Demystifying deep learning

Petar Veličković

Artificial Intelligence Group
Department of Computer Science and Technology, University of Cambridge, UK

London Data Science Summit **20 October 2017**

Introduction

- In this talk, I will guide you through a condensed story of how deep learning became what it was today, and where it's going.
- This will involve a journey through the essentials of how neural networks normally work, and an overview of some of their many variations and applications.
- **Not a deep learning tutorial!** (wait for Sunday. :))
- *Disclaimer*: Any views expressed (especially with respect to influential papers) are entirely my own, and influenced perhaps by the kinds of problems I'm solving. It's fairly certain that any deep learning researcher would give a different account of what's the most important work in the field. :)

Motivation: notMNIST

- Which characters do you see? (*How* did you conclude this?)

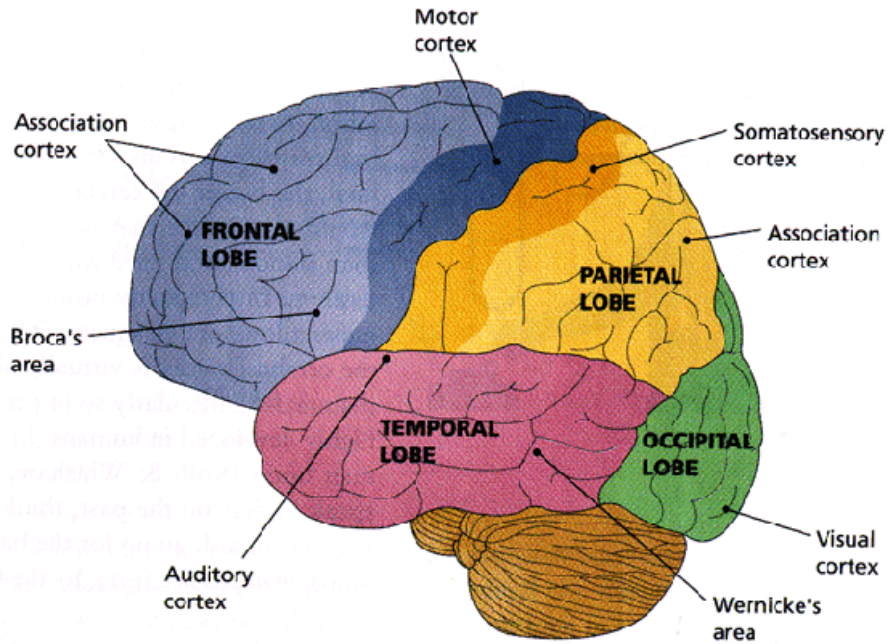
- Imagine someone asked you to write a program that recognises characters from arbitrary glyphs. . .

Intelligent systems

- Although the previous task was likely *simple* to you, you (probably) couldn't turn your thought process into a concise sequence of instructions for a program!
- Unlike a *"dumb"* program (that just blindly executes preprogrammed instructions), you've been exposed to a lot of `A` characters during your lifetimes, and eventually *"learnt"* the complex features making something an `A`!
- Desire to design such systems (capable of *generalising* from past experiences) is the essence of *machine learning*!
  - How many such systems do we know from nature?

Specialisation in the brain

- We know that *different parts* of the brain perform *different tasks*:

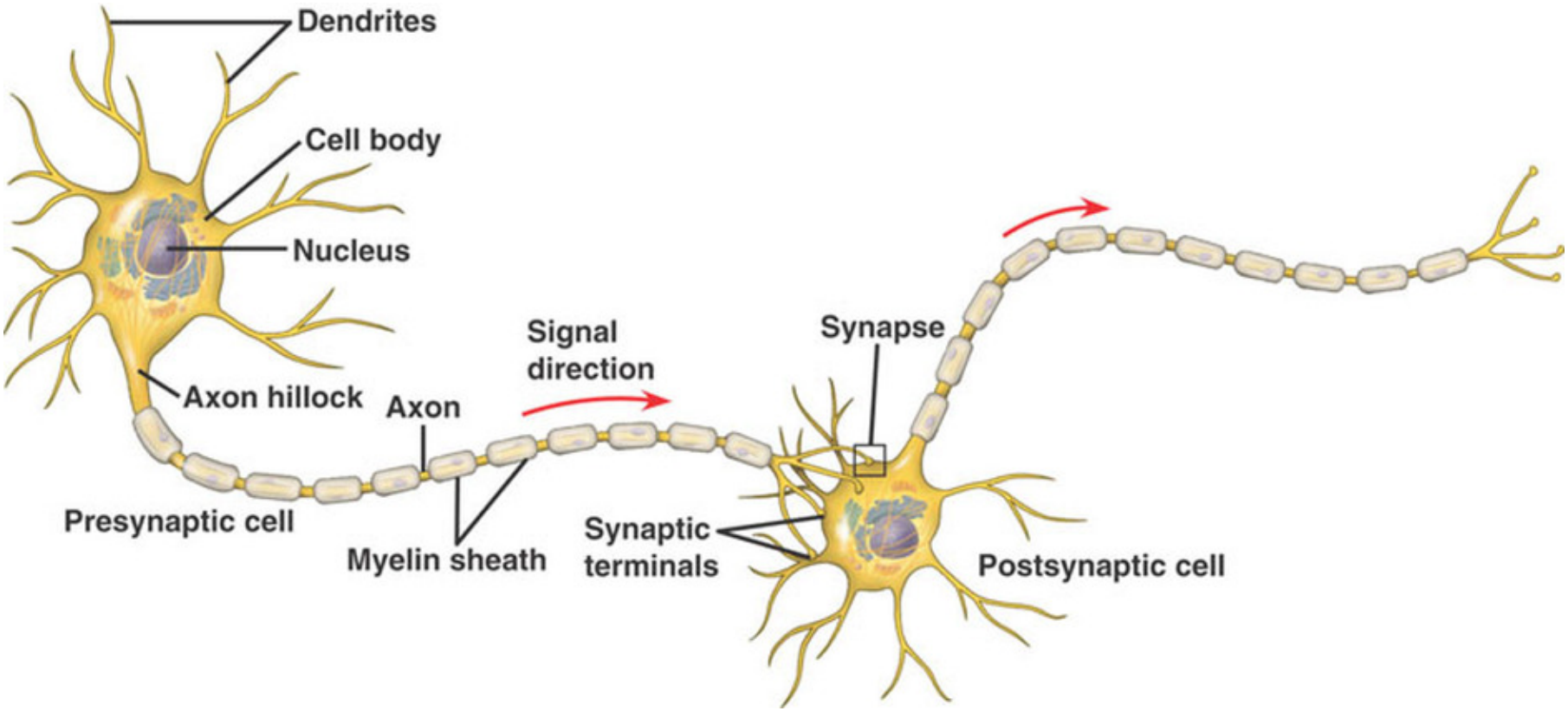


- There is increasing evidence that the brain:
  - Learns from *exposure to data*;
  - Is *not* preprogrammed!

Brain & data

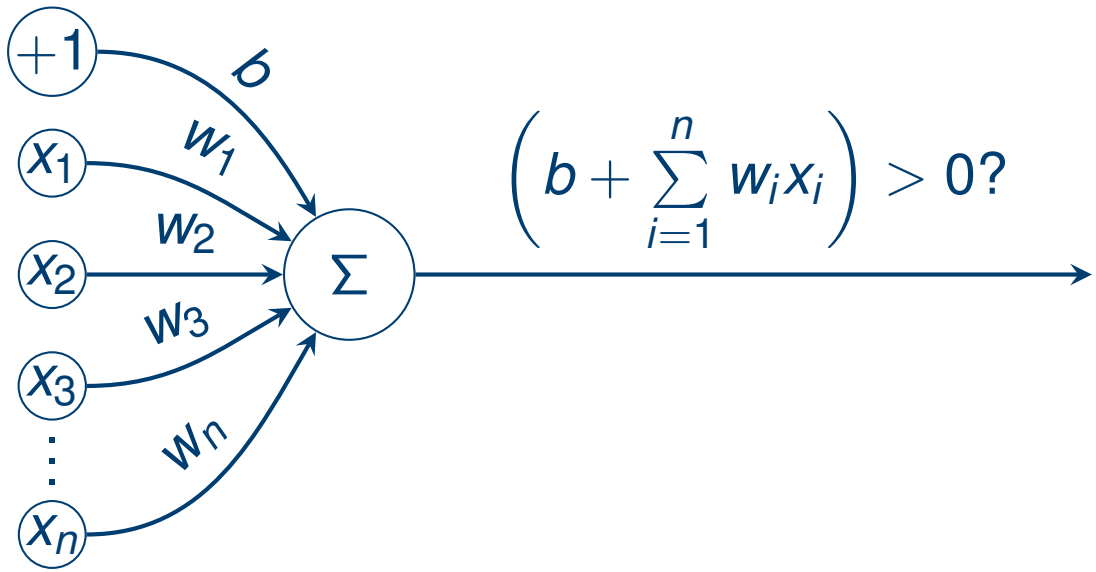
- The majority of what we know about the brain comes from studying *brain damage*:
  - Rerouting visual inputs into the auditory region of baby ferrets makes this region capable of dealing with visual input!
  - As far as we know (for now), the modified region works equally good as the visual cortex of healthy ferrets!
- If there are no major biological differences in learning to process different kinds of input. . .
- $\Longrightarrow$ the brain likely uses a *general learning algorithm*, capable of adapting to a wide spectrum of inputs.
- We'd very much like to capture this algorithm!

A *real* neuron!



An *artificial* neuron!

Within this context sometimes also called a *perceptron* (. . .)



This model of a *binary classifier* was created by Rosenblatt in 1957.

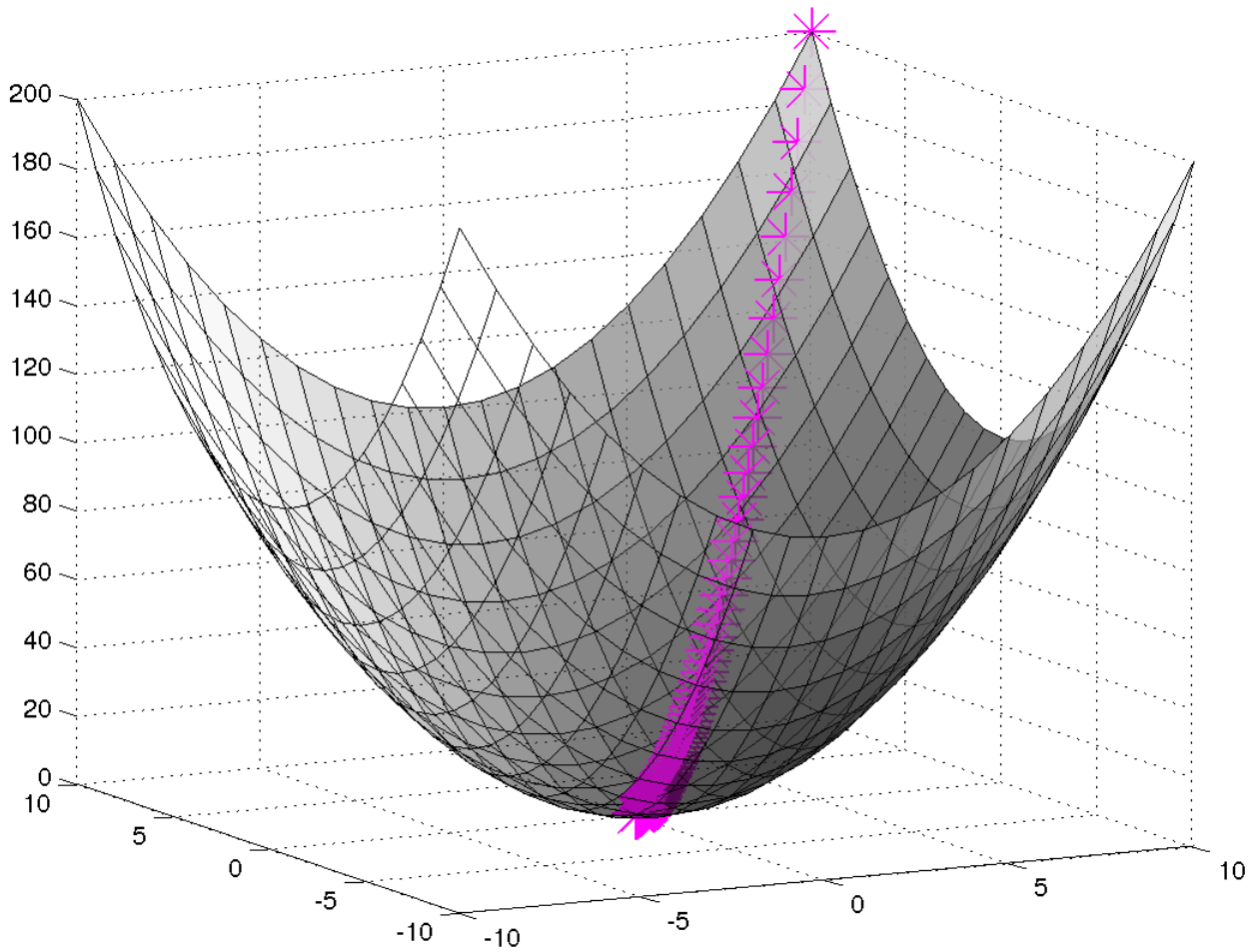
Gradient descent

- There needed to be a way to *learn* the neuron's parameters $(\vec{w}, b)$ *from data*.
- One very popular technique to do this was *gradient descent*—if we have a way of quantifying the errors (a differentiable *loss* function, $\mathcal{L}$) that our neuron makes on training data, we can then iteratively update our weights as

$$\vec{w}_{t+1} \leftarrow \vec{w}_t - \eta \frac{\partial \mathcal{L}}{\partial \vec{w}}\bigg|_{\vec{w}_t}$$

  where the gradients $\frac{\partial \mathcal{L}}{\partial \vec{w}}$ are computed on the training data.
- Here, η is the *learning rate*, and properly choosing it is *critical*—remember it, it will be important later on!

Gradient descent



Custom activations

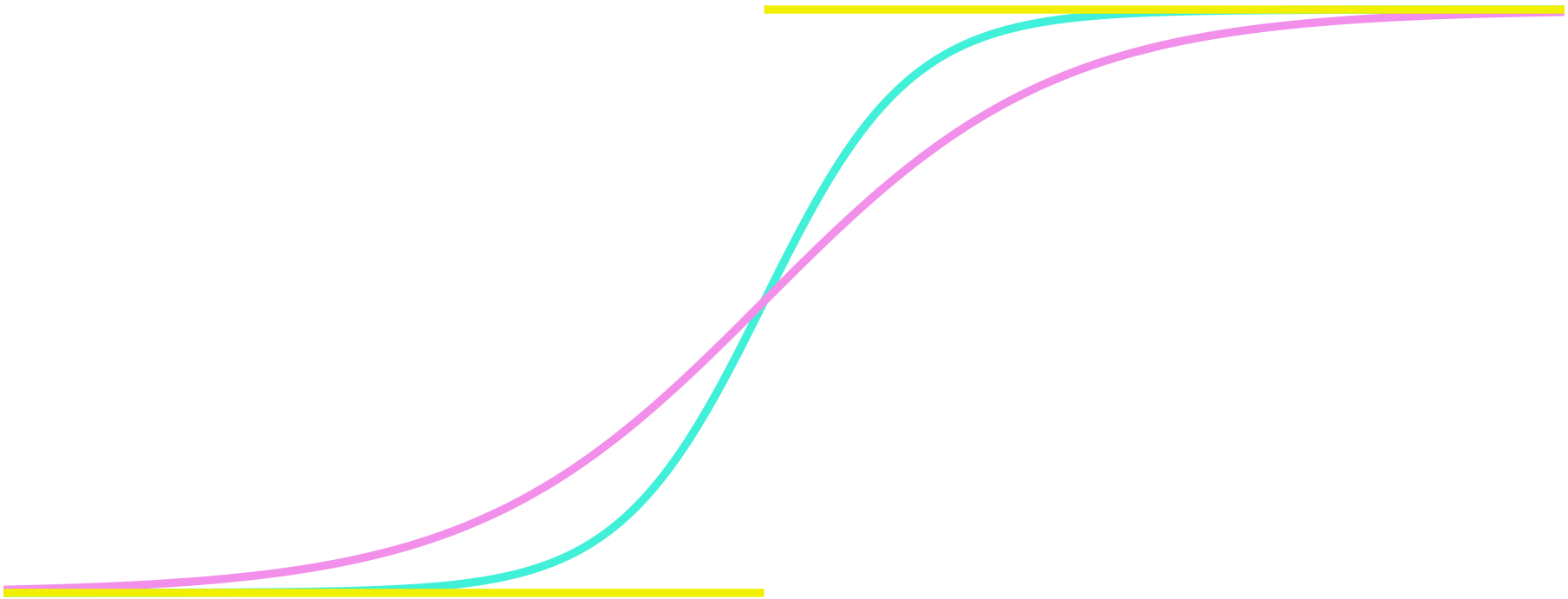
- Fundamental issue: the function we apply to our network's output (essentially the *step* function) is non-differentiable!
- Solution: use its smooth variant, the *logistic sigmoid*

$$\sigma(x) = \frac{1}{1 + \exp(-ax)}$$

  and classify depending on whether $\sigma(x) > 0.5$.
- We can generalise this to k-class classification using the *softmax* function:

$$\mathbb{P}(\textit{Class } i) = \textit{softmax}(\vec{x})_i = \frac{\exp(x_i)}{\sum_j \exp(x_j)}$$

The logistic function



The loss function

- If we now define a suitable loss function to minimise, we may train the neuron using gradient descent!
- If we don't know anything about the problem, we may often use the *squared error loss*—if our perceptron computes the function $h(\vec{x}; \vec{w}) = \sigma\left(b + \sum_{i=1}^{n} w_i x_i\right)$, then on training example $(\vec{x}_p, y_p)$:

