Universität Augsburg University

Deep Learning

CNNs & RNNs

Tuesday 19th November

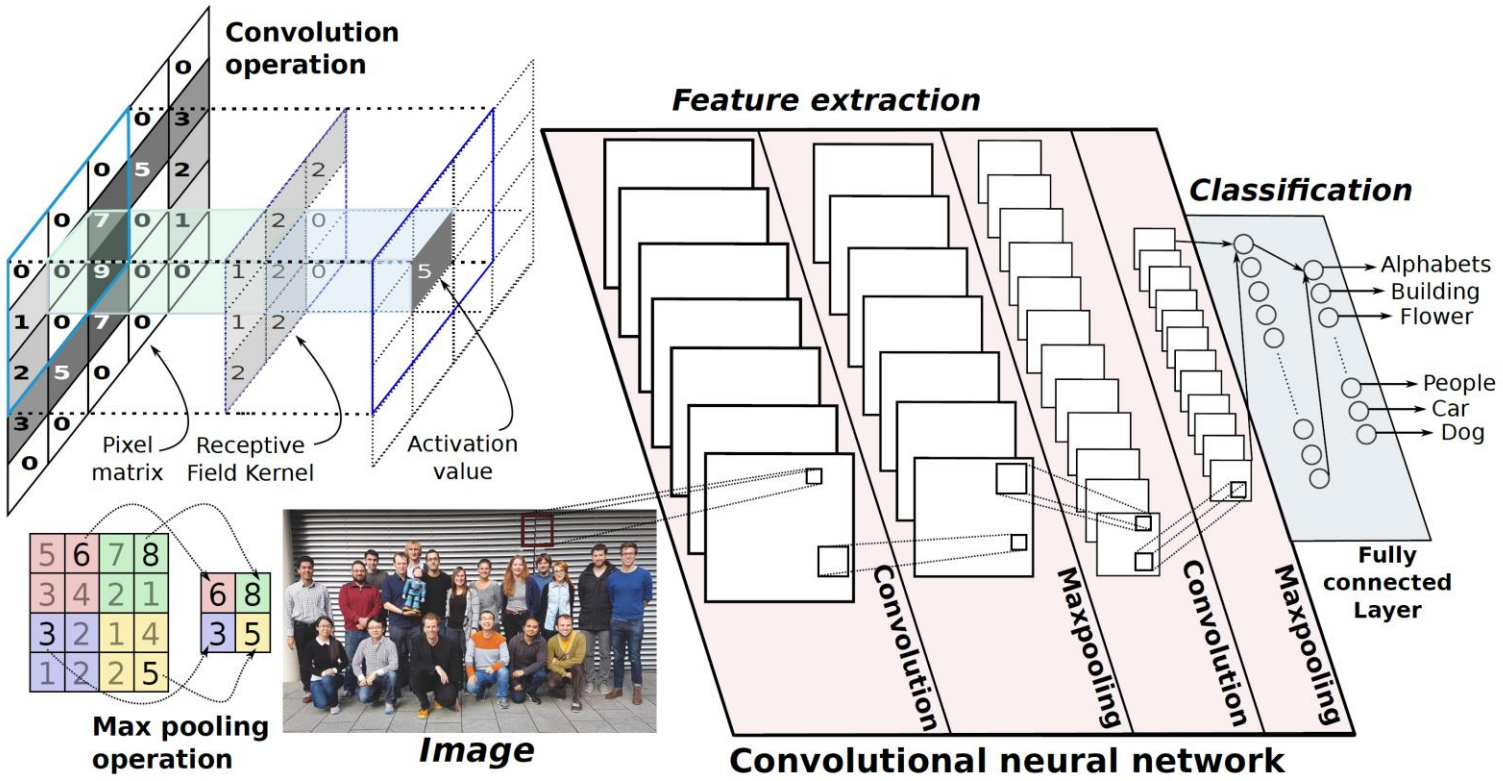
Dr. Nicholas Cummins

Convolutional Neural Network (CNNs)

Improve generalisation by reusing the same weights to detect the same feature in multiple places

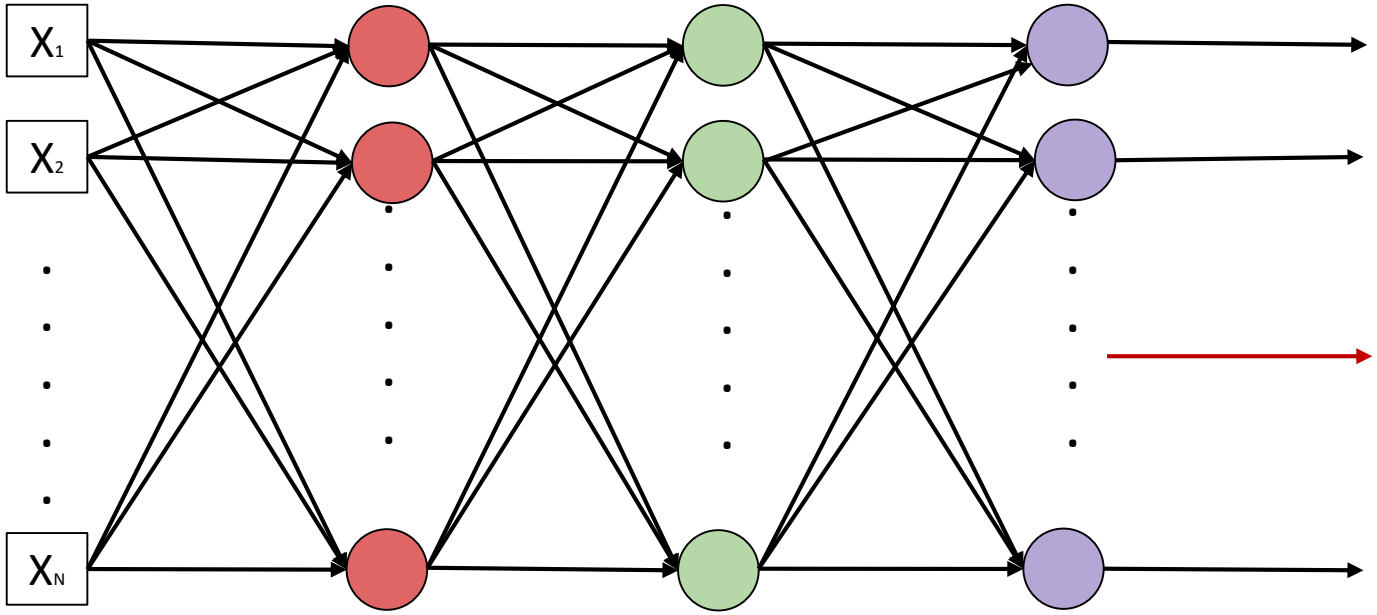
- **What is a Convulotional Neural Network?**
  - A CNN is a neural network with convolution operations instead of matrix multiplications in at least one of the layers
  - Special form of feed-forward network
    - Reuse the same neurons for repetitive convolution tasks
      - Convolution ≈ Filtering
    - Convolutional kernels recognise patterns in a signal
    - Reuses weights to detect the same patterns in multiple places
    - Reduces overfitting and leads to much more accurate models

- **Convolutional Neural Network (CNN)**
  - Convolutional kernels perform feature extraction



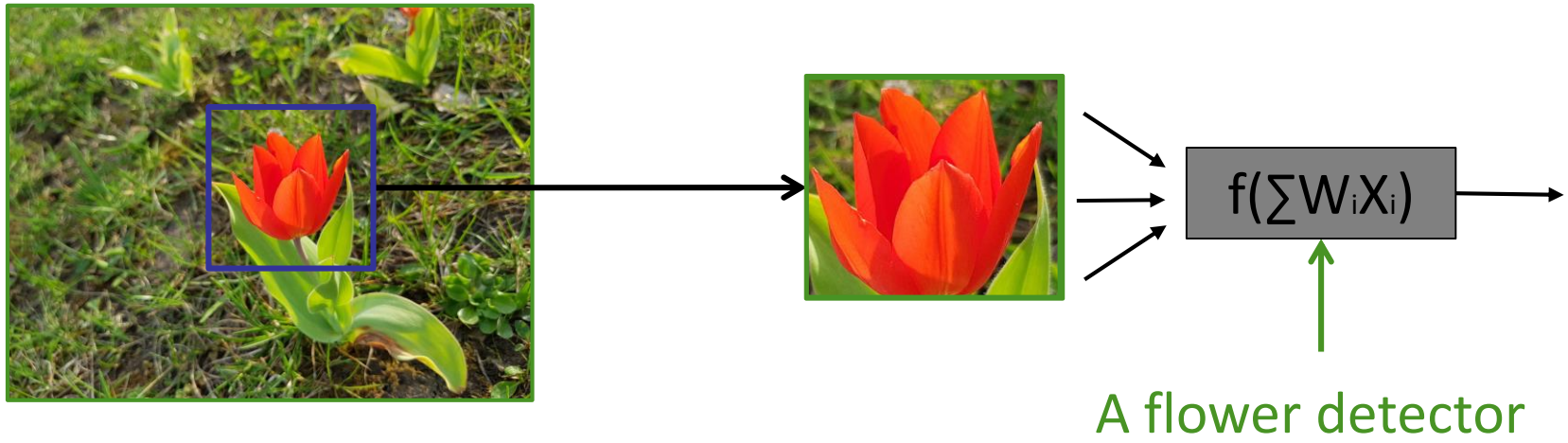
- **When applying a fully connected feedforward neural network**
  - Many thousands of weights and connections
  - **Feedforward networks can be simplified by considering properties of input signal**

An RGB image can be represented as pixels (32*32*3)

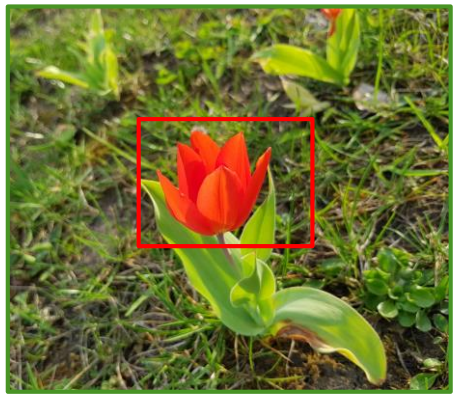


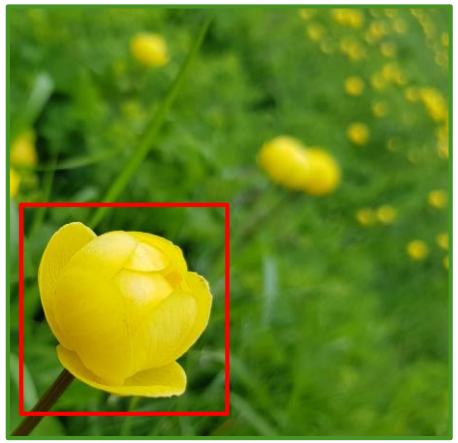
Input vector dimension
N= 32×32×3= 3072

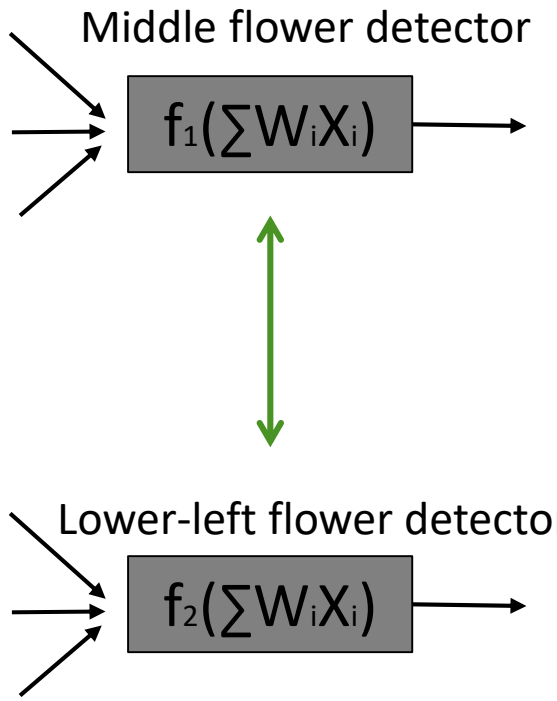
- **Key patterns can be much smaller than the whole signal**
  - Therefore a network does not have to see the whole signal to discover the pattern
  - Only need to be connected to small region (X_i)
  - Fewer parameters (W_i) are required



- **Similar patterns appear in different regions**

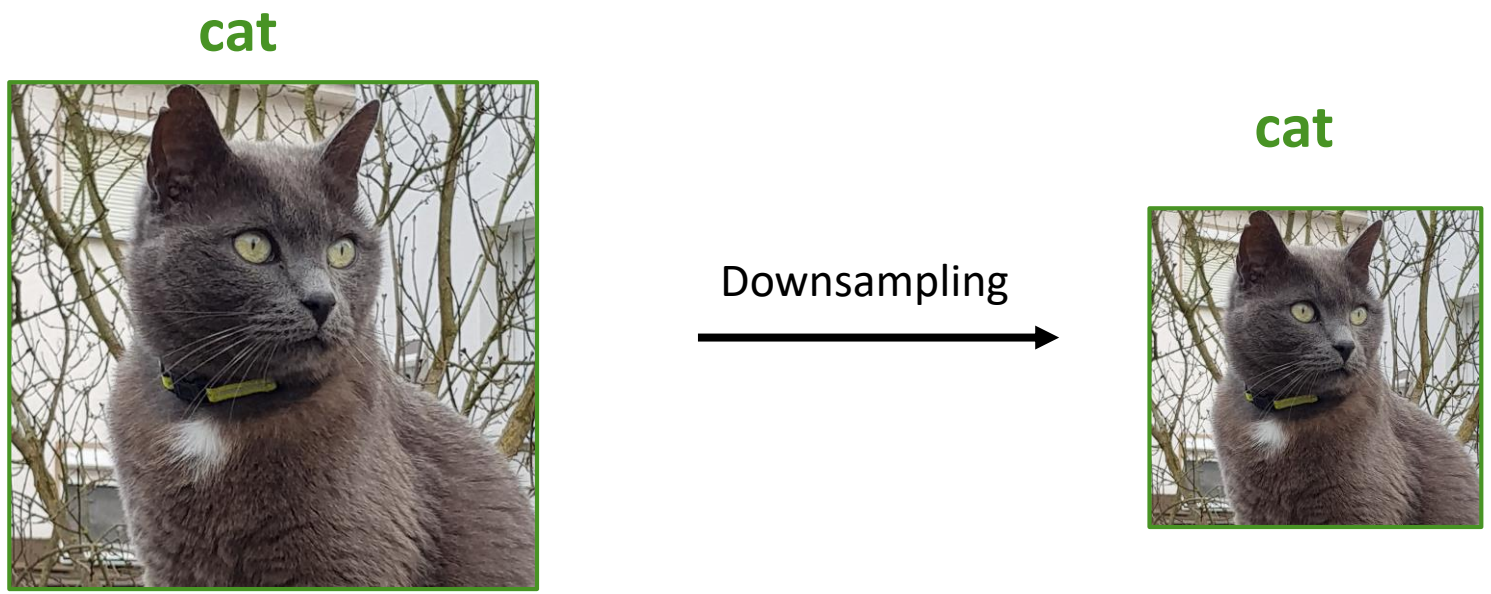






- Both networks are flower detectors
- They detect almost the same thing
- Therefore, they can use the same set of parameters

- **Downsampling does not change the key properties**
  - E.g., images are made smaller by downsampling
  - Less parameters for the network to process



CNN – Properties of Input Signal

Can repeat many times

Convolution

Max Pooling

Convolution

Max Pooling

Property 1: Some patterns are much smaller than the whole image

Property 2 : The same patterns appear in different regions

Property 3 : Downsampling the pixels does not change the object

Flatten

Fully Connected Feedforward Network

Prediction

Convolution

- Convolution is the manipulation of two *signals* to form a third
- *Notation:* Convolution is denoted by the star $*$ operator

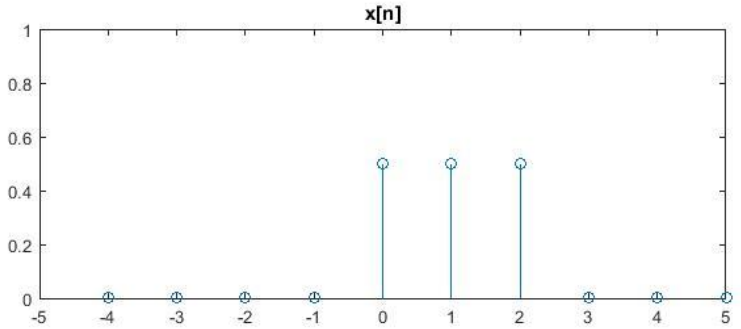
$$y[n] = x[n] * h[n] = \sum_{i=-\infty}^{\infty} x[i]h[n-i]$$

$$y[n] = x[m,n] * h[m,n] = \sum_{i=-\infty}^{\infty} \sum_{j=-\infty}^{\infty} x[i,j]h[m-i,n-j]$$

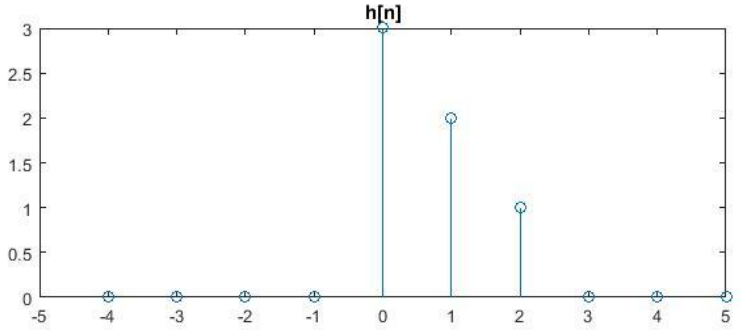
Convolution Worked Example:

- Calculate and plot $y[n] = x[n] * h[n]$ when:

$$x[n] = \begin{cases} 0.5 \; for \; n = 0,1,2 \\ 0 \; otherwise \end{cases}$$



$$h[n] = \begin{cases} 3,2,1 \; for \; n = 0,1,2 \\ 0 \; otherwise \end{cases}$$



Convolution Worked Example:

- Convolution can be expressed as an inner product

$$y[n] = x[n] * h[n]$$
$$y[n] = \sum_{k=-\infty}^{\infty} x[k]\, h[n-k]$$
$$y[n] = \sum_{k=-\infty}^{\infty} x[k]\, \tilde{h}\,[k-n]$$
$$y[n] = \langle x, \bar{h}_n \rangle$$

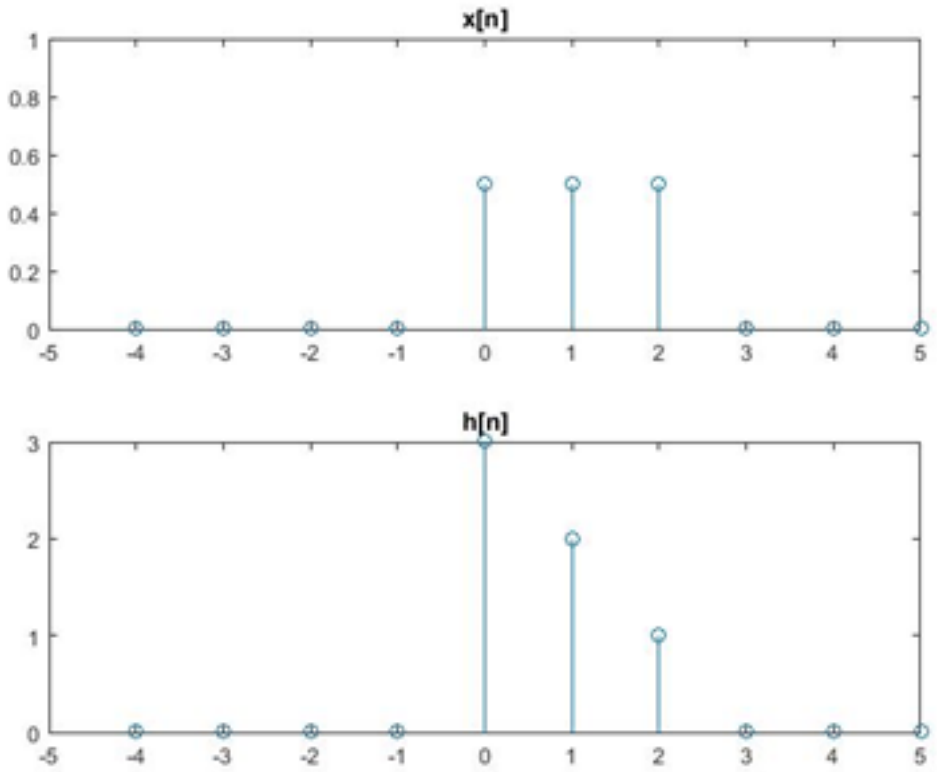
- $\bar{\boldsymbol{h}} \equiv h[-i]$ is a time reversed version $\boldsymbol{h} \equiv h[i]$ (**flip**)
- n^{th} output obtained by *delaying* $\bar{\boldsymbol{h}}$ n times (**shift**) and taking *inner product* (**multiple**)

Convolution Worked Example:

- $x = \{0.5,0.5,0.5\}, h = \{3,2,1\}$

Flip, shift, multiple:

- $y[0] = \langle x, \overline{\boldsymbol{h}}_0 \rangle$ where $\overline{\boldsymbol{h}}_0 = \{3,0,0\}$
- $y[0] = 1.5$
- $y[1] = \langle x, \overline{\boldsymbol{h}}_1 \rangle$ where $\overline{\boldsymbol{h}}_1 = \{2,3,0\}$
- $y[1] = 2.5$
- $y[2] = \langle x, \overline{\boldsymbol{h}}_2 \rangle$ where $\overline{\boldsymbol{h}}_1 = \{1,2,3\}$
- $y[2] = 3.0$

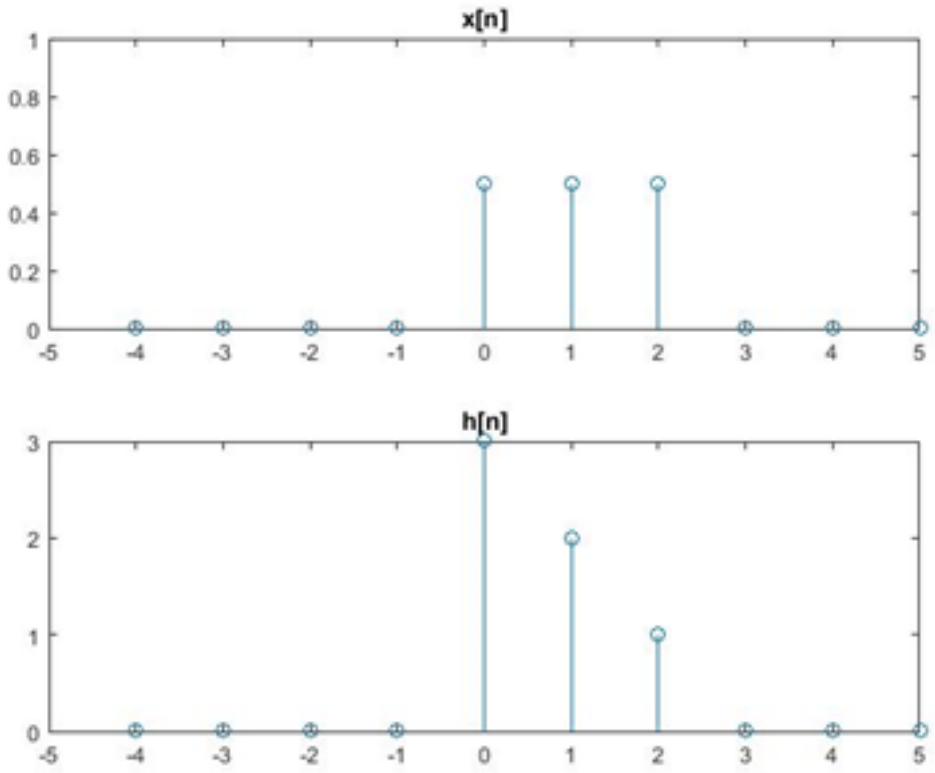


Convolution Worked Example:

- $x = \{0.5,0.5,0.5\}, h = \{3,2,1\}$

Flip, shift, multiple:

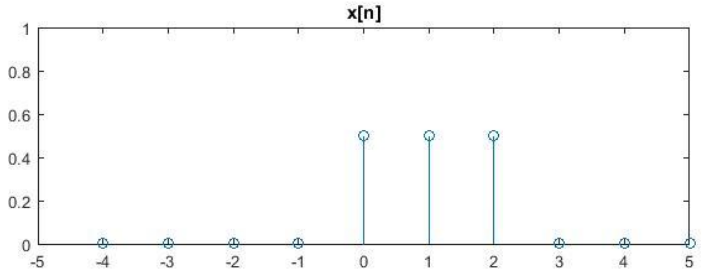
- $y[3] = \langle \boldsymbol{x}, \overline{\boldsymbol{h}}_3 \rangle$ where $\overline{\boldsymbol{h}}_0 = \{0,1,2\}$
- $y[3] = 1.5$
- $y[4] = \langle \boldsymbol{x}, \overline{\boldsymbol{h}}_1 \rangle$ where $\overline{\boldsymbol{h}}_1 = \{0,0,1\}$
- $y[4] = 0.5$



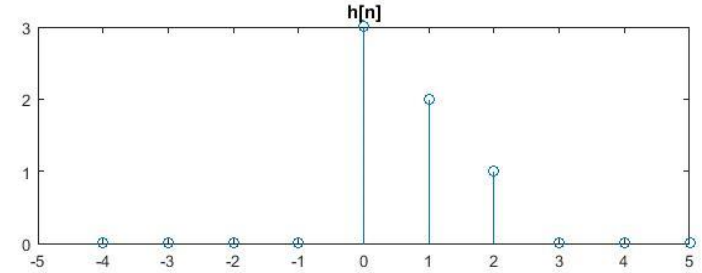
Therefore $\boldsymbol{y} = \boldsymbol{x} * \boldsymbol{h} = \{1.5,2.5,3,1.5,0.5\}$

Convolution Worked Example:

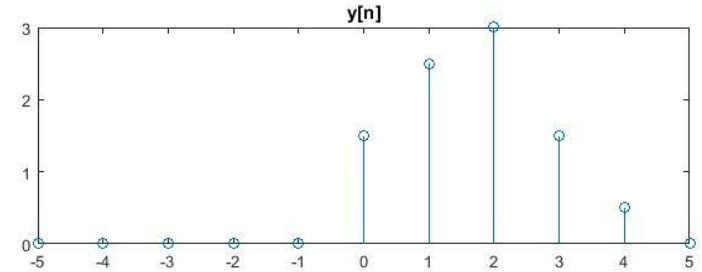
$$x[n] = \begin{cases} 0.5 \; for \; n = 0,1,2 \\ 0 \; otherwise \end{cases}$$



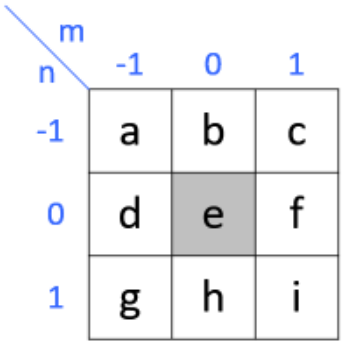
$$h[n] = \begin{cases} 3,2,1 \; for \; n = 0,1,2 \\ 0 \; otherwise \end{cases}$$



$$y[n] = x[n] * h[n]$$

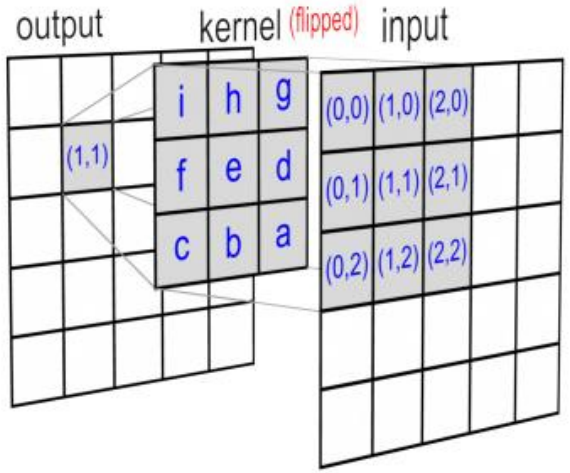


- **Convolution in two dimensions**



Example 2D Kernel

- The origin is located at the middle of kernel



2D Convolution

- The kernel is flipped in both the horizontal and vertical directions

- **Convolution in two dimensions**

1	2	3
4	5	6
7	8	9

$*$

-1	-2	-1
0	0	0
1	2	1

$=$

-13	-20	-17
-18	-24	-18
13	20	17

$$\boldsymbol{y}[\mathbf{1},\mathbf{1}] = \sum_{\boldsymbol{j}} \sum_{\boldsymbol{i}} \boldsymbol{x}[\boldsymbol{i},\boldsymbol{j}]\boldsymbol{h}[\mathbf{1}-\boldsymbol{i},\mathbf{1}-\boldsymbol{j}]$$

$$= x[0,0]h[1,1] + x[1,0]h[0,1] + x[2,0]h[-1,1]$$
$$+ x[0,1]h[1,0] + x[1,1]h[0,0] + x[2,1]h[-1,0]$$
$$+ x[0,2]h[1,-1] + x[1,0]h[0,-1] + x[2,2]h[-1,-1]$$
$$= 1 \cdot 1 + 2 \cdot 2 + 3 \cdot 1$$
$$= 4 \cdot 0 + 5 \cdot 0 + 6 \cdot 0$$
$$= 7 \cdot -1 + 8 \cdot -2 + 9 \cdot -1$$
$$= -24$$

- **Convolution and Correlation**

Input

x_{11}	x_{12}	x_{13}
x_{21}	x_{22}	x_{23}
x_{31}	x_{32}	x_{33}

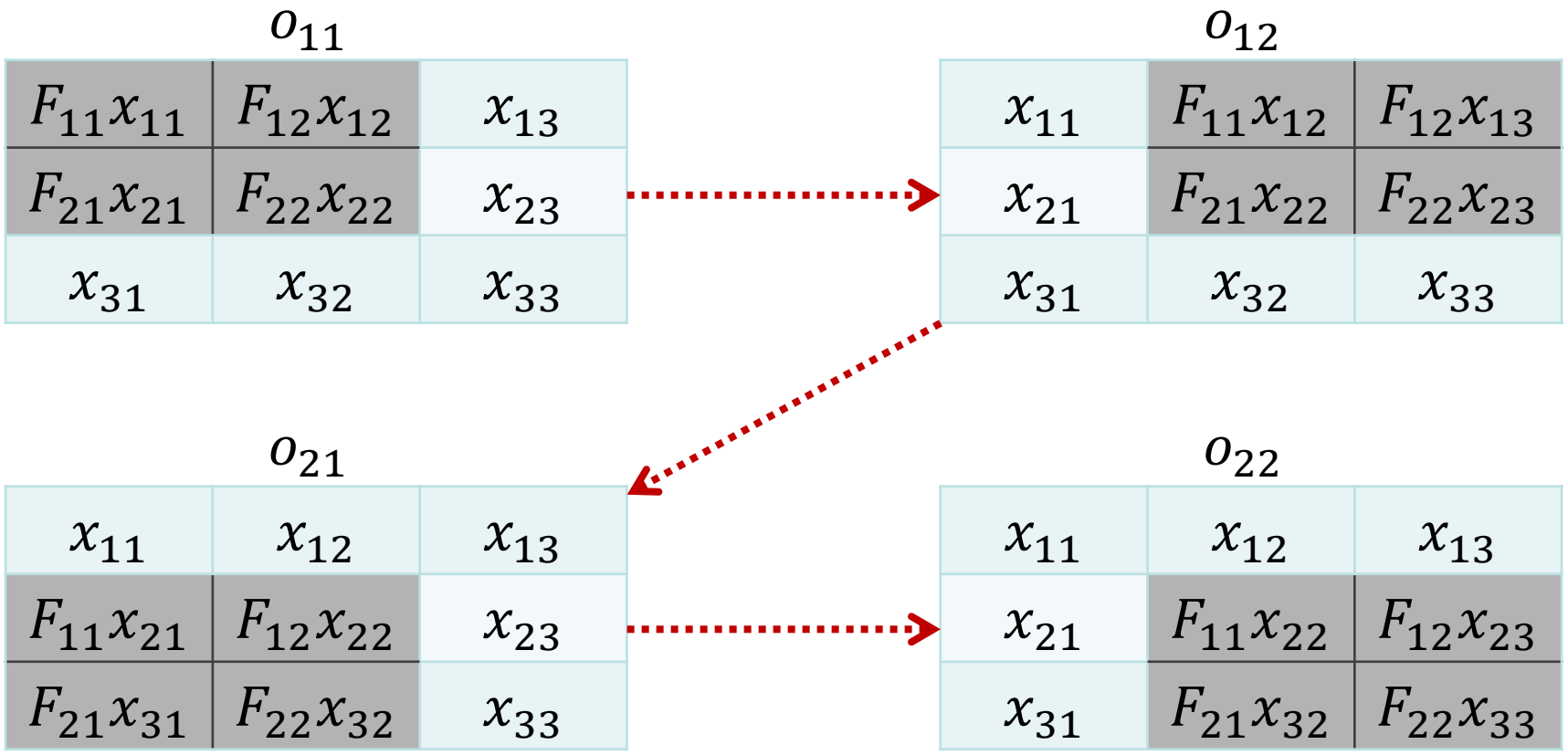
Filter

F_{11}	F_{12}
F_{21}	F_{22}

Correlation

$$o_{11} = F_{11}x_{11} + F_{12}x_{12} + F_{21}x_{21} + F_{22}x_{22}$$
$$o_{12} = F_{11}x_{12} + F_{12}x_{13} + F_{21}x_{22} + F_{22}x_{23}$$
$$o_{21} = F_{11}x_{21} + F_{12}x_{22} + F_{21}x_{31} + F_{22}x_{32}$$
$$o_{22} = F_{11}x_{22} + F_{12}x_{23} + F_{21}x_{32} + F_{22}x_{33}$$

