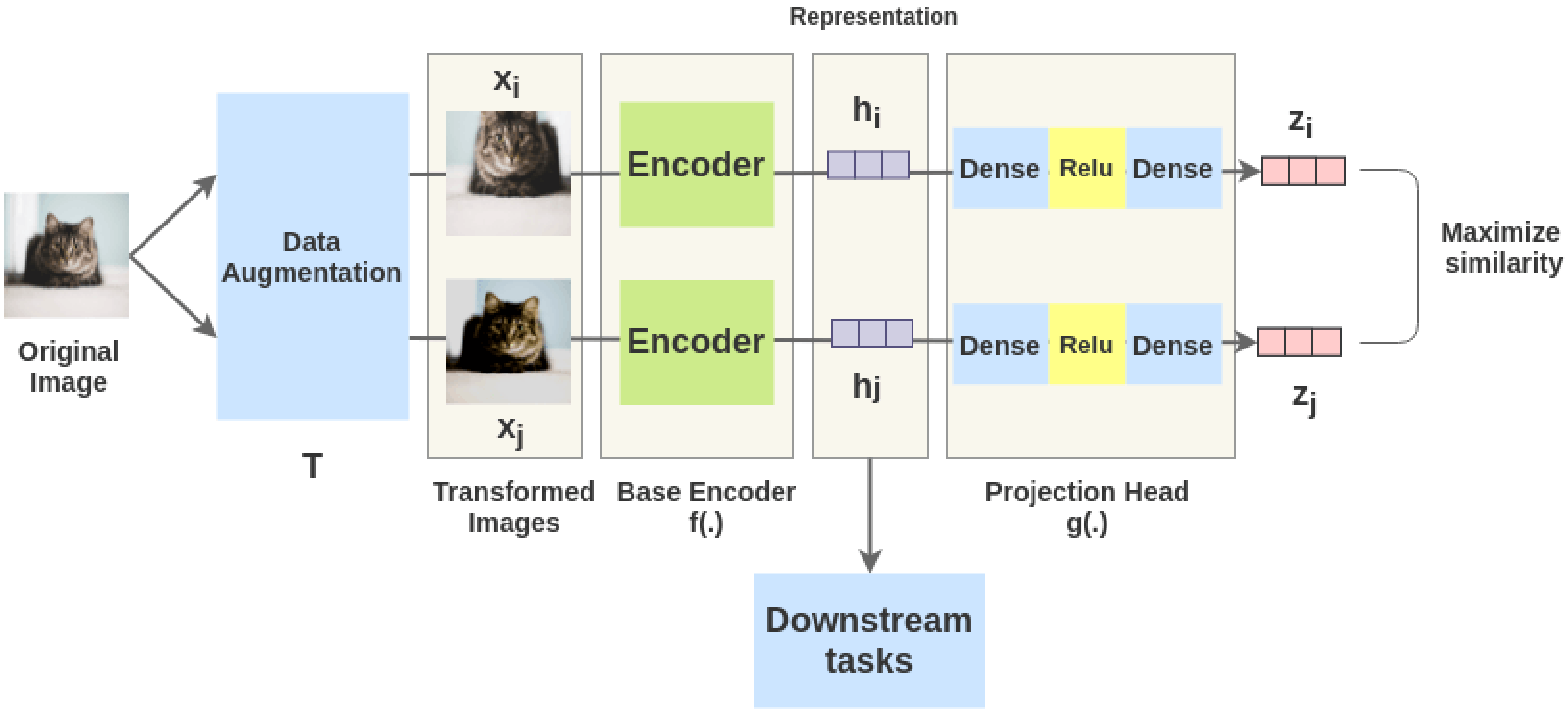
How to Learn Good Features?

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



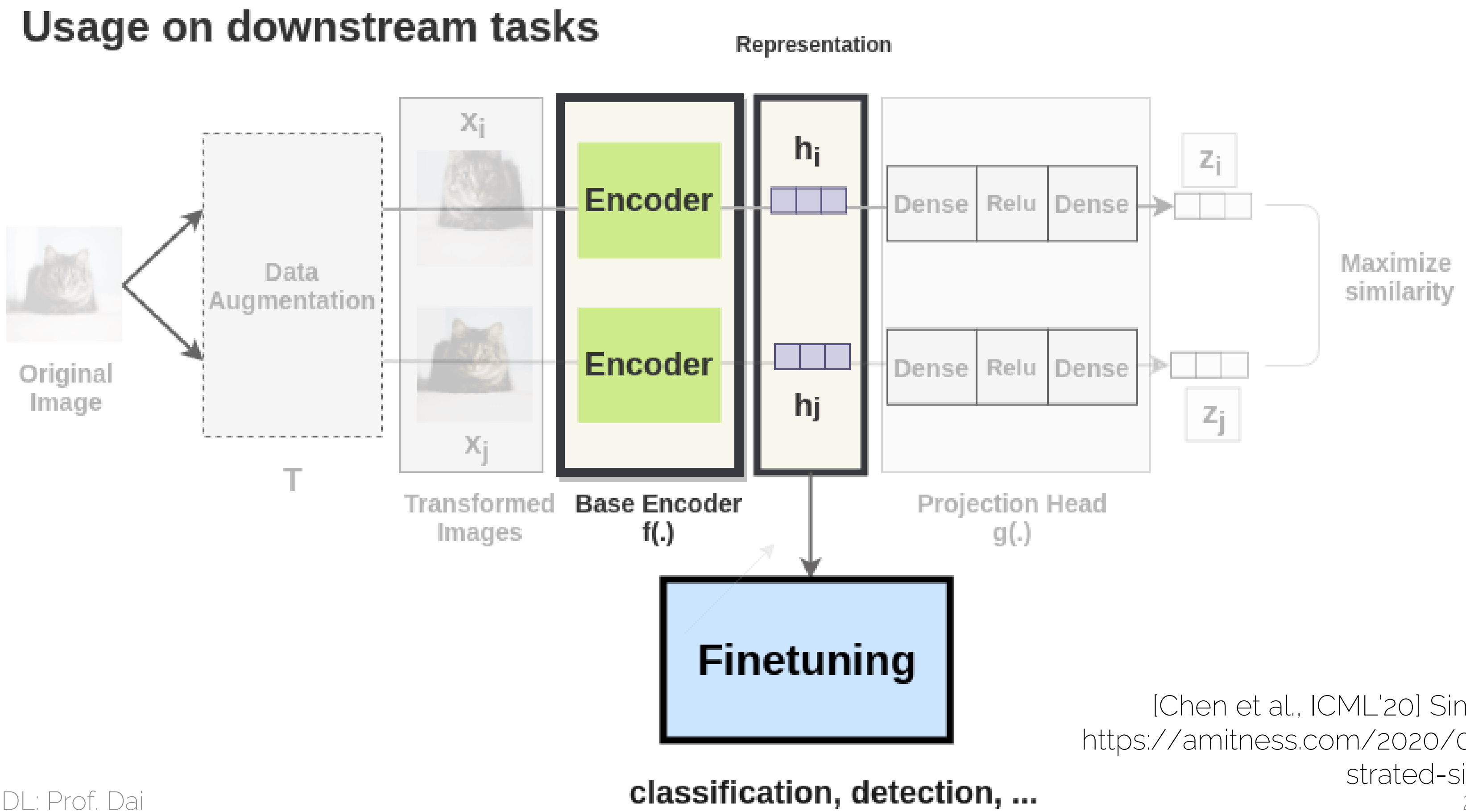
https://amitness.com/2020/03/illustrated-simclr/

How to Learn Good Features?



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

Apply to Downstream Tasks



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

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)