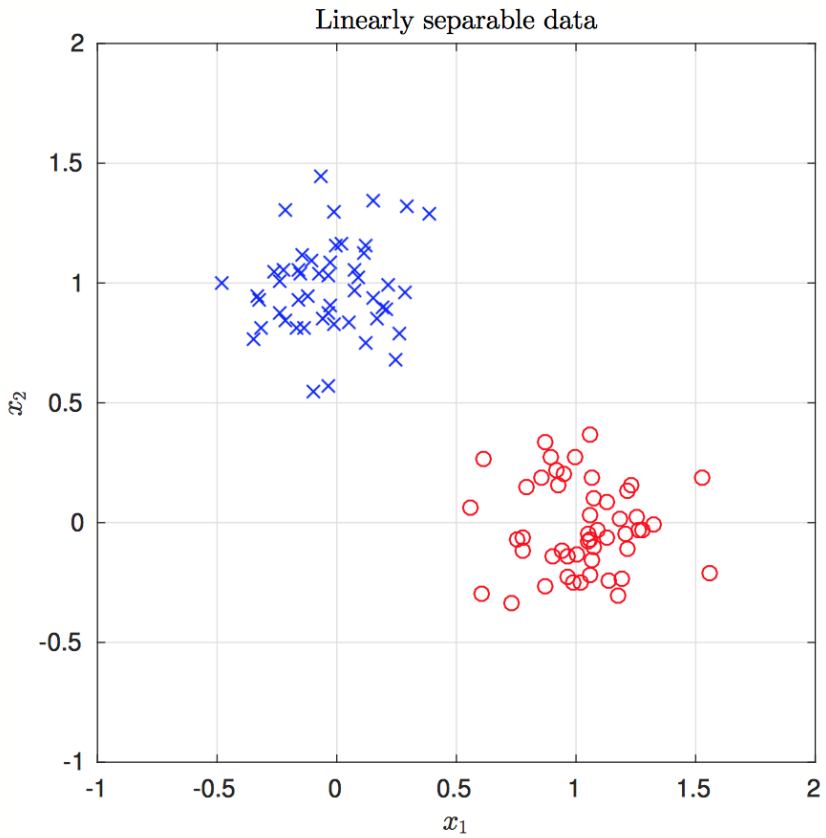
$$\mathcal{L}_p(\vec{w}) = (y_p - h(\vec{x}_p; \vec{w}))^2$$

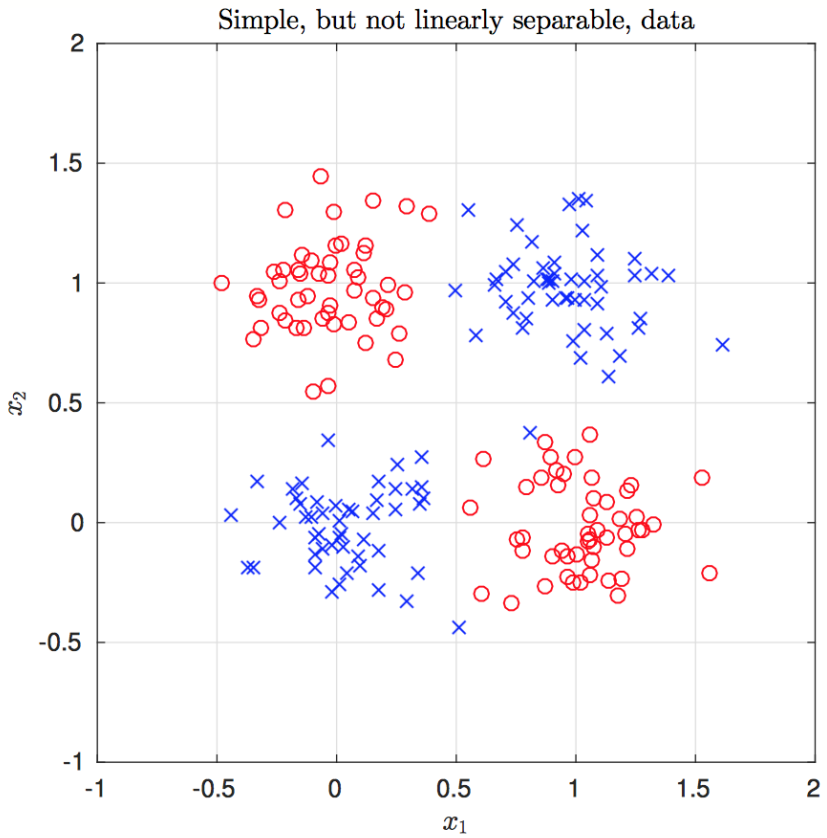
- Since the logistic function outputs a *probability*, we may exploit this to create a more informed loss—the *cross-entropy* loss:

$$\mathcal{L}_p(\vec{w}) = -y_p \log h(\vec{x}_p; \vec{w}) - (1 - y_p) \log(1 - h(\vec{x}_p; \vec{w}))$$

You may recognise the setup so far as *logistic regression*.

The infamous XOR problem



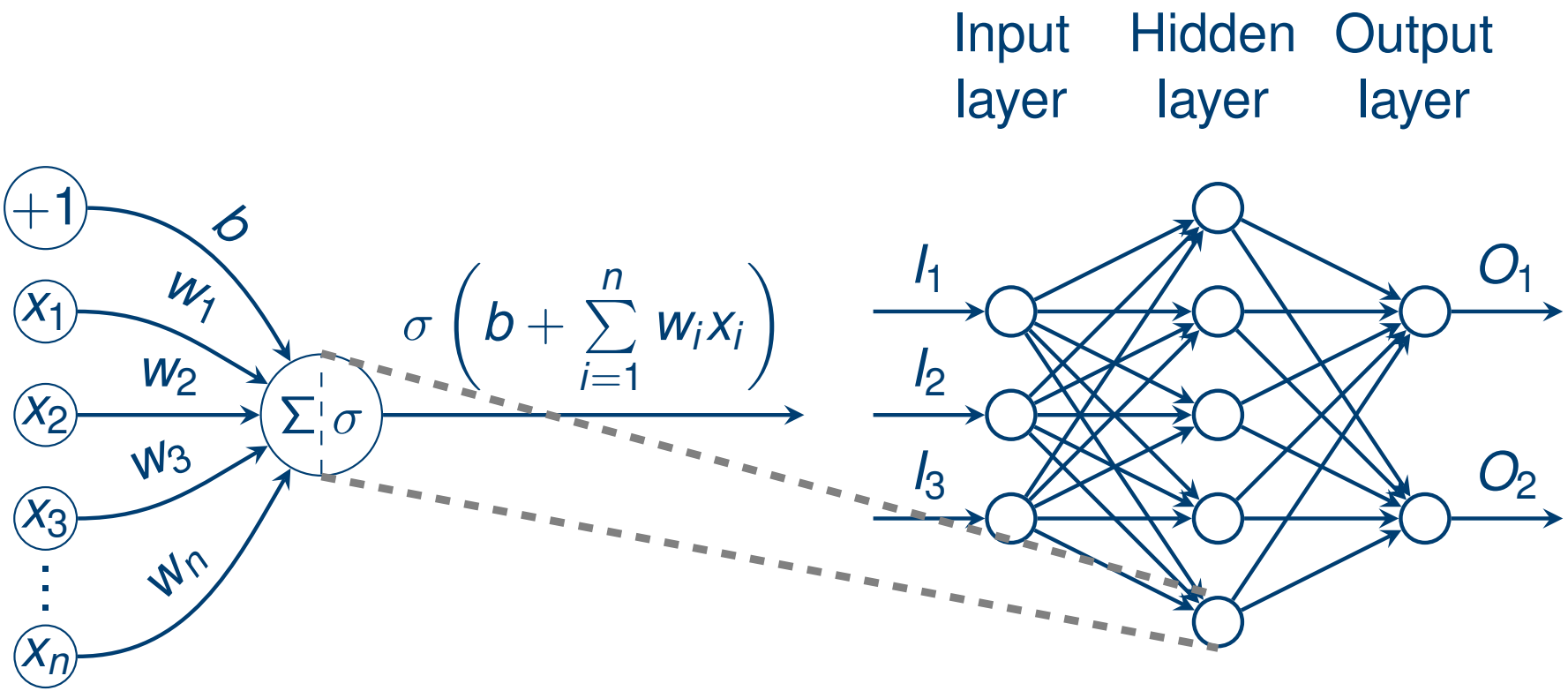


Neural networks and deep learning

- To get any further, we need to be able to introduce *nonlinearity* into the function our system computes.
- It is easy to extend a single neuron to a *neural network*—simply connect outputs of neurons to inputs of other neurons.
- If we apply nonlinear activation functions to intermediate outputs, this will introduce the desirable properties.
- Typically we organise neural networks in a sequence of *layers*, such that a single layer only processes output from the previous layer.

Multilayer perceptrons

The most potent feedforward architecture allows for full connectivity between layers—sometimes also called a *multilayer perceptron*.

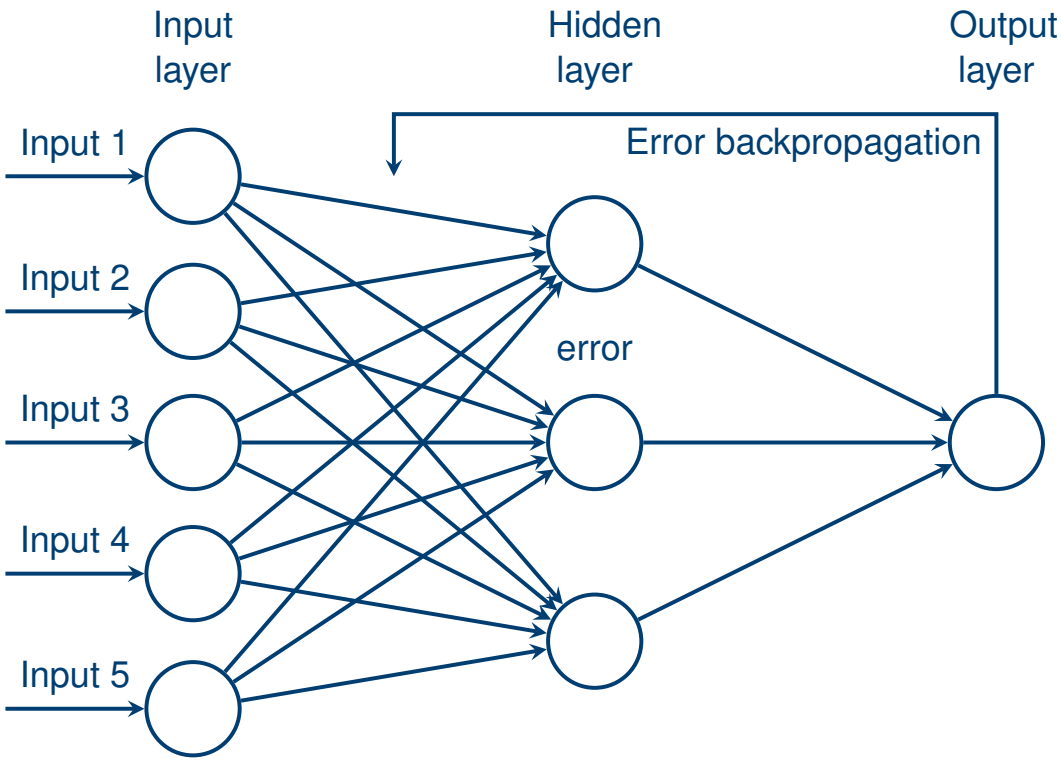


Backpropagation

- Variants of multilayer perceptrons (MLPs) have been known since the 1960s. In a way, **everything** that modern deep learning is utilising are specialised MLPs!
- Stacking neurons in this manner will preserve differentiability, so we can re-use gradient descent to train them once again.
- However, computing $\frac{\partial \mathcal{L}}{\partial w'}$ for an arbitrary weight w' in such a network was not initially very efficient...
- ...until 1985, when the *backpropagation* algorithm was introduced by Rumelhart, Hinton and Williams.

Backpropagation

Compute gradients directly at output neurons, and then propagate them backwards using the *chain rule*!



Hyperparameters

- Gradient descent optimises solely the weights and biases in the network. How about:
  - the number of hidden layers?
  - the amount of neurons in each hidden layer?
  - the activation functions of these neurons?
  - the number of iterations of gradient descent?
  - the learning rate?
  - …
- These parameters *must be fixed before training commences*! For this reason, we often call them **hyperparameters**.
- Optimising them remains an *extremely difficult problem*—we often can't do better than separating some of our training data for evaluating various combinations of hyperparameter values.

Neural network depth

- I'd like to highlight a specific hyperparameter: *the number of hidden layers*, i.e. *the network's depth*.
- What do you think, how many hidden layers are *sufficient* to learn any (bounded continuous) real function?

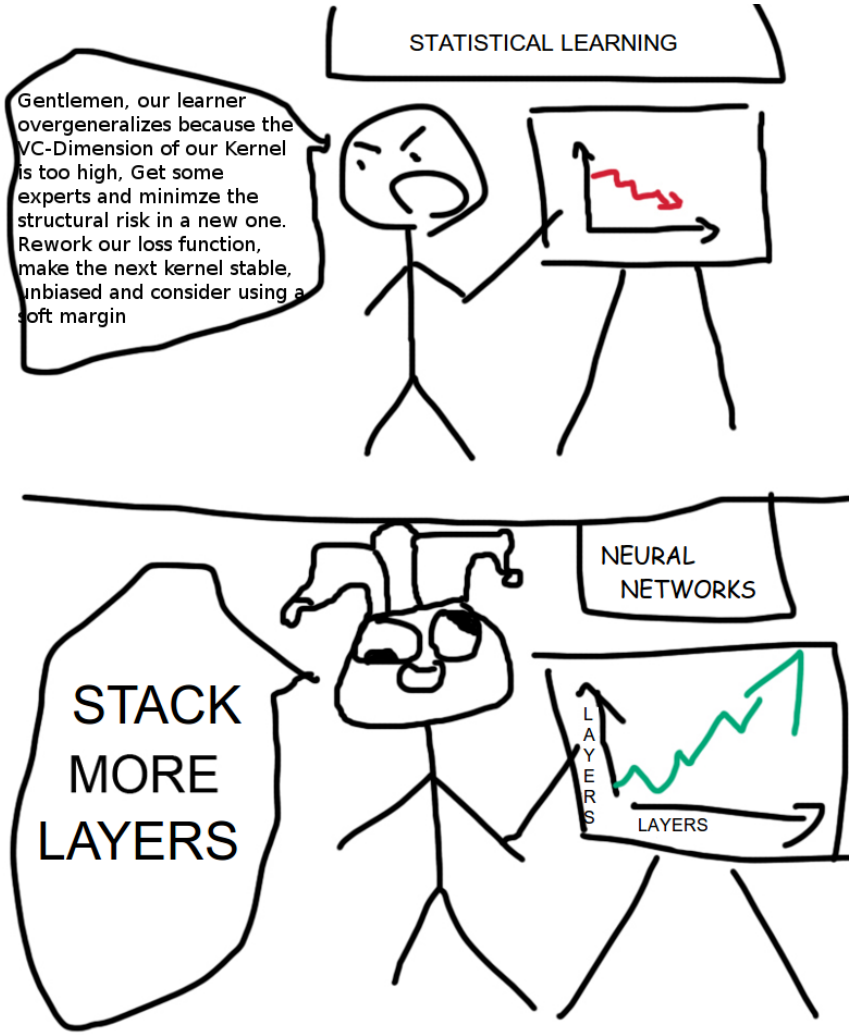
Neural network depth

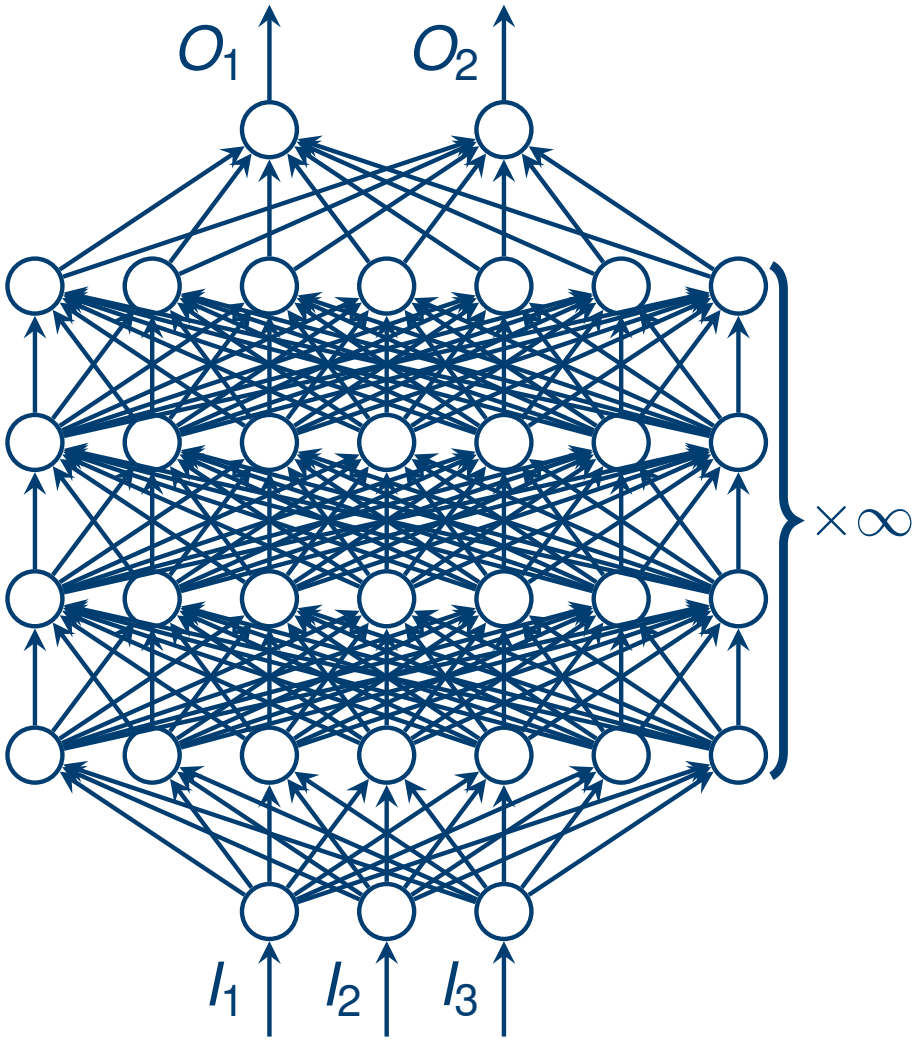
- I'd like to highlight a specific hyperparameter: *the number of hidden layers*, i.e. *the network's depth*.
- What do you think, how many hidden layers are *sufficient* to learn any (bounded continuous) real function?
- One! (*Cybenko's theorem*, 1989.)