- **Convolution and Correlation**



- **Convolution and Correlation**

Input

x_{11}	x_{12}	x_{13}
x_{21}	x_{22}	x_{23}
x_{31}	x_{32}	x_{33}

Filter

F_{11}	F_{12}
F_{21}	F_{22}

Convolution

$$o_{11} = F_{22}x_{11} + F_{21}x_{12} + F_{12}x_{21} + F_{11}x_{22}$$
$$o_{12} = F_{22}x_{12} + F_{21}x_{13} + F_{12}x_{22} + F_{11}x_{23}$$
$$o_{21} = F_{22}x_{21} + F_{21}x_{22} + F_{12}x_{31} + F_{11}x_{32}$$
$$o_{22} = F_{22}x_{22} + F_{21}x_{23} + F_{12}x_{32} + F_{11}x_{33}$$

- **Convolution and Correlation**
  - Convolution is correlation with the filter rotated 180 degrees

Correlation

Input

x_{11}	x_{12}	x_{13}
x_{21}	x_{22}	x_{23}
x_{31}	x_{32}	x_{33}

,

Filter

F_{11}	F_{12}
F_{21}	F_{22}

Convolution

Input

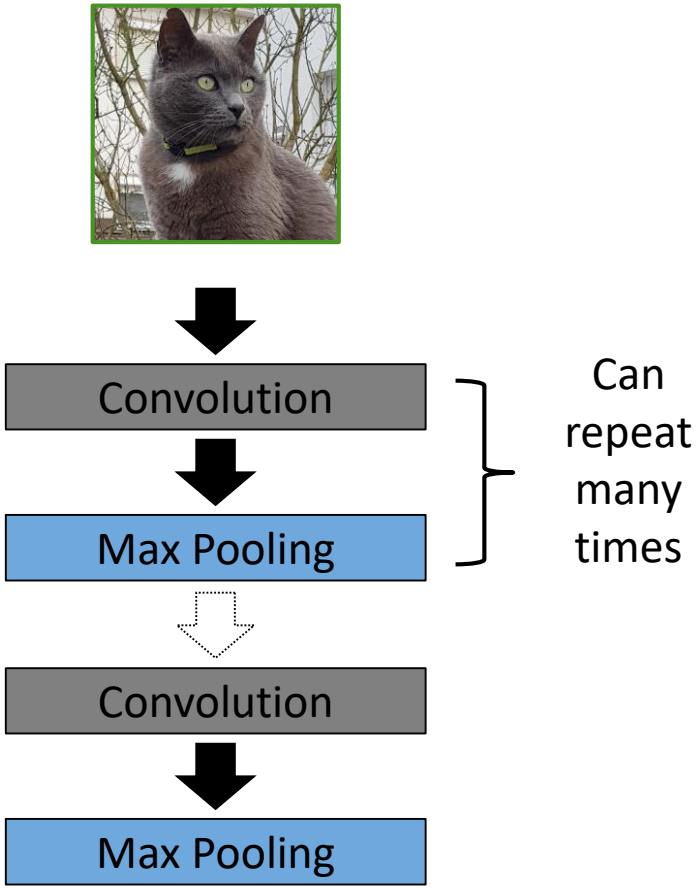
x_{11}	x_{12}	x_{13}
x_{21}	x_{22}	x_{23}
x_{31}	x_{32}	x_{33}

,

Filter

F_{22}	F_{21}
F_{12}	F_{11}

- **Convolutional Layers**
  - **Property 1:** Some patterns are much smaller than the whole image
  - **Property 2 :** The same patterns appear in different regions
- **Pooling Layers**
  - **Property 3** : Downsampling the pixels does not change the object



Convolutional Layers

0	**1**	0	0	**1**	0
0	**1**	0	0	**1**	0
0	**1**	0	0	**1**	0
1	0	0	0	0	**1**
0	**1**	0	0	**1**	0
0	0	**1**	**1**	0	0

6 x 6 image

Apply small filers to detect small patterns

Each filter has a size of **3 x 3**

-1	**1**	-1
-1	**1**	-1
-1	**1**	-1

Filter 1

1	-1	-1
-1	**1**	-1
-1	-1	**1**

Filter 2

. . . . .

Note: Only the size of the filters is specified; the weights are initialised to arbitrary values before the start of training.

The weights of the filters are learned through the CNN training process

- **Key Parameters**
  - **Filter size** – defines the height and width of the filter kernel
    - E.g., a filter kernel of size three would have nine weights
  - **Stride** – determines the number of steps to move in each spatial direction while performing convolution.
  - **Padding** –appends zeroes to the boundary of an image to control the size of the output of convolution
    - When we convolve an image of a specific size by a filter, the resulting image is generally smaller than the original image

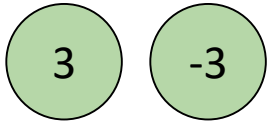
stride = 1

0	**1**	0	0	**1**	0
0	**1**	0	0	**1**	0
0	**1**	0	0	**1**	0
1	0	0	0	0	**1**
0	**1**	0	0	**1**	0
0	0	**1**	**1**	0	0

6 x 6 image

-1	**1**	-1
-1	**1**	-1
-1	**1**	-1

Filter 1



Compute the dot product between the filter and a small 3 x 3 chunk of the image

Note: We are technically performing correlation not convolution

Convolutional Layers

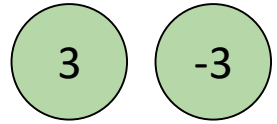
stride = 2

0	**1**	0	0	**1**	0
0	**1**	0	0	**1**	0
0	**1**	0	0	**1**	0
1	0	0	0	0	**1**
0	**1**	0	0	**1**	0
0	0	**1**	**1**	0	0

6 x 6 image

-1	**1**	-1
-1	**1**	-1
-1	**1**	-1

Filter 1



We set stride = 1 below

Compute the dot product between the filter and a small 3 x 3 chunk of the image

Convolutional Layers

stride = 1

0	**1**	0	0	**1**	0
0	**1**	0	0	**1**	0
0	**1**	0	0	**1**	0
1	0	0	0	0	**1**
0	**1**	0	0	**1**	0
0	0	**1**	**1**	0	0

6 x 6 image

-1	**1**	-1
-1	**1**	-1
-1	**1**	-1

Filter 1

3	-3	-3	3
1	-2	-2	1
1	-2	-2	1
-1	-1	-1	-1

4 x 4 image

Convolutional Layers

stride = 1, filter size = 3

0	**1**	0	0	**1**	0
0	**1**	0	0	**1**	0
0	**1**	0	0	**1**	0
1	0	0	0	0	**1**
0	**1**	0	0	**1**	0
0	0	**1**	**1**	0	0

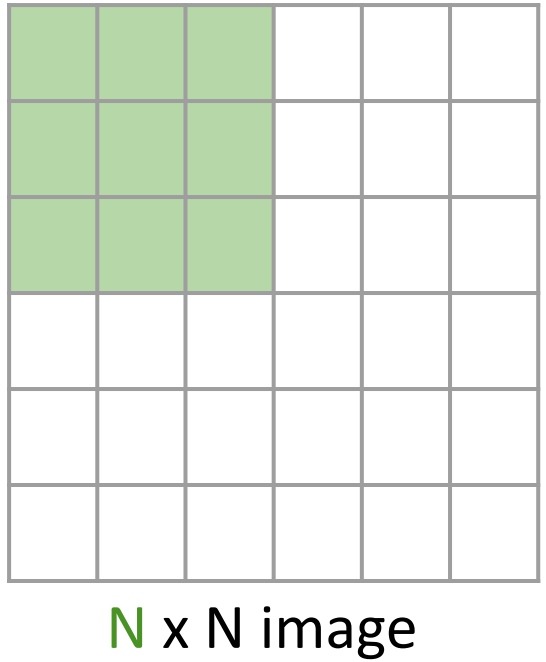
6 x 6 image

3	-3	-3	3
1	-2	-2	1
1	-2	-2	1
-1	-1	-1	-1

4 x 4 image

output size: (6 - 3) / 1 + 1 = 4

Convolutional Layers



output size: (N - F) / stride + 1

for example: N = 6, F = 3

stride = 1 -> (6-3)/1 + 1 = 4

stride = 2 -> (6-3)/2 + 1 = 2.5 :\

stride = 3 -> (6-3)/3 + 1 = 2

zero-padding to the border

0	0	0	0	0	0	0	0
0							0
0							0
0							0
0							0
0							0
0							0
0	0	0	0	0	0	0	0

N x N image

For example: N = 6, F = 3, stride = 1

Without 0-padding:

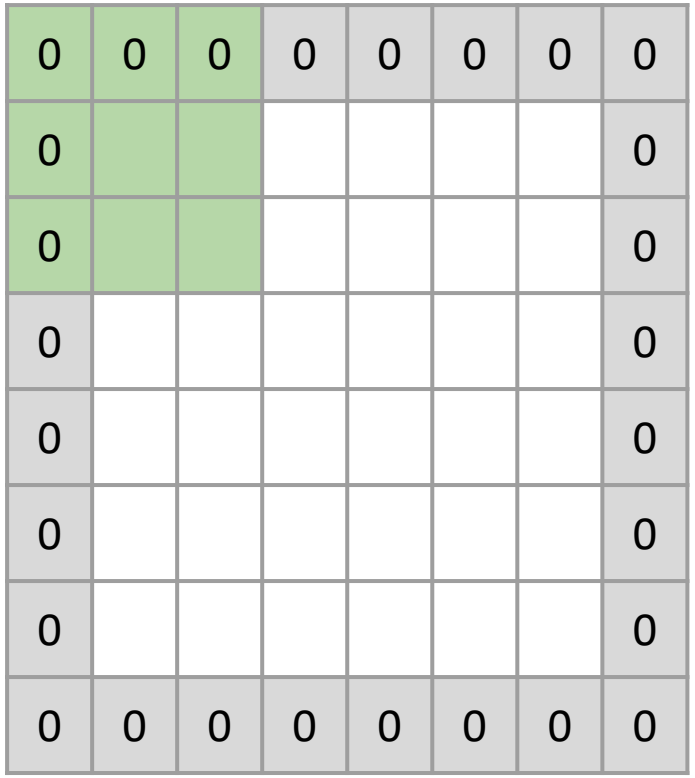
output size is (6-3)/1 + 1 = 4

With 0-padding with 1 pixel border:

output size is (6-3+2x1)/1 + 1 = 6

The output size is then the same as the input!

zero-padding to the border



N x N image

In general, stride=1, filters of size FxF,
then zero-padding with **(F-1)/2**,
to preserve size spatially

e.g. F = 3 -> zero pad with 1 pixel to the border
F = 5 -> zero pad with 2 pixels to the border
F = 7 -> zero pad with 3 pixels to the border

Convolutional Layers

stride = 1

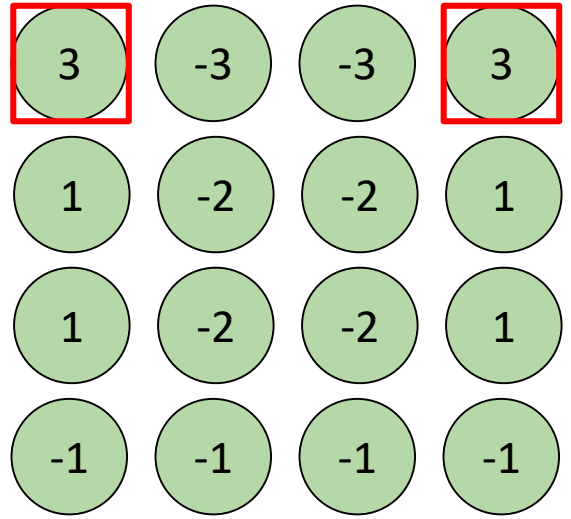
0	**1**	0	0	**1**	0
0	**1**	0	0	**1**	0
0	**1**	0	0	**1**	0
1	0	0	0	0	**1**
0	**1**	0	0	**1**	0
0	0	**1**	**1**	0	0

6 x 6 image

-1	**1**	-1
-1	**1**	-1
-1	**1**	-1

Filter 1

detect a vertical line



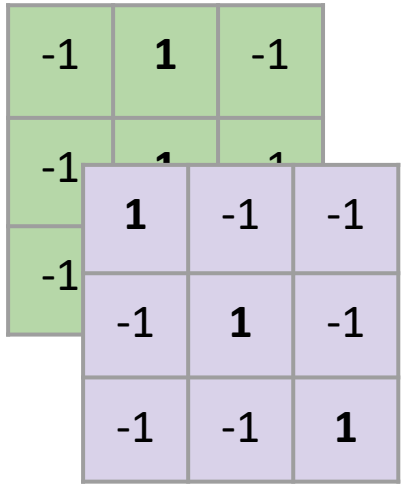
The same pattern in different locations are detected with the same filter

Convolutional Layers

stride = 1

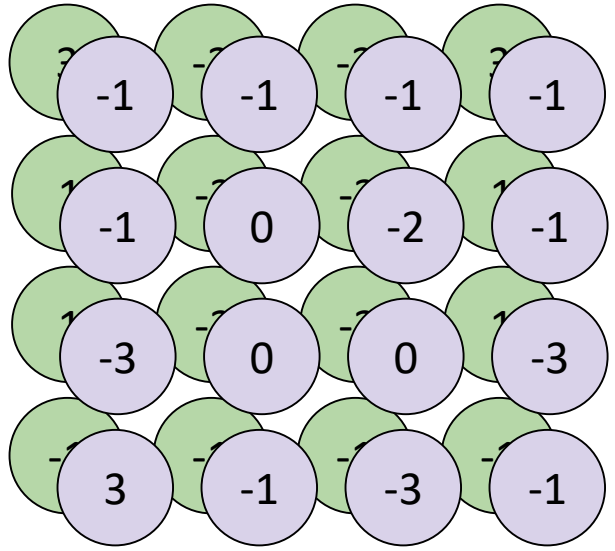
0	**1**	0	0	**1**	0
0	**1**	0	0	**1**	0
0	**1**	0	0	**1**	0
1	0	0	0	0	**1**
0	**1**	0	0	**1**	0
0	0	**1**	**1**	0	0

6 x 6 image



Filter 2

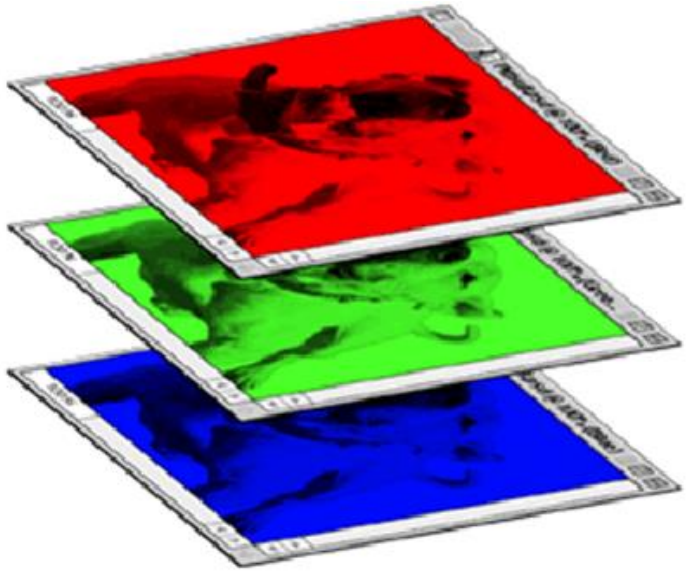
Feature Maps



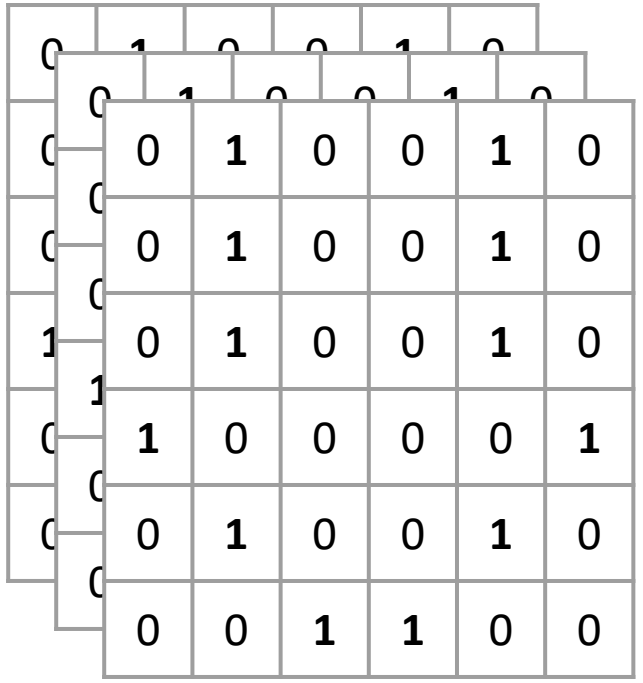
4 x 4 x (#filters)

Do the same process for every filter

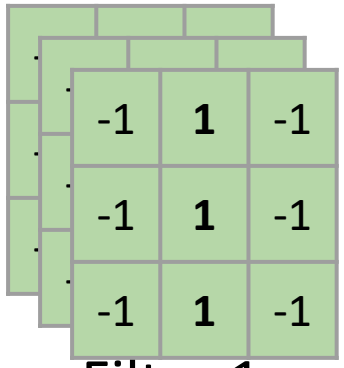
RGB images



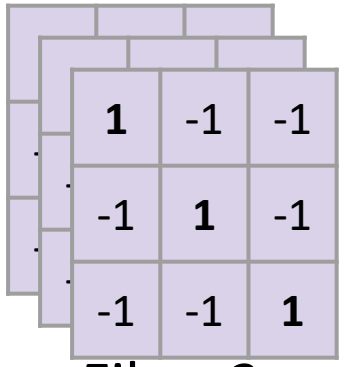
3 channels



6 x 6 x 3



Filter 1



Filter 2

Filters always extend the full depth of the input volume

3 x 3 x 3

- **Forward pass:**
  - Input X of size 3×3 using a single filter F of size 2×2, no padding and $stride = 1$, generating an output O of size 2×2

X_{11}	X_{12}	X_{13}
X_{21}	X_{22}	X_{23}
X_{31}	X_{32}	X_{33}

F_{11}	F_{12}
F_{21}	F_{22}

O_{11}	O_{12}
O_{21}	O_{22}

$$O_{11} = F_{11}x_{11} + F_{12}x_{12} + F_{21}x_{21} + F_{22}x_{22}$$

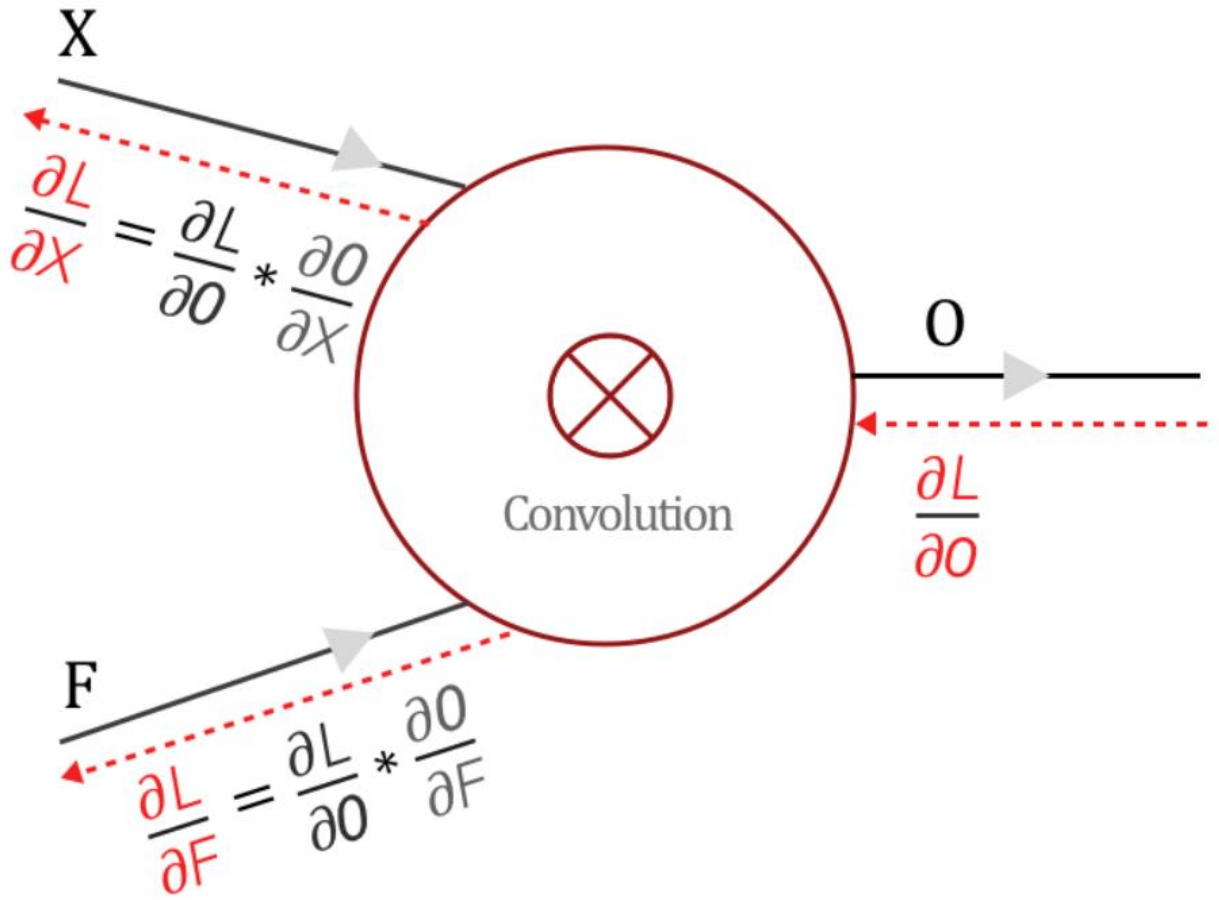
$$O_{12} = F_{11}x_{12} + F_{12}x_{13} + F_{21}x_{22} + F_{22}x_{22}$$

$$O_{21} = F_{11}x_{21} + F_{12}x_{22} + F_{21}x_{31} + F_{22}x_{32}$$

$$O_{22} = F_{11}x_{22} + F_{12}x_{23} + F_{21}x_{32} + F_{22}x_{33}$$

Image source: https://medium.com

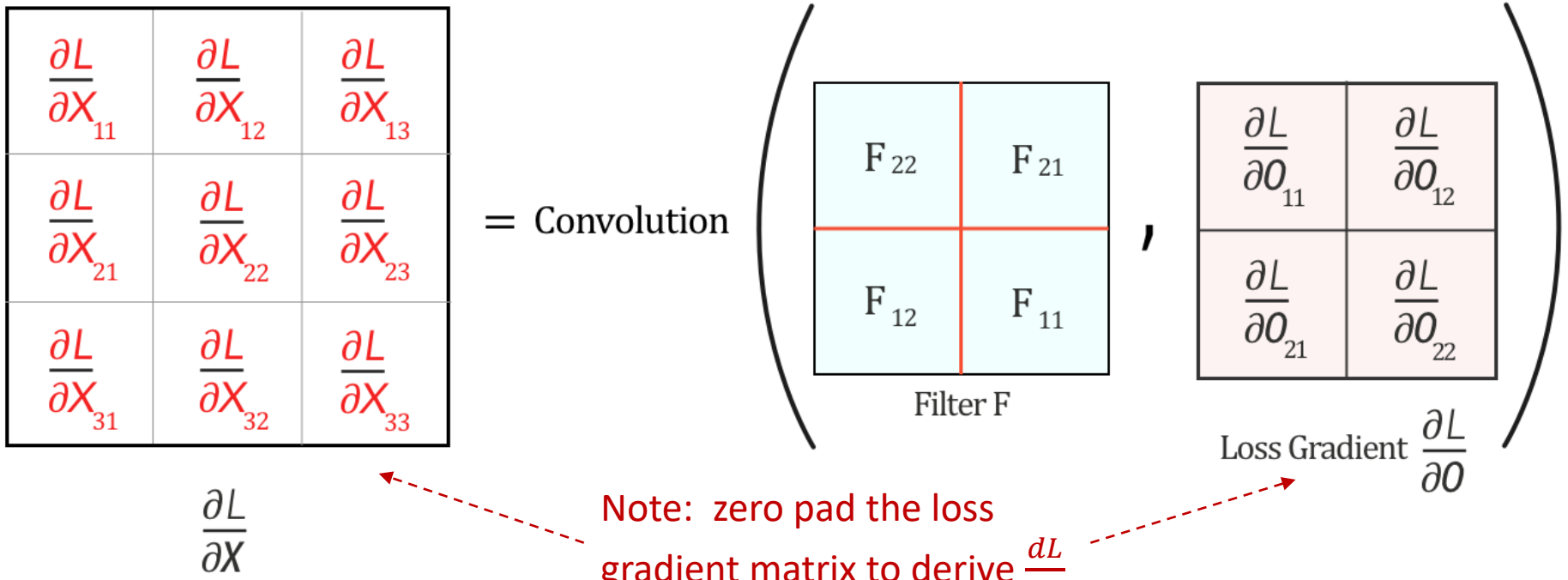
- **Backward pass:**
  - $\frac{\partial L}{\partial O}$ the loss gradient from the next layer
  - $\frac{\partial L}{\partial X}$ the loss gradient to be passed back to the proceeding layer
  - $\frac{\partial L}{\partial F}$ the loss gradient to update the filter



Convolutional Layers – Backpropagation

Image source: https://medium.com

- $\frac{\partial L}{\partial X}$ the loss gradient to be passed back
  - Convolution between a F rotated by 180° and the loss gradient $\frac{\partial L}{\partial O}$
    - I.e., "Proper" Convolution



Note: zero pad the loss gradient matrix to derive $\frac{dL}{dX}$

Convolutional Layers – Backpropagation

Image source: https://medium.com

- $\partial L/\partial F$ the loss gradient to update the filter
  - Convolution of input matrix X and loss gradient $\frac{\partial L}{\partial O}$

$$\begin{bmatrix} \frac{\partial L}{\partial F_{11}} & \frac{\partial L}{\partial F_{12}} \\ \frac{\partial L}{\partial F_{21}} & \frac{\partial L}{\partial F_{22}} \end{bmatrix} = \text{Convolution}\left(\begin{bmatrix} X_{11} & X_{12} & X_{13} \\ X_{21} & X_{22} & X_{23} \\ X_{31} & X_{32} & X_{33} \end{bmatrix}, \begin{bmatrix} \frac{\partial L}{\partial O_{11}} & \frac{\partial L}{\partial O_{12}} \\ \frac{\partial L}{\partial O_{21}} & \frac{\partial L}{\partial O_{22}} \end{bmatrix} \right)$$

Key Parameters:

- Accepts an input of size W_1 x H_1 X D_1
- Requires 4 hyperparameters:
  - Number of filters **K**
  - Size of the filters **F**
  - The stride **S**
  - The amount of zero padding **P**
- Produce an output of size W_2 x H_2 x D_2, where
  - $W_2 = (W_1 - F + 2P)/S + 1$
  - $H_2 = (H_1 - F + 2P)/S + 1$
  - $D_2 = K$
- With parameter sharing, it introduces **F** x **F** x D_1 weights per filter, for a total of (**F** x **F** x D_1)x **K** weights and **K** biases.

