Convolutional Neural Networks & Recurrent Neural Networks For Cybersecurity

A. Shahina
Professor Dept. IT
SSN College of Engineering

CNN for Computer Vision

Organization of this talk:

- Introduction to deep learning & their applications in cybersecurity
- Convolutional Neural Networks
- A case study with CNN for cybersecurity
- Recurrent Neural Neworks
- A case study with RNN for cybersecurity

Deep Learning and Cybersecurity

Intrusion Detection and Prevention Systems

Malware detection -Deep learning algorithms are capable of detecting more advanced threats and are not reliant on remembering known signatures and common attack patterns.

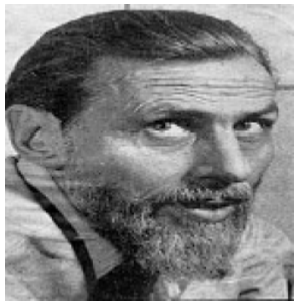
Spam and Social Engineering Detection

Network Traffic Analysis for malicious activities

User Behavior Analytics for recognizing insider threats and employees using their legitimate access with malicious intent

Artificial Neural Networks: The Beginnings

W. S. McCulloch and W. Pitts (1943) Logical calculus of the ideas immanent in nervous activity. Philosophy of Science 10(1), 18-24.



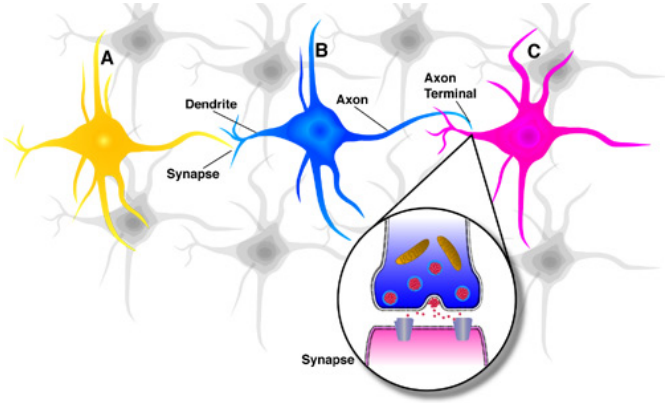
Warren McCulloch

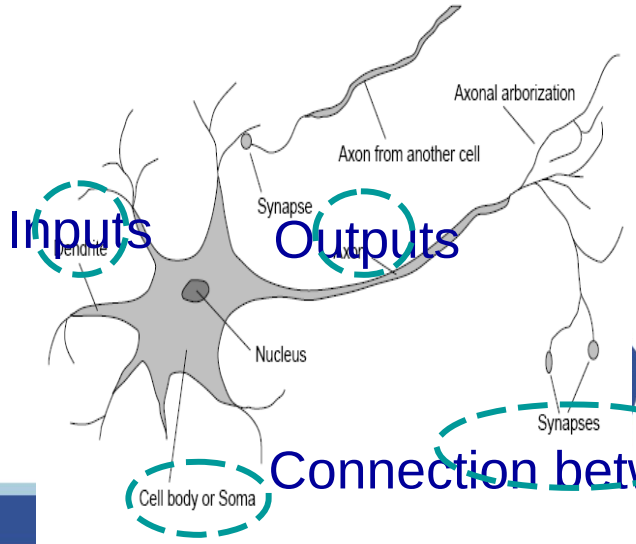
Walter Pitts

Revolutionary Idea: think of neural tissue as circuitry performing mathematical computations!

Biological Inspiration

Idea : To make the computer more robust, intelligent, and learn, …
Let's model our computer software (and/or hardware) after the brain

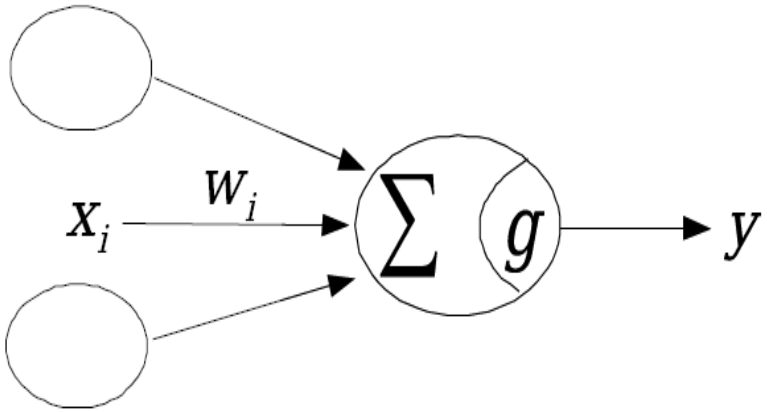




The McCulloch-Pitts Neuron

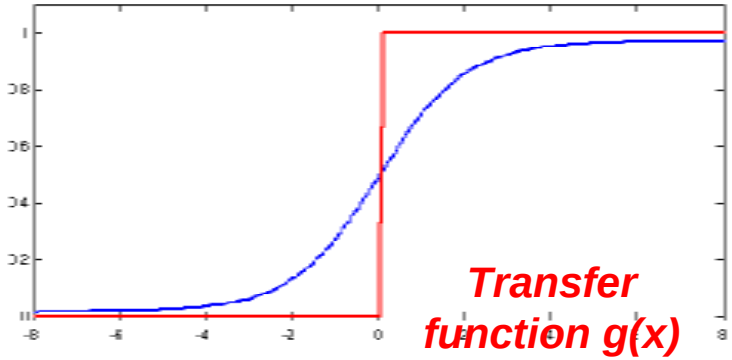
Linear weighted sum of inputs:

$$z = \sum_i w_i x_i$$



Nonlinear, possibly **stochastic** **transfer function**:

$$y = g(z)$$



Transfer function g(x)

Learning rule:

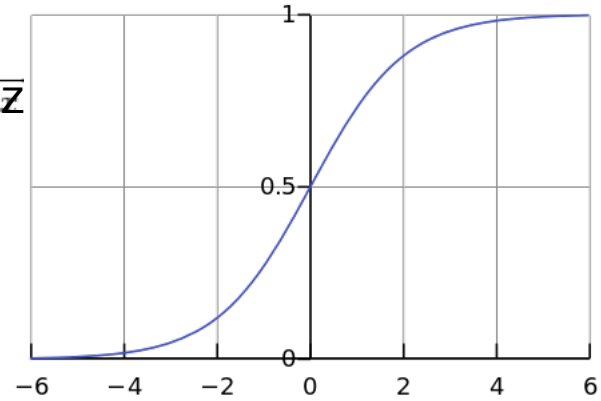
$$\Delta w_i = \cdots$$

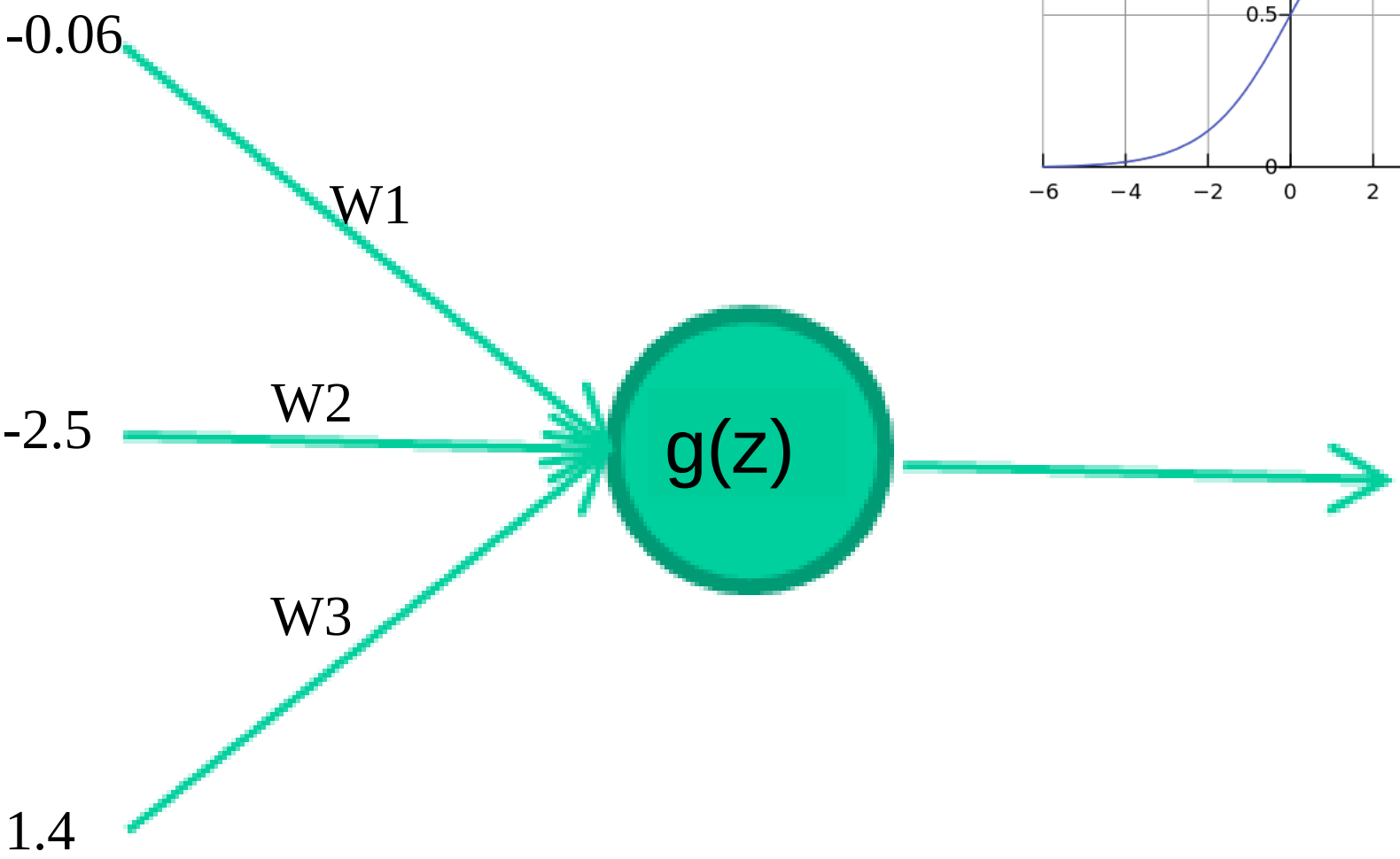
Only when sum exceeds the threshold limit will neuron fire

Weights can enhance or inhibit

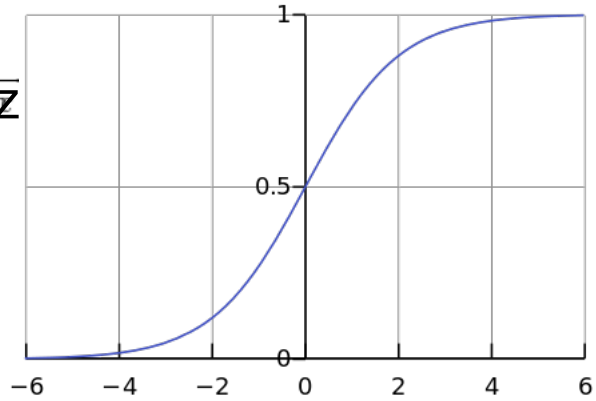
Collective behaviour of neurons is what's interesting for intelligent data processing

$$g(z) = \frac{1}{1+e^{-z}}$$





$g(z) = \frac{1}{1+e^{-z}}$



-0.06

2.7

-2.5

-8.6

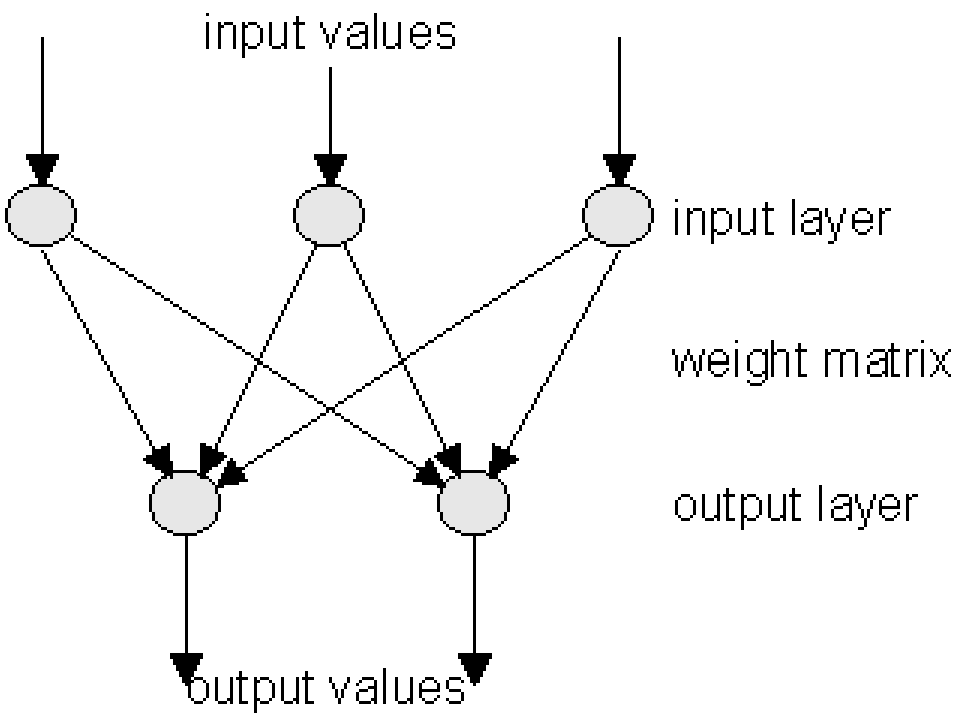
g(z)

0.002

1.4

$z = -0.06 \times 2.7 + 2.5 \times 8.6 + 1.4 \times 0.002 = 21.34$

Perceptron Structure



Learning for Perceptron

1. Initialize w_{ij} with random values
2. Repeat until $w_{ij}(t + 1) \approx w_{ij}(t)$:
   - Pick pattern p from training set
   - Feed input to network and calculate the output
   - Update the weights according to

$$w_{ij}(t + 1) = w_{ij}(t) - \Delta w_{ij}$$

$$\text{where } \Delta w_{ij} = -\eta \, \delta E/\delta w_{ij.}$$

1. *When no change (within some accuracy) occurs, the weights are frozen and network is ready to use on data it has never seen*

Perceptrons : Limitation

- Recession
  - 1969 Minsky-Papert: limitations of perceptron model
    → Linear Separability in Perceptrons

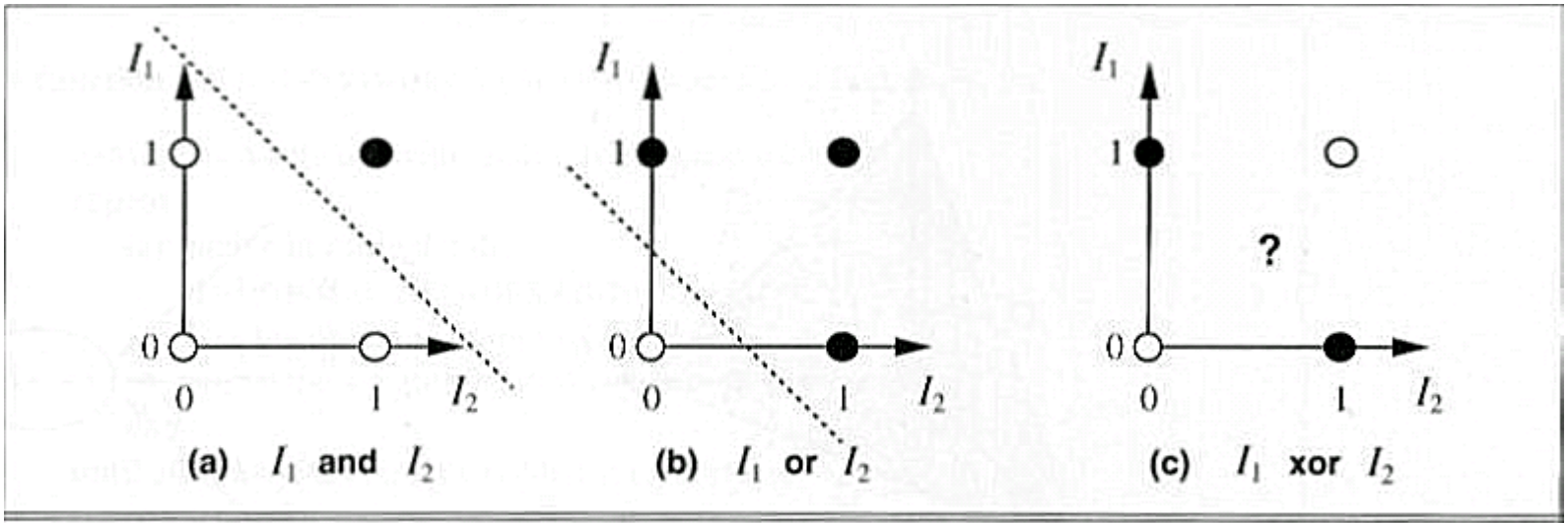
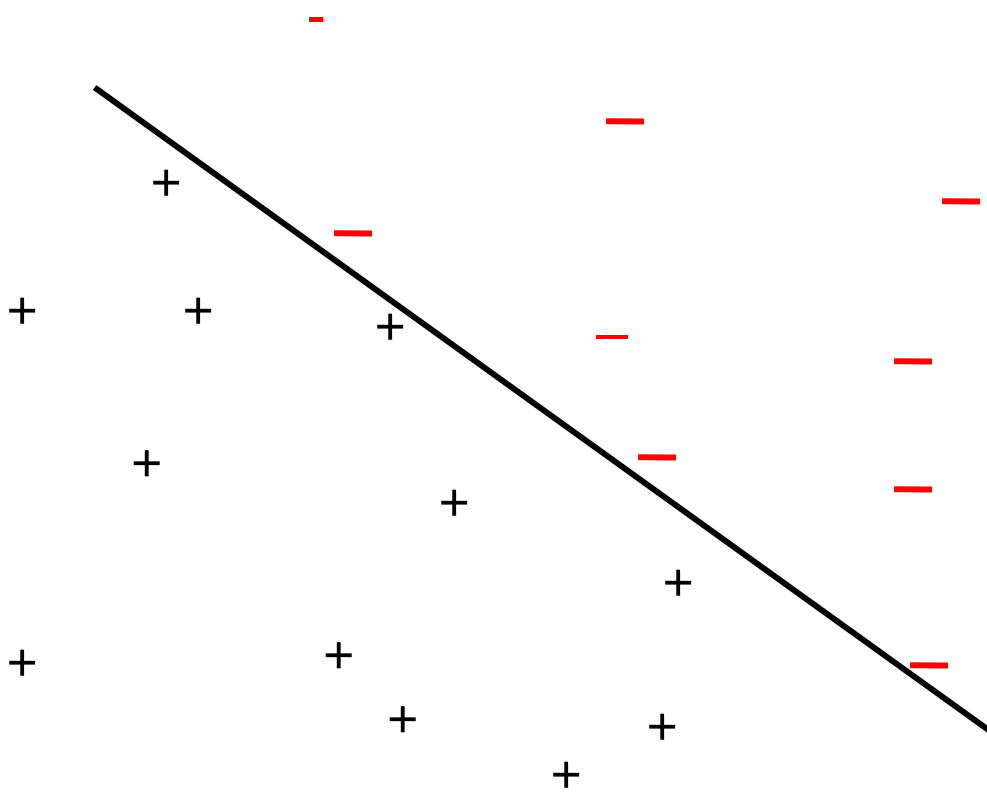
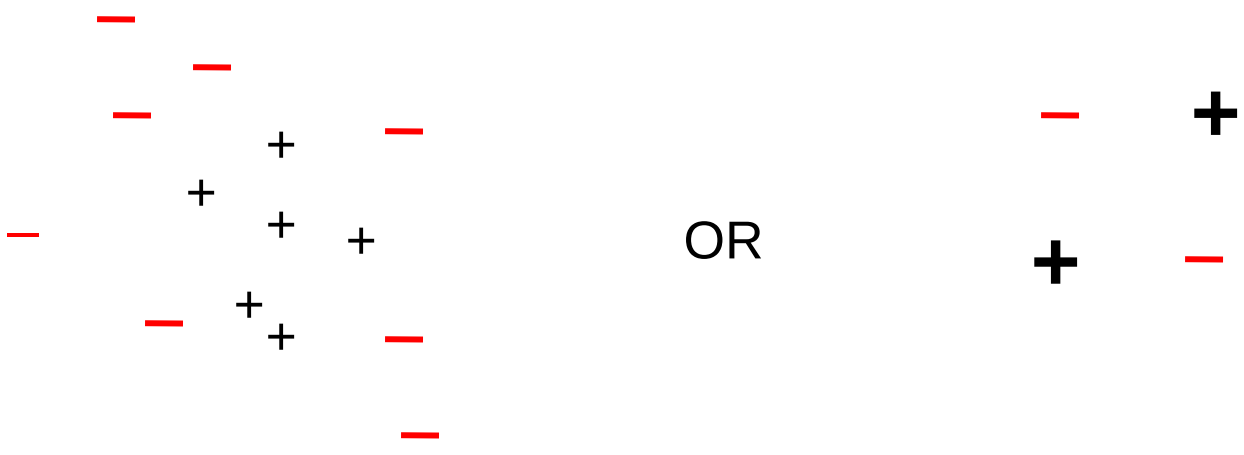


Figure 19.9 Linear separability in perceptrons.