Neural network depth

- I'd like to highlight a specific hyperparameter: *the number of hidden layers*, i.e. *the network's depth*.
- What do you think, how many hidden layers are *sufficient* to learn any (bounded continuous) real function?
- One! (*Cybenko's theorem*, 1989.)
- However, the proof is *not constructive*, i.e. does not give the optimal width of this layer *or* a training algorithm.
- *We must go deeper. . .*
  - Every network with > 1 hidden layer is considered *deep*!
  - Today's *state-of-the-art* networks often have over 150 layers.

Deep neural networks





Quiz: What do we have here?

DeepBlue vs. AlphaGo

- Main idea (roughly) the same: *assume that a grandmaster is only capable of thinking k steps ahead*—then generate a (near-)optimal move when considering $k' > k$ steps ahead.
  - DeepBlue does this exhaustively, AlphaGo sparsely (discarding many "highly unlikely" moves).
- One of the key issues: when *stopping exploration*, how do we determine the *advantage* that player 1 has?

DeepBlue: Gather a team of *chess experts*, and define a function $f : \mathit{Board} \to \mathbb{R}$, to define this advantage.

AlphaGo: Feed the raw state of the board to a deep neural network, and have it *learn* the advantage function *by itself*.

- This highlights an important *paradigm shift* brought about by deep learning...

Feature engineering

- Historically, machine learning problems were tackled by defining a set of *features* to be manually extracted from raw data, and given as inputs for *"shallow"* models.
  - Many scientists built *entire PhDs* focusing on features of interest for just one such problem!
  - Generalisability: very small (often zero)!
- With deep learning, the network *learns the best features by itself*, directly from *raw data*!
  - For the first time connected researchers from fully distinct areas, e.g. *natural language processing* and *computer vision*.
  - $\Longrightarrow$ a person capable of working with deep neural networks may readily apply their knowledge to create state-of-the-art models in virtually any domain (assuming a large dataset)!

Representation learning

- As inputs propagate through the layers, the network captures more complex *representations* of them.
- It will be extremely valuable for us to be able to reason about these representations!
- Typically, models that deal with *images* will tend to have the best visualisations.
- Therefore, I will now provide a brief introduction to these models (convolutional neural networks). Then we can look into the kinds of representations they capture. . .

Working with images

- Simple fully-connected neural networks (as described already) typically fail on high-dimensional datasets (e.g. images).
  - Treating each pixel as an independent input...
  - ...results in $h \times w \times d$ new parameters per neuron in the first hidden layer...
  - ...quickly deteriorating as images become larger—requiring exponentially more data to properly fit those parameters!
- **Key idea:** downsample the image until it is small enough to be tackled by such a network!
  - Would ideally want to extract some useful features first...
- $\implies$ exploit spatial structure!

The *convolution* operator

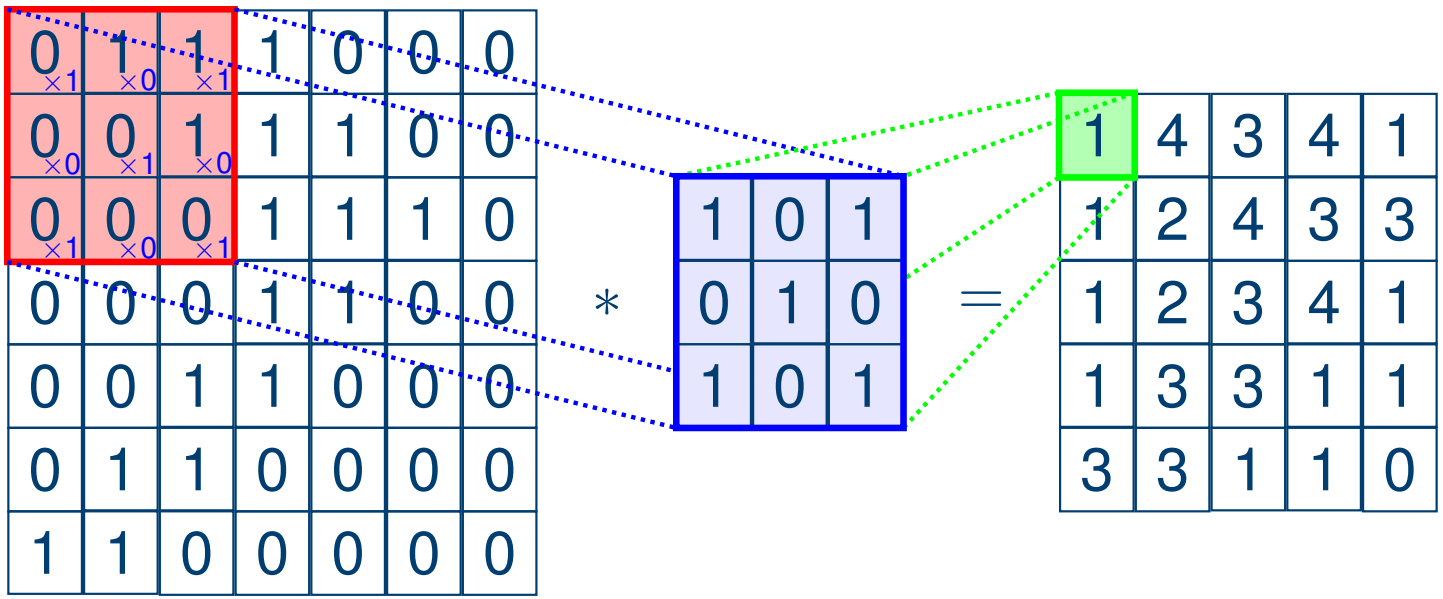
Enter the *convolution* operator

- Define a small (e.g. 3×3) matrix (the kernel, $\mathbf{K}$).
- Overlay it in all possible ways over the input image, $\mathbf{I}$.
- Record sums of elementwise products in a new image.

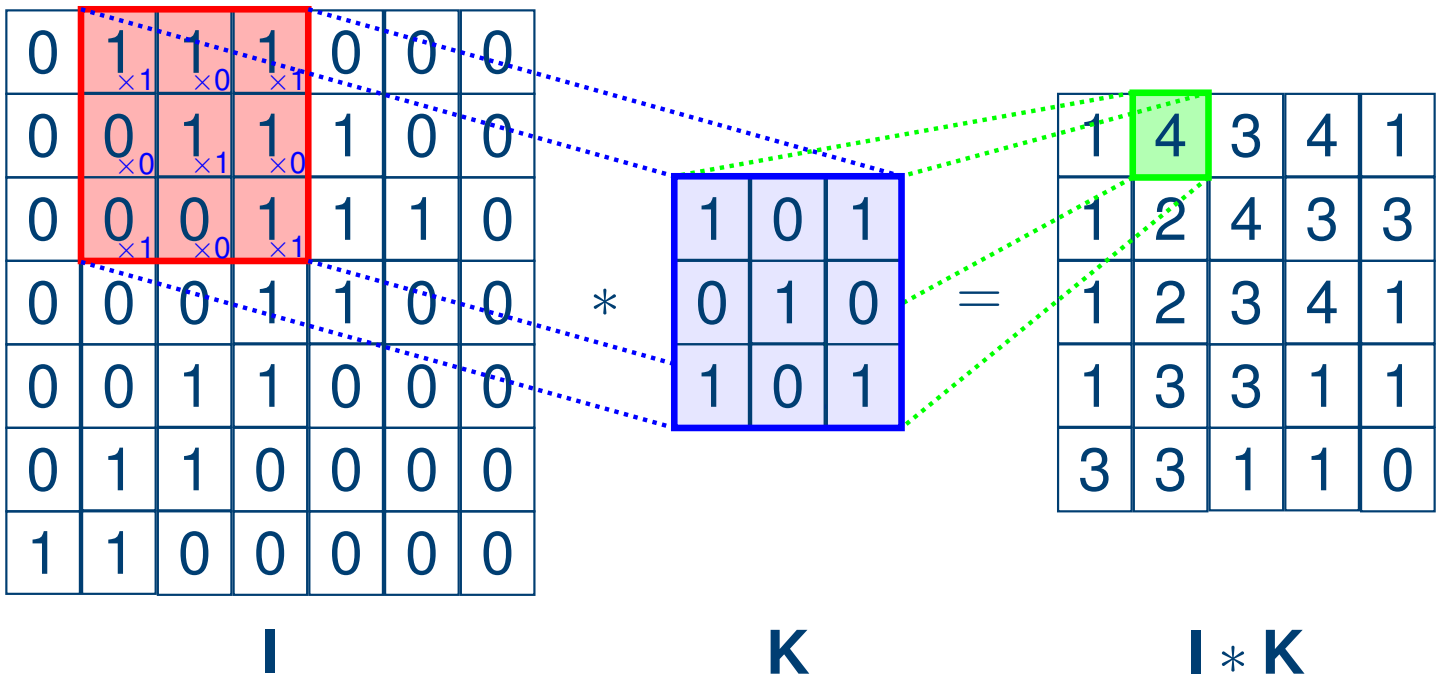
$$(\mathbf{I} * \mathbf{K})_{xy} = \sum_{i=1}^{h} \sum_{j=1}^{w} \mathbf{K}_{ij} \cdot \mathbf{I}_{x+i-1,y+j-1}$$

- This operator exploits structure—neighbouring pixels influence one another stronger than ones on opposite corners!
- Start with *random kernels*—and let the network find the optimal ones on its own!