Common settings:

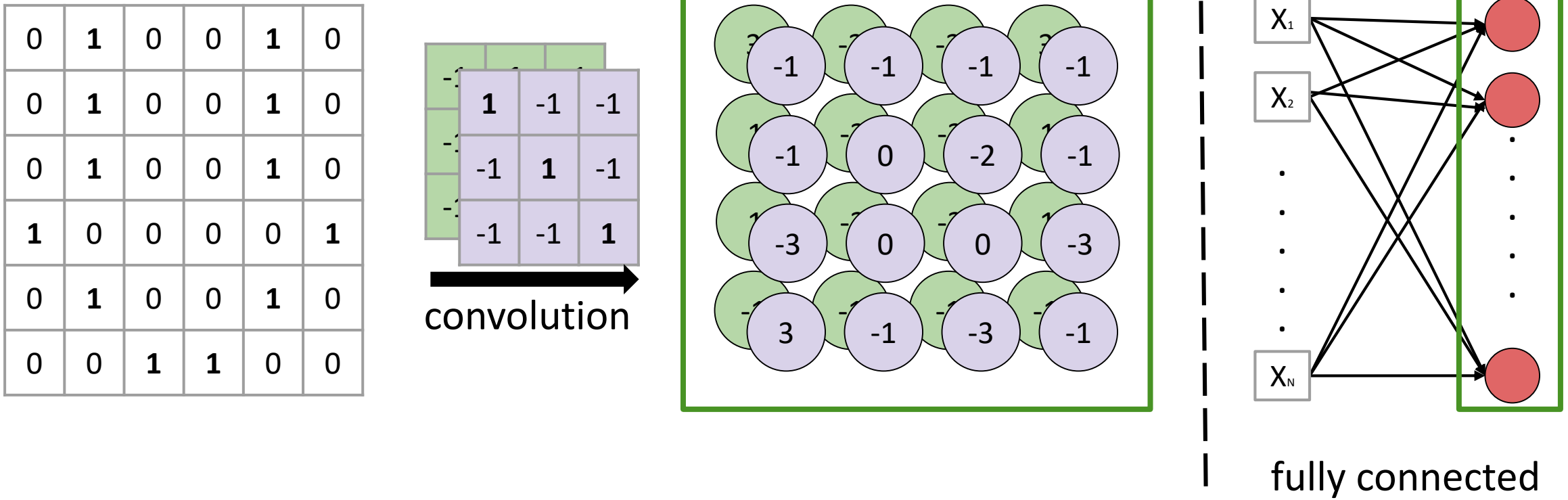
K: powers of 2, such as 32, 64, 128, 512

F = 3, S=1, P=2

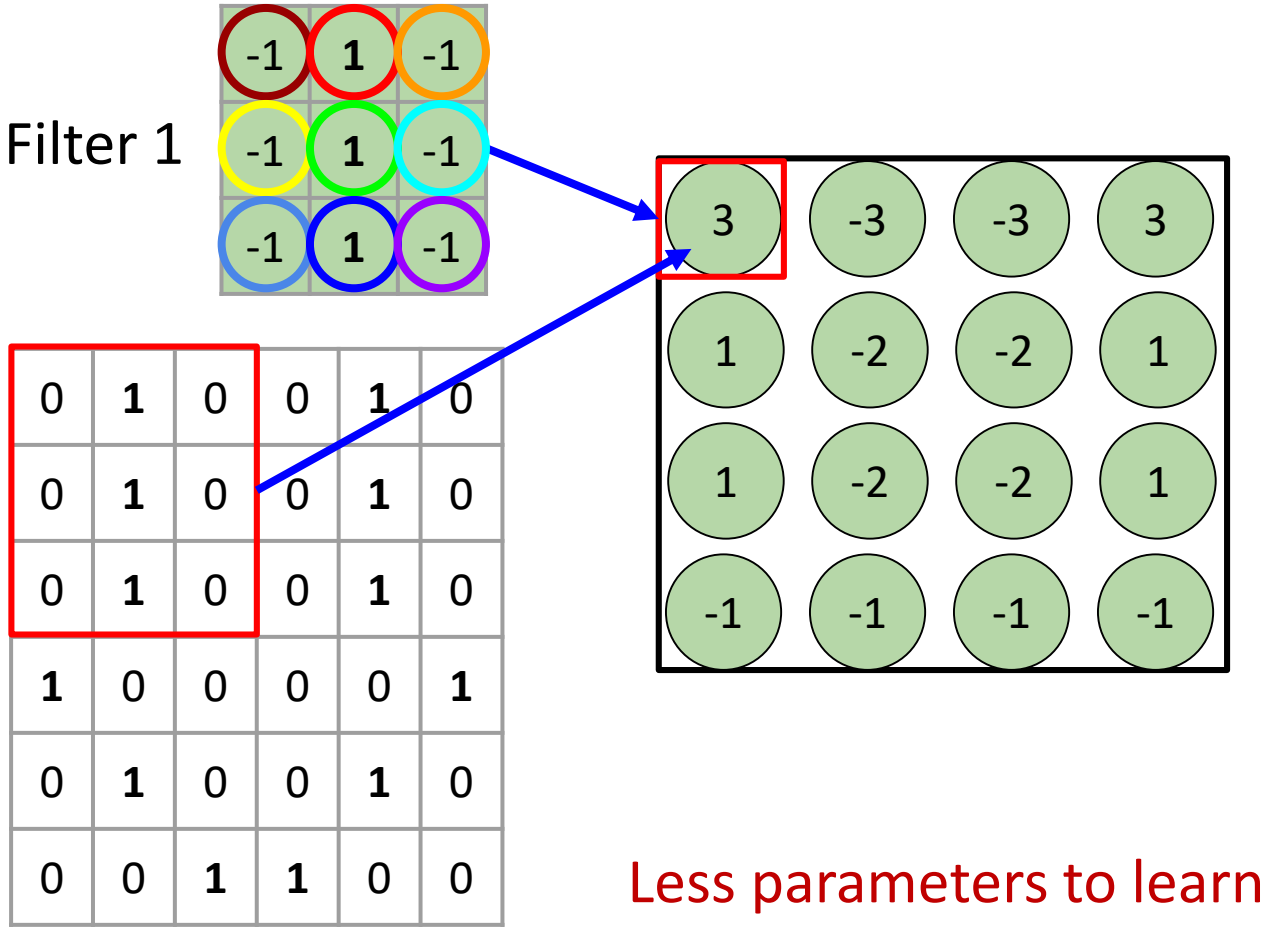
F = 5, S=1, P=2

… …

Convolution vs Fully Connected

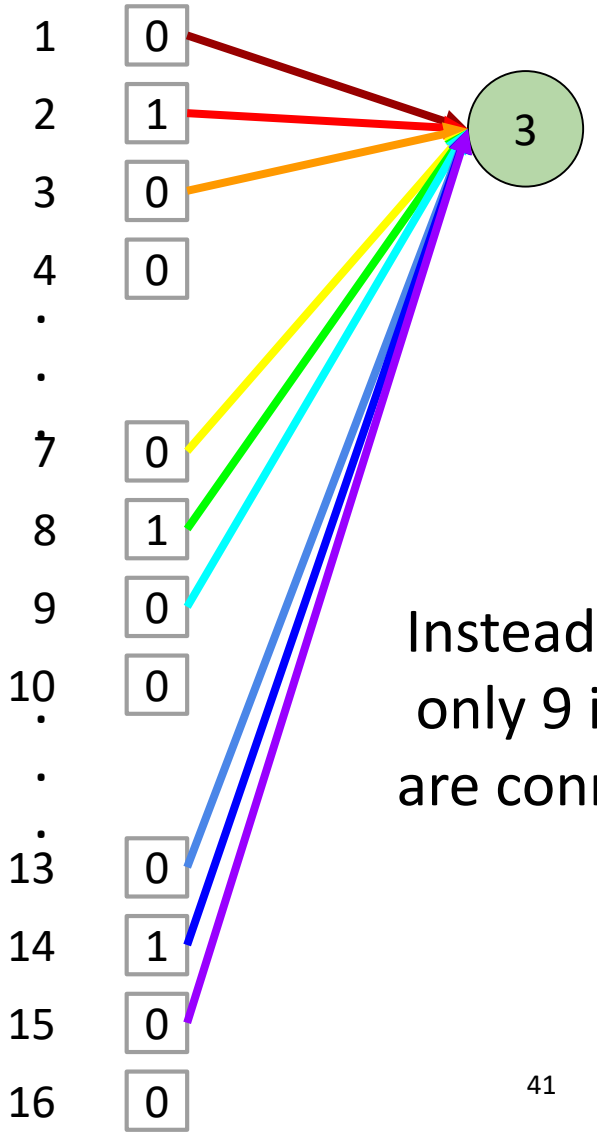


Convolution vs Fully Connected



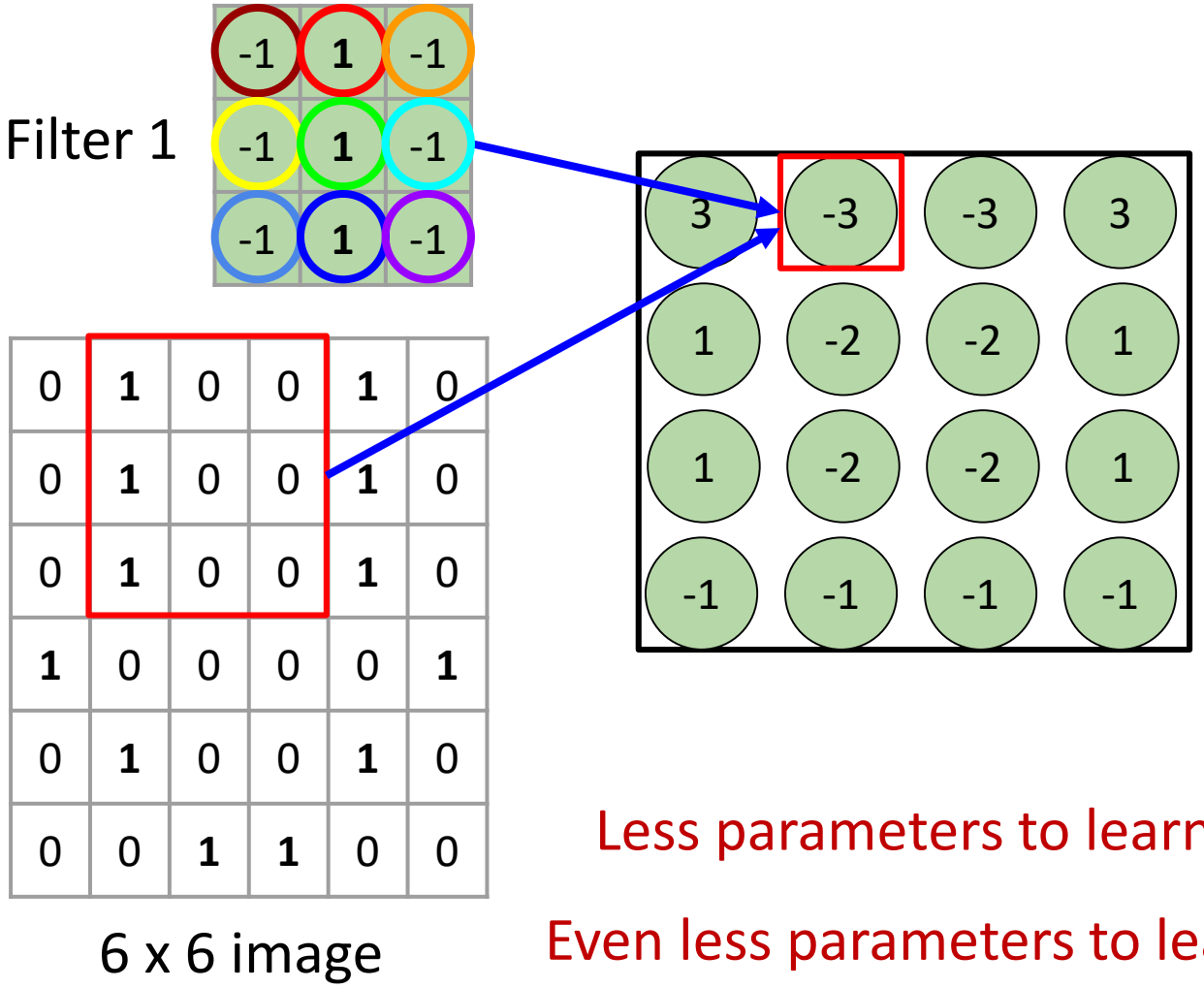
6 x 6 image

Less parameters to learn!



Instead of 36, only 9 inputs are connected

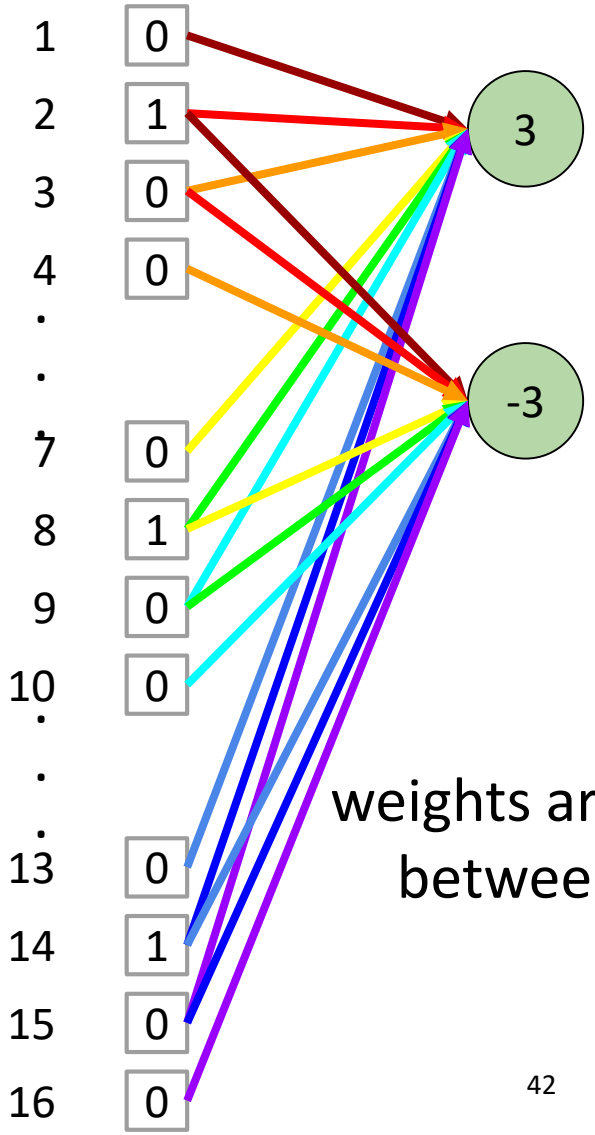
Convolution vs Fully Connected



6 x 6 image

Less parameters to learn!

Even less parameters to learn!

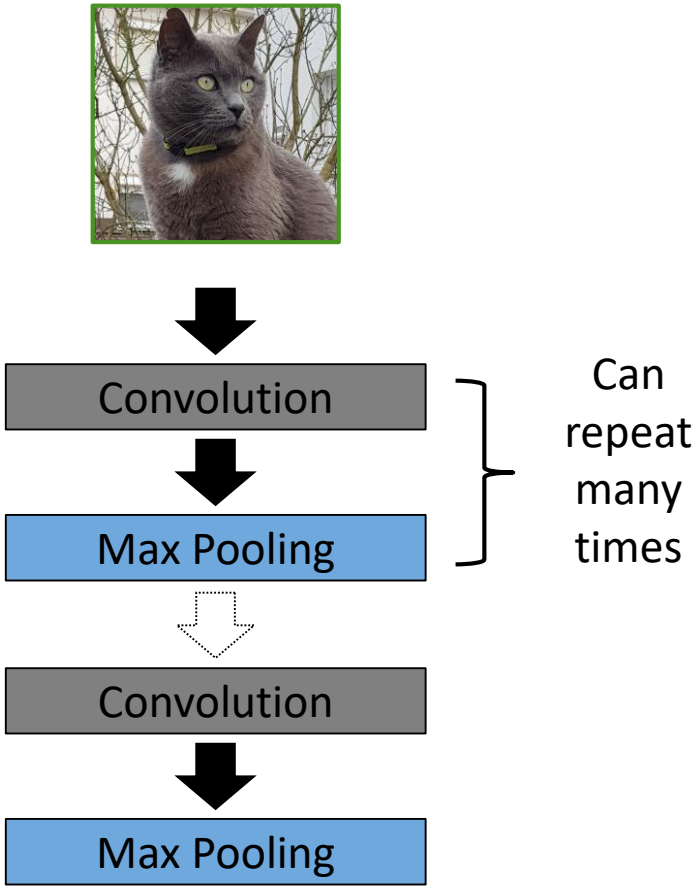


weights are shared between cells

- **The core idea**
  - Instead of having a large, dense linear layer with a connection from every input to every output, we have lots of small convolutional layers
    - Convolutional layers usually with fewer inputs and a single output
  - The result is a smaller subset of kernel predictions, which are used as input to the next layer.
  - Convolutional layers usually have many kernels

- **The core idea**
  - Each kernel to learn a particular pattern and then search for the existence of that pattern somewhere in the image
  - A single, small set of weights can train over a much larger set of training examples
  - This changes the ratio of weights to datapoints on which those weights are being trained
  - This has a powerful impact on the network, drastically reducing its ability to overfit to training data and increasing its ability to generalise

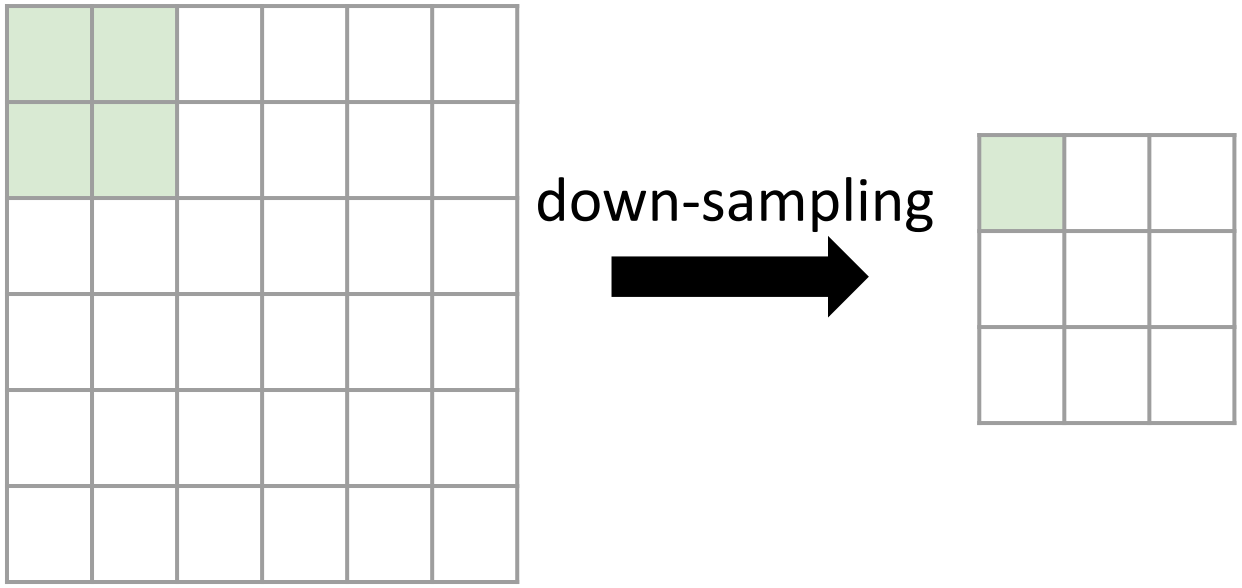
- **Convolutional Layers**
  - **Property 1:** Some patterns are much smaller than the whole image
  - **Property 2 :** The same patterns appear in different regions
- **Pooling Layers**
  - **Property 3** : Downsampling the pixels does not change the object



Max Pooling

Pooling layers are usually present after a convolutional layer.

They provide a down-sampled version of the convolution output.



In this example, a 2x2 region is used as input of the pooling.

There are different types of pooling, the most used is max pooling.

Max Pooling

Filter 1

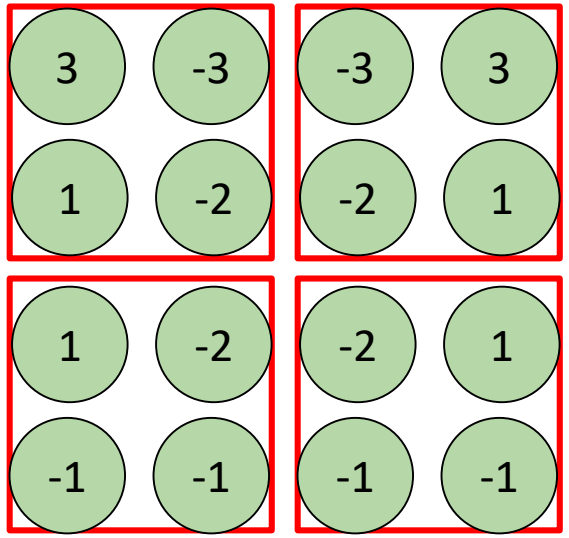
-1	**1**	-1
-1	**1**	-1
-1	**1**	-1

Pooling size = 2 x 2

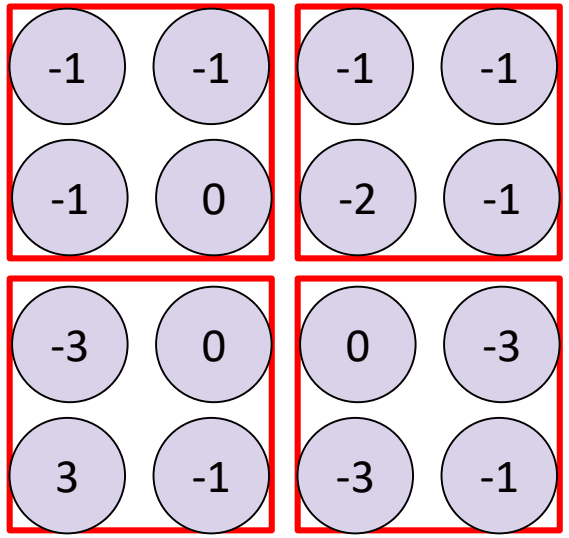
Stride = 2

Filter 2

1	-1	-1
-1	**1**	-1
-1	-1	**1**



Feature map 1



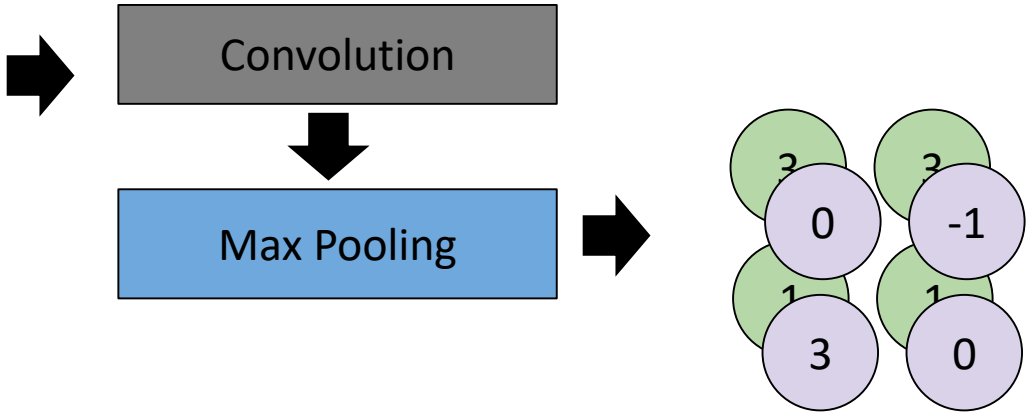
Feature map 2

Operates over each feature map independently

Invariant to small differences in the input

Max Pooling

0	**1**	0	0	**1**	0
0	**1**	0	0	**1**	0
0	**1**	0	0	**1**	0
1	0	0	0	0	**1**
0	**1**	0	0	**1**	0
0	0	**1**	**1**	0	0



6 x 6 image

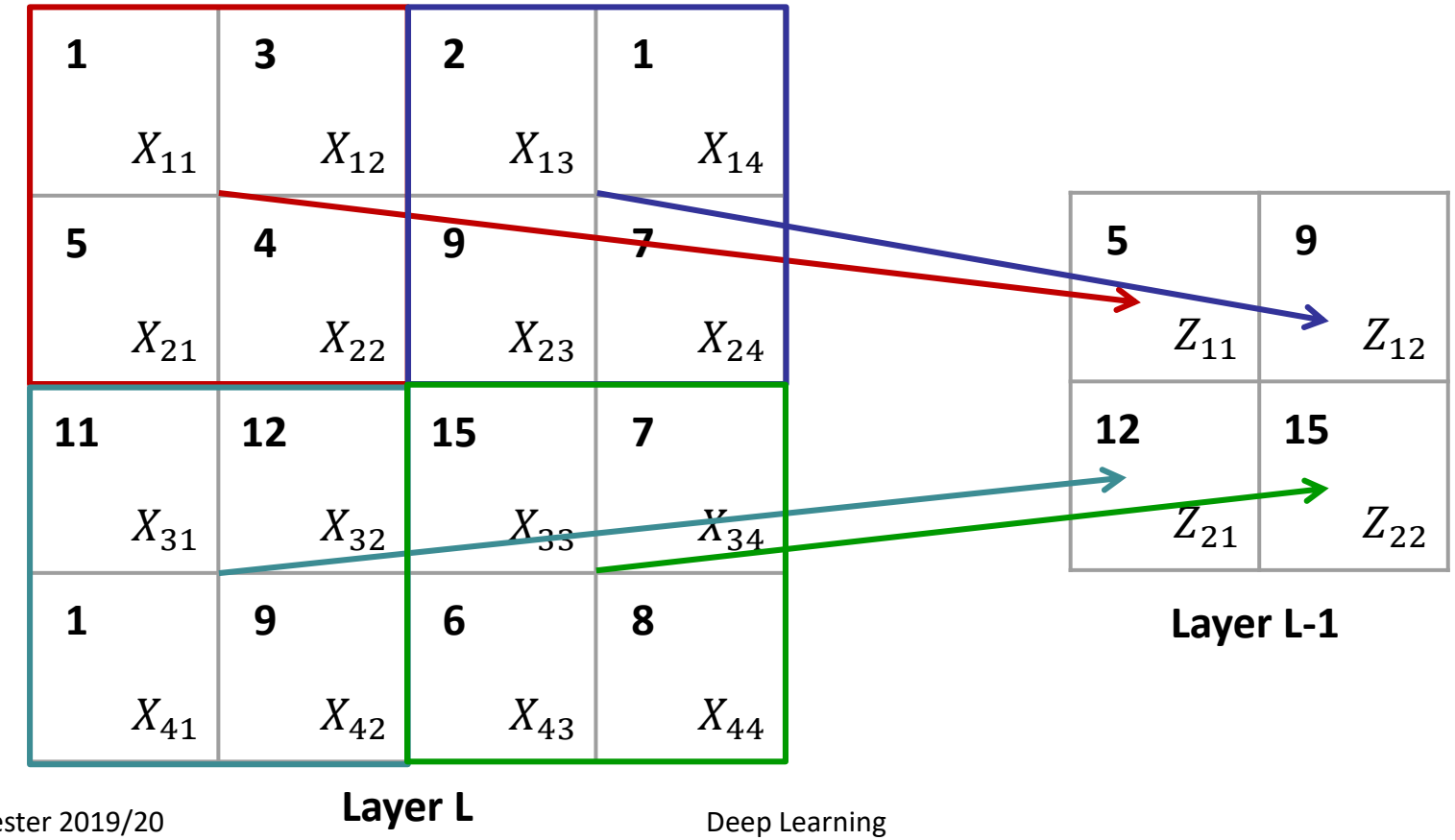
each filter is a channel

2 feature maps

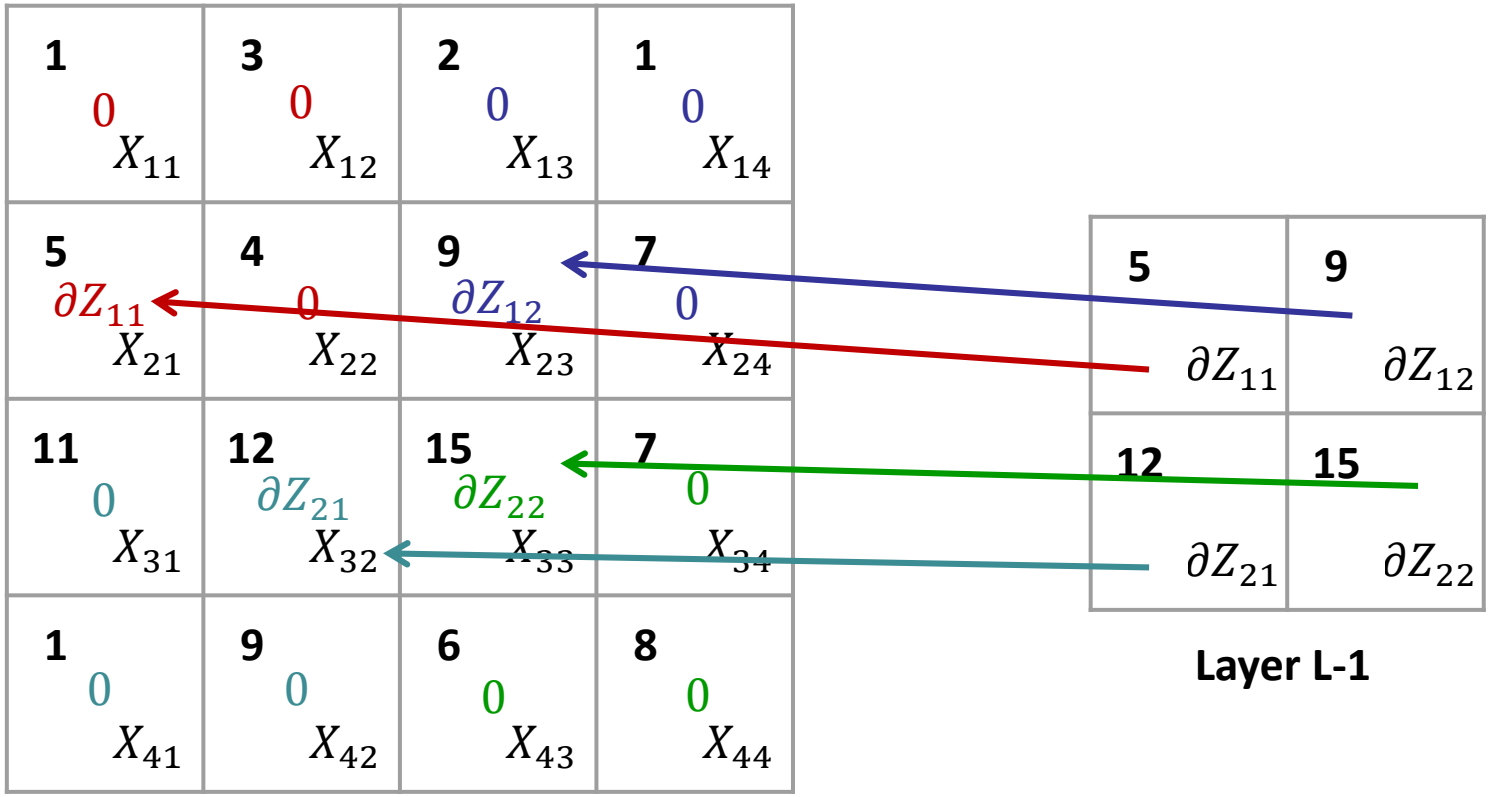
each of size 2 x 2

Smaller and more manageable

- **Worked Example**
  - Receptive field for max pooling is 2×2, and the stride size is 2



- **Worked Example**
  - Gradient gets passed to the cell it came from, all other cells get 0



Max Pooling – Parameters

Key Parameters:

- Accepts an input of size $\mathbf{W_1}$ x $\mathbf{H_1}$ X $\mathbf{D_1}$
- Requires 2 hyperparameters:
  - Size of the filters **F**
  - The stride **S**
- Produce an output of size $\mathbf{W_2}$ x $\mathbf{H_2}$ x $\mathbf{D_2}$, where
  - $\mathbf{W_2} = (\mathbf{W_1} - \mathbf{F})/\mathbf{S} + 1$
  - $\mathbf{H_2} = (\mathbf{H_1} - \mathbf{F})/\mathbf{S} + 1$
  - $\mathbf{D_2} = \mathbf{D_1}$
- It introduces **zero** learnable parameters since it computes a fixed function of the input.