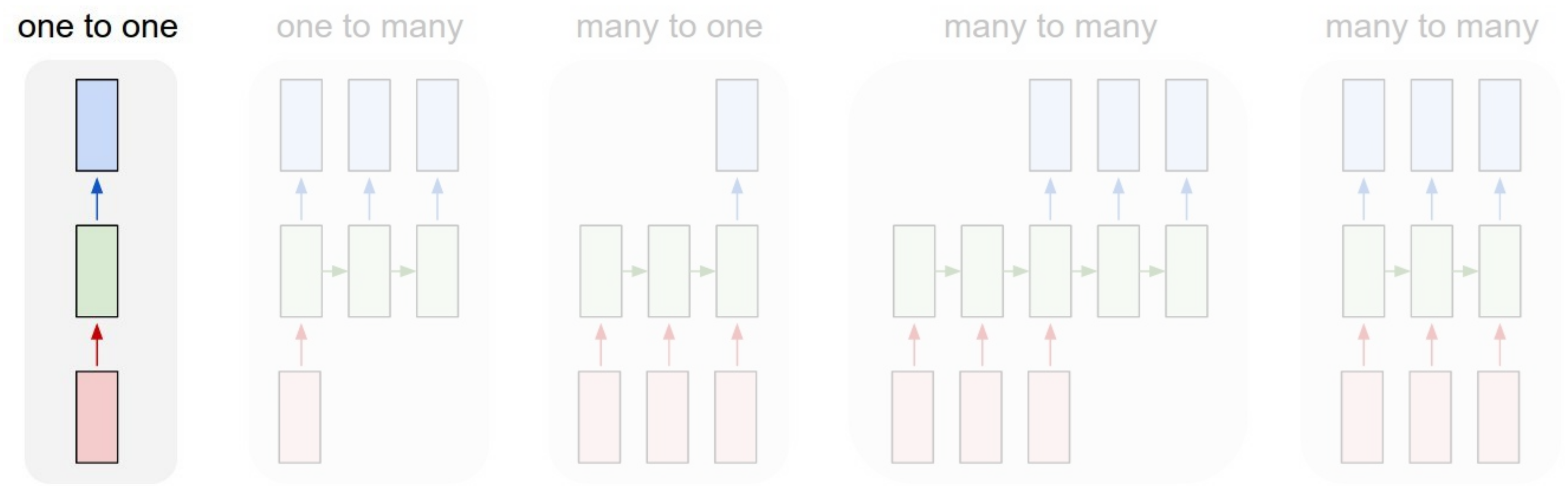
Recurrent Neural Networks

Processing Sequences

- Recurrent neural networks process sequence data

- Input/output can be sequences

RNNs are Flexible



Classical neural networks for image classification

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

RNNs are Flexible

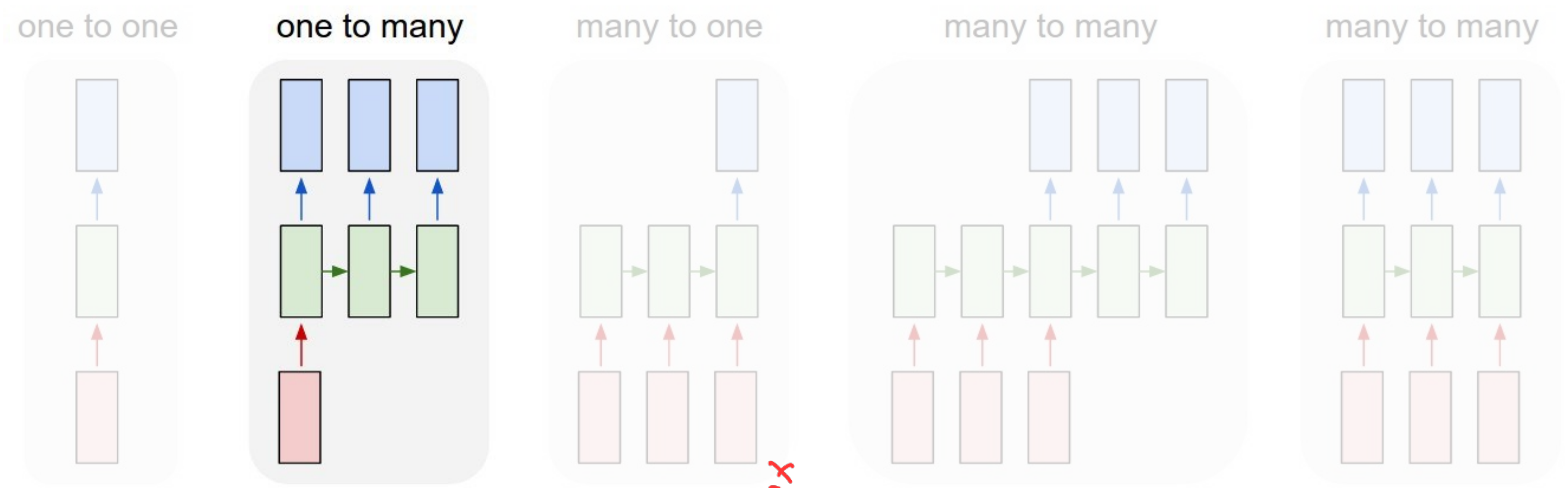


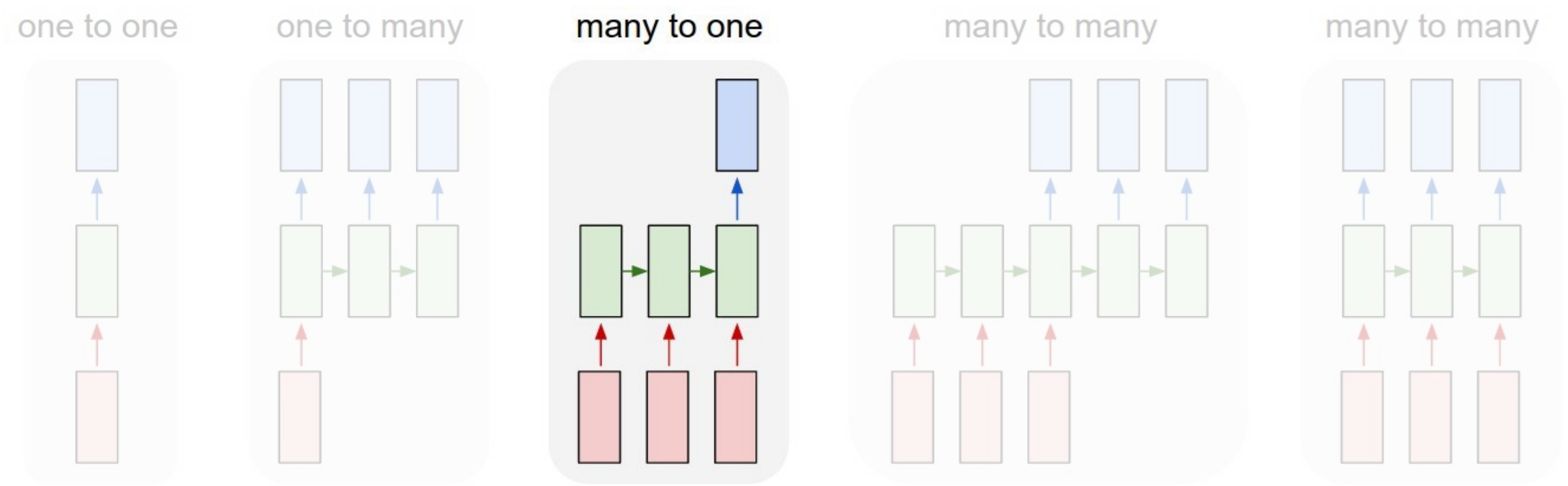
Image captioning

→ Vbo event

3 caption for image

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

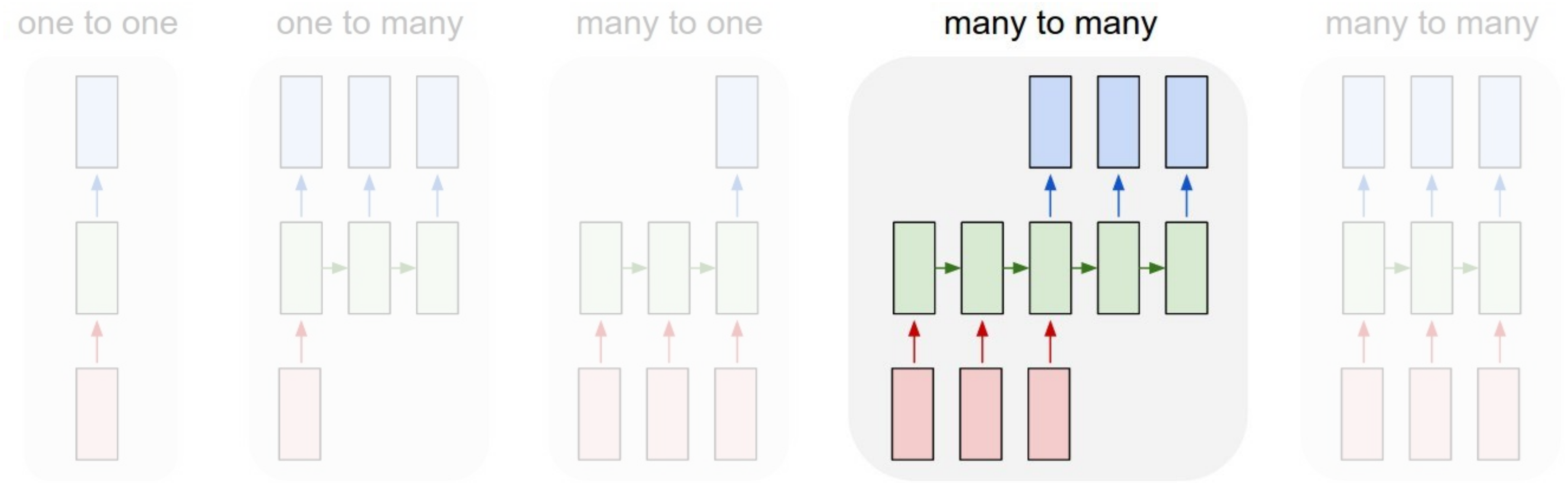
RNNs are Flexible



Language recognition

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

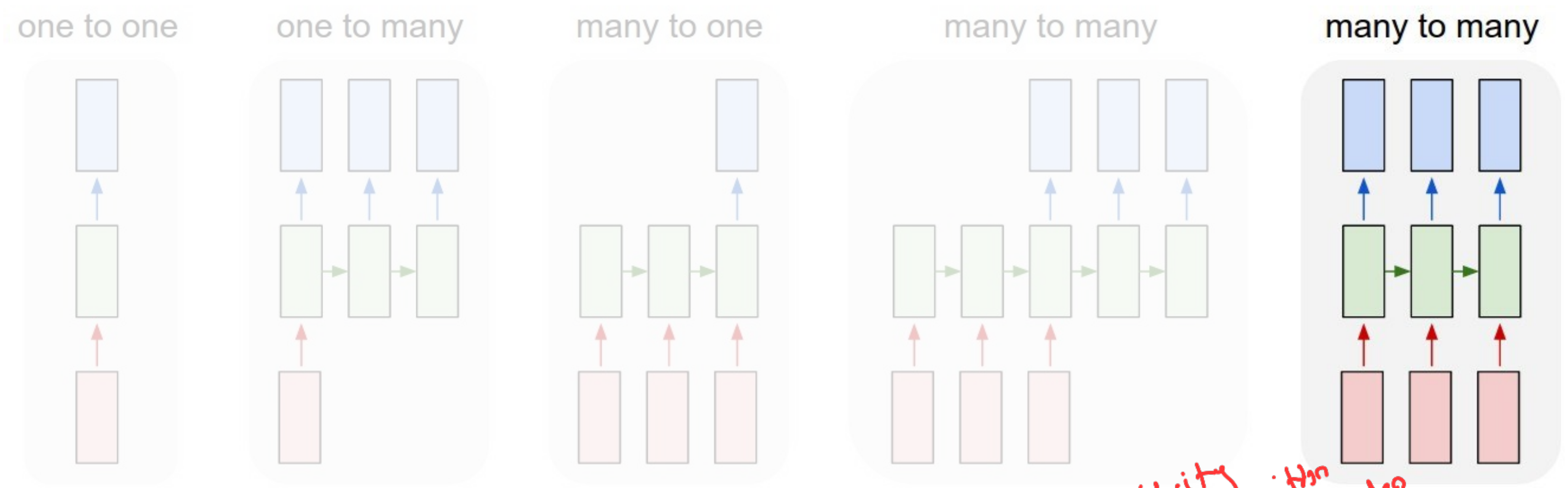
RNNs are Flexible



Machine translation

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

RNNs are Flexible

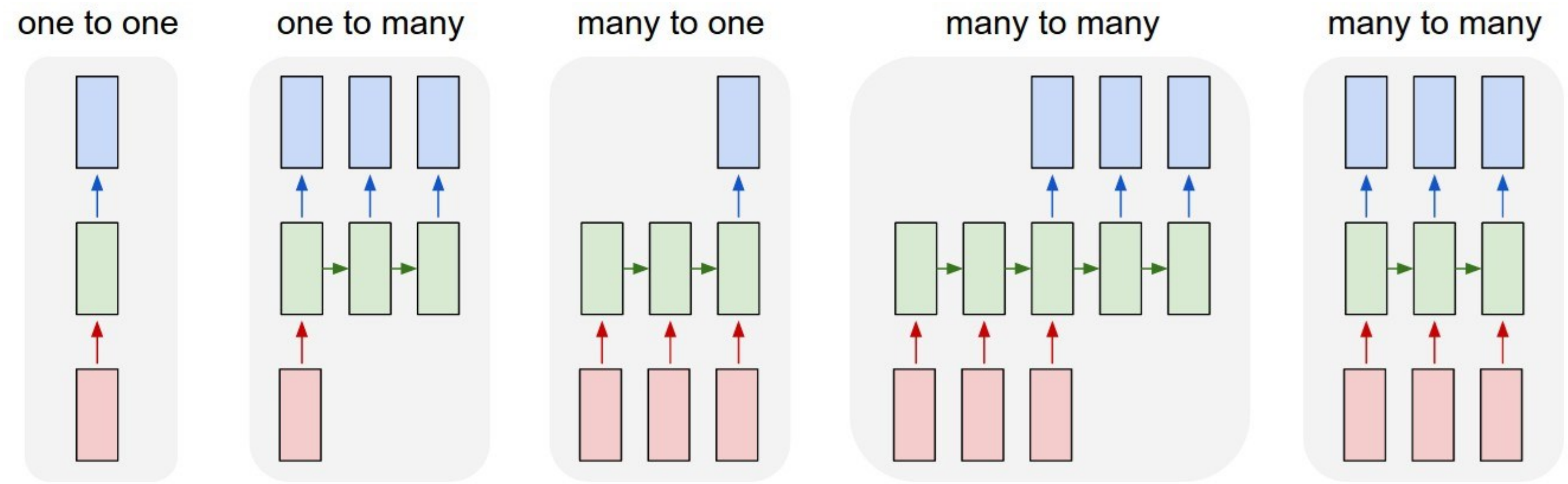


Activity Recognition in video

Event classification

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

RNNs are Flexible

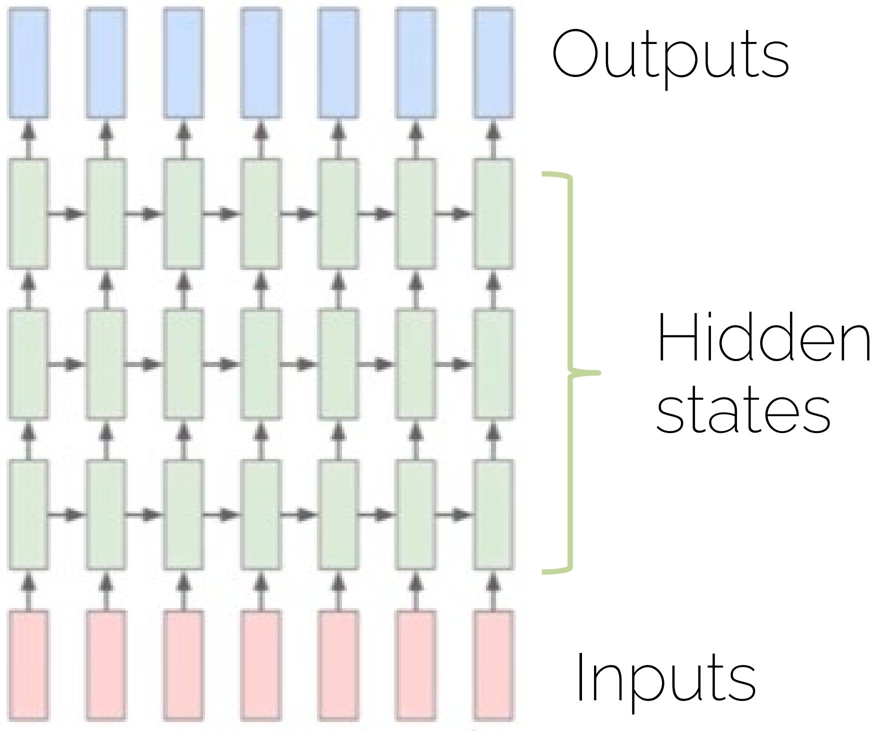


Event classification

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

Basic Structure of an RNN

- Multi-layer RNN



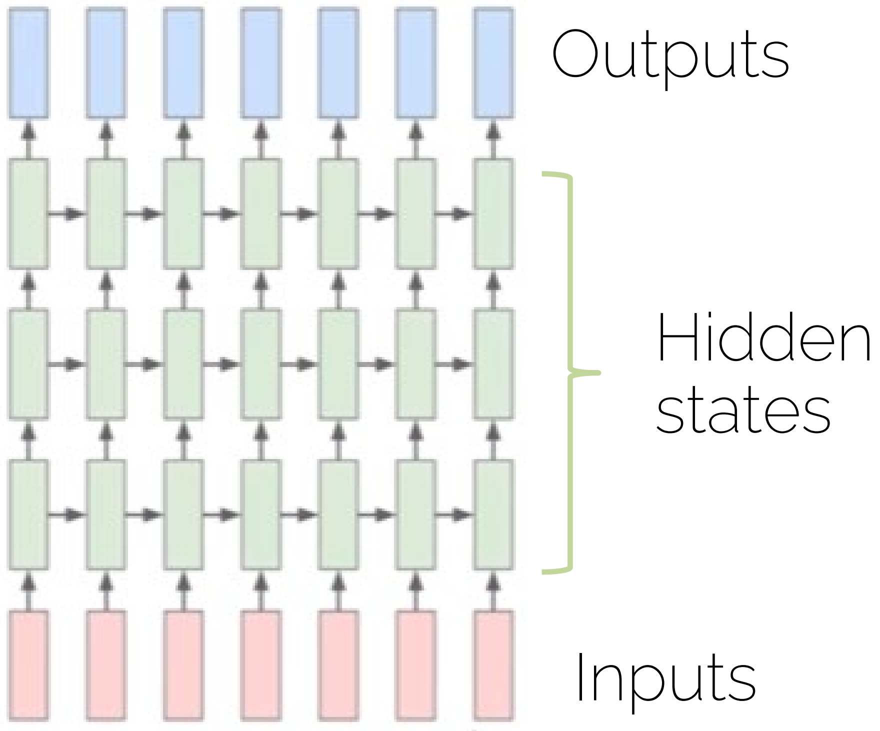
Basic Structure of an RNN

- Multi-layer RNN

The hidden state will have its own internal dynamics

↓

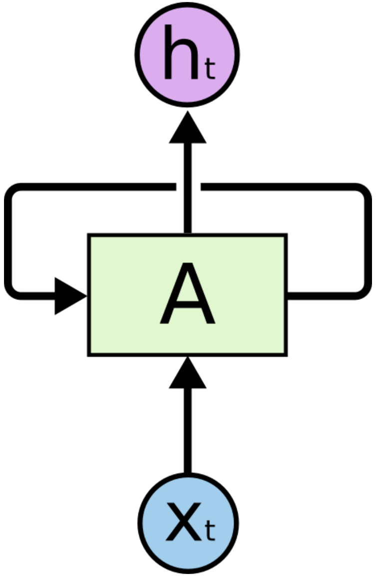
More expressive model!