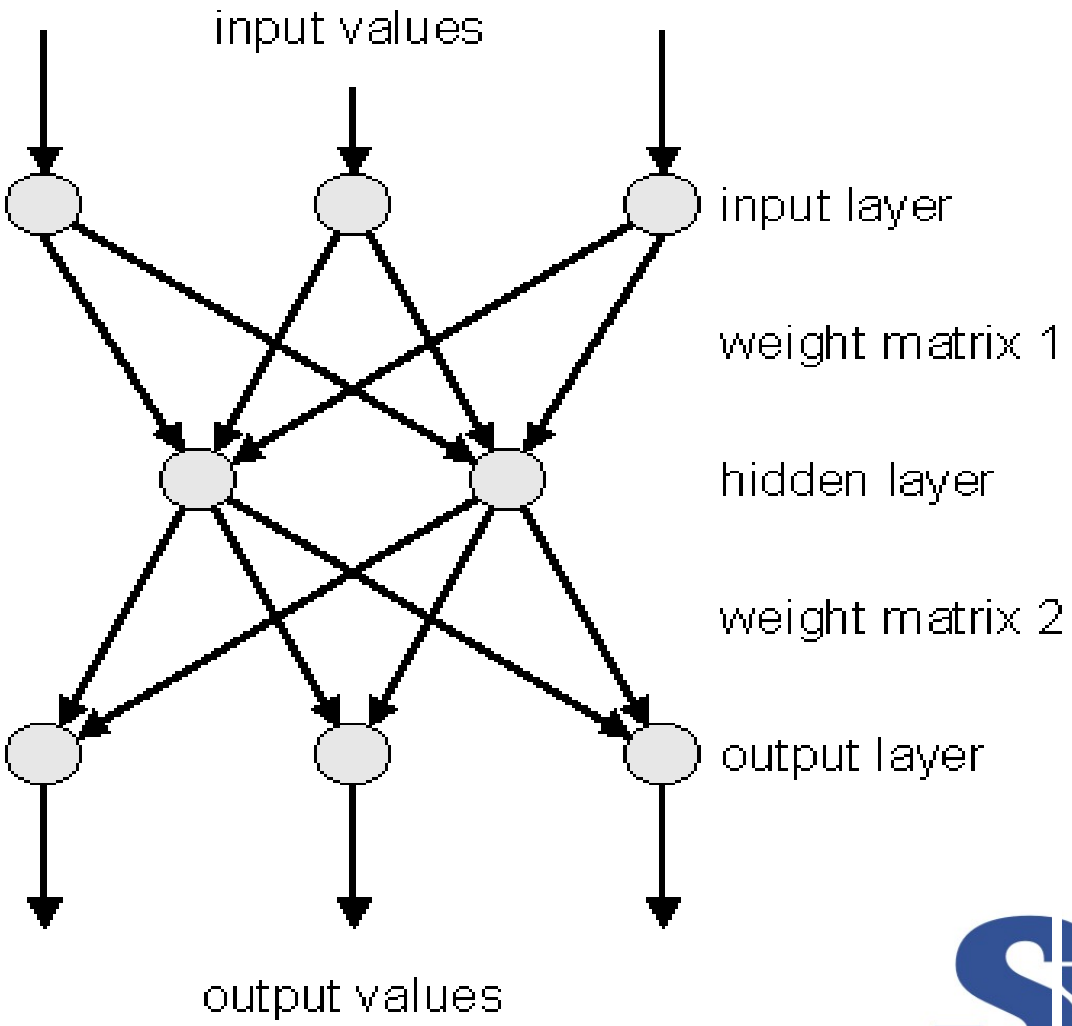
Linearly Separable



NOT linearly Separable

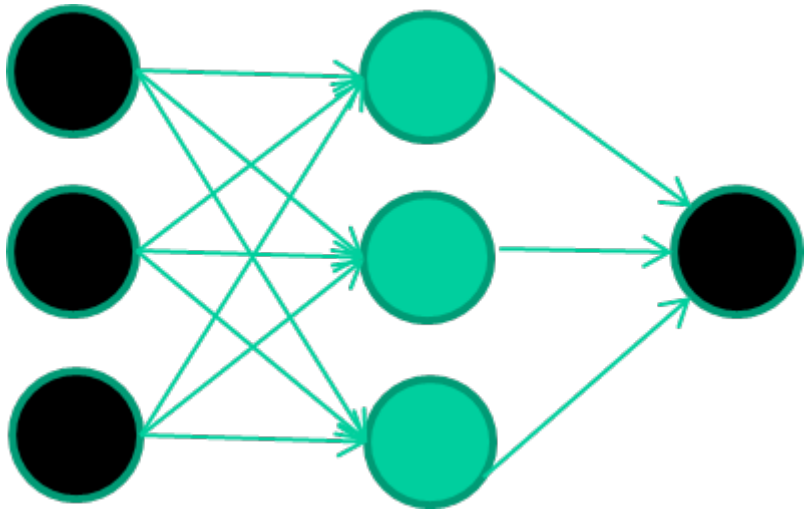


MLP Structure



A dataset

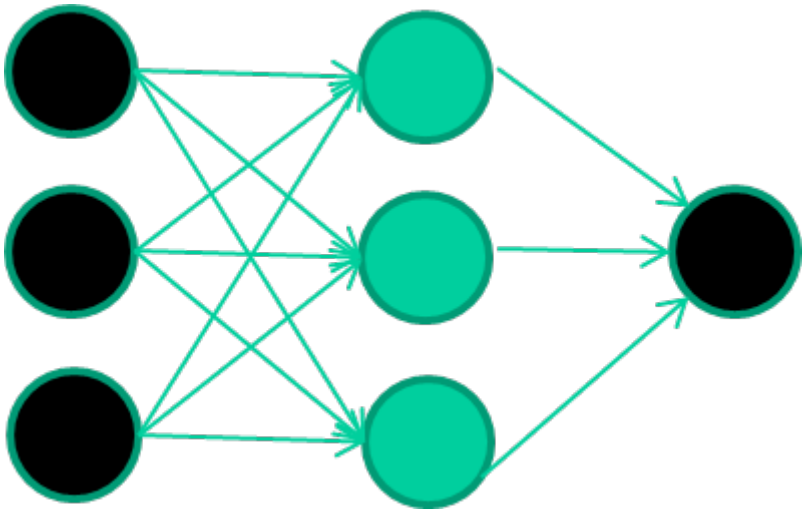
Fields			***class***
1.4	2.7	1.9	0
3.8	3.4	3.2	0
6.4	2.8	1.7	1
4.1	0.1	0.2	0
etc …			



Training the neural network

Fields			***class***
1.4	2.7	1.9	0
3.8	3.4	3.2	0
6.4	2.8	1.7	1
4.1	0.1	0.2	0

etc …

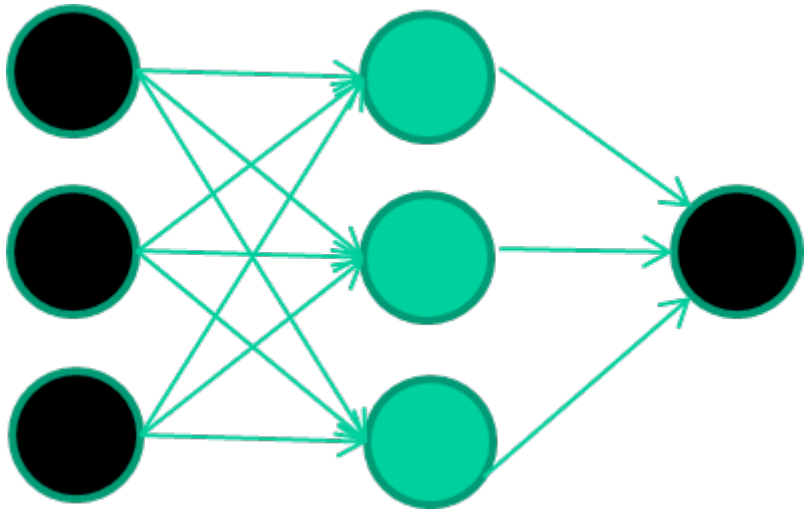


Training data

Fields			***class***
1.4	2.7	1.9	0
3.8	3.4	3.2	0
6.4	2.8	1.7	1
4.1	0.1	0.2	0

etc …

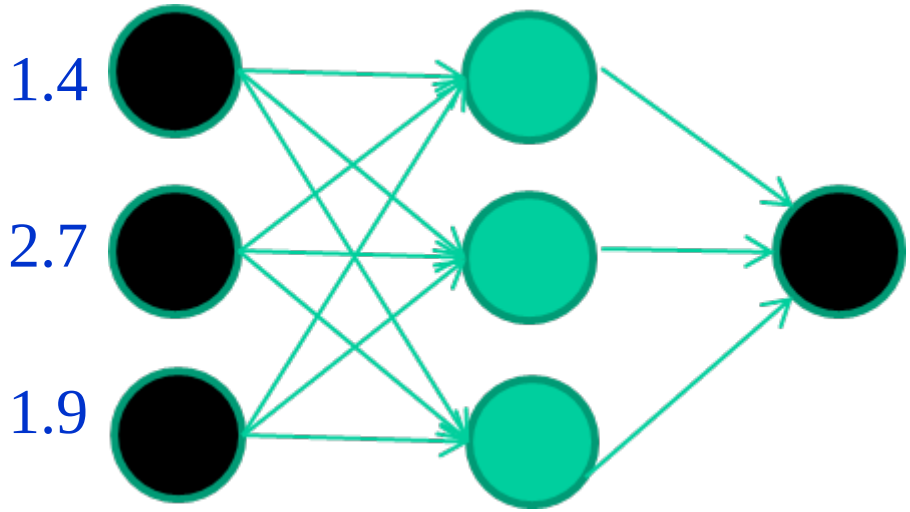
Initialise with random weights



Training data

Fields			***class***
1.4	2.7	1.9	0
3.8	3.4	3.2	0
6.4	2.8	1.7	1
4.1	0.1	0.2	0
etc …			

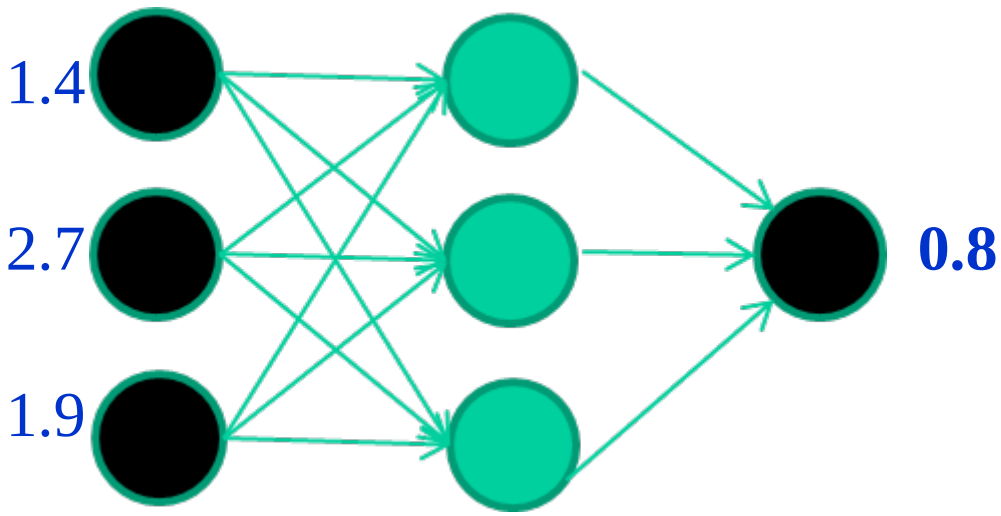
Present a training pattern



Training data

Fields			***class***
1.4	2.7	1.9	0
3.8	3.4	3.2	0
6.4	2.8	1.7	1
4.1	0.1	0.2	0
etc …			

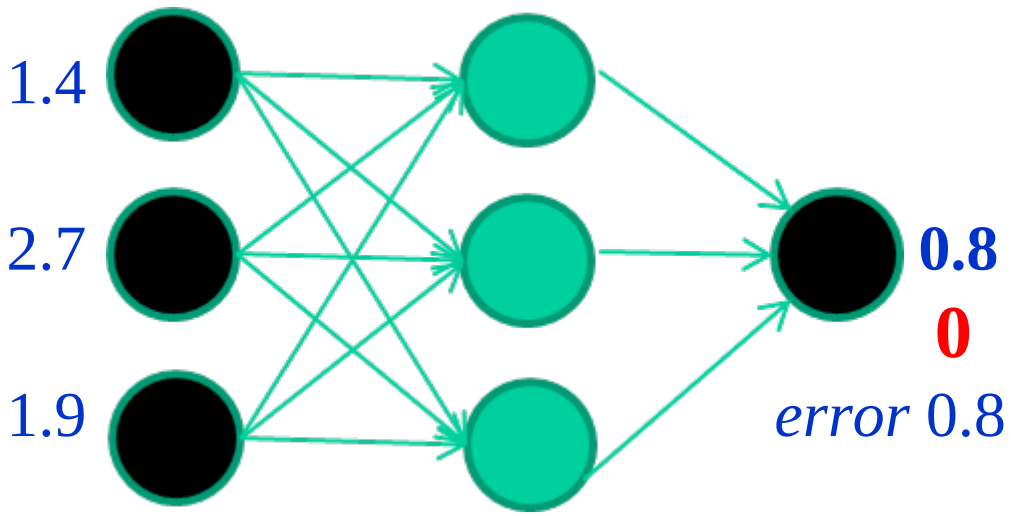
Feed it through to get output



Training data

Fields			***class***
1.4	2.7	1.9	0
3.8	3.4	3.2	0
6.4	2.8	1.7	1
4.1	0.1	0.2	0
etc …			

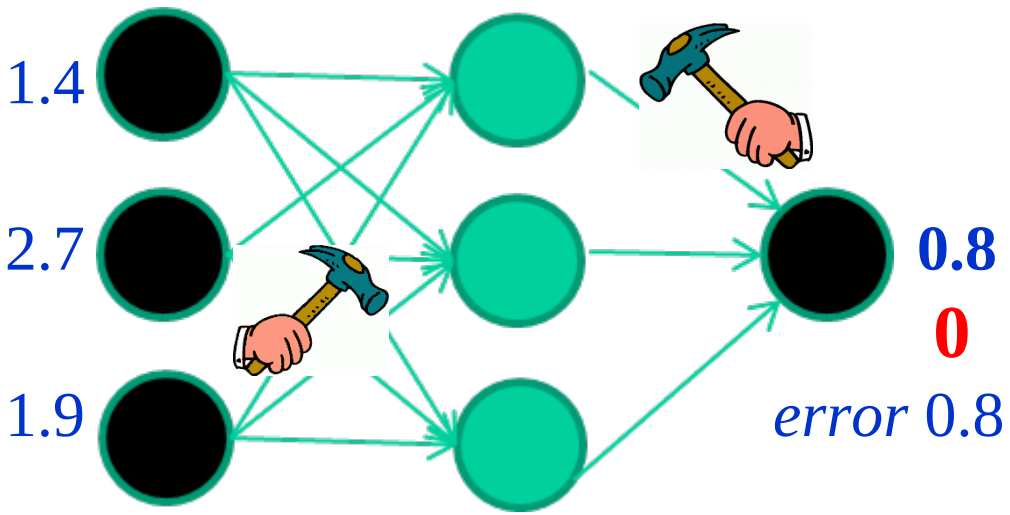
Compare with target output



Training data

Fields			***class***
1.4	2.7	1.9	0
3.8	3.4	3.2	0
6.4	2.8	1.7	1
4.1	0.1	0.2	0
etc …			

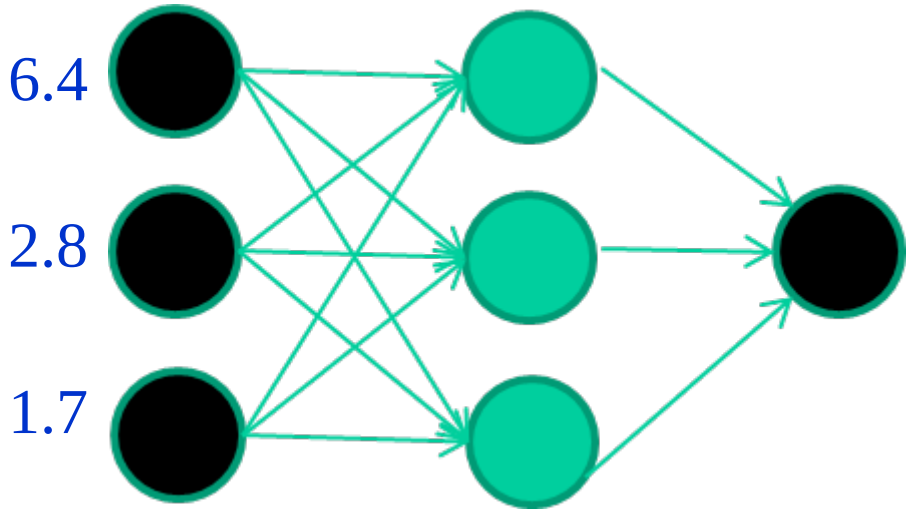
Adjust weights based on error



Training data

Fields			***class***
1.4	2.7	1.9	0
3.8	3.4	3.2	0
6.4	2.8	1.7	1
4.1	0.1	0.2	0
etc …			

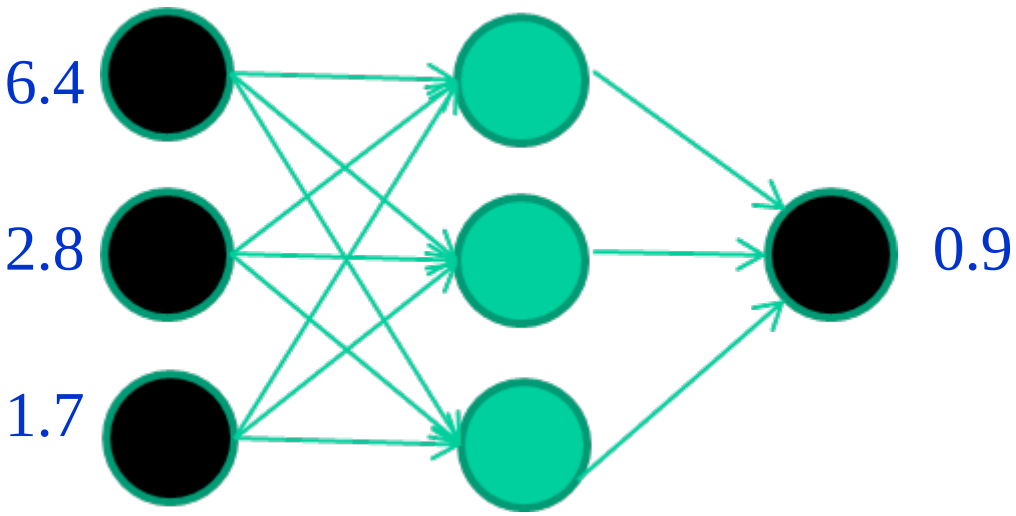
Present a training pattern



Training data

Fields			***class***
1.4	2.7	1.9	0
3.8	3.4	3.2	0
6.4	2.8	1.7	1
4.1	0.1	0.2	0
etc …			

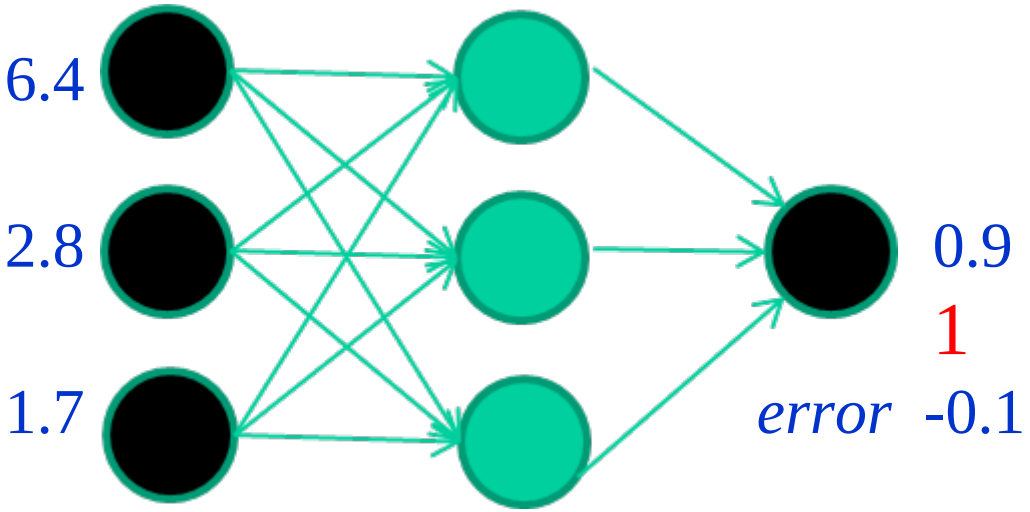
Feed it through to get output



Training data

Fields			***class***
1.4	2.7	1.9	0
3.8	3.4	3.2	0
6.4	2.8	1.7	1
4.1	0.1	0.2	0
etc …			

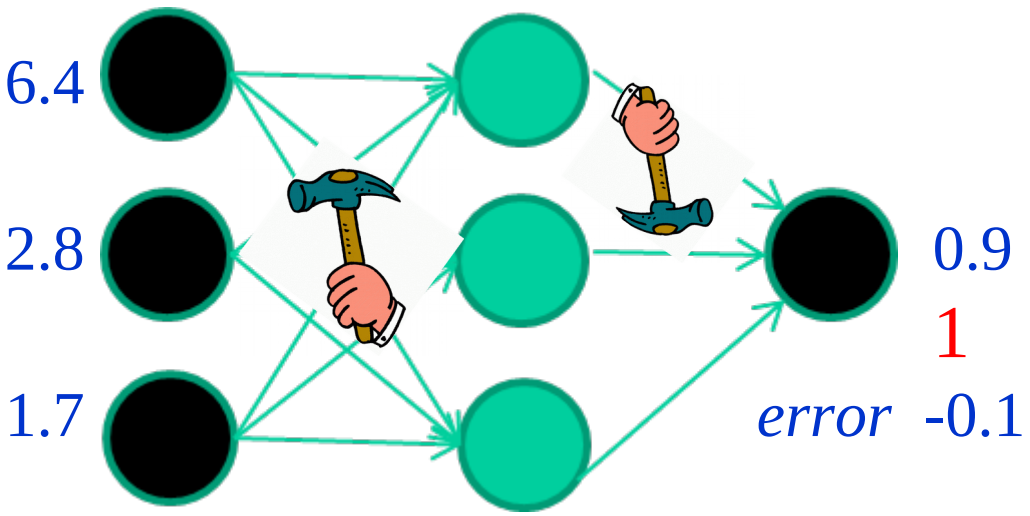
Compare with target output



Training data

Fields			***class***
1.4	2.7	1.9	0
3.8	3.4	3.2	0
6.4	2.8	1.7	1
4.1	0.1	0.2	0
etc …			

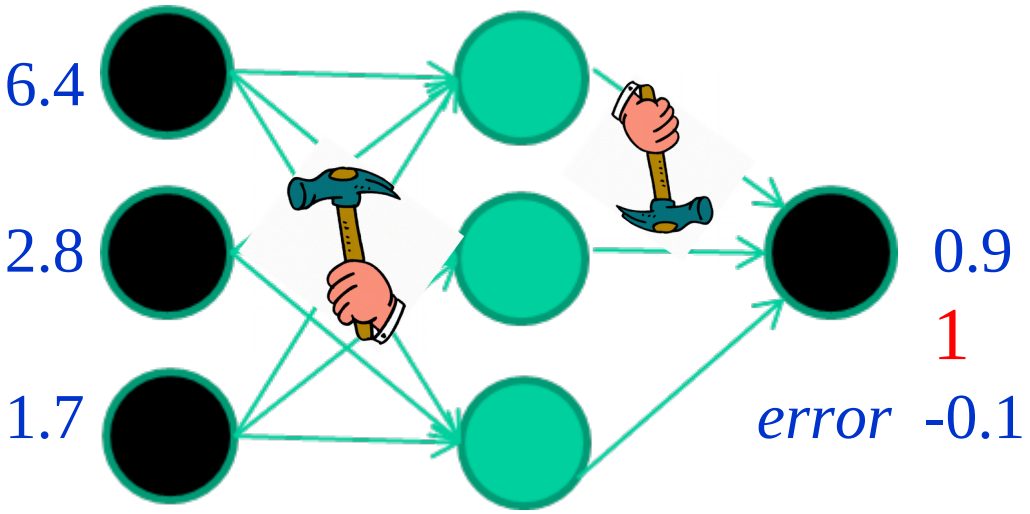
Adjust weights based on error



Training data

Fields			***class***
1.4	2.7	1.9	0
3.8	3.4	3.2	0
6.4	2.8	1.7	1
4.1	0.1	0.2	0
etc …			

And so on ….