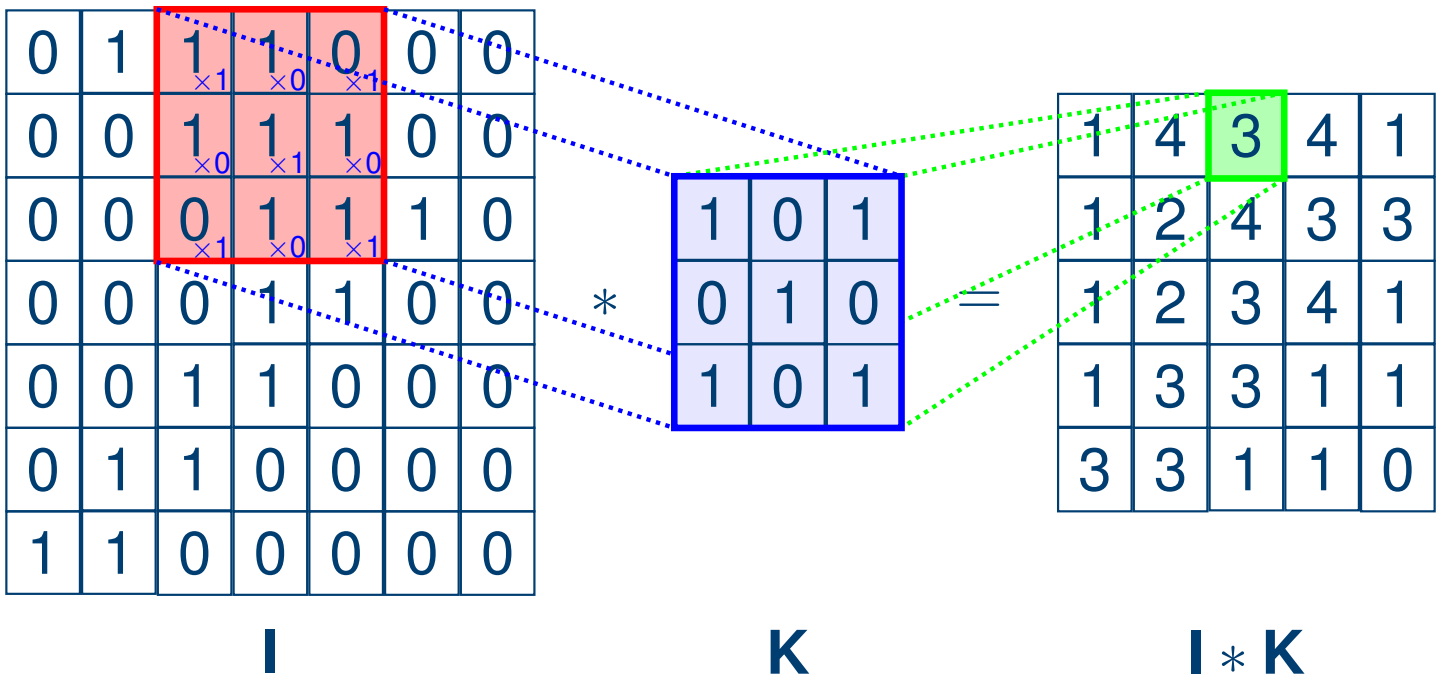
Convolution example



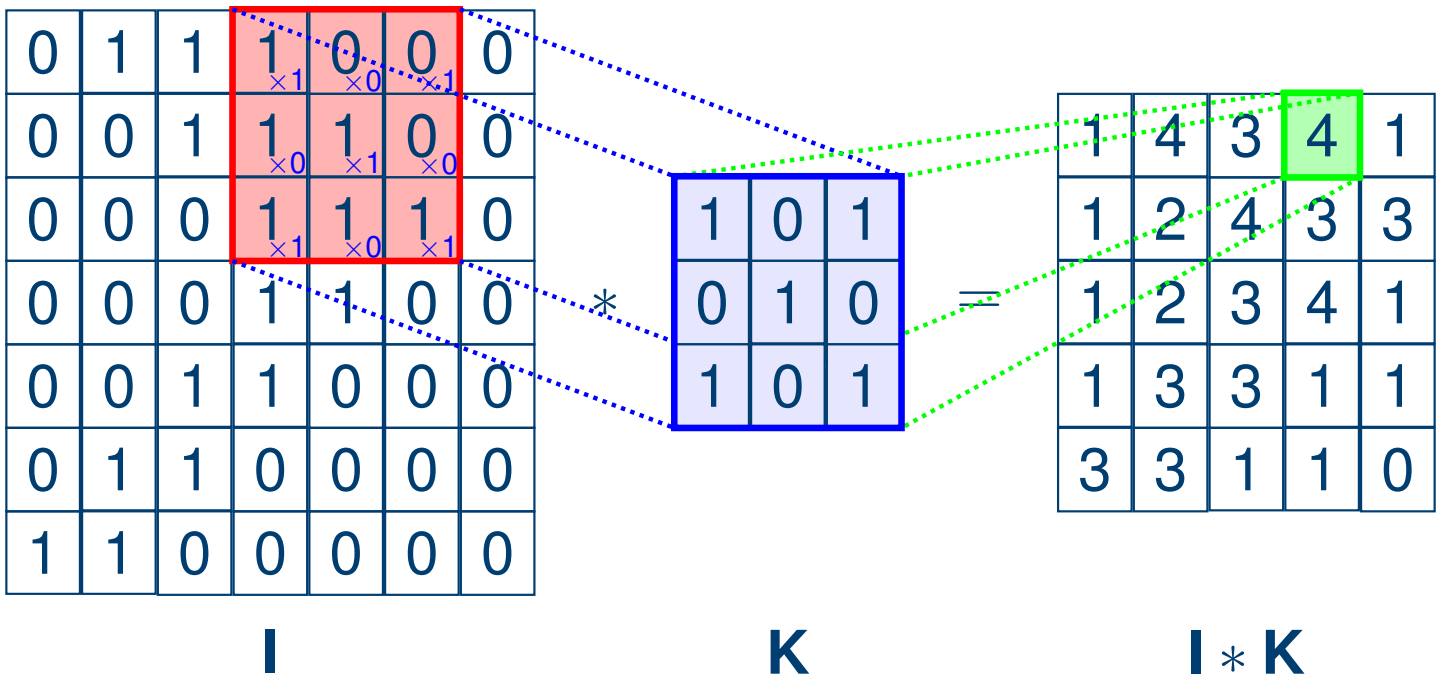
Convolution example



Convolution example

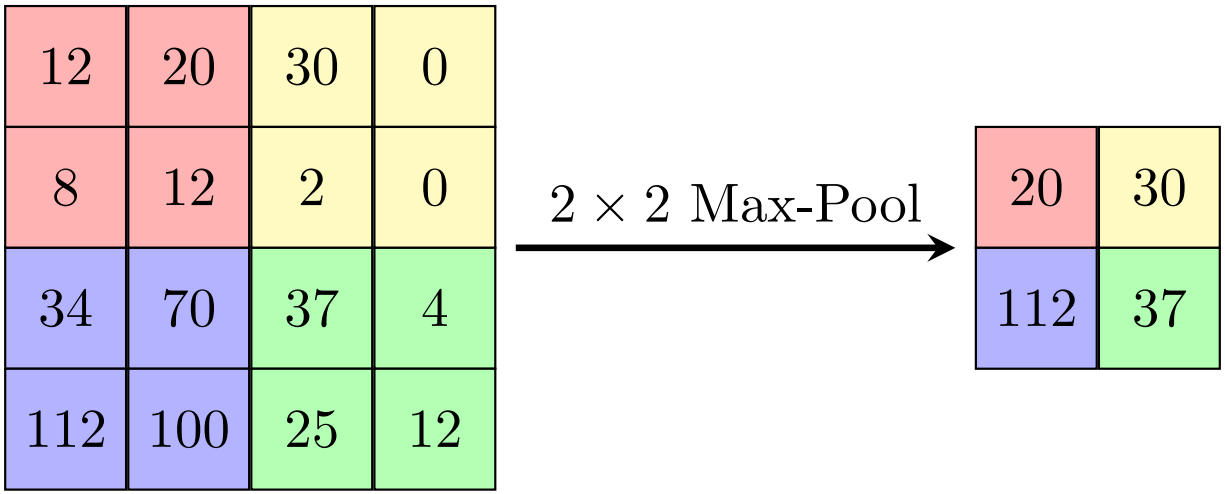


Convolution example



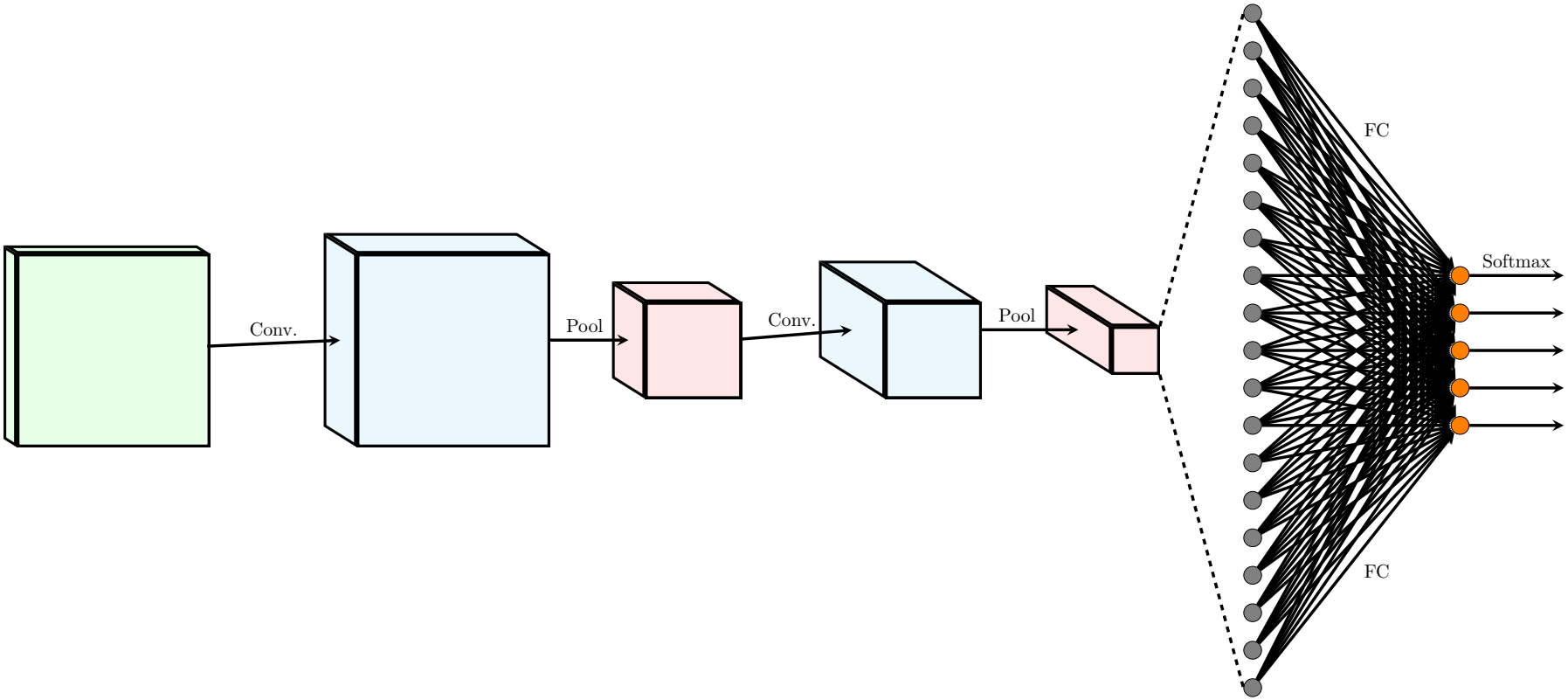
Downsampling ($\sim$ *max-pooling*)

Convolutions *light up* when they detect a particular feature in a region of the image. Therefore, when downsampling, it is a good idea to preserve maximally activated parts. This is the inspiration behind the *max-pooling* operation.



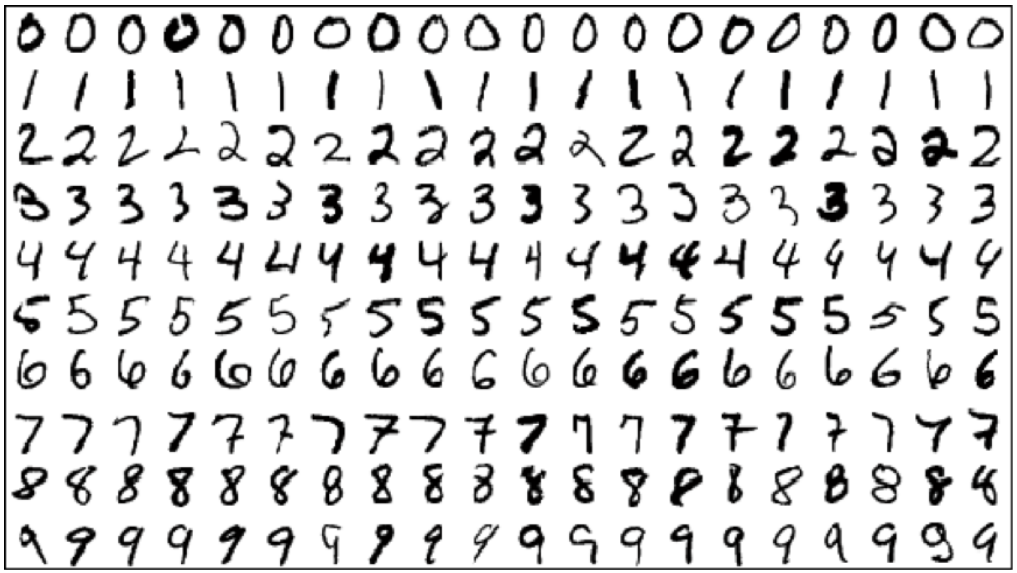
Stacking convolutions and poolings

Rough rule of thumb: increase the *depth* (number of convolutions) as the *height* and *width* decrease.

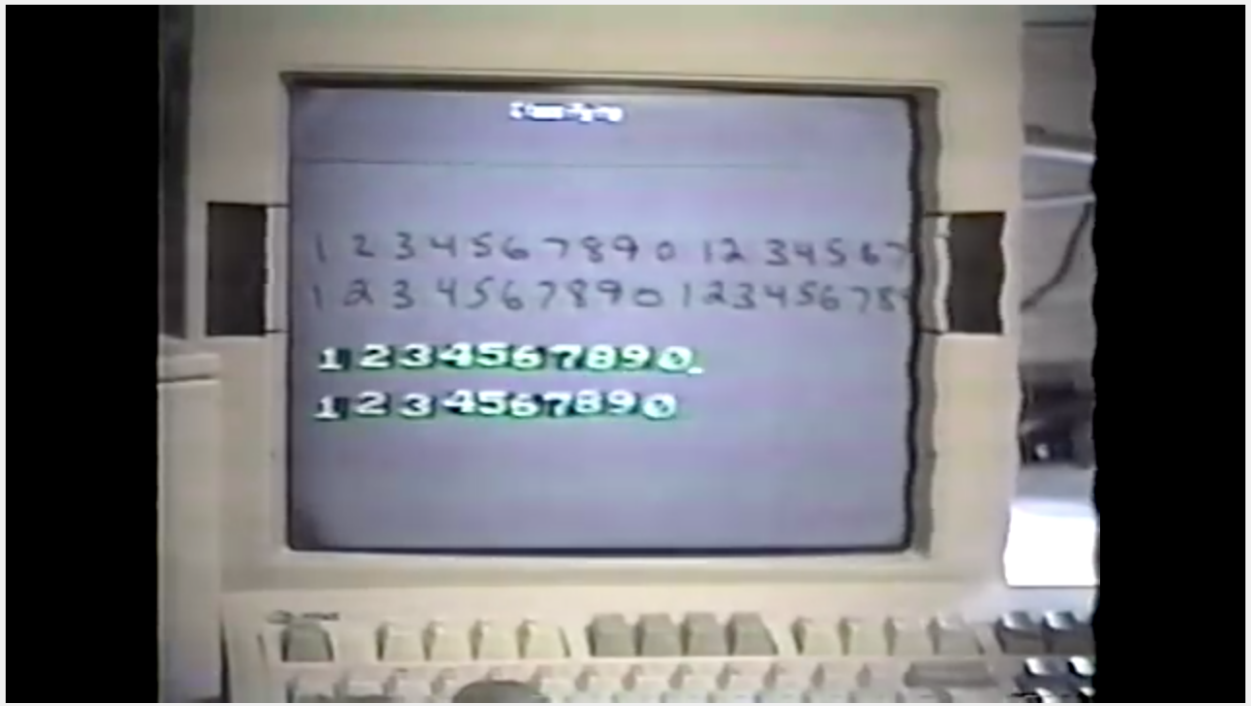


CNN representations

- Convolutional neural networks are by no means a *new* idea. . . they've been known since the late 1970s!
- Popularised by LeCun *et al.* in 1989 to classify handwritten digits (the MNIST dataset, now a standard benchmark).



LeNet-1



Convolutional Network Demo from 1993

Yann LeCun

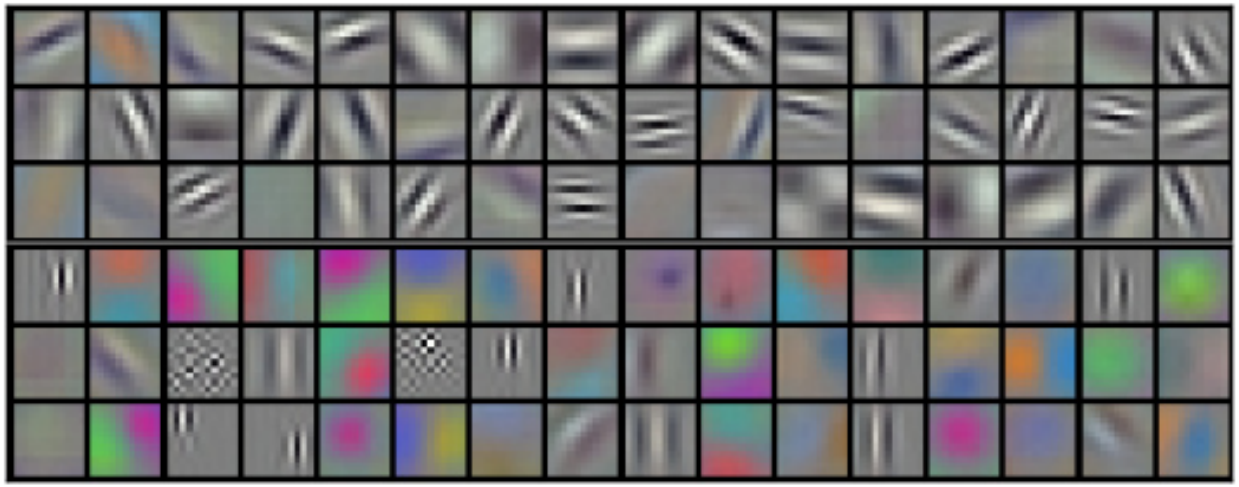
Subscribe 626

42,259 views

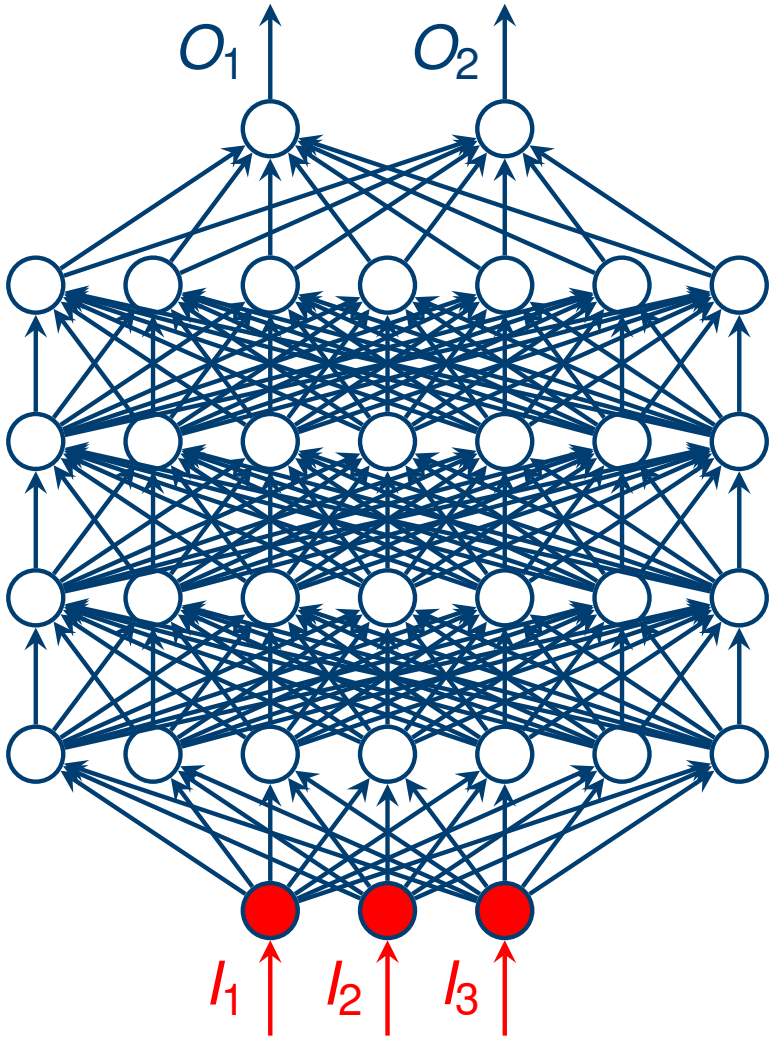
Add to Share More 256 1

Observing kernels

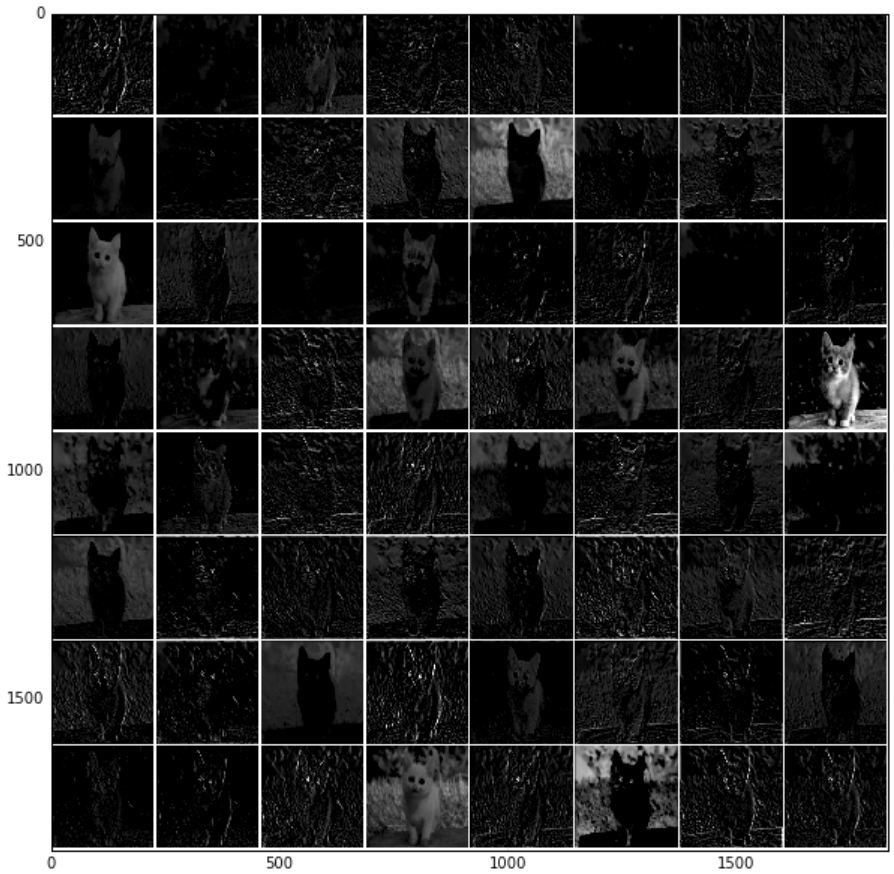
- Typically, as the kernels are small, gaining useful information from them becomes difficult already *past the first layer*.
- However, the first layer of kernels reveals something *magical*... In almost all cases, these kernels will learn to become *edge detectors*!