Basic Structure of an RNN

- We want to have notion of "time" or "sequence"

Hidden state



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

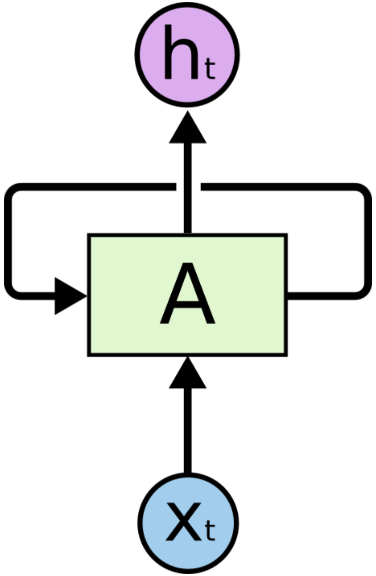
Previous hidden state

input

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

Basic Structure of an RNN

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



Hidden state

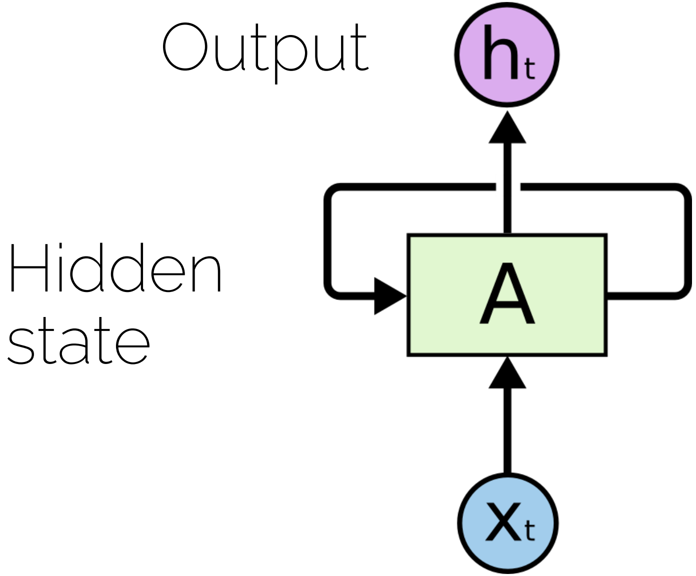
$$\boldsymbol{A}_t = \boldsymbol{\theta}_c \boldsymbol{A}_{t-1} + \boldsymbol{\theta}_x \boldsymbol{x}_t$$

Parameters to be learned

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

Basic Structure of an RNN

- We want to have notion of "time" or "sequence"



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

$$\boldsymbol{h_t = \theta_h A_t}$$

Note: non-linearities ignored for now

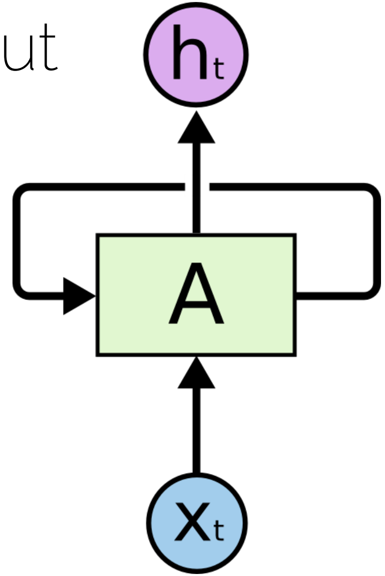
[Olah, https://colah.github.io '15] Understanding LSTMs

Basic Structure of an RNN

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

Output

Hidden state



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

$$\boldsymbol{h}_t = \boldsymbol{\theta}_h \boldsymbol{A}_t$$

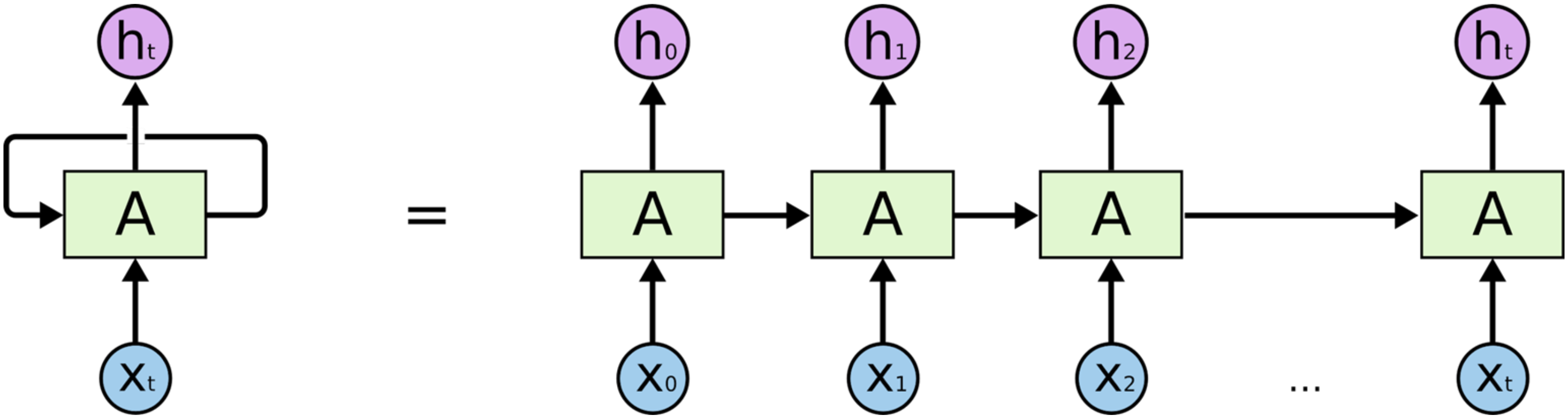
$\boxed{\text{Same parameters for each time step = generalization!}}$

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

Basic Structure of an RNN

- Unrolling RNNs

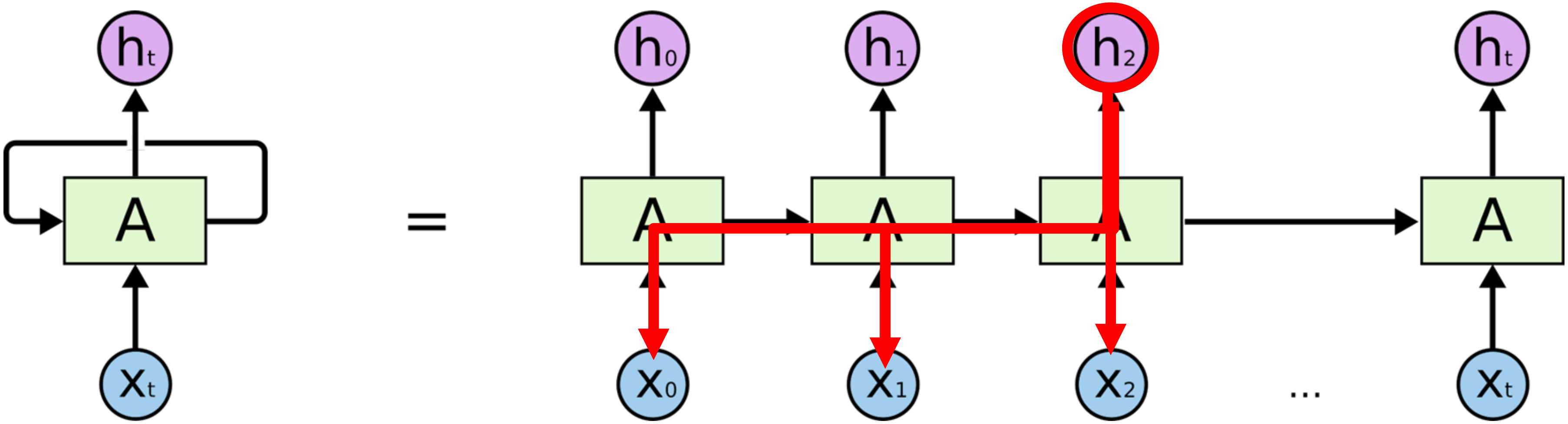
Same function for the hidden layers



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

Basic Structure of an RNN

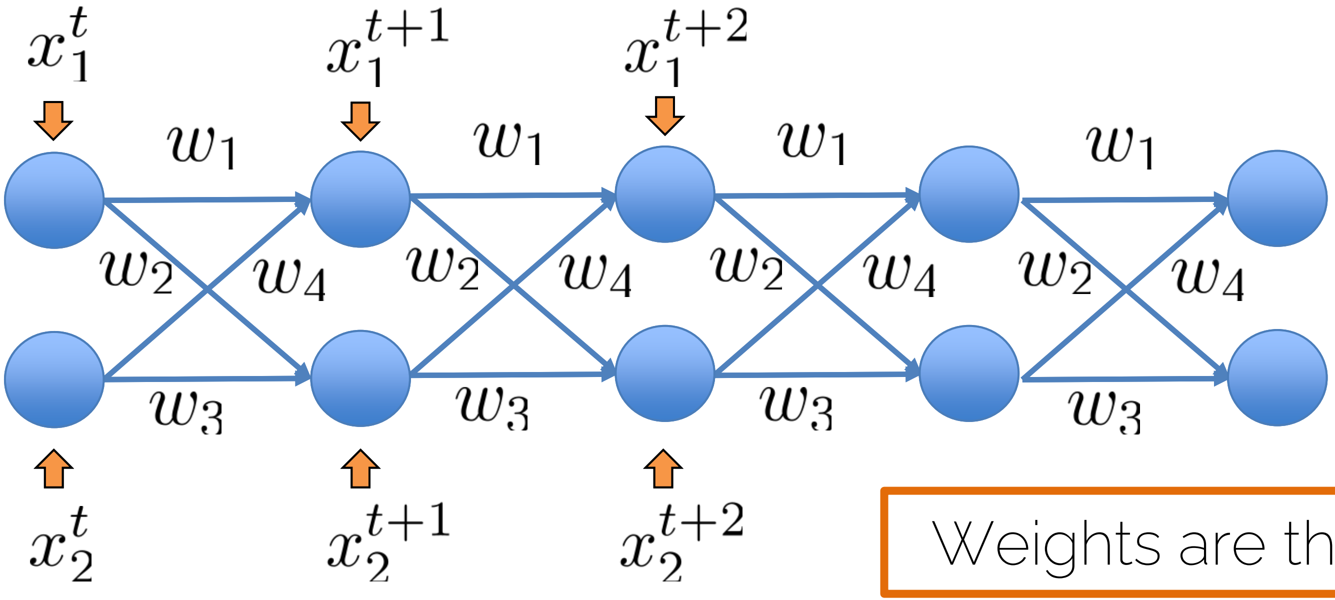
- Unrolling RNNs



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

Basic Structure of an RNN

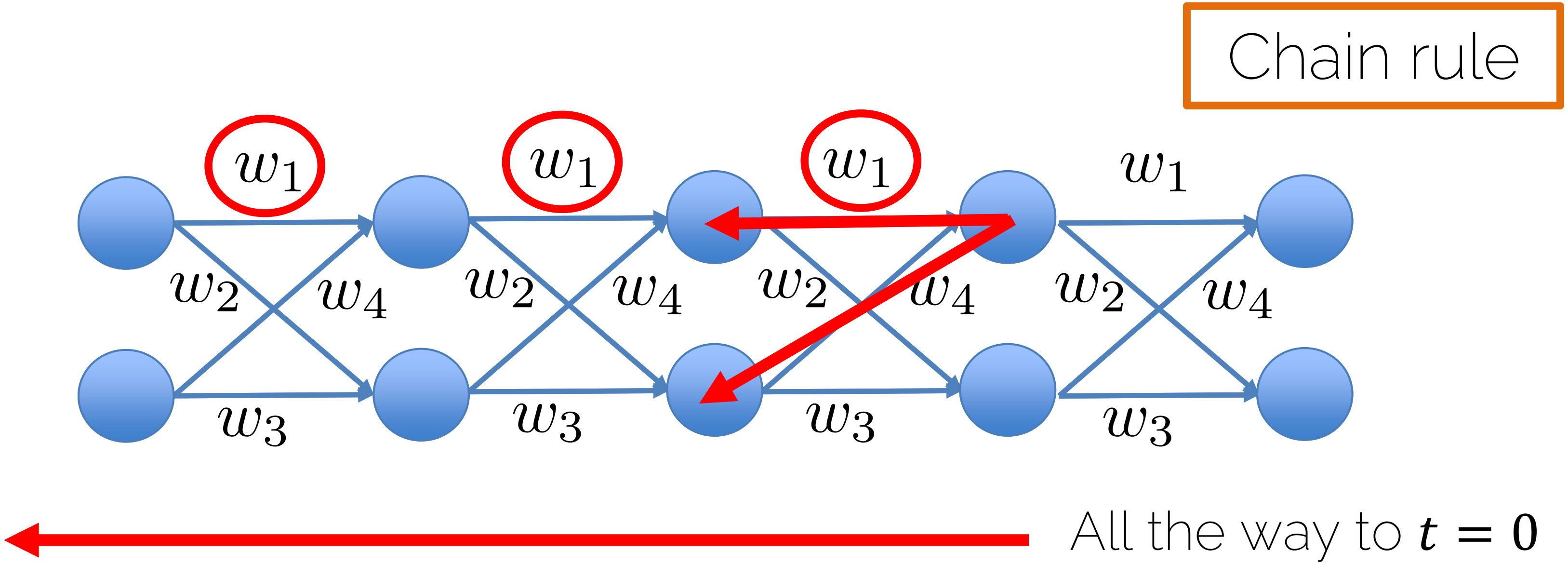
- Unrolling RNNs as feedforward nets



Weights are the same!

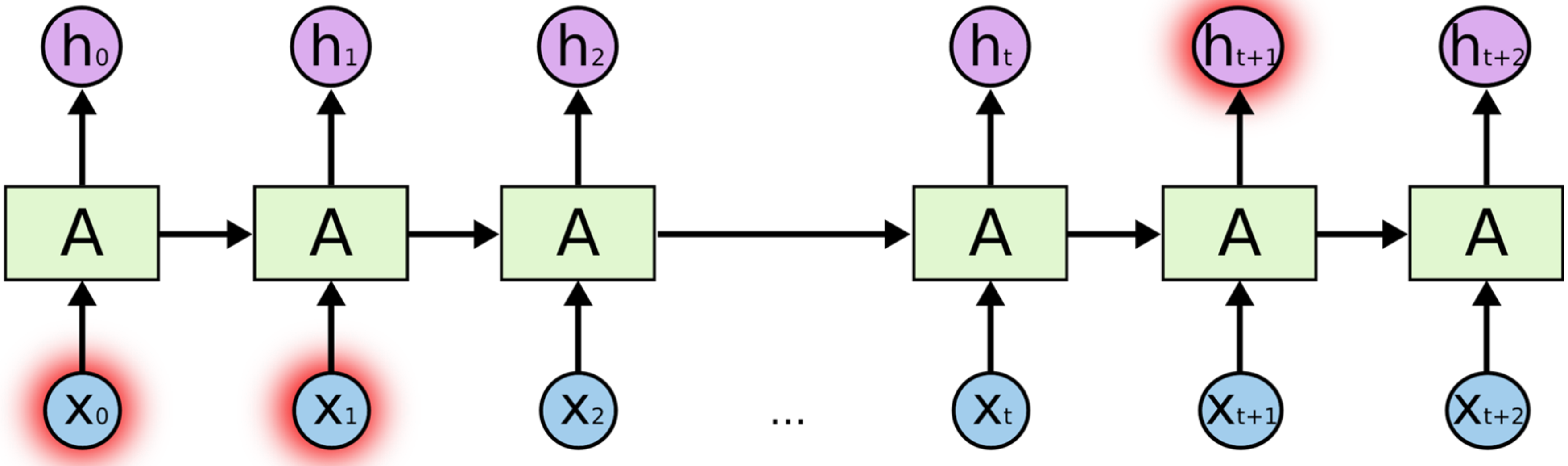
Backprop through an RNN

- 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.

