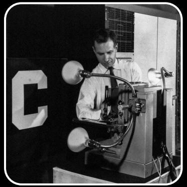
# Advanced networks

*Creative Machine Learning - Course 03*

*Pr. Philippe Esling*
esling@ircam.fr

# Brief history of AI

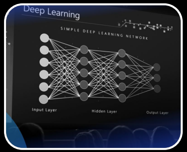
## 1943 - Neuron
First model by McCulloch & Pitts (purely theoretical)



## 1957 - Perceptron
Actual **learning machine** built by Frank Rosenblatt
Learns character recognition analogically

## 1986 - Backpropagation
First to learn neural networks efficiently (*G. Hinton*)

## This lesson 1989 - *Convolutional NN*
Mimicking the vision system in cats (*Y. LeCun*)



## *Lesson #4* 2012 - *Deep learning*


Layerwise training to have deeper architectures
Swoop all state-of-art in classification competitions

## *Lesson #6* 2015 - *Generative model*
First wave of interest in generating data
Led to current model craze (VAEs, GANs, Diffusion)

**2012 onwards Deep learning era**

# Convolutional Neural Networks (CNN)

## Major concept in modern research

- Specialized type of neural network for processing **spatial** and **temporal** data
- Key idea is to use *convolutional layers* to learn relevant features

**Principle**

Convolutional layers consist of a set of learnable filters (called kernels)
- Kernels are convolved with the input to produce a set of activations.
- Filters are akin to implementing *shared weights*
- Provides spatial invariance in pattern recognition

## The key ideas we will discuss

- **Convolutions** allowing to process temporal and spatial features
- **Pooling** as a way to reduce output dimensionality
- **Invariance** vs **equivariance** in the network

# Convolution

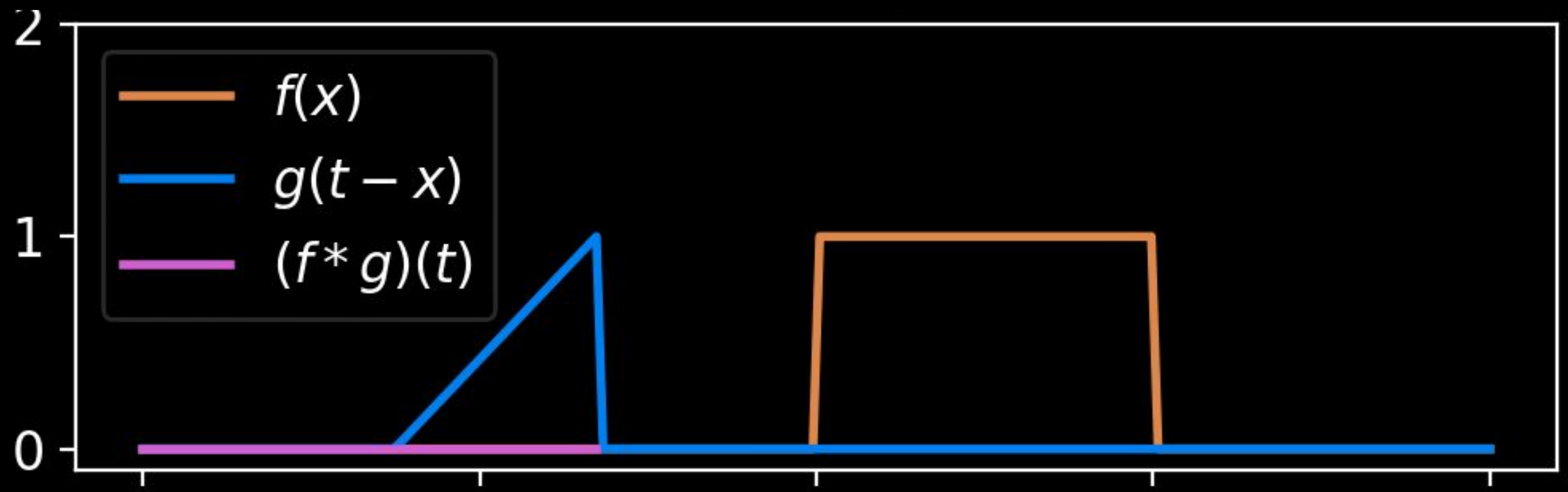
## The convolution operator

Series of local products between a **kernel** $\mathbf{k} \in \mathbb{R}^M$ and **neighborhoods** of input $\mathbf{x} \in \mathbb{R}^n$

$$(\mathbf{x} \star \mathbf{k})[n] = \sum_{m=-M}^{M} \mathbf{x}[n-m]\mathbf{k}[m].$$

## Understanding convolution

- Kernel is *sliding* over a wider vector
- Can be seen as a filter or feature detector



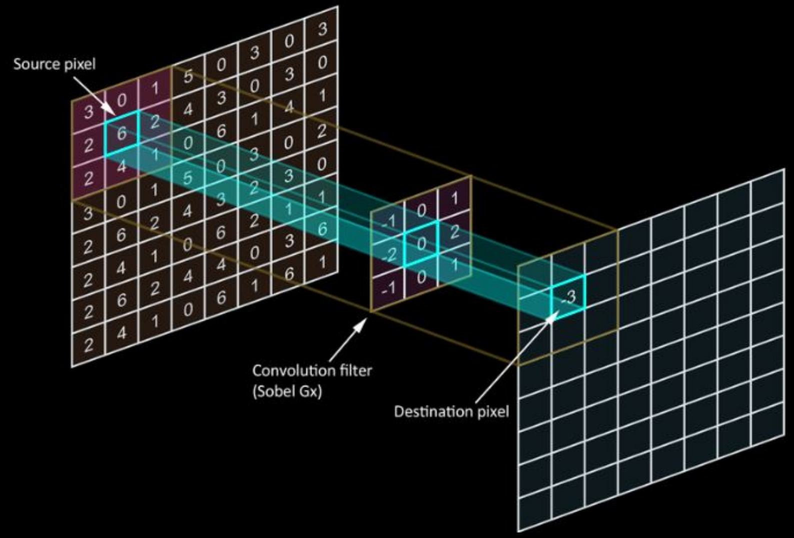
# Convolution

## How about higher-dimensional input ?

What happens if we have a matrix input (eg. image) $\mathbf{X} \in \mathbb{R}^{H \times W}$

$$(\mathbf{X} \star \mathbf{K})[n,m] = \sum_{i=-M}^{M} \sum_{j=-M}^{M} \mathbf{X}[n-i, m-j]\mathbf{K}[i,j]$$

Natural extension to 2-dimensional kernel $\mathbf{K} \in \mathbb{R}^{K \times K}$



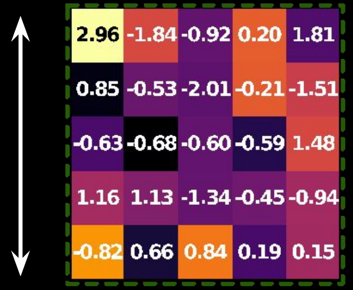

## 2-dimensional convolution

- Also interpreted as filter or feature detector
- Convolved across both dimension of input
- *Sliding* the small kernel over the matrix