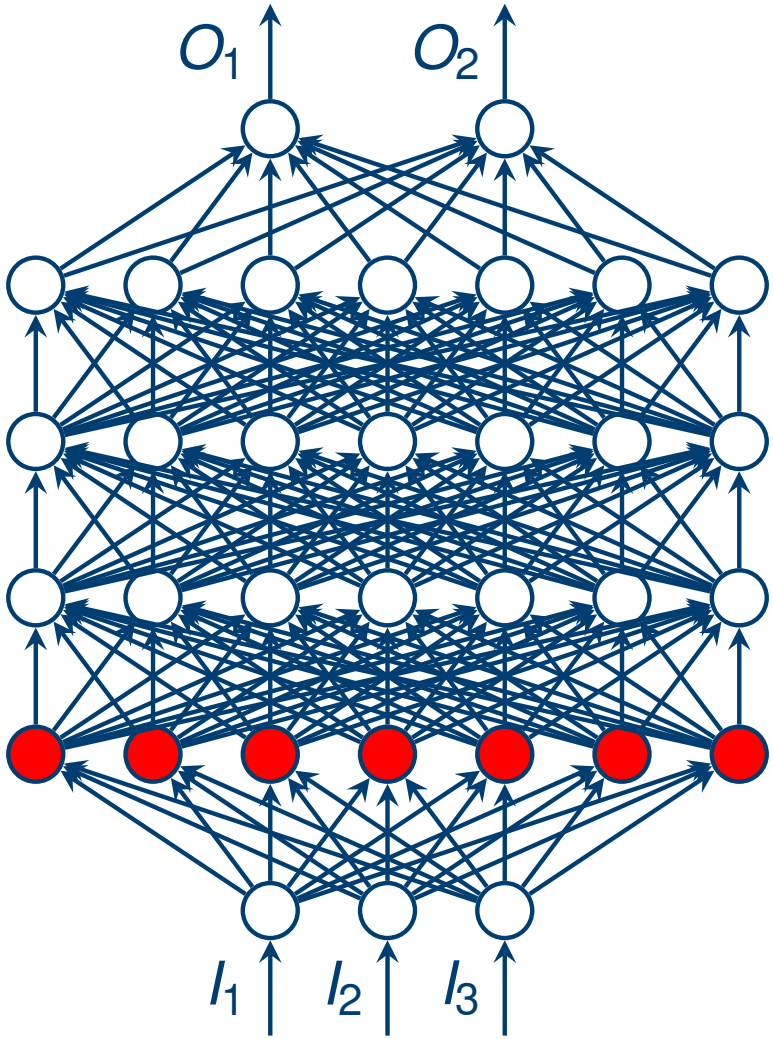


Passing data through the network: *Input*

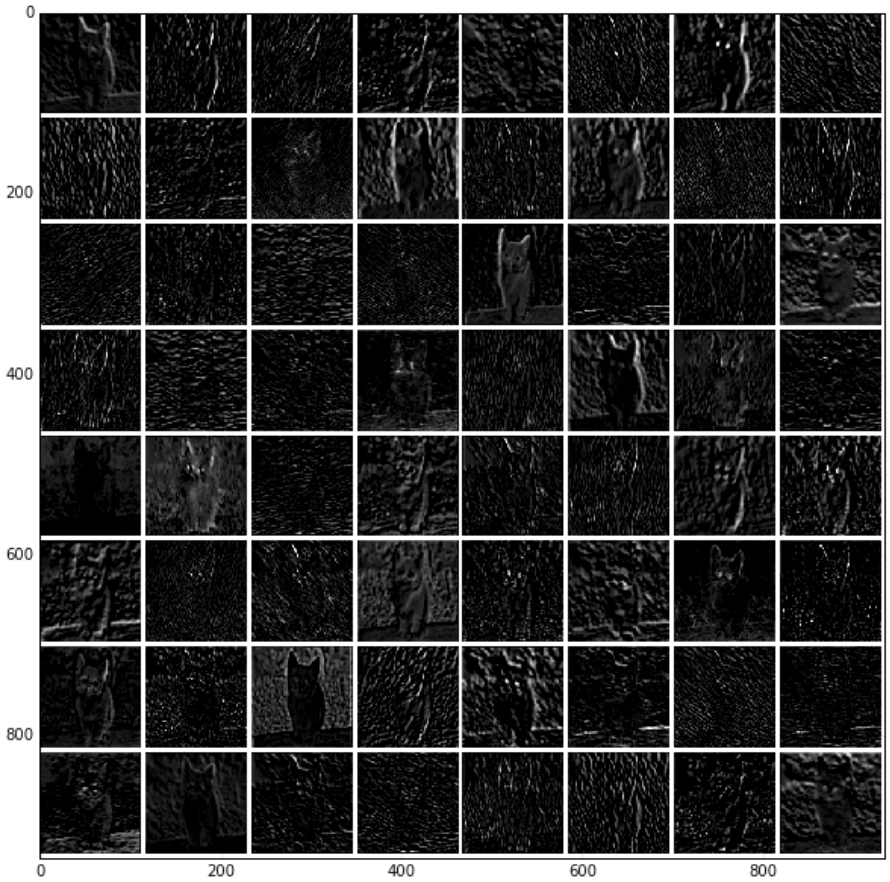


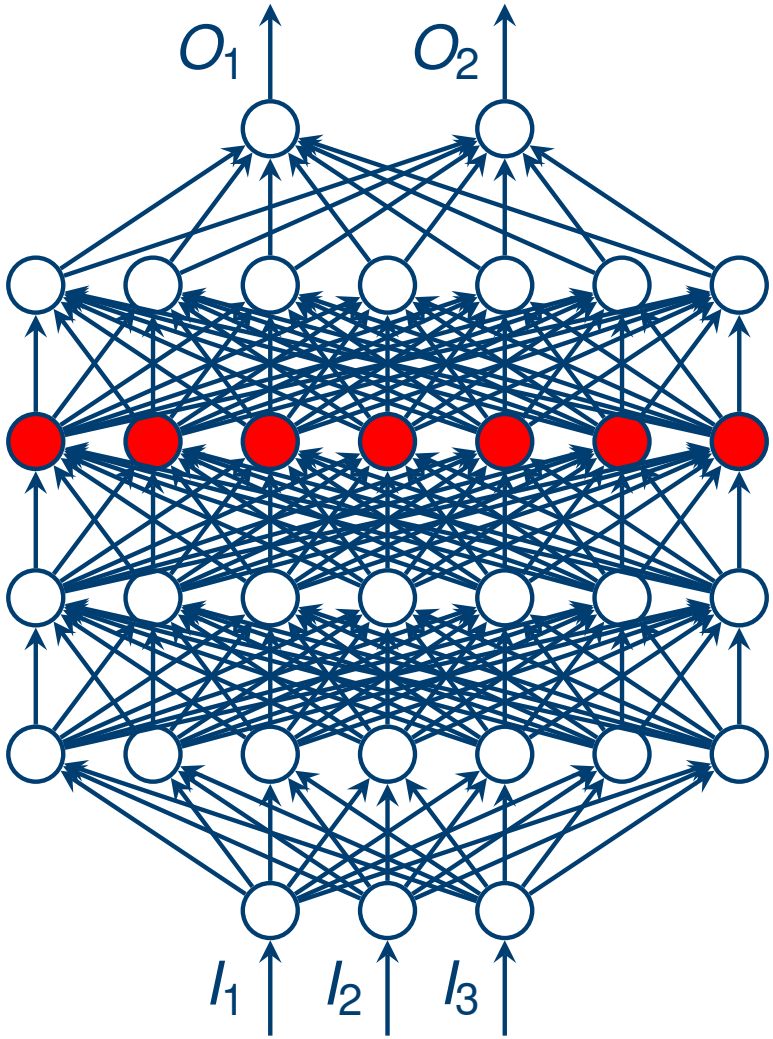
Passing data through the network: *Shallow layer*



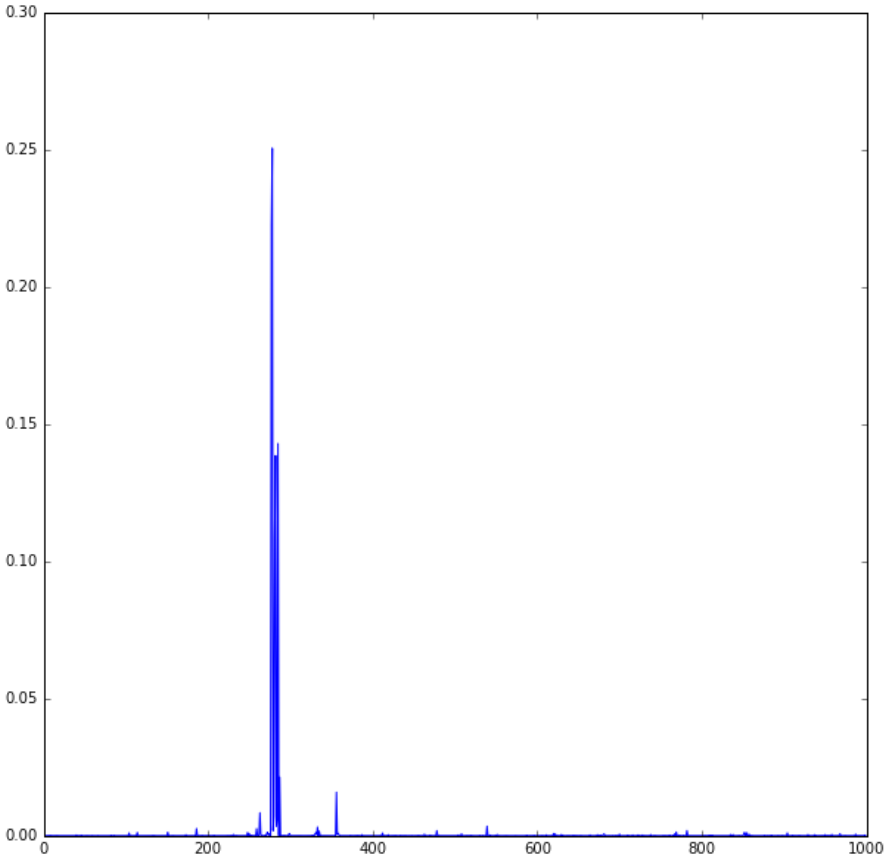


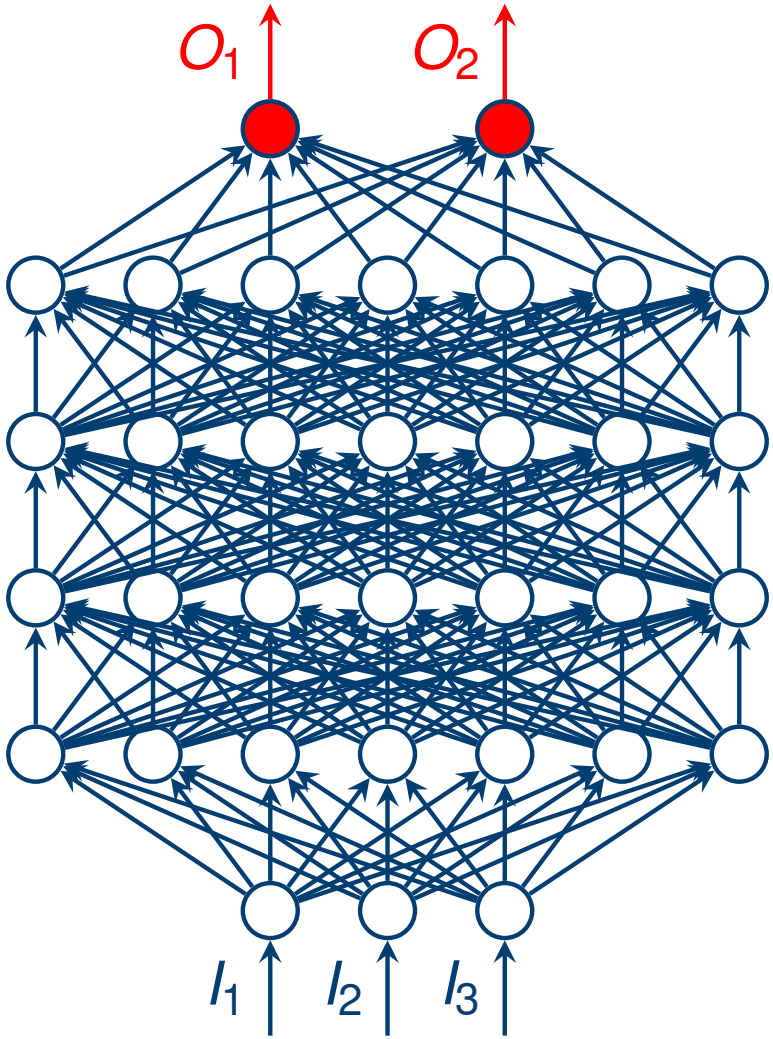
Passing data through the network: *Deep layer*





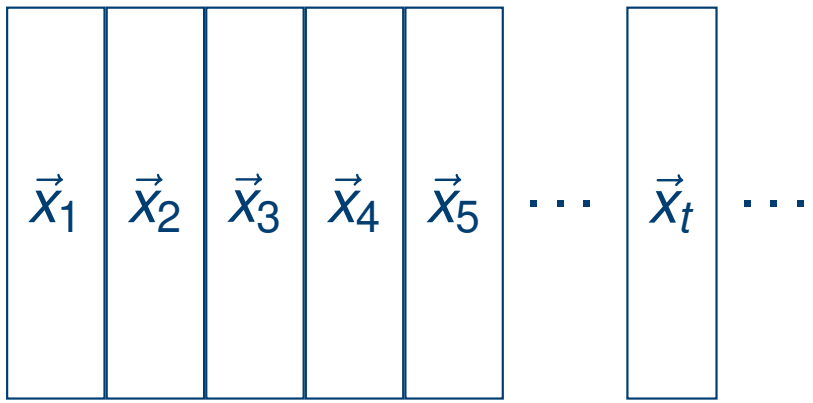
Passing data through the network: *Output*





Sequential inputs

- Now, consider a classification problem where the input is *sequential*—a sequence consisting of arbitrarily many *steps*, wherein at each step we have *n features*.
- This kind of input corresponds nicely to problems involving *sound* or *natural language*.



- The fully connected layers will no longer even *work*, as they expect a *fixed-size* input!

Making it work

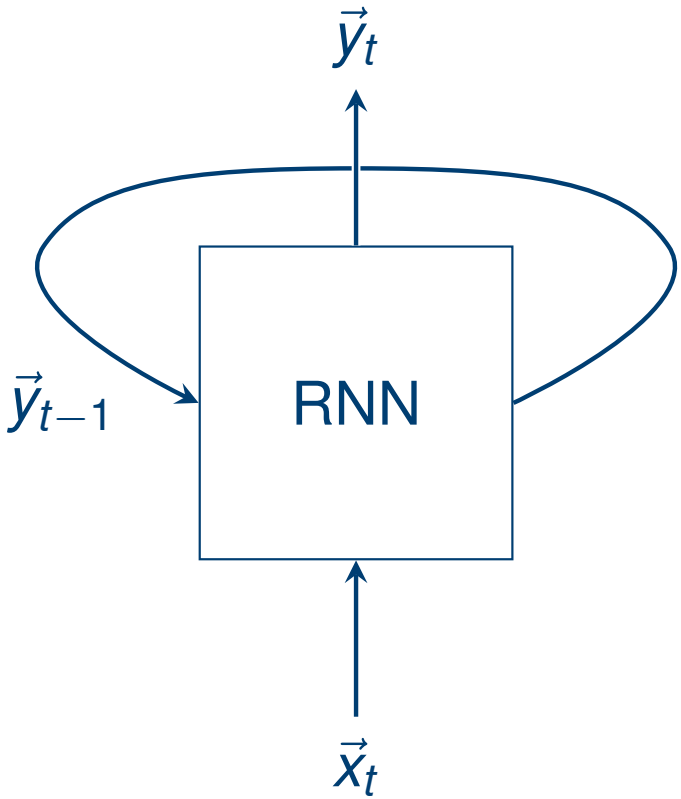
Key ideas:

- *Summarize* the entire input into m features (describing the most important patterns for classifying it);
- Exploit relations between *adjacent steps*—process the input in a *step-by-step* manner, iteratively building up the features, $\vec{h}$:

$$\vec{h}_t = f(\vec{x}_t, \vec{h}_{t-1})$$

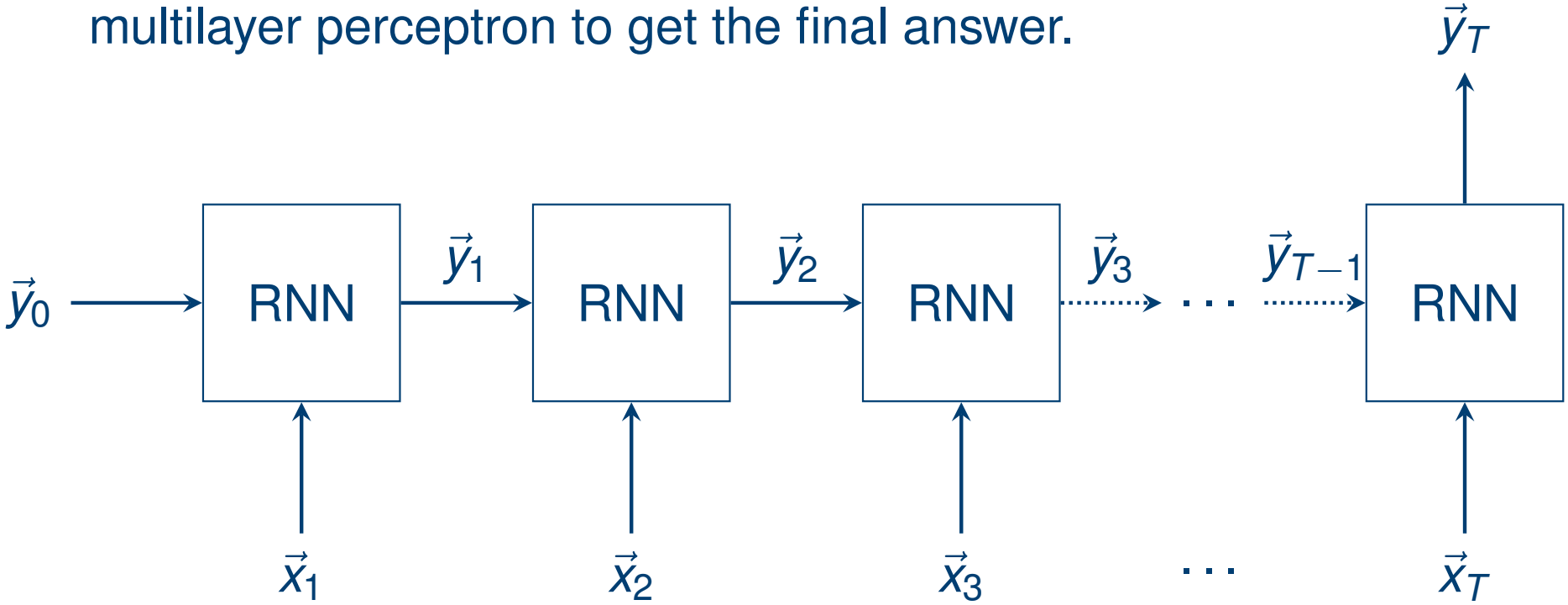
- If we declare a pattern to be *interesting*, then it does not matter **when** it occurs in the sequence $\implies$ employ *weight sharing*!