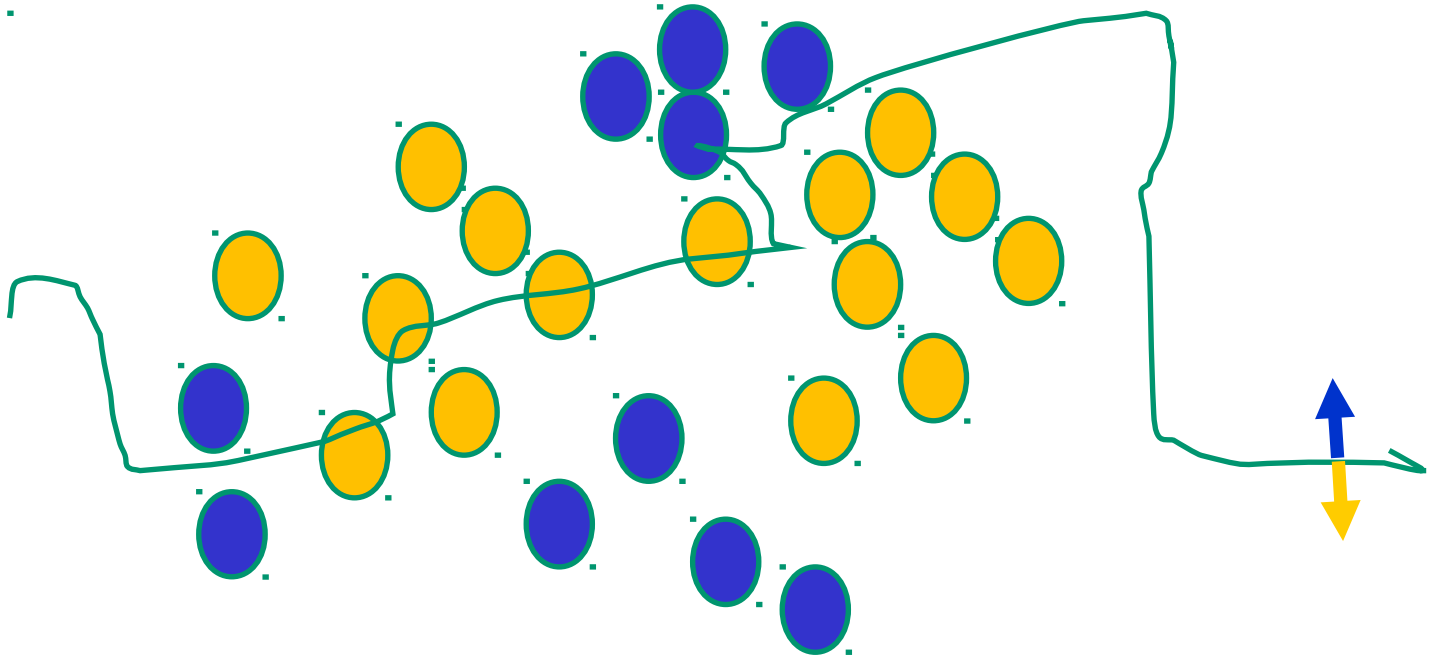


Repeat this thousands, maybe millions of times – each time taking a random training instance, and making slight weight adjustments

Algorithms for weight adjustment are designed to make changes that will reduce the error

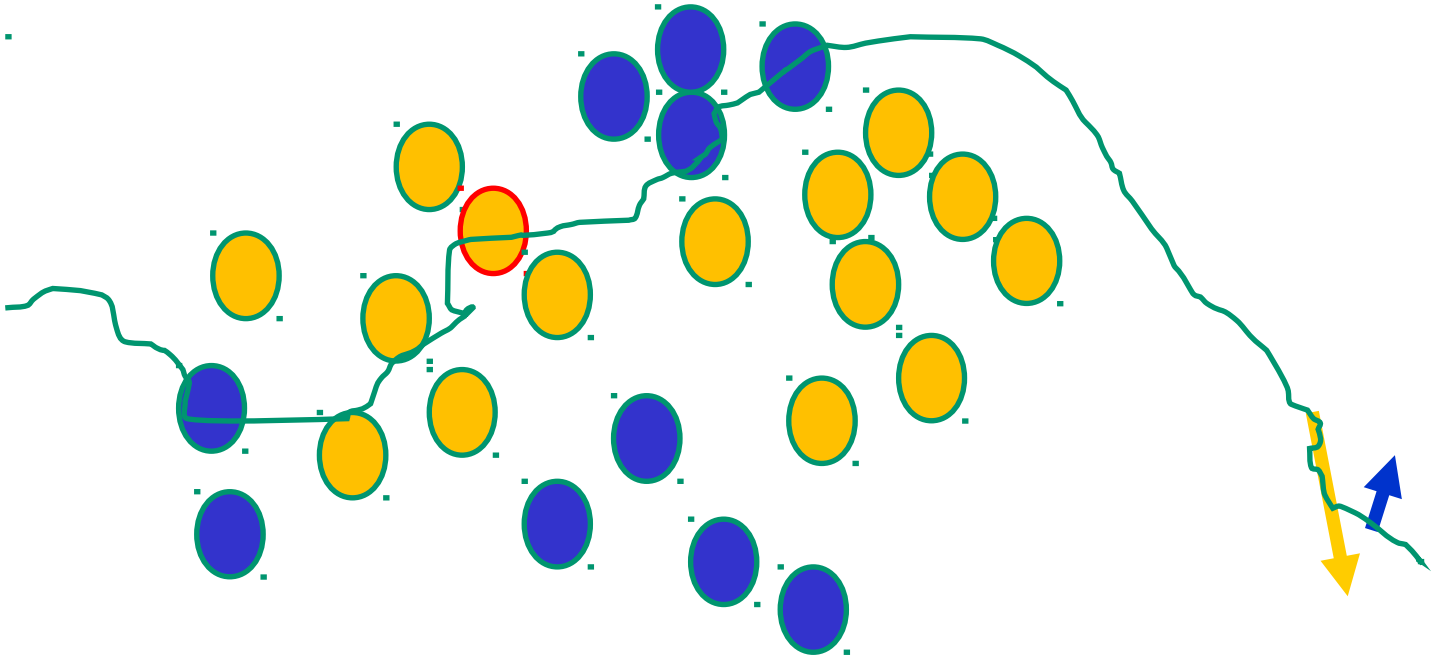
The decision boundary perspective...

Initial random weights



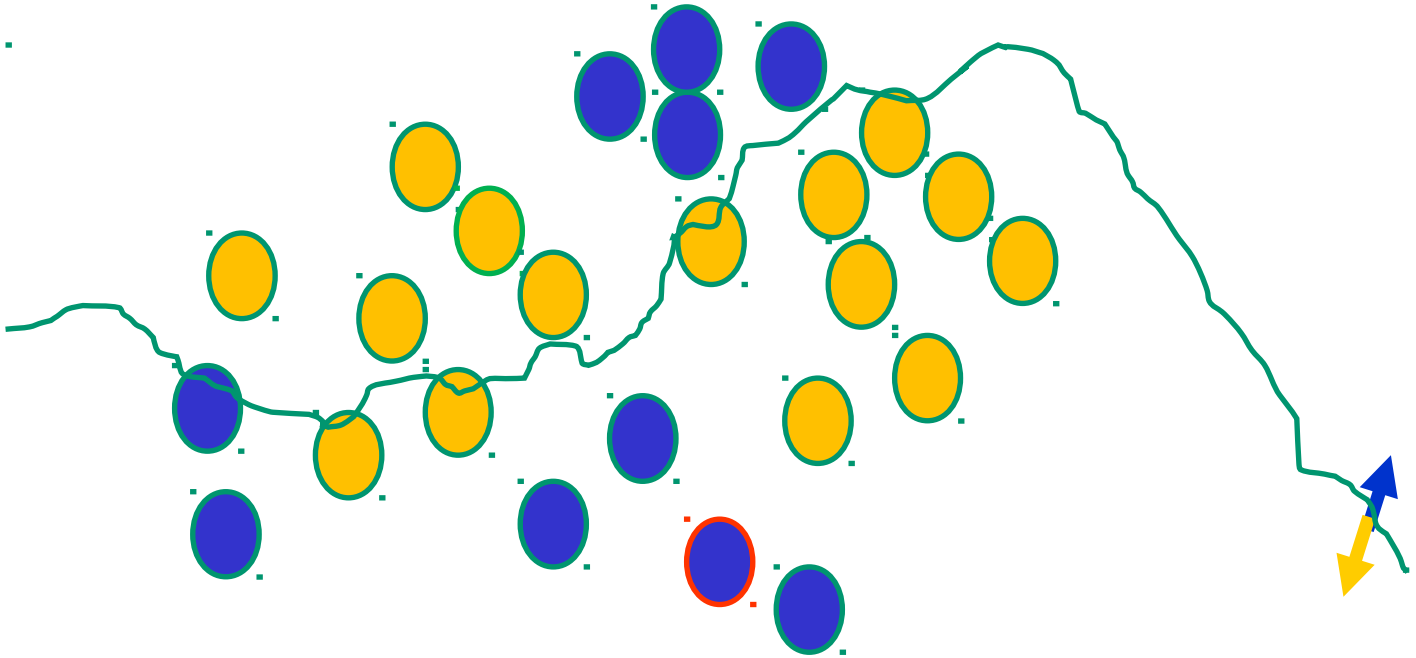
The decision boundary perspective...

Present a training instance / adjust the weights



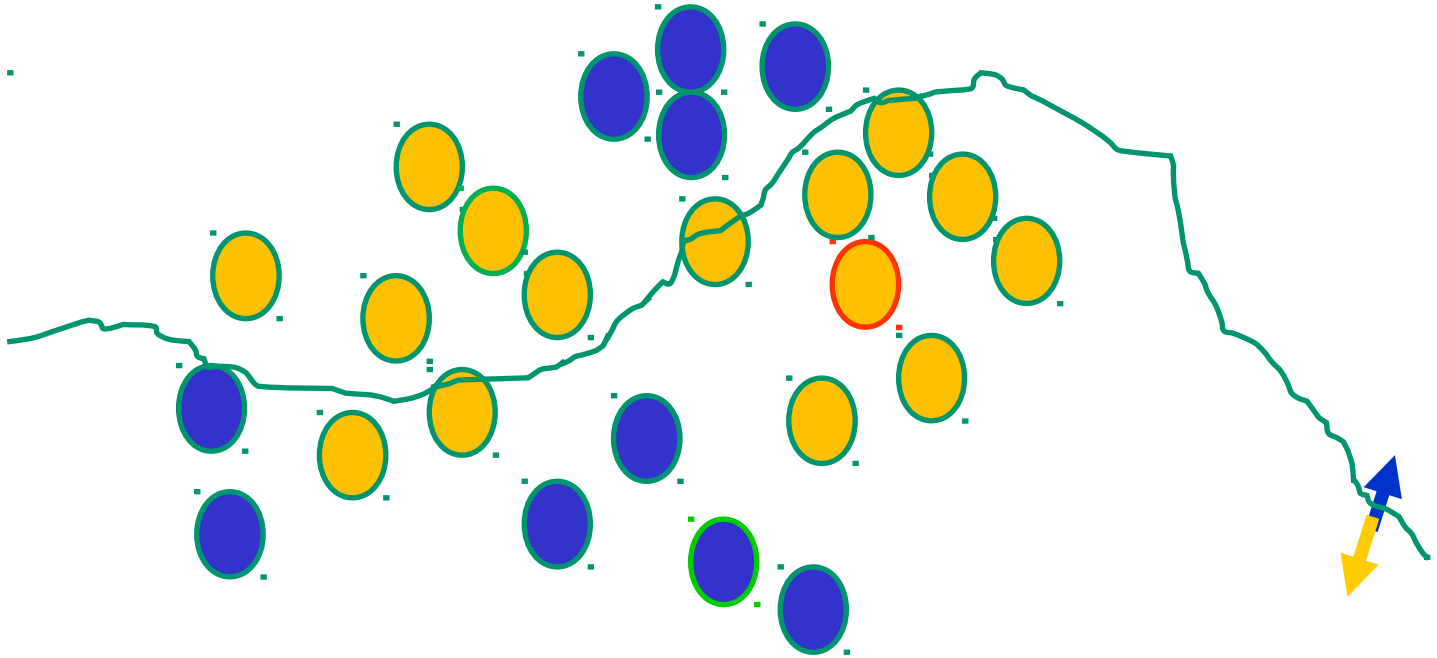
The decision boundary perspective...

Present a training instance / adjust the weights



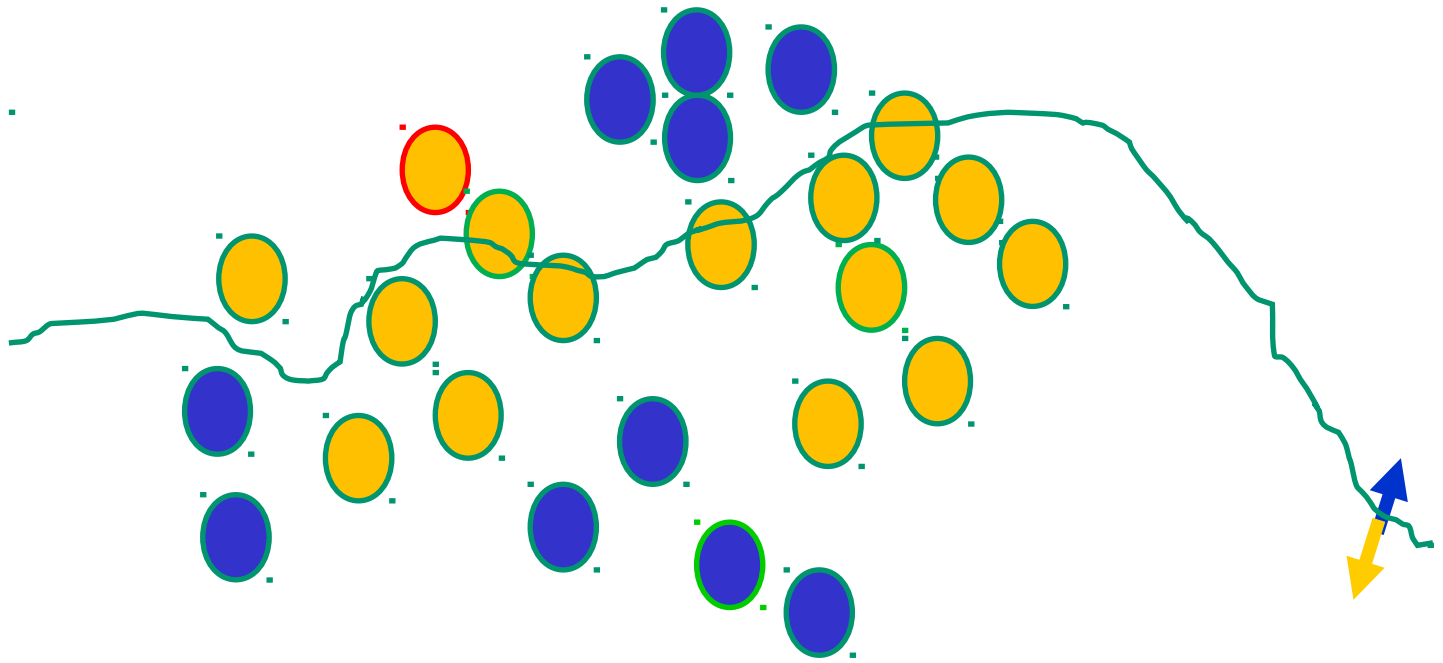
The decision boundary perspective...

Present a training instance / adjust the weights



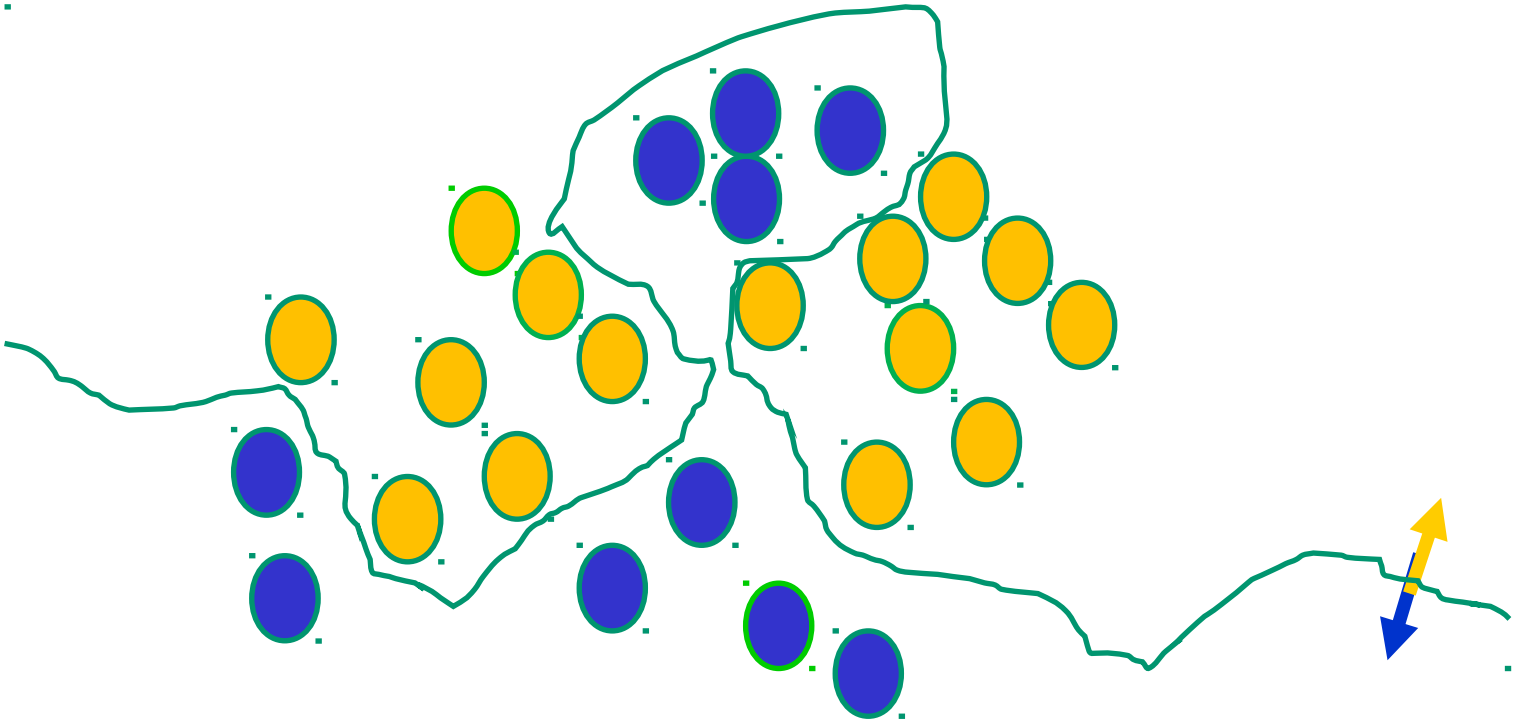
The decision boundary perspective...

Present a training instance / adjust the weights



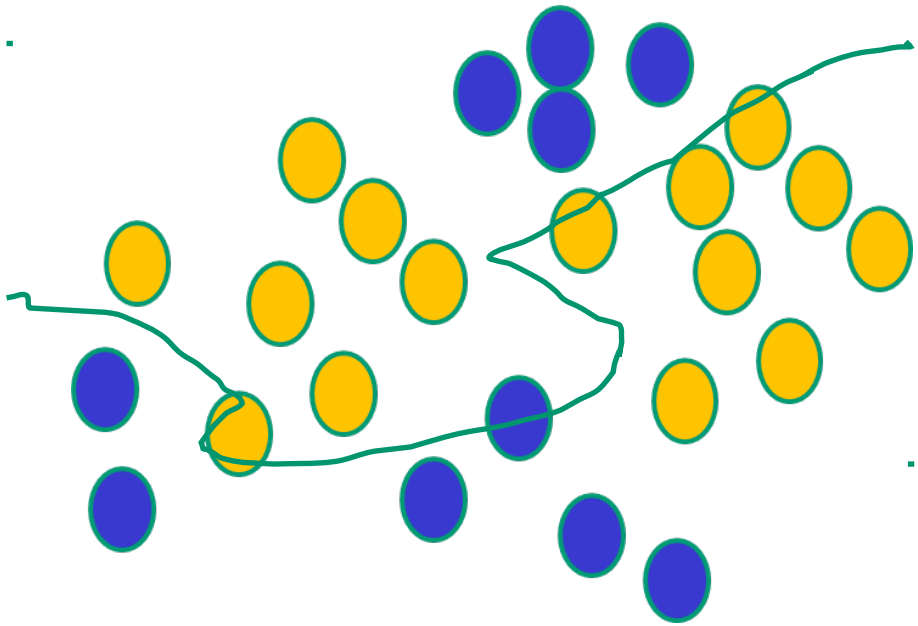
The decision boundary perspective…

Eventually ….