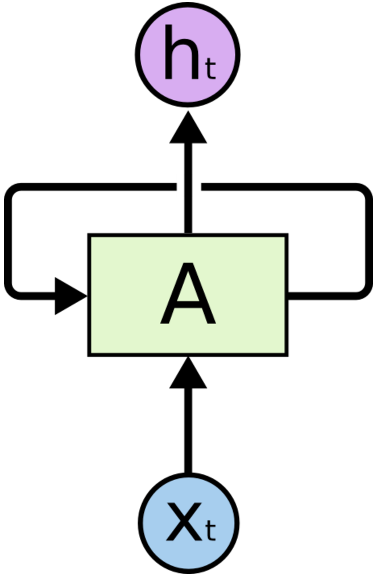
[Olah, https://colah.github.io '15] Understanding LSTMs

Long-term Dependencies

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

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

Same weights are multiplied over and over again



Long-term Dependencies

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

What happens to small weights?

Vanishing gradient

What happens to large weights?

Exploding gradient

Long-term Dependencies

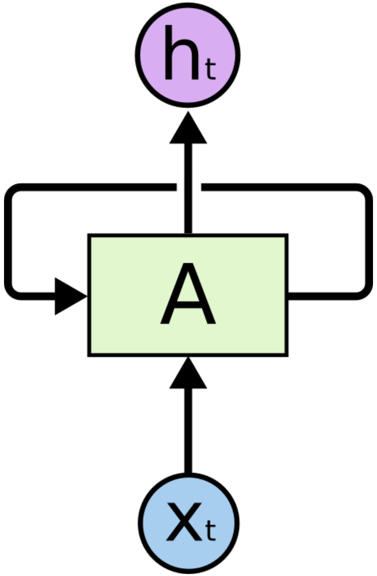
- Simple recurrence $\boldsymbol{A_t = \theta_c}^{t}\boldsymbol{A}_0$

- If $\boldsymbol{\theta}$ admits eigendecomposition

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

Matrix of eigenvectors

Diagonal of this matrix are the eigenvalues

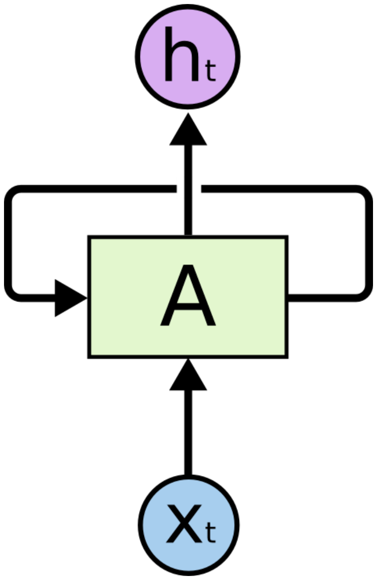


Long-term Dependencies

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

- If $\boldsymbol{\theta}$ admits eigendecomposition

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



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

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

Long-term Dependencies

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

What happens to eigenvalues with magnitude less than one?

Vanishing gradient

What happens to eigenvalues with magnitude larger than one?

Exploding gradient

Gradient clipping

→ Set threshold value for gradient
Avoid grad too large

Long-term Dependencies

- Simple recurrence $\boldsymbol{A}_t = \boldsymbol{\theta_c}^t \boldsymbol{A}_0$

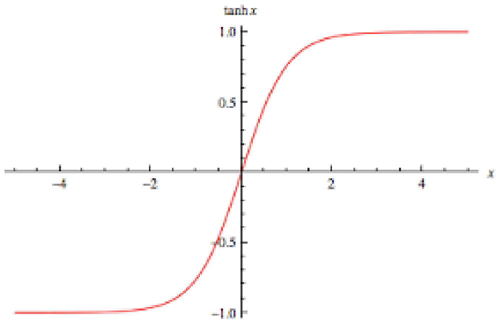
Let us just make a matrix with eigenvalues = 1

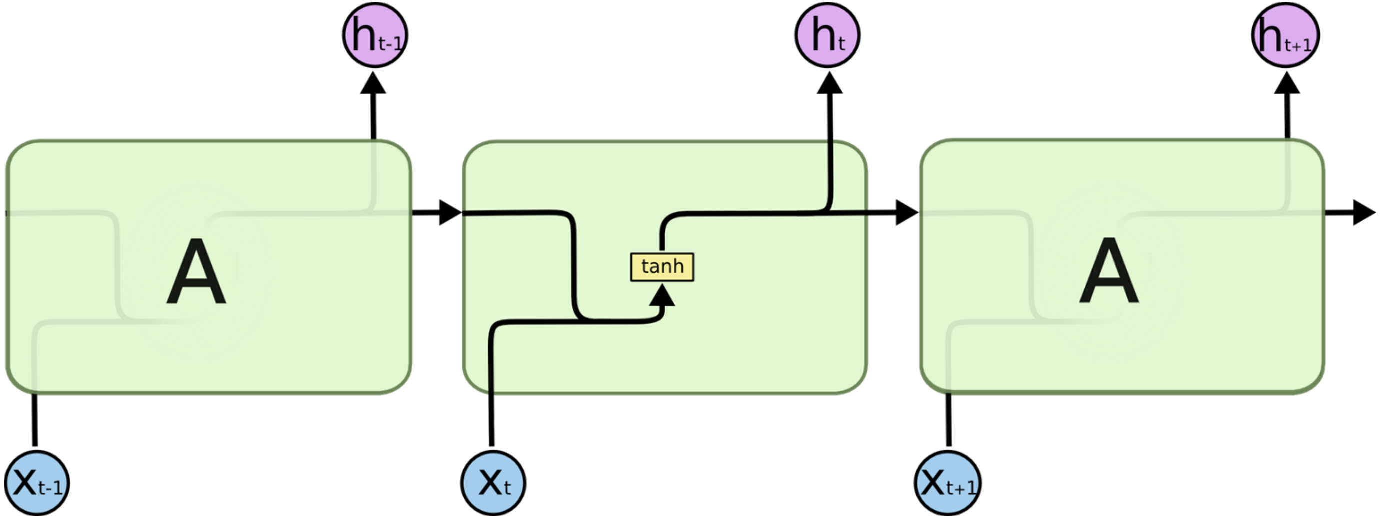
Allow the **cell** to maintain its "*state*"

Vanishing Gradient

- 1. From the weights $A_t = \theta_c{}^t A_0$

- 2. From the activation functions ($tanh$)





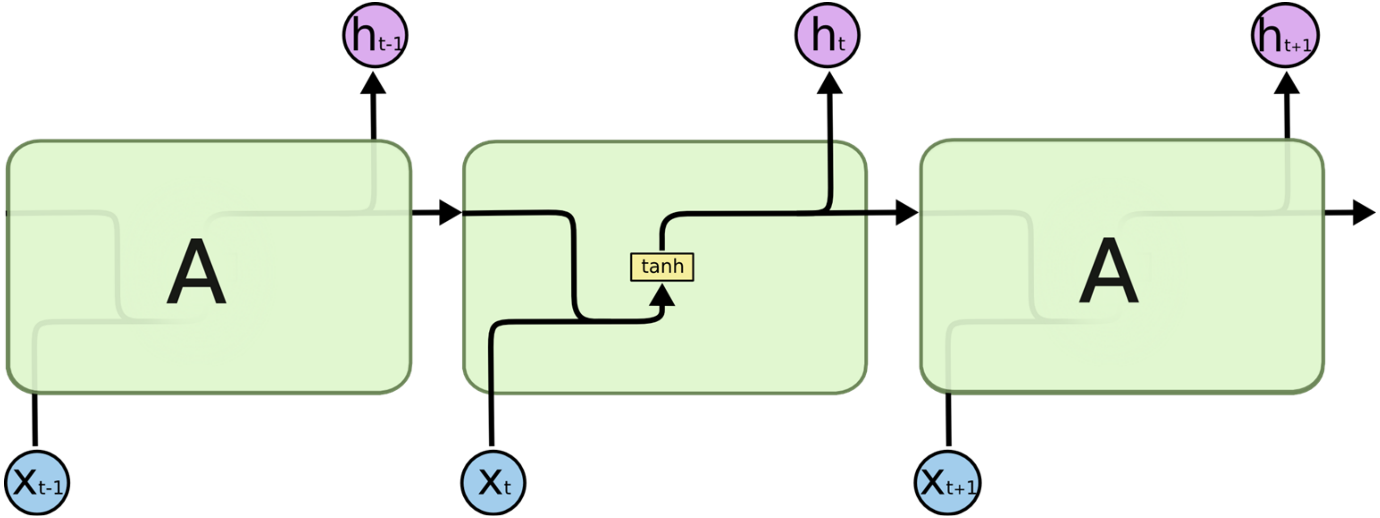
[Olah, https://colah.github.io '15] Understanding LSTMs

Vanishing Gradient

- 1. From the weights $\boldsymbol{A}_t = \cancel{\boldsymbol{\theta}}^t \boldsymbol{A}_0$

  1

- 2. From the activation functions ($\boldsymbol{tanh}$) ?



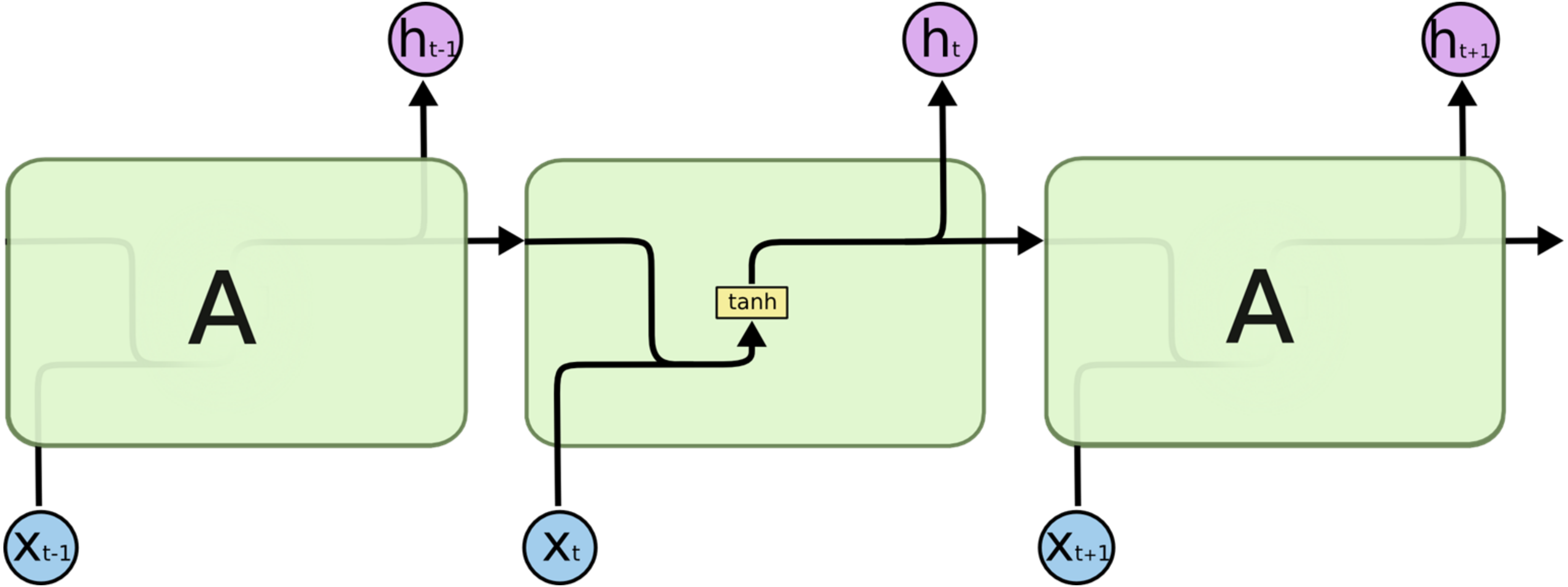
[Olah, https://colah.github.io '15] Understanding LSTMs