An RNN cell



Unrolling the cell. . .

Compute $\vec{y}_T$ iteratively, then feed it into the usual multilayer perceptron to get the final answer.

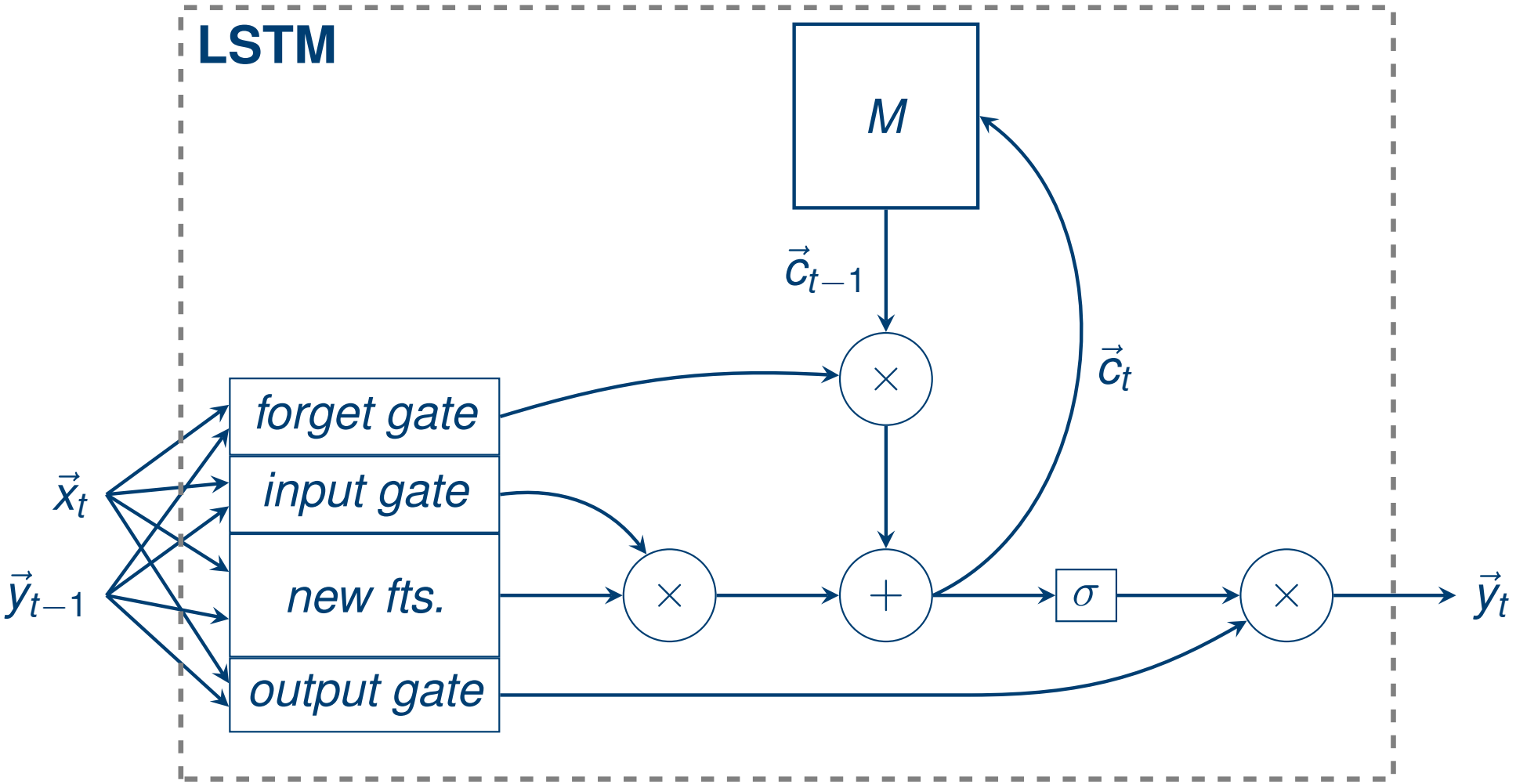


N.B. Every RNN block has the *same* parameters!

RNN variants

- Initial versions (SimpleRNN) introduced by Jordan (1986), Elman (1990). Simply apply a fully-connected layer on both $\vec{x}_t$ and $\vec{y}_{t-1}$, and apply an activation.
- Suffers from *vanishing gradients* over long paths (as $\sigma'(x) < 1$)...cannot capture *long-term dependencies*!
- The problem is solved by the *long short-term memory* (LSTM) model (Hochreiter and Schmidhuber, 1997). The LSTM cell explicitly learns (from data!) the proportion by which it *forgets* the result of its previous computations.
- Several models proposed since then, but none improve significantly on the LSTM on average.

An LSTM block



Where are we?

- By now, we've covered all the essential architectures that are used across the board for modern deep learning...
- ...and we're not even in the 21st century yet!
- It turns out that we had the required methodology all along, but several key factors were missing...
  - *demanding gamers* ($\sim$ GPU development!)
  - *big companies* ($\sim$ lots of resources, and *DL frameworks*!)
  - *big data* ($\sim$ for big parameter spaces!)
- So, what's exactly **new**, *methodologically*?

Improving gradient descent

- In the past, gradient descent was performed in a *batch* fashion, by accumulating gradients over the entire training set:

$$\vec{w} \leftarrow \vec{w} - \eta \sum_{p=1}^{m} \nabla \mathcal{L}_p(\vec{w})$$

  For big datasets, this may not only be remarkably expensive for just one step, but may not even fit within memory constraints!

Stochastic gradient descent (SGD)

- This problem may be solved by using *stochastic gradient descent*—at one iteration consider *only one* (randomly chosen) training example $(x_{p'}, y_{p'})$:

$$\vec{w} \leftarrow \vec{w} - \eta \nabla \mathcal{L}_{p'}(\vec{w})$$

- In practice, we use the "golden middle"—consider a randomly chosen *minibatch* of examples, $\mathcal{B} = \{(x_1, y_1), \ldots, (x_{bs}, y_{bs})\}$.

$$\vec{w} \leftarrow \vec{w} - \eta \sum_{p=1}^{bs} \nabla \mathcal{L}_p(\vec{w})$$

- We also have efficent methods to automatically estimate the optimal choice of η (such as *Adam* and *RMSProp*).

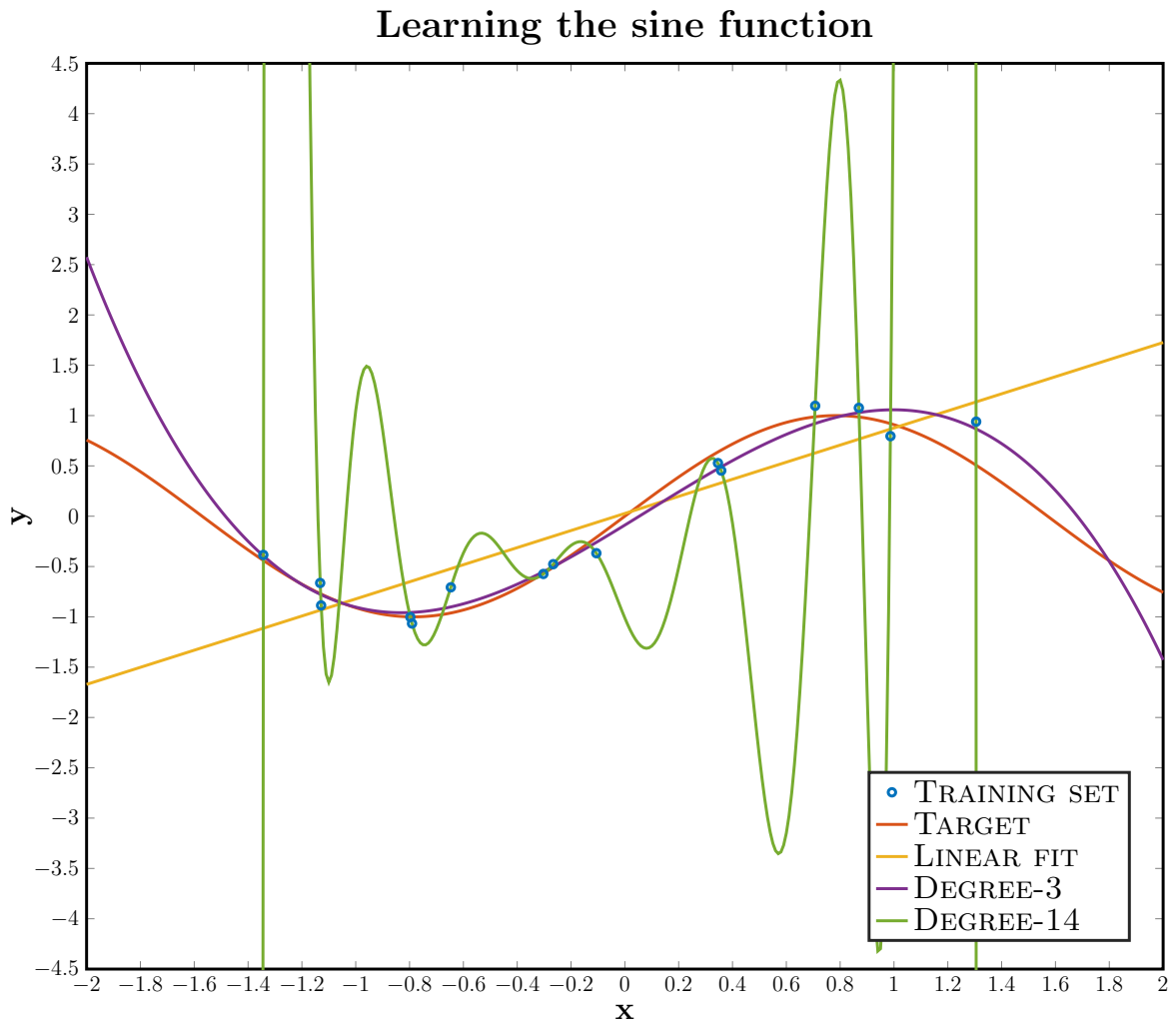
Vanishing gradients strike back

- As networks get deeper, the same kind of vanishing gradient problem that we used LSTMs to solve appears with non-recurrent networks.
- We cannot really re-use ideas from LSTMs here cleanly...
- Solution: change the activation function! The rectified linear unit (ReLU) is the simplest nonlinearity not to suffer from this problem:

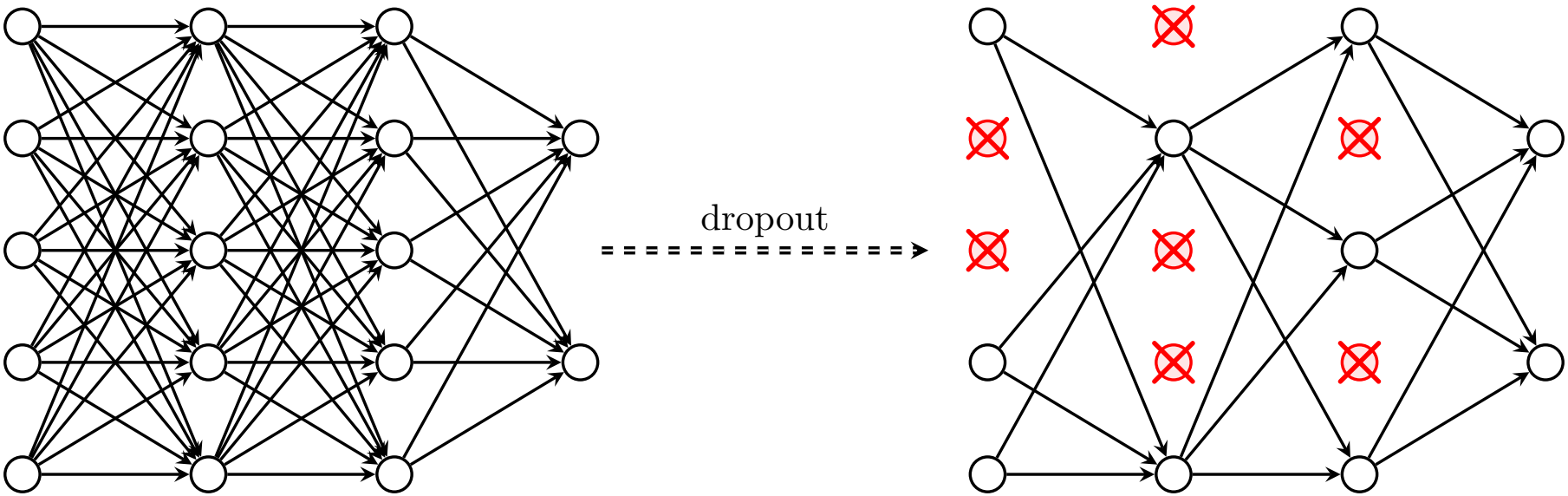
$$ReLU(x) = \max(0, x)$$

  as its gradient is exactly 1 when "active" and 0 when "dead".
- First deployed by Nair and Hinton in 2010, now *ubiquitous* across deep learning architectures.

Better *regularisation*

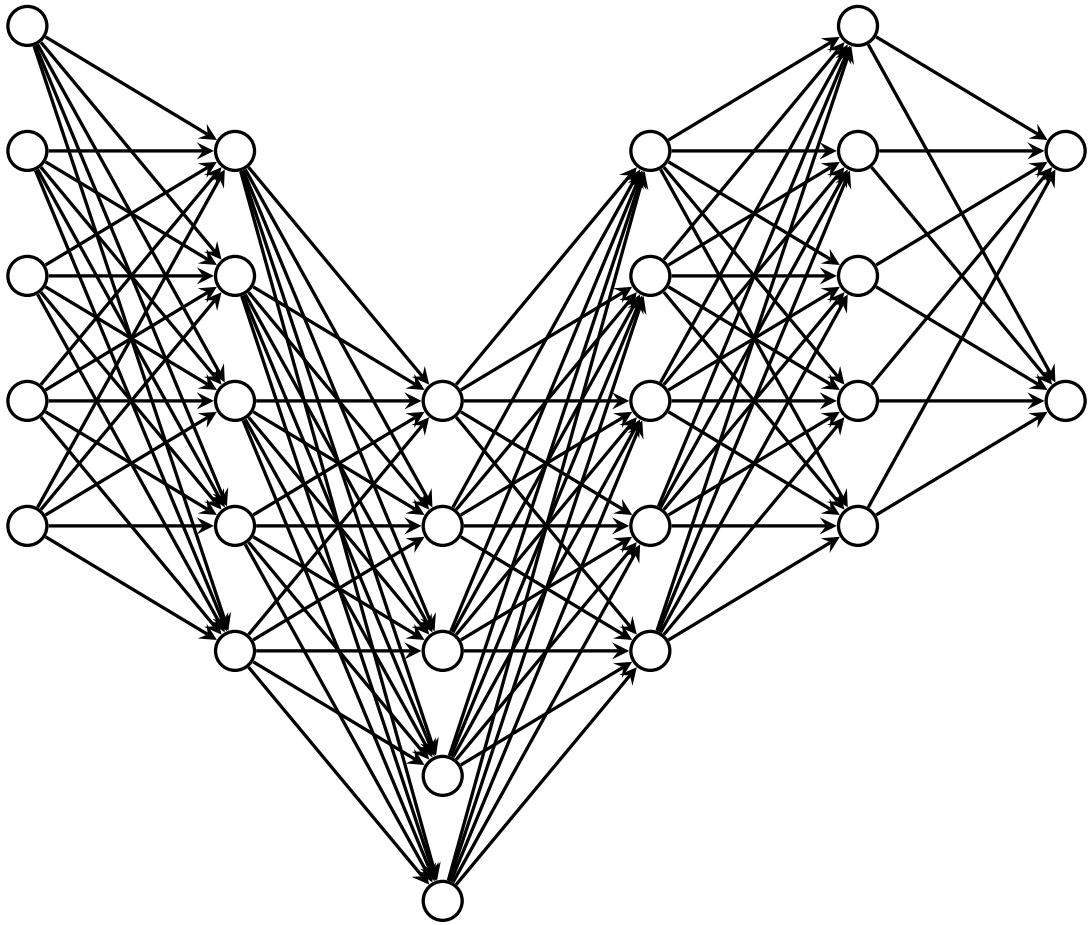


Dropout



- Randomly “kill” each neuron in a layer with probability p during training only. . . ?!

Batch normalisation



- “Internal covariance shift”. . . ?!

Batch normalisation

- Solution: *renormalise* outputs of the current layer across the current batch, $\mathcal{B} = \{x_1, \ldots, x_m\}$ (but allow the network to "revert" if necessary)!

