Alice's Adventures in a differentiable wonderland

More on Neural Network Architectures

Simone Scardapane (https://www.sscardapane.it/)

This set of slides is prepared by **Alfan F. Wicaksono**
Information Retrieval, CS UI

* Some slides were originally made by **Alfan**

“Convolutional” Neural Networks

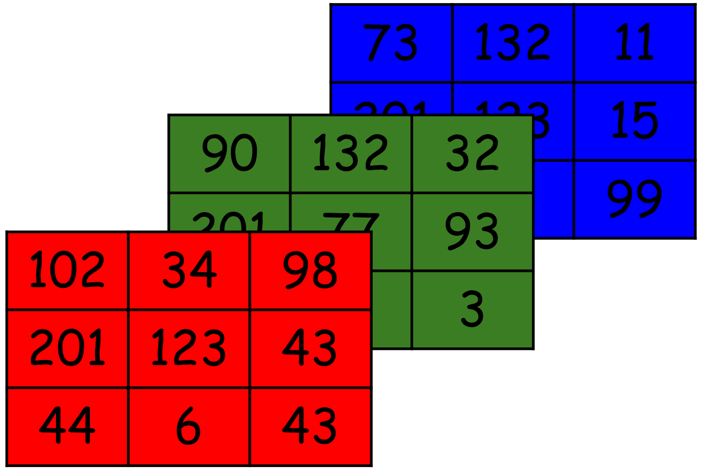
Why fully-connected layers are not enough?

- Fully-connected layers are important historically, but less so from a practical point of view: **on unstructured data**, FC layers are generally outperformed by other alternatives, such as random forests, gradient boosting trees, or well tuned support vector machines.

- This is not true, however, as soon as we consider other types of data, **having some structure** that can be exploited in the design of the layers and of the model.

Why fully-connected layers are not enough?

An image can be described by a tensor **X ~ (c, h, w)**, where **h** is the **height** of the image, **w** the **width** of the image, and **c** is the number of **channels** (which can be 1 for black and white images, or 3 for color images, RGB).




```
>>> img = torch.tensor([[[102, 34, 98],
                         [201, 123, 43],
                         [44, 6, 43]],
                        [[90, 132, 32],
                         [201, 77, 93],
                         [44, 6, 3]],
                        [[73, 132, 11],
                         [201, 123, 15],
                         [44, 6, 99]]])
```

An image 3 x 3 pixels with 3 channels

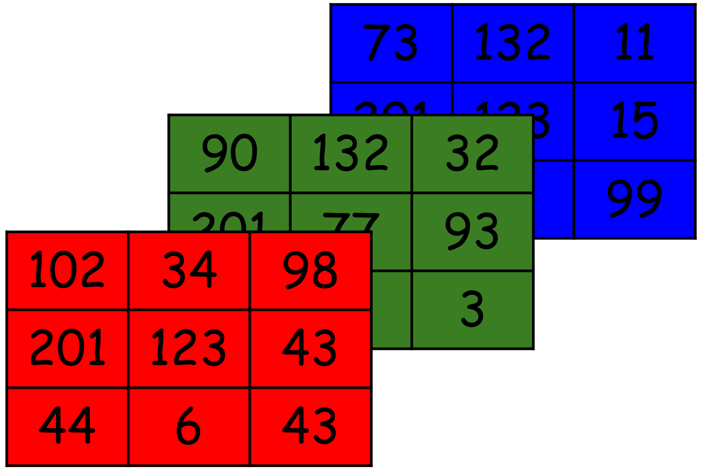
Why fully-connected layers are not enough?

In order to use a fully-connected layer, we would need to **"flatten" (vectorize)** the image:

$$\mathbf{h} = \phi(\mathbf{W} \cdot \boxed{\mathrm{vect}(X)})$$

Flattened image

```
>>> img.reshape(-1)

tensor([102,  34,  98, 201, 123,  43,  44,
6,  43,  90, 132,  32, 201,  77, 93,  44,   6,
3,  73, 132,  11, 201, 123,  15,  44,   6,
99])
```



An image 3 x 3 pixels with 3 channels

Why fully-connected layers are not enough?

This leads directly to an issue, which is that the layer has a **huge number of parameters**.

Considering, for example, a **(1024, 1024)** image in **RGB**, keeping the **same dimensionality** in output results in **(1024 * 1024 * 3) ^ 2** = **9.895.604.649.984** parameters!

Tensor ~ 3 * 1024 * 1024

$$\mathbf{h} = \phi(\mathbf{W} \cdot \boxed{\mathrm{vect}(X)}\,)$$

Tensor ~ 3 * 1024 * 1024

Flattened image

Fully-connected layer on a 1D sequence

As a running example to visualize what follows, consider a 1D sequence (we will consider 1D sequences more in-depth later on; for now, you can think of this as "*4 pixels with a single channel*"):

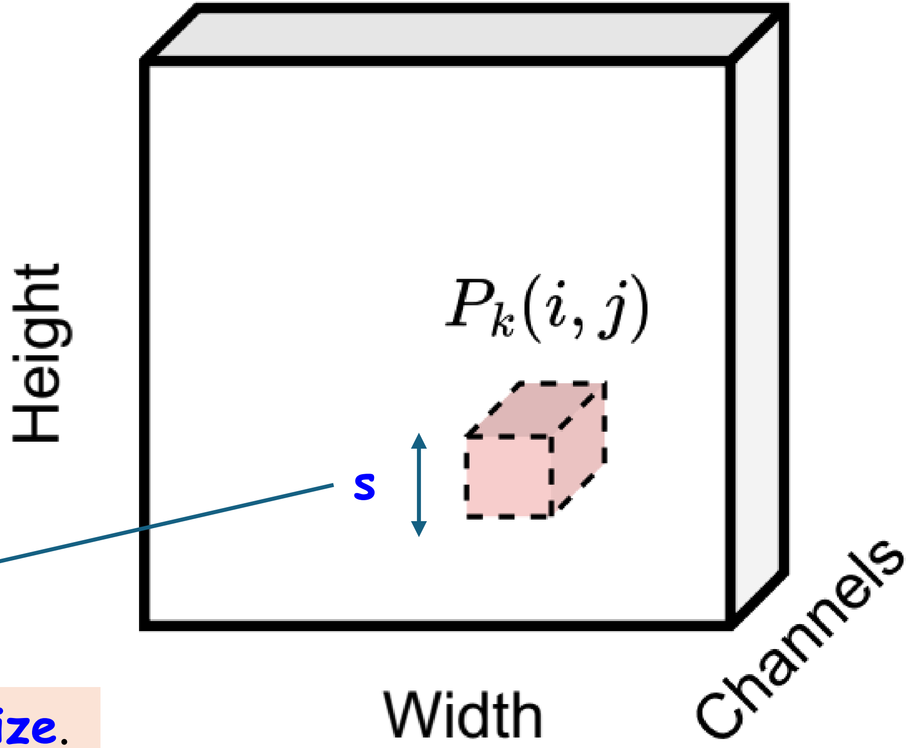
$$\mathbf{x} = \left[x_1, x_2, x_3, x_4\right]$$

In this case, we do not need any reshaping operations, and the previous layer (with $c' = 1$) can be written as:

$$\begin{bmatrix} h_1 \\ h_2 \\ h_3 \\ h_4 \end{bmatrix} = \begin{bmatrix} W_{11} & W_{12} & W_{13} & W_{14} \\ W_{21} & W_{22} & W_{23} & W_{24} \\ W_{31} & W_{32} & W_{33} & W_{34} \\ W_{41} & W_{42} & W_{43} & W_{44} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix}$$

A Patch

Figure F.7.1: *Given a tensor (h, w, c) and a maximum distance k, the **patch** $P_k(i, j)$ (shown in red) is a $(2k+1, 2k+1, c)$ tensor collecting all pixels at distance at most k from the pixel in position (i, j).*



$s = 2k + 1$; we call s the **filter size** / **kernel size**.

Definition D.7.1 (Image patch) *Given an image X, we define the **patch** $P_k(i, j)$ as the sub-image centered at (i, j) and containing all pixels at distance equal or lower than k:*

$$P_k(i, j) = [X]_{i-k:i+k, j-k:j+k, :}$$

Locally-connected layers

Definition D.7.2 (Local layer) *Given an input image* $X \sim (h, w, c)$*, a layer* $f(X) \sim (h, w, c')$ *is* ***local*** *if there exists a k such that:*

$$[f(X)]_{ij} = f(P_k(i,j))$$

This has to hold for all pixels of the image.

Flattened patch | Shape = (c * s * s)

$$H_{ij} = \phi\left(\mathbf{W}_{ij} \cdot \mathrm{vect}(P_k(i,j))\right)$$

Position-dependent weight matrix | Shape = (c' , c * s * s)

Total number of parameters = **h * w * (s * s * c * c')**

For comparison, in the fully-connected layers, we had **(h * w)² * (c * c')**

Locally-connected layer on a 1D sequence

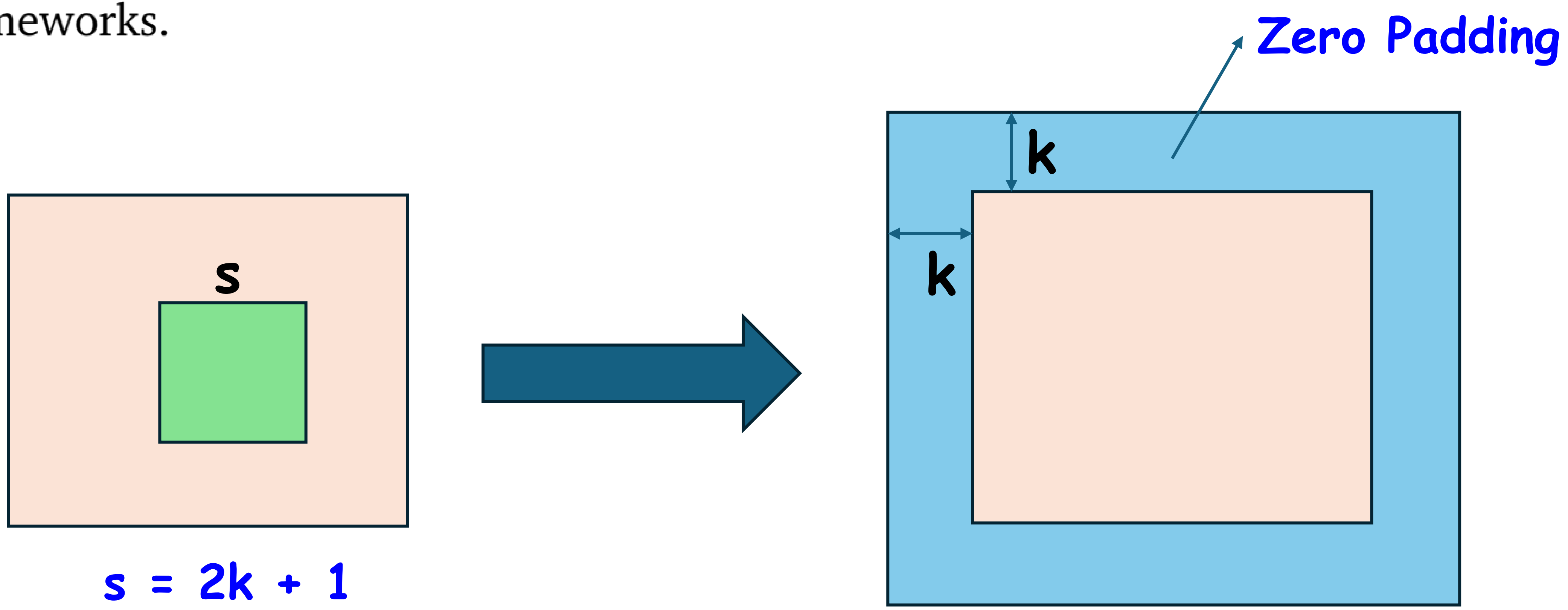
Considering our toy example, assuming for example $k = 1$ (hence $s = 3$) we can write the resulting operation as:

$$\begin{bmatrix} h_1 \\ h_2 \\ h_3 \\ h_4 \end{bmatrix} = \begin{bmatrix} W_{11} & W_{12} & W_{13} & 0 & 0 & 0 \\ 0 & W_{21} & W_{22} & W_{23} & 0 & 0 \\ 0 & 0 & W_{31} & W_{32} & W_{33} & 0 \\ 0 & 0 & 0 & W_{41} & W_{42} & W_{43} \end{bmatrix} \begin{bmatrix} 0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ 0 \end{bmatrix}$$

Zero Padding

Zero-Padding

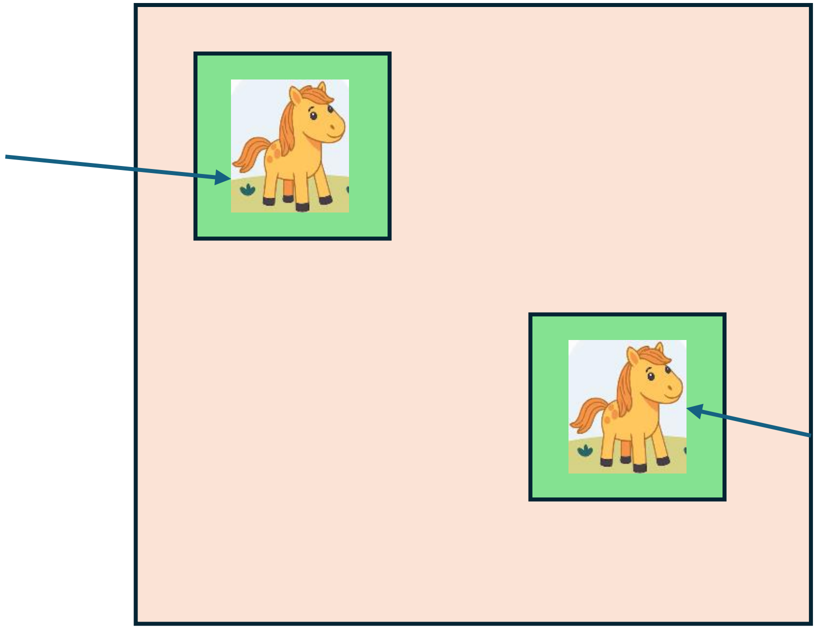
This technique is called **zero-padding**. In an image, for a kernel size $2k+1$ we need exactly k rows and columns of 0 on each side to ensure that the operation is valid for each pixel. Otherwise, the output cannot be computed close to the borders, and the output tensor will have shape $(h-2k, w-2k, c')$. Both are valid options in most frameworks.