Long Short Term Memory

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

Long-Short Term Memory Units

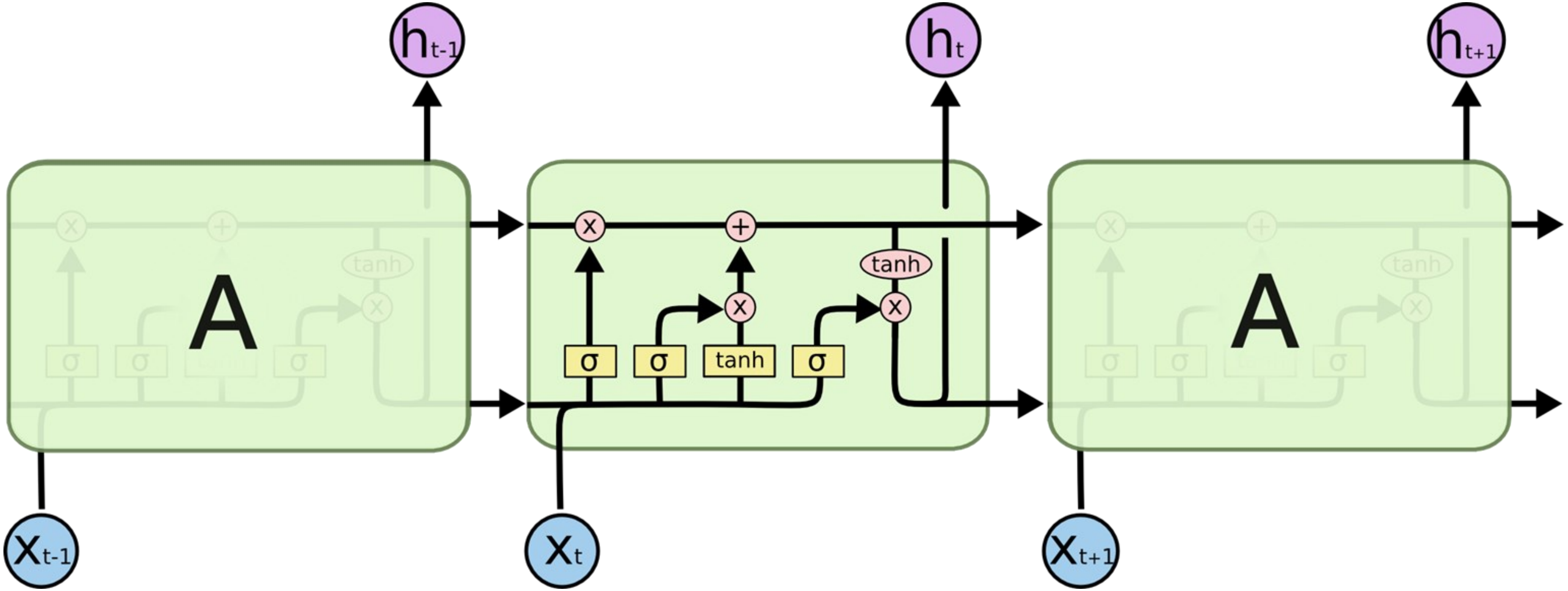
- Simple RNN has **tanh** as non-linearity



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

Long-Short Term Memory Units

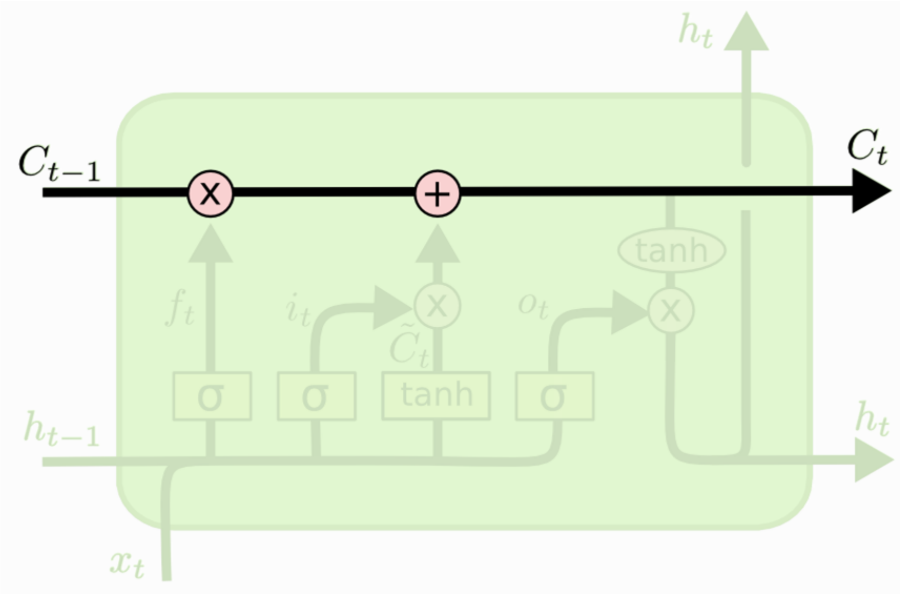
LSTM



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

Long-Short Term Memory Units

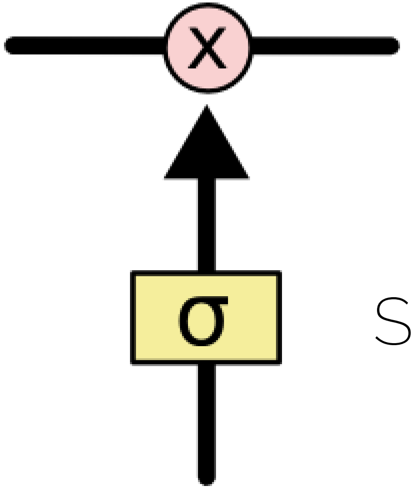
- Key ingredients
- Cell = transports the information through the unit



[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

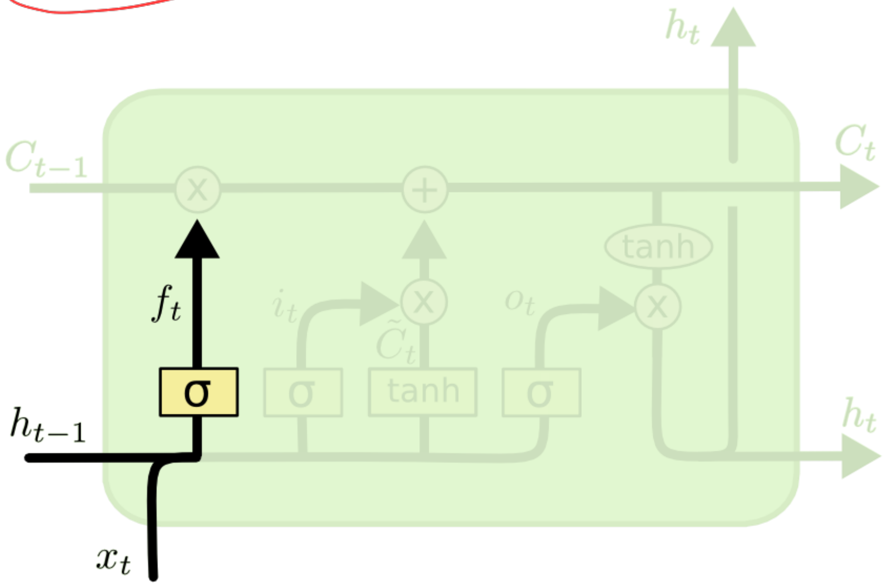


Sigmoid

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

LSTM: Step by Step

- Forget gate $\boldsymbol{f}_t = sigm(\boldsymbol{\theta}_{xf}\boldsymbol{x}_t + \boldsymbol{\theta}_{hf}\boldsymbol{h}_{t-1} + \boldsymbol{b}_f)$



Decides when to erase the cell state

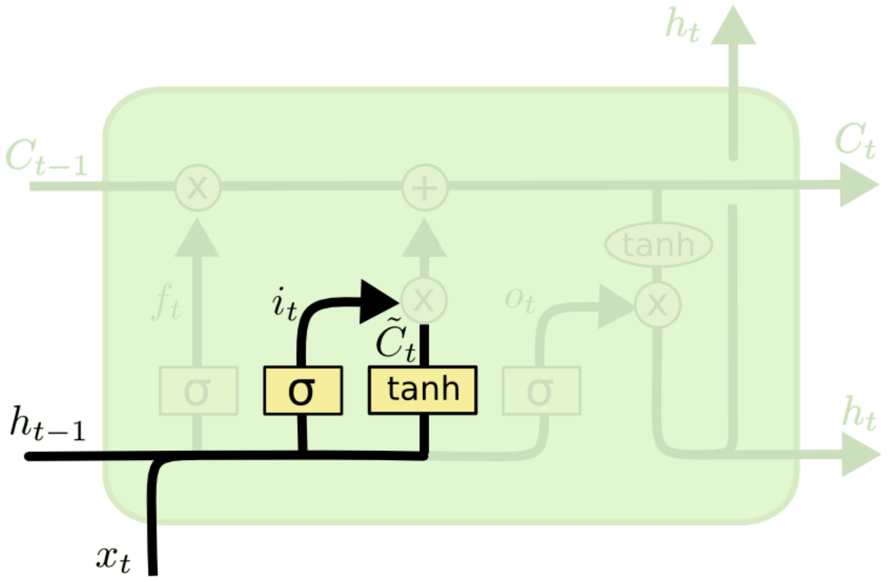
Sigmoid = output between **0** (forget) and **1** (keep)

if it is close to zero, we could forget previous state.

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

LSTM: Step by Step

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



Decides which values will be updated

if it is close to -1 extract new information

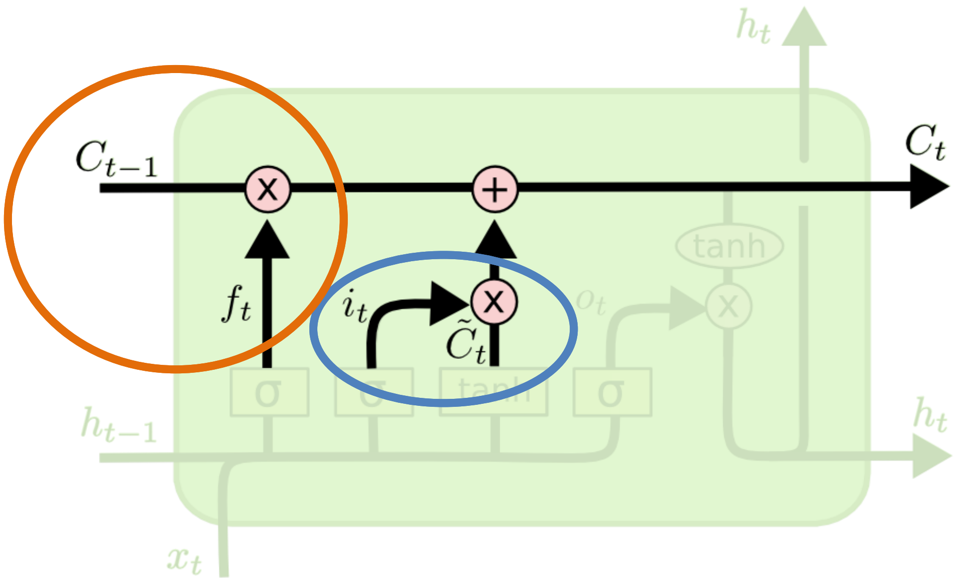
New cell state, output from a tanh $(-1,1)$

if it is close to 1 add new information

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

LSTM: Step by Step

- Element-wise operations



$$\boldsymbol{c}_t = \boldsymbol{f}_t \odot \boldsymbol{c}_{t-1} + \boldsymbol{i}_t \odot \boldsymbol{g}_t$$