Some ‘by the way’ points

NNs use nonlinear g(z) so they can draw complex boundaries, but keep the data unchanged



Some other ‘by the way’ points

NNs use nonlinear *f*(x) so they

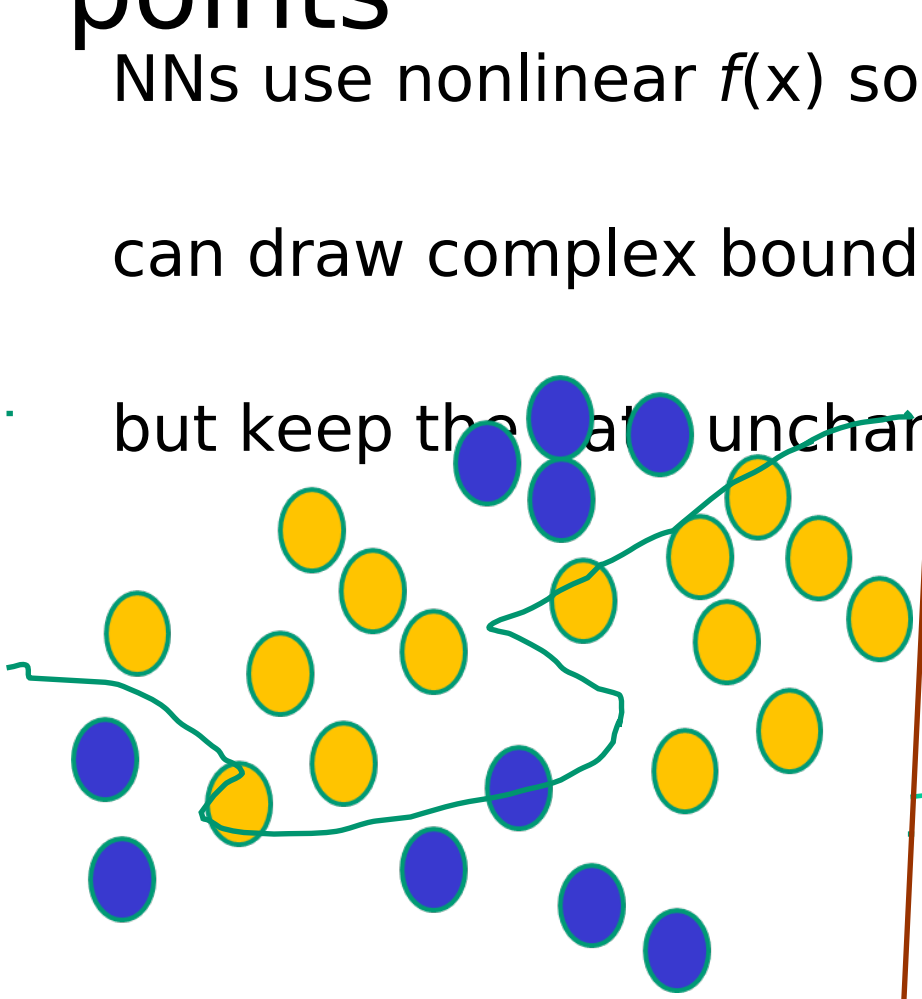
can draw complex boundaries,

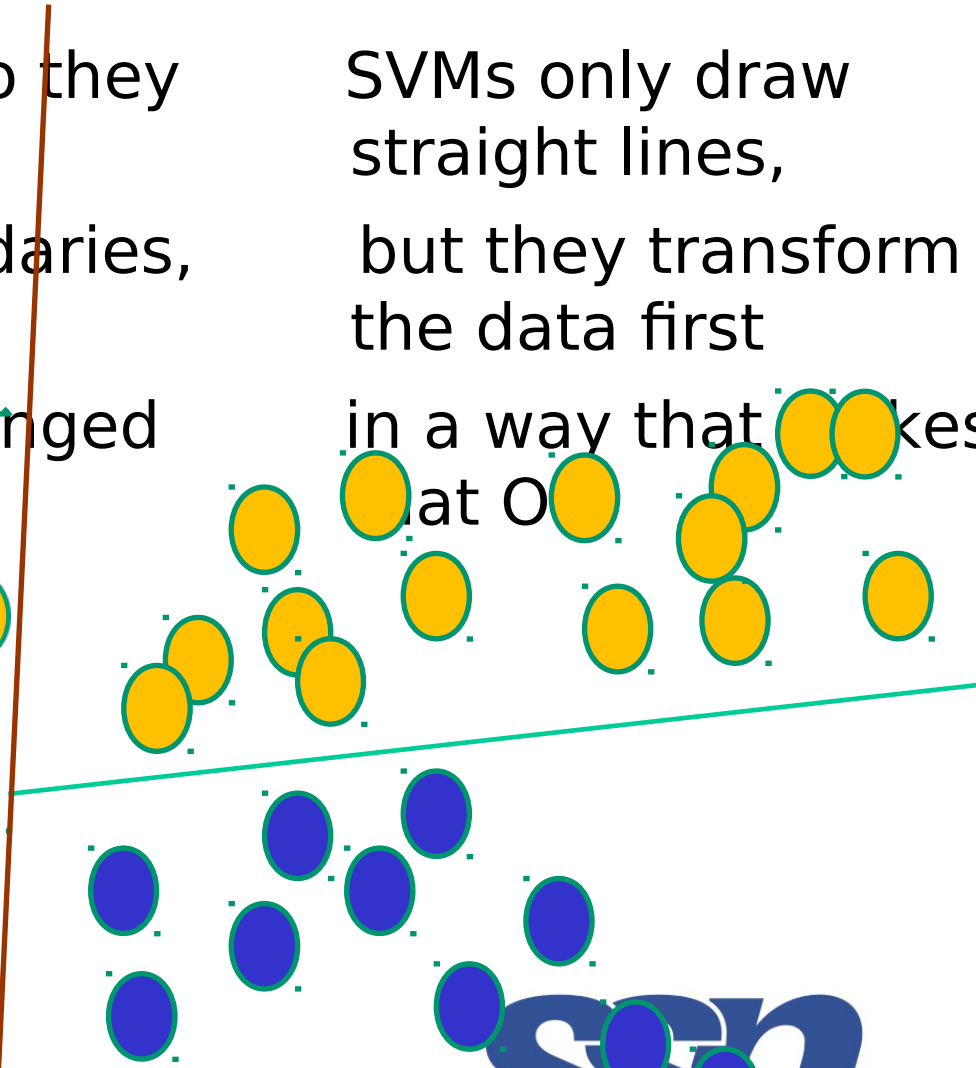
but keep the data unchanged

SVMs only draw straight lines,

but they transform the data first

in a way that makes it OK





Feature detectors

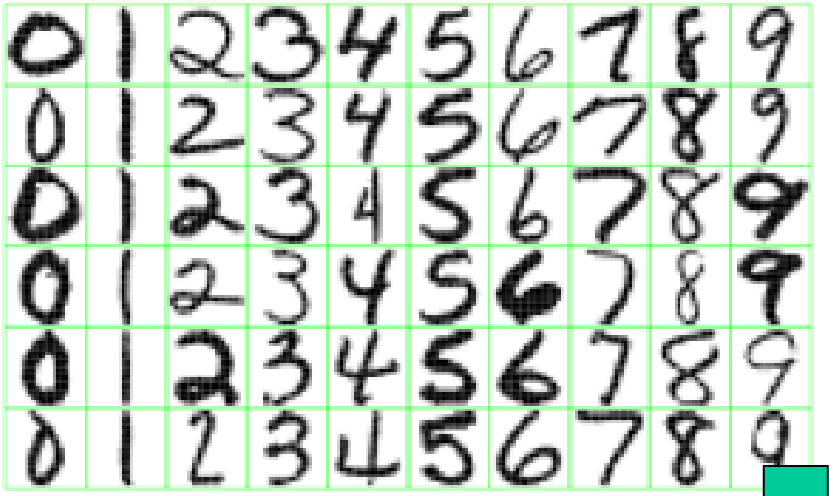
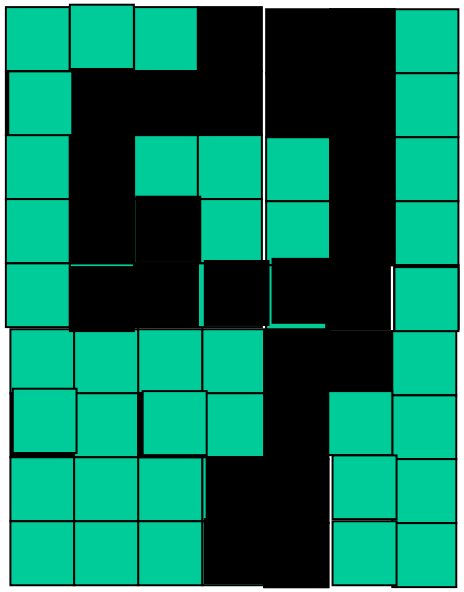
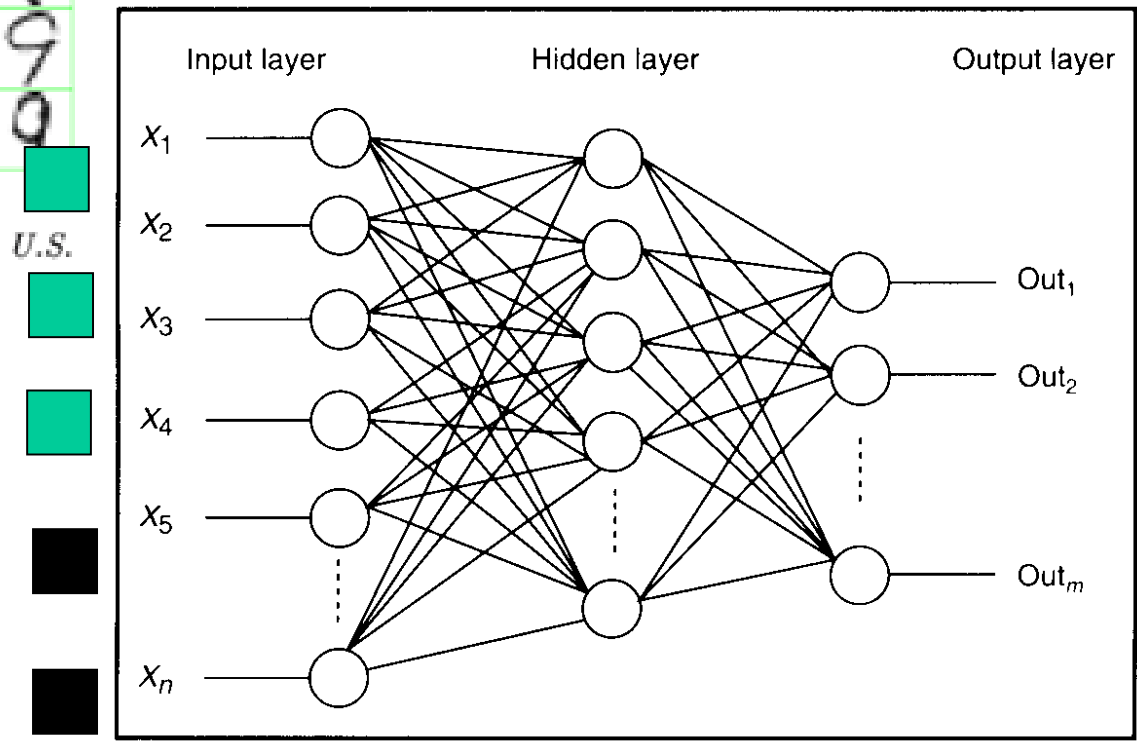
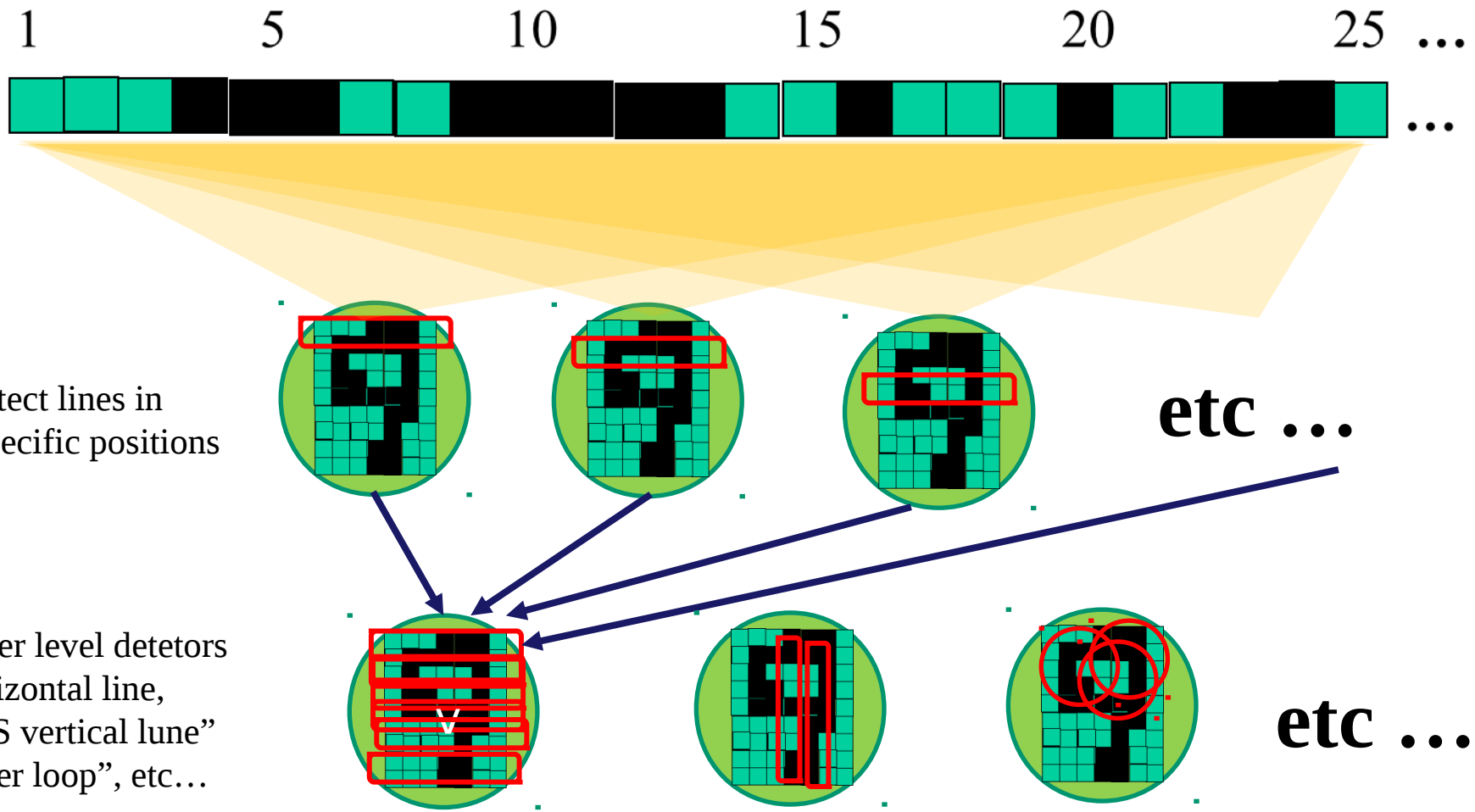


Figure 1.2: *Examples of handwritten digits from U.S. postal envelopes.*

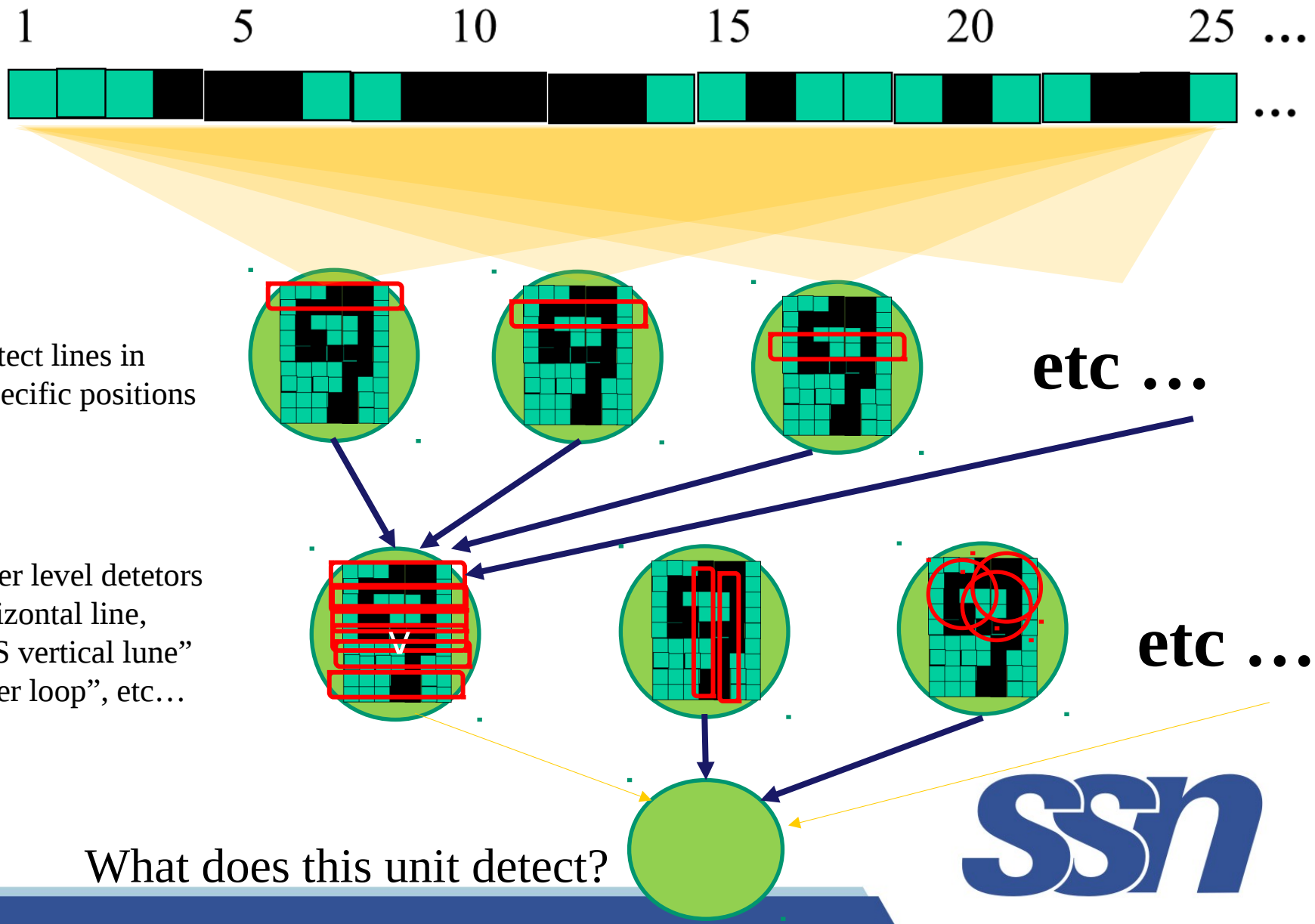




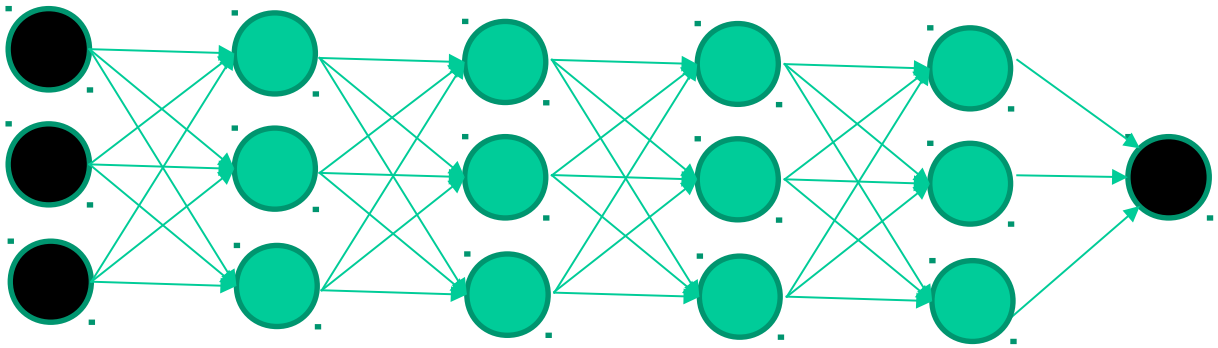
successive layers can learn higher-level features …



successive layers can learn higher-level features …



So: *multiple layers make sense*



So: *multiple layers make sense*

Your brain works that way

Hierarchical organization

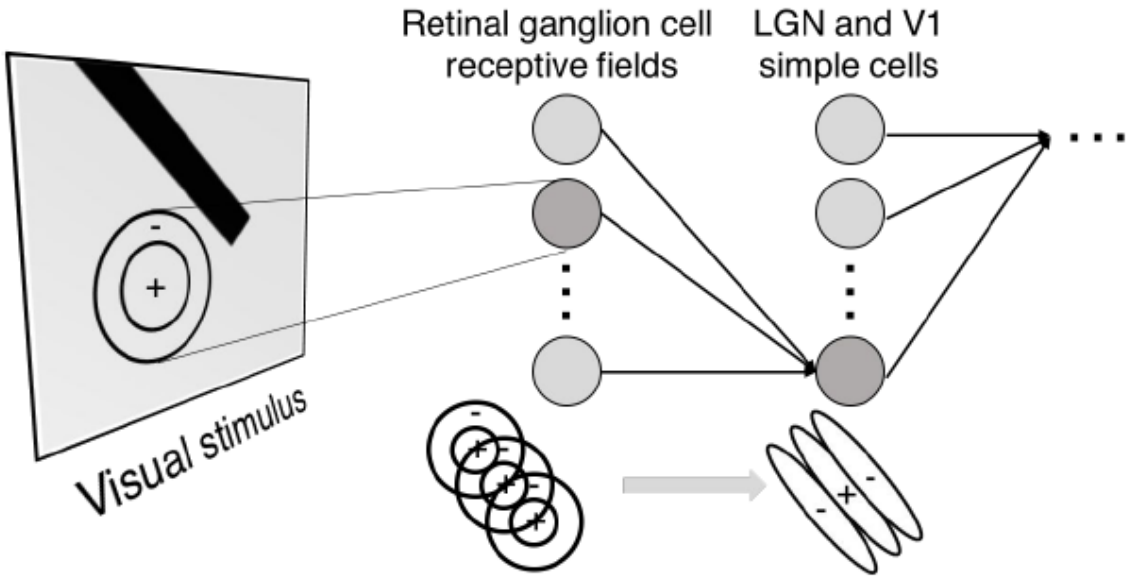


Illustration of hierarchical organization in early visual pathways by Lane McIntosh, copyright CS231n 2017

Simple cells:
Response to light orientation

Complex cells:
Response to light orientation and movement

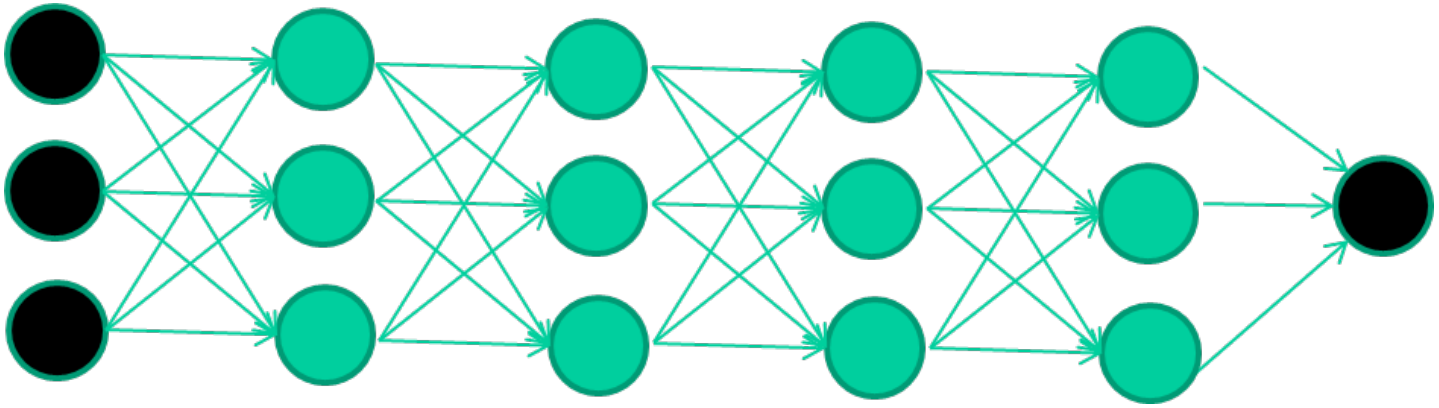
Hypercomplex cells:
response to movement with an end point

No response

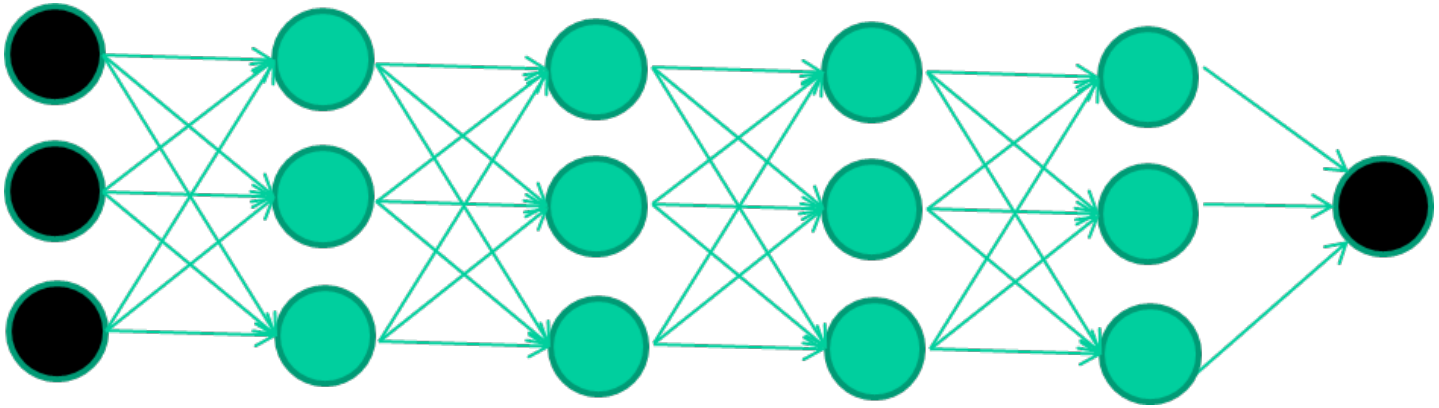
Response (end point)