Common settings:

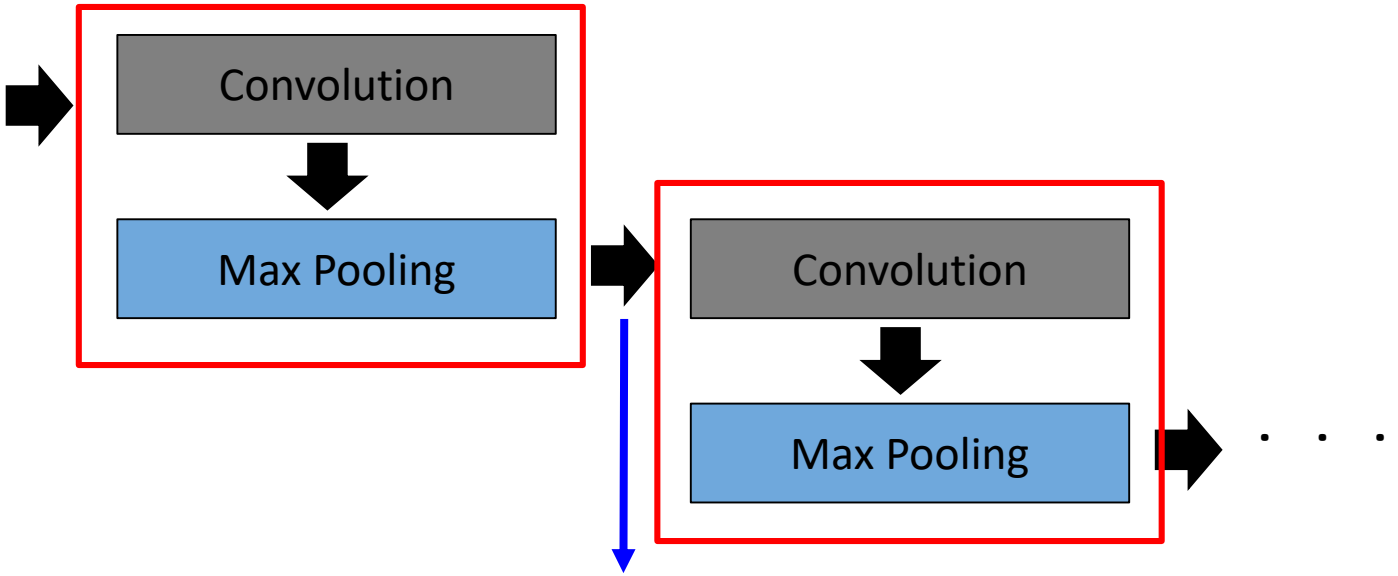
F = 2, S=2

F = 3, S=2

… …

Convolve, Pool, Repeat

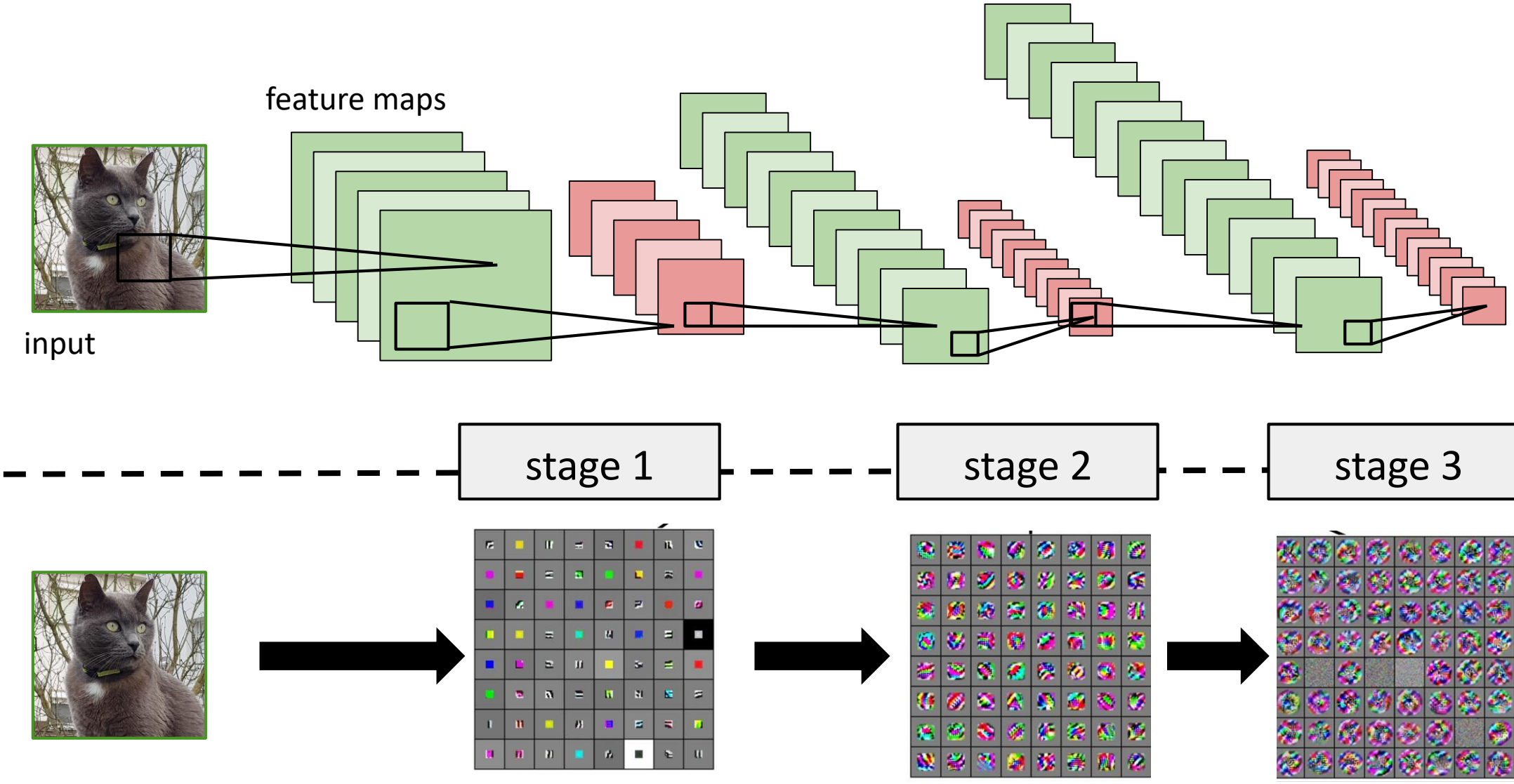
Can be repeat many times



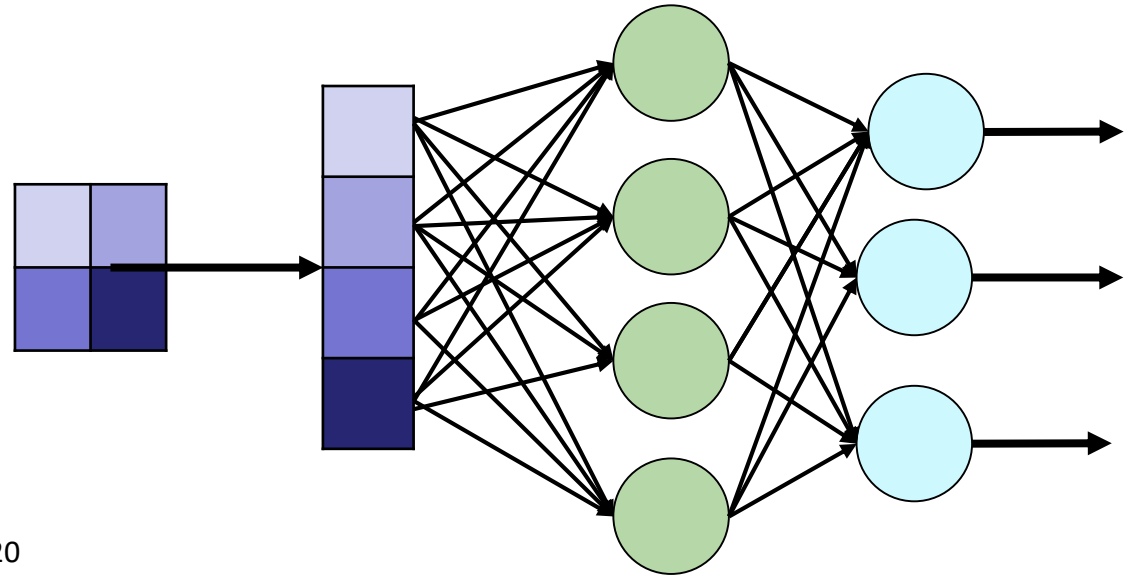
Output can be regarded as new images:

- Smaller than the original images
- The depth of new images is the number of filters

Representation learning with CNN



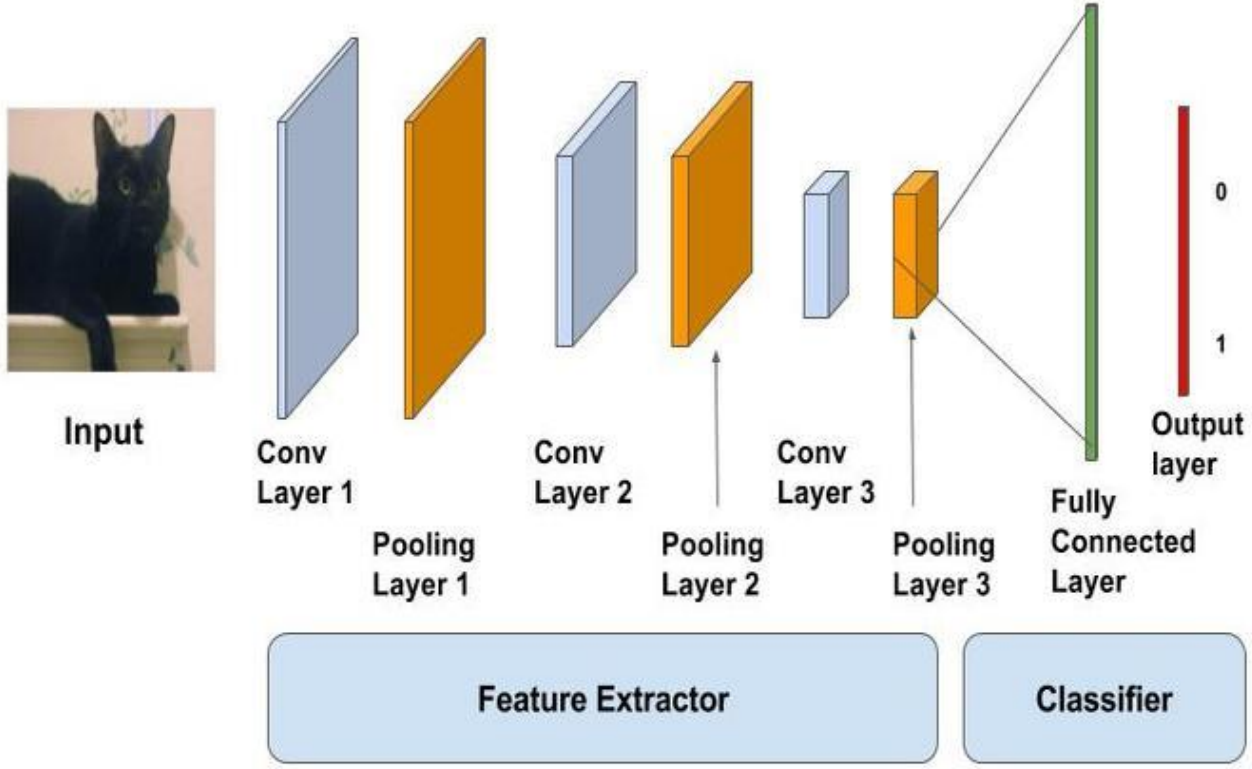
- **Flatten the feature maps**
  - Takes the output of the previous layers, and “flattens” them and turns them into a single vector
  - This representation is then feed into a standard feedforward network to perform classification



Fully Connected Layers

Image source: https://www.learnopencv.com/

- **Flatten the feature maps**
  - **First fully connected layer**:
    - Takes the inputs from the feature analysis and applies weights to predict the correct label
  - **Fully connected output layer**:
    - Gives the final probabilities for each label



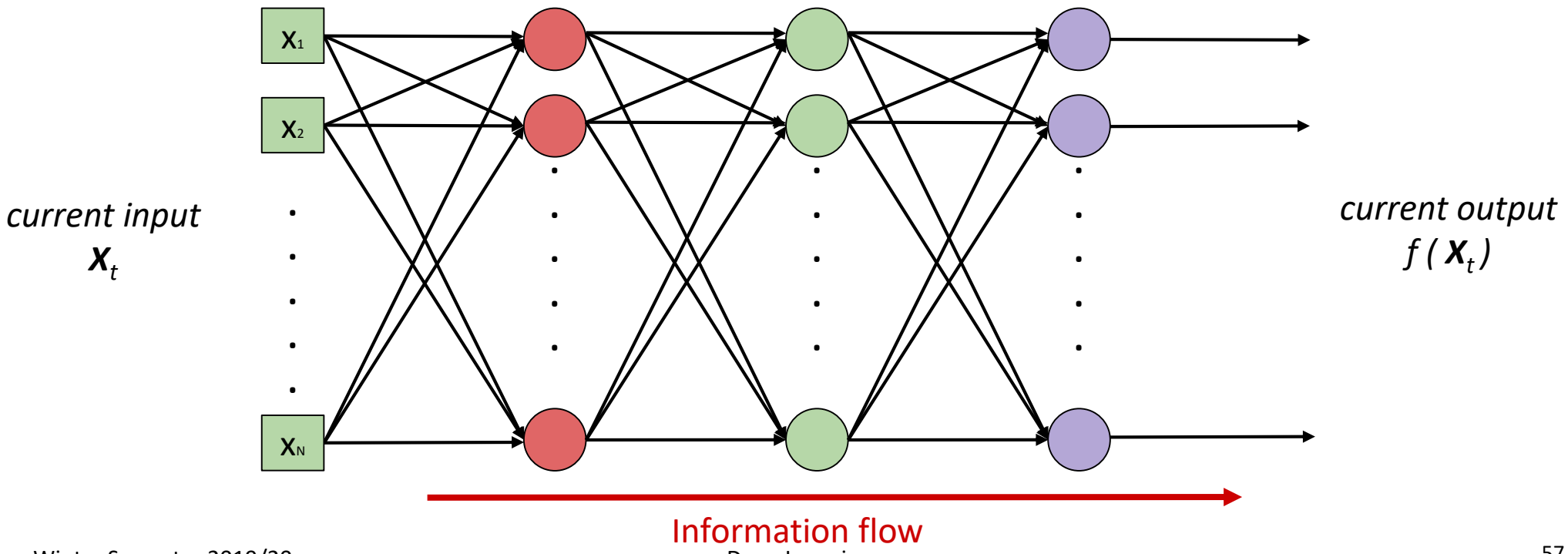
Universität Augsburg University

Reccurent Neural Network (RNNs)

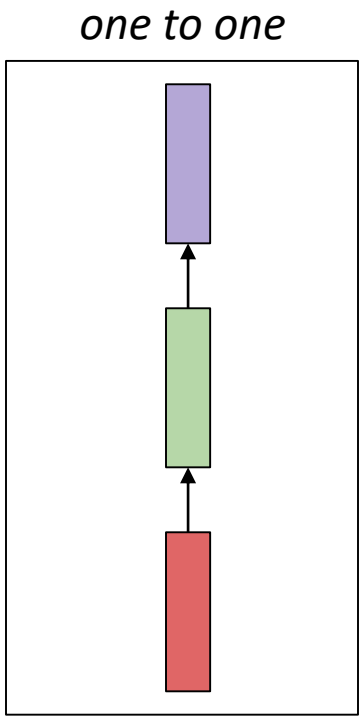
Perform predict over arbitrary-length sequences

"Vanilla" Neural Networks

- Information only propagates forward through the network
  - Previous, and future, results do not affect current prediction



“Vanilla” Neural Networks



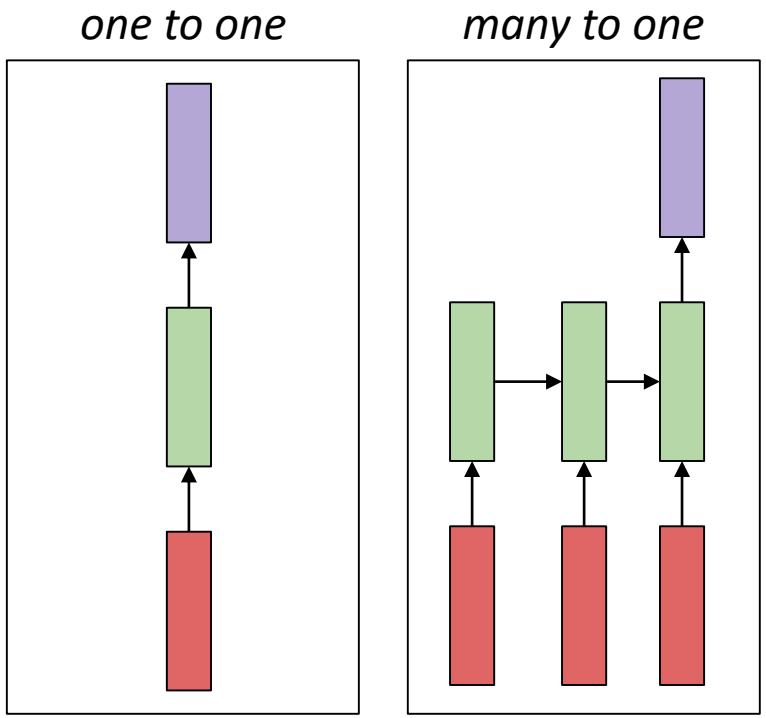
Feedforward Neural Network:

- *one input → one output*
- *classifier or regressor*
- ***no context** information*

Neural Network needs **memory**!

Can the information/knowledge from **previous inputs** be included?

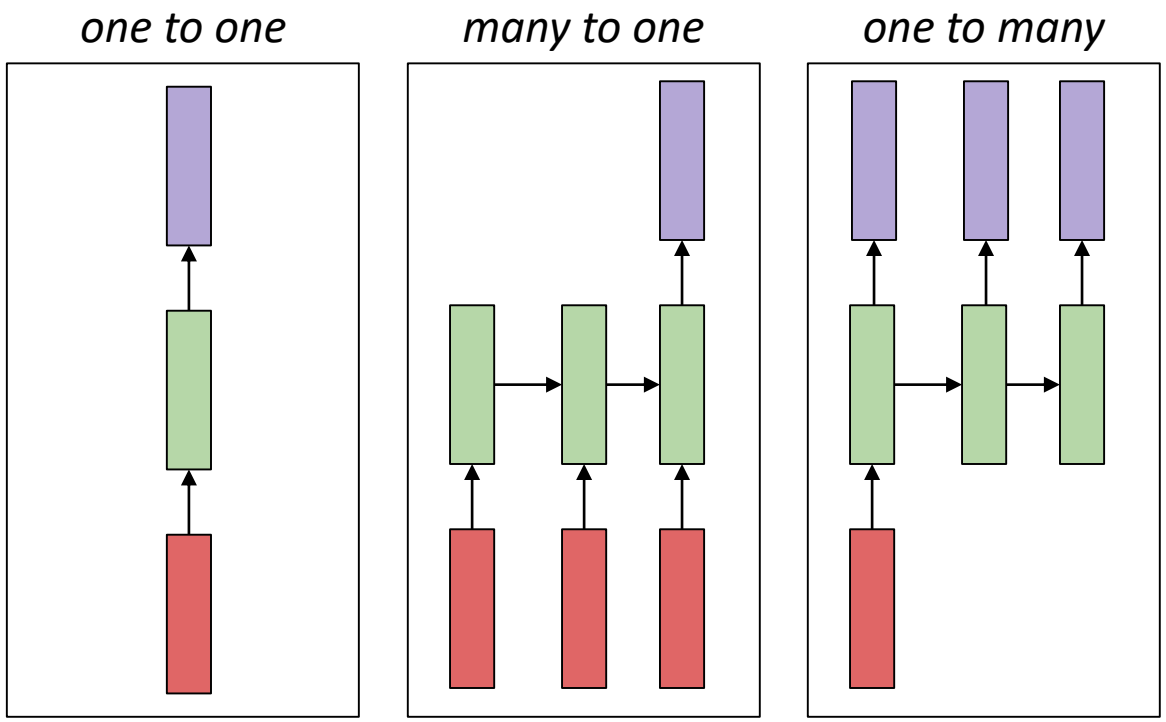
Processing sequential inputs/outputs



Flexible to handle sequential inputs of varying length

- *speech recognition: speech waves → one word*
- *sentiment classification: words → positive/negative*
- *video classification: images → one category*

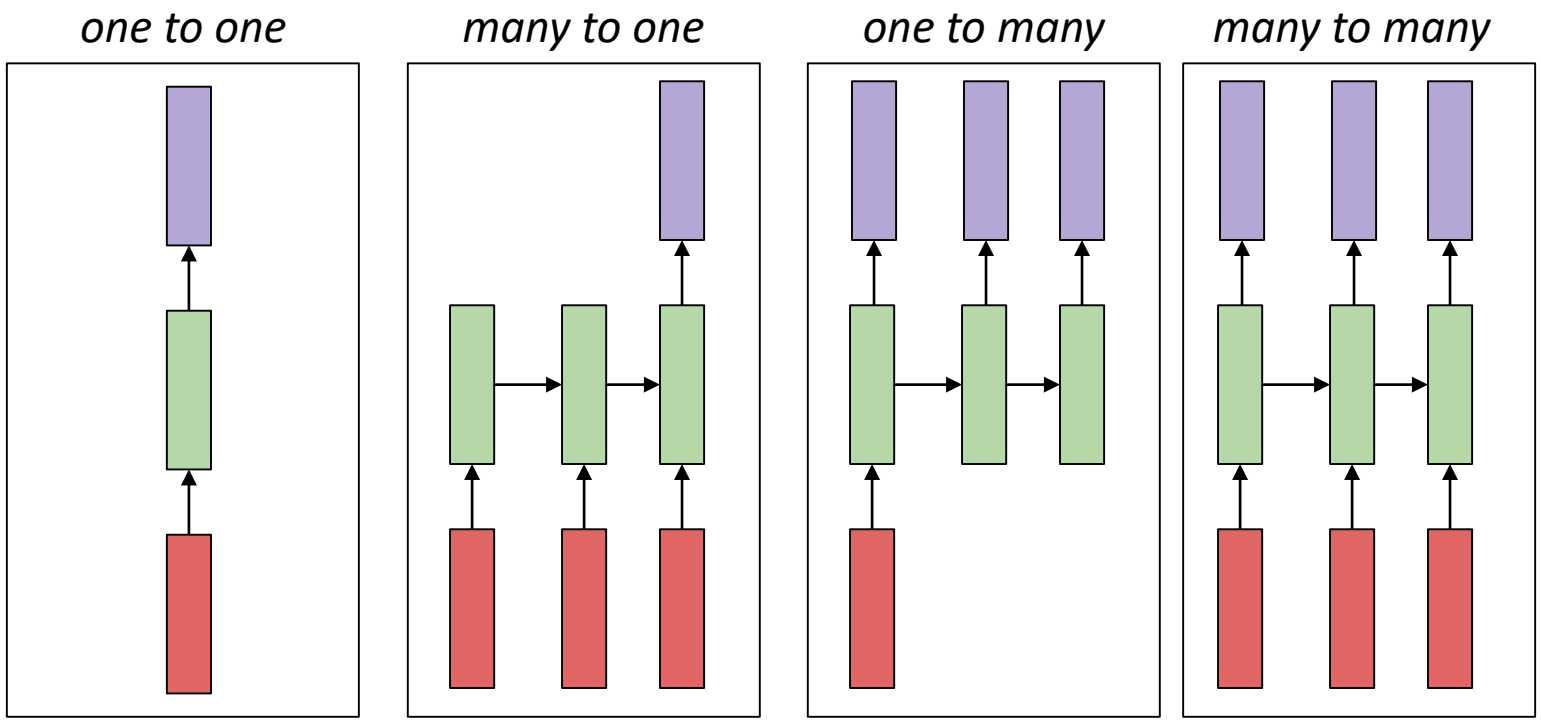
Processing sequential inputs/outputs



To generate sequential outputs

- *music composition*
- *image captioning*
- *natural text generation*

Processing sequential inputs/outputs

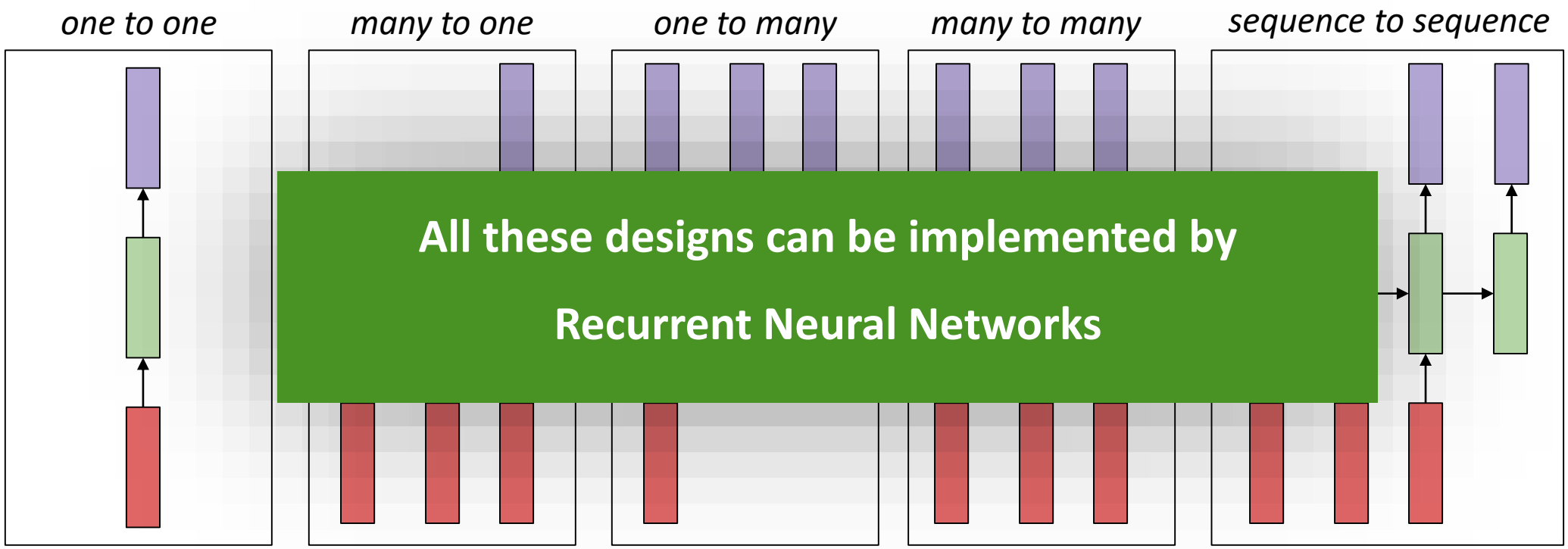


Inputs and outputs of same length

- *continuous emotion prediction*
- *speech enhancement*
- *video classification on frame level*

Processing sequential inputs/outputs

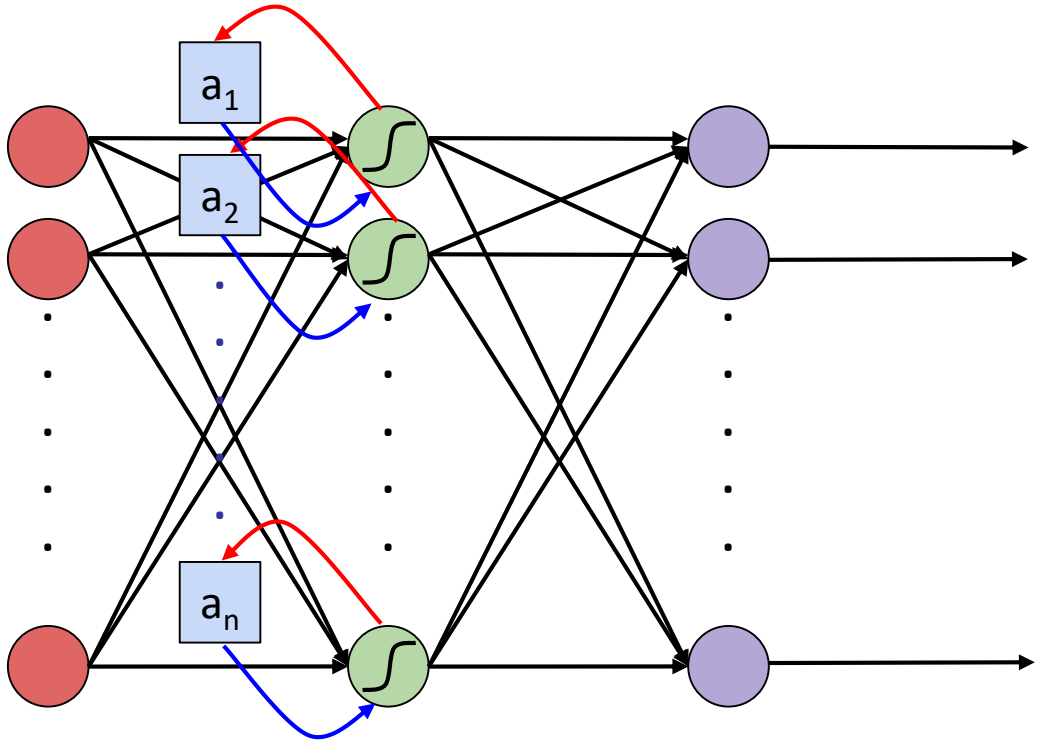
Different lengths:
- ➢ *machine translation*
- ➢ *question answering*



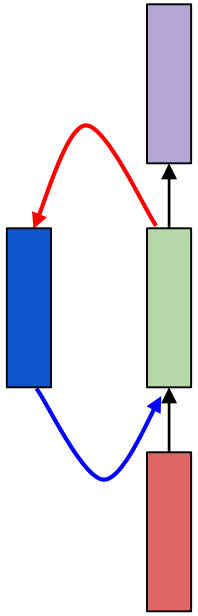
Inclusion of feedback into the network structure

Output of hidden layer are **stored** in the memory

Values in the memory are considered as **additional input** in the next time step



Inclusion of feedback into the network structure



memory to store the hidden states

Process a sequence of vectors ***x*** by applying a *recurrence formula* at every time step:

$$h_t = f_w(h_{t-1}, x_t)$$

current state

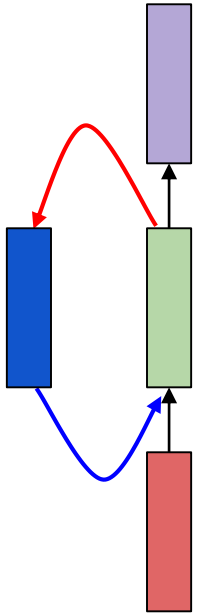
function with parameters W

state in memory (previous state)

current input vector

The same function and the same parameters are used at every time step.