Translation Equivariance

In a locally-connected layer, two identical patches can result in different outputs based on their location: some content on pixel $(5, 2)$, for example, will be processed differently than the same content on pixel $(39, 81)$ because the two matrices $\mathbf{W}_{5,2}$ and $\mathbf{W}_{39,81}$ are different. For the most part, however, we can assume that this information is irrelevant: informally, "a horse is a horse", irrespective of its positioning on the input image. We can formalize this with a property called **translation equivariance**.

$P_k(5,2)$



$P_k(39,81)$

In locally-connected layer,

$f(P_k(5,2))$!= $f(P_k(39,81))$

karena $W_{5,2}$ beda dengan $W_{39,81}$

Translation Equivariance

In a locally-connected layer, two identical patches can result in different outputs based on their location: some content on pixel $(5,2)$, for example, will be processed differently than the same content on pixel $(39,81)$ because the two matrices $\mathbf{W}_{5,2}$ and $\mathbf{W}_{39,81}$ are different. For the most part, however, we can assume that this information is irrelevant: informally, "a horse is a horse", irrespective of its positioning on the input image. We can formalize this with a property called **translation equivariance**.

Definition D.7.3 (Translation equivariance) *We say that a layer $H = f(X)$ is **translation equivariant** if:*

$$\underbrace{P_k(i,j) = P_k(i',j')}_{\text{Identical patches}} \quad \text{implies} \quad \underbrace{f(P_k(i,j)) = f(P_k(i',j'))}_{\text{Identical outputs}}$$

Translation Equivariance & Convolutional layers

We do **weight sharing**, ting every position share the same set of weights:

$$H_{ij} = \phi(\underbrace{\boxed{\mathbf{W}}}_{\text{Weight matrix does not depend on } (i,j)} \cdot \operatorname{vect}(P_k(i,j)))$$

Definition D.7.4 (Convolutional layer) *Given an image $X \sim (h, w, c)$ and a kernel size $s = 2k + 1$, a **convolutional layer** $H = \mathrm{Conv2D}(X)$ is defined element-wise by:*

$$H_{ij} = \mathbf{W} \cdot \operatorname{vect}(P_k(i,j)) + \mathbf{b} \tag{E.7.4}$$

The trainable parameters are $\mathbf{W} \sim (c', ssc)$ and $\mathbf{b} \sim (c')$. The hyper-parameters are k, c', and (eventually) whether to apply zero-padding or not. In the former case the output has shape (h, w, c'), in the latter case it has shape $(h - 2k, w - 2k, c')$.

Translation Equivariance & Convolutional layers

We do **weight sharing**, ting every position share the same set of weights:

$$H_{ij} = \mathbf{W} \cdot \text{vect}(P_k(i, j)) + \mathbf{b}$$

Shape = (c')

Weight matrix does not depend on (i, j)

Shape = (c', s * s * c)

Fully-connected layer -> Total number of parameters = **(h * w)² * (c * c')**

Locally-connected layer -> Total number of parameters = **h * w * (s * s * c * c')**

Convolutional layer -> Total number of parameters = **(s * s * c * c') + c'**

Convolutional layer on a 1D sequence

Considering our toy example, assuming for example $k = 1$ (hence $s = 3$) we can write the resulting operation as:

$$\begin{bmatrix} h_1 \\ h_2 \\ h_3 \\ h_4 \end{bmatrix} = \begin{bmatrix} W_{11} & W_{12} & W_{13} & 0 & 0 & 0 \\ 0 & W_{21} & W_{22} & W_{23} & 0 & 0 \\ 0 & 0 & W_{31} & W_{32} & W_{33} & 0 \\ 0 & 0 & 0 & W_{41} & W_{42} & W_{43} \end{bmatrix} \begin{bmatrix} 0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ 0 \end{bmatrix}$$

Locally-connected Layer

$$\begin{bmatrix} h_1 \\ h_2 \\ h_3 \\ h_4 \end{bmatrix} = \begin{bmatrix} W_1 & W_2 & W_3 & 0 & 0 & 0 \\ 0 & W_1 & W_2 & W_3 & 0 & 0 \\ 0 & 0 & W_1 & W_2 & W_3 & 0 \\ 0 & 0 & 0 & W_1 & W_2 & W_3 \end{bmatrix} \begin{bmatrix} 0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ 0 \end{bmatrix}$$

Convolutional Layer

2D Convolutional layer on PyTorch

```
import torch

# random 20 images, each with 3 channels and
# h = 50 pixels & w = 100 pixels
random_images = torch.randn(20, 3, 50, 100)

# c = in_channel = 3
# c′ = out_channel = 6
# kernel size = (3 x 3)
# padding = 'same' meaning that the output must have
#the same shape as the input, 'valid' meaning no padding
conv_layer = torch.nn.Conv2d(3, 6, (3, 3), padding = "same")

hidden = conv_layer(random_images)

print(hidden.shape) #[20, 6, 50, 100]
```

The weight **W** is inside this function, and is initialized randomly.

1D Convolutional Layers for Sequences

Suppose each word is represented as a vector of length 2, meaning that it has 2 channels; and we use a kernel of size 3 and output channel = 3.

Channel 2	Channel 1	
1	2	semangat
0.2	1	belajar
1	1	hingga
3	2	akhir
1	0.5	hidup
0.2	1	kamu

What is the size of our weight matrix **W**?

Suppose each word is represented as a vector of length 2, meaning that it has 2 channels; and we use a kernel of size 3 and output channel = 3.

1	2	semangat
0.2	1	belajar
1	1	hingga
3	2	akhir
1	0.5	hidup
0.2	1	kamu

out channel

in channel * kernel size

0	0	1
1	0	1
0	0	0
0	1	1
1	0	1
1	0	1

Randomly initialized

Suppose each word is represented as a vector of length 2, meaning that it has 2 channels; and we use a kernel of size 3 and output channel = 3.

1	2	semangat
0.2	1	belajar
1	1	hingga
3	2	akhir
1	0.5	hidup
0.2	1	kamu

in channel * kernel size

out channel

0	0	1
1	0	1
0	0	0
0	1	1
1	0	1
1	0	1

Weights associated with input channel 1 & output channel 1

Weights associated with input channel 2 & output channel 1

Randomly initialized

Suppose each word is represented as a vector of length 2, meaning that it has 2 channels; and we use a kernel of size 3 and output channel = 3.

0	0	
1	2	semangat
0.2	1	belajar
1	1	hingga
3	2	akhir
1	0.5	hidup
0.2	1	kamu
0	0	

Suppose we use **zero padding** so that the length of output sequence is the same as that of input sequence.

Suppose each word is represented as a vector of length 2, meaning that it has 2 channels; and we use a kernel of size 3 and output channel = 3.

0	0	
1	2	**semangat**
0.2	1	belajar
1	1	hingga
3	2	akhir
1	0.5	hidup
0.2	1	kamu
0	0	

Flatten, from (in channel, k size) to (in channel * k size)

0	2	1	0	1	0.2

×

W

0	0	1
1	0	1
0	0	0
0	1	1
1	0	1
1	0	1

In this example, we omit the **bias** term for simplicity

Suppose each word is represented as a vector of length 2, meaning that it has 2 channels; and we use a kernel of size 3 and output channel = 3.

0	0	
1	2	**semangat**
0.2	1	belajar
1	1	hingga
3	2	akhir
1	0.5	hidup
0.2	1	kamu
0	0	

3.2	0	3.2

Flattten, from (in channel, k size) to (in channel * k size)

0	2	1	0	1	0.2

×

W

0	0	1
1	0	1
0	0	0
0	1	1
1	0	1
1	0	1

=

3.2	0	3.2

Suppose each word is represented as a vector of length 2, meaning that it has 2 channels; and we use a kernel of size 3 and output channel = 3.

0	0	
1	2	semangat
0.2	1	**belajar**
1	1	hingga
3	2	akhir
1	0.5	hidup
0.2	1	kamu
0	0	

3.2	0	3.2
2.2	1	5.2

$$\begin{bmatrix} 2 & 1 & 1 & 1 & 0.2 & 1 \end{bmatrix} \times \underbrace{\begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 1 \\ 0 & 0 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 0 & 1 \end{bmatrix}}_{\mathbf{W}} = \begin{bmatrix} 2.2 & 1 & 5.2 \end{bmatrix}$$

Suppose each word is represented as a vector of length 2, meaning that it has 2 channels; and we use a kernel of size 3 and output channel = 3.

0	0	
1	2	semangat
0.2	1	belajar
1	1	**hingga**
3	2	akhir
1	0.5	hidup
0.2	1	kamu
0	0	

3.2	0	3.2
2.2	1	5.2
5	0.2	6.2

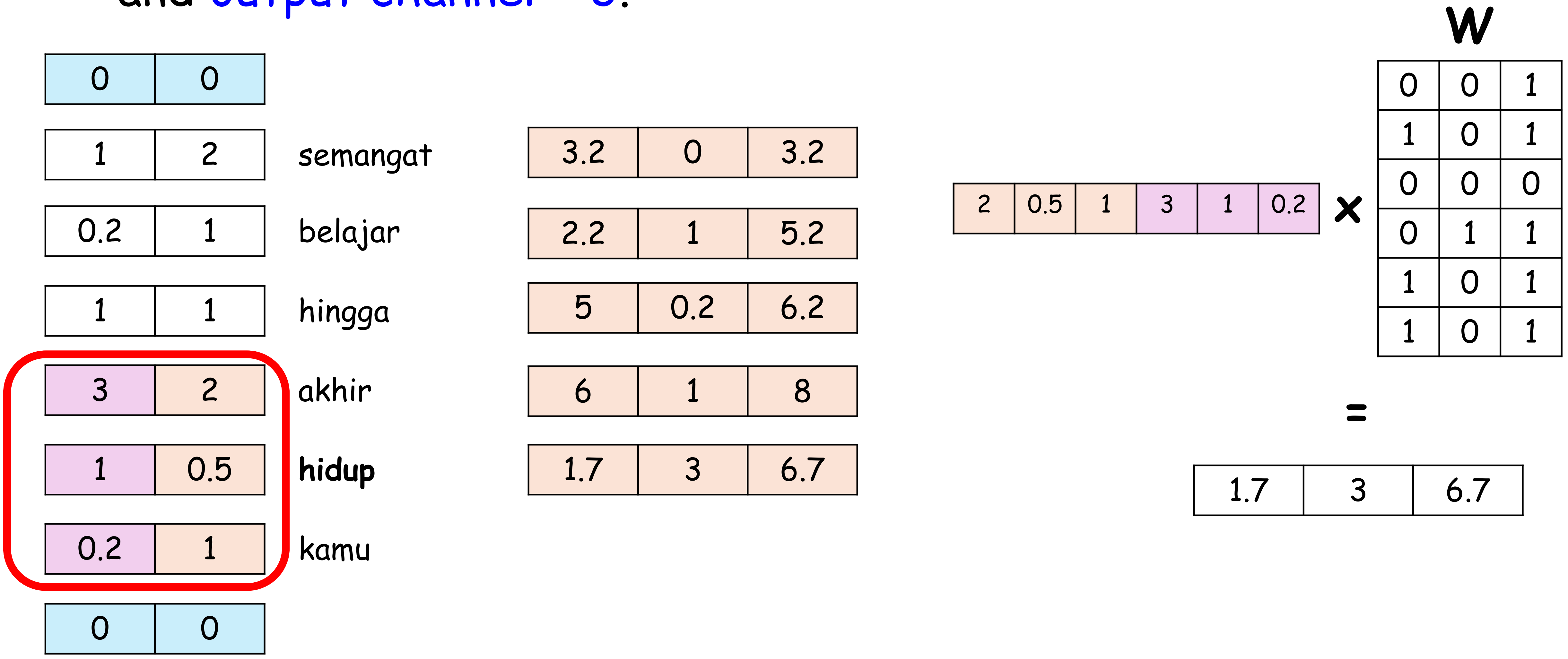
$$\begin{bmatrix} 1 & 1 & 2 & 0.2 & 1 & 3 \end{bmatrix} \times \underbrace{\begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 1 \\ 0 & 0 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 0 & 1 \end{bmatrix}}_{\mathbf{W}} = \begin{bmatrix} 5 & 0.2 & 6.2 \end{bmatrix}$$

Suppose each word is represented as a vector of length 2, meaning that it has 2 channels; and we use a kernel of size 3 and output channel = 3.

0	0				
1	2	semangat	3.2	0	3.2
0.2	1	belajar	2.2	1	5.2
1	1	hingga	5	0.2	6.2
3	2	**akhir**	6	1	8
1	0.5	hidup			
0.2	1	kamu			
0	0				

$$\begin{bmatrix} 1 & 2 & 0.5 & 1 & 3 & 1 \end{bmatrix} \times \underbrace{\begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 1 \\ 0 & 0 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 0 & 1 \end{bmatrix}}_{\mathbf{W}} = \begin{bmatrix} 6 & 1 & 8 \end{bmatrix}$$

Suppose each word is represented as a vector of length 2, meaning that it has 2 channels; and we use a kernel of size 3 and output channel = 3.



Suppose each word is represented as a vector of length 2, meaning that it has 2 channels; and we use a kernel of size 3 and output channel = 3.

0	0				
1	2	semangat	3.2	0	3.2
0.2	1	belajar	2.2	1	5.2
1	1	hingga	5	0.2	6.2
3	2	akhir	6	1	8
1	0.5	hidup	1.7	3	6.7
0.2	1	**kamu**	1.2	1	2.7
0	0				

$$\begin{bmatrix} 0.5 & 1 & 0 & 1 & 0.2 & 0 \end{bmatrix} \times \overset{\mathbf{W}}{\begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 1 \\ 0 & 0 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 0 & 1 \end{bmatrix}} = \begin{bmatrix} 1.2 & 1 & 2.7 \end{bmatrix}$$

Suppose each word is represented as a vector of length 2, meaning that it has 2 channels; and we use a kernel of size 3 and output channel = 3.