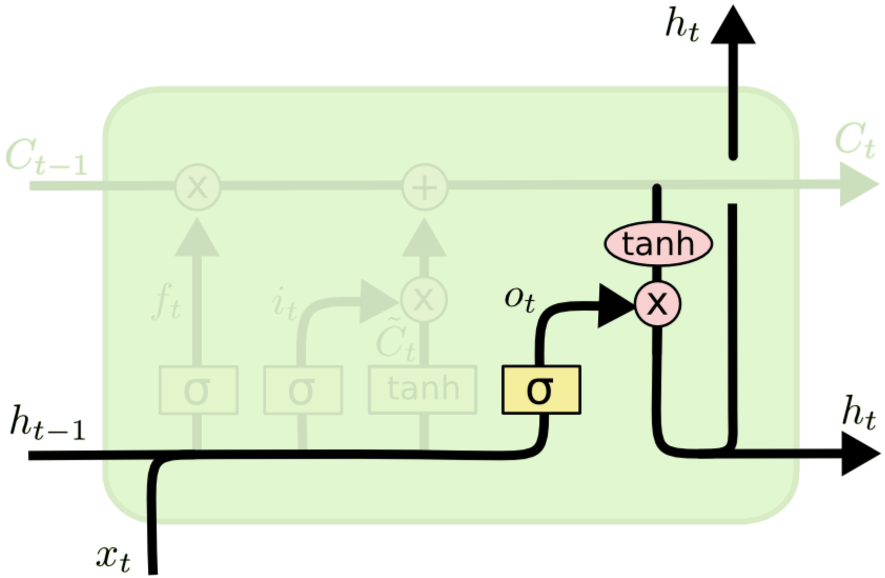
Previous states

Current state

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

LSTM: Step by Step

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



Decides which values will be outputted

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

LSTM: Step by Step

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

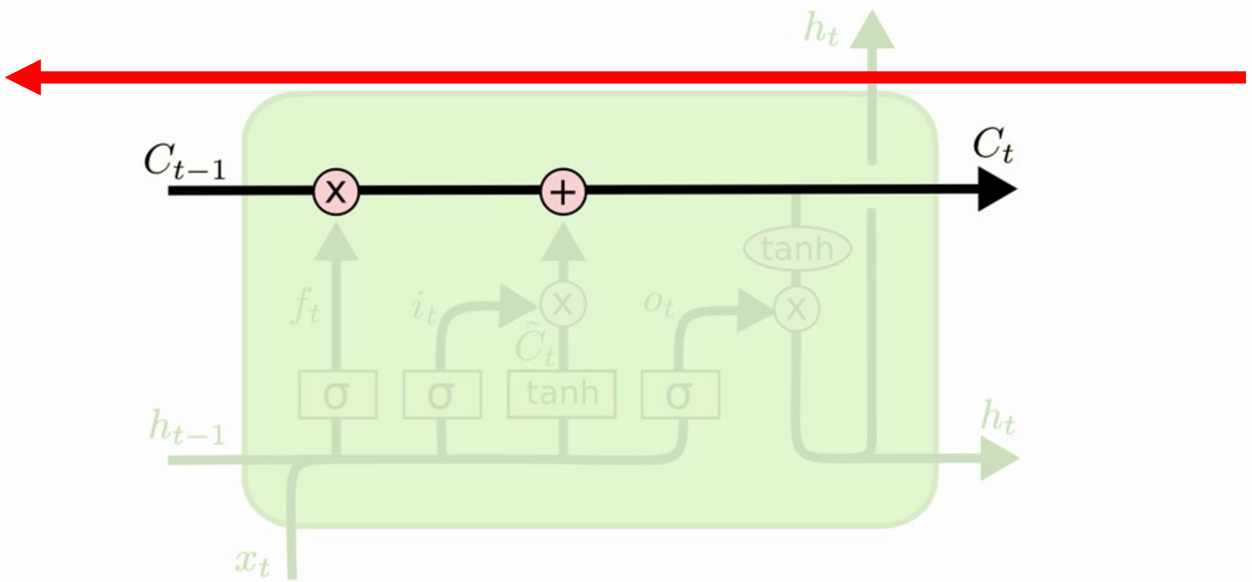
LSTM: Step by Step

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

Learned through backpropagation

LSTM

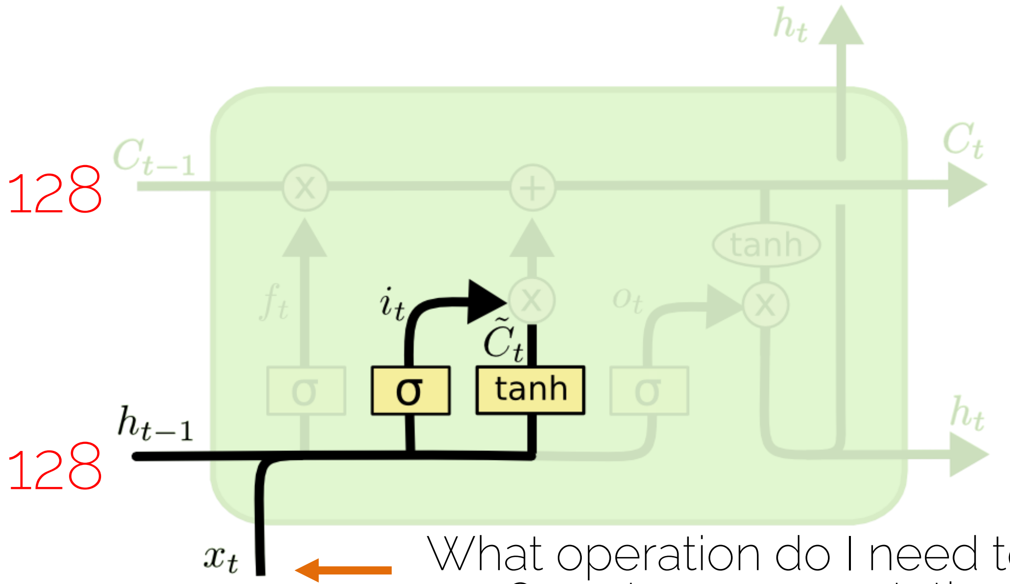
- Highway for the gradient to flow



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

LSTM: Dimensions

- Cell update $$\overset{128}{\boldsymbol{g}_t} = tanh(\boldsymbol{\theta}_{xg}\boldsymbol{x}_t + \overset{128}{\boldsymbol{\theta}_{hg}}\overset{128}{\boldsymbol{h}_{t-1}} + \boldsymbol{b}_g)$$