**Several properties influence its behavior**

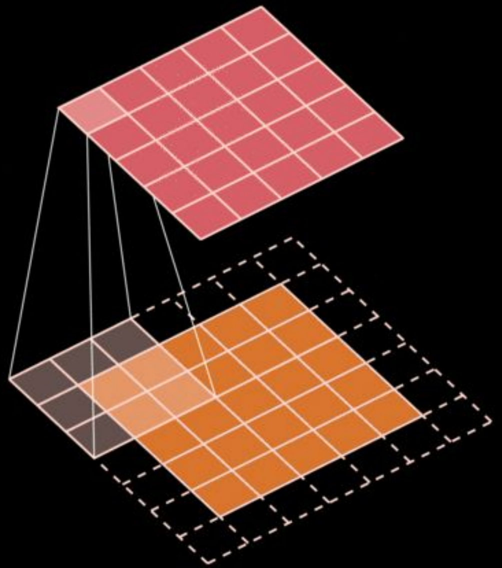
# Convolution parameters

## Major parameters that influence the behavior of convolutions

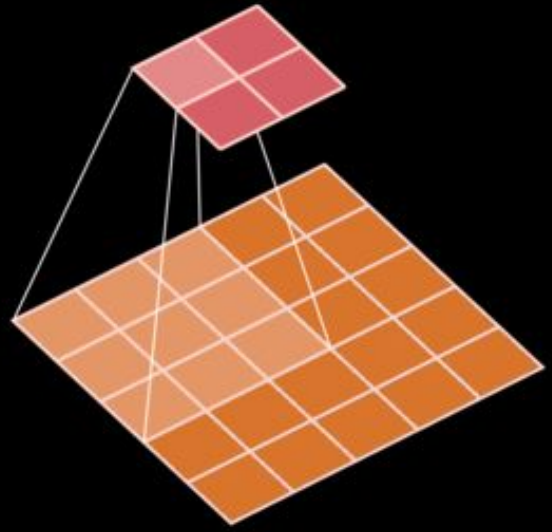



- ***Size*** of the kernel (usually square)
- ***Padding*** (adding zeros to preserve dimensionality)
- ***Stride*** (hop size of application)
- ***Dilation*** (extent of application)

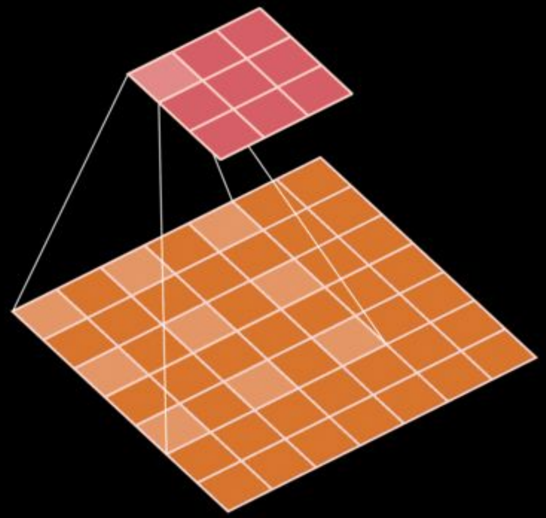
***padding***



***stride***



***dilation***



# Convolutional Neural Networks (CNN)

## Extending neural networks with convolutions

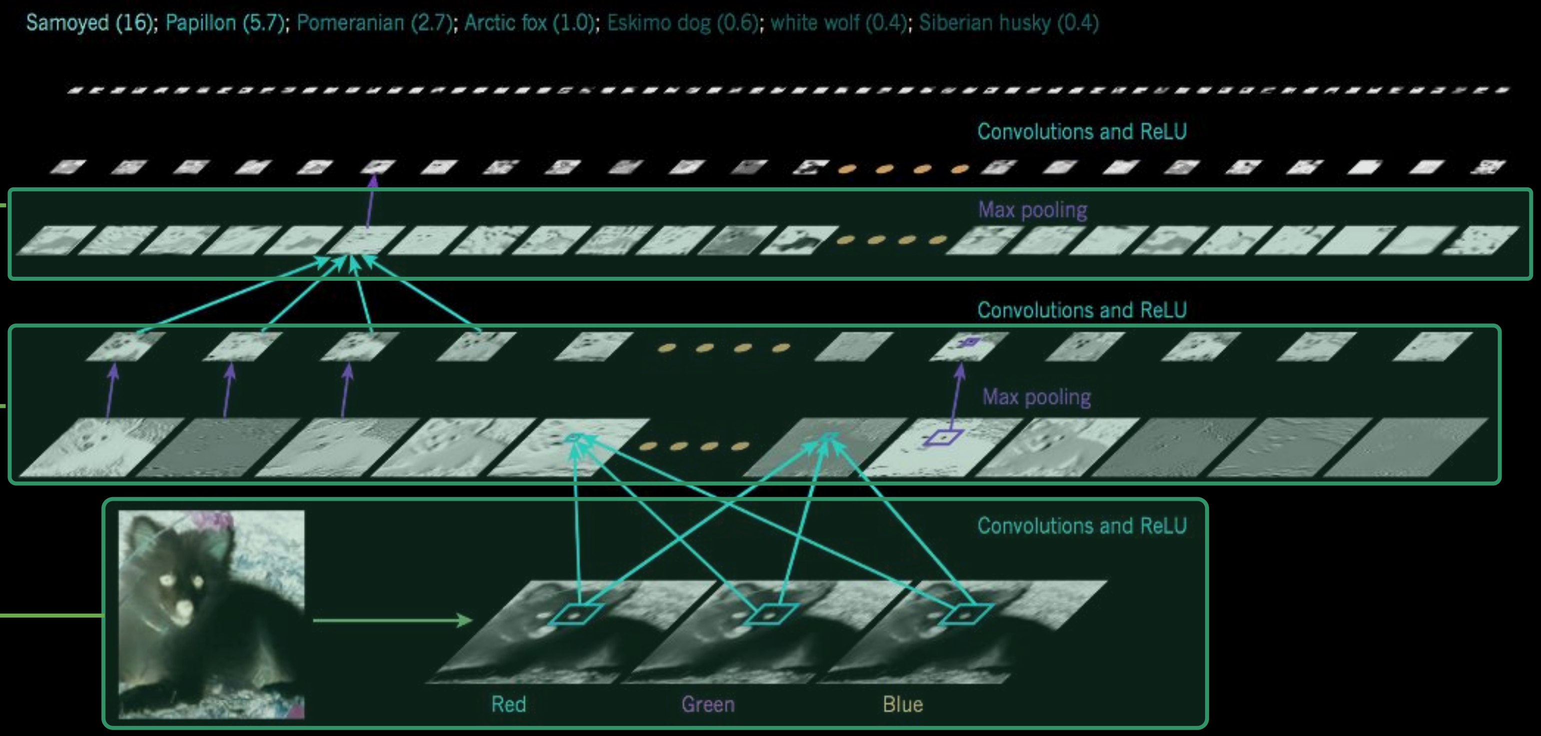
- Inspired by behavior of the visual cortex in cats (V1)
- CNN **replace the affine transforms (dense layers) by convolutions**
- Allows processing data with a grid-like or array topology