1	2	semangat
0.2	1	belajar
1	1	hingga
3	2	akhir
1	0.5	hidup
0.2	1	kamu

Conv1D

3.2	0	3.2
2.2	1	5.2
5	0.2	6.2
6	1	8
1.7	3	6.7
1.2	1	2.7

You can pass them to any other neural network layer that you like.

Here is the precise procedure …

The output value of the layer with input size `(batch size, in channel, sequence len)` and output `(batch size, out channel, sequence len)` can be described as

The number of in channels

$$out(x_i, c_j^{out}) = bias(c_j^{out}) + \sum_{k=0}^{|C^{in}|-1} weight(c_j^{out}, k) \star in(x_i, k)$$

j^{th} value in the out channel of x_i

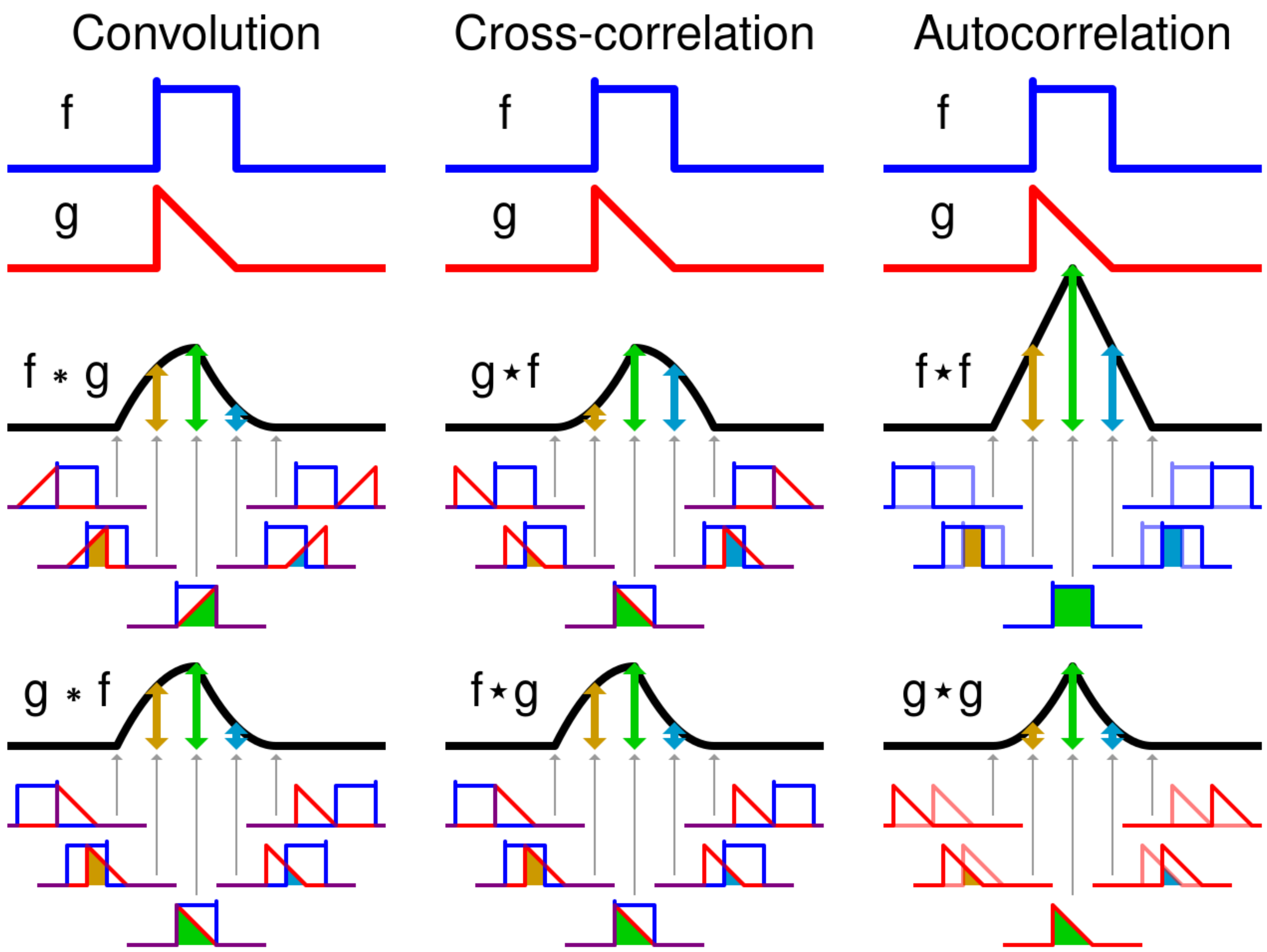
Cross-correlation operator

In signal processing, “Convolution” is different from “Cross-Correlation”.

Yes, You’re right!

“Convolutional NNs” are actually not really “Convolutional” !
They are just doing **“Cross-Correlation”**.

300 point!
Find the difference between “Convolution” and “Cross-Correlation” → https://www.kaggle.com/discussions/general/225375



https://en.wikipedia.org/wiki/Convolution

The codes, from the scratch (well, not really …)

```
import torch
import torch.nn as nn


class Linear(nn.Module):
    def __init__(self, n_inputs, n_outputs):
        super().__init__()
        self.n_inputs = n_inputs
        self.n_outputs = n_outputs
        self.W = nn.Parameter(torch.Tensor(self.n_inputs, self.n_outputs))
        self.init_weights()

    def init_weights(self):
        for param in self.parameters():
            nn.init.uniform_(param, -0.1, 0.1)

    def forward(self, x):
        return x @ self.W
```

For simplicity, we assume no bias

The codes, from the scratch (well, not really …)

```
class Conv1D(nn.Module):
    def __init__(self, in_channels, out_channels, kernel_width):
        """ kernel_width harus ganjil """
        super(Conv1D, self).__init__()
        self.in_channels = in_channels
        self.out_channels = out_channels
        self.kernel_width = kernel_width
        self.pad_size = (self.kernel_width - 1) // 2
        self.kernel = Linear(kernel_width * in_channels, out_channels)

    def forward(self, x):
        # padding
        x = nn.functional.pad(x, (self.pad_size, self.pad_size), "constant", 0)

        l = []
        for i in range(self.pad_size, x.shape[2] - self.pad_size):
            patch = x[:, :, i - self.pad_size: i + self.pad_size + 1]
            patch = patch.reshape(x.shape[0], self.in_channels * self.kernel_width)
            l.append(self.kernel(patch))

        return torch.stack(l, dim=2)
```

Wait, … what is “torch.stack()” ?

```
>>> x = torch.tensor([[1,2,3], [4,5,6]])

>>> torch.stack([x, x], dim=1)

tensor([[[1, 2, 3],
         [1, 2, 3]],

        [[4, 5, 6],
         [4, 5, 6]]])


>>> torch.stack([x, x], dim=2)

tensor([[[1, 1],
         [2, 2],
         [3, 3]],

        [[4, 4],
         [5, 5],
         [6, 6]]])
```

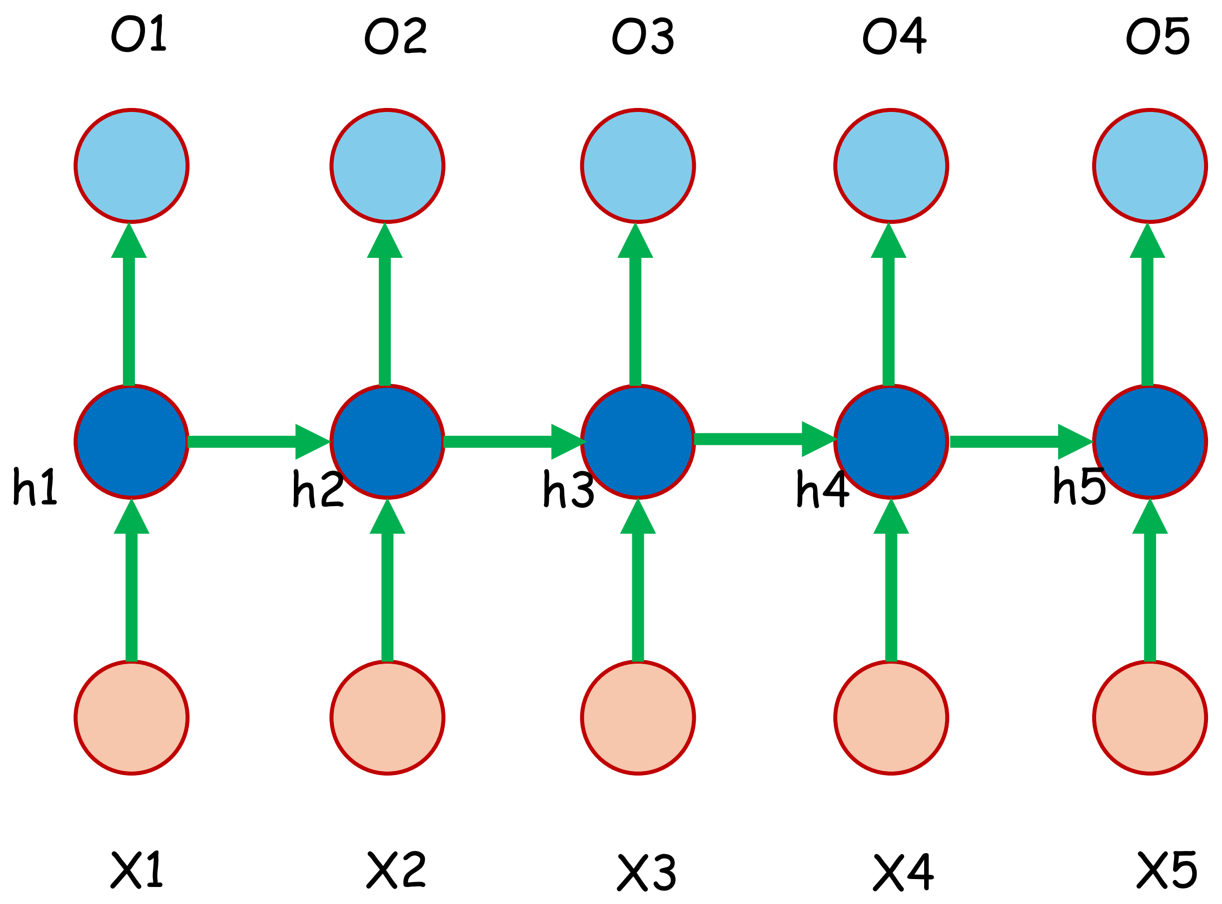
```
>>> torch.stack([x, x], dim=0)

tensor([[[1, 2, 3],
         [4, 5, 6]],

        [[1, 2, 3],
         [4, 5, 6]]])
```

The codes, from the scratch (well, not really …)

```
seq = torch.tensor([[[1.,2.,3.,4.,5.], [5.,4.,3.,2.,1.]],
                    [[1.,2.,3.,4.,5.], [5.,4.,3.,2.,1.]]])

print(seq.shape)                # torch.size([2, 2, 5])

conv = Conv1D(2, 7, 3)

print(conv(seq).shape)          # torch.size([2, 7, 5])
```

Recurrent Neural Networks

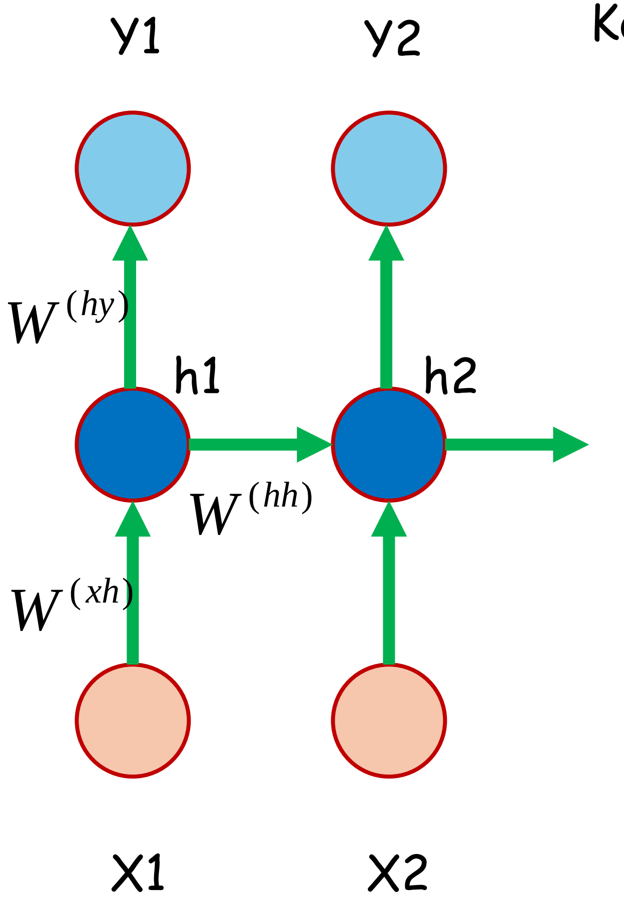
Recurrent Neural Networks



One of the famous Deep Learning Architectures in the NLP community

Recurrent Neural Networks (RNNs)

Misal, ada **I** input unit, **K** output unit, dan **H** hidden unit (state).