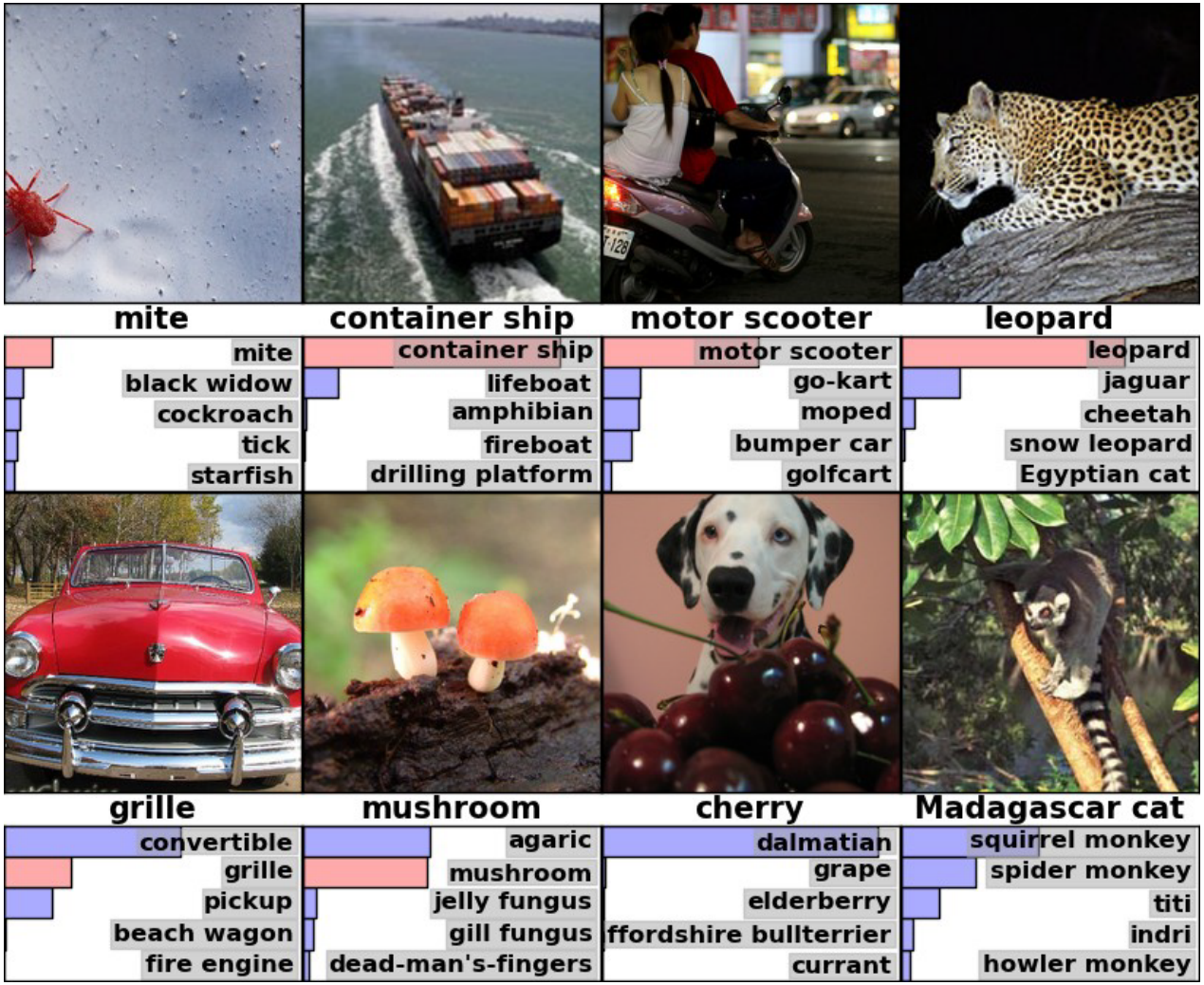
$$\mu_{\mathcal{B}} = \frac{1}{m}\sum_{i=1}^{m} x_i \quad \sigma^2_{\mathcal{B}} = \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2$$

$$\hat{x}_i = \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \varepsilon}} \qquad y_i = \gamma\hat{x}_i + \beta$$

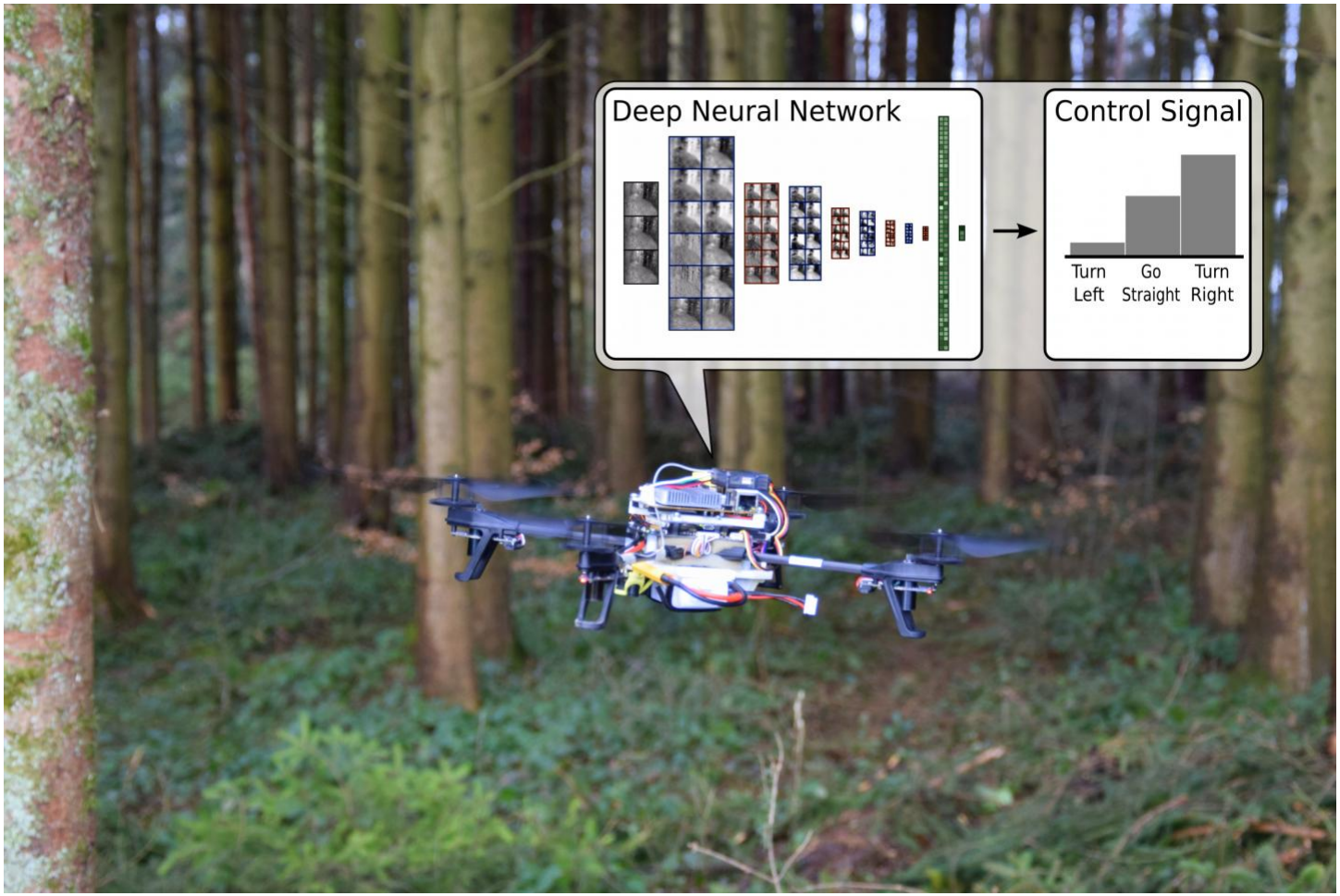
  where γ and β are *trainable*!
- Now ubiquitously used across deeper networks
  - Published in February 2015, $\sim$ 1500 citations by now!

Traffic sign classification

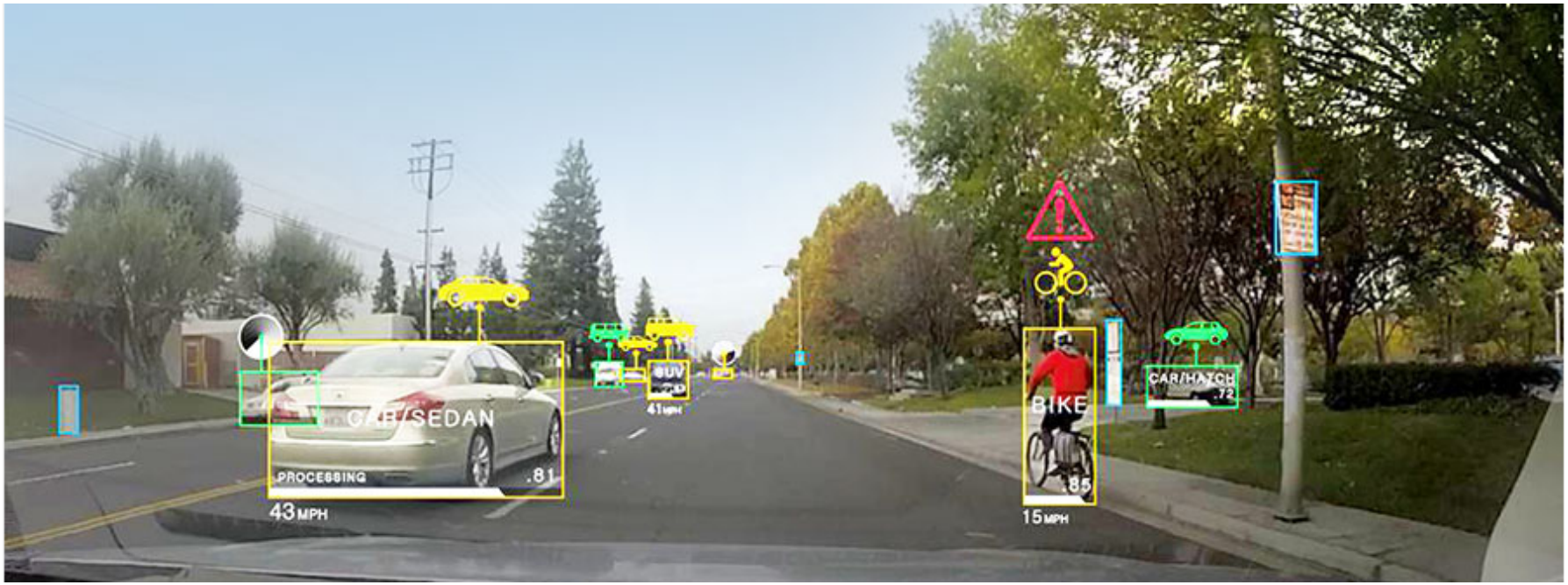
ImageNet classification



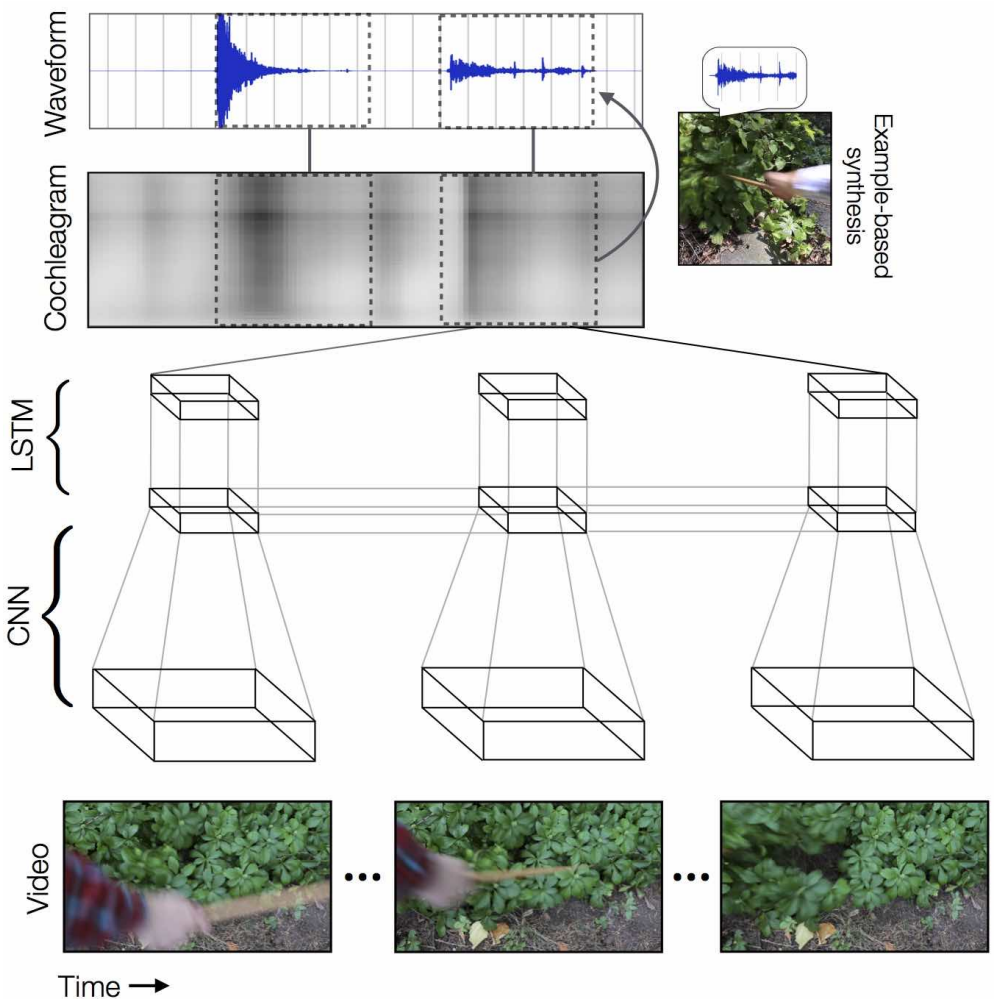
“Simple” navigation



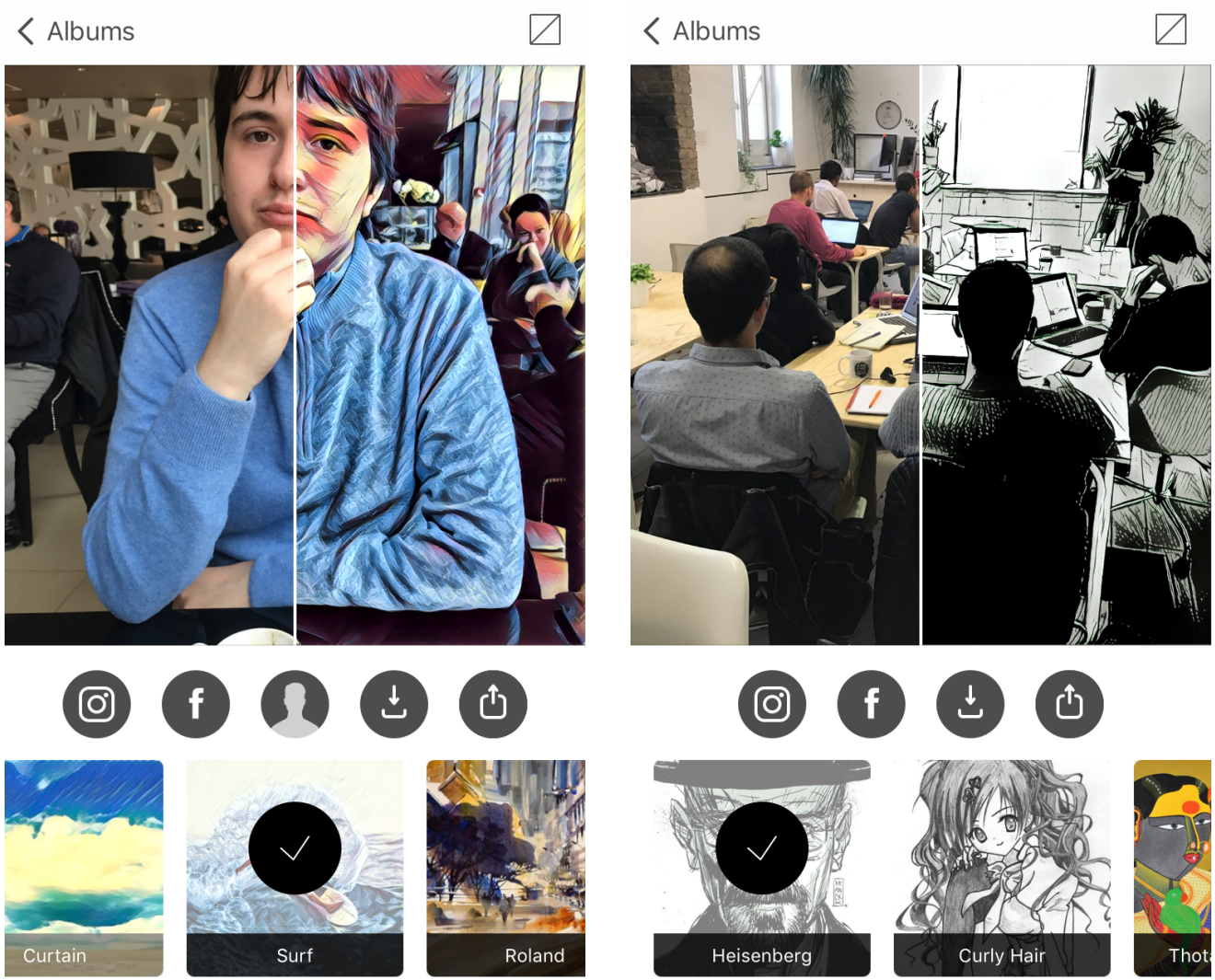
Self-driving cars



“Greatest hits”