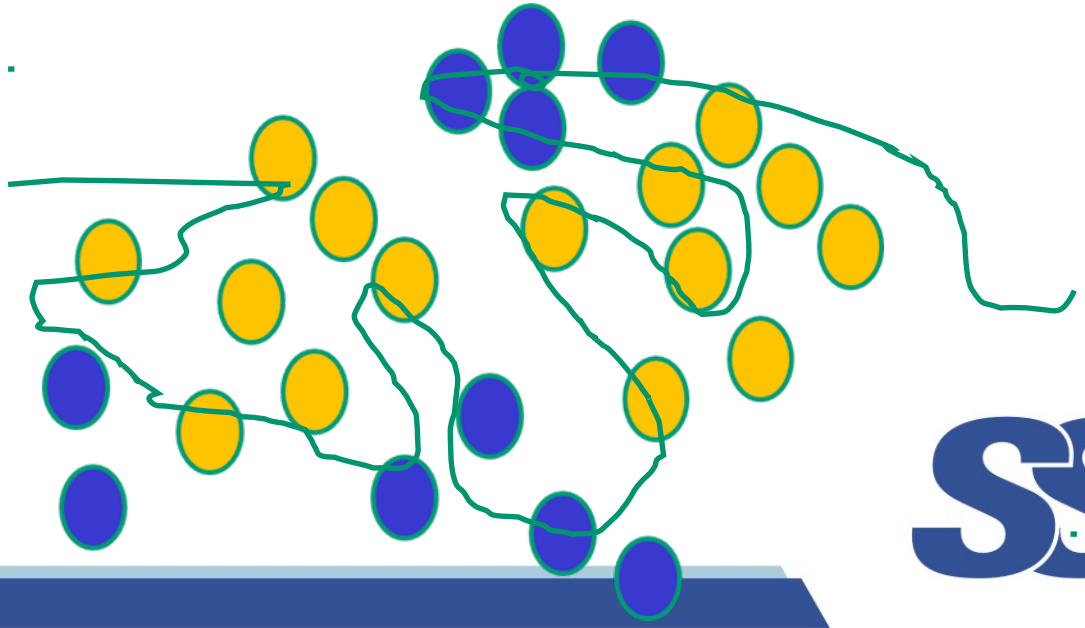
So: *multiple layers make sense*

Many-layer neural network architectures should be capable of learning the true underlying features and ‘feature logic’, and therefore generalise very well …



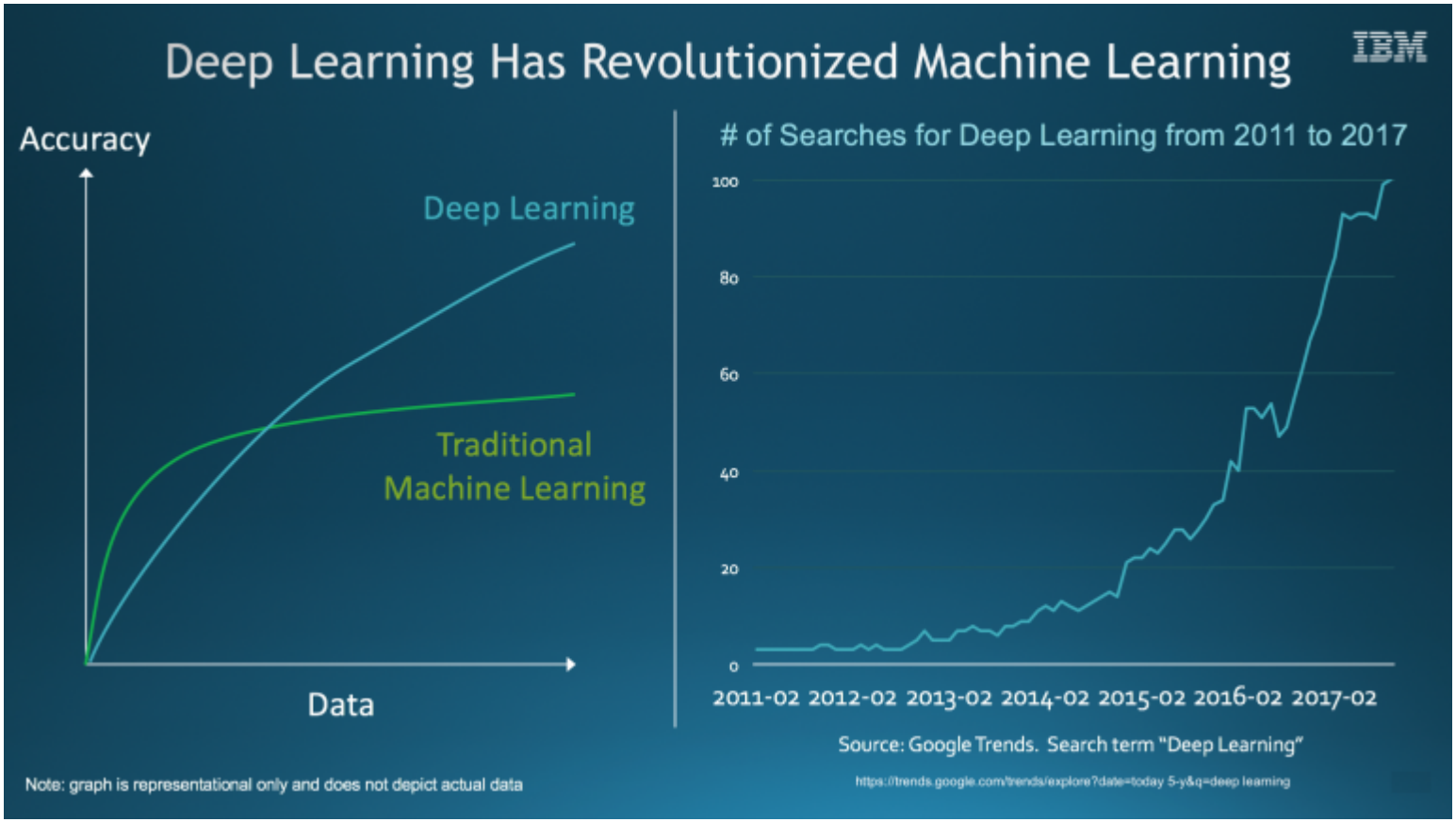
But, until very recently, our weight-learning algorithms simply did not work on multi-layer architectures





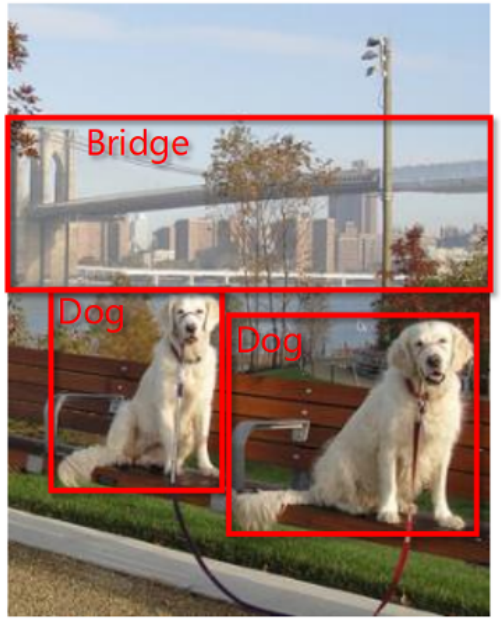
Along came deep learning ...

Along came deep learning ...

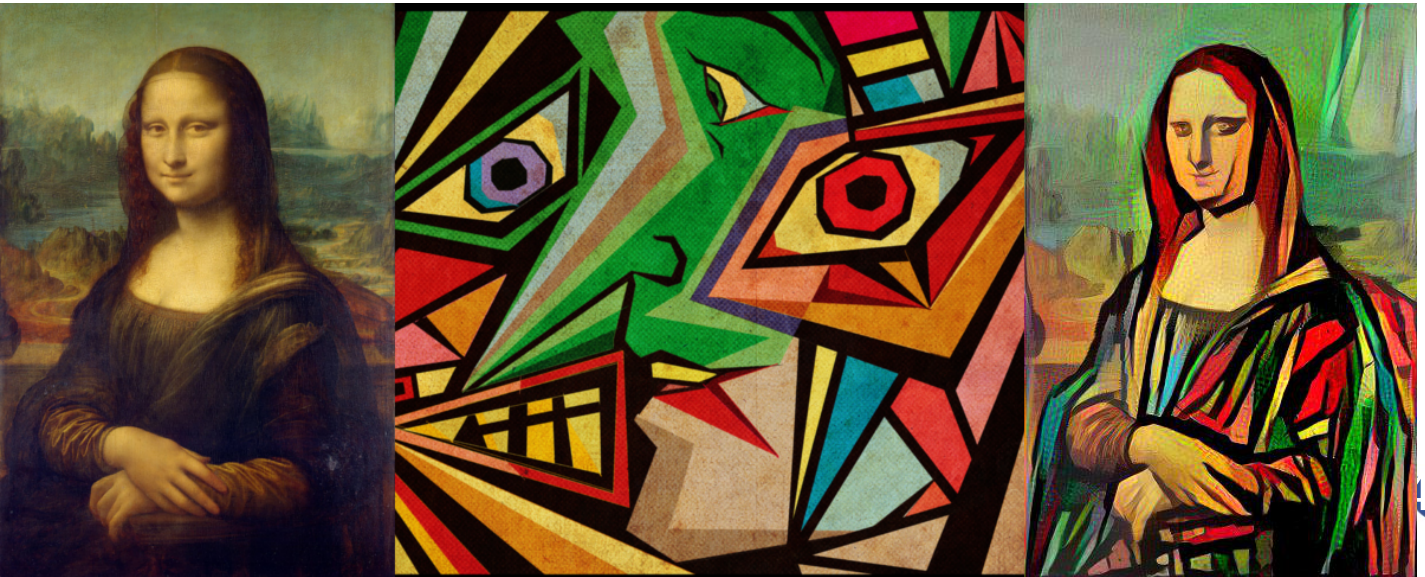


Convolutional Neural Networks

- Some Applications
  - Image classification
  - Object detection
  - Neural style transfer

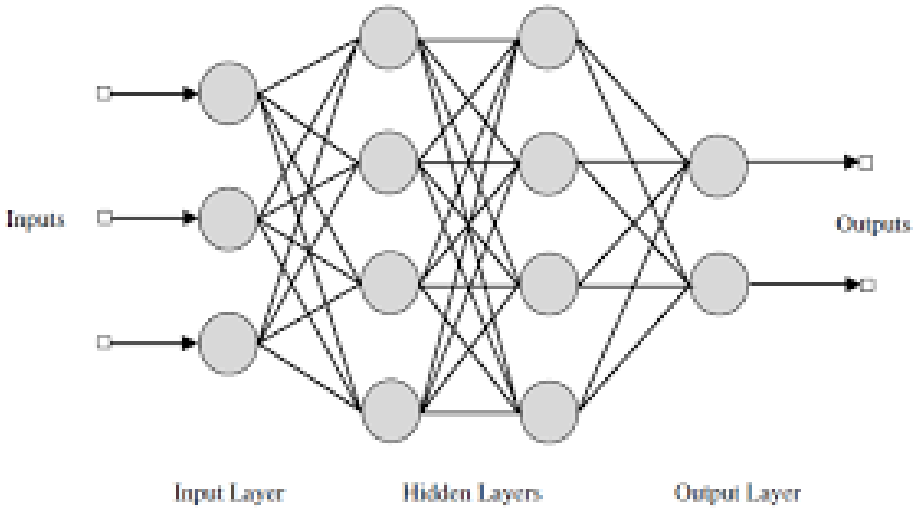


Object detection, still a lot harder



Conventional NN - # parameters

- Assume you have a 64x64x3 (RGB) image → 12288 input features
- If 1000x1000x3 image → 3 million (M) features
- If 1st hidden layer = 1000 neurons →
- Weight matrix = 1000x3M = 3 billion parameters (very large)
- Difficult to get large data to avoid overfitting



Convolution – building block for CNN

- 6x6 image

3	0	1	2	7	4
1	5	8	9	3	1
2	7	2	5	1	3
0	1	3	1	7	8
4	2	1	6	2	8
2	4	5	2	3	9

*

3x3 filter
vertical edge detector

1	0	-1
1	0	-1
1	0	-1

=

4x4 output

-5	-4	0	8
-10	-2	2	3
0	-2	-4	-7
-3	-2	-3	-16

- Helps to detect vertical edges in an image

Convolution – building block for CNN

- 6x6 image

10	10	10	0	0	0
10	10	10	0	0	0
10	10	10	0	0	0
10	10	10	0	0	0
10	10	10	0	0	0
10	10	10	0	0	0

*

3x3 filter
vertical edge detector

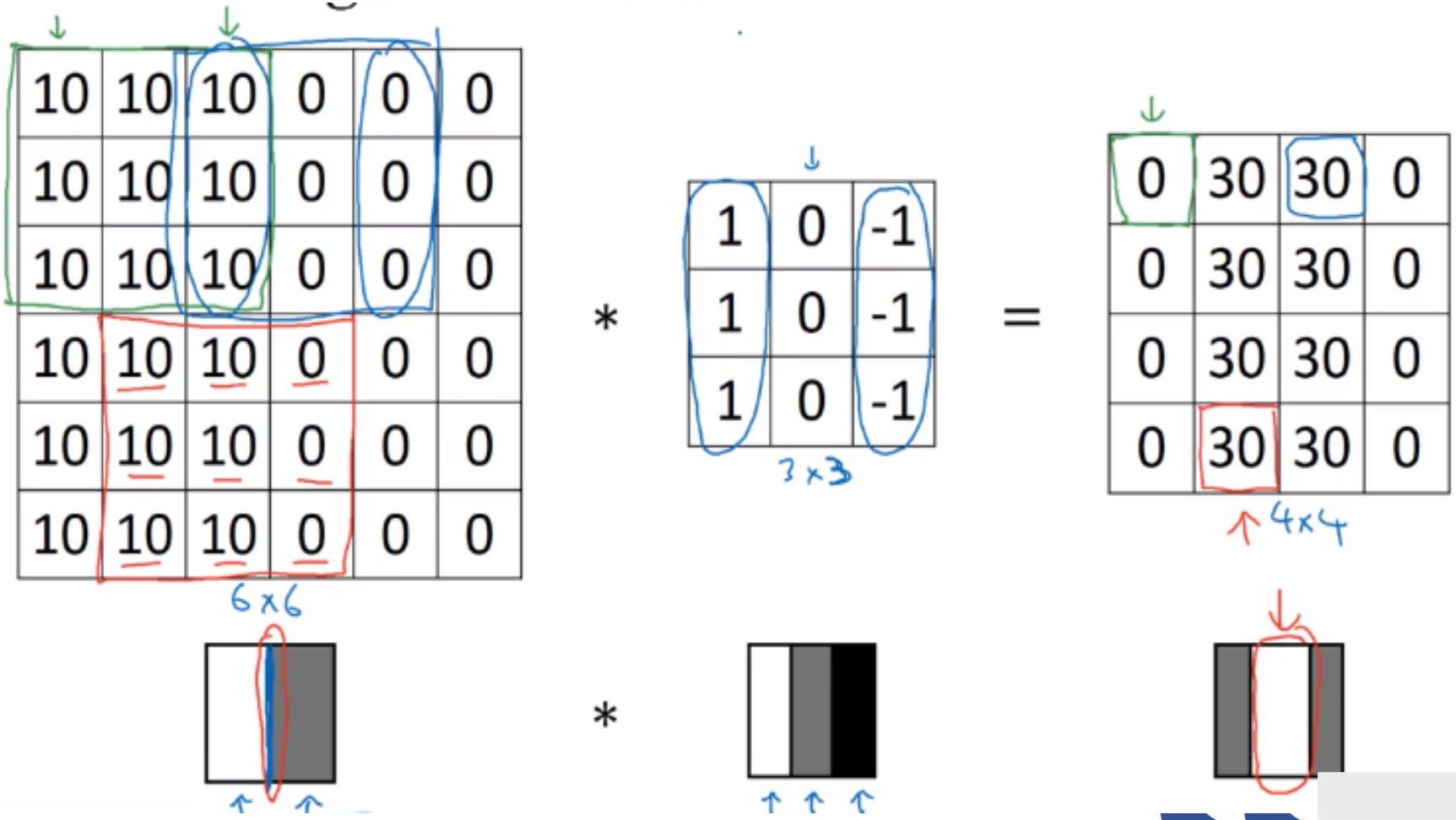
1	0	-1
1	0	-1
1	0	-1

=

4x4 output

0	30	30	0
0	30	30	0
0	30	30	0
0	30	30	0

Convolution – building block for CNN



Convolution – building block for CNN

- 6x6 image

0	0	0	10	10	10
0	0	0	10	10	10
0	0	0	10	10	10
0	0	0	10	10	10
0	0	0	10	10	10
0	0	0	10	10	10

*

3x3 filter
vertical edge detector

1	0	-1
1	0	-1
1	0	-1

Light pixels

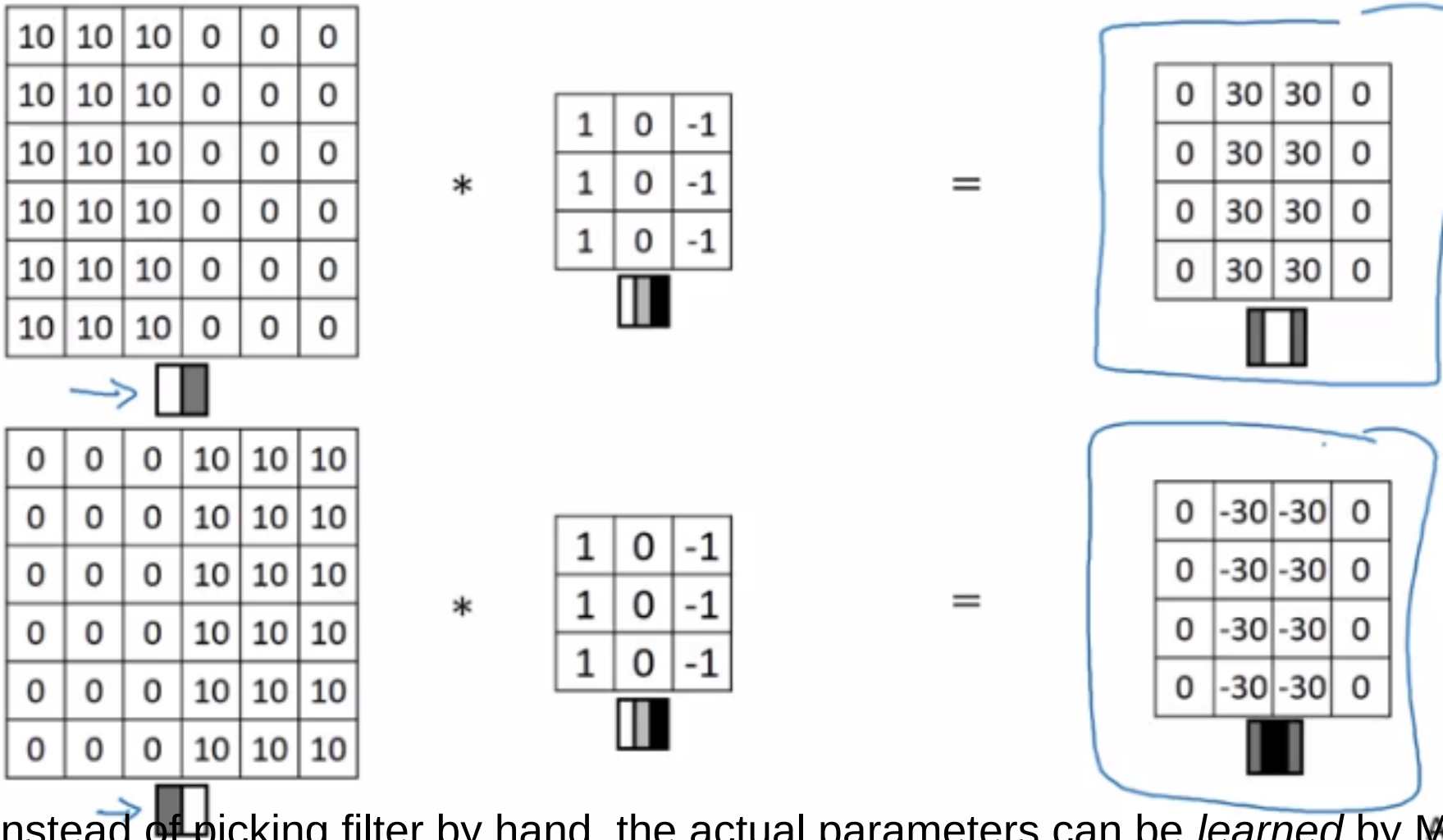
dark pixels

=

4x4 output

0	-30	-30	0
0	-30	-30	0
0	-30	-30	0
0	-30	-30	0

Convolution – building block for CNN



Instead of picking filter by hand, the actual parameters can be *learned* by M

Padding

- Zero padding, p=1. Take a 6x6 image and pad with 1 layer of 0 on all 4 edges to get 8x8 image

0	0	0	0	0	0	0	0
0	3	0	1	2	7	4	0
0	1	5	8	9	3	1	0
0	2	7	2	5	1	3	0
0	0	1	3	1	7	8	0
0	4	2	1	6	2	8	0
0	2	4	5	2	3	9	0
0	0	0	0	0	0	0	0

* [3x3 filter] = [6x6 output]

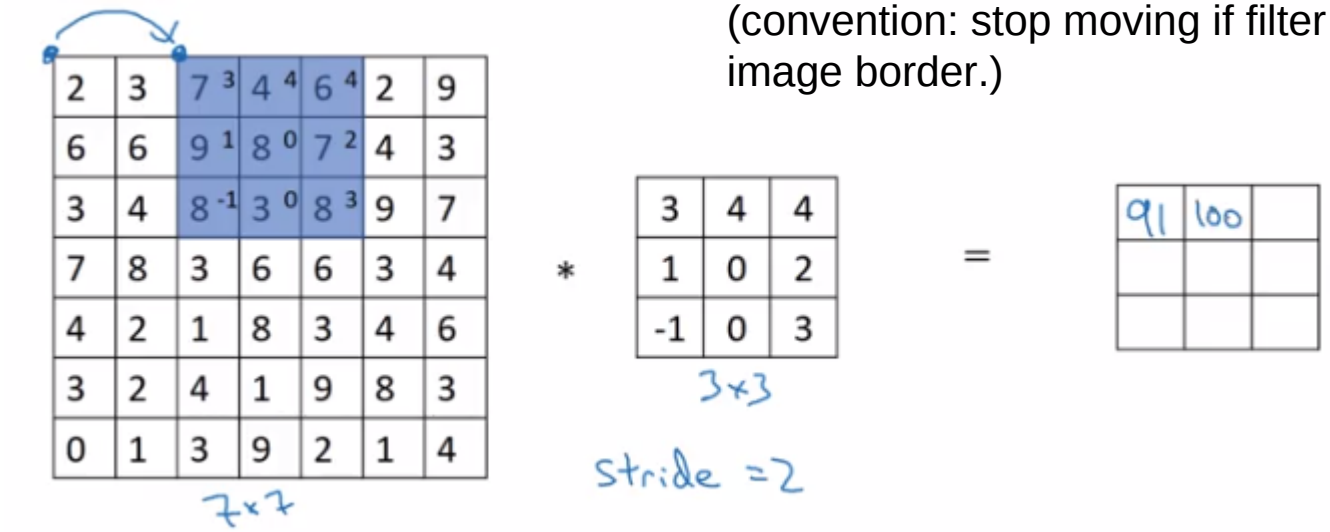
- Valid * : p=0 → nxn * fxf = (n-f+1) x (n-f+1)
- Valid * : p>0 → nxn * fxf = (n+2p-f+1) x (n+2p-f+1)
- (so i/p and o/p size is same) = nxn