Komputasi RNNs untuk **satu sample**:

$h_t \in R^{1 \times H}$ $\quad x_t \in R^{1 \times I}$ $\quad y_t \in R^{1 \times K}$

$W^{xh} \in R^{I \times H}$

$W^{hh} \in R^{H \times H}$

$W^{hy} \in R^{H \times K}$

h_t adalah **contextual word embedding** dari kata **X_t** yang mengandung informasi kata-kata sebelumnya.

$$h_t = \tanh(x_t W^{xh} + h_{t-1} W^{hh})$$

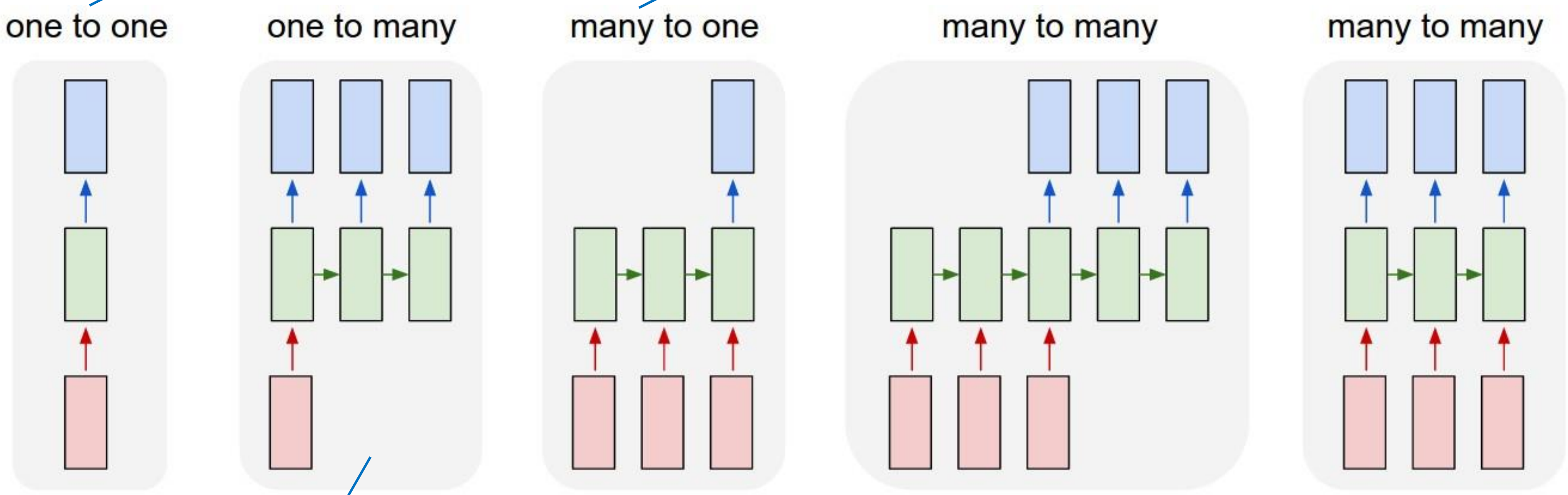
$$h_0 = 0$$

$$y_t = activation(h_t W^{hy})$$

Recurrent Neural Networks (RNNs)

Not RNNs
(Vanilla Feed-Forward NNs)

Sequence Input
(e.g. Sentence Classification)



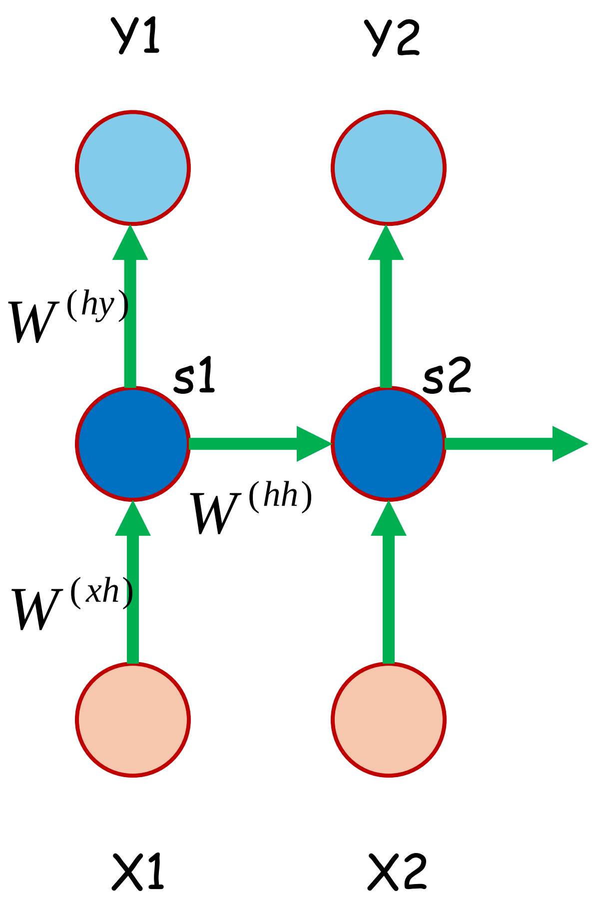
Sequence Output
(e.g. Image Captioning)

Sequence Input/Output
(e.g. Machine Translation)

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

RNNs as Causal Language Models

Vektor representasi hidden pada timestep t, yang berukuran **d x 1** (vektor kolom)



$$P(x_1, \ldots, x_T) = \prod_{t=1}^{T} P(x_t | x_{t-1}, \ldots, x_1)$$

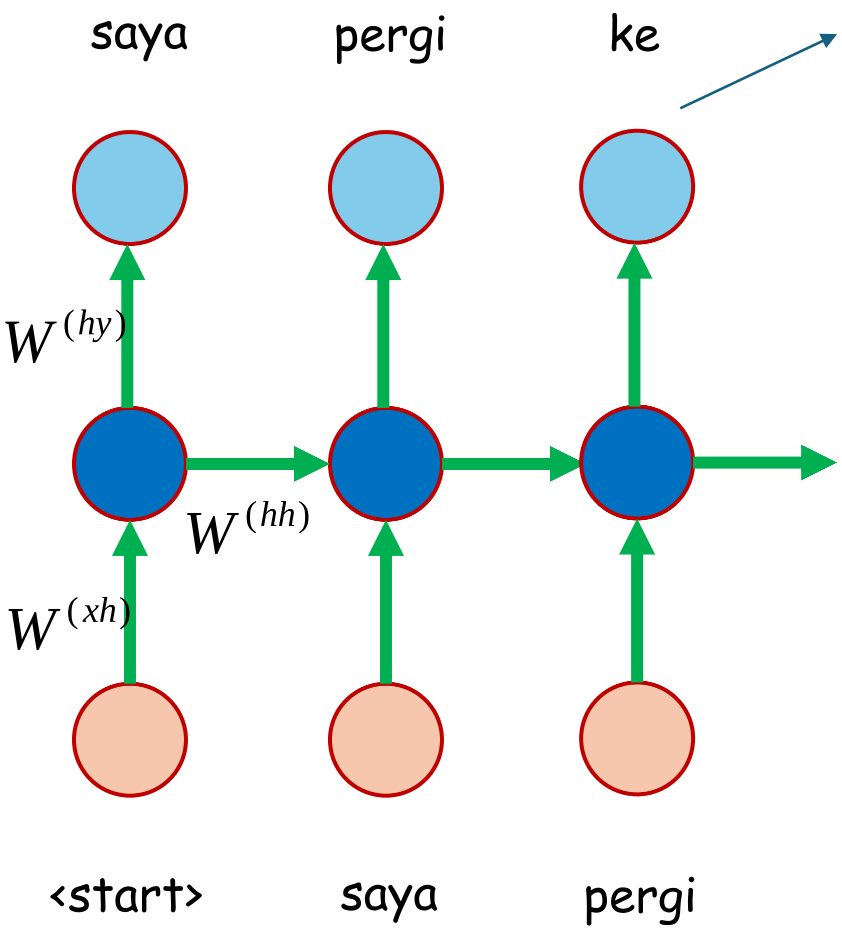
$$P(\boldsymbol{x_t} | x_{t-1}, \ldots, x_1) = \begin{bmatrix} P(x_t = w_1) \\ P(x_t = w_2) \\ \ldots \\ P(x_t = w_{|V|}) \end{bmatrix} = softmax\{W^{(hy)}. s_t\}$$

Matriks berukuran **|V| x d**

Vektor berukuran **|V| x 1**, dimana setiap baris merupakan nilai probabilitas sebuah kata $\mathbf{w_t}$ muncul pada posisi **t**.

$$V = \{w_1, w_2, \ldots w_{|V|}\}$$

RNNs as Causal Language Models



$$\begin{array}{l} \begin{bmatrix} 0.01 \\ 0.03 \\ \mathbf{0.83} \\ \dots \\ 0.06 \end{bmatrix} \end{array} \begin{array}{l} \text{anak} \\ \text{kamu} \\ \text{ke} \\ \\ \text{sepakbola} \end{array}$$

Vektor pada output adalah vektor kolom yang berukuran sebesar ukuran vocabulary.

Matriks parameter pada RNNs dapat dilatih secara **unsupervised**, dengan membuat output RNNs berupa kalimat yang **digeser 1** ke kiri dari kalimat input.

Categorical Cross Entropy dapat digunakan sebagai loss function pada output di setiap timestep.

```python
import torch
import pandas as pd
import numpy as np

from collections import Counter
from torch import nn, optim
from torch.utils.data import DataLoader
```

Tutorial Link:
https://colab.research.google.com/drive/1MzyrG6oJhcuFNblvAuYZ_gTrgJubBwrv?usp=sharing

```python
class RNNCell(nn.Module):
    def __init__(self, n_inputs, n_hiddens):
        super().__init__()
        self.h = Linear(n_hiddens, n_hiddens)
        self.x = Linear(n_inputs, n_hiddens)

    def forward(self, input, hidden):
        return torch.tanh(self.h(hidden) + self.x(input))
```

Tutorial Link:
https://colab.research.google.com/drive/1MzyrG6oJhcuFNblvAuYZ_gTrgJubBwrv?usp=sharing

```python
class RNN(nn.Module):
    def __init__(self, n_inputs, n_hiddens, cell):
        super().__init__()
        self.n_inputs = n_inputs
        self.n_hiddens = n_hiddens
        self.cell = cell(n_inputs, n_hiddens)

    def forward(self, inputs, prev_hidden_state):
        """ inputs: [batch_size, sequence_length, embedding_size] """
        outputs = []
        hidden_state = prev_hidden_state
        n_steps = inputs.shape[1]
        for i in range(n_steps):
            hidden_state = self.cell(inputs[:, i], hidden_state)
            outputs.append(hidden_state)

        return torch.stack(outputs, dim=1), hidden_state
```

Tutorial Link:
https://colab.research.google.com/drive/1MzyrG6oJhcuFNblvAuYZ_gTrgJubBwrv?usp=sharing

```python
class Dataset(torch.utils.data.Dataset):
    def __init__(
        self,
        sequence_length,
        documents, # list of strings
    ):
        self.sequence_length = sequence_length
        self.words = self.load_words(documents)
        self.uniq_words = self.get_uniq_words()

        self.index_to_word = {index: word for index, word in enumerate(self.uniq_words)}
        self.word_to_index = {word: index for index, word in enumerate(self.uniq_words)}

        self.words_indexes = [self.word_to_index[w] for w in self.words]

    def load_words(self, documents):
        text = ""
        for doc in documents:
          text += doc + " "
        return text.split(' ')
```

Tutorial Link:
https://colab.research.google.com/drive/1MzyrG6oJhcuFNblvAuYZ_gTrgJubBwrv?usp=sharing

```python
def get_uniq_words(self):
    word_counts = Counter(self.words)
    return sorted(word_counts, key=word_counts.get, reverse=True)

def __len__(self):
    return len(self.words_indexes) - self.sequence_length

def __getitem__(self, index):
    return (
        torch.tensor(self.words_indexes[index:index+self.sequence_length]),
        torch.tensor(self.words_indexes[index+1:index+self.sequence_length+1]),
    )
```

Tutorial Link:
https://colab.research.google.com/drive/1MzyrG6oJhcuFNblvAuYZ_gTrgJubBwrv?usp=sharing

```python
class Model(nn.Module):
    def __init__(self, dataset):
        super(Model, self).__init__()
        self.rnn_size = 16
        self.embedding_dim = 16
        self.n_vocab = len(dataset.uniq_words)

        self.embedding = nn.Embedding(
            num_embeddings=self.n_vocab,
            embedding_dim=self.embedding_dim
        )
        self.rnn = RNN(
            n_inputs=self.embedding_dim,
            n_hiddens=self.rnn_size,
            cell=RNNCell
        )
        self.fc = Linear(self.rnn_size, self.n_vocab)
```

Tutorial Link:
https://colab.research.google.com/drive/1MzyrG6oJhcuFNblvAuYZ_gTrgJubBwrv?usp=sharing

```python
def forward(self, x, prev_state):
    embed = self.embedding(x)
    output, state = self.rnn(embed, prev_state)
    logits = self.fc(output)
    return logits, state

def init_state(self, batch_size):
    return torch.zeros(batch_size, self.rnn_size)
```

Tutorial Link:
https://colab.research.google.com/drive/1MzyrG6oJhcuFNblvAuYZ_gTrgJubBwrv?usp=sharing

```python
def train(dataset, model, batch_size, max_epochs=400):
    model.train()

    dataloader = DataLoader(dataset, batch_size=batch_size)
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.Adam(model.parameters(), lr=0.001)

    for epoch in range(max_epochs):
        h_state = model.init_state(batch_size) #hidden state awal, NOL

        for batch, (x, y) in enumerate(dataloader):
            y_pred, h_state = model(x, h_state)
            loss = criterion(y_pred.transpose(1, 2), y)

            loss.backward()
            optimizer.step()

            #h_state detached from current graph; tapi isi tetap sama agar berlanjut
            #graph batch sekarang jangan nyambung dengan batch berikutnya; tetapi nilai
            #h_state harus berlanjut dari satu batch ke batch berikutnya. Caranya adalah
            #dengan detach() ini.   ---> keberlanjutan state ini disebut “statefull”
            h_state = h_state.detach()

            optimizer.zero_grad()
```

```python
def predict(dataset, model, text, next_words=20):
    model.eval()

    words = text.split(' ')
    h_state = model.init_state(len(words))

    for i in range(0, next_words):
        x = torch.tensor([[dataset.word_to_index[w] for w in words[i:]]])
        y_pred, h_state = model(x, h_state)

        last_word_logits = y_pred[0][-1]
        p = torch.nn.functional.softmax(last_word_logits, dim=0).detach().numpy()

        # random choice
        #word_index = np.random.choice(len(last_word_logits), p=p)

        # the best one
        word_index = np.argmax(p)

        words.append(dataset.index_to_word[word_index])

    return words
```

```python
documents = ["saya pergi ke depok",
             "di depok makan sayuran",
             "dan buah nangka yang segar",
             "angin bertiup kencang",
             "tanda hujan akan turun di jalan margonda"]


# make a dataset
dataset = Dataset(2, documents)

# the model
model = Model(dataset)

# train
train(dataset, model, 2)

# try prompt the model with "saya pergi"
print(predict(dataset, model, "saya pergi", next_words=20))
```