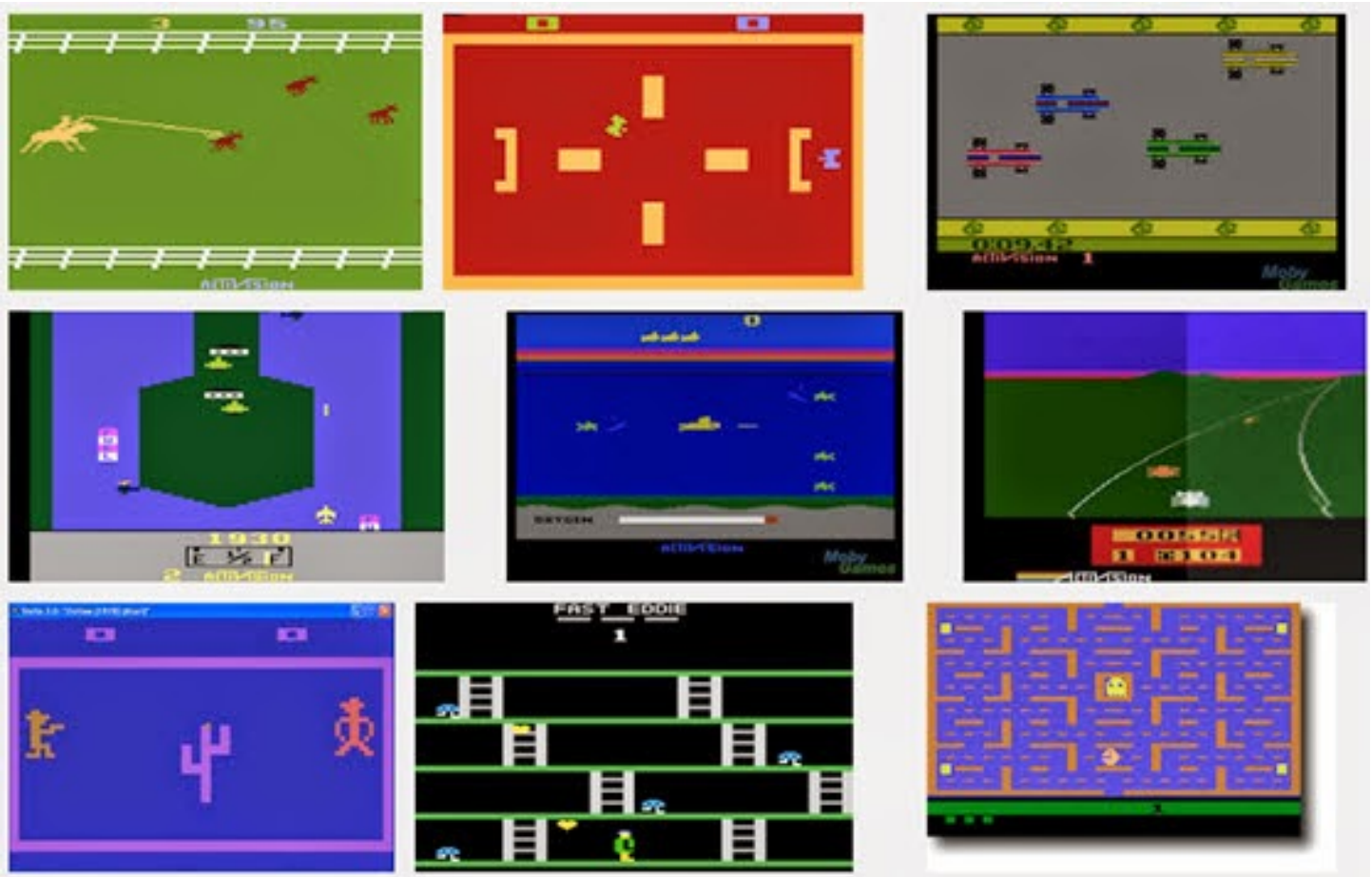


Neural style transfer

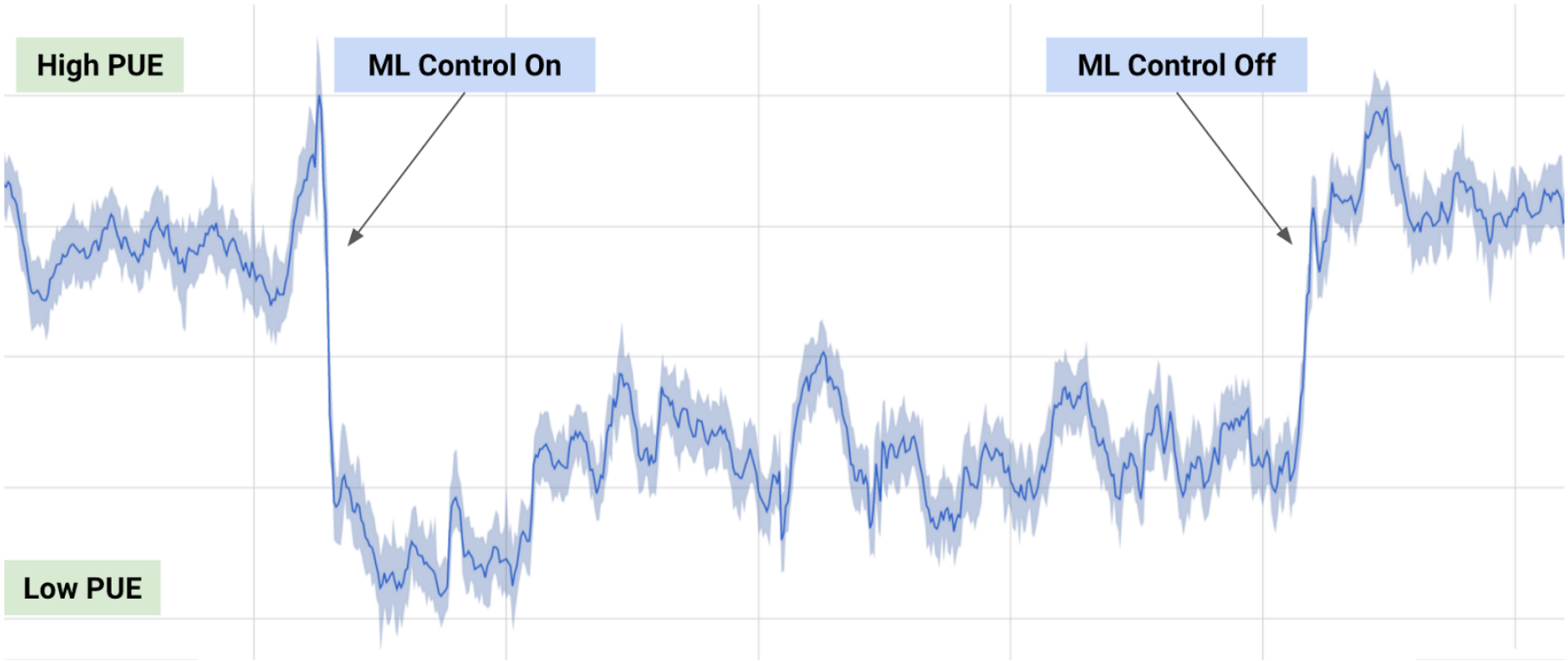


Video captioning

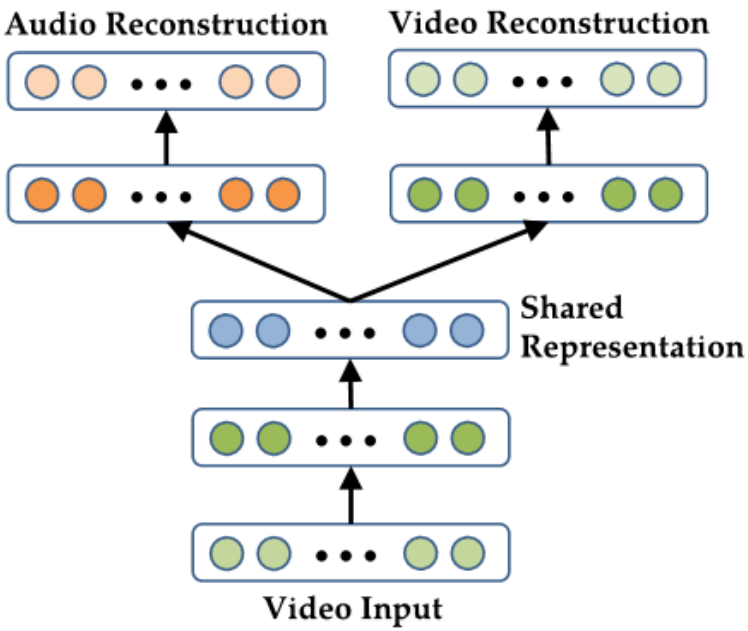
Playing Atari. . .



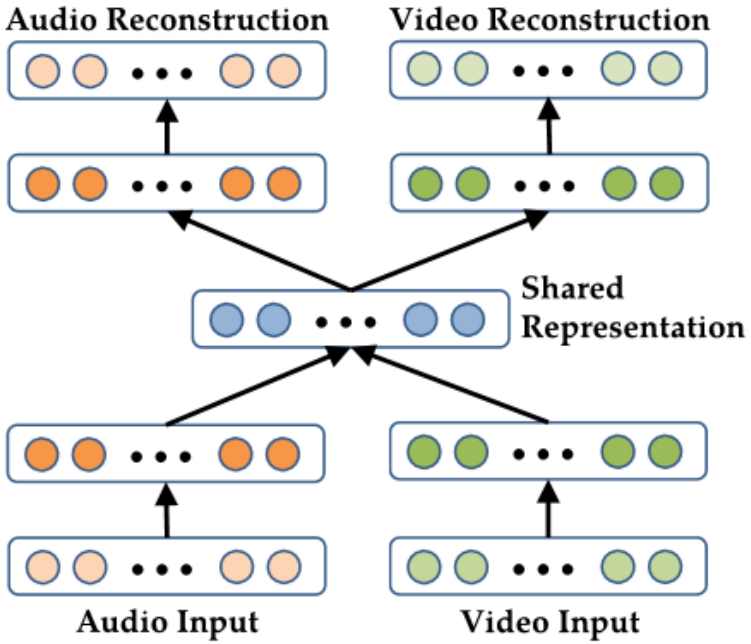
. . . and Cooling



Challenge: *Multimodal data fusion*

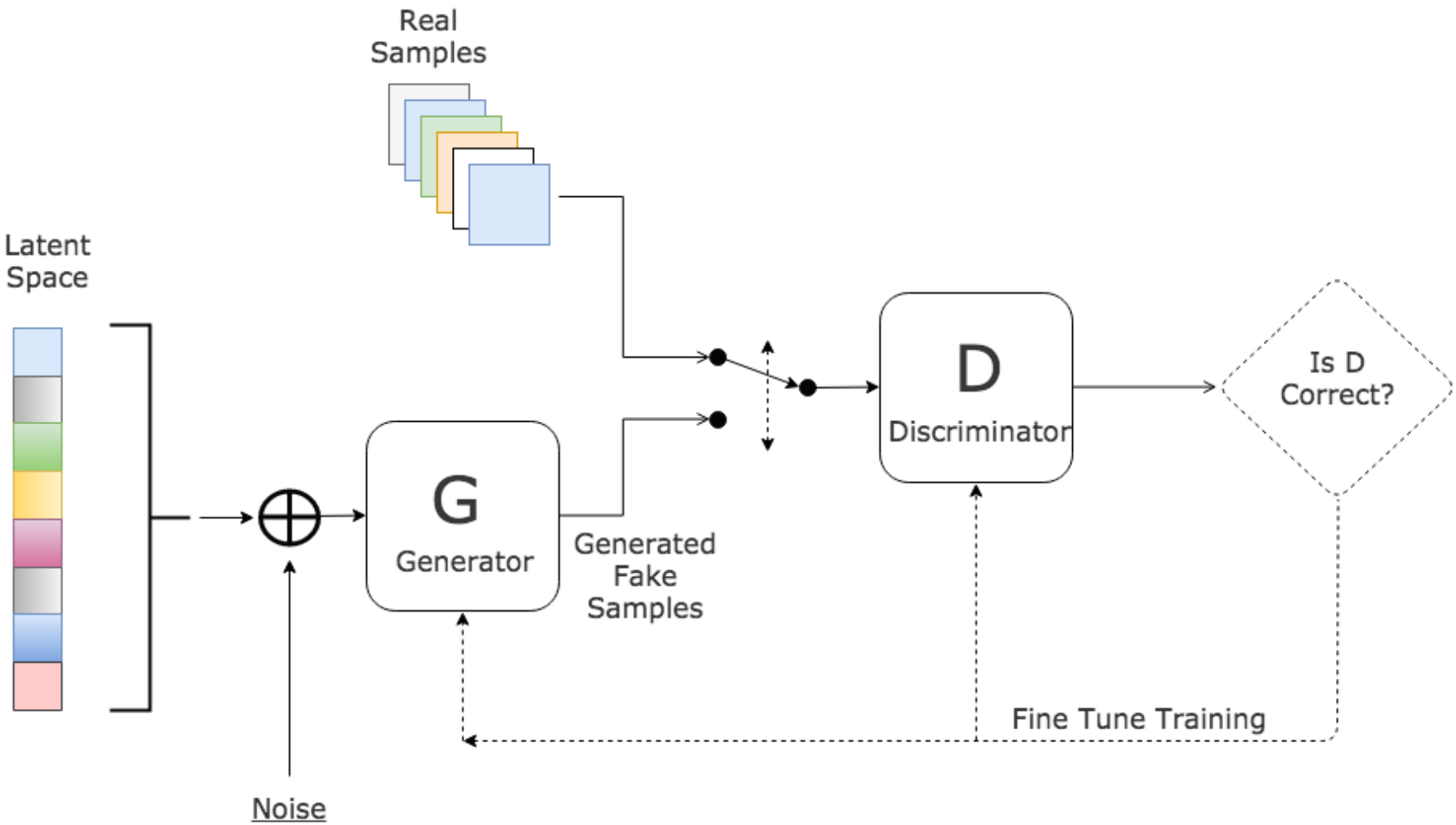


(a) Video-Only Deep Autoencoder

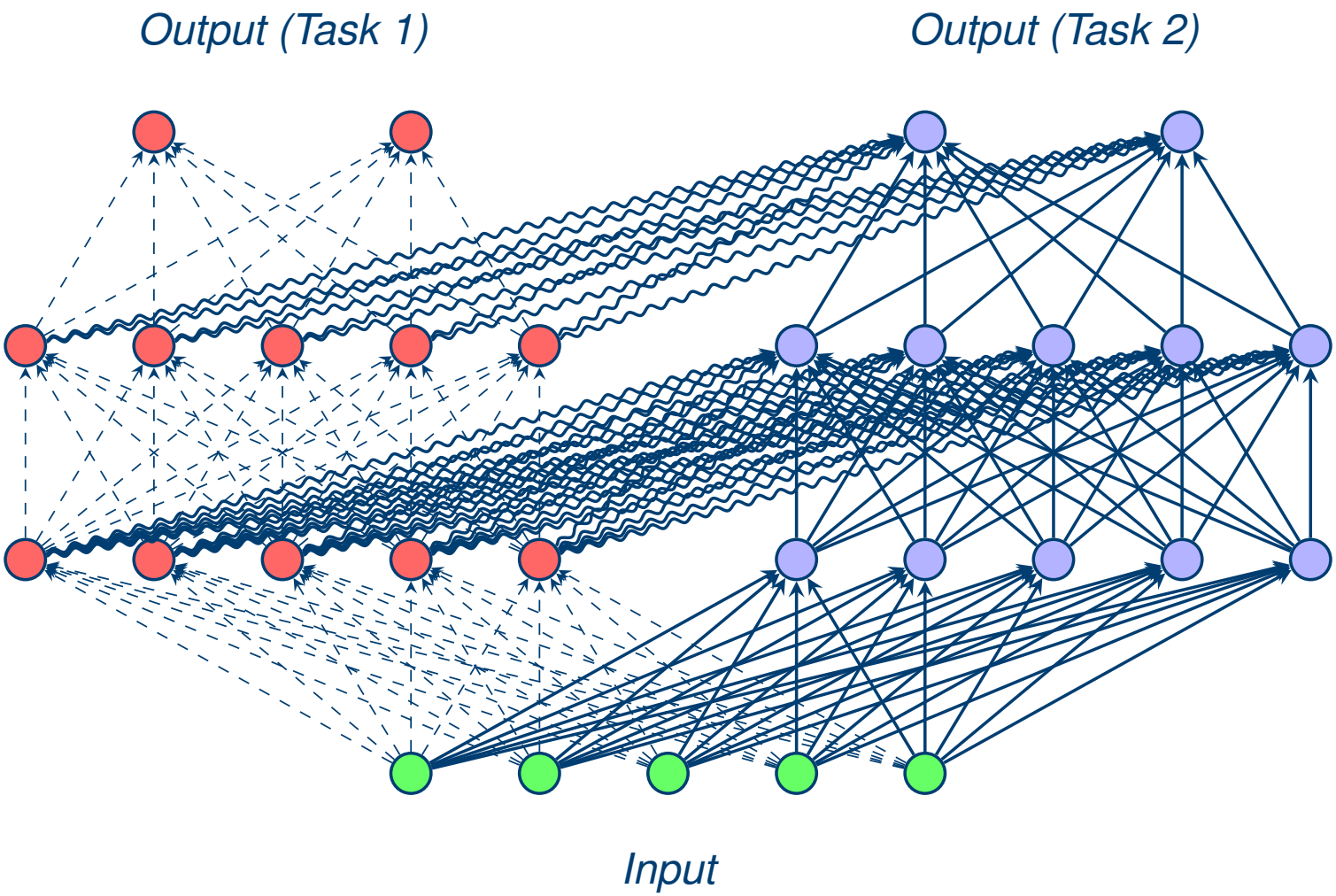


(b) Bimodal Deep Autoencoder

Challenge: *Unsupervised learning*



Challenge: *General AI*



Thank you!

Questions?

petar.velickovic@cst.cam.ac.uk