Padding

- image *shrinks* after convolution.
  Input n*n image, convolve with f*f filter ⇒ output shape = (n-f+1) * (n-f+1)
  downside:
- image shrinks on every step (if 100 layer → shrinks to very small images)
- pixels *at corner* are less used in the output
- ⇒ *pad the image* so that output shape is invariant.
- if p = padding amount (width of padded border)
  → output shape = (n+2p-f+1) * (n+2p-f+1)
  Terminology: *valid* and *same* convolutions:
- **valid** convolution: no padding, output shape = (n-f+1) * (n-f+1)
- **same** convolution: output size equals input size. i.e. filter width p = (f-1) / 2 (only works *when f is odd* — this is also a convention in CV, partially because this way there'll be a central filter)

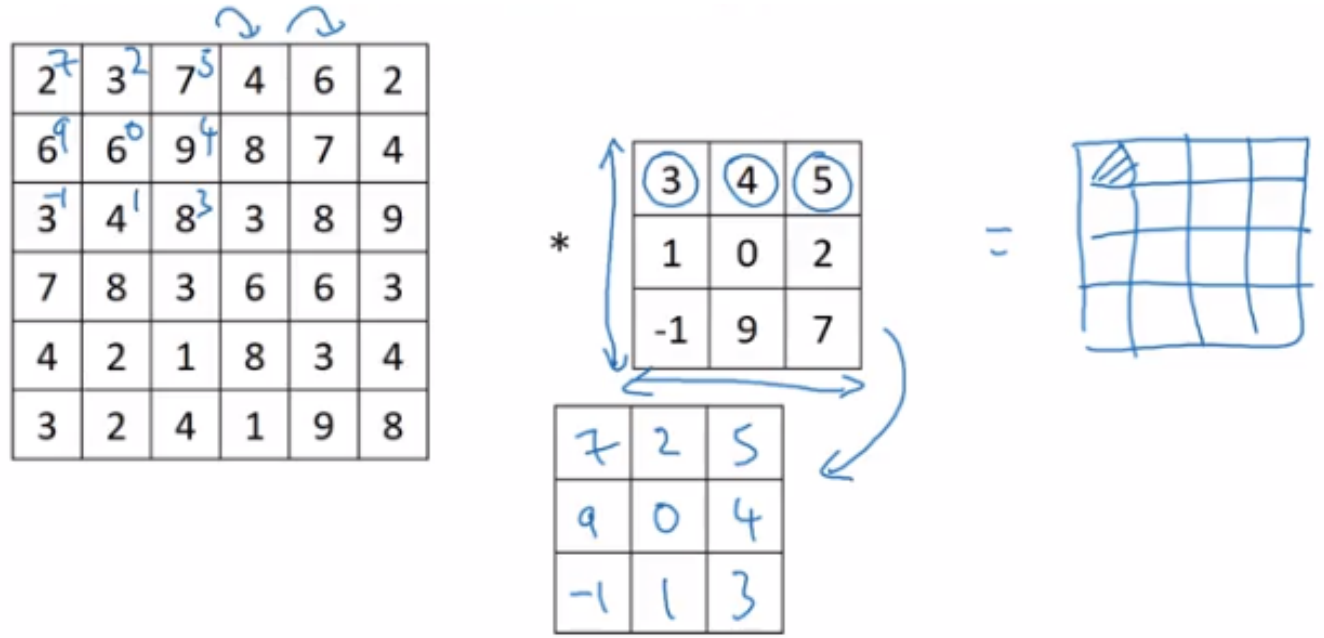
Strided Convolution

- Example stride = 2 in convolution:



- if input image n*n, filter size f*f, padding = f, stride = s
  ⇒ output shape = (floor((n+2p-f)/s) + 1) * (floor((n+2p-f)/s) + 1)
- nxn * fxf = $\left(\frac{n+2p-f+1}{2}\right) \times \left(\frac{n+2p-f+1}{2}\right)$

Note on cross-correlation v.s. convolution

- The operation discribed before is called "cross-correlation".
- 

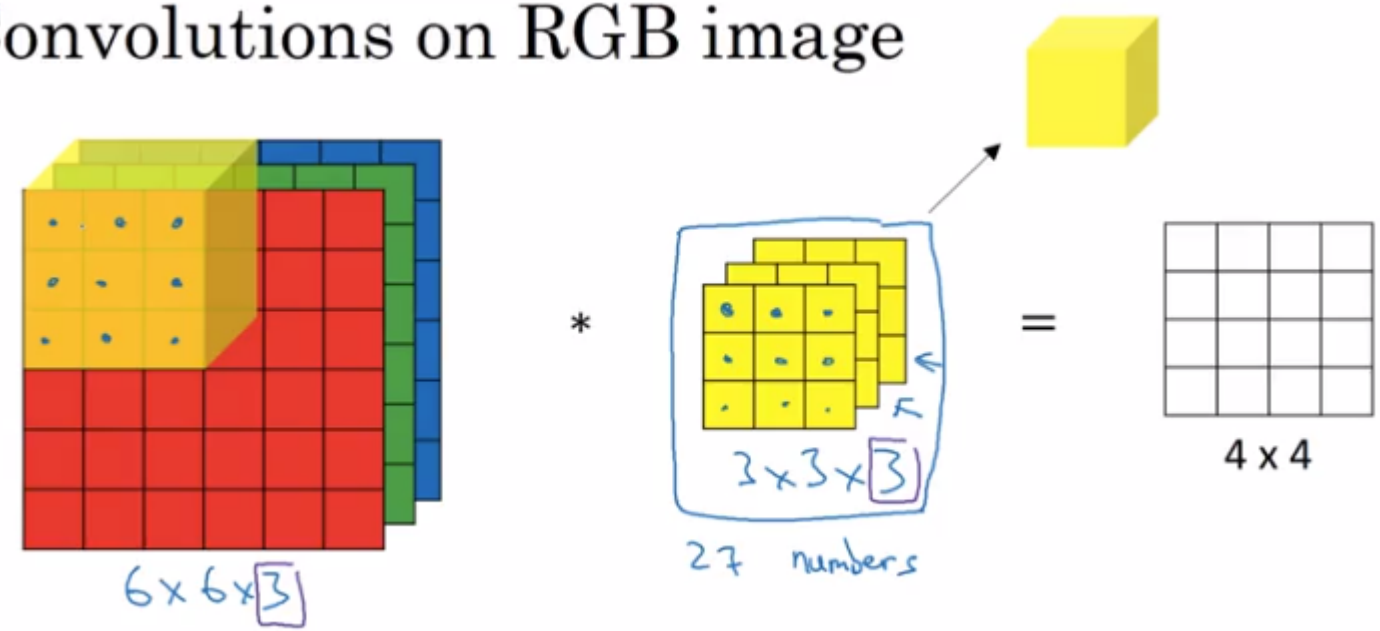
- (Why doing the flipping in math: to ensure assosative law for convolution — (A*B*)C=A(*B*C).)

Convolution Over Volume

- example: convolutions on RGB image

- image size = 6x*6x*3 = height * width * #channels
- filter size = 3x*3x*3, (convention: *filter's #channels matches the image*)
- output size = 4x*4(x*1) — **output is 2D for each filter**.

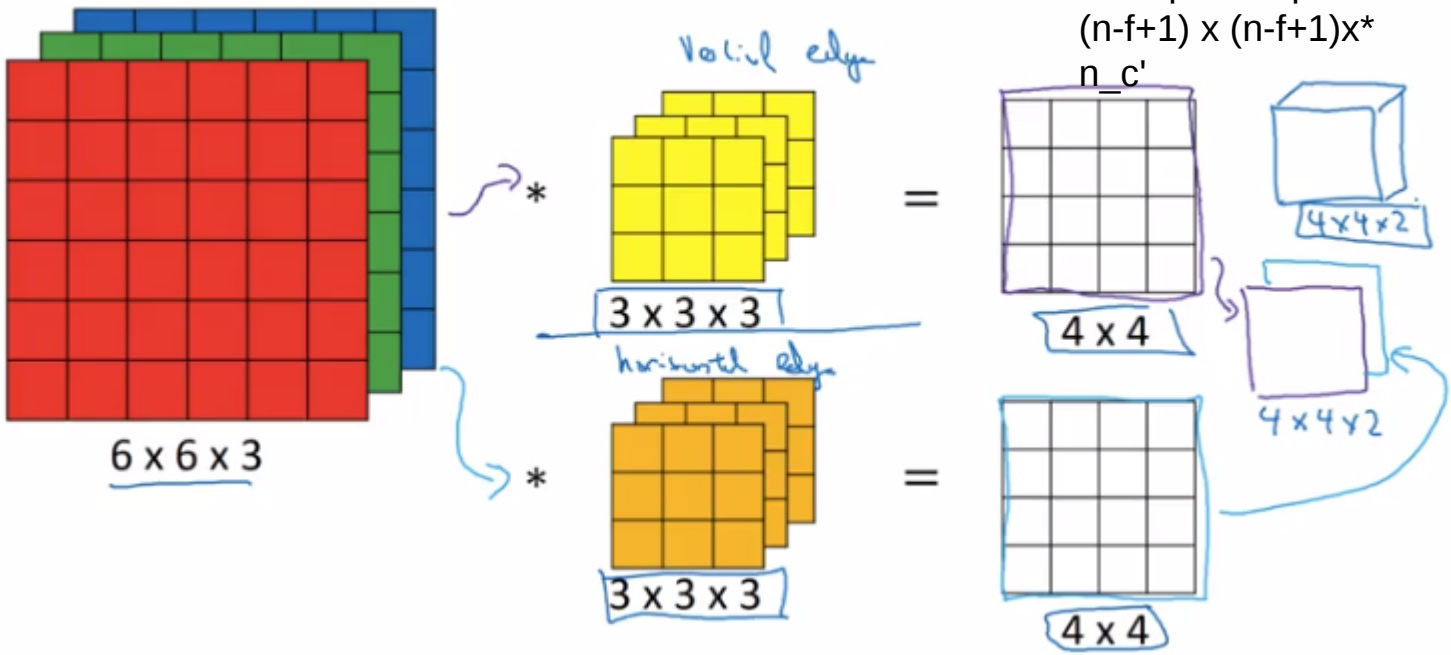
Convolutions on RGB image



Summary of dimensions:
input shape = nxnxn_c
filter shape = fxfxn_c

Multiple filters:
take >1 filters
stack outputs together to form an *output volume*.

filters = n_c'
⇒ output shape = (n-f+1) x (n-f+1)x* n_c'

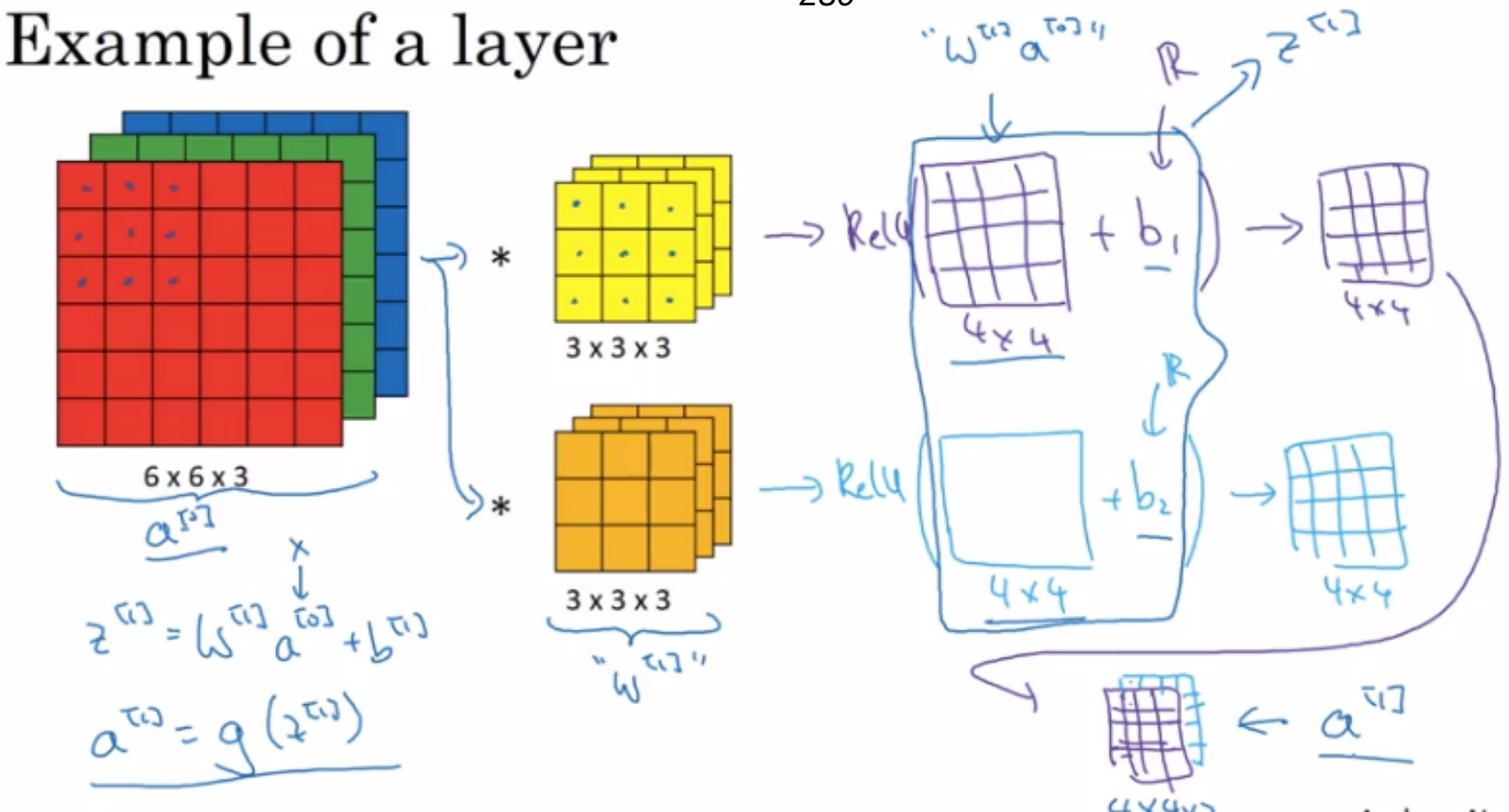


One Layer of a Convolutional Network

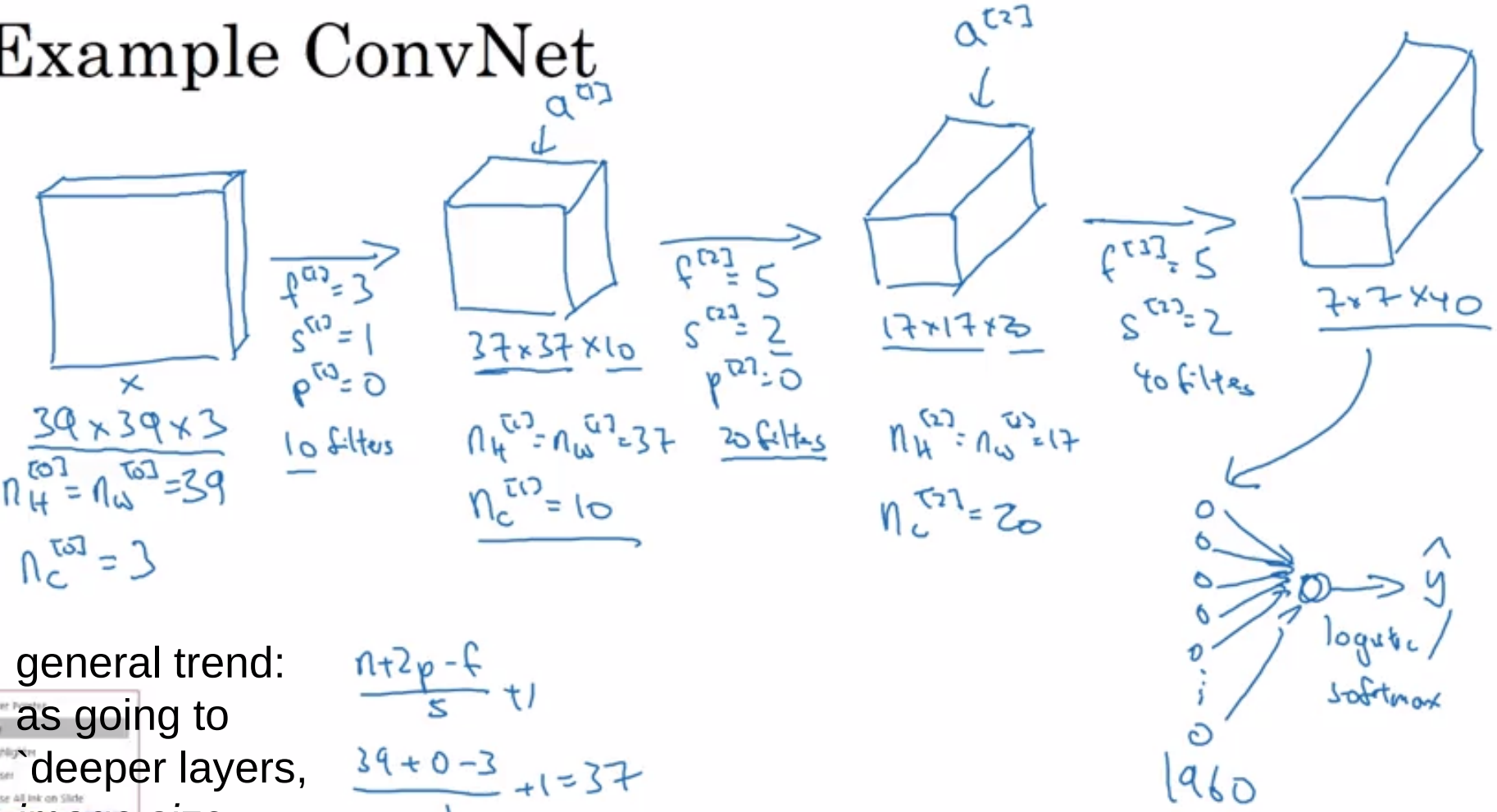
For each filter's output: *add bias b,*
then apply nonlinear activation function.

One layer of a CNN:

with analogy to normall NN:
linear operation (matrix mul V.S. convolution)
bias
nonlinear activation
difference: Number of parameters doesn't depend on input dimension: even for very large images.
For 10 filters: # parameters = (3x3x3 +1bias)x10 = 280



Example ConvNet



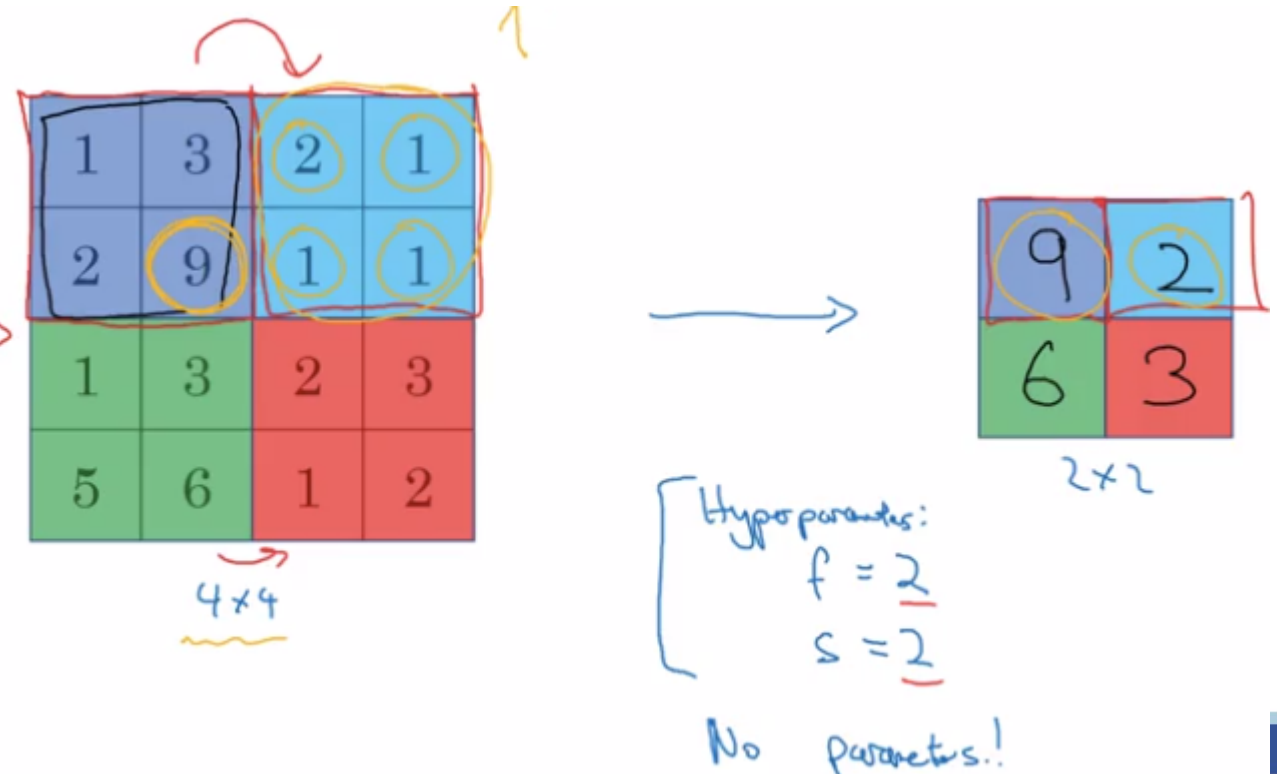
general trend:
as going to deeper layers, *image size shrinks, #channels increases*.

$$\frac{n+2p-f}{s}+1$$

$$\frac{39+0-3}{1}+1=37$$

Pooling Layers

- Pooling layers makes CNN more robust.
- **Max pooling**
  divide input into regions, take max of each region.
  Hyperparams:
- (common choice) *filter size f=2 or 3, strid size s=2, padding p=0.*
- note: *no params to learn* for max pooling layer, pooling layer not counted in #layers (conv-pool as a single layer)

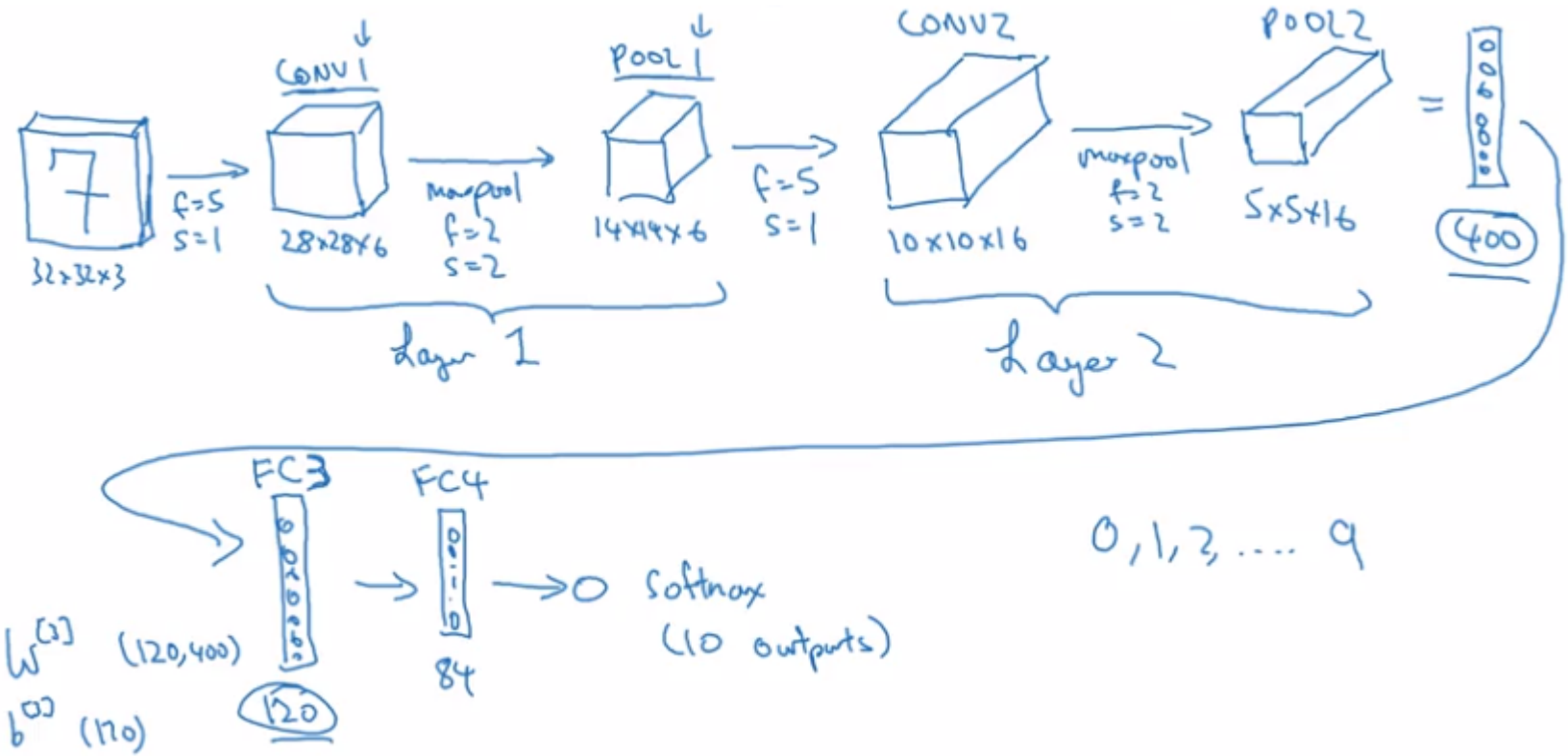


Intuition: a large number indicates a detected feature in that region → preserved after pooling.