Tutorial Link:
https://colab.research.google.com/drive/1MzyrG6oJhcuFNblvAuYZ_gTrgJubBwrv?usp=sharing

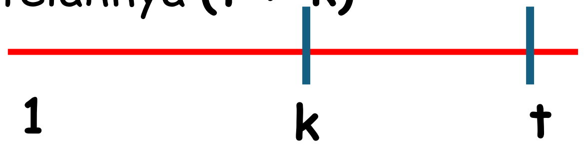
Recurrent Neural Networks (RNNs)

Back Propagation Through Time (BPTT)

Misal, untuk parameter antar state:

$$\frac{\partial L_t}{\partial W^{(hh)}} = \sum_{k=1}^{t} \frac{\partial L_t}{\partial h_t} \cdot \frac{\partial h_t}{\partial h_k} \cdot \frac{\partial^+ h_k}{\partial W^{(hh)}}$$

Term-term ini disebut **temporal contribution**: bagaimana $\mathbf{W}^{(hh)}$ pada step **k** mempengaruhi cost pada step-step setelahnya **(t > k)**



Diputus sampai **k step** ke belakang. Di sini artinya "immediate derivative", yaitu $\mathbf{h}_{k-1}$ dianggap konstan terhadap $\mathbf{W}^{(hh)}$.

$$\frac{\partial h_t}{\partial h_k} = \frac{\partial h_t}{\partial h_{t-1}} \cdot \frac{\partial h_{t-1}}{\partial h_{t-2}} \cdots \frac{\partial h_{k+2}}{\partial h_{k+1}} \cdot \frac{\partial h_{k+1}}{\partial h_k}$$

$$\frac{\partial^+ h_k}{\partial W^{(hh)}} = \frac{\partial^+ \left(W^{(xh)} \cdot x_t + W^{(hh)} \cdot s_{t-1}\right)}{\partial W^{(hh)}} = s_{t-1}$$

Recurrent Neural Networks (RNNs)

Vanishing & Exploding Gradient Problems

Bengio et al., (1994) said that "*the **exploding gradients problem** refers to the large increase in the norm of the gradient during training. Such events are caused by the explosion of the long term components, which can grow exponentially more then short term ones.*"

*And "The **vanishing gradients problem** refers to the opposite behaviour, when long term components go exponentially fast to norm 0, making it impossible for the model to learn correlation between temporally distant events."*

Kok bisa terjadi? Coba lihat salah satu temporal component dari sebelumnya:

$$\frac{\partial h_t}{\partial h_k} = \frac{\partial h_t}{\partial h_{t-1}} \cdot \frac{\partial h_{t-1}}{\partial h_{t-2}} \cdots \frac{\partial h_{k+2}}{\partial h_{k+1}} \cdot \frac{\partial h_{k+1}}{\partial h_k}$$

*In the same way a product of **t - k** real numbers can shrink to zero or explode to infinity, so does this product of Matrices. (**Pascanu et al.,**)*

Bengio, Y., Simard, P., and Frasconi, P. (1994). Learning long-term dependencies with gradient descent is difficult. IEEE Transactions on Neural Networks

Recurrent Neural Networks (RNNs)

Vanishing & Exploding Gradient Problems

Sequential Jacobian biasa digunakan untuk analisis penggunaan konteks pada RNNs.

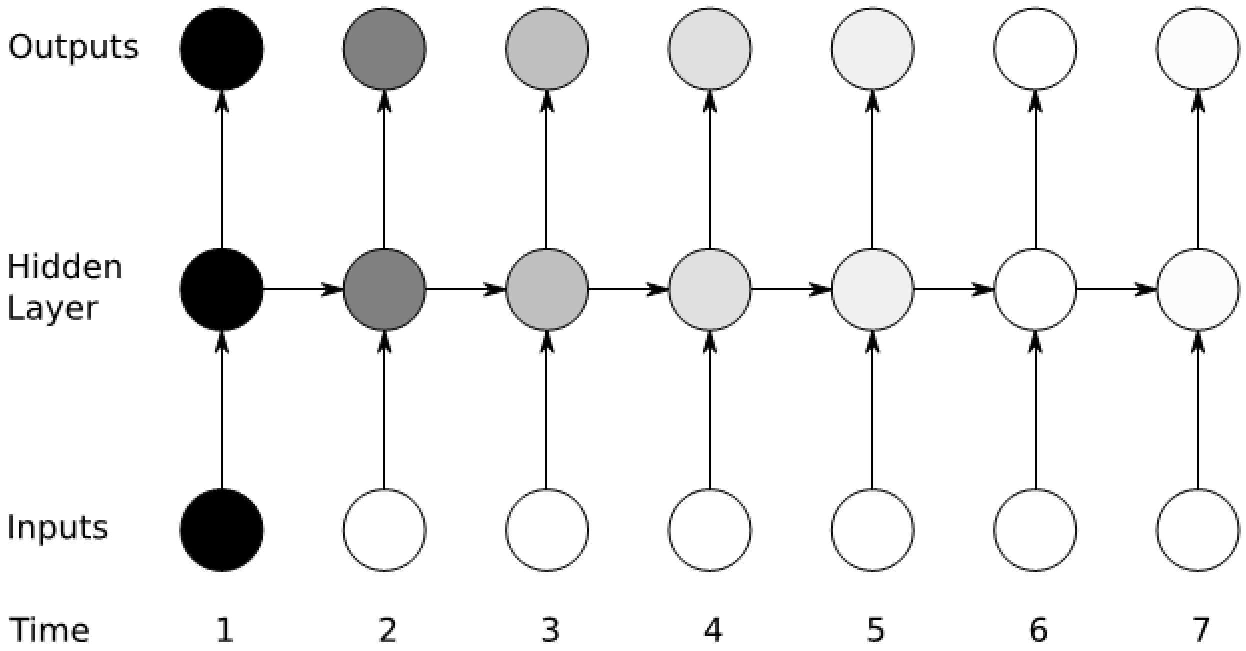


Figure 4.1: **The vanishing gradient problem for RNNs.** The shading of the nodes in the unfolded network indicates their sensitivity to the inputs at time one (the darker the shade, the greater the sensitivity). The sensitivity decays over time as new inputs overwrite the activations of the hidden layer, and the network 'forgets' the first inputs.

Alex Graves, Supervised Sequence Labelling with Recurrent Neural Networks

Recurrent Neural Networks (RNNs)

Solusi untuk Vanishing Gradient Problem

1) Penggunaan ***non-gradient*** *based training algorithms* (Simulated Annealing, Discrete Error Propagation, etc.) **(Bengio et al., 1994)**

2) Definisikan arsitektur baru di dalam **RNN Cell**!, seperti **Long-Short Term Memory (LSTM) (Hochreiter & Schmidhuber, 1997)**.

3) Untuk metode yang lain, silakan merujuk **(Pascanu et al., 2013)**.

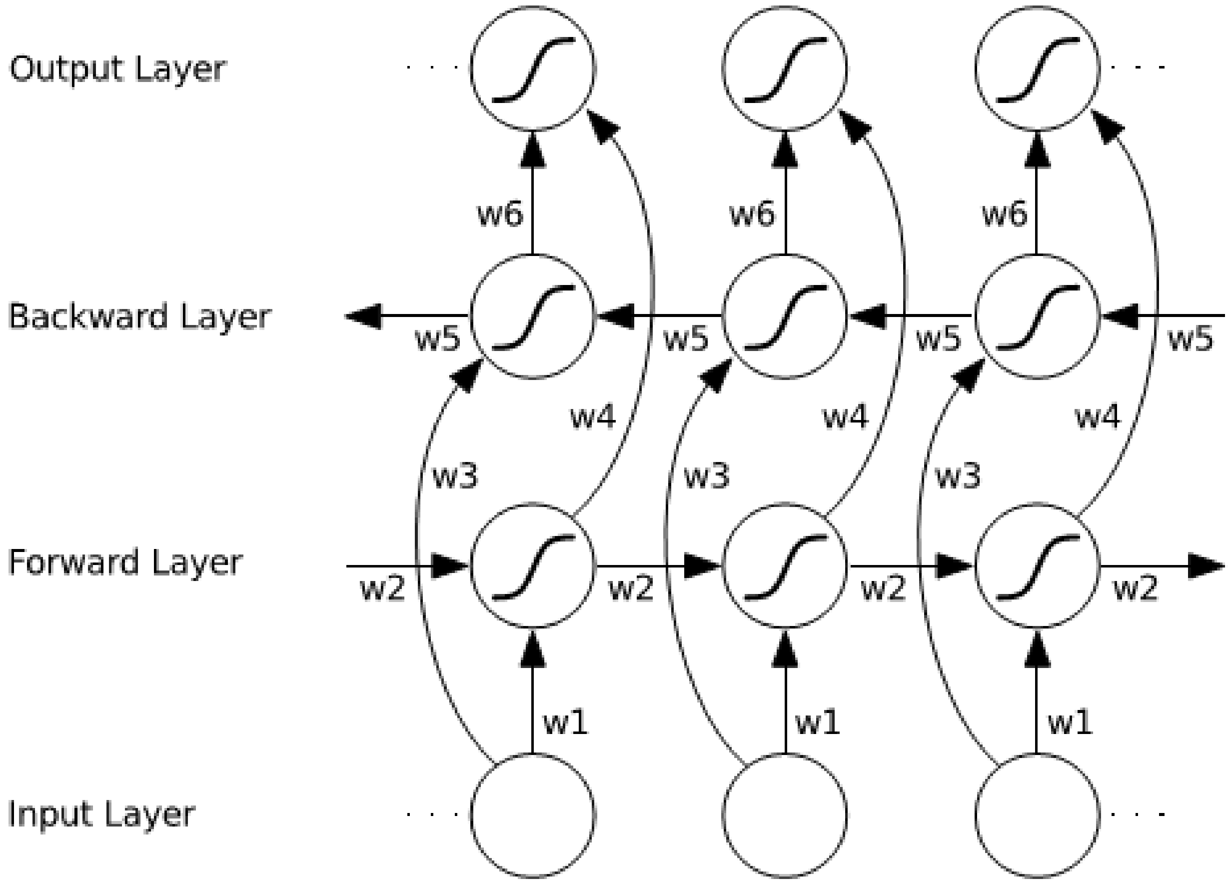
Pascanu et al., On the difficulty of training Recurrent Neural Networks, 2013

S. Hochreiter and J. Schmidhuber. Long Short-Term Memory. Neural Computation,9(8):1735 1780, 1997

Y. Bengio, P. Simard, and P. Frasconi. Learning Long-Term Dependencies with Gradient Descent is Difficult. IEEE Transactions on Neural Networks, 1994

Recurrent Neural Networks (RNNs)

Variant: Bi-Directional RNNs



Alex Graves, *Supervised Sequence Labelling with Recurrent Neural Networks*

Long Short Term Memory Networks

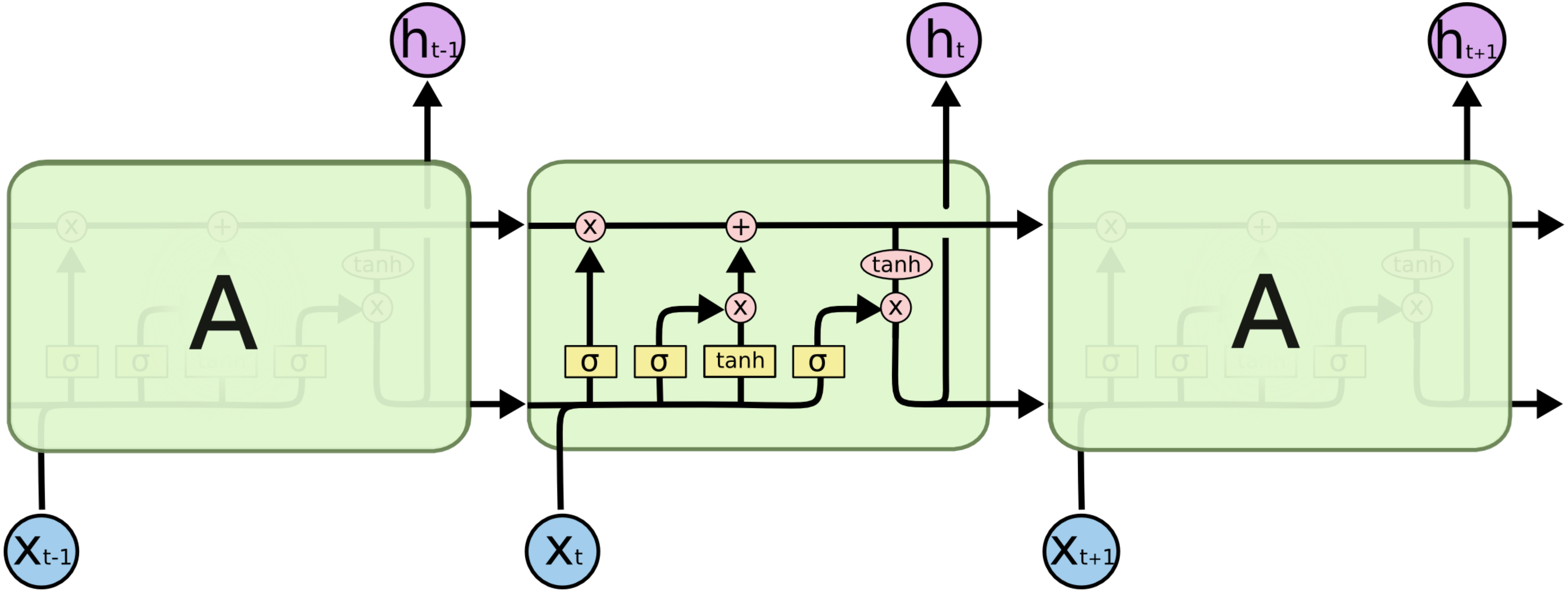
Long-Short Term Memory (LSTM)

1. The LSTM architecture consists of a set of **recurrently connected subnets**, known as **memory blocks**.
2. These blocks can be thought of as a differentiable version of the memory chips in a digital computer.
3. Each block contains:
   1. Self-connected memory cells
   2. Three **multiplicative** units (gates)
      1. Input gates (analogue of write operation)
      2. Output gates (analogue of read operation)
      3. Forget gates ((analogue of reset operation))