### Key concepts

Kernels are **convolved across all channels and summed**

Convolutional layer produces **feature maps** (for each kernel)
$\mathbf{F} \in \mathbb{R}^{K \times N \times M}$

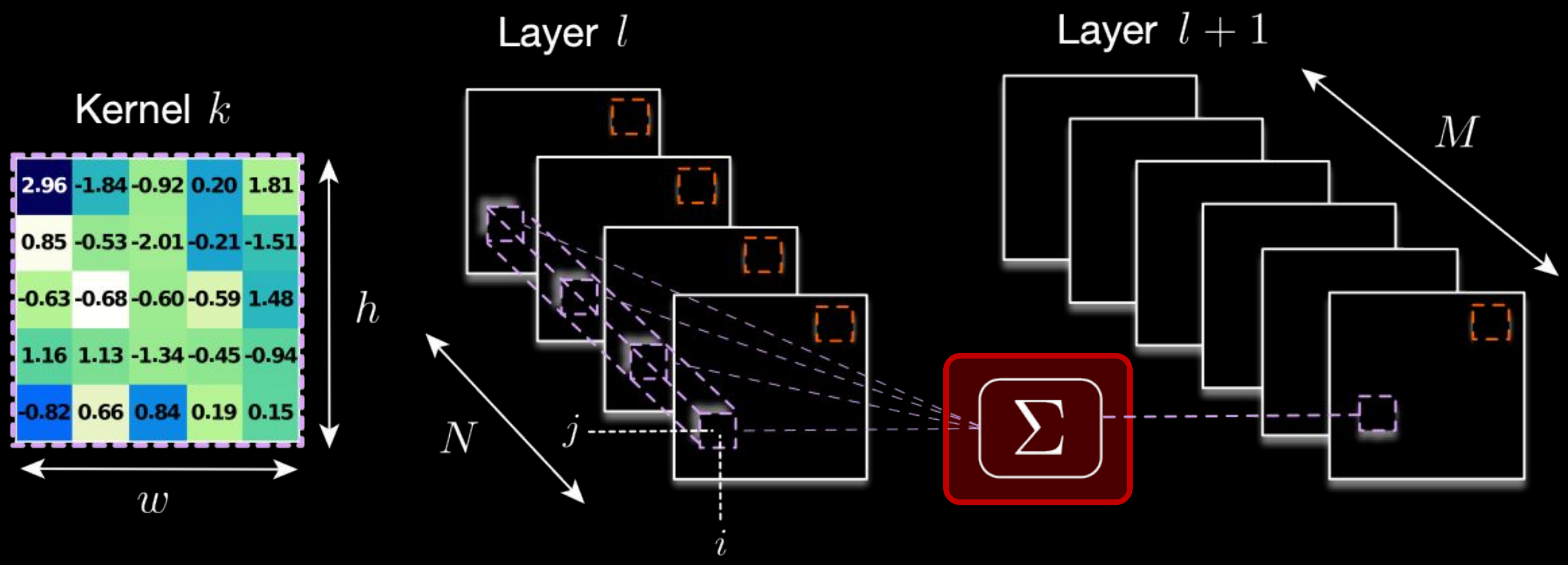
Input has several **channels**
$\mathbf{X} \in \mathbb{R}^{C \times H \times W}$



# Convolutional layer

## How to apply the convolution **across different channels**

- We aim to replace linear layers by convolutional ones
- However, the set of kernels produce **channels of feature maps**
- Solution is to **sum** across the output of convolution on each feature map

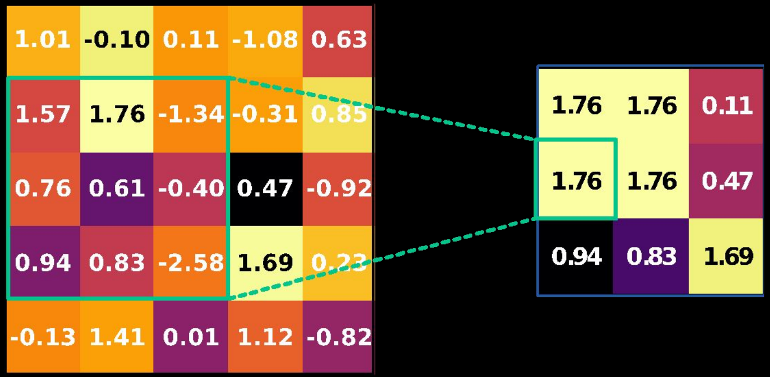



- After this, we also apply a **non-linear activation function**
- And also (eventually) apply **pooling** to the output

# Pooling operation

- Convolutions largely increase the dimensionality
- Pooling aims to summarize information in regions
- Allows to reduce the spatial dimensionality of the feature maps

Goals | Increase computational efficiency of the network
Provides robustness and spatial invariance




$$\mathbf{Y}_{i,j} = \frac{1}{k^2} \sum_{u=-k/2}^{k+2} \sum_{v=-k/2}^{k+2} \mathbf{X}_{(i+u),(j+v)}$$

Output of a pooling layer is a smaller **feature map**
Same number of channels as input.

- Provides *translational invariance*
- Allows increasingly large *receptive field*

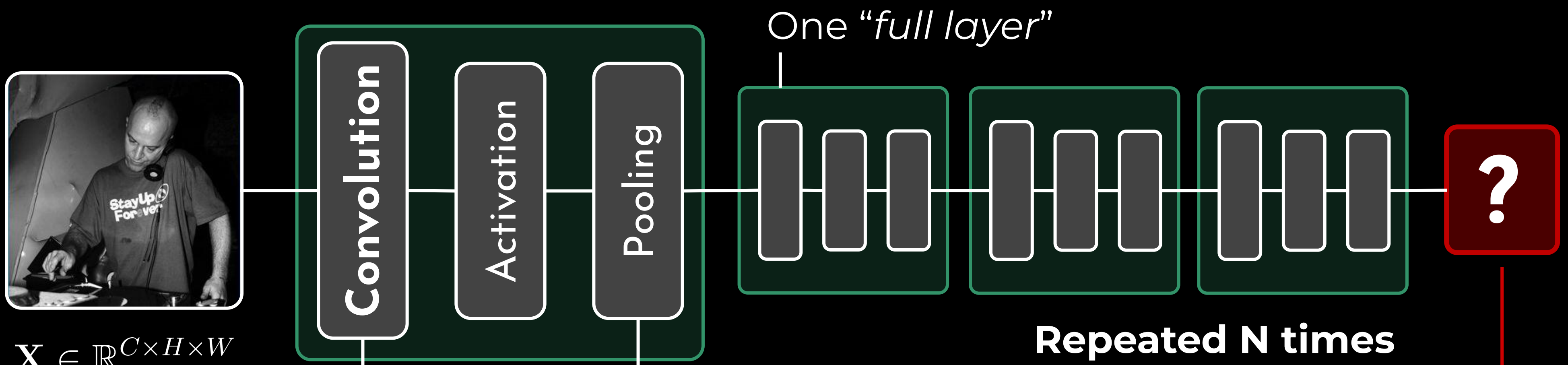
## Common types of pooling layers

Max pooling — Compute maximum value in non-overlapping rectangular regions

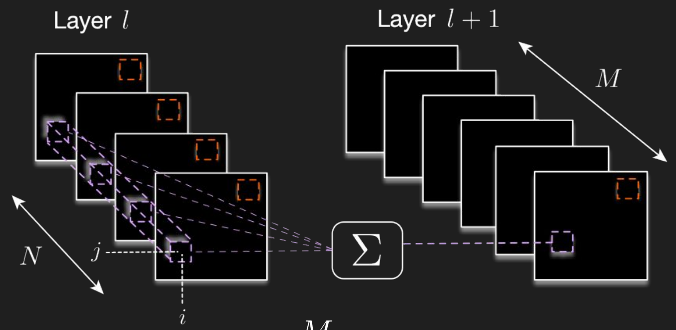
Average pooling — Compute average value in non-overlapping rectangular regions