Inclusion of feedback into the network structure



Process a sequence of vectors ***x*** by applying a *recurrence formula* at every time step:

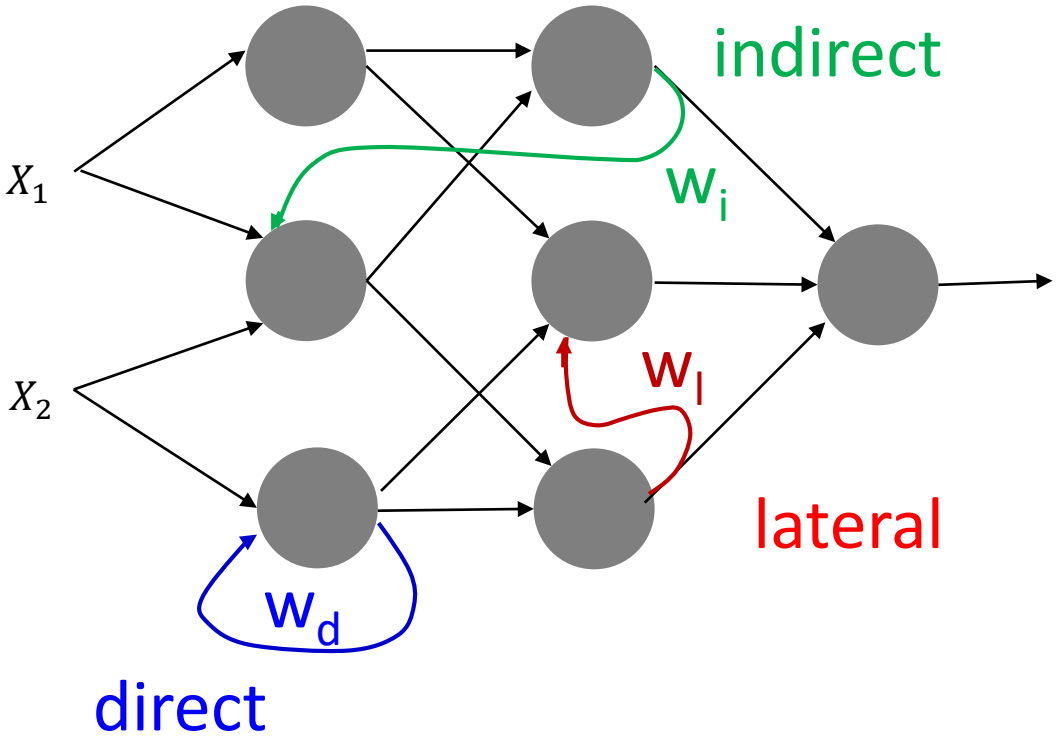
$$h_t = f_w(h_{t-1}, x_t)$$

$$h_t = tahn(W_{hh} h_{t-1} + W_{xh} x_t)$$

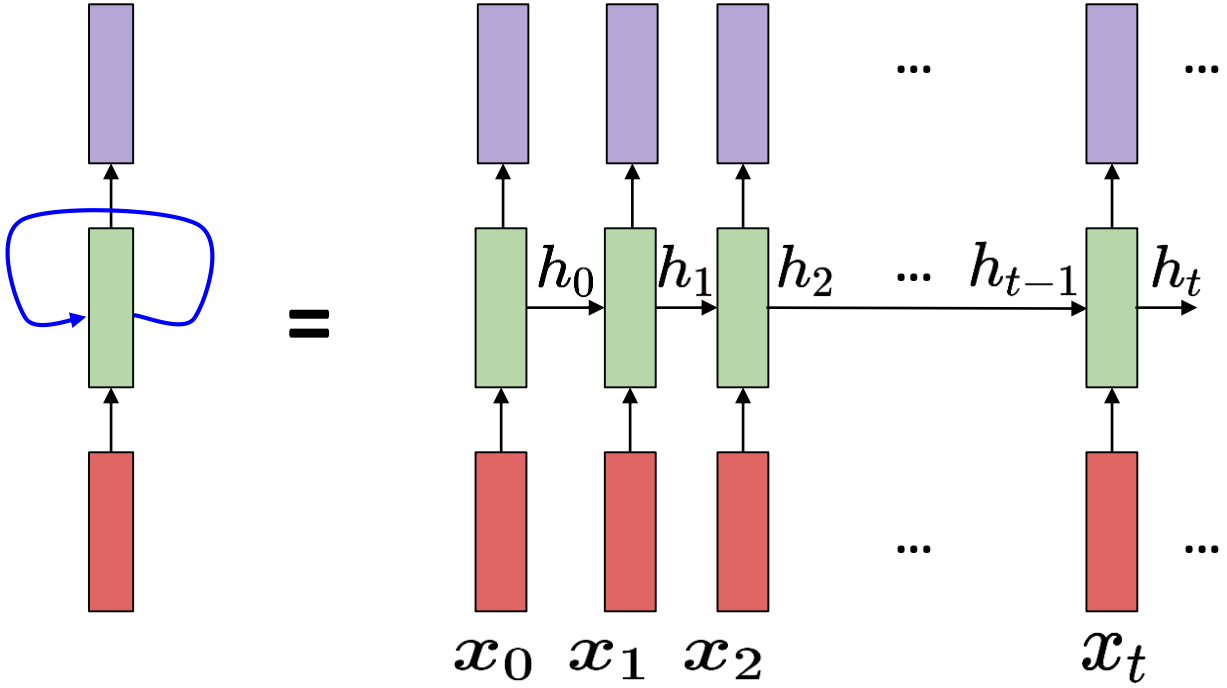
$$y_t = W_{hy} h_t$$

Recurrent Connections

- Direct feedback
  - There exists a connection from a neuron j to itself with weight $w_{j,j}$
- Indirect feedback
  - Activity is sent back to the previous layer in the neural network
- Lateral feedback:
  - Connect neurons within a layer

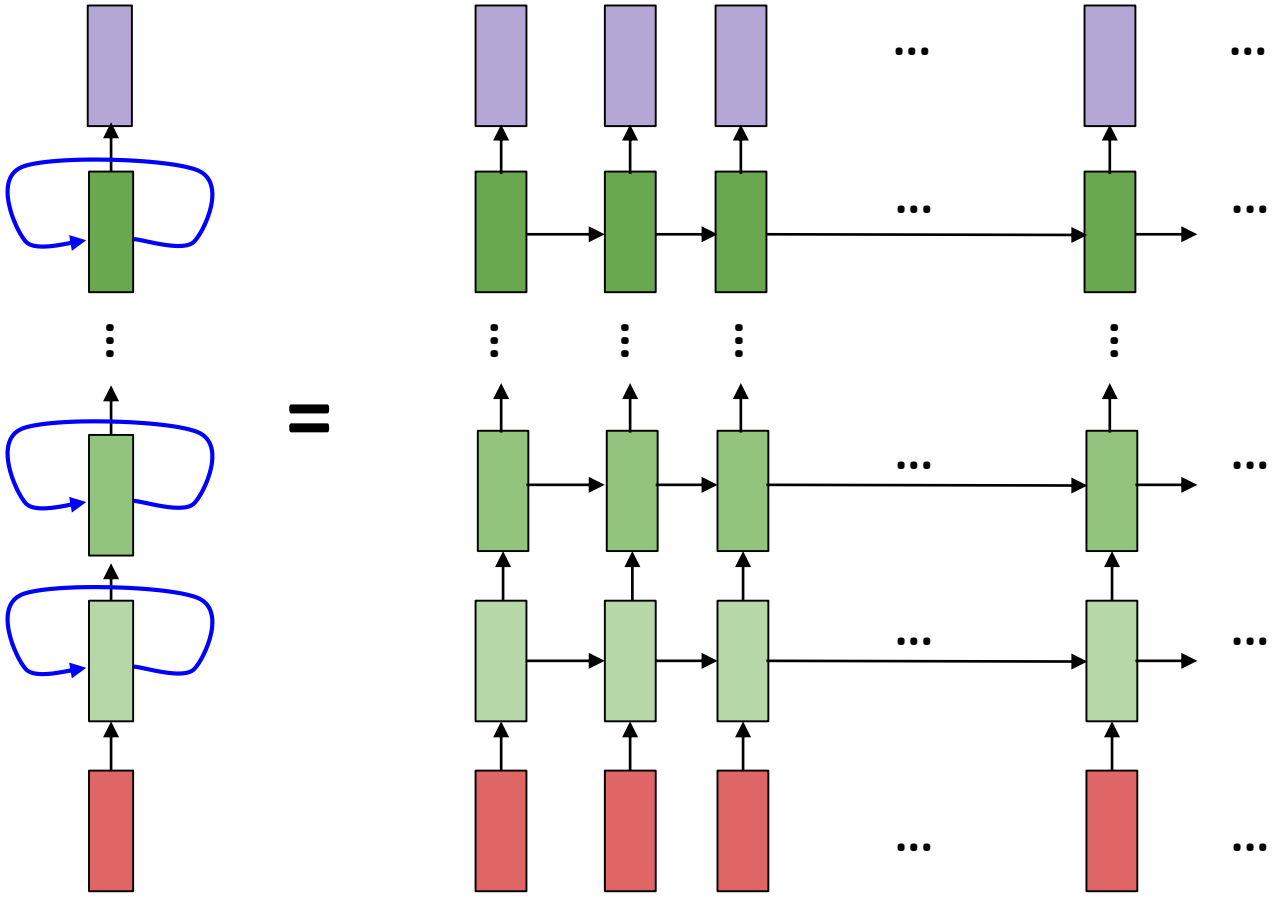


Unrolled RNN



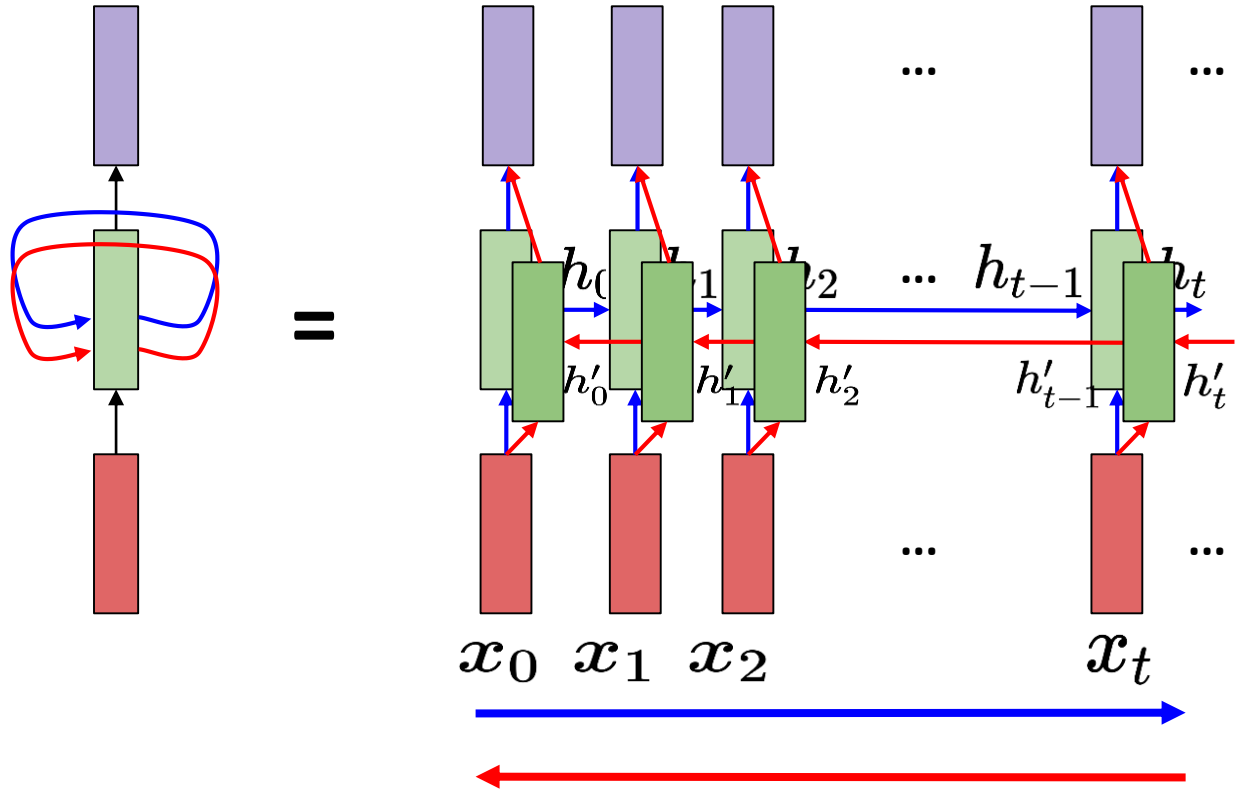
- Reuse the same weight matrix at every time step
- Makes the network easier to train

Deep RNN



- Not easy to train
- Regularisation helps
  - *Batch normalisation*
  - *Dropout*
  - *These topics will be covered in future lecture*

Bidirectional RNN

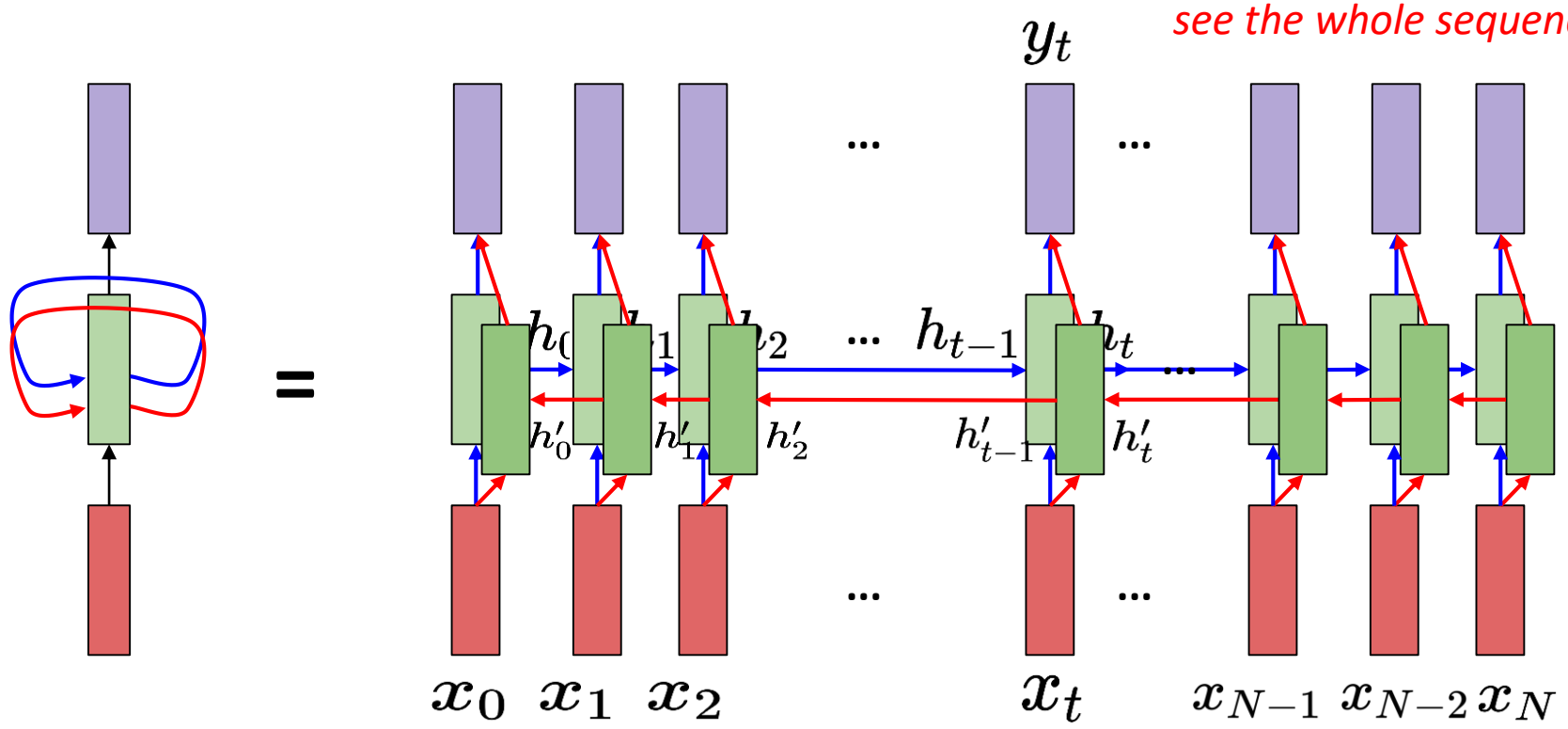


Two hidden layers:

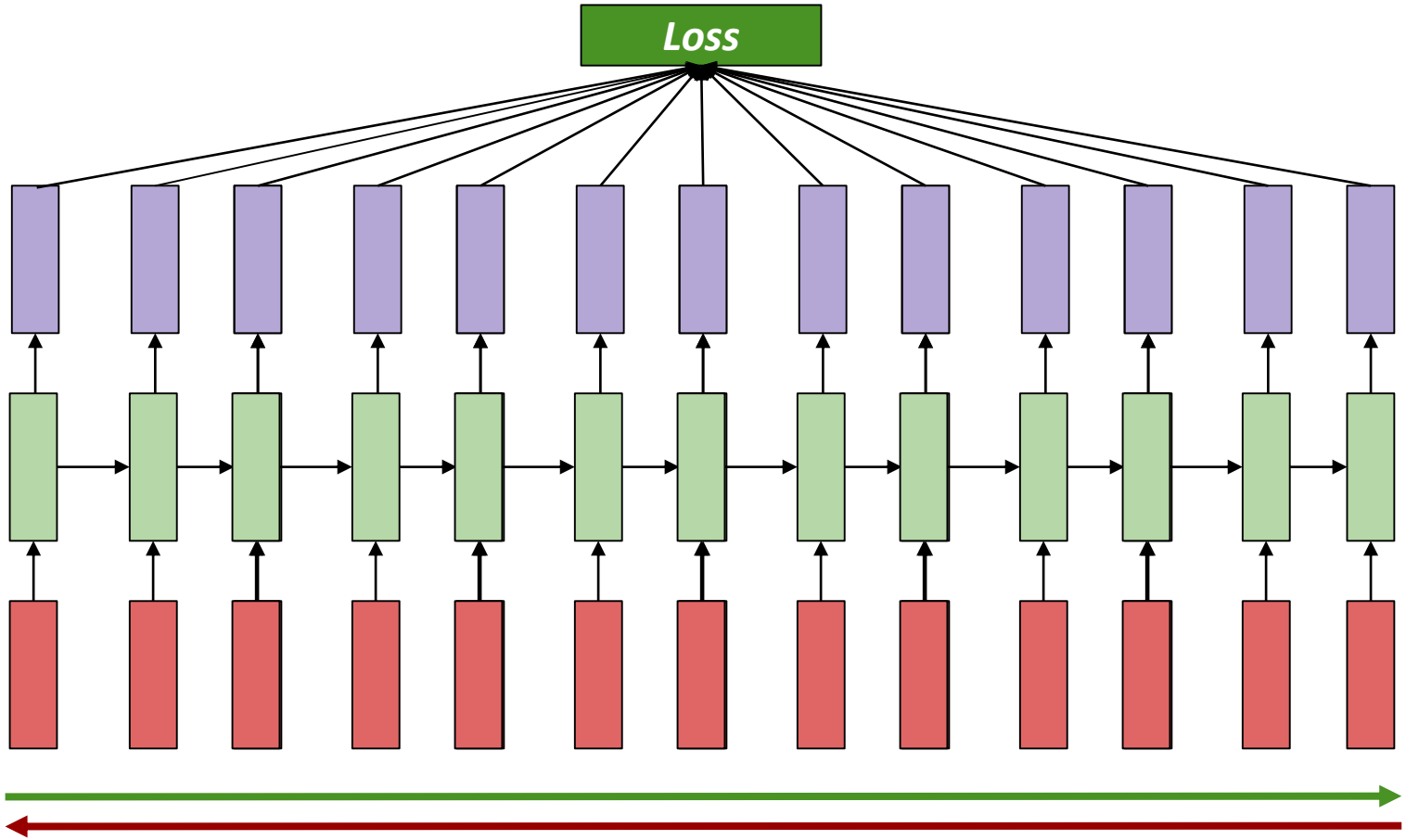
- *Scan the input sequences separately*
- *One forward and one backward*
- *Connect to the same output layer*
- *Learn both past and future context*

Bidirectional RNN

To determine y_t you have to see the whole sequence



Backpropagation through time (BPTT)



Forward

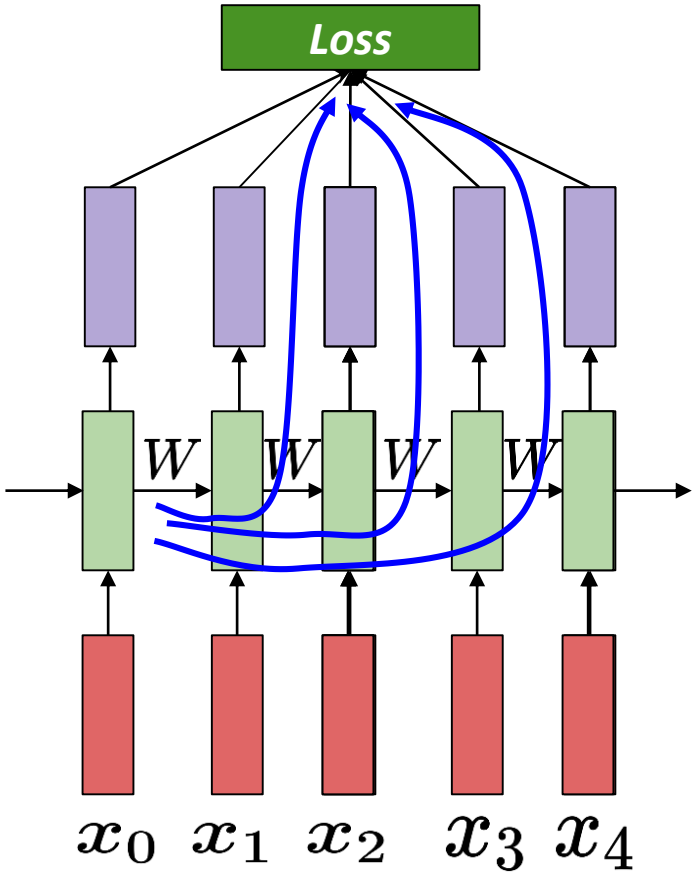
Run through entire sequence to compute the ***loss***

Backward

Run through entire sequence to compute the **gradient** and update the weight matrix:

$$w \leftarrow w - \eta \partial L / \partial w$$

Gradient flow in simple RNNs



$$w \leftarrow w - \eta \partial L / \partial w$$

Issue: **W** occurs each timestep

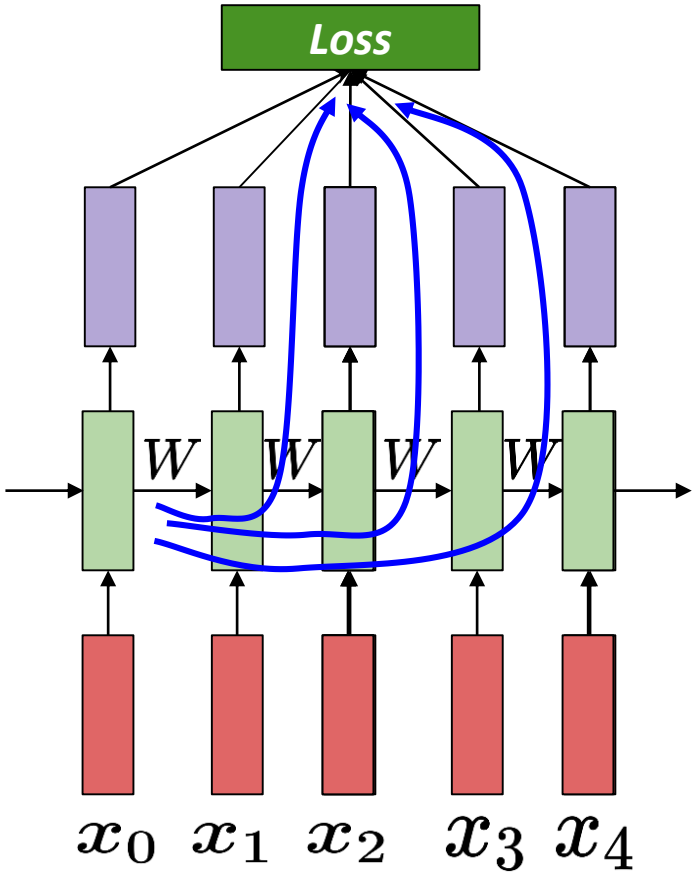
Every path from **W** to ***Loss*** is one dependency

All paths from **W** to ***Loss*** need to be involved

We need to sum up the gradients at each time step

$$\frac{\partial L}{\partial w} = \sum_{j=0}^{T} \frac{\partial L_j}{\partial w}$$

Gradient flow in simple RNNs



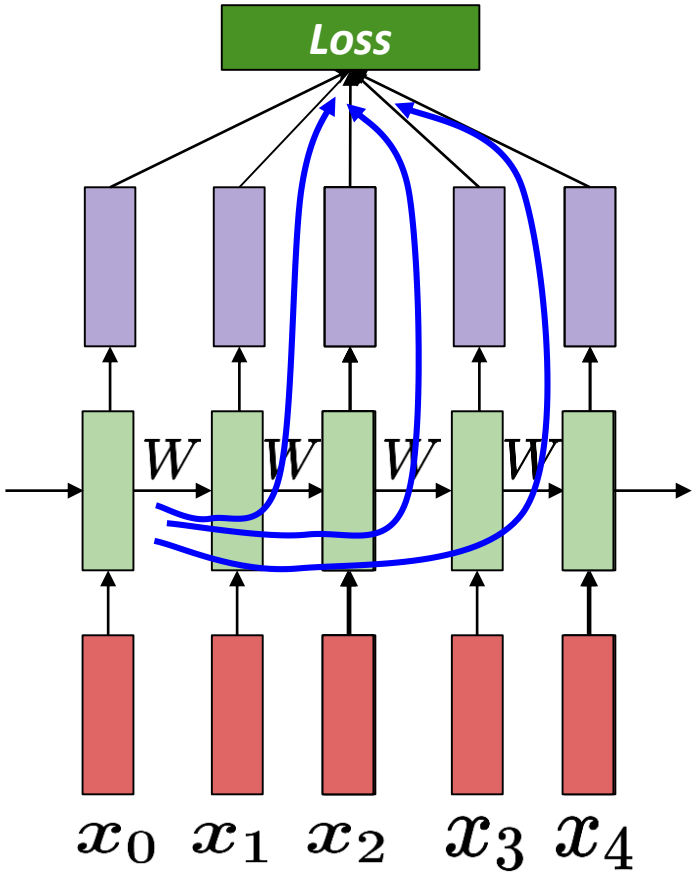
We need to **sum** up the **gradients** at each time step

$$\frac{\partial L}{\partial w} = \sum_{j=0}^{T} \frac{\partial L_j}{\partial w}$$

To calculate these gradients we use **multiple iterations** of chain rule of differentiation

$$\frac{\partial L}{\partial w} = \sum_{j=0}^{T} \sum_{k=1}^{j} \frac{\partial L_j}{\partial y_j} \frac{\partial y_j}{\partial h_j} \left(\prod_{t=k+1}^{j} \frac{\partial h_t}{\partial h_{t-1}} \right) \frac{\partial h_k}{\partial w}$$

Gradient flow in simple RNNs



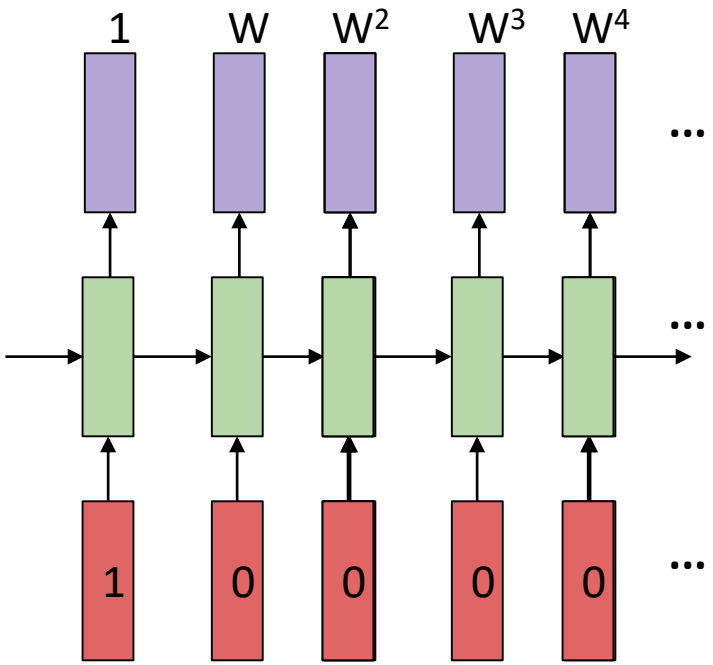
$$\frac{\partial L}{\partial w} = \sum_{j=0}^{T} \sum_{k=1}^{j} \frac{\partial L_j}{\partial y_j} \frac{\partial y_j}{\partial h_j} \left(\prod_{t=k+1}^{j} \frac{\partial h_t}{\partial h_{t-1}} \right) \frac{\partial h_k}{\partial w}$$

$$h_t = tahn(W_{hh} h_{t-1} + W_{xh} x_t)$$

computing gradient of h_{t-1} involves W_{hh}

Repeated matrix multiplications leads to ***vanishing*** and ***exploding*** gradients.

Gradient flow in simple RNNs



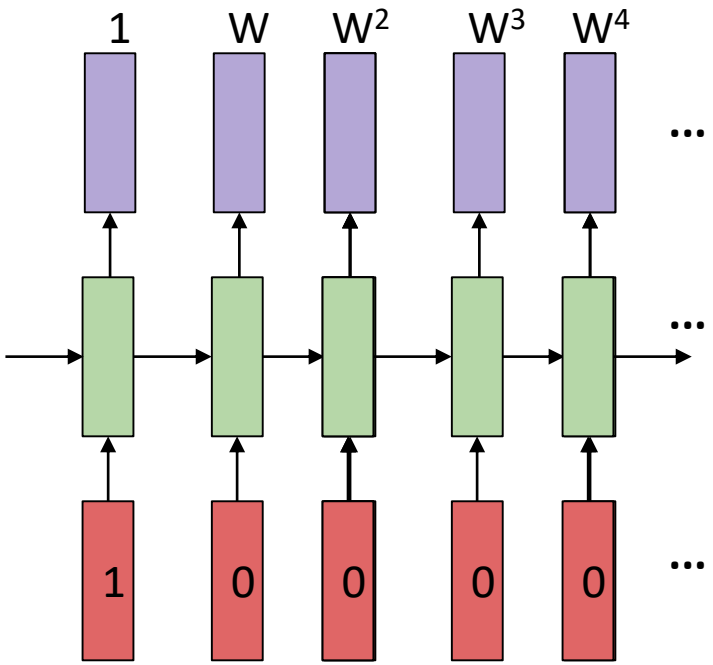
Toy Example:

- 1-d input, x_0=1, and x_t=0 (t>0)
- Sequence length is **1000**
- W_{xh}=W_{hy}=1
- Linear activation function

$$h_t = W_{hh} h_{t-1} + x_t$$
$$y_t = h_t$$

$$y_{1000} = W_{hh}^{999}$$

Gradient flow in simple RNNs



$$y_{1000} = W_{hh}^{999}$$

$w_{hh} = 1$ → $y_{1000} = 1$

$w_{hh} = 1.01$ → $y_{1000} \approx 20959$

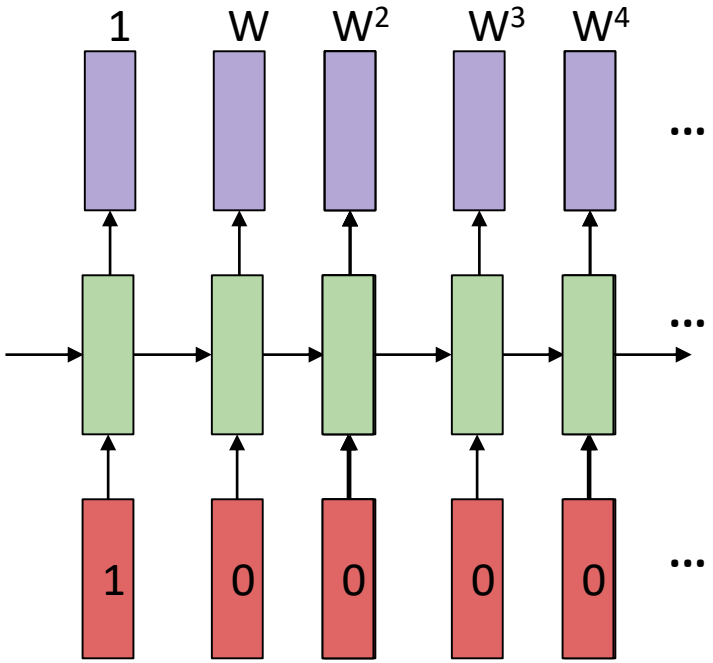
$\partial L / \partial w$ is too large

➔ **Exploding gradients**

- Regulate via gradient clipping
  - Scale gradient if value gets to big

$$if\ \|g\| > threshold,\ g \leftarrow \frac{threshold \times g}{\|g\|}$$

Gradient flow in simple RNNs



$$y_{1000} = W_{hh}^{999}$$

$w_{hh} = 0.99 \rightarrow y_{1000} \approx 0$

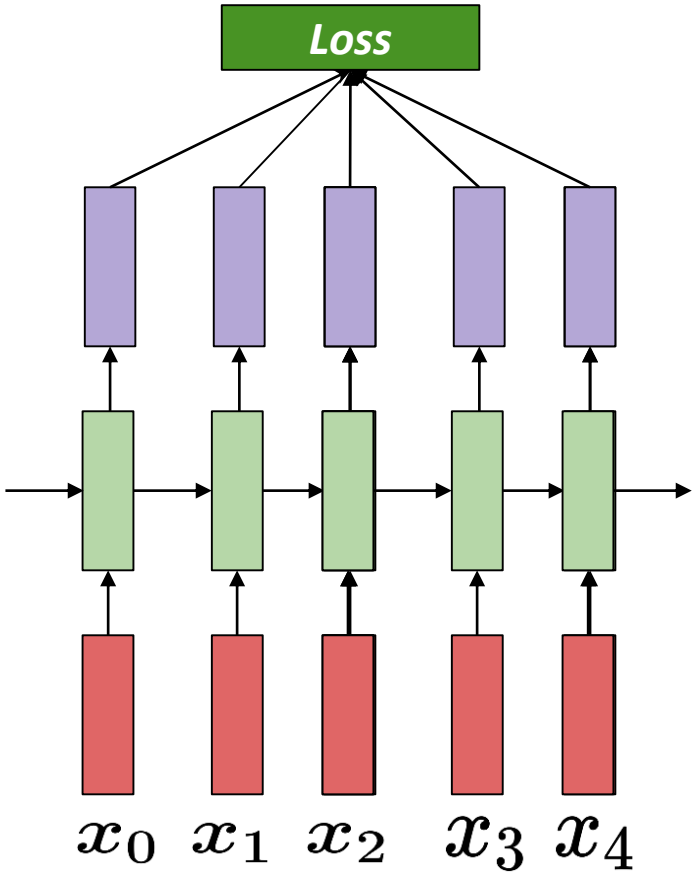
$w_{hh} = 0.01 \rightarrow y_{1000} \approx 0$

$\partial L / \partial w$ is too small

→ Vanishing gradients

- Need to change RNN architecture
- **Gated RNNs**

Identity-RNN



- Network weights are initialised to the identity matrix
- Activation functions are all set to ReLU
- This encourages the network computations to stay close to the identity function
- **Gadient** does **not decay** as derivatives stay either 0 or 1
- Error is propagated all the way back

$$\frac{\partial L}{\partial w} = \sum_{j=0}^{T} \sum_{k=1}^{j} \frac{\partial L_j}{\partial y_j} \frac{\partial y_j}{\partial h_j} \left(\prod_{t=k+1}^{j} \frac{\partial h_t}{\partial h_{t-1}} \right) \frac{\partial h_k}{\partial w}$$

$$h_t = h_{t-1} + f(x_t) \quad \rightarrow \left(\frac{\partial h_t}{\partial h_{t-1}} \right) = 1$$

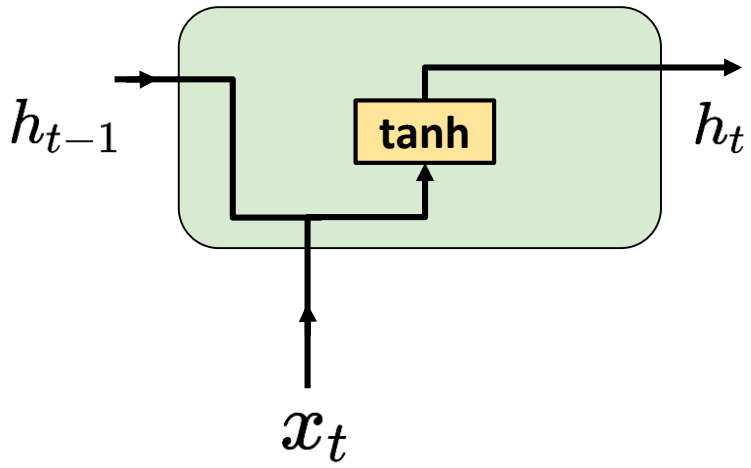
Highly effective sequence models

- Gated RNNs are based on the idea of creating paths that have derivatives that neither vanish nor explode
- Gated RNNs have connection weights that may change at each time step
- Gated RNNs also allow a network to forget an old state
- Instead of manually deciding when to clear the state, the network to learn to decide when to do it.