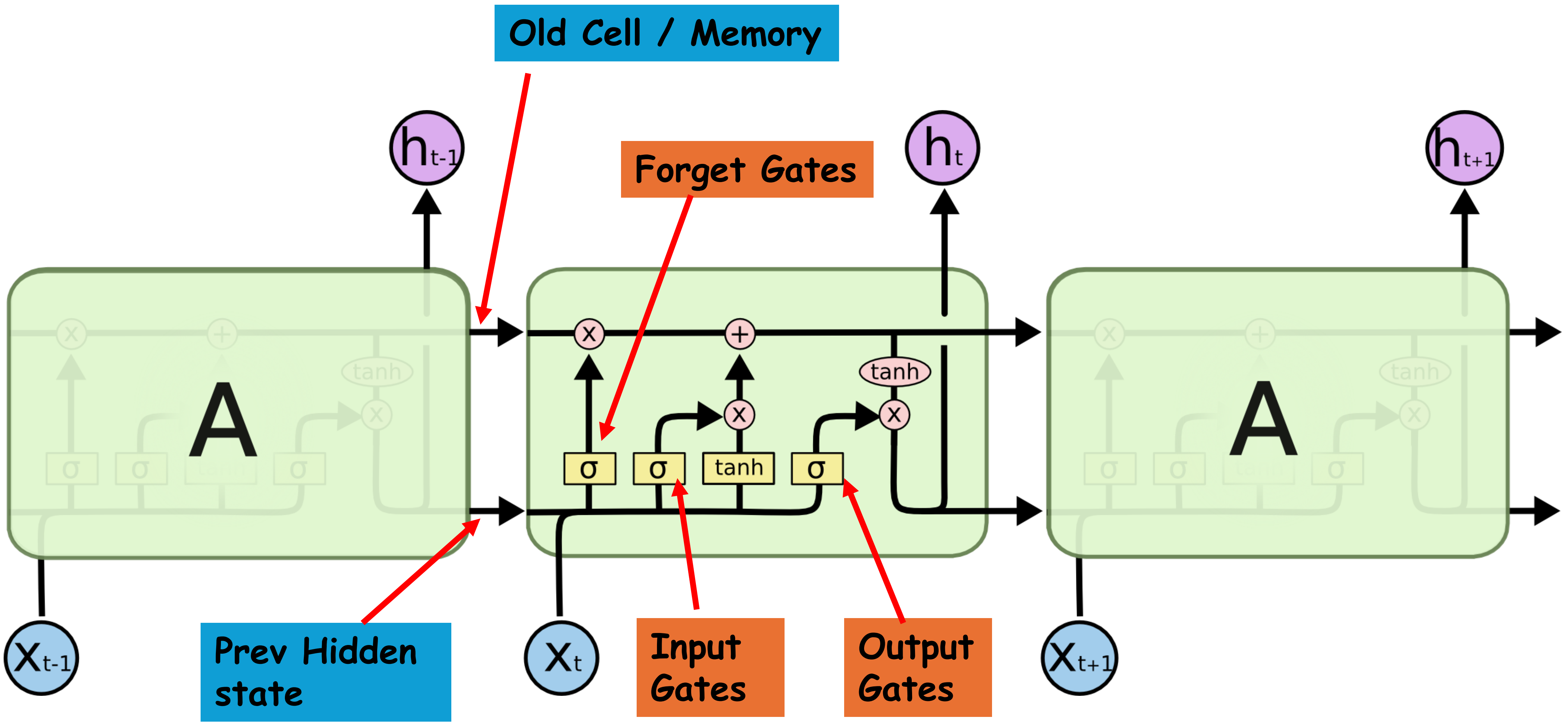
S. Hochreiter and J. Schmidhuber. Long Short-Term Memory. Neural Computation,9(8):1735 1780, 1997

Long-Short Term Memory (LSTM)

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

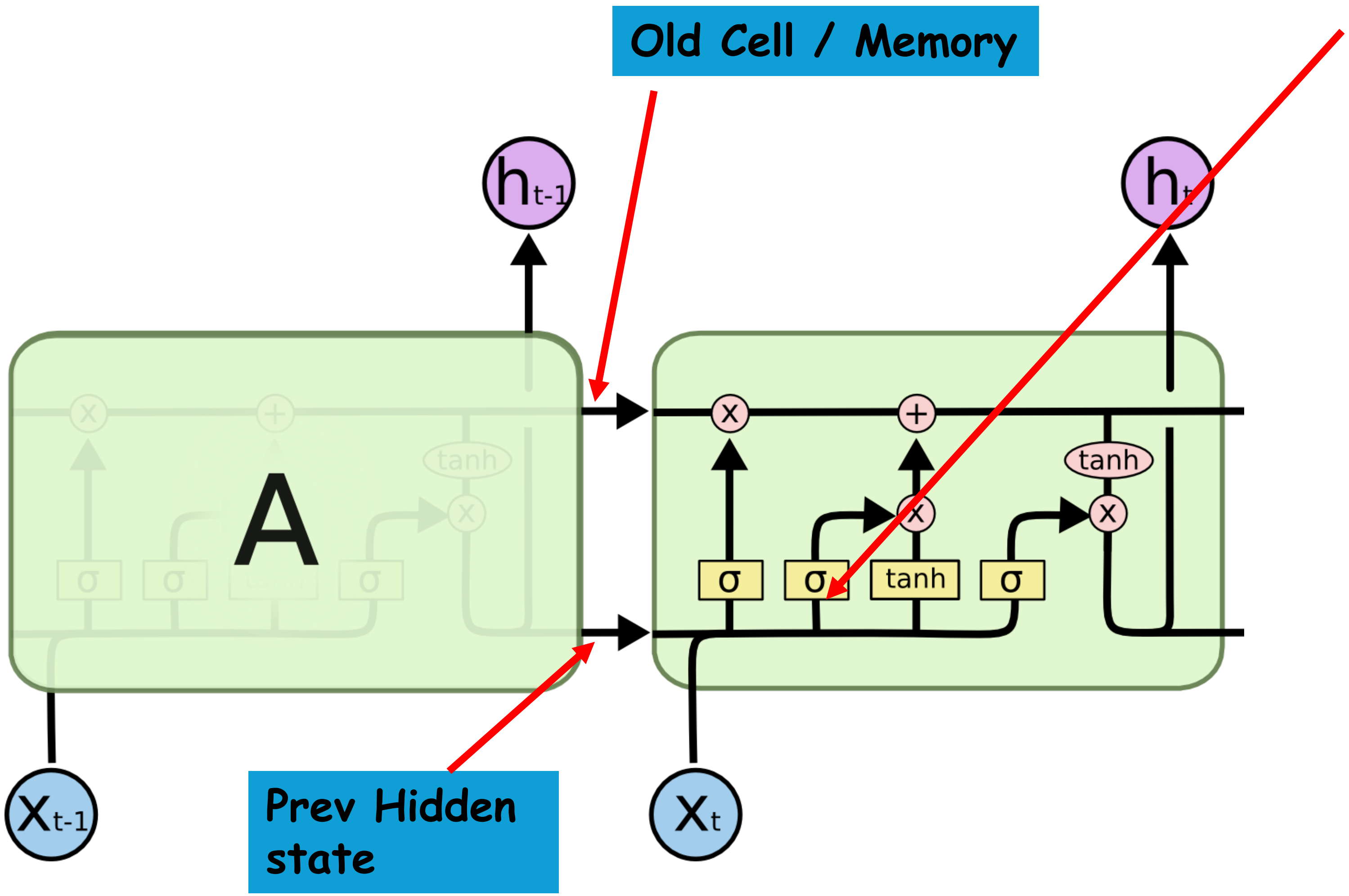


Long-Short Term Memory (LSTM)



Long-Short Term Memory (LSTM)

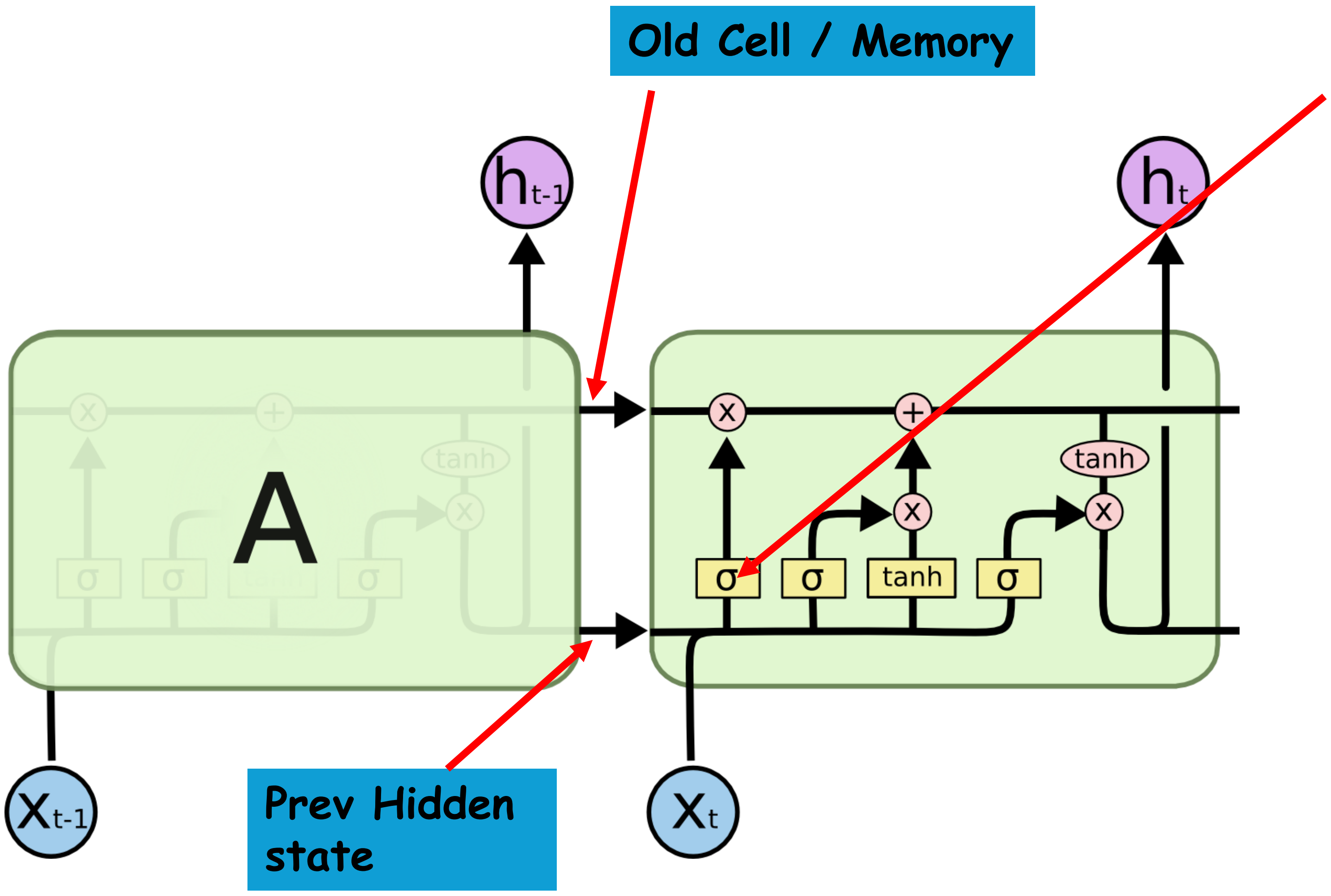
$$i_t = \sigma(W_{ii}x_t + W_{hi}h_{t-1} + b_i)$$



Long-Short Term Memory (LSTM)

$$i_t = \sigma(W_{ii}x_t + W_{hi}h_{t-1} + b_i)$$

$$f_t = \sigma\left(W_{if}x_t + W_{hf}h_{t-1} + b_f\right)$$

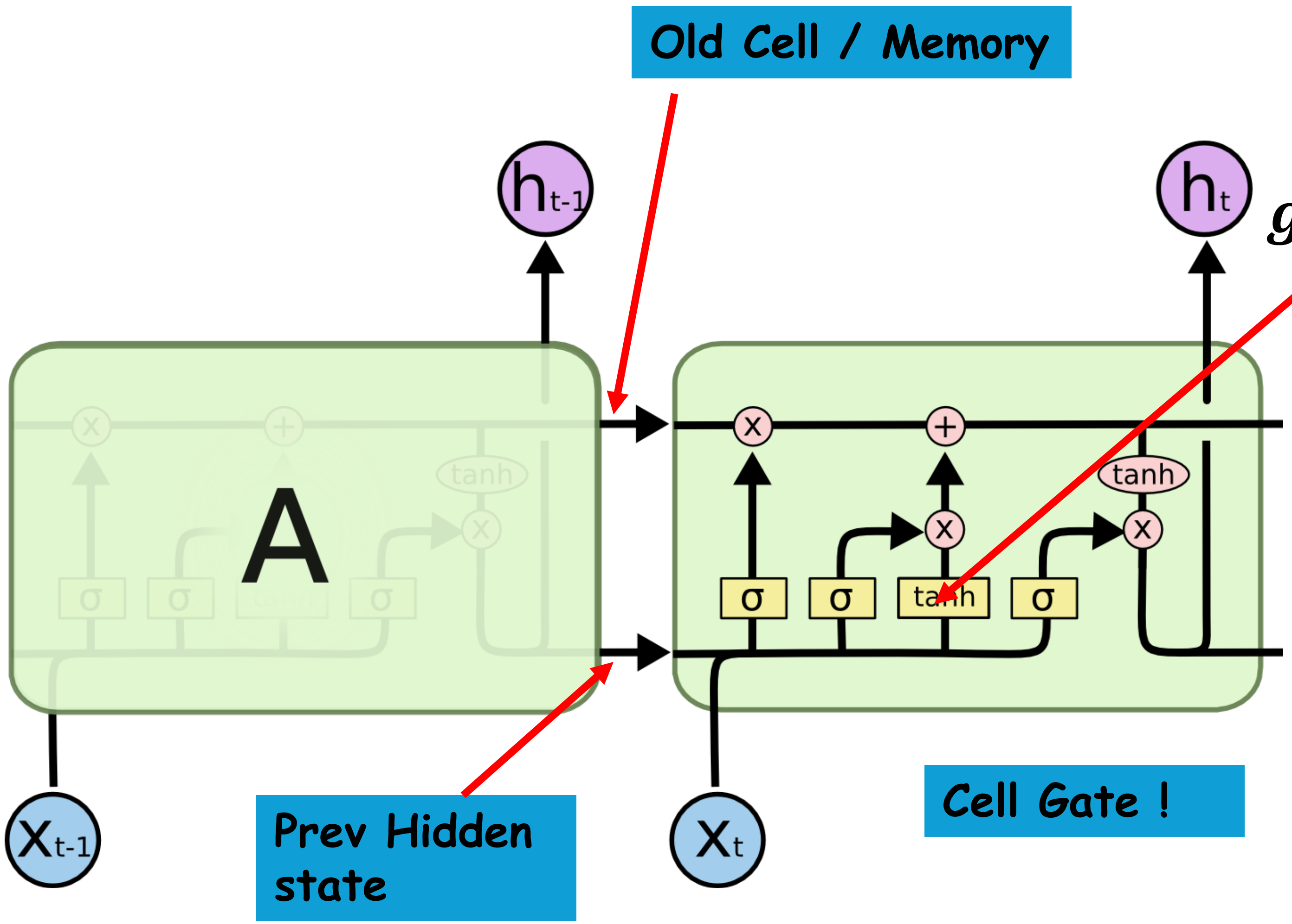


Long-Short Term Memory (LSTM)