# Typical architecture

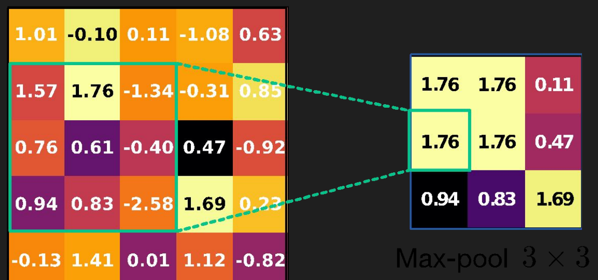
## Full architecture of a CNN




$\mathbf{X} \in \mathbb{R}^{C \times H \times W}$




$$(\mathbf{x} \star \mathbf{k})[n] = \sum_{m=-M}^{M} \mathbf{x}[n-m]\mathbf{k}[m].$$




$$\mathbf{Y}_{i,j} = \frac{1}{k^2} \sum_{u=-k/2}^{k+2} \sum_{v=-k/2}^{k+2} \mathbf{X}_{(i+u),(j+v)}$$

Output is a **set of feature maps**

How to process ?

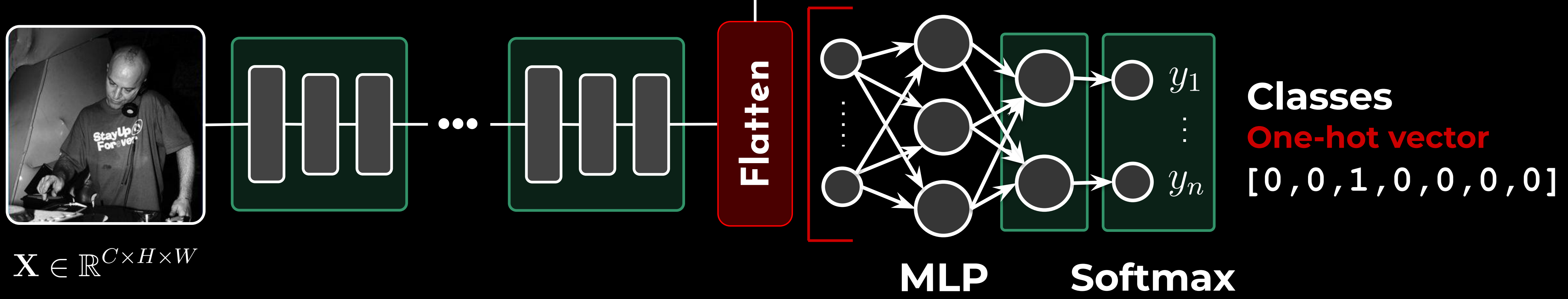
What loss to use ?

# Final dense layer and softmax

## What loss to use with spatial data?

- Sometimes problem is inherently spatial ... but what about classification
- Convolutional layers are usually followed by a **final dense** network

**Flatten** the feature maps as a **one-dimensional vector**




Softmax function

- Maps an arbitrary vector to a probability distribution
- Often used as final activation function for multi-class classification

$$\text{softmax}(\mathbf{x})_i = \frac{e^{x_i}}{\sum_{j=1}^{K} e^{x_j}}$$

# Backpropagation ?

Exactly the same concept as before with a few kinks

# Batch normalization

**Regularization**

Improve training by **normalizing the inputs of each layer**.
Mitigates vanishing and exploding gradient and allow higher learning rates.
Very successful regularization technique (**deep learning era**)

$$\tilde{\mathbf{x}} = \frac{\mathbf{x} - \mathbb{E}[\mathbf{x}]}{\sqrt{\mathrm{Var}[\mathbf{x}] + \epsilon}}$$

- $\mathbf{x}$ input to a layer
- $\tilde{\mathbf{x}}$ normalized input
- $\mathbb{E}[\mathbf{x}]$ mean of the batch
- $\mathrm{Var}[\mathbf{x}]$ variance of the batch
- $\epsilon$ small constant

Compute the mean and variance of the inputs over a mini-batch
Normalize the inputs to **zero mean and unit variance**.

$$\tilde{\mathbf{x}} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$$

**Learnable parameters**

Introduce learnable parameters to then scale and shift the normalized inputs

$$\hat{\mathbf{x}} = \gamma\tilde{\mathbf{x}} + \beta$$

**Interpretation**

Reduces internal covariate shift (change in the input distribution).
Reduces dependence on the gradient magnitude (larger learning rates).
Batch Normalization regularizes the model and reduces overfitting.

**Can be applied to any type of layer**

# Recurrent Neural Networks (RNN)

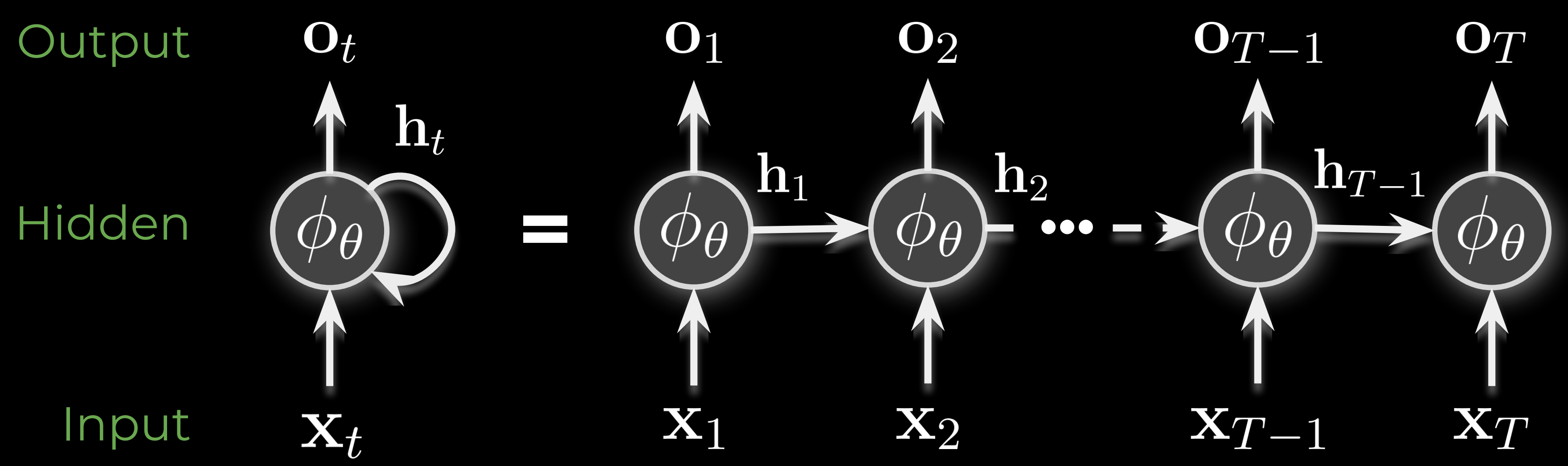
Motivation

How can our networks process **temporal** inputs
We would like to introduce some **time dependencies**
Hence, we need to account for the **previous hidden states**

$$\boxed{\mathbf{h}_t = \phi_\theta\left(\mathbf{h}_{t-1}, \mathbf{x}_t\right)}$$