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

```python
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
```

```python
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

Łukasz Kaiser* Google Brain lukaszkaiser@google.com

Illia Polosukhin* ‡ illia.polosukhin@gmail.com

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

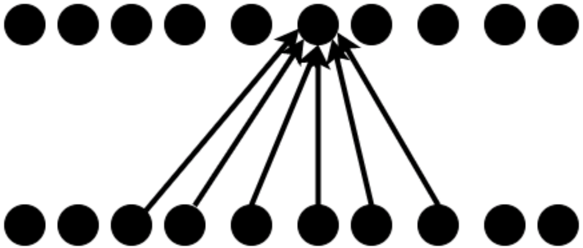
Łukasz Kaiser* Google Brain lukaszkaiser@google.com

Illia Polosukhin* ‡ illia.polosukhin@gmail.com

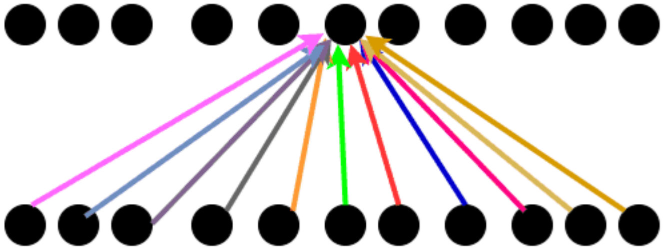
~62,000 citations in 5 years!

Attention vs convolution

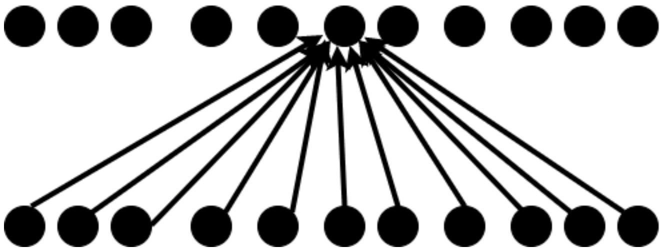
Convolution



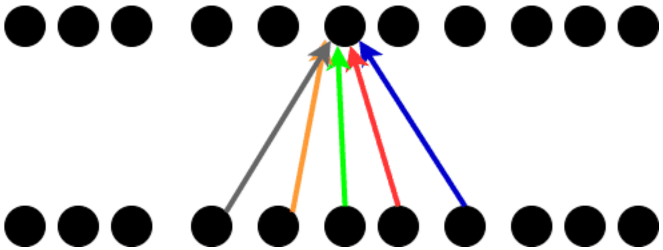
Global attention



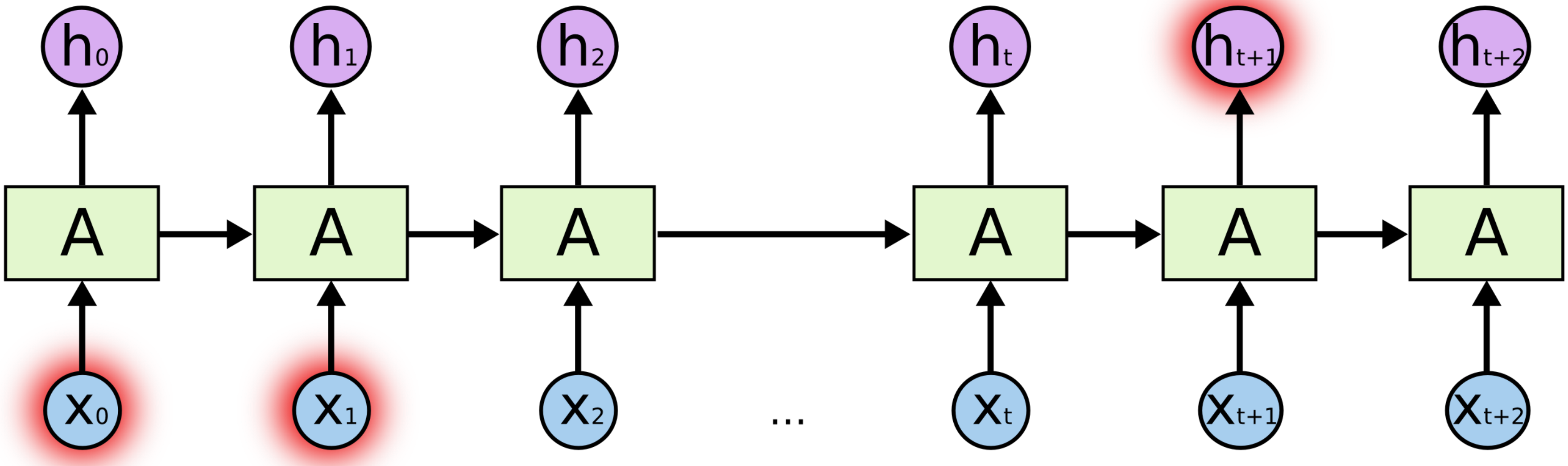
Fully Connected layer



Local attention



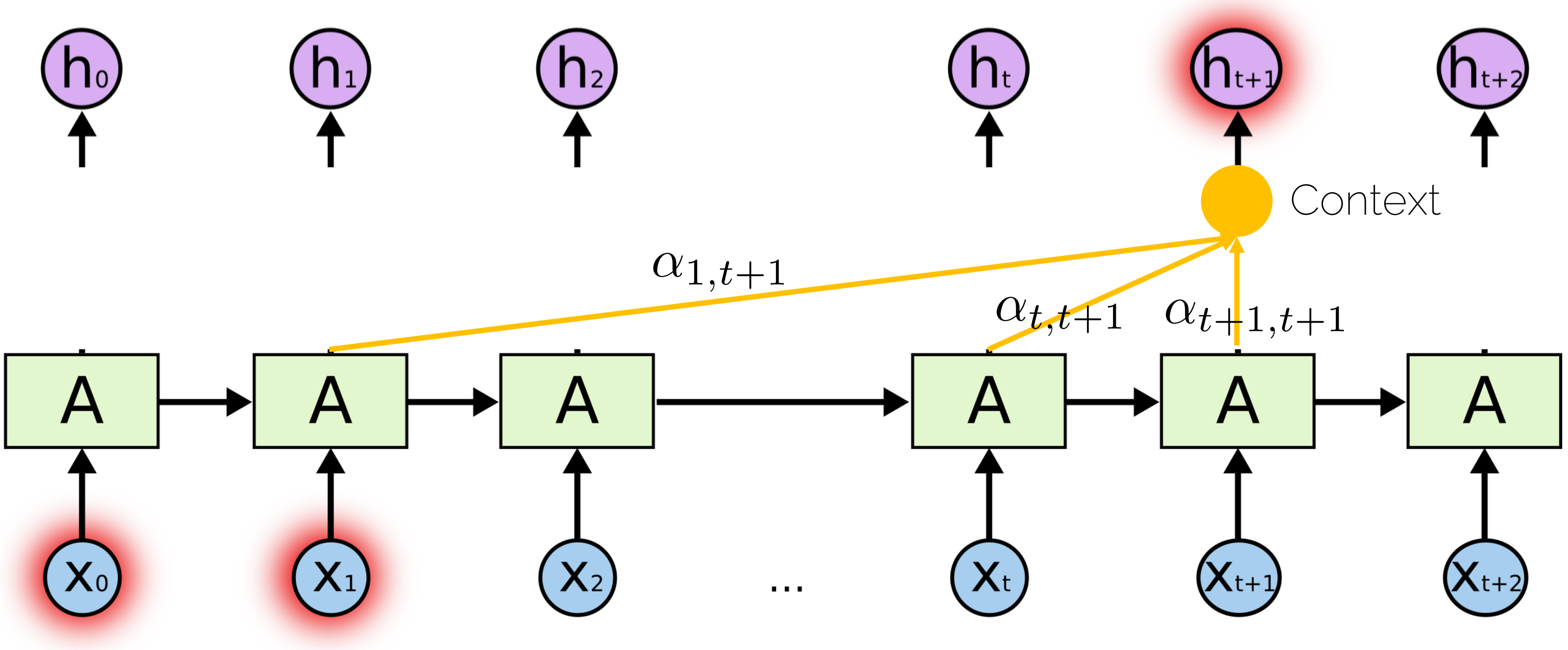
Long-Term Dependencies



I moved to Germany …   so I speak German fluently.

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

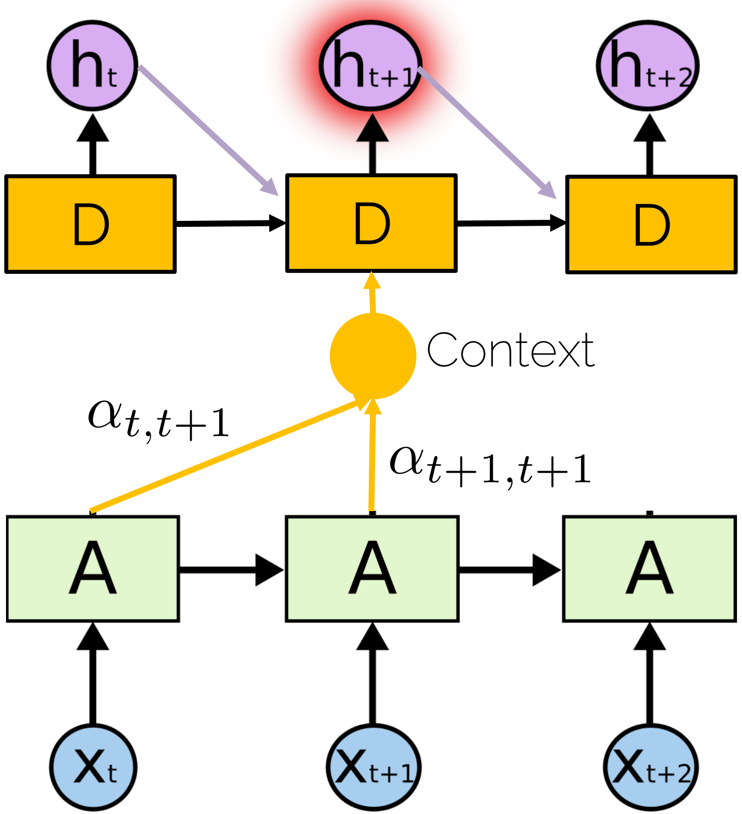
Attention: Intuition



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

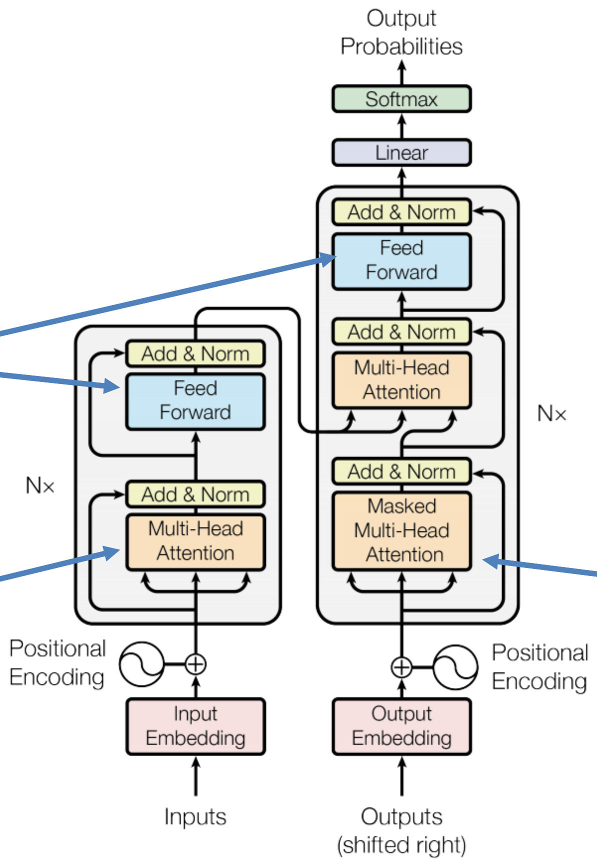


Transformers

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

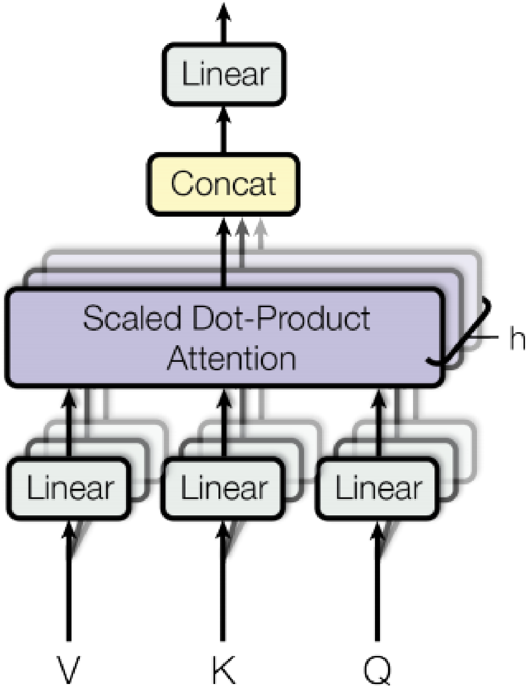


Fully connected layer

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.

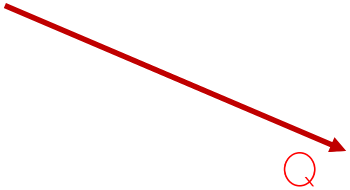
Multiply queries with keys

Get the values

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

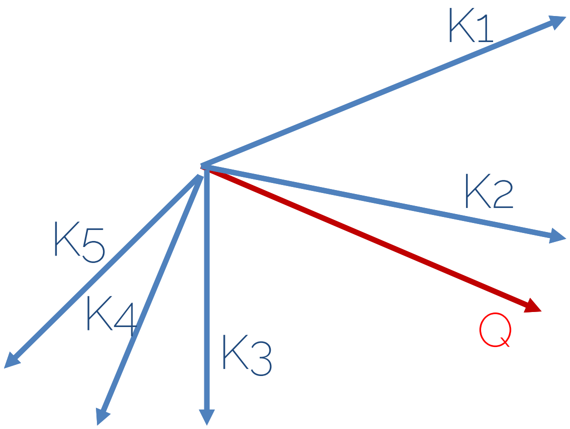
To train them well, divide by $\sqrt{d_k}$, "probably" because for large values of the key's dimension, the dot product grows large in magnitude, pushing the softmax function into regions where it has extremely small gradients.

Multi-Head Attention

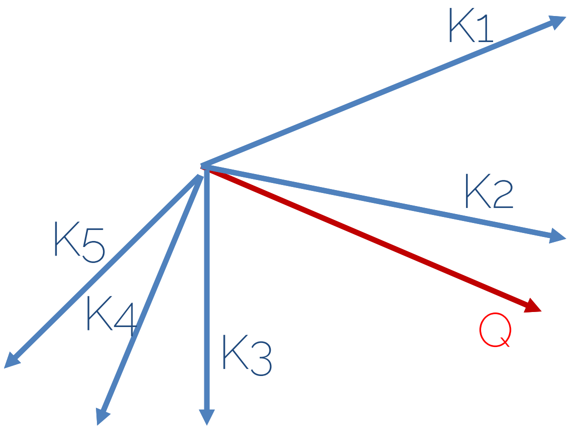


Adapted from Y. Kilcher

Multi-Head Attention

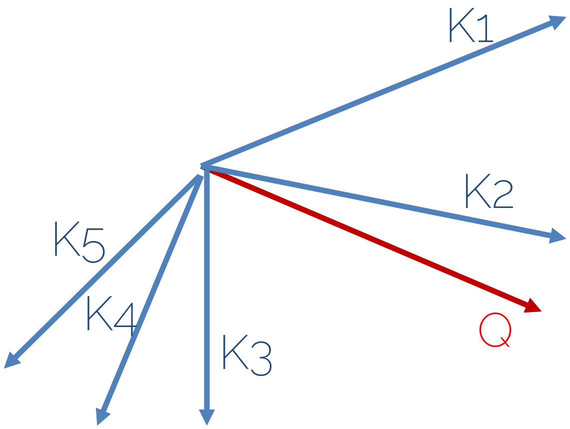


Multi-Head Attention



Values
V1
V2
V3
V4
V5

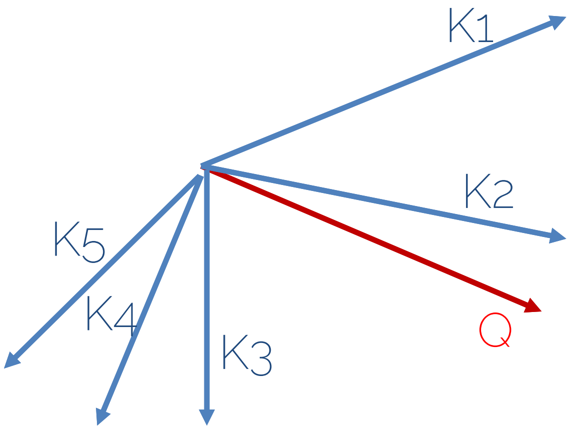
Multi-Head Attention



Values
V1
V2
V3
V4
V5

QK^T Essentially, dot product between (<Q,K1>), (<Q,K2>), (<Q,K3>), (<Q,K4>), (<Q,K5>).

Multi-Head Attention

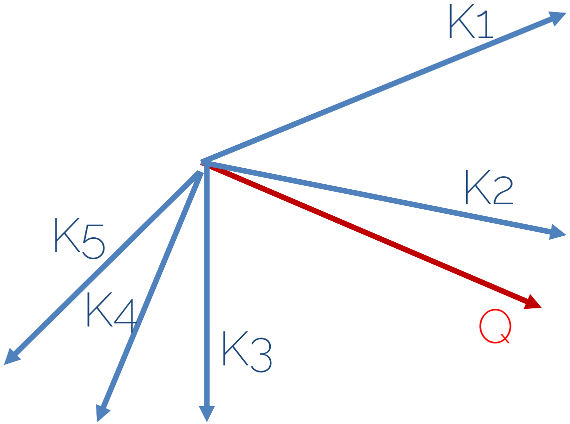


Values
V1
V2
V3
V4
V5

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

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

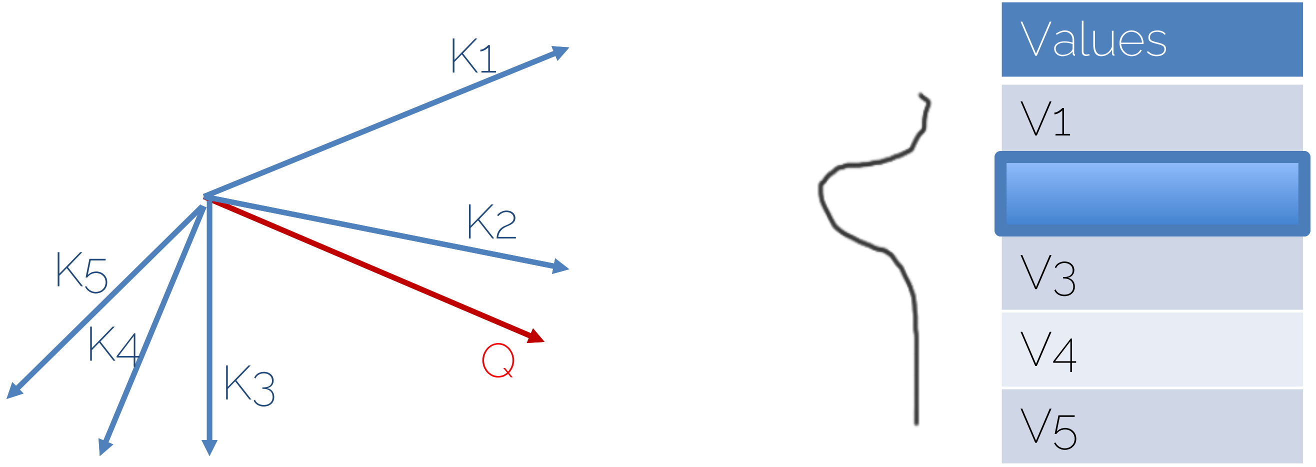


Values
V1
V2
V3
V4
V5

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

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

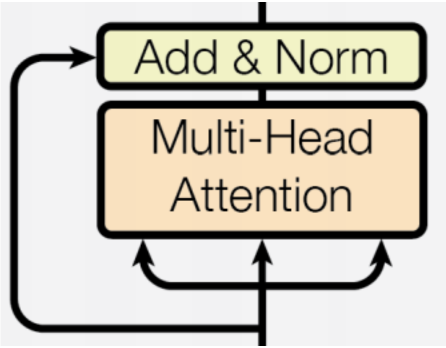


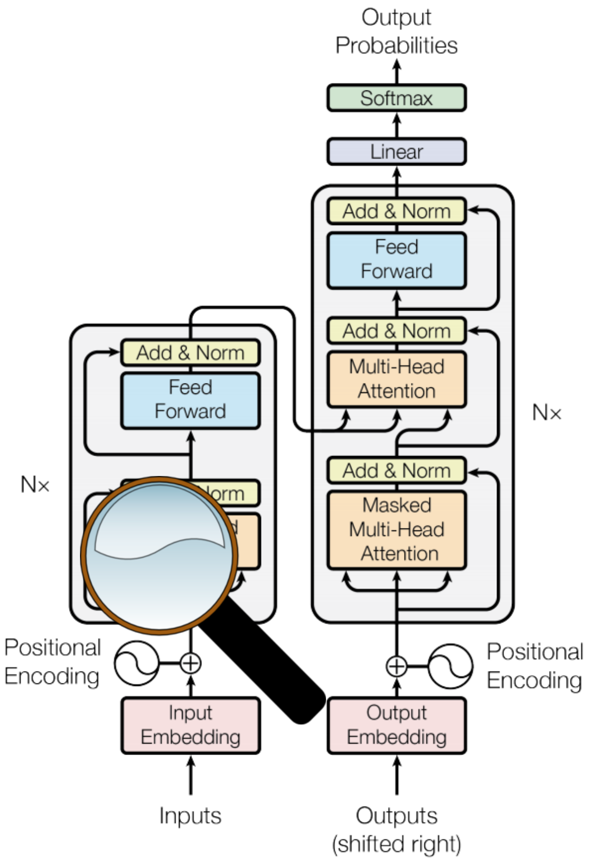
$$\mathrm{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)$$

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

Transformers – a closer look

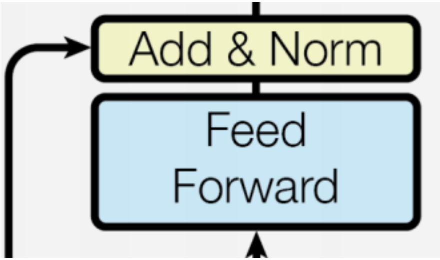
K parallel
attention heads.