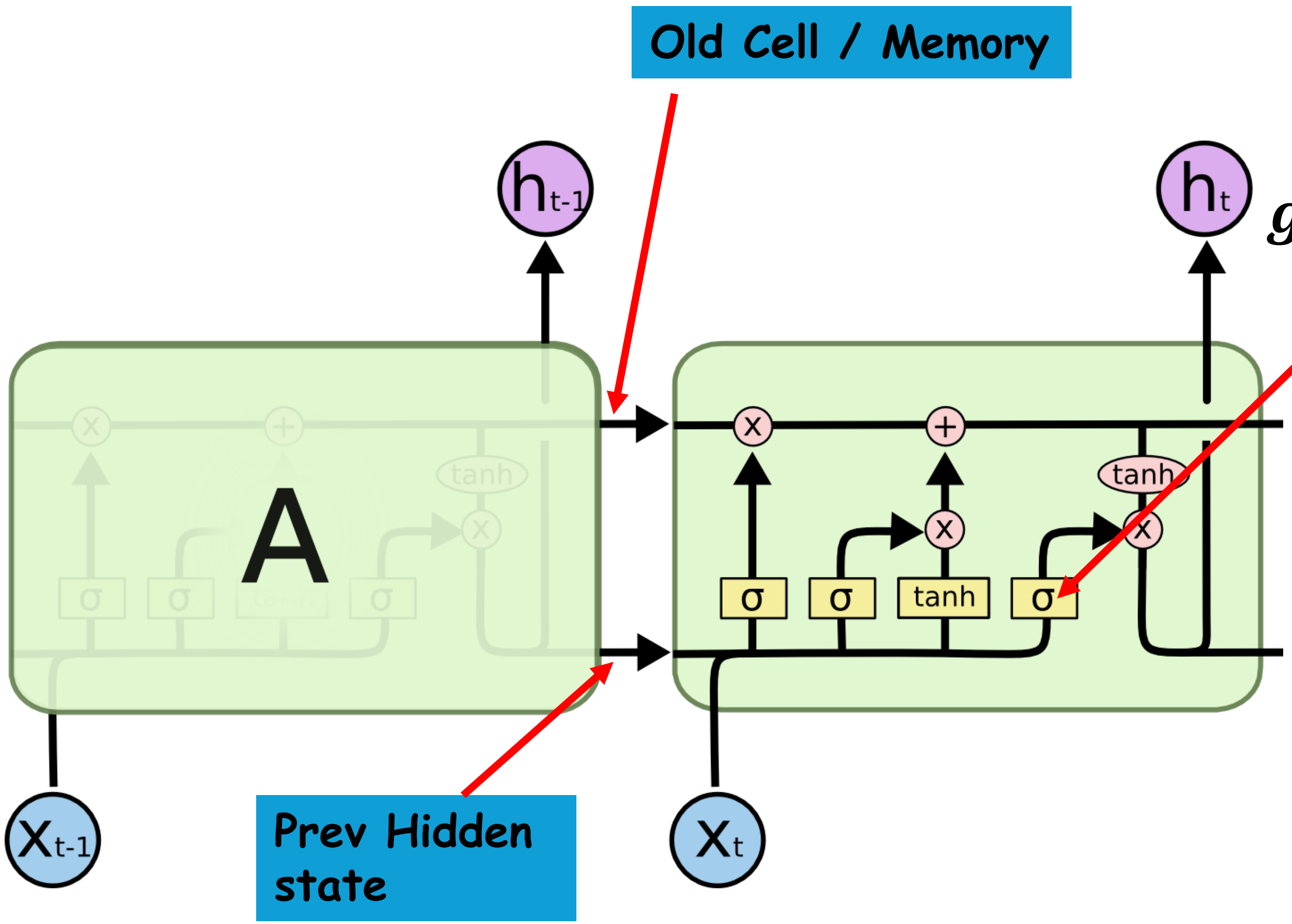
$$i_t = \sigma(W_{ii}x_t + W_{hi}h_{t-1} + b_i)$$

$$f_t = \sigma(W_{if}x_t + W_{hf}h_{t-1} + b_f)$$

$$g_t = tanh(W_{ig}x_t + W_{hg}h_{t-1} + b_g)$$



Long-Short Term Memory (LSTM)



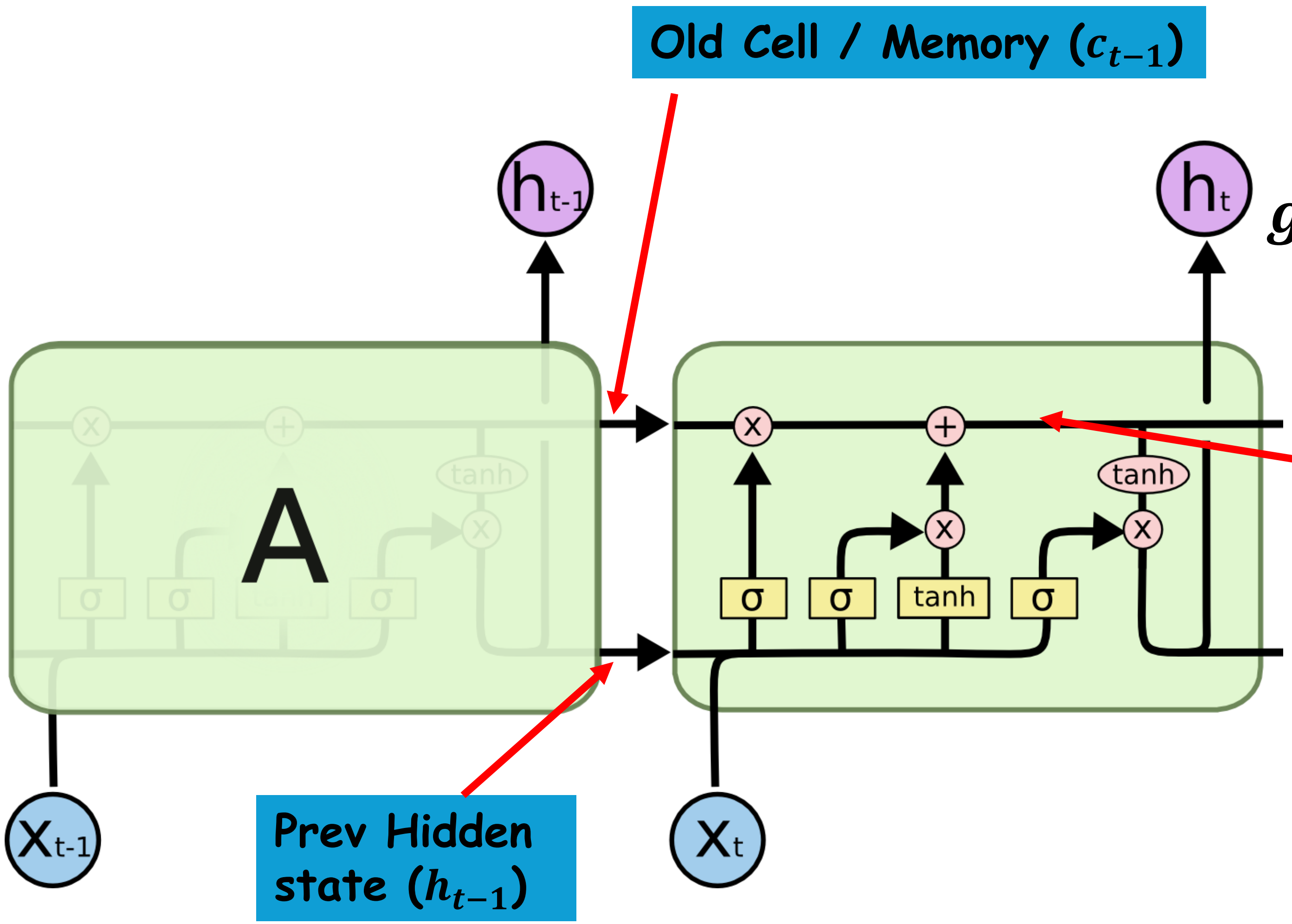
$$i_t = \sigma(W_{ii}x_t + W_{hi}h_{t-1} + b_i)$$

$$f_t = \sigma(W_{if}x_t + W_{hf}h_{t-1} + b_f)$$

$$g_t = tanh(W_{ig}x_t + W_{hg}h_{t-1} + b_g)$$

$$o_t = \sigma(W_{io}x_t + W_{ho}h_{t-1} + b_o)$$

Long-Short Term Memory (LSTM)



$$i_t = \sigma(W_{ii}x_t + W_{hi}h_{t-1} + b_i)$$

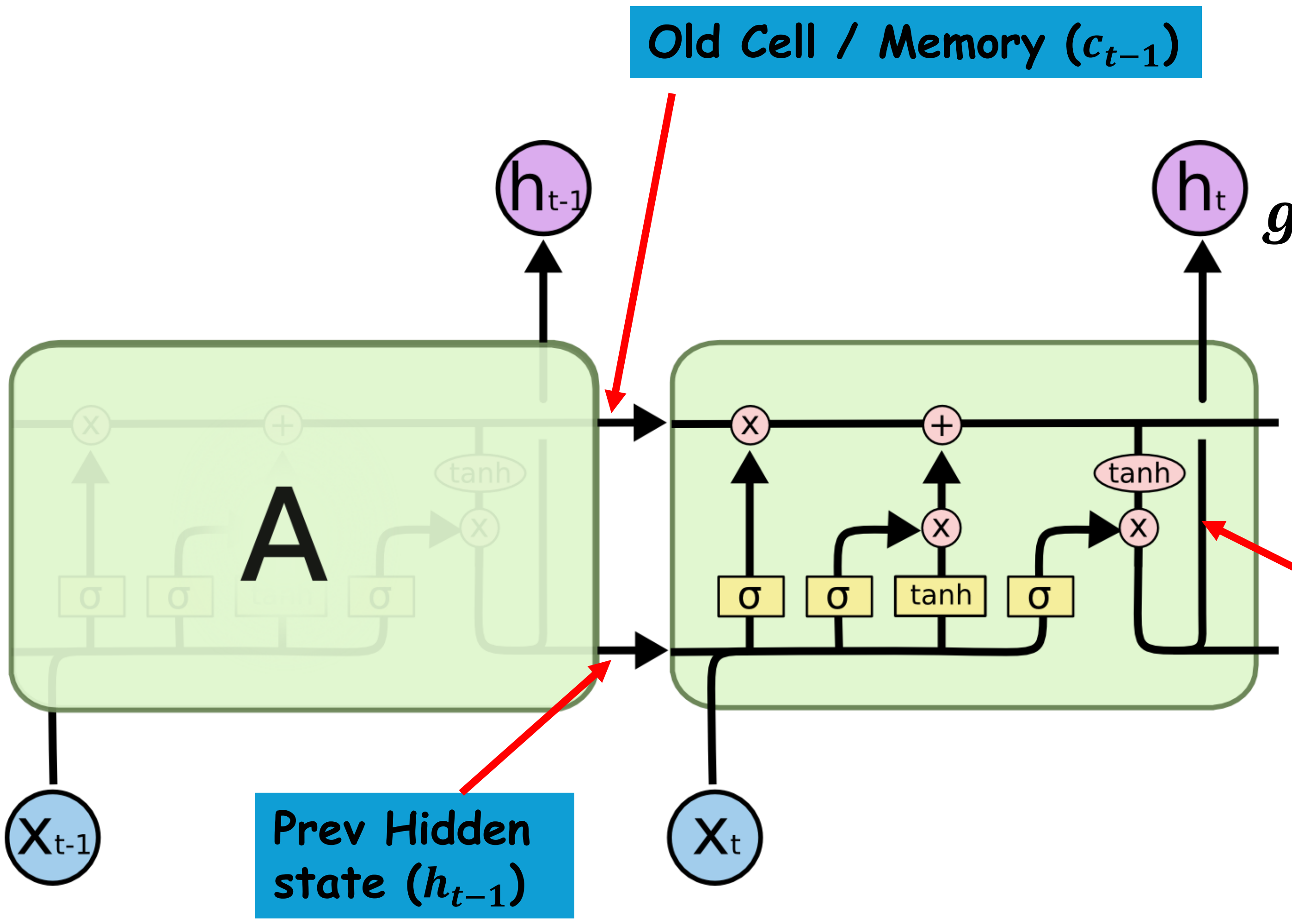
$$f_t = \sigma(W_{if}x_t + W_{hf}h_{t-1} + b_f)$$

$$g_t = tanh(W_{ig}x_t + W_{hg}h_{t-1} + b_g)$$

$$o_t = \sigma(W_{io}x_t + W_{ho}h_{t-1} + b_o)$$

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

Long-Short Term Memory (LSTM)



$$i_t = \sigma(W_{ii}x_t + W_{hi}h_{t-1} + b_i)$$

$$f_t = \sigma(W_{if}x_t + W_{hf}h_{t-1} + b_f)$$

$$g_t = tanh(W_{ig}x_t + W_{hg}h_{t-1} + b_g)$$

$$o_t = \sigma(W_{io}x_t + W_{ho}h_{t-1} + b_o)$$

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

$$h_t = o_t * tanh(c_t)$$

```python
class LSTMCell(nn.Module):
    def __init__(self, input_size, hidden_size):
        super(LSTMCell, self).__init__()
        self.hidden_size = hidden_size
        self.input_size = input_size

        # Input gate components
        self.W_ii = nn.Parameter(torch.Tensor(hidden_size, input_size))
        self.W_hi = nn.Parameter(torch.Tensor(hidden_size, hidden_size))
        self.b_i = nn.Parameter(torch.Tensor(hidden_size))

        # Forget gate components
        self.W_if = nn.Parameter(torch.Tensor(hidden_size, input_size))
        self.W_hf = nn.Parameter(torch.Tensor(hidden_size, hidden_size))
        self.b_f = nn.Parameter(torch.Tensor(hidden_size))

        # Cell gate components
        self.W_ig = nn.Parameter(torch.Tensor(hidden_size, input_size))
        self.W_hg = nn.Parameter(torch.Tensor(hidden_size, hidden_size))
        self.b_g = nn.Parameter(torch.Tensor(hidden_size))
```