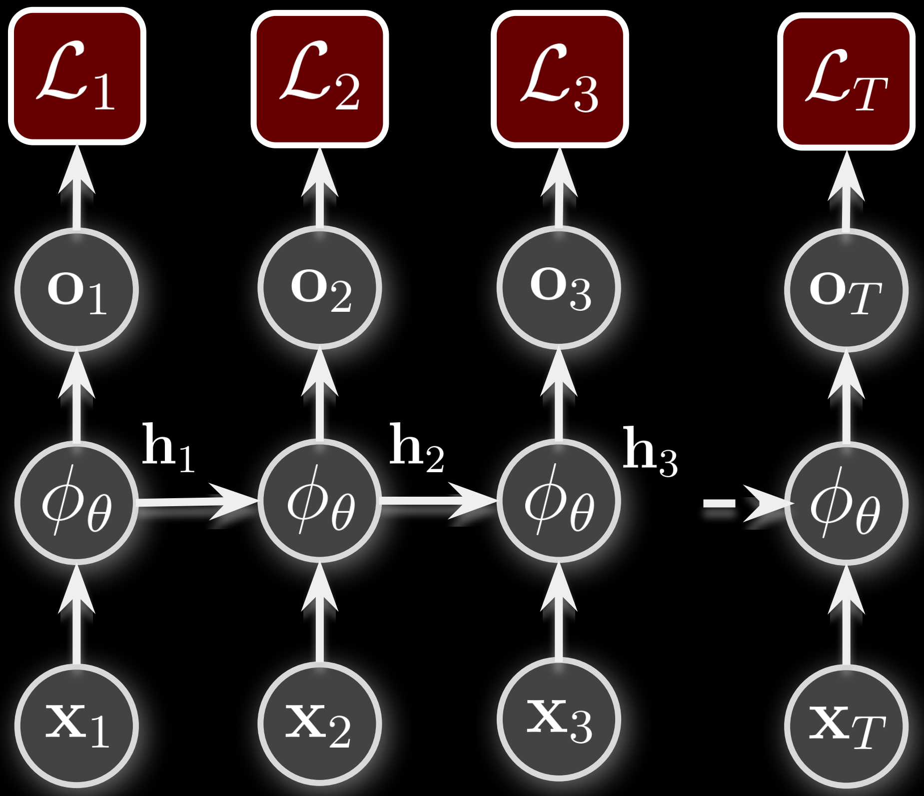
Introduce **loops**, problematic for backpropagation

Idea of *unfolding* recurrent networks



# Recurrent Neural Network (RNN)

## How to perform training




We can have a loss for each time point $\mathcal{L}_i$

Therefore, our total loss $\mathcal{L}$ is defined as

$$\mathcal{L} = \sum_{i=1}^{T} \mathcal{L}_i$$

Objective is to update the weights

$$\mathbf{W}^{(it+1)} = \mathbf{W}^{(it)} + \eta . \frac{\delta \mathcal{L}}{\delta \mathbf{W}^{(it)}}$$

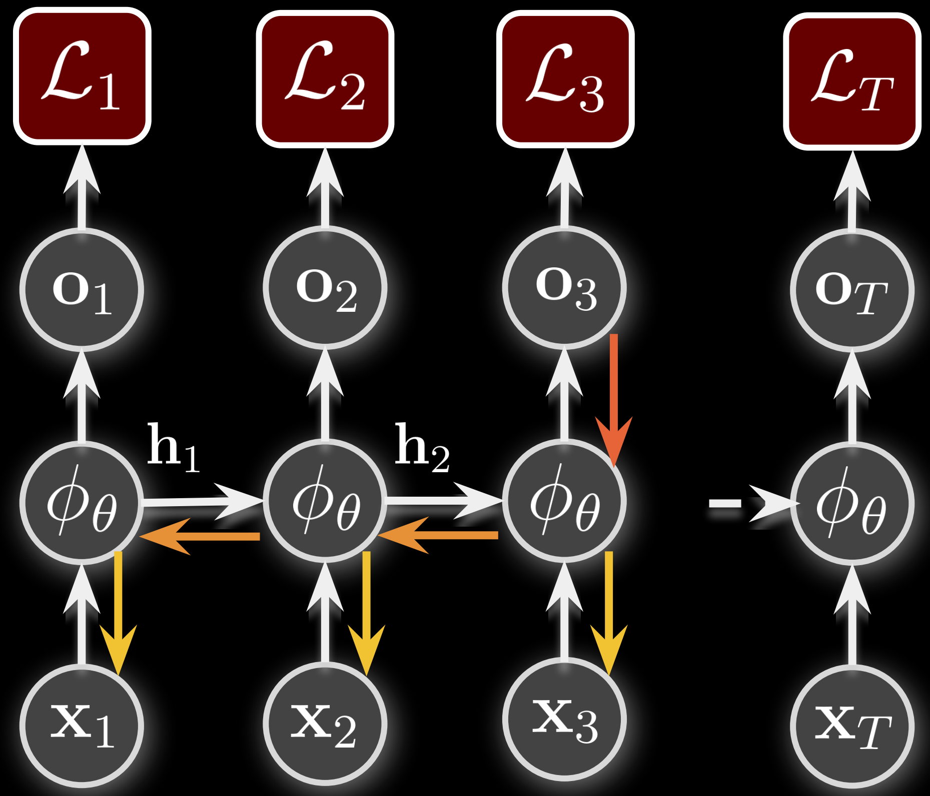
**Issue** The **same** $\mathbf{W}$ occurs each timestep

Every path from $\mathbf{W}$ to $\mathcal{L}_i$ is one dependency

**Solution:** Find all paths from $\mathbf{W}$ to all $\mathcal{L}_i$

# Recurrent Neural Network (RNN)

## Computing a given path in the loss




If we take a given time step

$$\frac{\delta \mathcal{L}_j}{\delta \mathbf{W}} = \sum_{k=1}^{j} \frac{\delta \mathcal{L}_j}{\delta \mathbf{h}_k} \boxed{\frac{\delta \mathbf{h}_k}{\delta \mathbf{W}}}$$

How to compute one dependency ?