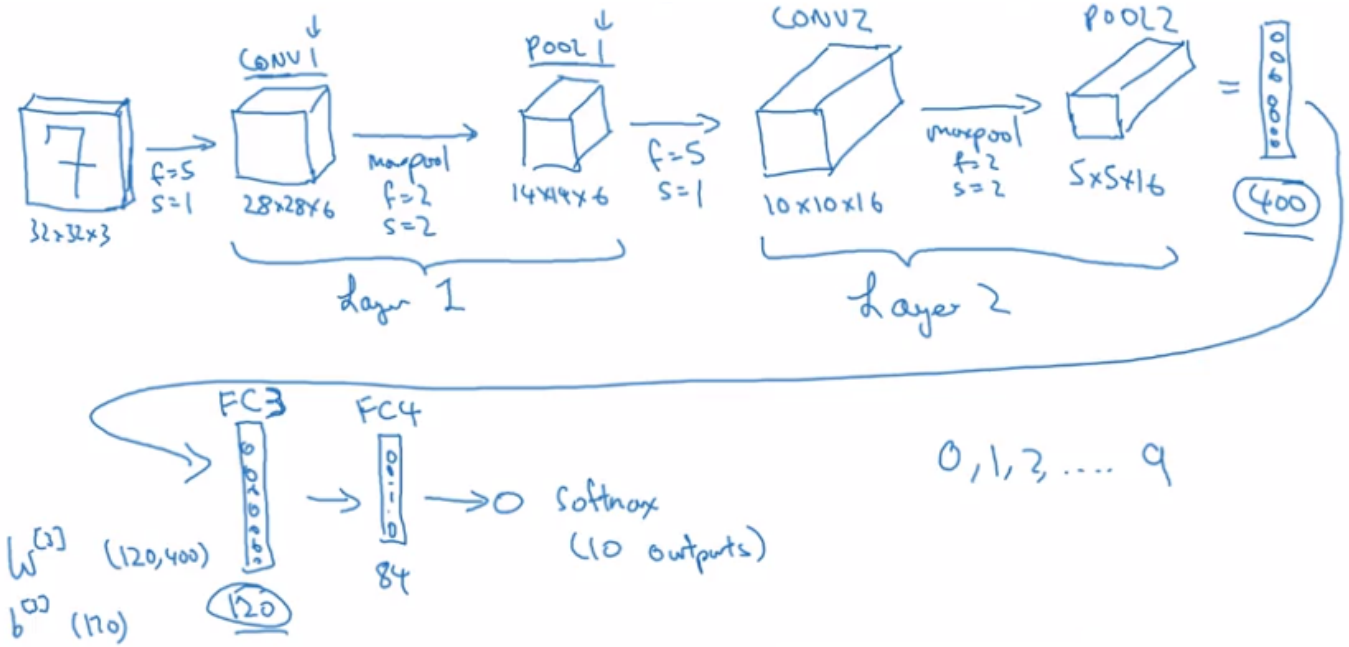
Formula of dimension floor((n+2p-f+1)/s + 1) holds for POOL layer as well.
Output of max pooling: the same #channels as input (i.e. do maxpooling on each channel).

Average pooling
Less often used than max pooling.
Typical usecase: collapse 771000 activation into 111000.

CNN Example
LeNet-5



CNN Example *LeNet-5*



	Activation shape	Activation Size	# parameters
Input:	(32,32,3)	3,072 $a^{[0]}$	0
CONV1 (f=5, s=1)	(28,28,8)	6,272	208
POOL1	(14,14,8)	1,568	0
CONV2 (f=5, s=1)	(10,10,16)	1,600	416
POOL2	(5,5,16)	400	0
FC3	(120,1)	120	48,001
FC4	(84,1)	84	10,081
Softmax	(10,1)	10	841

Why Convolutions?

- **Why Convolutions?**
- 2 main advantages of CONV over FC: *parameter sharing; sparcity of connections.*
- **Parameter sharing**:
  A feature detector useful in one part of img is probably useful in another part as well.
  → no need to learn separate feature detectors in different parts.

- **Sparcity of connections**:
  For each output value depends only on a small number of inputs (the pixels near that position)

- Invariance to translation...

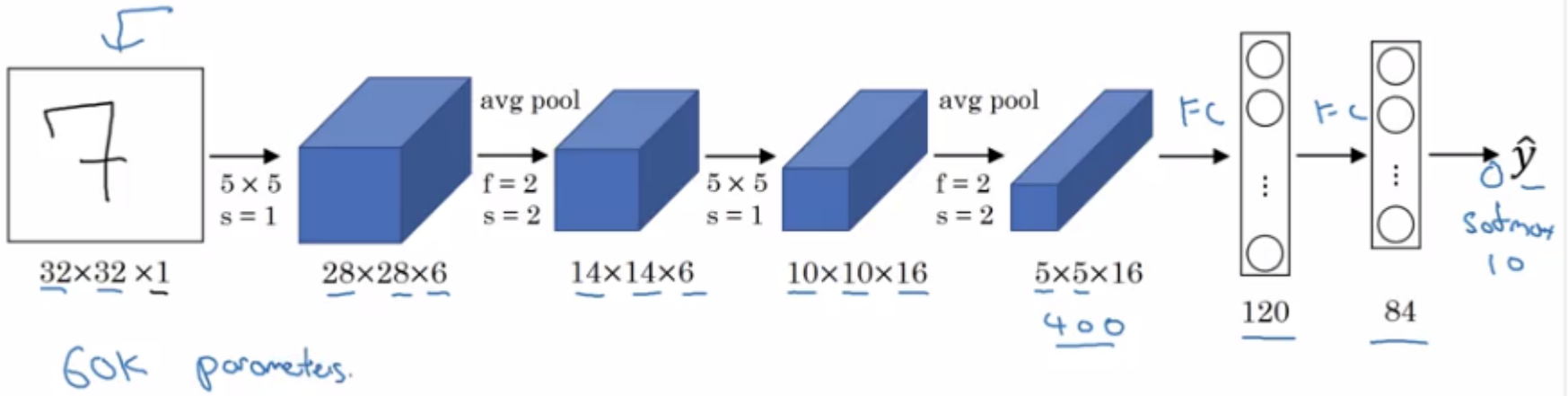
Case Studies

- classic networks:
  - LeNet-5
  - AlexNet
  - VGG
- ResNet (152-layer NN)
- Inception

LeNet-5 (1998)

- **LeNet-5**(1998)

  Goal: recognize hand-written digits.
  image → 2 CONV-MEANPOOL layers, all CONV are valid (without padding) → 2 FC → softmax

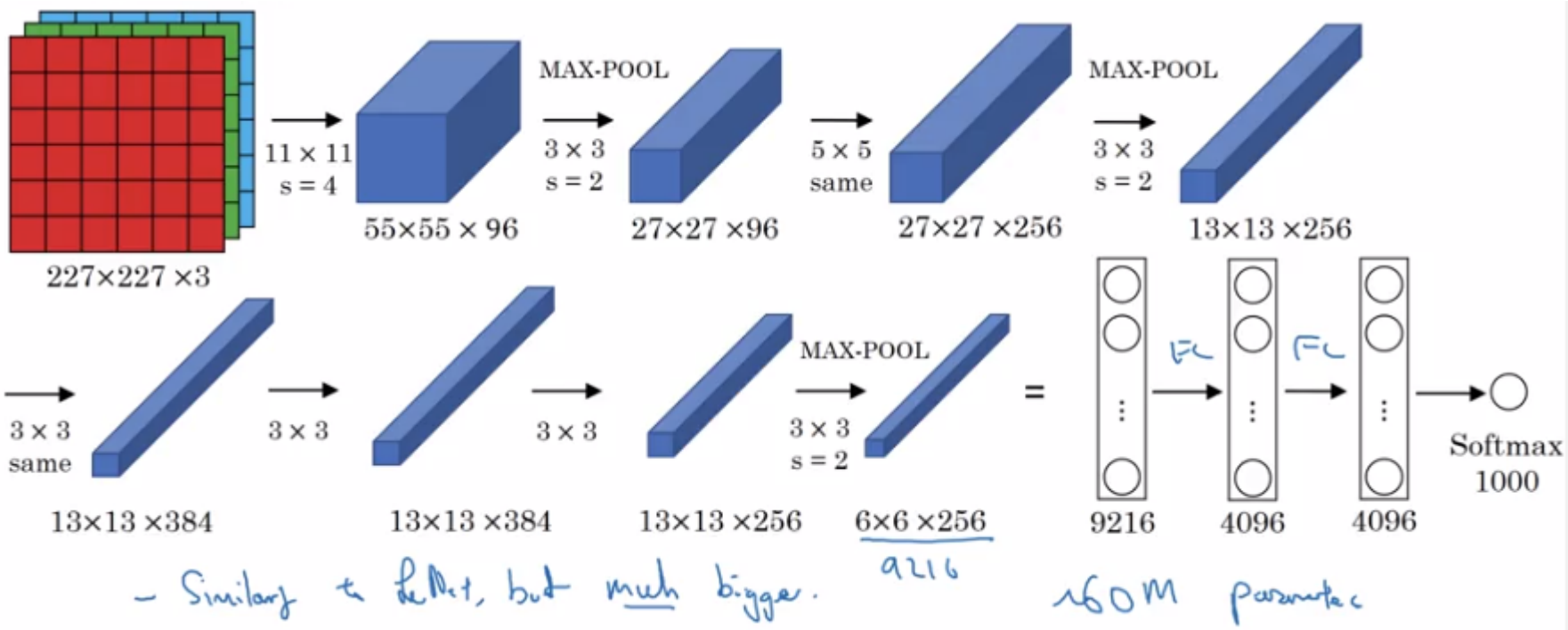


takeaway (patterns still used today):

- as go deeper, n_H, n_W goes down, n_C goes up
- conv-pool repeated some times, then FC-FC-output
- used sigmoid/tanh as activation, instead of ReLU. has non-linearity after pooling

AlexNet

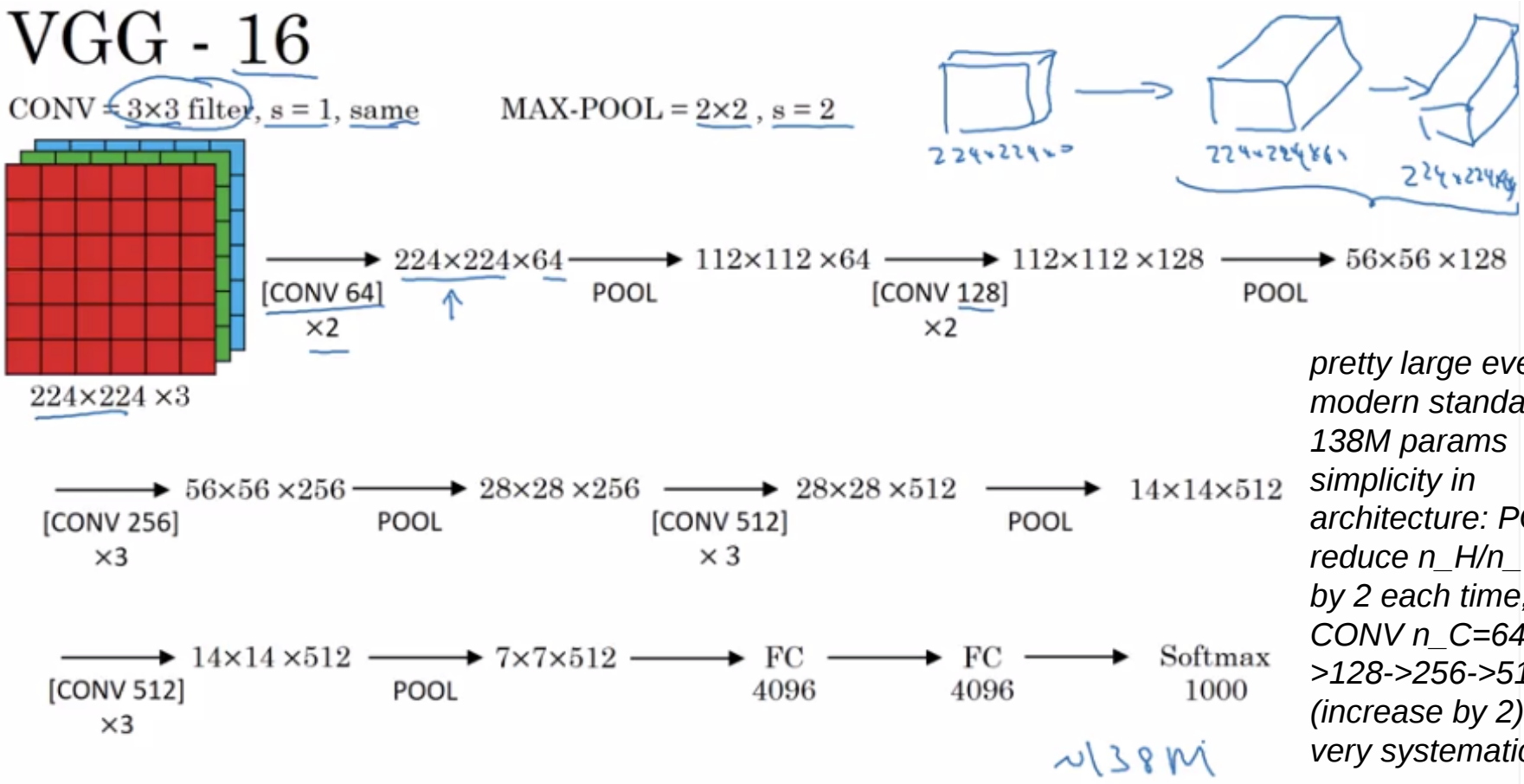
- Same pattern: conv-maxpool layers → FC layers → softmax but much more params.



- use ReLU as activation, multi-GPU training
- "local response normalization" (LRN): normalize across all channels (not widely used today), a lot hyperparams to pick

VGG-16

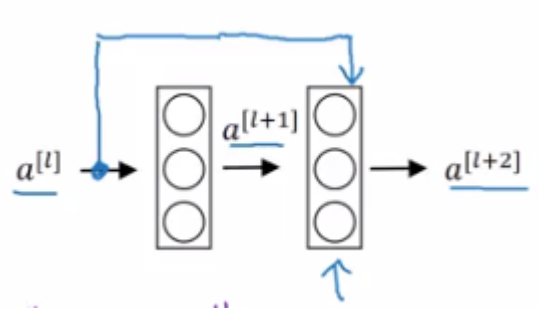
- Much less hyperparams:
  All **CONV: *33,s=1,padding=same, MAXPOOL: 22,s=2***
  → e.g. "(CONV 64) * 2" meaning 2 conv layers (3*3,s=1,padding=same) of 64 channels.

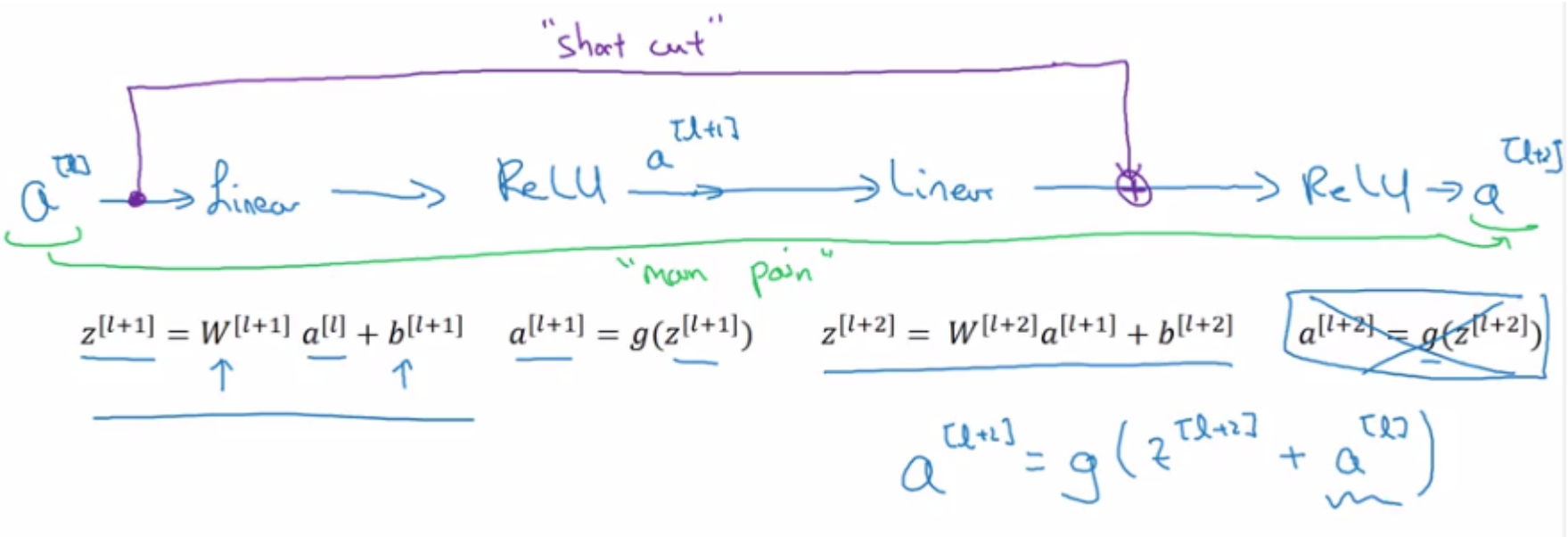


pretty large even by modern standard: 138M params simplicity in architecture: POOL reduce n_H/n_W by 2 each time; CONV n_C=64->128->256->512 (increase by 2), very systematic.

ResNets

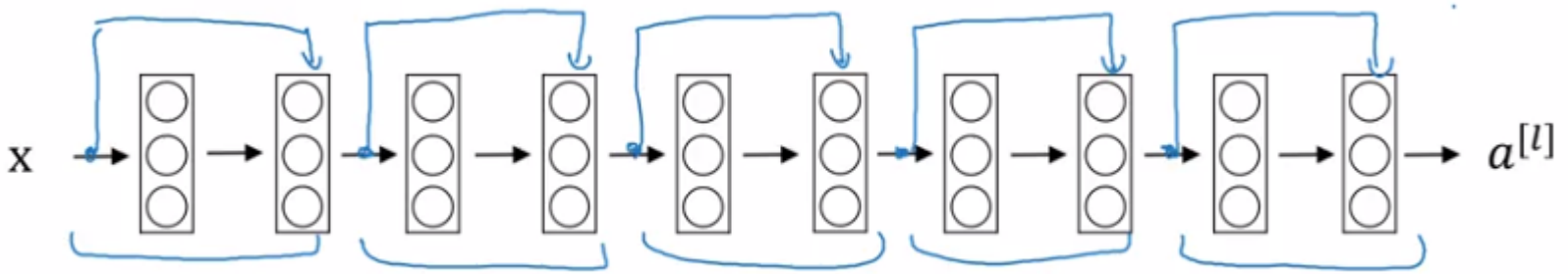
- Very deep NN are hard to train → ResNet: *skip connections*, to be able to train ~100 layers NN.
- Normal NN: from a[l] to a[l+2], two linear & ReLU operations. "*main path*". ResNet: a[l] taks shortcut and *goes directly to a[l+2]'s non-linearity*. "*shortcut*" / "*skip connection*".

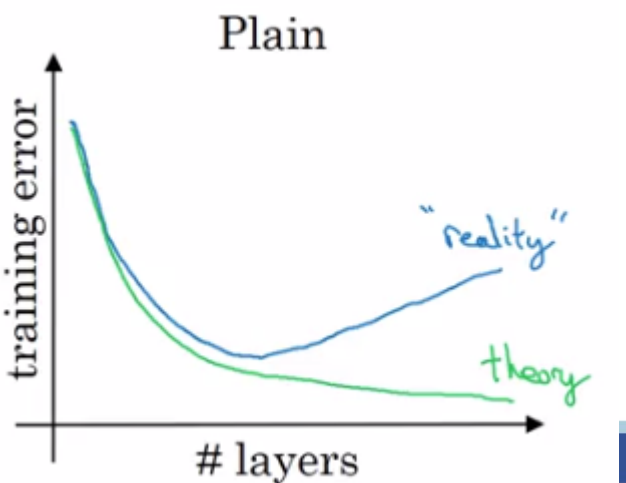


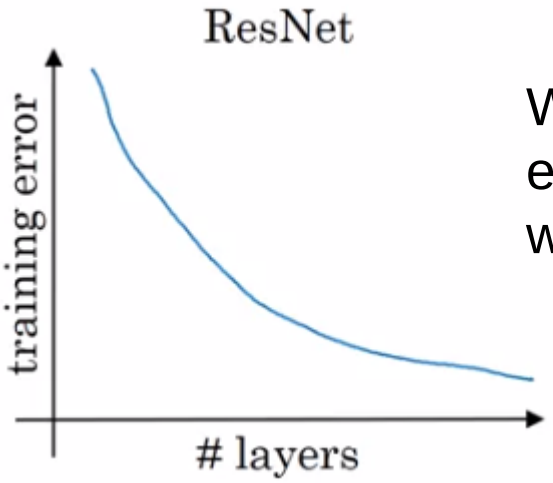


ResNets

- Using residual block allows training *very deep* NN:
  stack them to get ResNet (i.e. add shortcuts to "plain" NN).
- Problem of training plain NN: *training error goes up (in practice) when having deeper NN*.
  Because deeper NN are harder to train (vanishing/exploding gradients, etc.)