Tutorial Link (LSTMs for causal language modeling):
https://colab.research.google.com/drive/1aY4R_zq-Bw0HcW9L1q6WN0JYIxwsA-bN?usp=sharing

```python
... Continued
        # Output gate components
        self.W_io = nn.Parameter(torch.Tensor(hidden_size, input_size))
        self.W_ho = nn.Parameter(torch.Tensor(hidden_size, hidden_size))
        self.b_o = nn.Parameter(torch.Tensor(hidden_size))

        self.init_weights()

    def init_weights(self):
        for param in self.parameters():
            nn.init.uniform_(param, -0.1, 0.1)

    def forward(self, x, hidden):
        h_prev, c_prev = hidden

        i_t = torch.sigmoid(x @ self.W_ii.T + h_prev @ self.W_hi.T + self.b_i)
        f_t = torch.sigmoid(x @ self.W_if.T + h_prev @ self.W_hf.T + self.b_f)
        g_t = torch.tanh(x @ self.W_ig.T + h_prev @ self.W_hg.T + self.b_g)
        o_t = torch.sigmoid(x @ self.W_io.T + h_prev @ self.W_ho.T + self.b_o)

        c_t = f_t * c_prev + i_t * g_t
        h_t = o_t * torch.tanh(c_t)

        return (h_t, c_t)
```

```python
class LSTM(nn.Module):
    def __init__(self, input_size, hidden_size):
        super(LSTM, self).__init__()
        self.hidden_size = hidden_size
        self.cell = LSTMCell(input_size, hidden_size)

    def forward(self, x, prev_state):
        batch_size, seq_len, _ = x.size()
        h, c = prev_state

        outputs = []
        for t in range(seq_len):
            x_t = x[:, t, :]
            (h, c) = self.cell(x_t, (h, c))
            outputs.append(h)

        return torch.stack(outputs, dim=1), (h, c)
```

Tutorial Link (LSTMs for causal language modeling):

https://colab.research.google.com/drive/1aY4R_zq-Bw0HcW9L1q6WN0JYIxwsA-bN?usp=sharing

Long-Short Term Memory (LSTM)

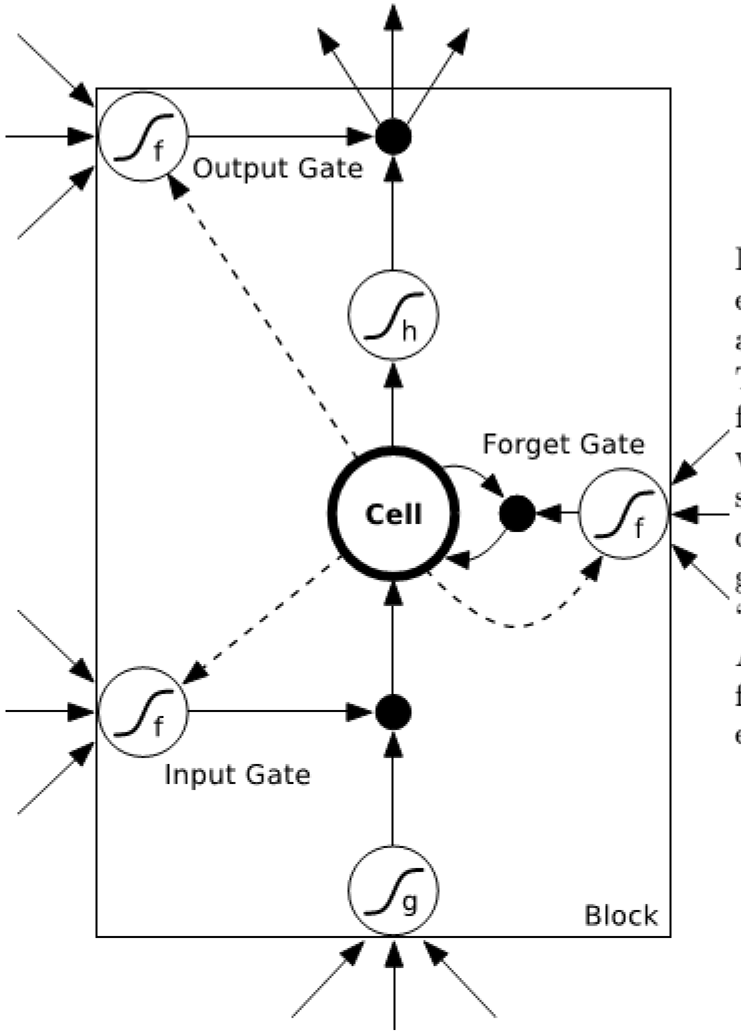
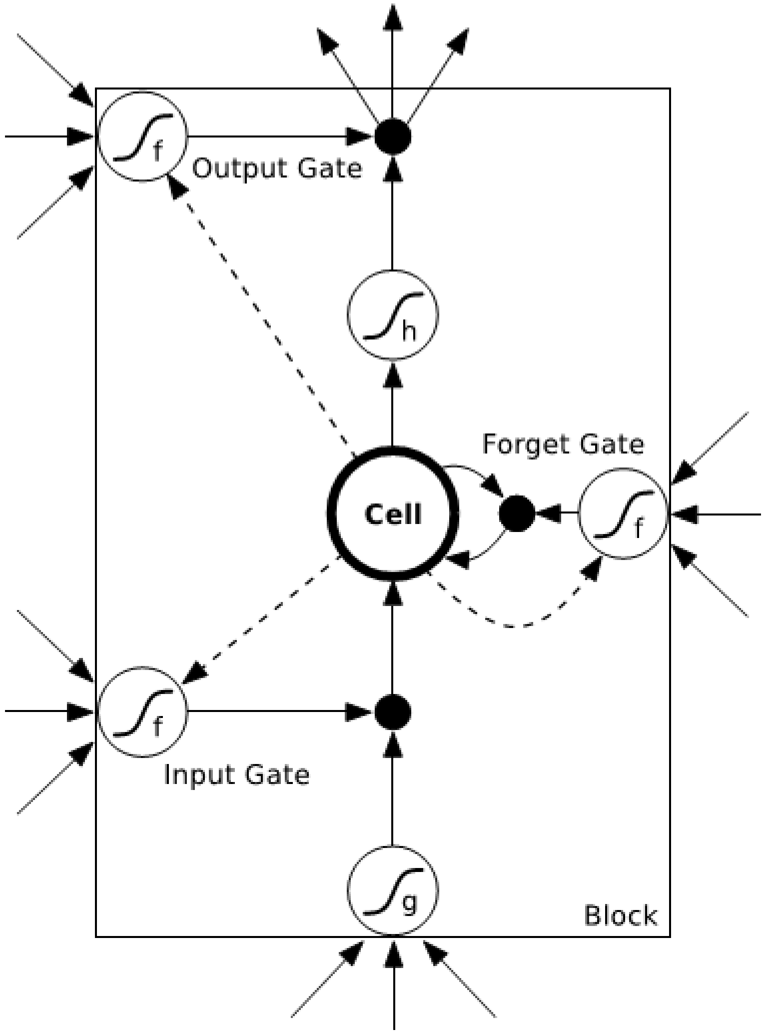


Figure 4.2: **LSTM memory block with one cell.** The three gates are nonlinear summation units that collect activations from inside and outside the block, and control the activation of the cell via multiplications (small black circles). The input and output gates multiply the input and output of the cell while the forget gate multiplies the cell's previous state. No activation function is applied within the cell. The gate activation function 'f' is usually the logistic sigmoid, so that the gate activations are between 0 (gate closed) and 1 (gate open). The cell input and output activation functions ('g' and 'h') are usually tanh or logistic sigmoid, though in some cases 'h' is the identity function. The weighted 'peephole' connections from the cell to the gates are shown with dashed lines. All other connections within the block are unweighted (or equivalently, have a fixed weight of 1.0). The only outputs from the block to the rest of the network emanate from the output gate multiplication.

Alex Graves, Supervised Sequence Labelling with Recurrent Neural Networks

S. Hochreiter and J. Schmidhuber. Long Short-Term Memory. Neural Computation,9(8):1735 1780, 1997

Long-Short Term Memory (LSTM)



The **multiplicative gates** allow LSTM memory cells to store and access information over long periods of time, thereby **mitigating the vanishing gradient problem**.

For example, as long as the input gate remains closed (i.e. has an activation near 0), the activation of the cell will not be overwritten by the new inputs arriving in the network, and can therefore be made available to the net much later in the sequence, by opening the output gate.

Alex Graves, Supervised Sequence Labelling with Recurrent Neural Networks

S. Hochreiter and J. Schmidhuber. Long Short-Term Memory. Neural Computation,9(8):1735 1780, 1997

Long-Short Term Memory (LSTM)

Preservation of Gradient Information pada LSTM

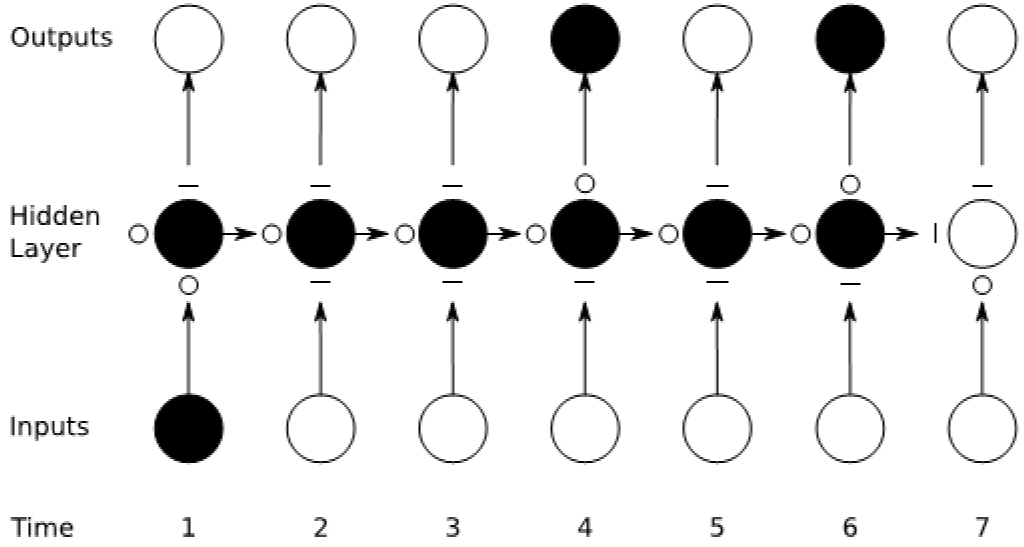
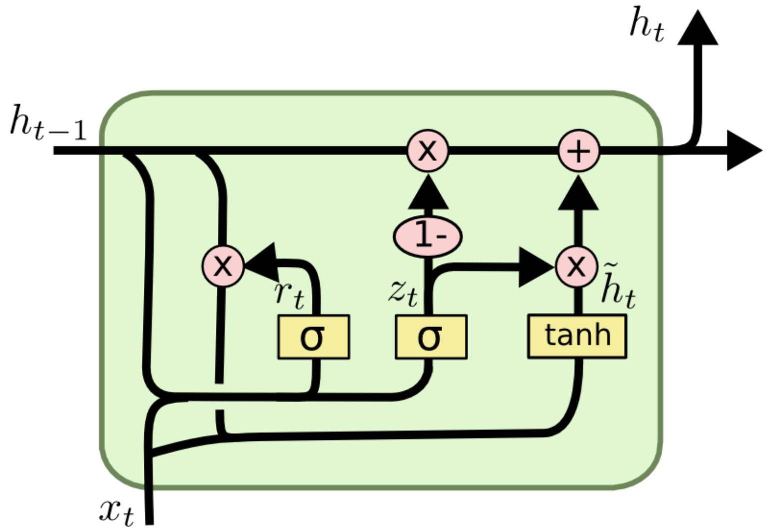


Figure 4.4: **Preservation of gradient information by LSTM.** As in Figure 4.1 the shading of the nodes indicates their sensitivity to the inputs at time one; in this case the black nodes are maximally sensitive and the white nodes are entirely insensitive. The state of the input, forget, and output gates are displayed below, to the left and above the hidden layer respectively. For simplicity, all gates are either entirely open ('O') or closed ('—'). The memory cell 'remembers' the first input as long as the forget gate is open and the input gate is closed. The sensitivity of the output layer can be switched on and off by the output gate without affecting the cell.

Alex Graves, Supervised Sequence Labelling with Recurrent Neural Networks

S. Hochreiter and J. Schmidhuber. Long Short-Term Memory. Neural Computation,9(8):1735 1780, 1997

Another Variant: Gated Recurrent Units (GRUs)



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

500 Point **for those who implement GRUs from the scratch, like we did for LSTMs**