- **The LSTM Recurrent Unit**
  - The internals structure of an LSTM unit allows is to keep or forget information over time
  - Cell State: the memory of the network, carries relevant information throughout the processing of the sequence
  - Information get's added or removed to the cell state via gates
    - Forget gate: what is relevant to keep from prior step
    - Input gate: what information is relevant to add from current step
    - Output gate: what the next hidden state should be
  - The gates learns to determine what is relevant during training

Long Short-term Memory (LSTM)

Simple RNN unit



Gated LSTM unit

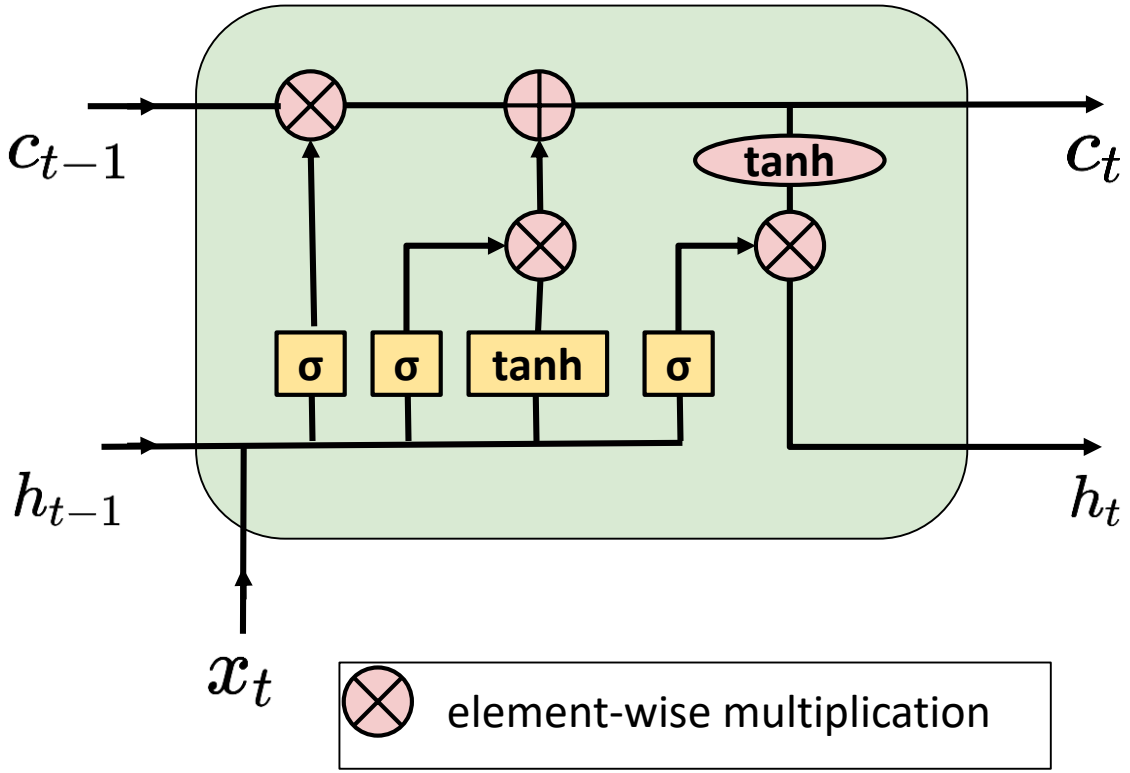


Image source: https://colah.github.io/

- **Unrolled LSTM network**
  - There are four interacting networks: cell state, input gate, forget gate, output gate

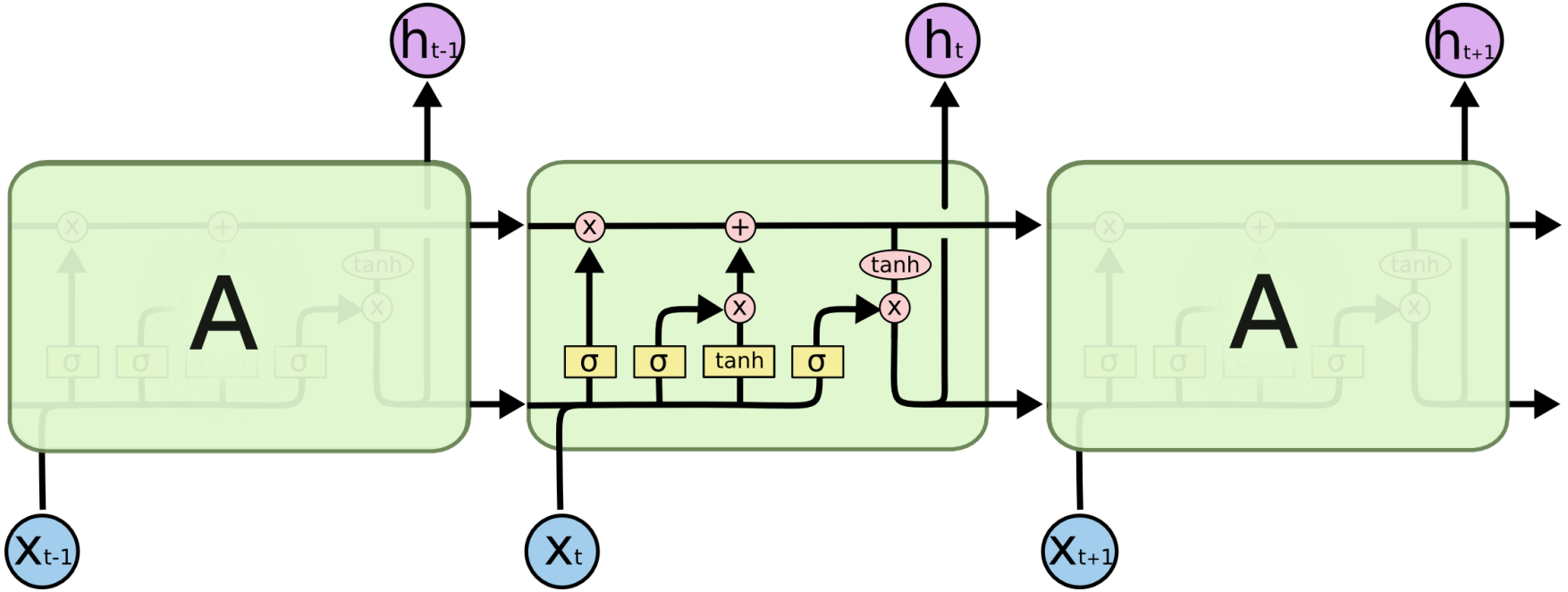
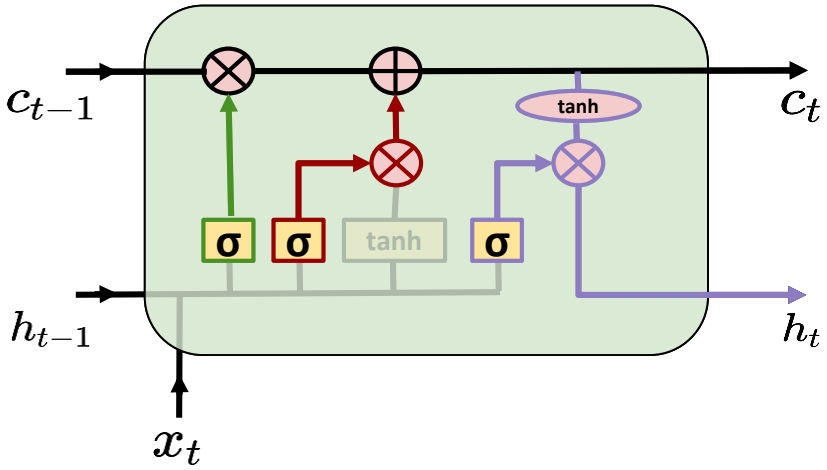


Image source: https://colah.github.io/

Long Short-term Memory (LSTM)



$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ tanh \end{pmatrix} W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}$$

$$c_t = f \odot c_{t-1} + i \odot g$$

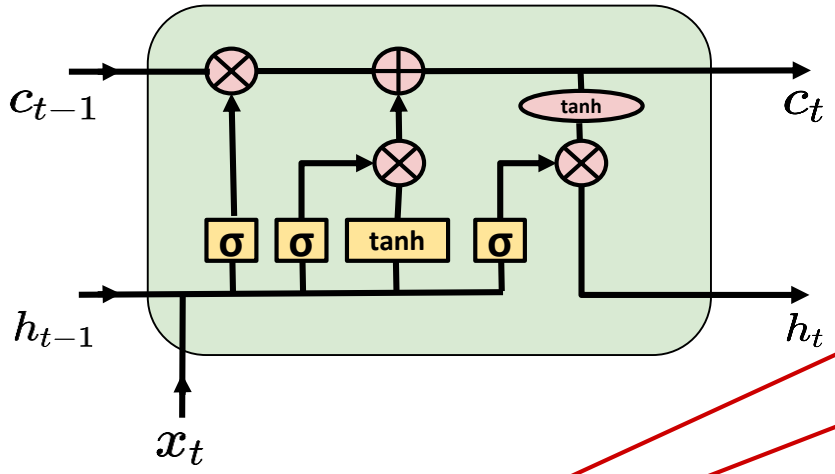
$$h_t = o \odot tanh(c_t)$$

Cell state *c*:

- Core of LSTM unit
- Information is removed or added to the cell
- Controlled by three gates:
  - *Input gate: how much to write to cell*
  - *Forget gate: how much to erase/keep in the cell*
  - *Output gate: how much to output from the cell*

Image source: https://colah.github.io/

Long Short-term Memory (LSTM)



$$g_t = tanh(W_g \cdot [h_{t-1}, x_t] + b_g)$$

$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$$

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)$$

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

$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o)$$

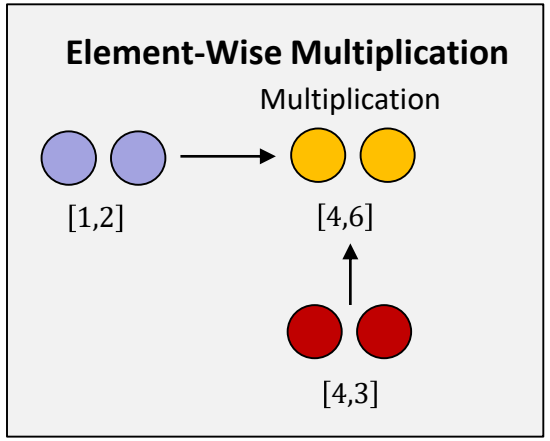
$$h_t = o_t \odot tanh(c_t)$$

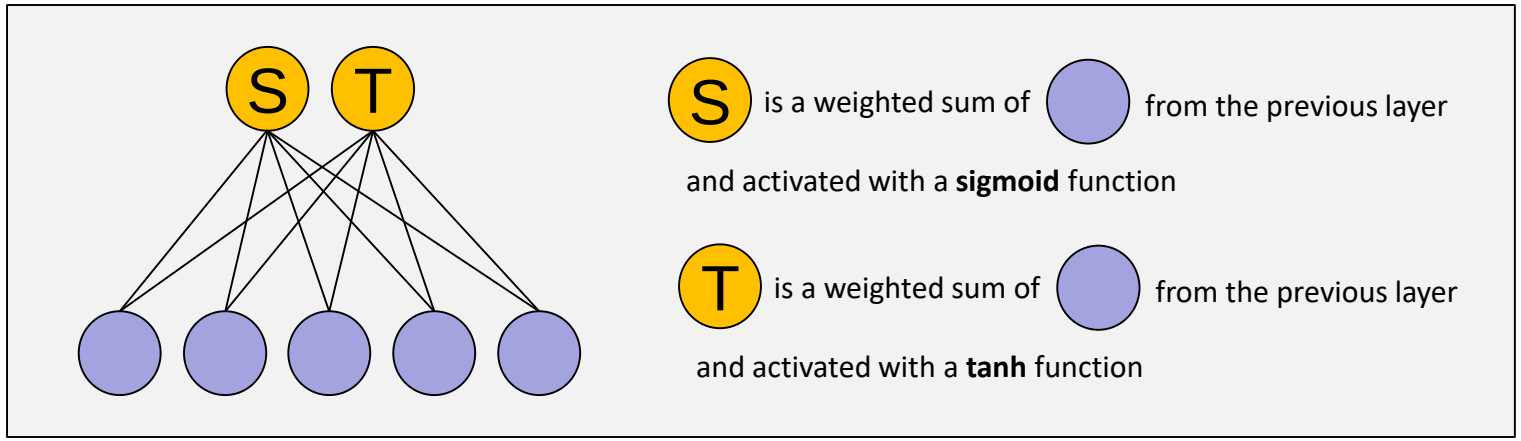
More weights to learn

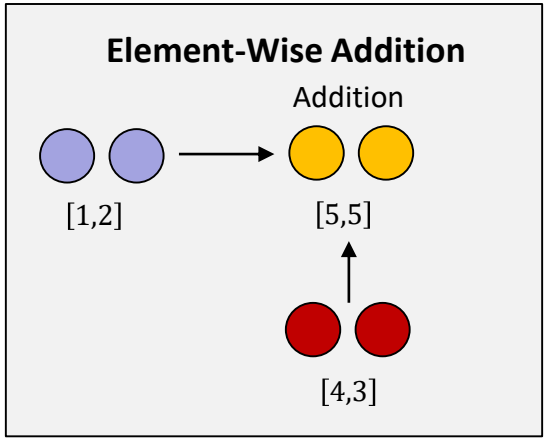
As in the *simple RNN*
We reuse the same weight matrix at every time step

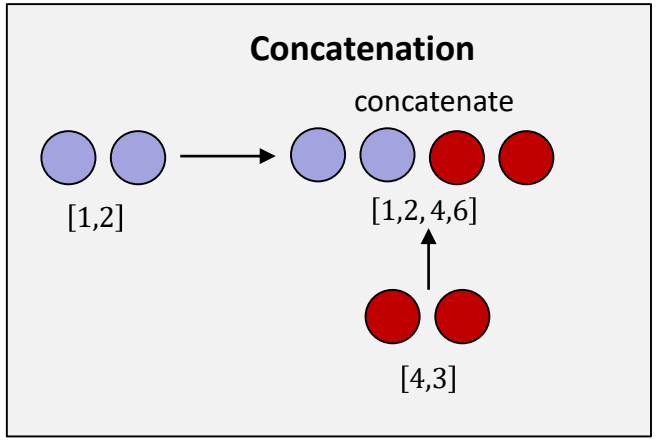
Image source: https://colah.github.io/

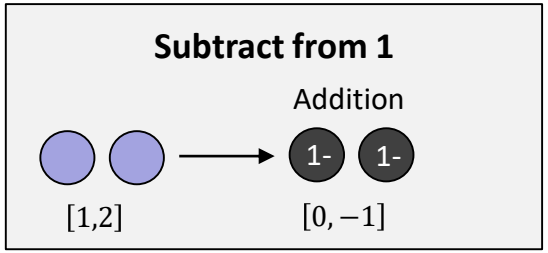
Guide to upcoming illustrations



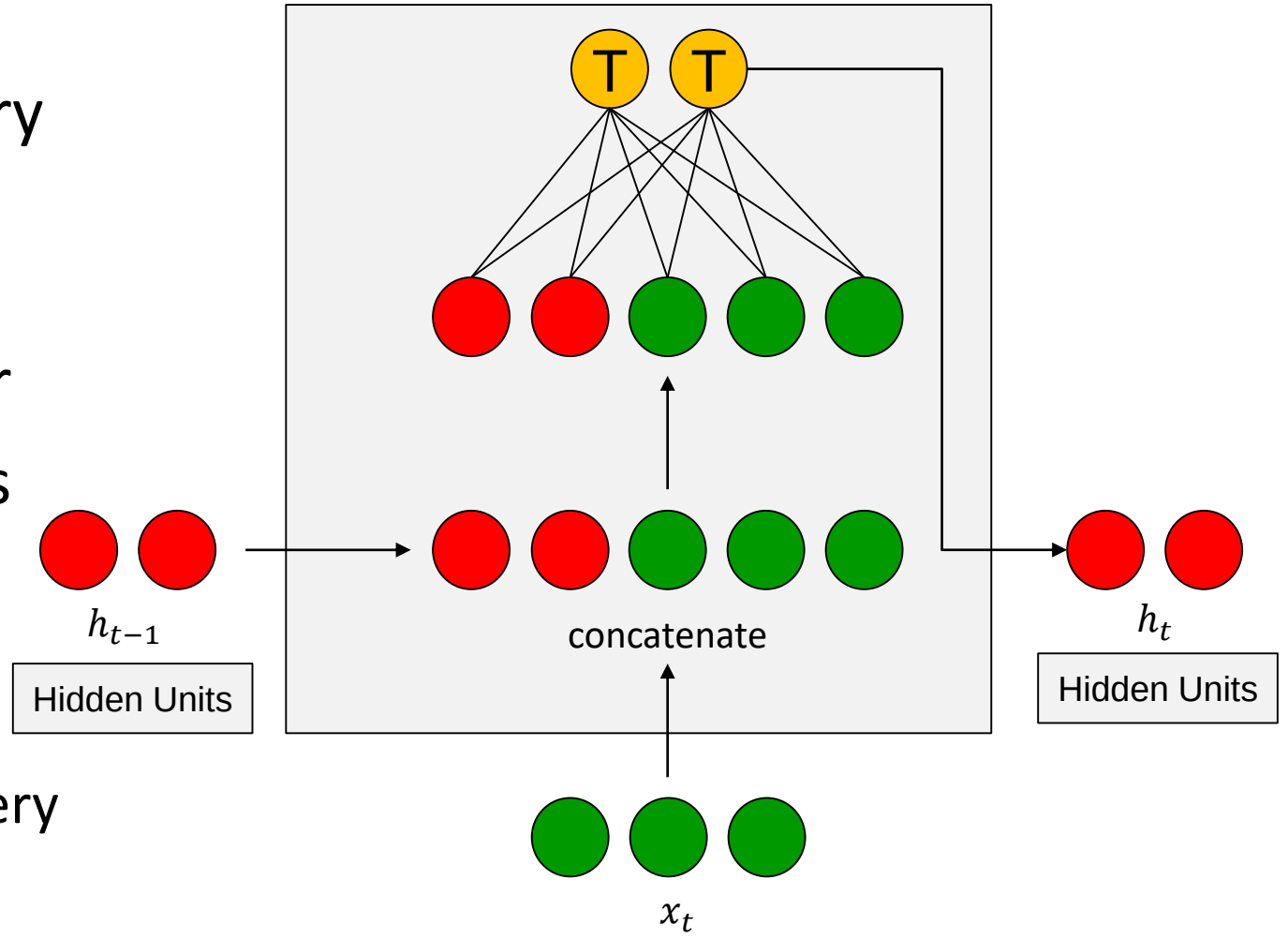




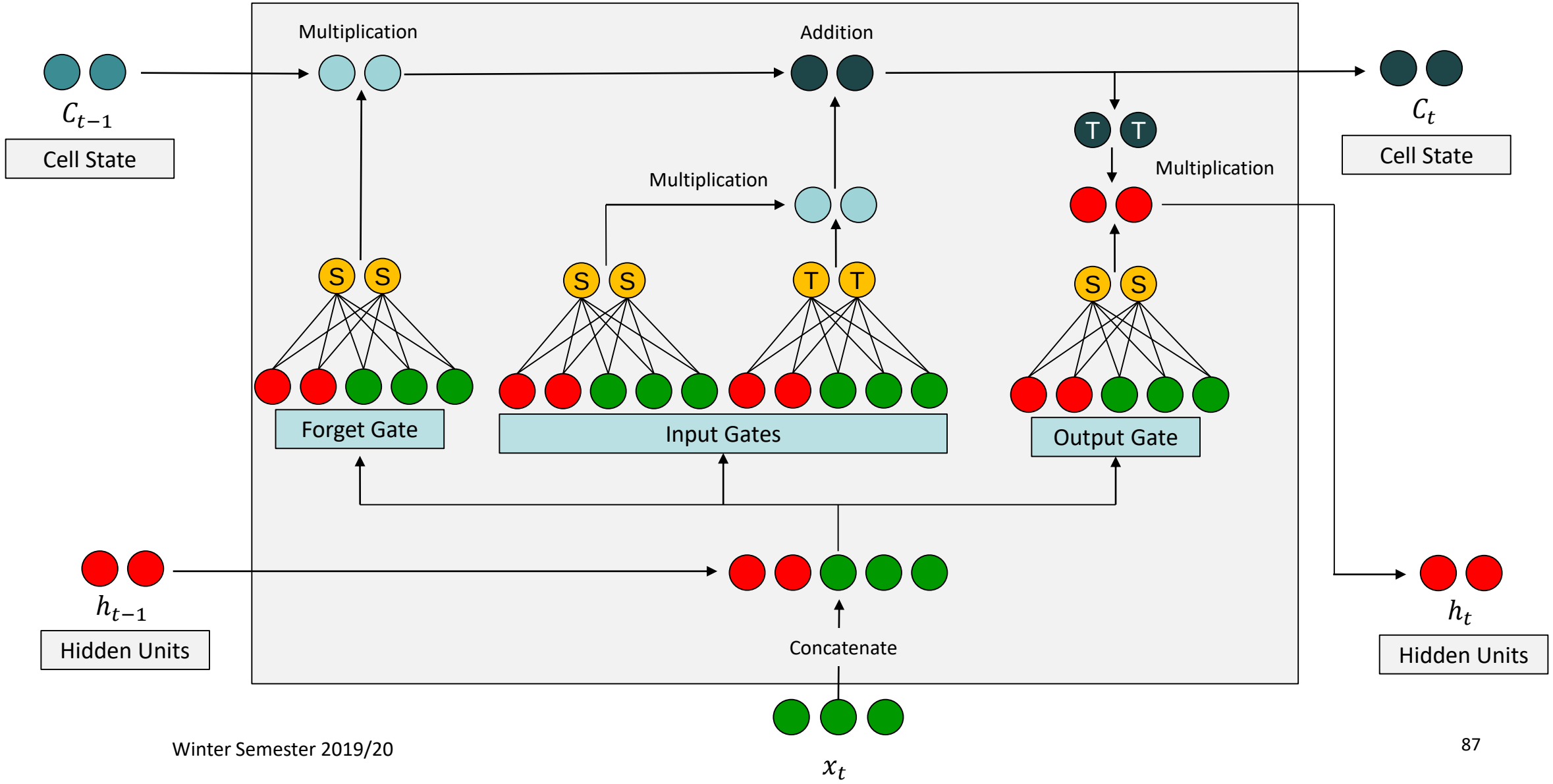




- **Vanilla RNNs**
  - Suffer from short-term memory
  - If a sequence is long enough they cannot carry information from earlier time steps to later
  - Vanishing/Exploding Gradients
    - If a gradient value becomes extremely small, it doesn't contribute too much learning
    - Large error gradients result in very large updates during training.

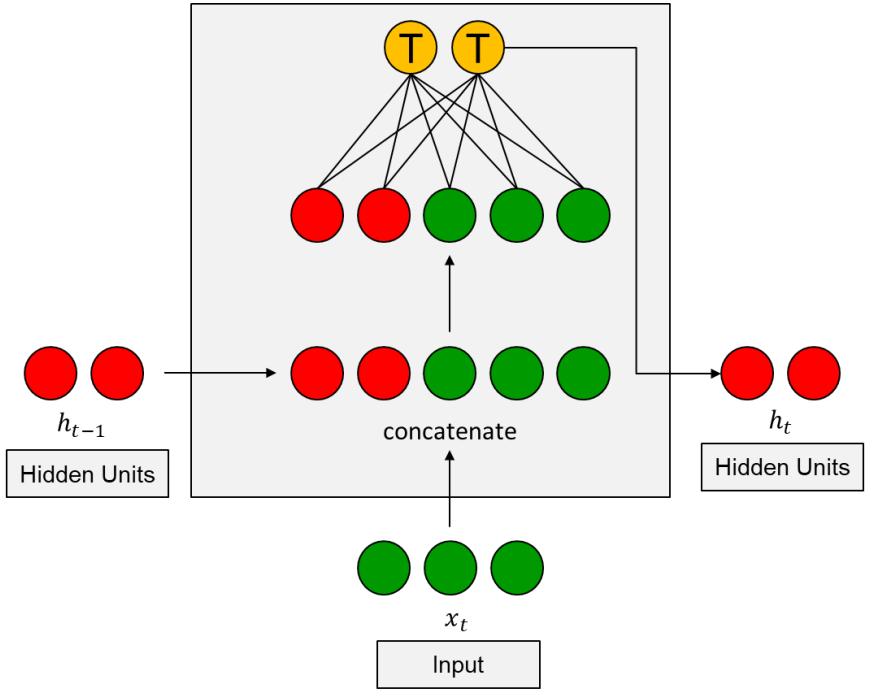


Long Short-term Memory (LSTM)



Long Short-term Memory (LSTM)

Simple RNN unit



Gated LSTM unit

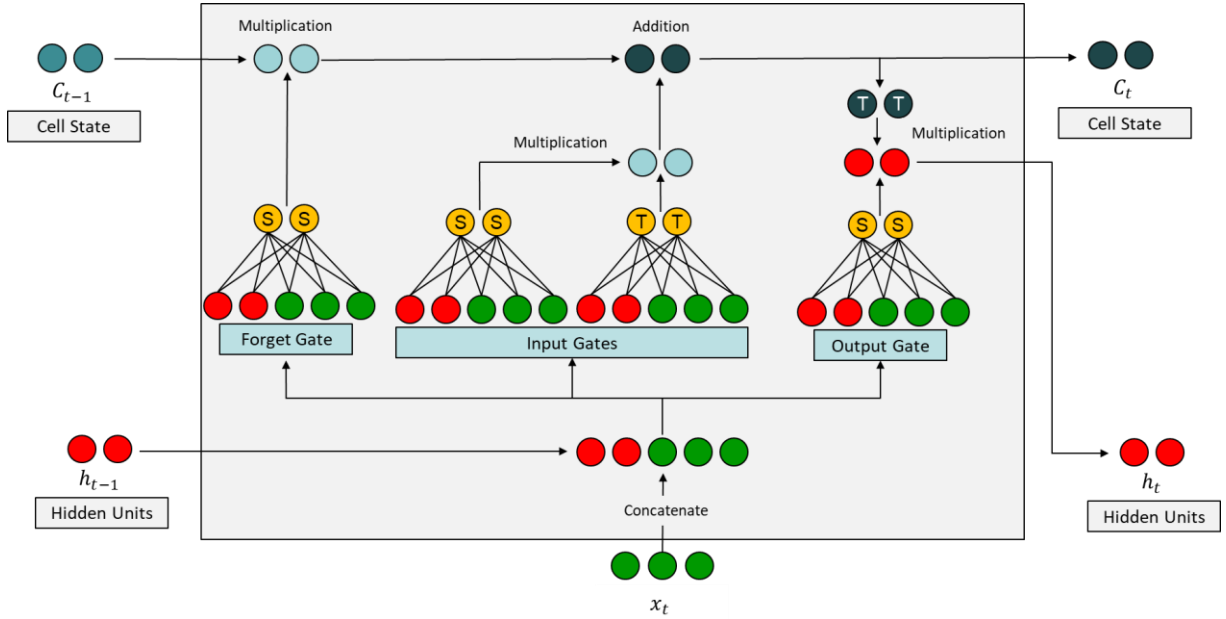
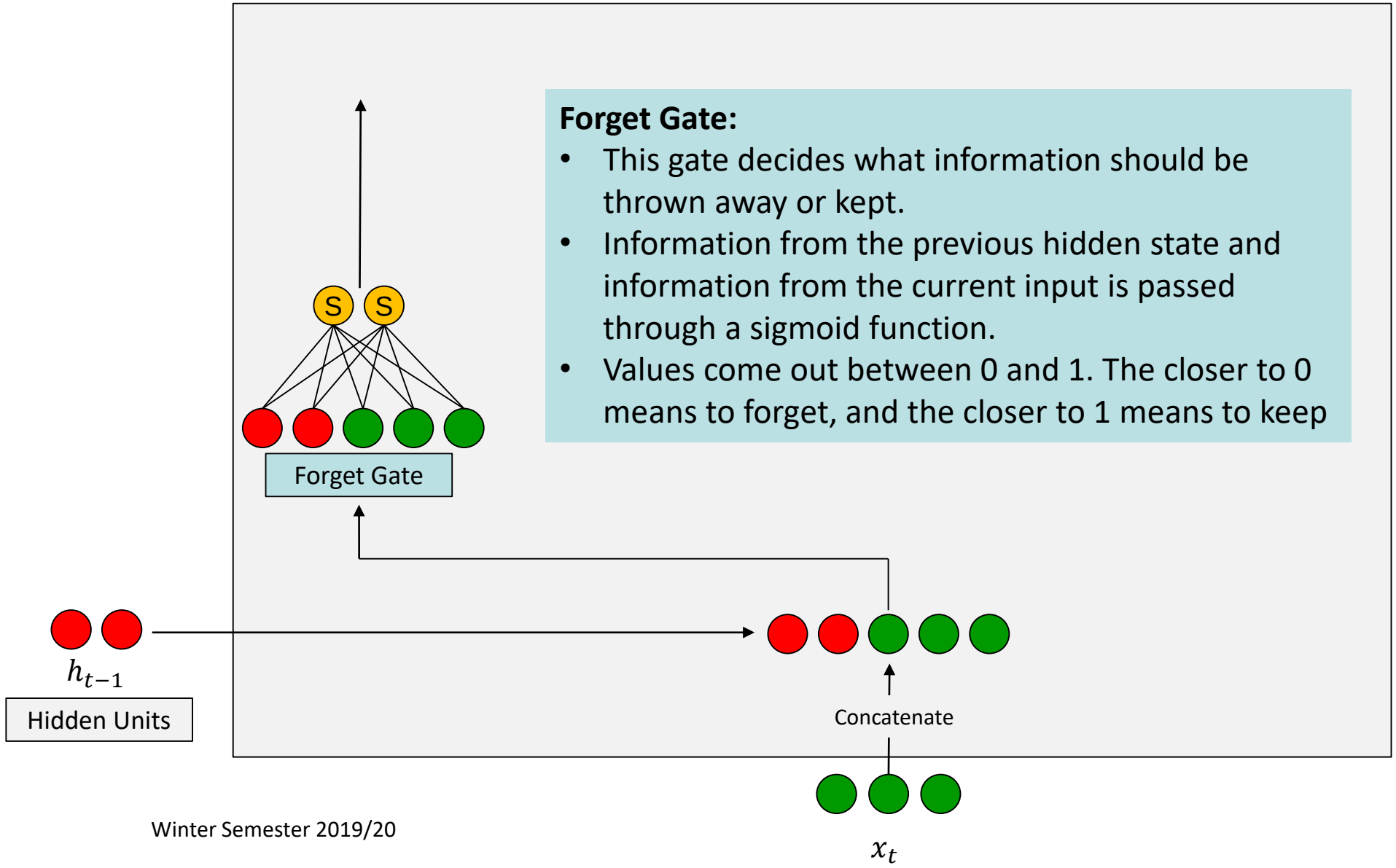


Image source: https://colah.github.io/

Long Short-term Memory (LSTM)