**Use chain rule to fill missing steps**

$$\frac{\delta \mathcal{L}_j}{\delta \mathbf{W}} = \sum_{k=1}^{j} \frac{\delta \mathcal{L}_j}{\delta \mathbf{o}_j} \boxed{\frac{\delta \mathbf{o}_j}{\delta \mathbf{h}_j}} \boxed{\frac{\delta \mathbf{h}_j}{\delta \mathbf{h}_k}} \boxed{\frac{\delta \mathbf{h}_k}{\delta \mathbf{W}}}$$

# Problems with RNN

RNN are a powerful class of networks for temporal dependencies

However, they suffer from a number of issues

**Gradient issues** As we have just seen, backpropagation needs a **long multiplicative series**

$$\frac{\partial \mathcal{L}}{\partial \mathbf{w}} = \sum_{t=1}^{T} \frac{\partial \mathcal{L}_t}{\partial \mathbf{w}} = \sum_{t=1}^{T} \frac{\partial \mathcal{L}_t}{\partial \mathbf{h}_t} \frac{\partial \mathbf{h}_t}{\partial \mathbf{w}} = \sum_{t=1}^{T} \frac{\partial \mathcal{L}_t}{\partial \mathbf{h}_t} \boxed{\left( \prod_{k=t+1}^{T} \frac{\partial \mathbf{h}_k}{\partial \mathbf{h}_{k-1}} \right)} \frac{\partial \mathbf{h}_t}{\partial \mathbf{w}}$$

This introduces training instabilities and difficulties to learn long-term dependencies.

**Vanishing gradients** Gradients becomes increasingly small, **inability to learn long-term**

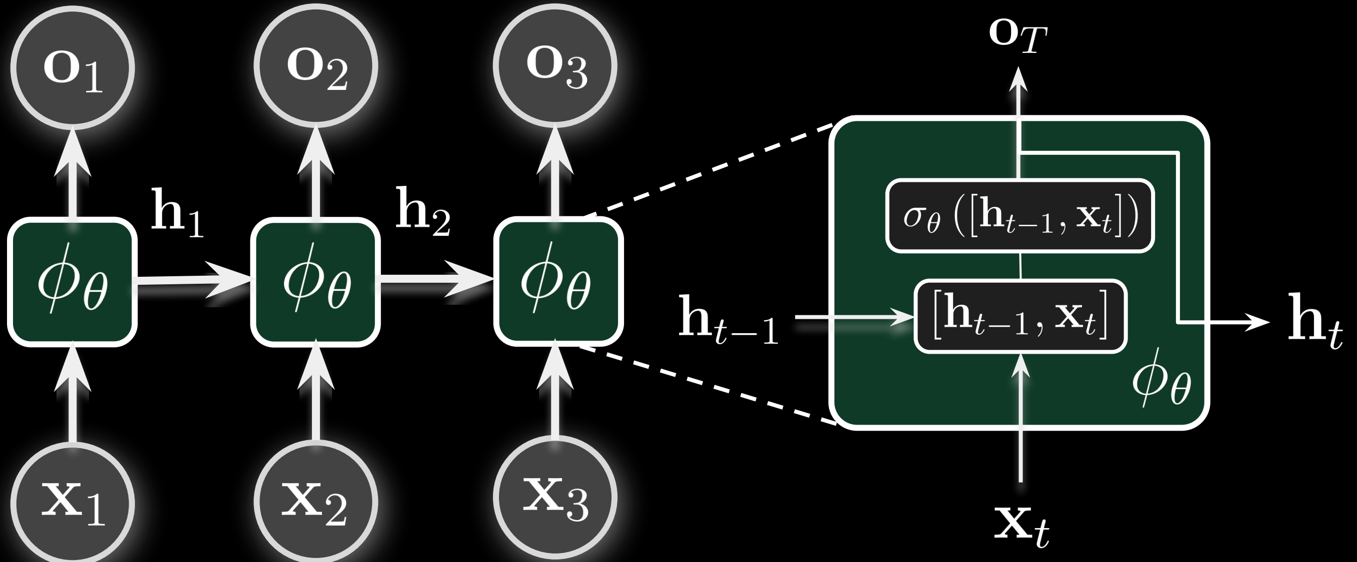
**Exploding gradients** Gradients becomes increasingly large, **training instability**

**Memory limitation** | RNNs have fixed-length memory (steps of backpropagation)
Cannot remember information too far back in the past.

**Slow training** | RNNs are forced to process one input at a time
Slow training and limits their parallelization.

# Recurrent Neural Network (RNN)

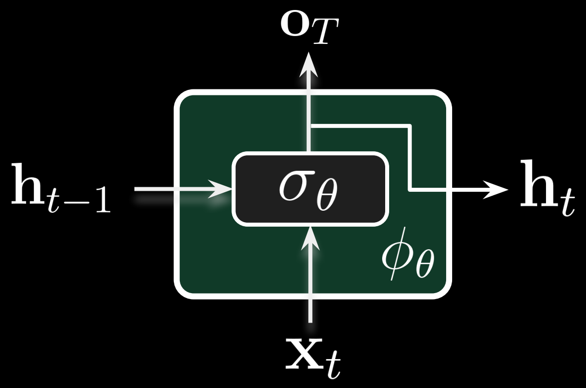
## Notion of **cell** containing replicated operations




**Simplest cell possible**

Sometimes called “Vanilla RNN”
Concatenates input and hidden
Apply some operation

**Simplified notation for cells**



# Long Short-Term Memory (LSTM)

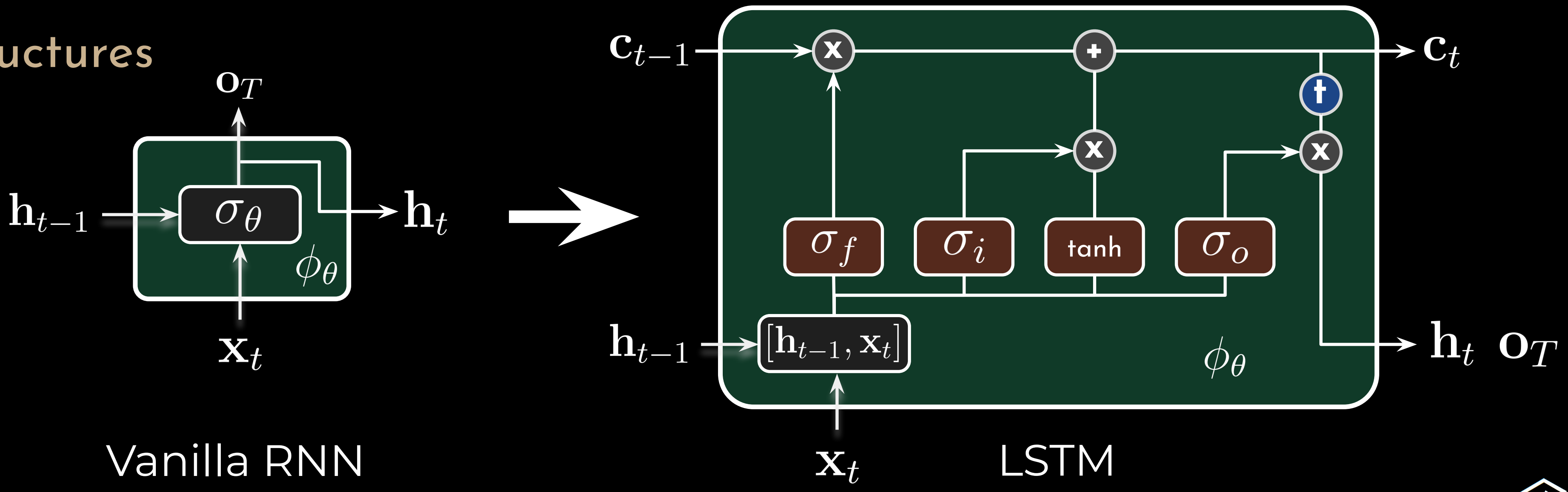
**Motivation** | Trying to address the vanishing and exploding gradients
Idea is to control the **flow of information** in the network with **gates**

Composed of a set of gates, and an external **cell state** (extra temporal hidden)

- **Input gate** Controls the flow of information into the cell state.
- **Forget gate** Controls the extent to which the previous cell state is retained.
- **Output gate** Controls the flow of information from the cell state to the hidden state.