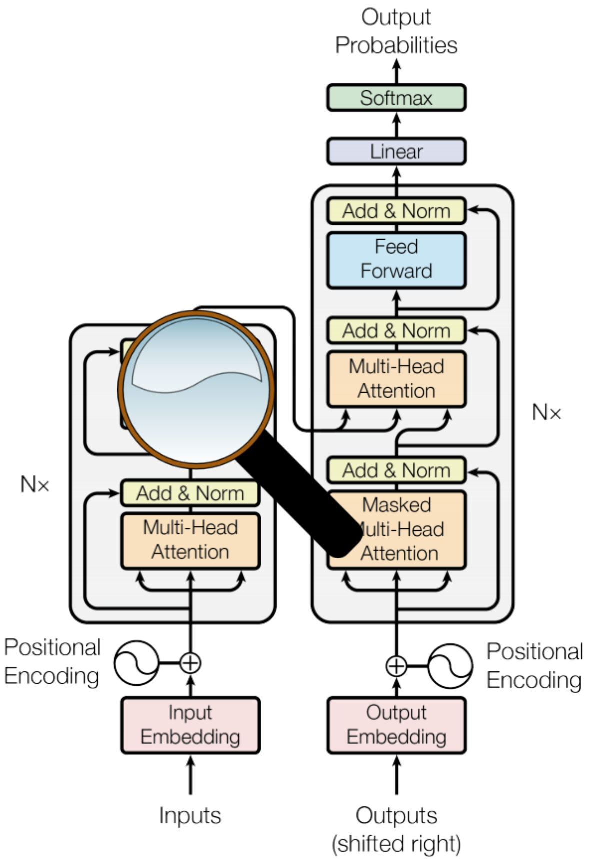




Transformers – a closer look

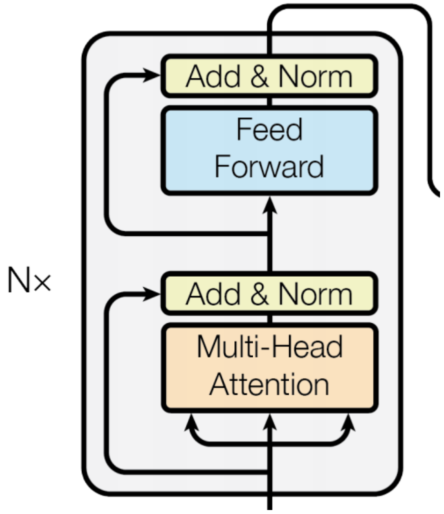
Good old fully-connected layers.

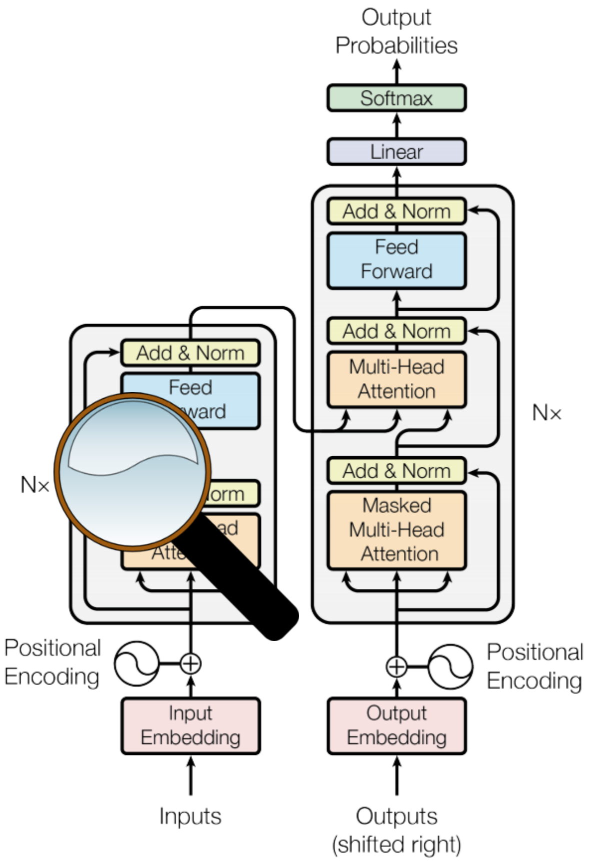




Transformers – a closer look

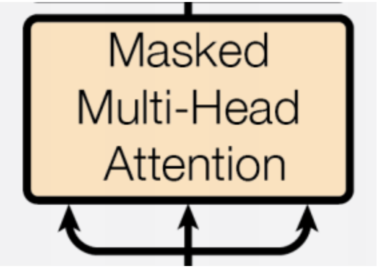
N layers of attention followed by FC

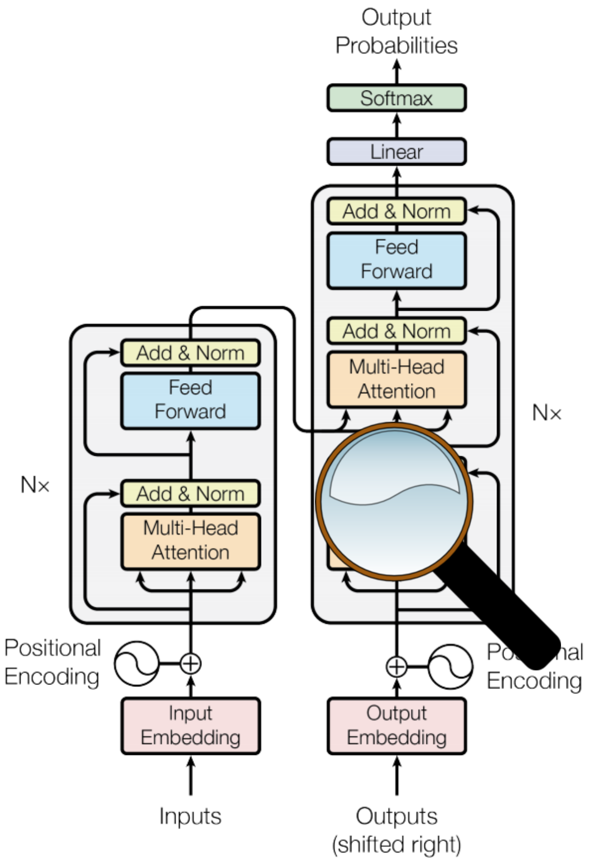




Transformers – a closer look

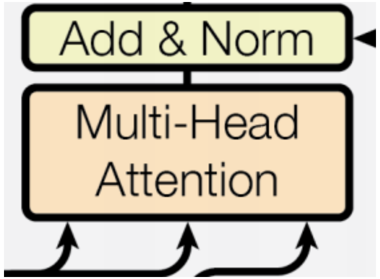
Same as multi-head attention, but masked. Ensures that the predictions for position i can depend only on the known outputs at positions less than i.

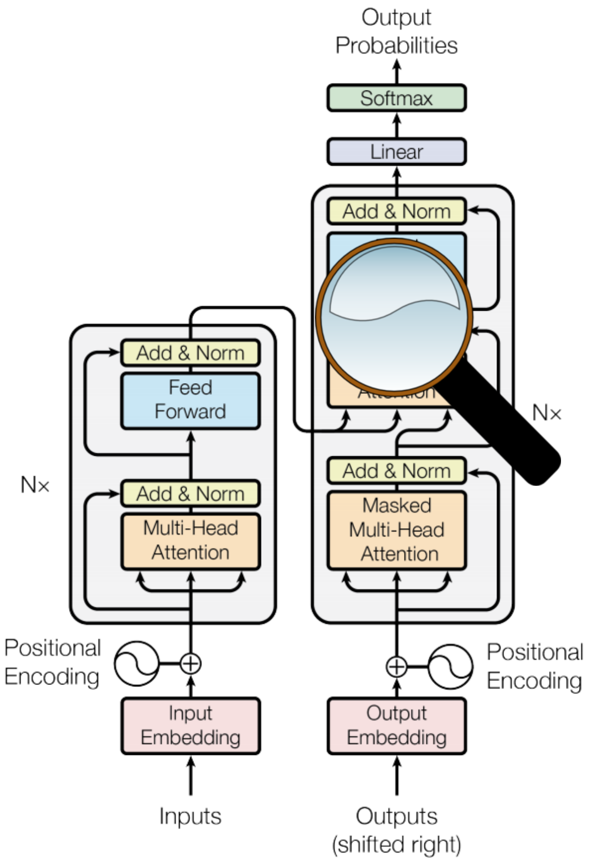




Transformers – a closer look

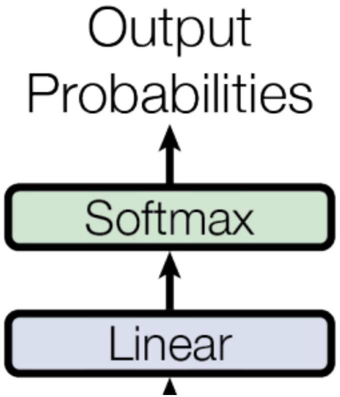
Multi-headed attention between encoder and the decoder.

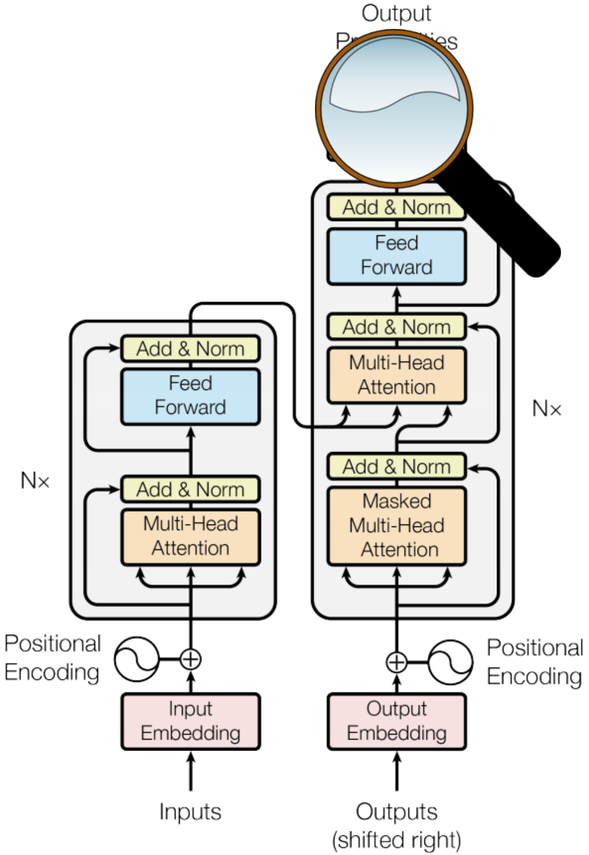




Transformers – a closer look

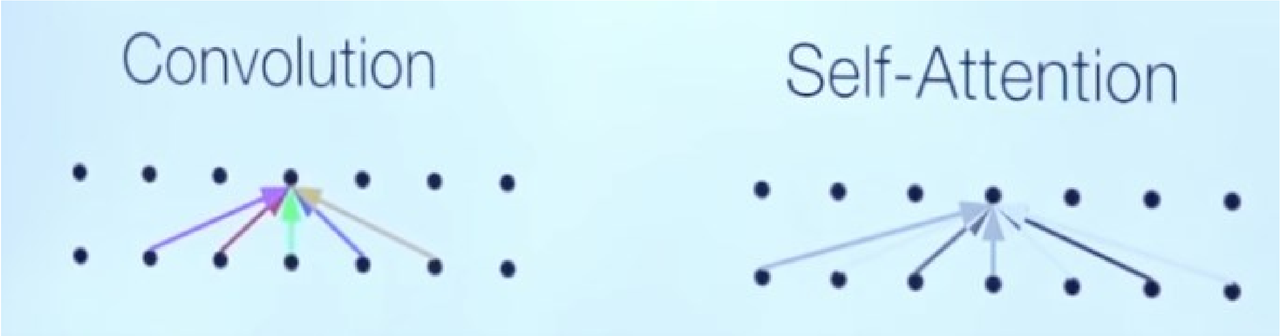
Projection and prediction.





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.



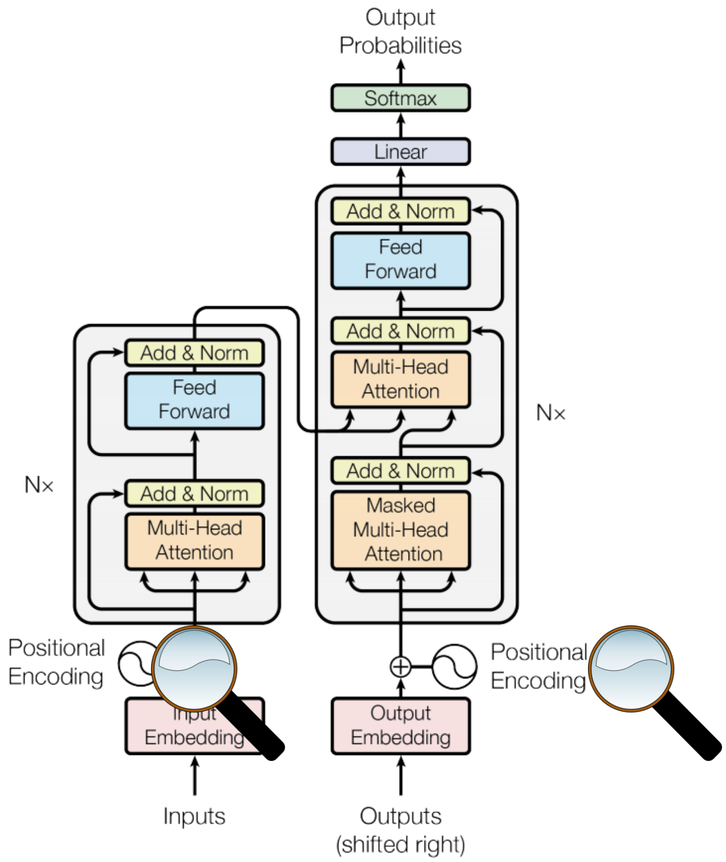
Transformers – a closer look

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

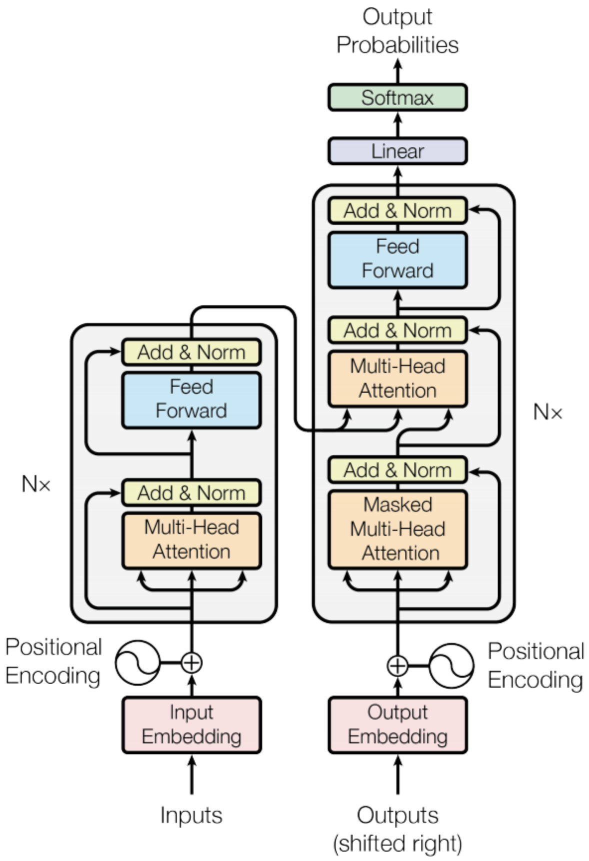
dimension

$$PE_{(pos,2i)} = \sin\left(\frac{pos}{10000^{2i/d_{\text{model}}}}\right)$$

$$PE_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{2i/d_{\text{model}}}}\right)$$



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 – training tricks

- ADAM optimizer with proportional learning rate:

$$lrate = d_{\text{model}}^{-0.5} \cdot \min(step\_num^{-0.5}, step\_num \cdot warmup\_steps^{-1.5})$$

- Residual dropout
- Label smoothing
- Checkpoint averaging

Transformers - results

Table 2: The Transformer achieves better BLEU scores than previous state-of-the-art models on the English-to-German and English-to-French newstest2014 tests at a fraction of the training cost.

Model	BLEU		Training Cost (FLOPs)	
	EN-DE	EN-FR	EN-DE	EN-FR
ByteNet [15]	23.75			
Deep-Att + PosUnk [32]		39.2		$1.0 \cdot 10^{20}$
GNMT + RL [31]	24.6	39.92	$2.3 \cdot 10^{19}$	$1.4 \cdot 10^{20}$
ConvS2S [8]	25.16	40.46	$9.6 \cdot 10^{18}$	$1.5 \cdot 10^{20}$
MoE [26]	26.03	40.56	$2.0 \cdot 10^{19}$	$1.2 \cdot 10^{20}$
Deep-Att + PosUnk Ensemble [32]		40.4		$8.0 \cdot 10^{20}$
GNMT + RL Ensemble [31]	26.30	41.16	$1.8 \cdot 10^{20}$	$1.1 \cdot 10^{21}$
ConvS2S Ensemble [8]	26.36	**41.29**	$7.7 \cdot 10^{19}$	$1.2 \cdot 10^{21}$
Transformer (base model)	27.3	38.1	$\mathbf{3.3 \cdot 10^{18}}$	
Transformer (big)	**28.4**	**41.0**	$2.3 \cdot 10^{19}$	

Transformers - summary

- Significantly improved SOTA in machine translation
- Launched a new deep-learning revolution in MLP
- Building block of NLP models like BERT (Google) or GPT/ChatGPT (OpenAI)
- BERT has been heavily used in Google Search

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

See you next time!