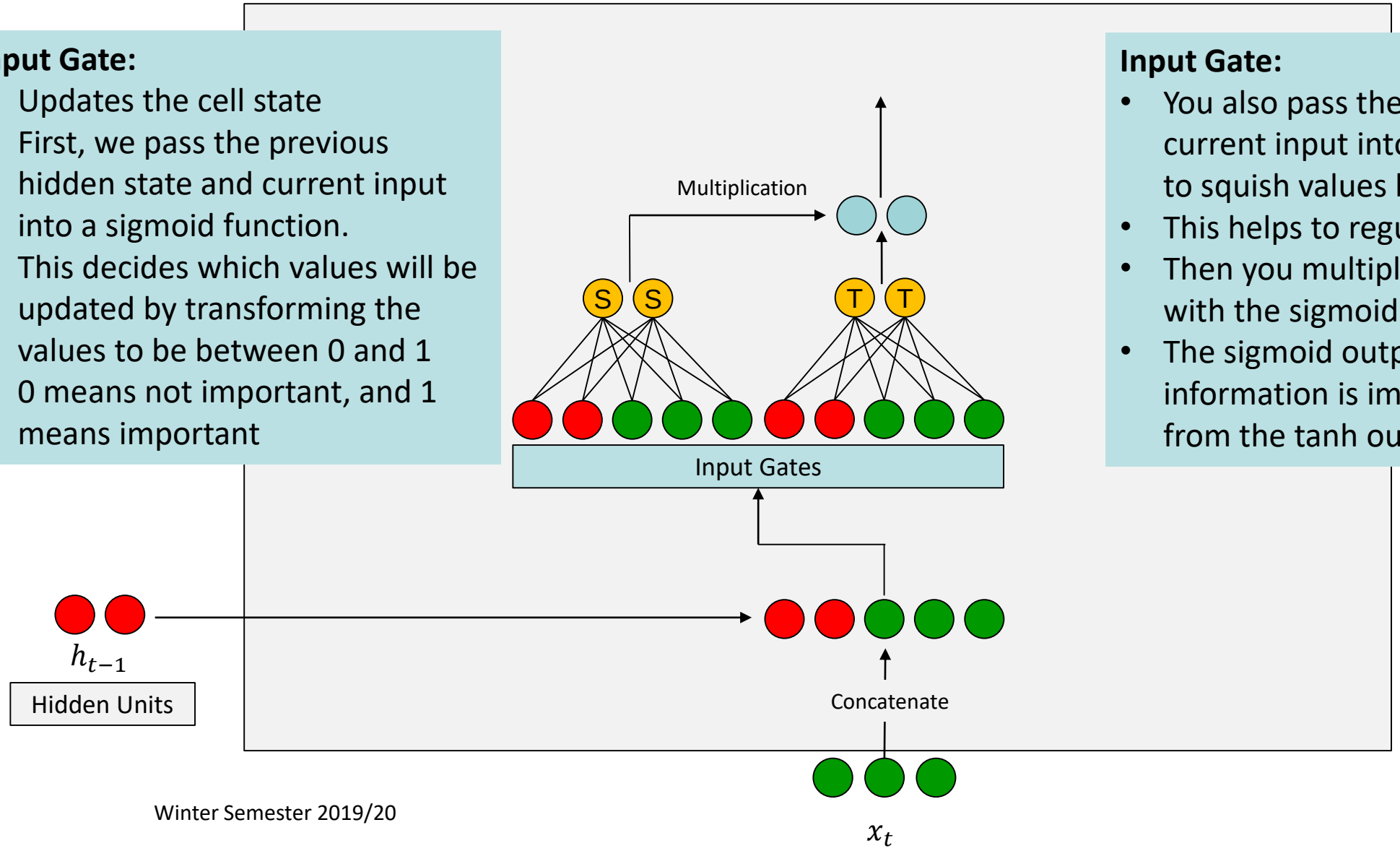
Forget Gate:

- This gate decides what information should be thrown away or kept.
- Information from the previous hidden state and information from the current input is passed through a sigmoid function.
- Values come out between 0 and 1. The closer to 0 means to forget, and the closer to 1 means to keep

Long Short-term Memory (LSTM)

Input Gate:

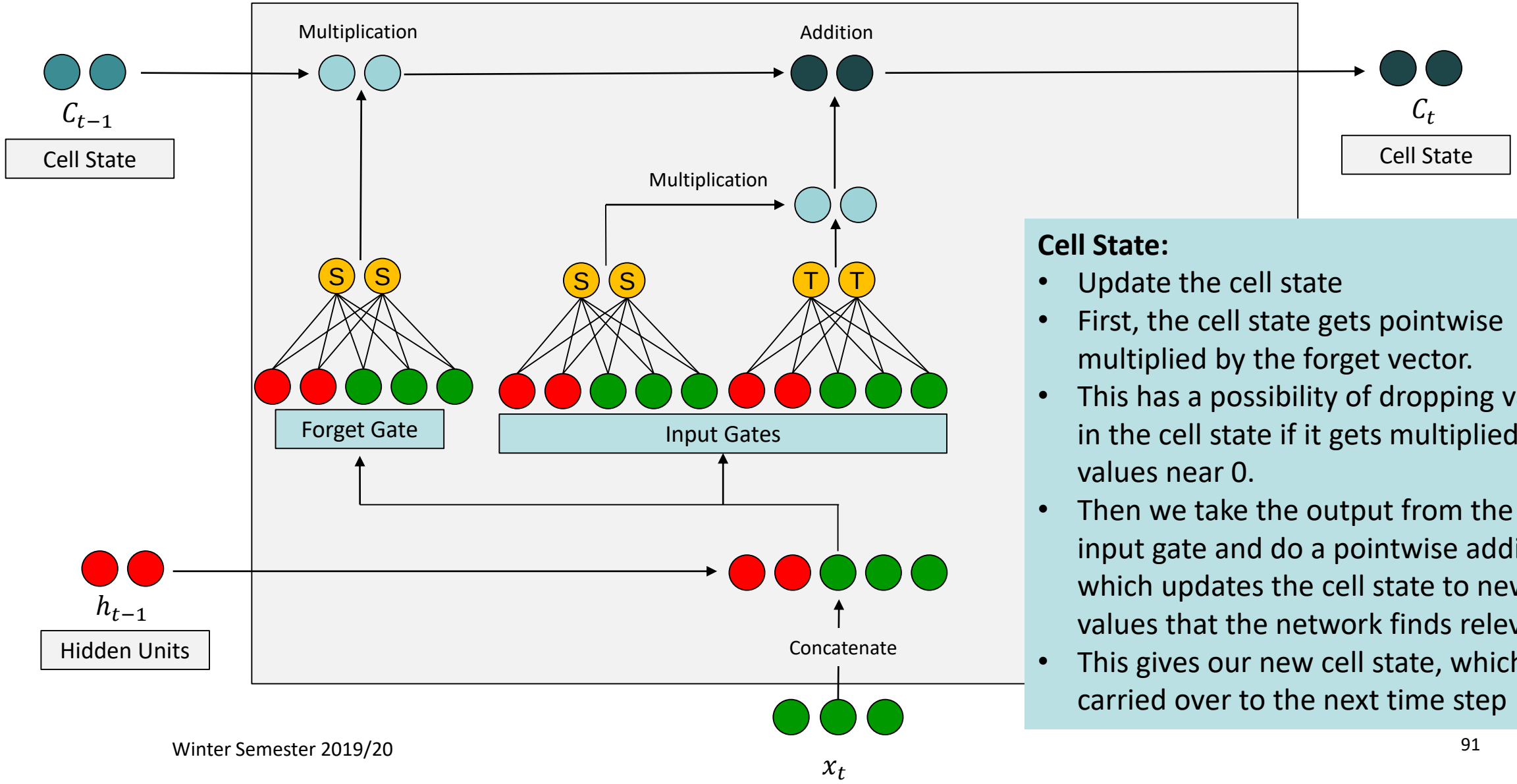
- Updates the cell state
- First, we pass the previous hidden state and current input into a sigmoid function.
- This decides which values will be updated by transforming the values to be between 0 and 1
- 0 means not important, and 1 means important



Input Gate:

- You also pass the hidden state and current input into the tanh function to squish values between -1 and 1
- This helps to regulate the network.
- Then you multiply the tanh output with the sigmoid output.
- The sigmoid output will decide which information is important to keep from the tanh output

Long Short-term Memory (LSTM)



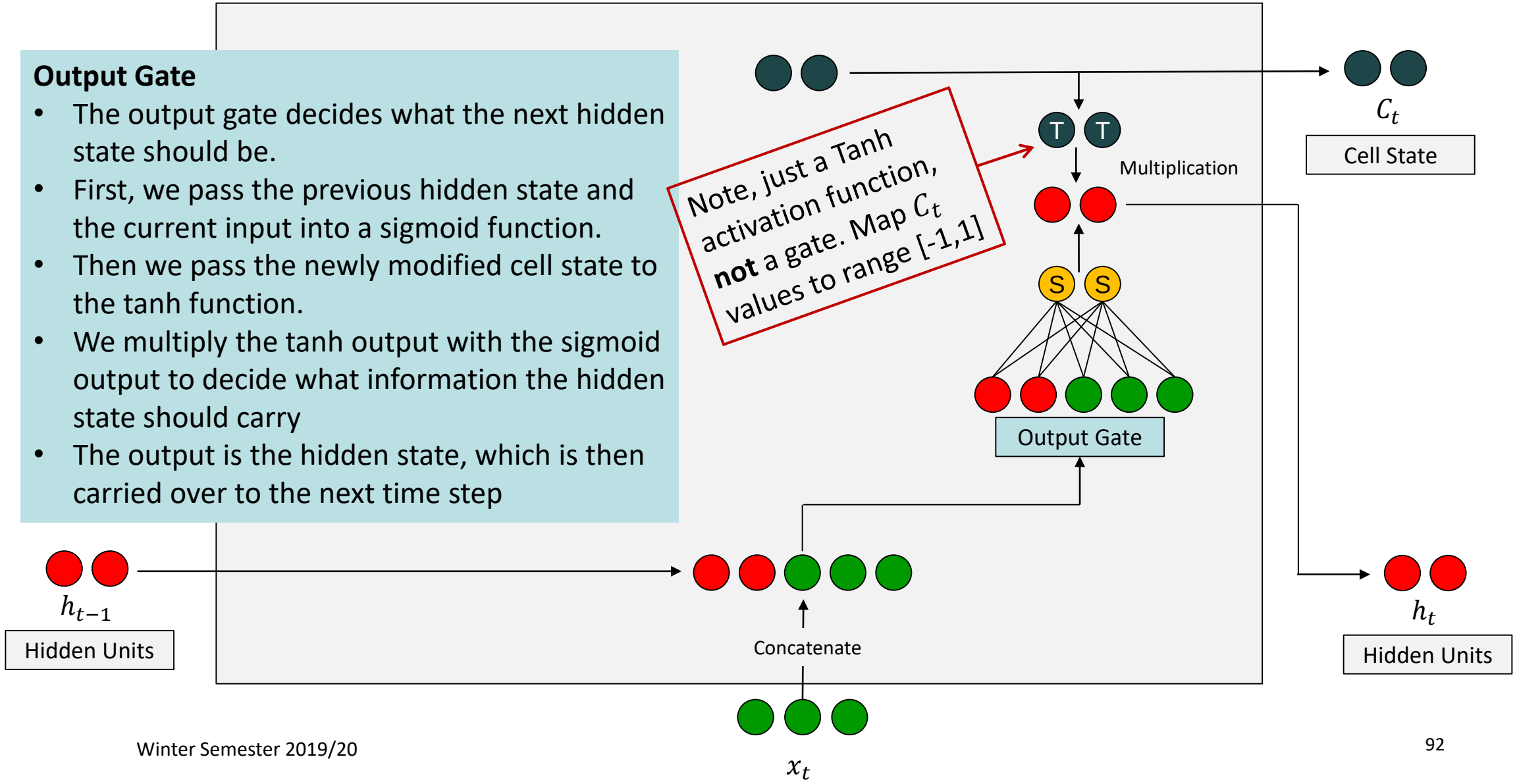
Cell State:

- Update the cell state
- First, the cell state gets pointwise multiplied by the forget vector.
- This has a possibility of dropping values in the cell state if it gets multiplied by values near 0.
- Then we take the output from the input gate and do a pointwise addition which updates the cell state to new values that the network finds relevant
- This gives our new cell state, which is carried over to the next time step

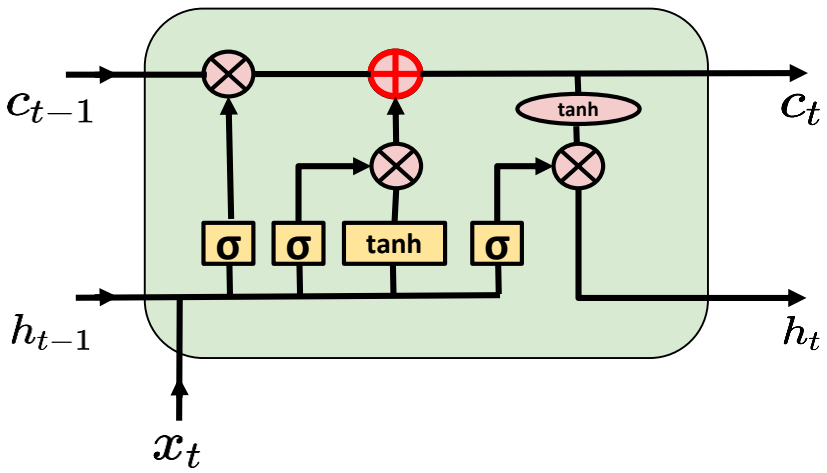
Long Short-term Memory (LSTM)

Output Gate

- The output gate decides what the next hidden state should be.
- First, we pass the previous hidden state and the current input into a sigmoid function.
- Then we pass the newly modified cell state to the tanh function.
- We multiply the tanh output with the sigmoid output to decide what information the hidden state should carry
- The output is the hidden state, which is then carried over to the next time step



Long Short-term Memory (LSTM)



LSTM can deal with the gradient vanishing:

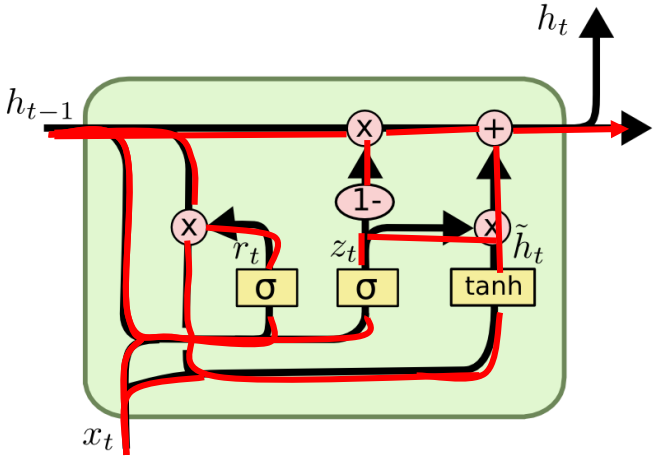
- Previous cell state and input are added together
- The influence never disappears unless the forget gate is completely closed
- Instead of computing new state as a matrix product with the old one, it rather computes the additive interactions
- Gradient flow is therefore improved
- Store/retrieve information over longer time periods

Image source: https://colah.github.io/

- **The Gated Recurrent Unit (GRU)**
  - A simplified variant of the LSTM unit
  - LSTM has 3 gates: input, output and forget
  - The GRU we only have two gates an update and a reset
    - Update gate: Decides how much of previous memory to keep
    - Reset gate: How to combine new input with previous value
  - Unlike the LSTM, the GRU has no persistent cell state

Gated Recurrent Unit

- Single gating unit simultaneously controls the forgetting factor and the decision to update the state unit



$$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t] + b_z\right)$$

$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t] + b_r\right)$$

$$g_t = tanh\left(W_g \cdot [r_t \odot h_{t-1}, x_t] + b_g\right)$$

$$h_t = z_t \odot h_{t-1} + (1 - z_t) \odot g_t$$

Image source: https://colah.github.io/

- **Unrolled GRU network**
  - There are three interacting networks: reset gate, update gate, output gate

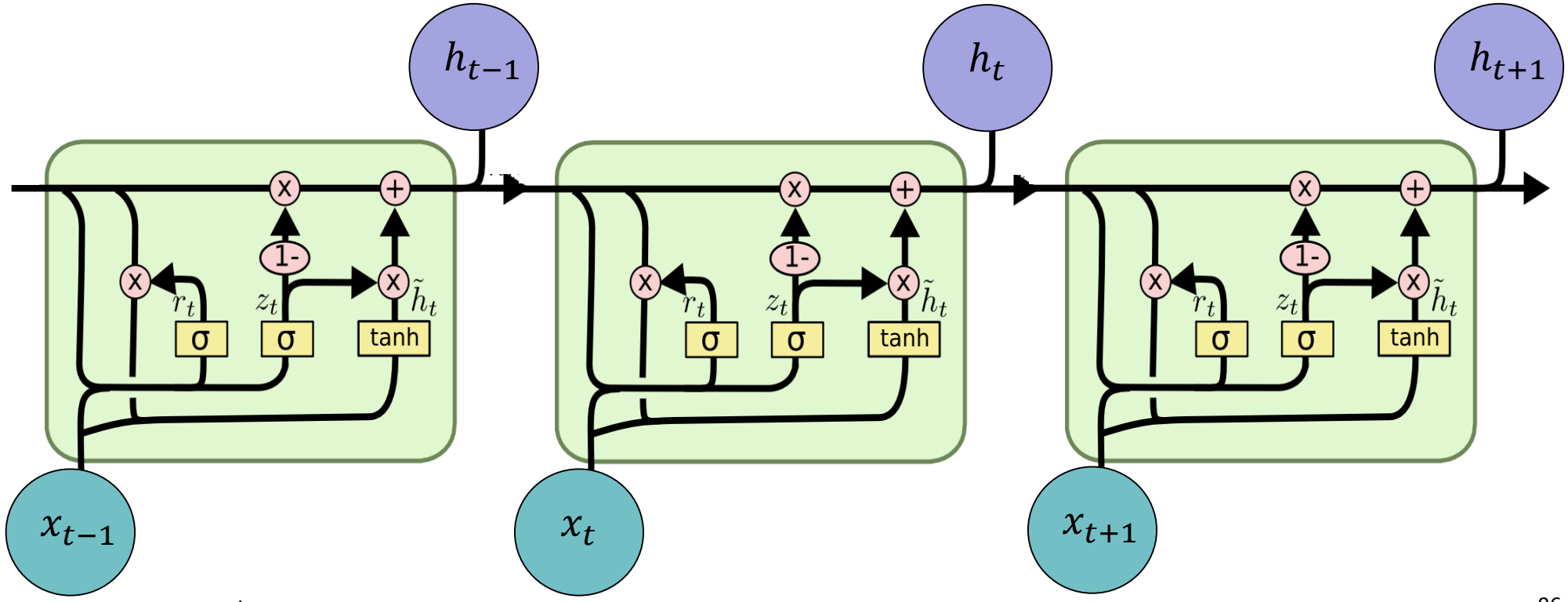
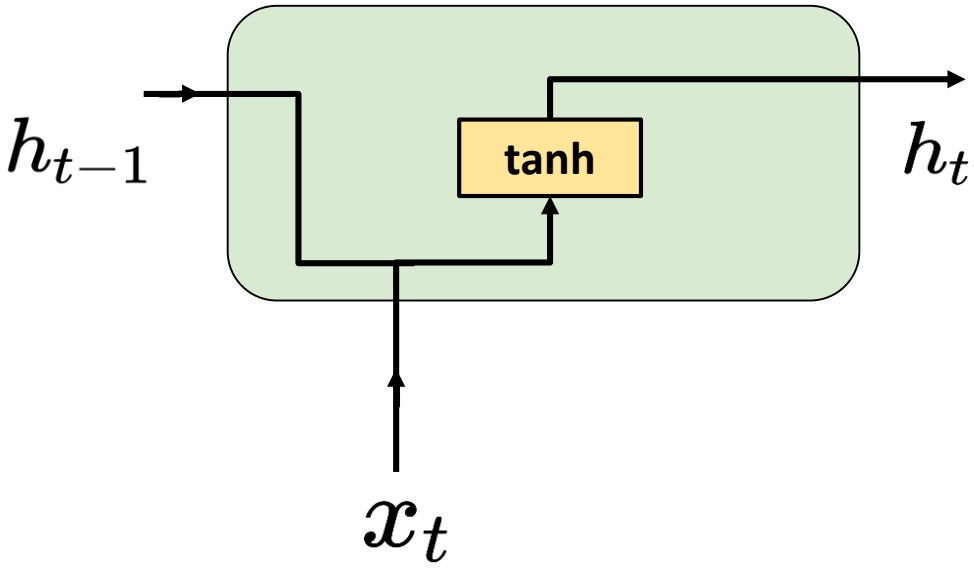


Image source: https://colah.github.io/

Gated Recurrent Unit (GRU)

Simple RNN unit



Gated Recurrent Unit

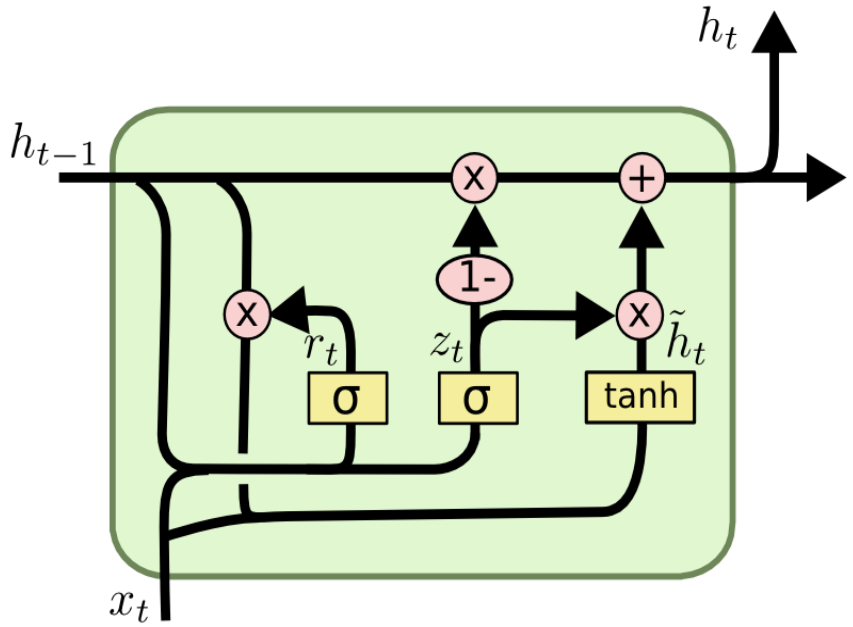
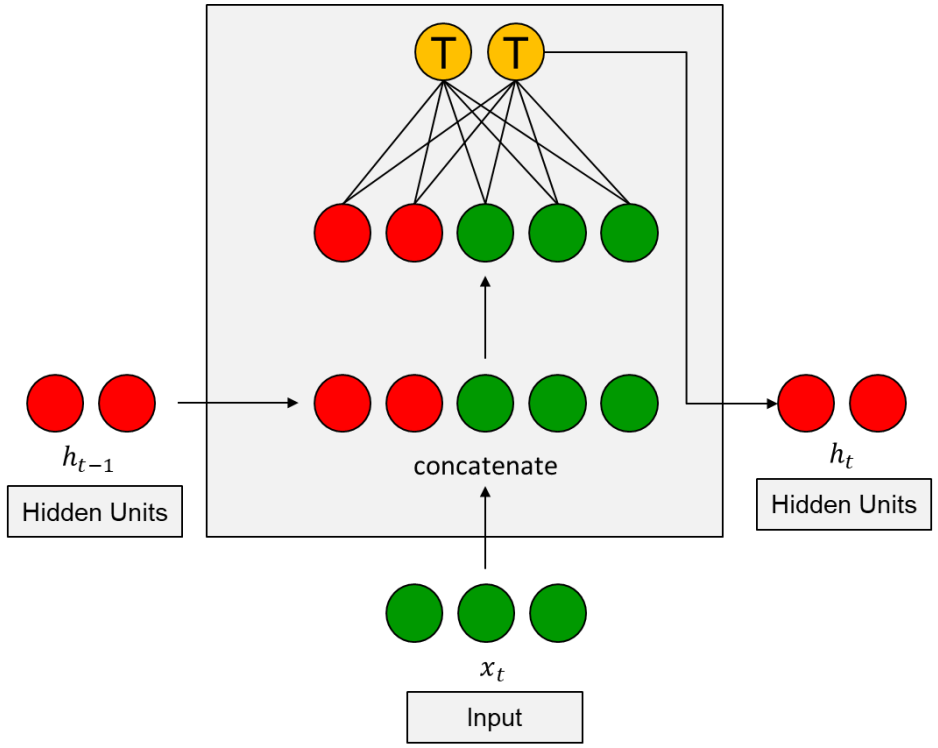


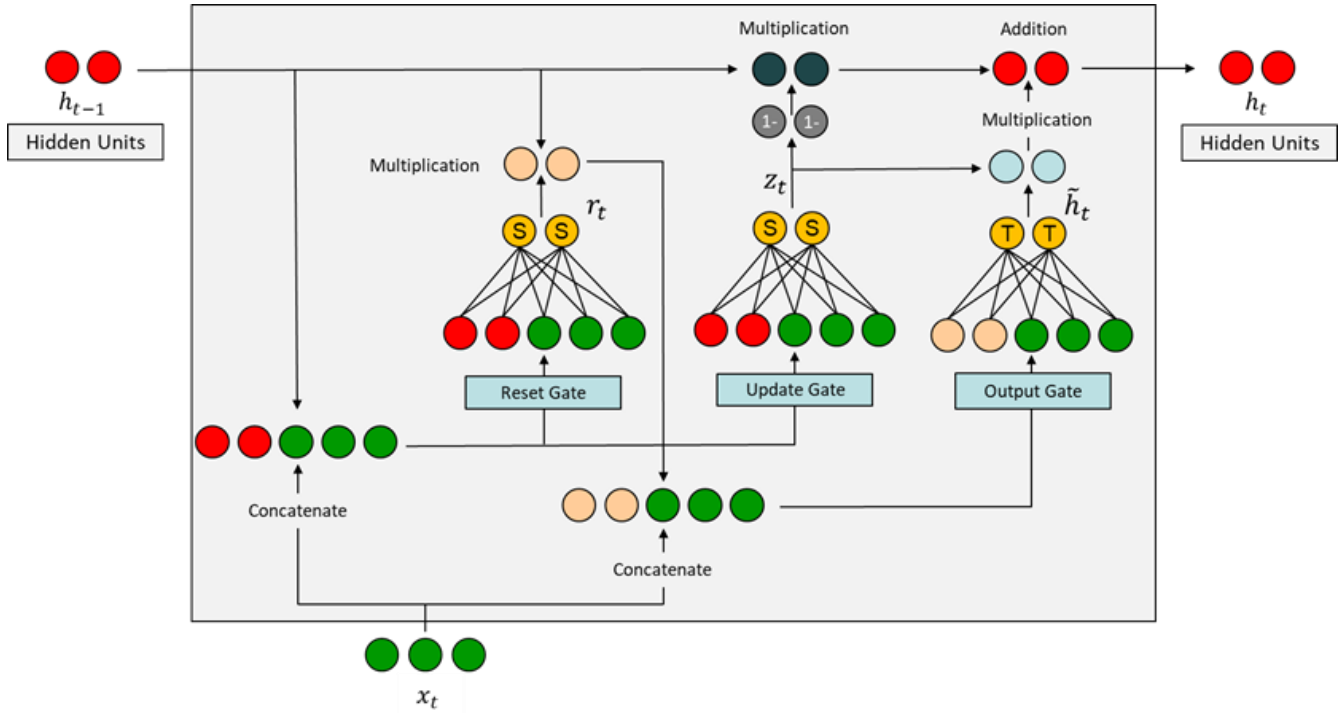
Image source: https://colah.github.io/

Gated Recurrent Unit (GRU)