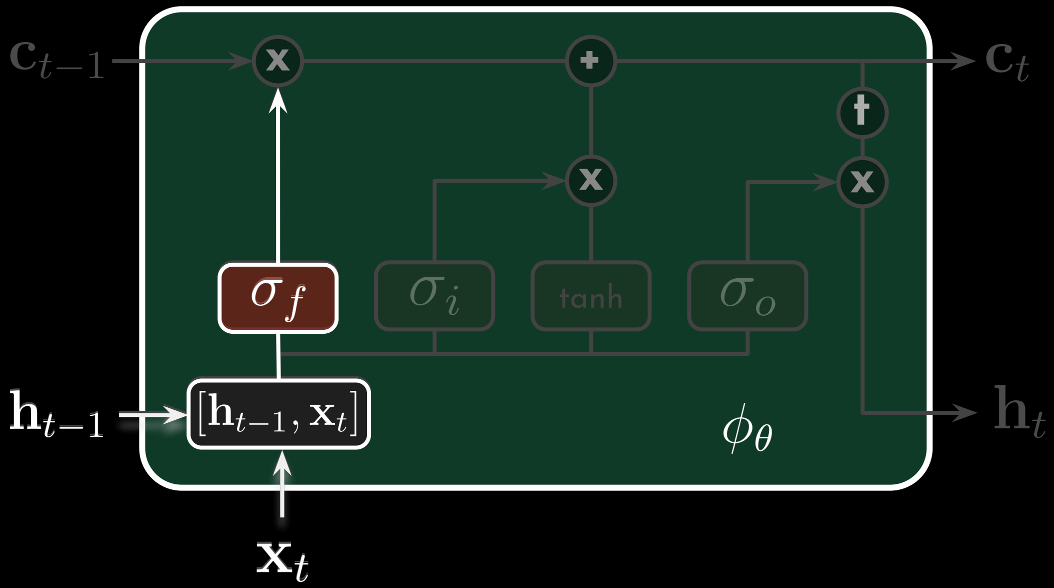
**Cell state** Extra temporal state, can be seen as a **long-term memory** (enclosing unit gradient)

## Cell structures




Vanilla RNN LSTM

# Long Short-Term Memory (LSTM)




## Forget gate

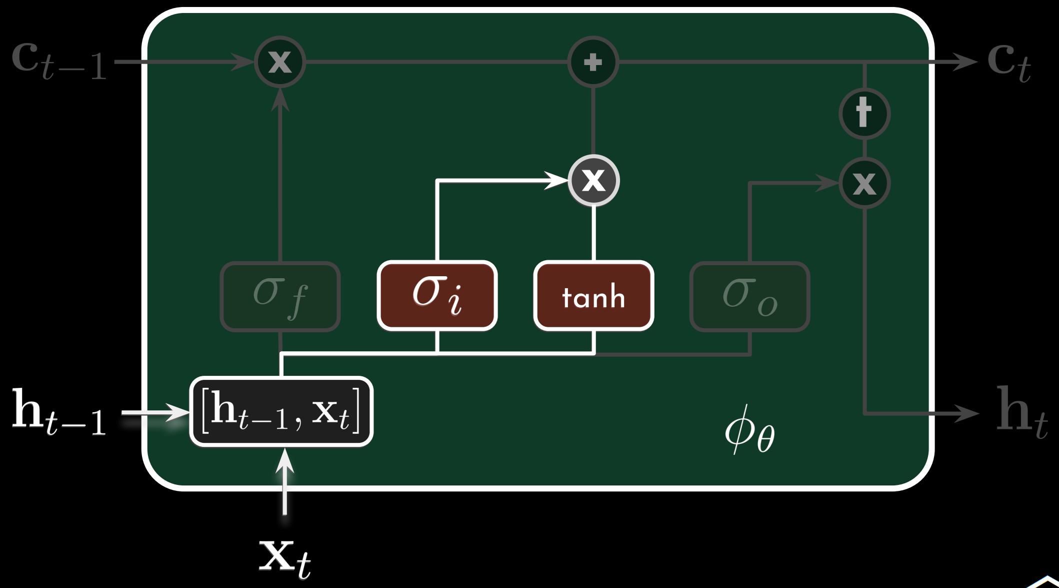
Controls how much previous cell state is retained.

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

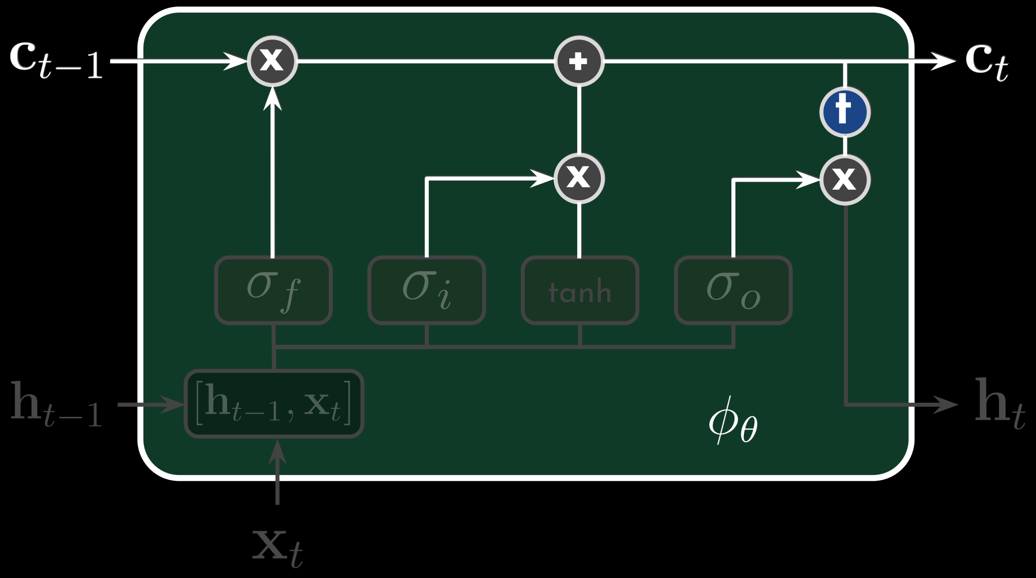
## Input gate

Controls flow of information into the cell state.