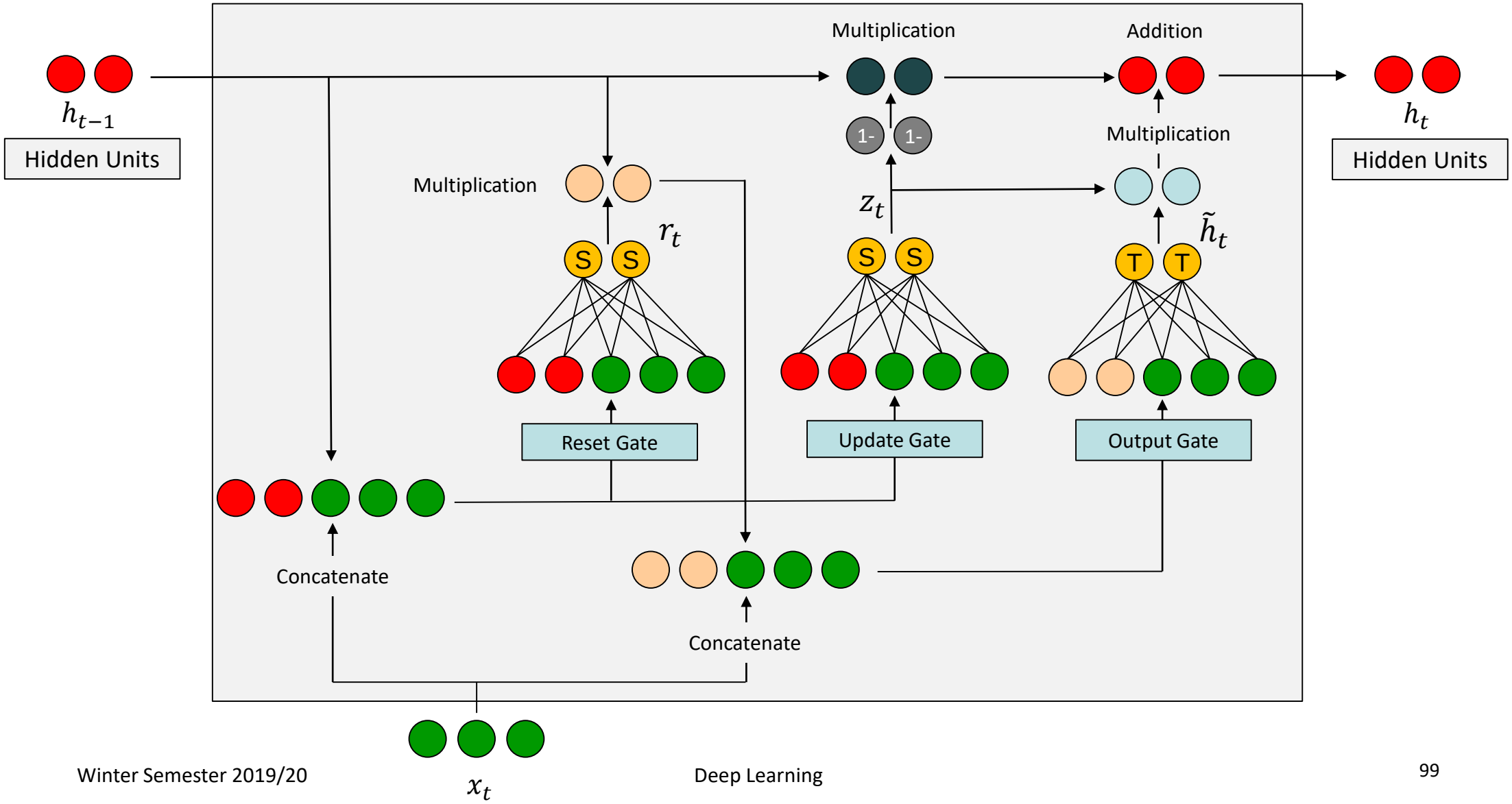
Simple RNN unit



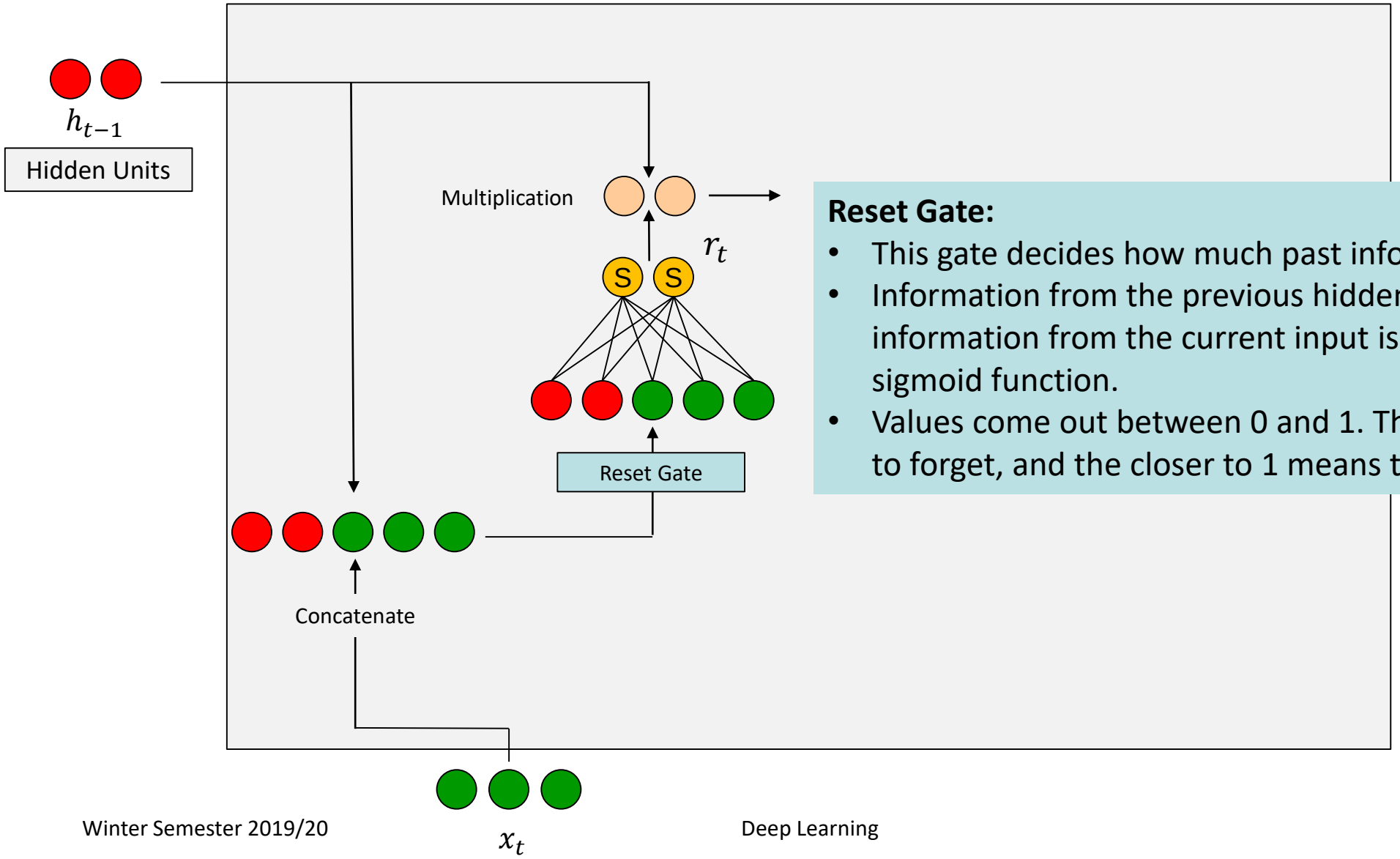
Gated Recurrent Unit



Gated Reccurent Unit



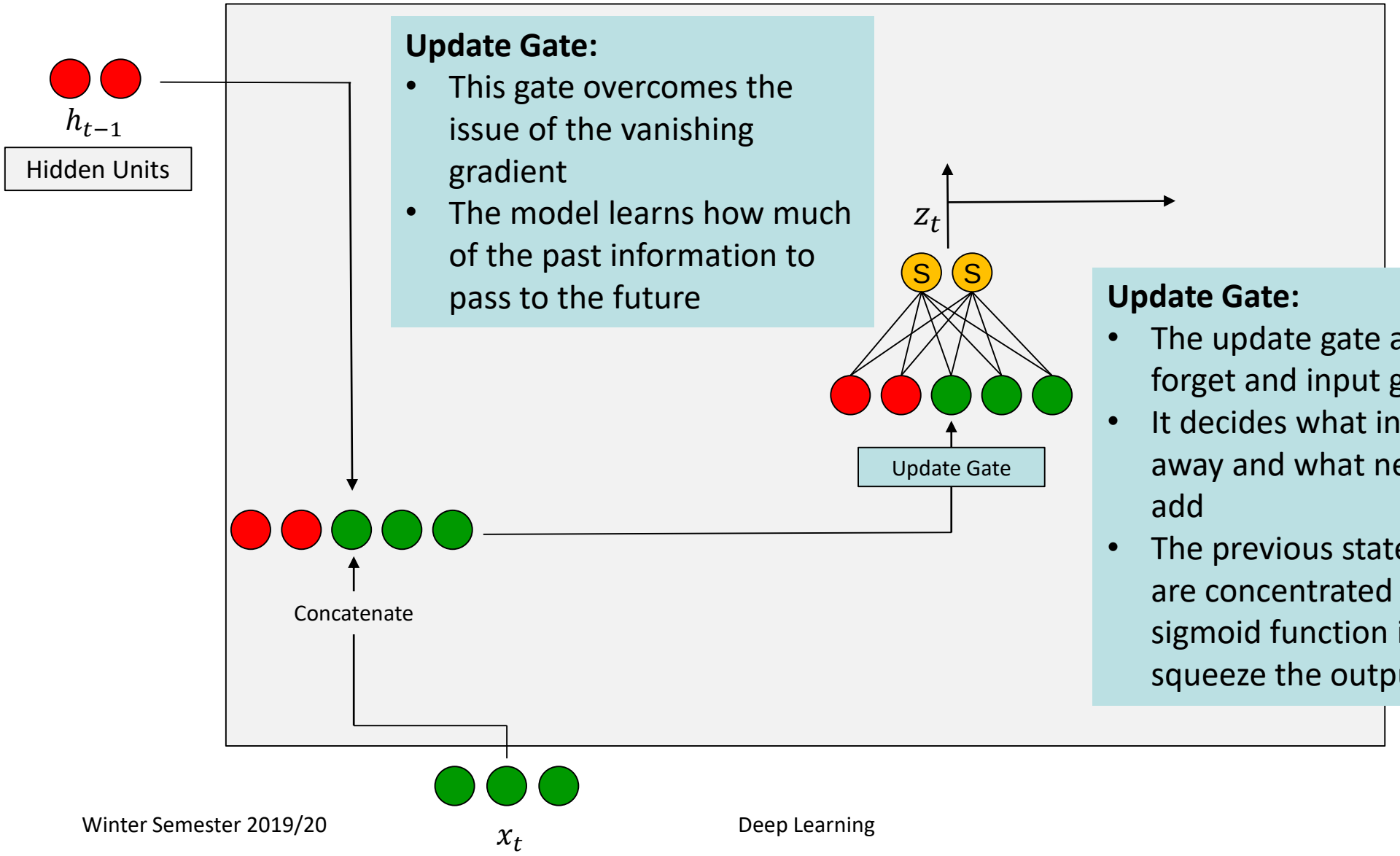
Gated Reccurent Unit



Reset Gate:

- This gate decides how much past information to forget
- Information from the previous hidden state and information from the current input is passed through a sigmoid function.
- Values come out between 0 and 1. The closer to 0 means to forget, and the closer to 1 means to keep

Gated Reccurent Unit



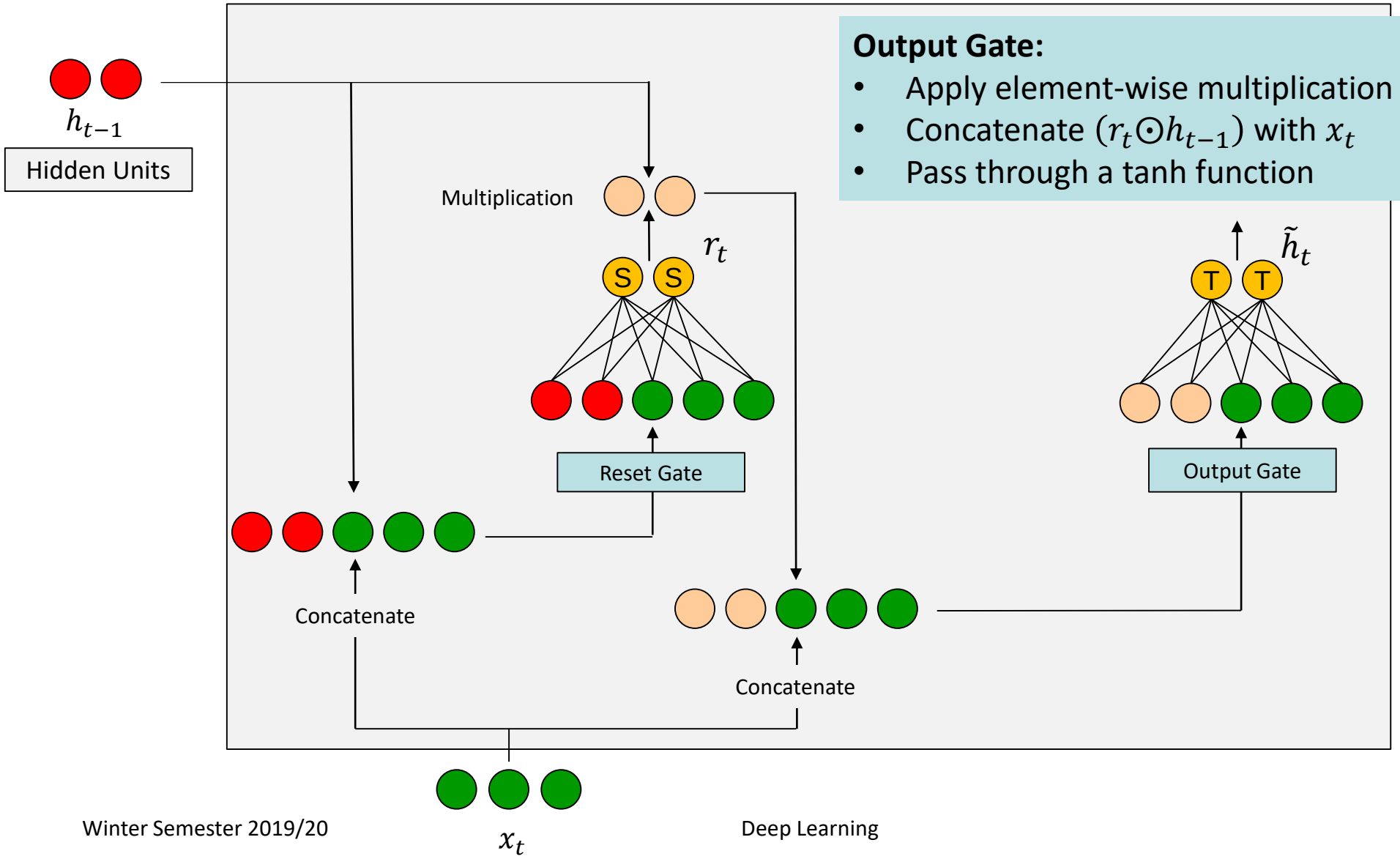
Update Gate:

- This gate overcomes the issue of the vanishing gradient
- The model learns how much of the past information to pass to the future

Update Gate:

- The update gate acts similar to the forget and input gate of an LSTM.
- It decides what information to throw away and what new information to add
- The previous state and current input are concentrated together and a sigmoid function is applies to squeeze the output between 0 and 1

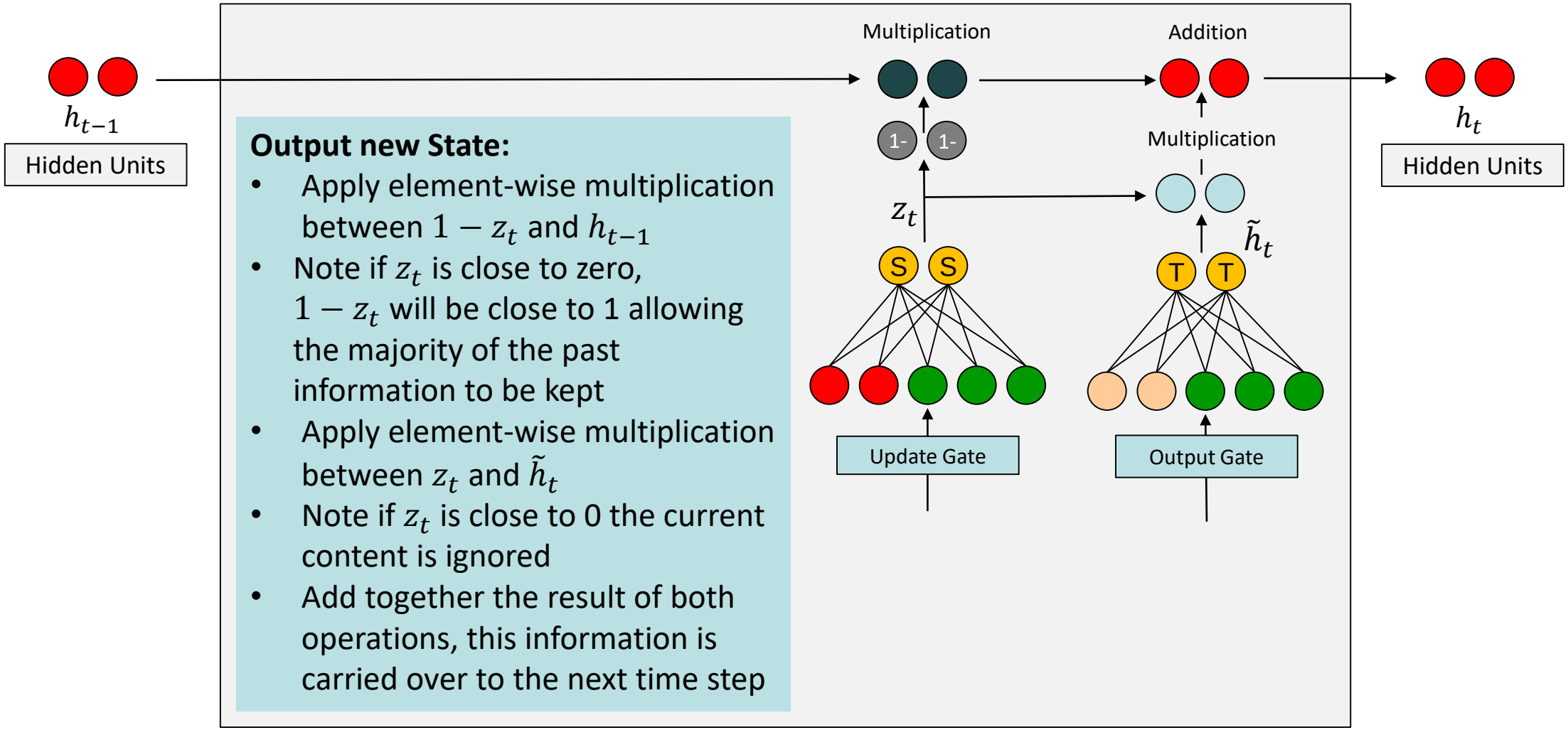
Gated Reccurent Unit



Output Gate:

- Apply element-wise multiplication between r_t and h_{t-1}
- Concatenate $(r_t \odot h_{t-1})$ with x_t
- Pass through a tanh function

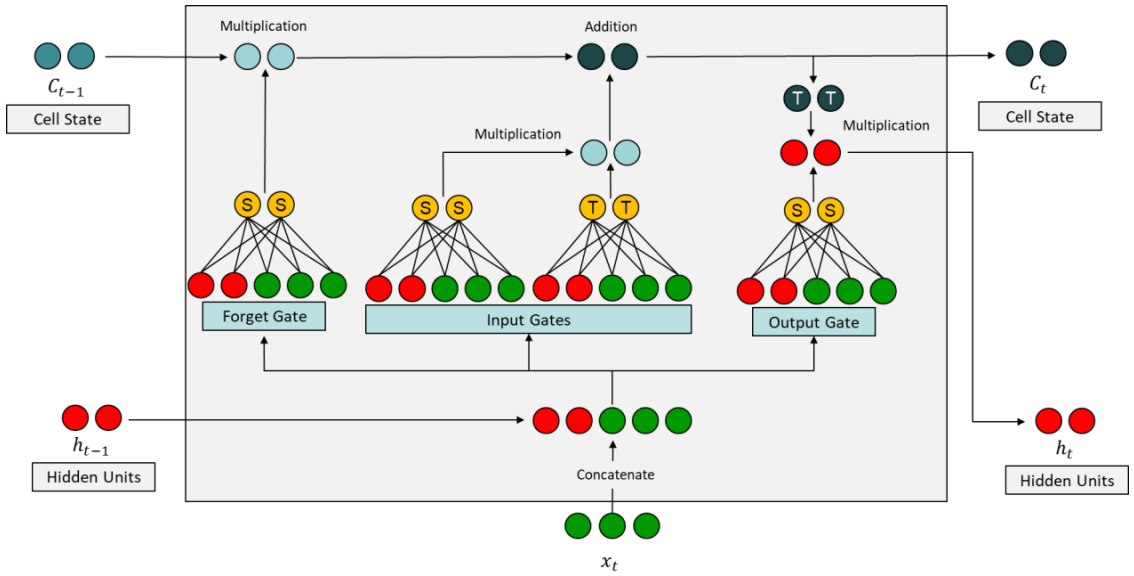
Gated Reccurent Unit

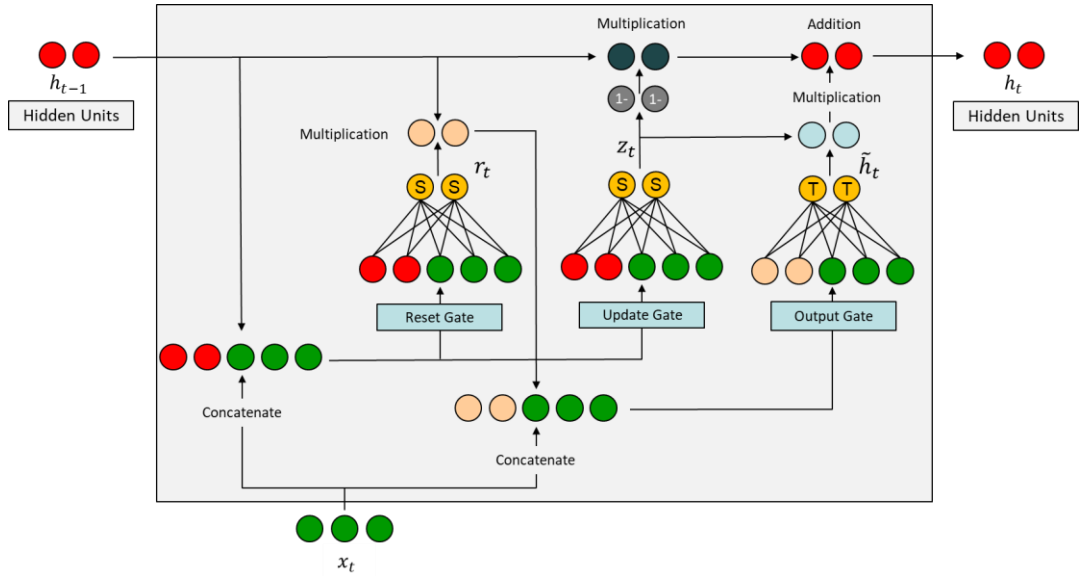


Output new State:

- Apply element-wise multiplication between $1 - z_t$ and h_{t-1}
- Note if z_t is close to zero, $1 - z_t$ will be close to 1 allowing the majority of the past information to be kept
- Apply element-wise multiplication between z_t and $\tilde{h}_t$
- Note if z_t is close to 0 the current content is ignored
- Add together the result of both operations, this information is carried over to the next time step

- **Comparing GRU and LSTM**
  - Both GRU and LSTM are better than a simple RNN
  - GRU is faster to train with fewer parameters
  - GRU can perform better than LSTM on certain tasks





RNN - Summary

- RNN's incorporation of feedback into the network structure to allow for processing sequence data
- Vanilla RNNs suffers from short-term memory
  - Vanishing Gradient issues
- LSTM's and GRU's were created as a method to mitigate short-term memory using gating mechanisms
  - Gates are just neural networks that regulate the flow of information flowing through the sequence chain

End-to-End Network

A network the accepts ‘raw’ input and produces output.
The learning that optimizes the network weights by considering the inputs and outputs directly

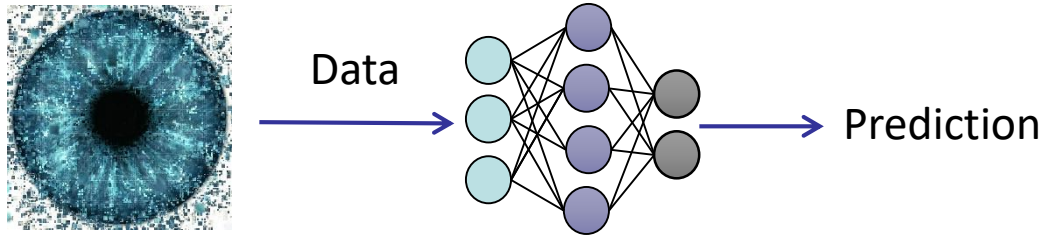
Universität Augsburg University

Images Sourced: https://pixabay.com

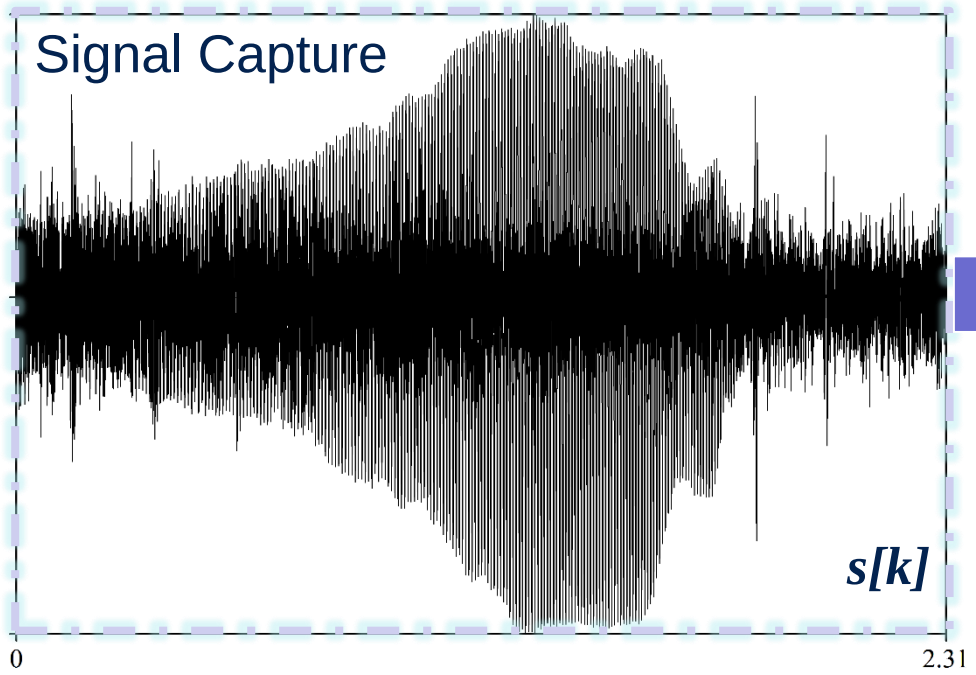
End to End Processing

- Conventional Machine learning relies of features to reduce data complexity & make patterns more visible

- Using CNNs and RNNs high-level features can be learnt from data in an incremental manner.
  - Eliminates the need of domain expertise and feature extraction.



End-2-End Learning



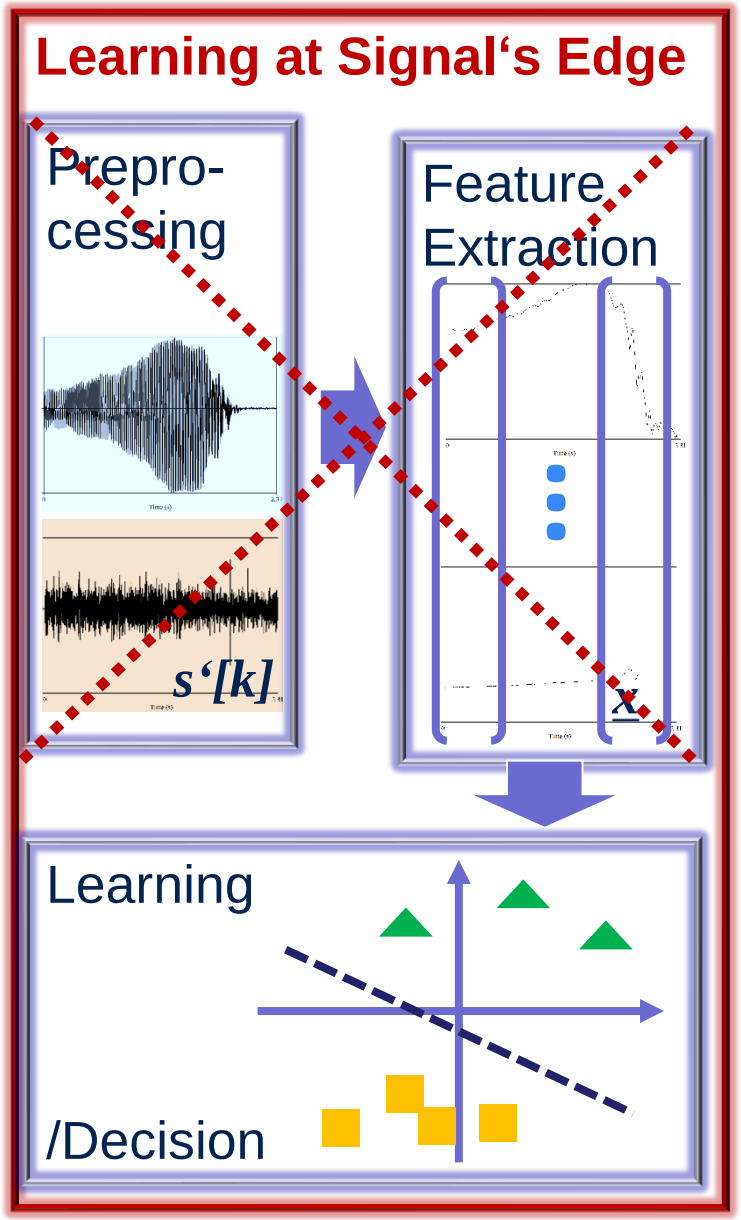
"Big Data vs Little Labels"

Efficient Weakly Supervised

<label>

(e.g.. "burst")

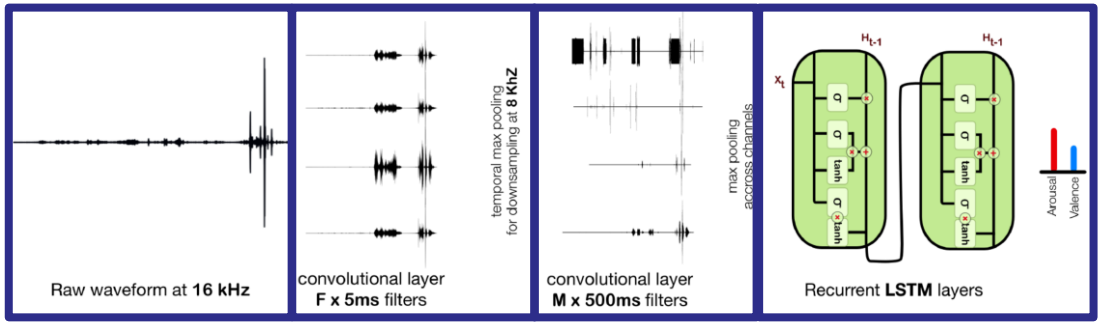
y

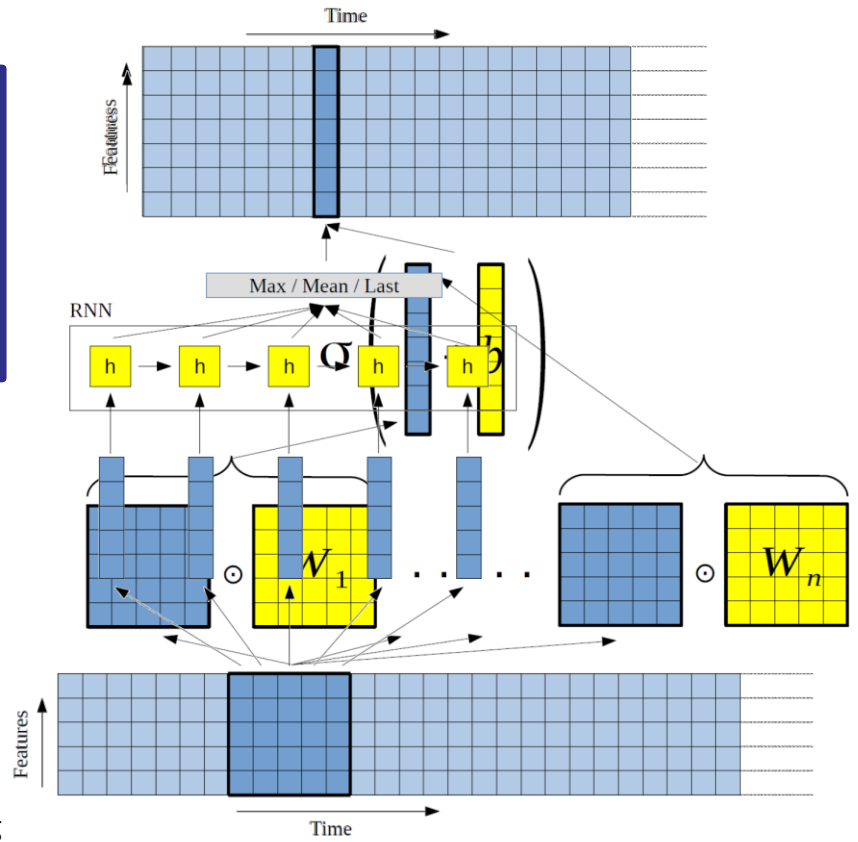


Pattern Recognition 2.0?

- CNN + LSTM → CLSTM ?

Arousal	CCC
Baseline	.366
e2e	.686

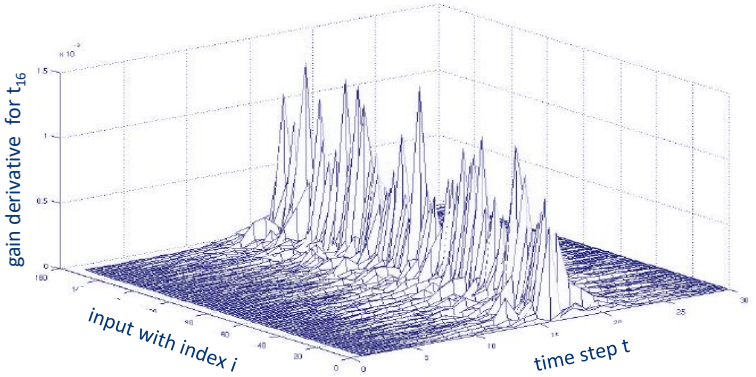


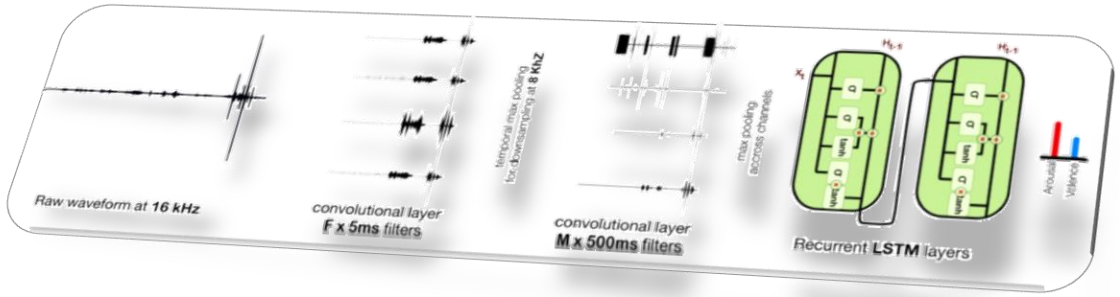


G. Trigeorgis, F. Ringeval, R. Bruckner, E. Marchi, M. Nicolaou, B. Schuller, and S. Zafeiriou, "Adieu Features? End-to-End Speech Emotion Recognition using a Deep Convolutional Recurrent Network," in Proceedings 41st IEEE International Conference on Acoustics, Speech, and Signal Processing, ICASSP 2016, (Shanghai, P. R. China), pp. 5200–5204, IEEE, IEEE, March 2016.

End-to-End – a black box?

- CNN activations correlate with standard speech features





energy range (.77)

loudness (.73)

F0 mean (.71)