$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$$
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C)$$



# Long Short-Term Memory (LSTM)




## Cell state update

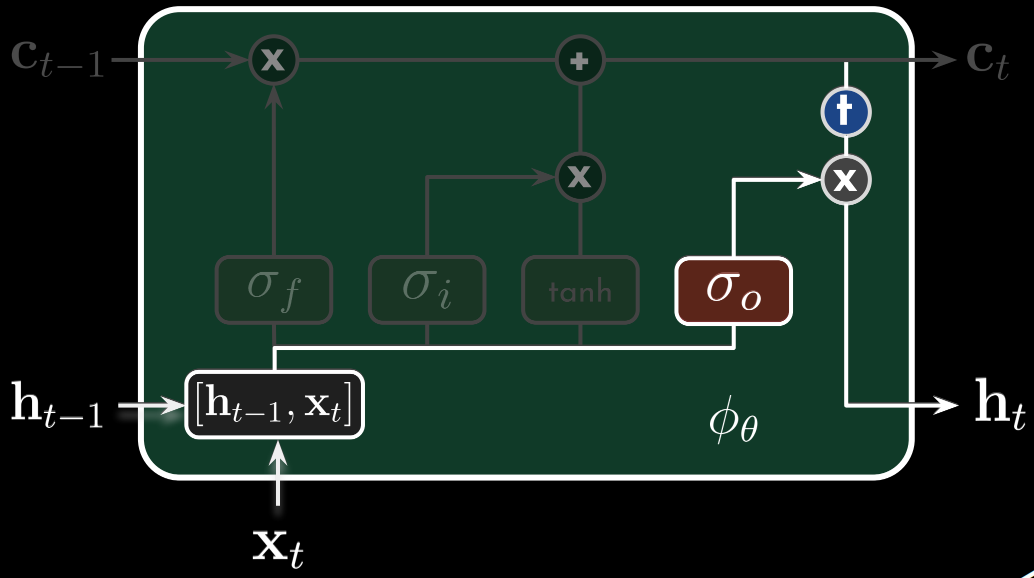
Combines previous state with candidate state, modulated by the input and forget gates.

$$C_t = f_t \odot C_{t-1} + i_t \odot \tilde{C}_t$$

## Output gate

Controls flow from cell to hidden state.

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

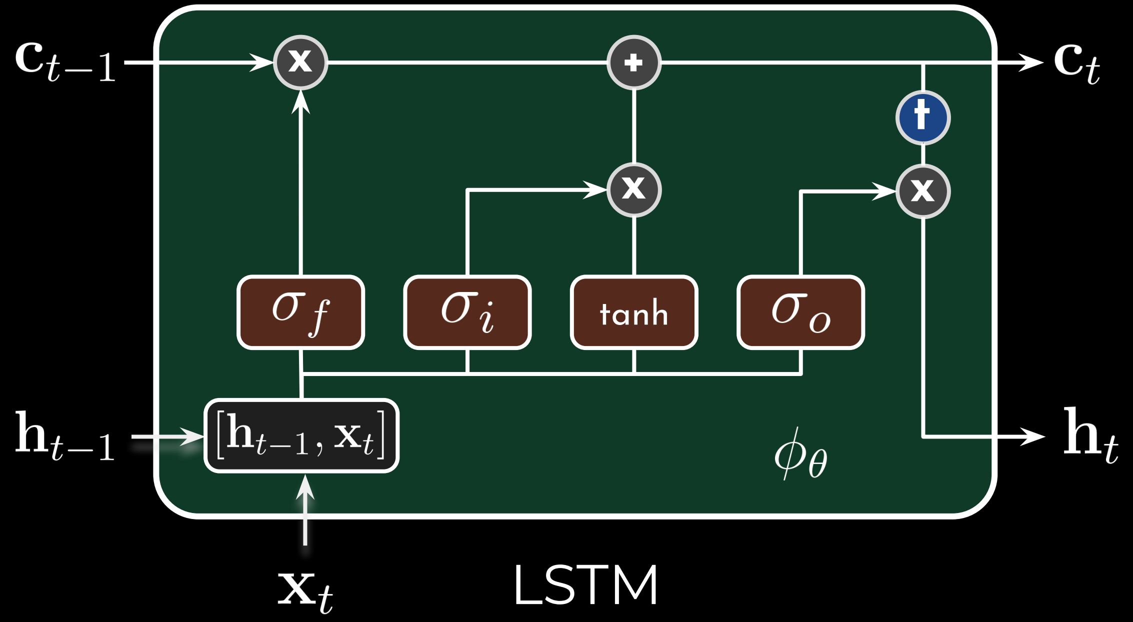


# Long Short-Term Memory (LSTM)

Trying to address the vanishing and exploding gradients

Idea is to control the **flow of information** in the network with **gates**

## Mathematical summary




LSTM

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

Controls flow of information into the cell state.

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

Controls how much previous cell state is retained.

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

Controls flow from cell to hidden state.

**Cell state update**

$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C)$$
$$C_t = f_t \odot C_{t-1} + i_t \odot \tilde{C}_t$$

Computes new cell state

**Hidden state update** $h_t = o_t \odot \tanh(C_t)$

Computes new hidden state

# Gated Recurrent Units (GRU)

**Cell structure**

Simplified version of LSTM with fewer gates and **single hidden**.
Only **update** and **reset** gates control the flow of information a

**Update gate**

Controls how much previous hidden state is retained and combined with new input.

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

**Reset gate**

Controls how much previous hidden state is used to compute candidate hidden state.

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

**Candidate hidden state**

Computes hidden state based on input and previous hidden modulated by reset gate.

$$\tilde{h}_t = \tanh(W_h \cdot [r_t \odot h_{t-1}, x_t] + b_h)$$

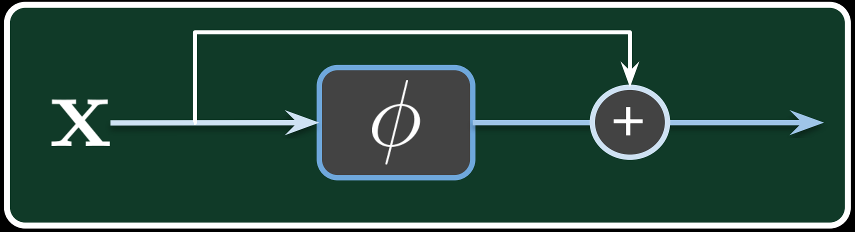
More computationally efficient than LSTM
Also usually more stable in the training
However, results (accuracy) vary

# Residual Networks

**Motivation:** Solving vanishing and exploding gradient in deep networks.

**Key idea:** Add a skip connection that allows the gradient to bypass layers.

Residual blocks $\psi(\mathbf{x}) = \phi(\mathbf{x}) + \mathbf{x}$




Identity shortcut: No additional parameters, only direct connection

$$\psi(\mathbf{x}) = \phi(\mathbf{x}) + \mathbf{x}$$

Projection shortcut: Linear projection to match potentially changing dimensions

$$\psi(\mathbf{x}) = \phi(\mathbf{x}) + \mathbf{W}_s\mathbf{x}$$

Stacking residual blocks: Residual blocks are stacked to create deep ResNets.

**Example**: *ResNet-50*, *ResNet-101*, and *ResNet-152*, where the numbers indicate the total number of layers.

Benefits
- Improved gradient flow through the network.
- Mitigates the vanishing/exploding gradient problem.
- Enables training of very deep neural networks.

**Same idea obviously applies to convolutional networks**