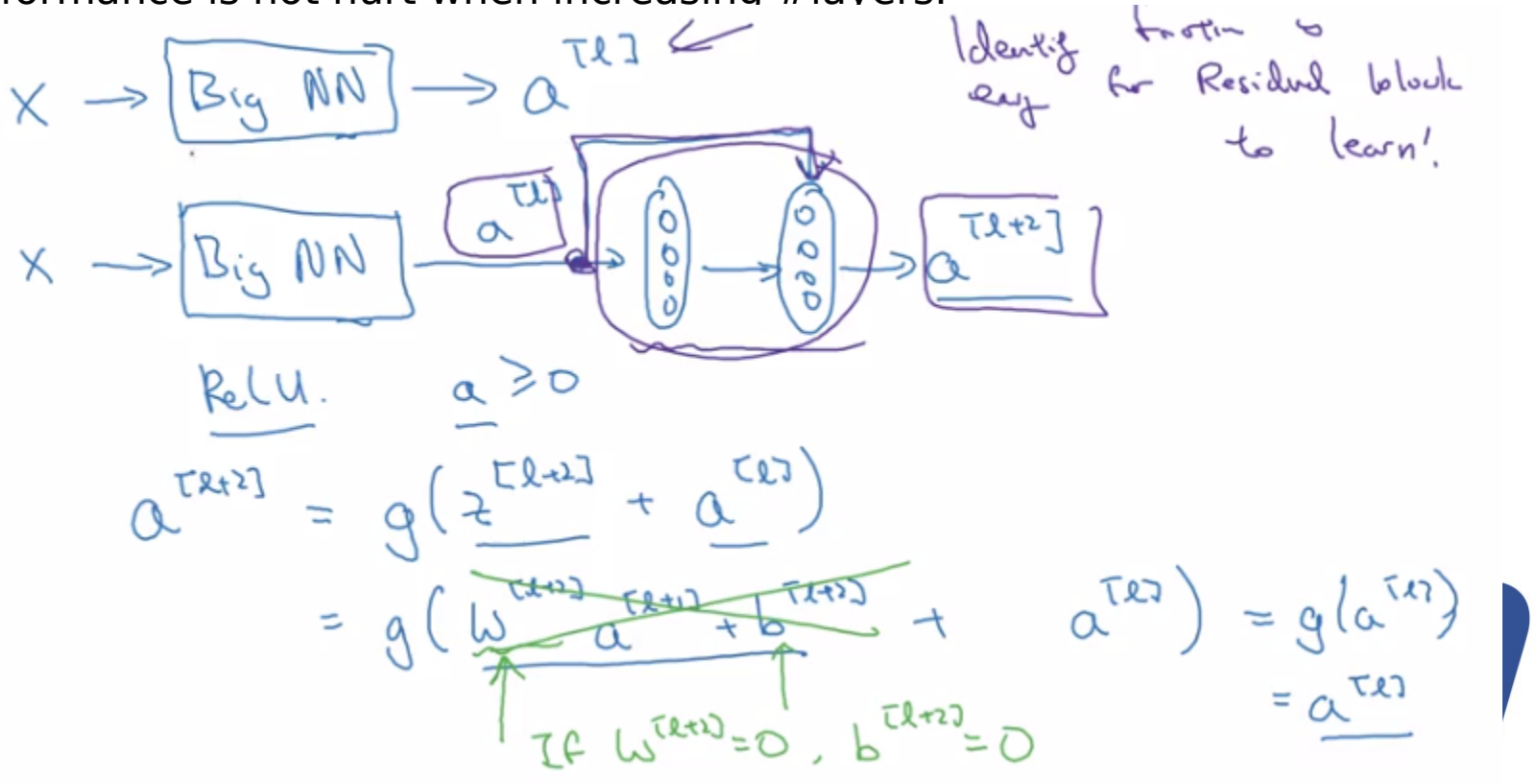




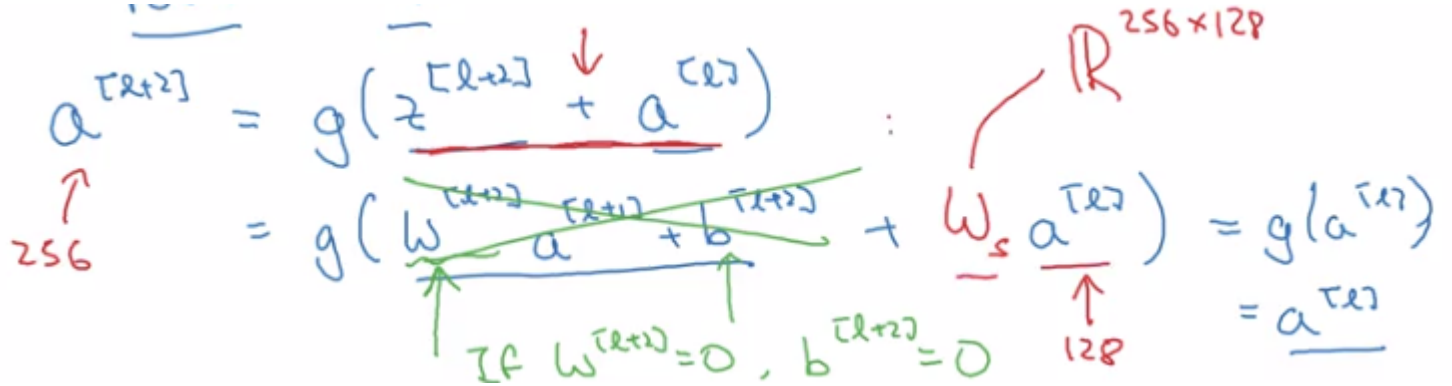
With ResNet: training error goes down even with deeper layers.

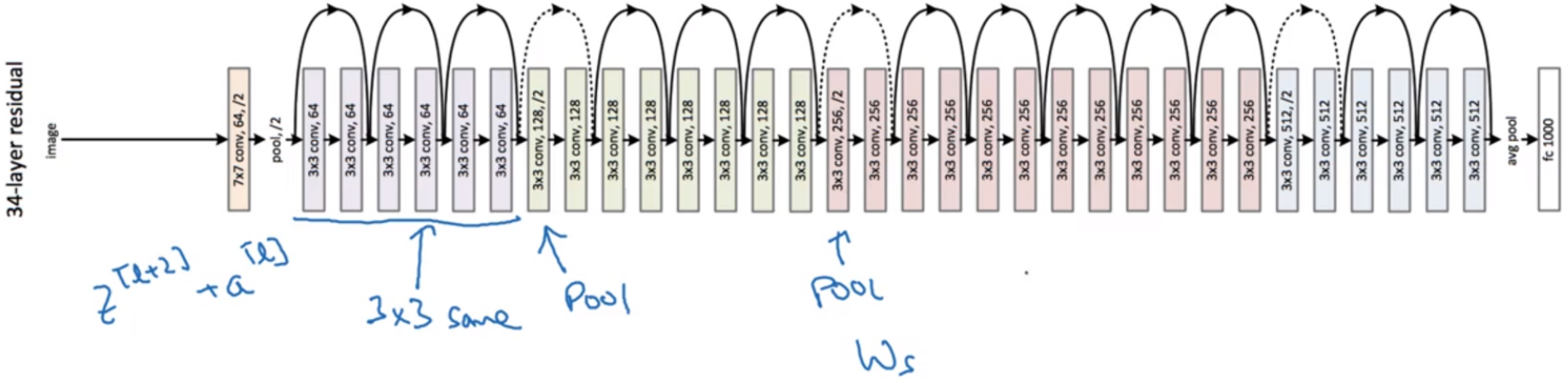
Why ResNets Work

- a[l+2] = g(z[l+2] + a[l]) = g(w[l+1] * a[l+1] + b[l+1] + a[l])
- → note: when applying weight decay, w can be small (w~=0, b~=0)
  ⇒ a[l+2] ~= g(a[l]) = a[l] (assume g=ReLU)
  ⇒ it's easy to get a[l+2]=a[l], i.e. *identity function from a[l] to a[l+2] is easily learned*
  → whereas in plain NN, it's difficult to learn an identity function between layers, thus more layers make result *worse*
  → adding 2 layers doesn't hurt the network to learn a shallower NN's function, i.e. performance is not hurt when increasing #layers.
  →



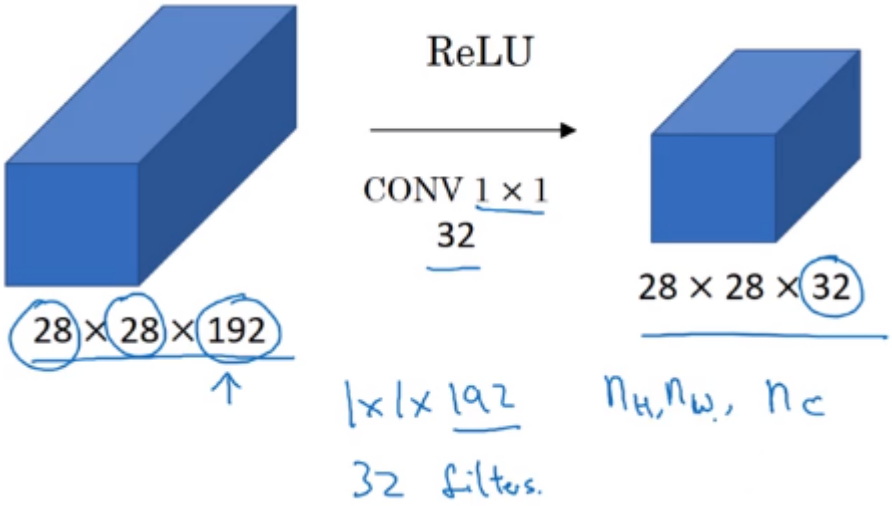
- z[l+2] and a[l] have the same dimension (so that they can be added in g) → i.e. many "same" padding are used to preserve dimension.
- If their dimensions are not matched (e.g. for pooling layers) → add extra w_s to be applied on a[l].





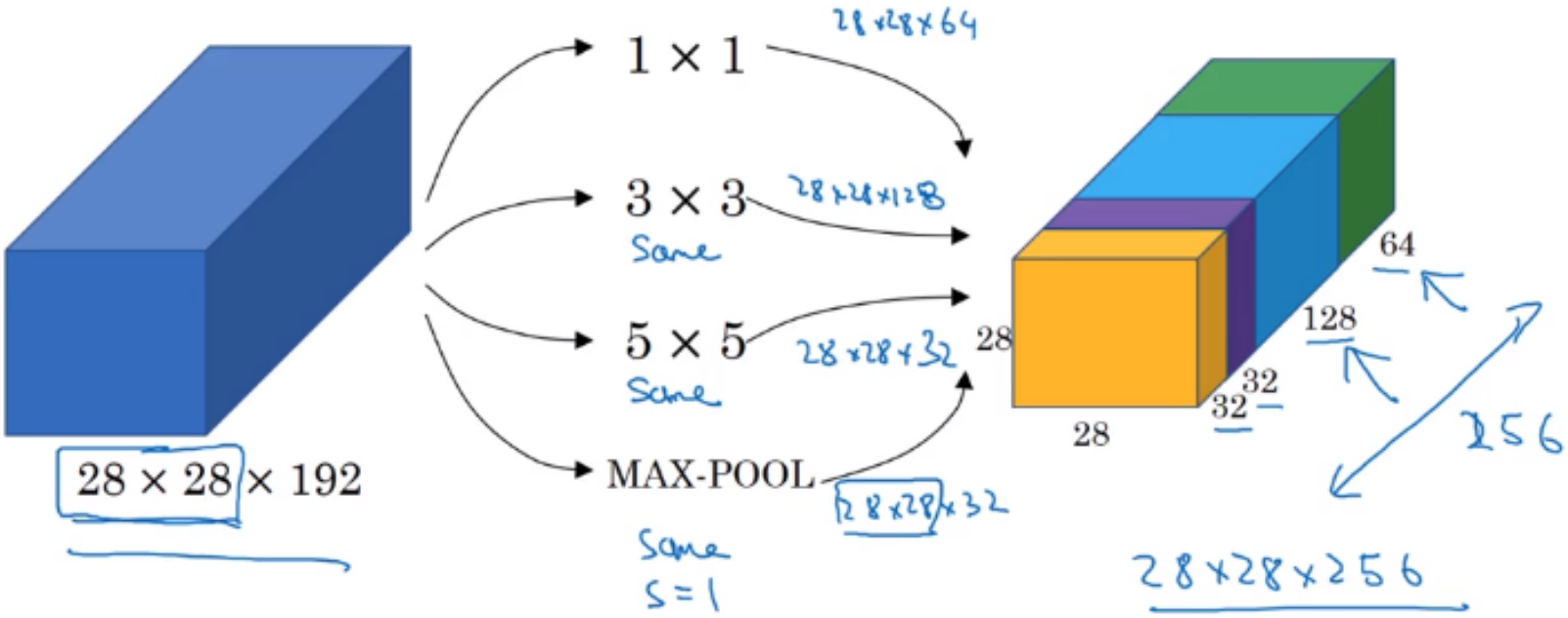
Networks in Networks and 1x1 Convolutions

- Using 1*1 conv: for one single channel, just multiply the input image(slice) by a constant...
  But for >1 channels: each output number is inner prod of input channel "slice" and conv filter.



Inception Network Motivation

- Instead of choosing filter size, *do them all in parallel.*
  note: use SAME padding & stride=1 to have the same n_H, n_W



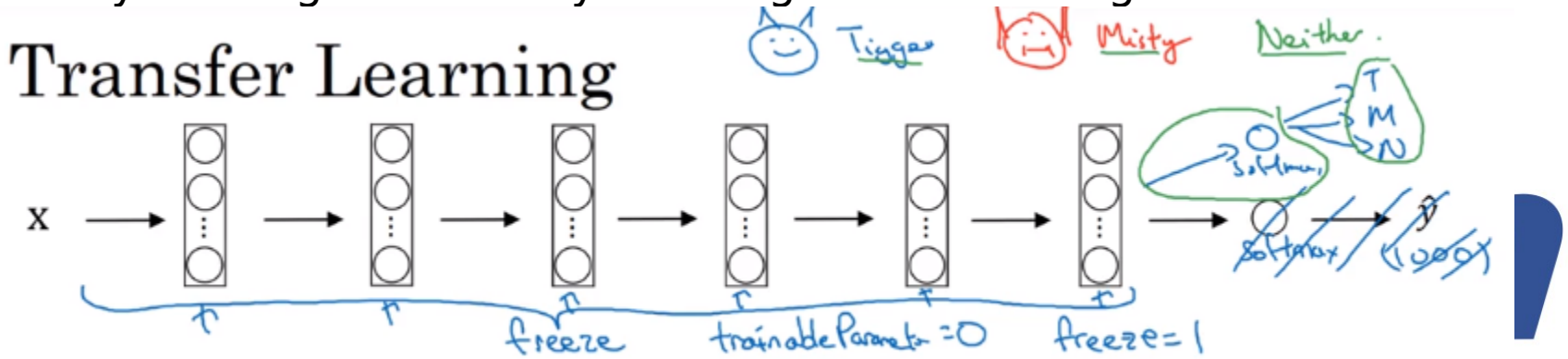
- Problem: computation cost.

Using Open-Source Implementation

- Transfer Learning

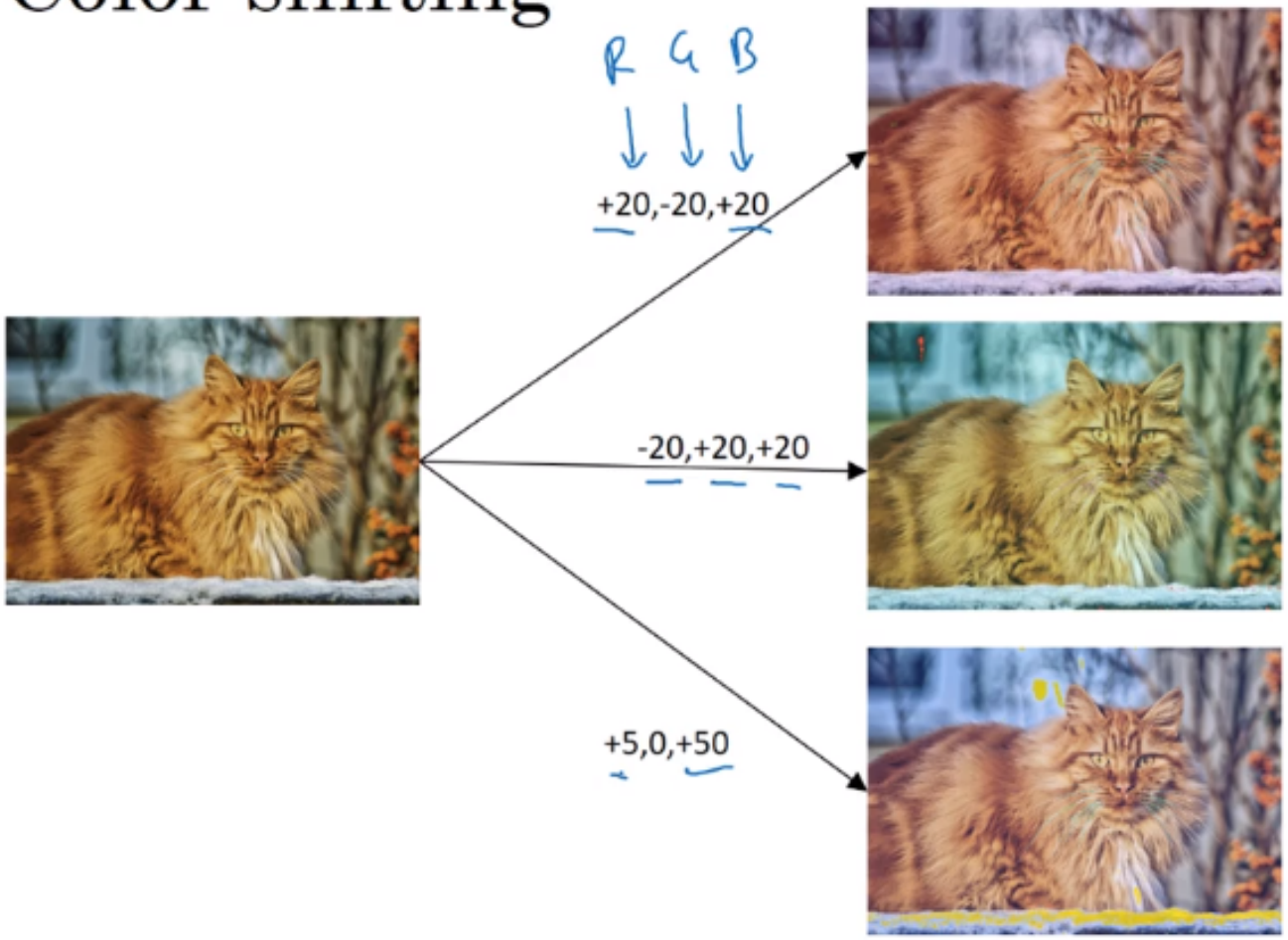
Download weights of other's NN as pretrained params.
→ pretrained params are trained on huge datasets, and takes weeks to train on multiple GPUs.
example: cat detector
3 class: tigger/misty/neither

- training set at hand is small
- → download both code and weights online
- e.g. ImageNet NN
  → change last layer's softmax
  → all Conv/Pool layers set *frozen* (not trainable)
  → only training softmax layer's weight with training set.



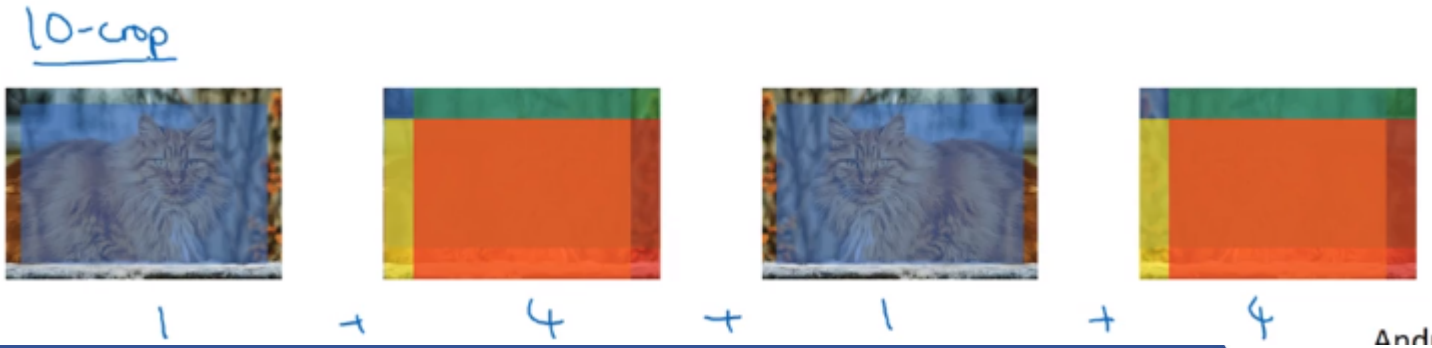
Data Augmentation

Color shifting



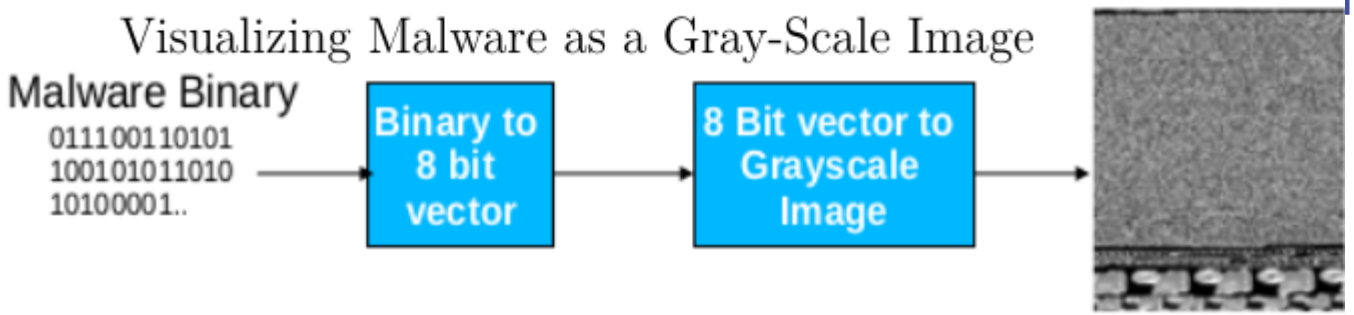
Tips for doing well on benchmarks/winning competitions

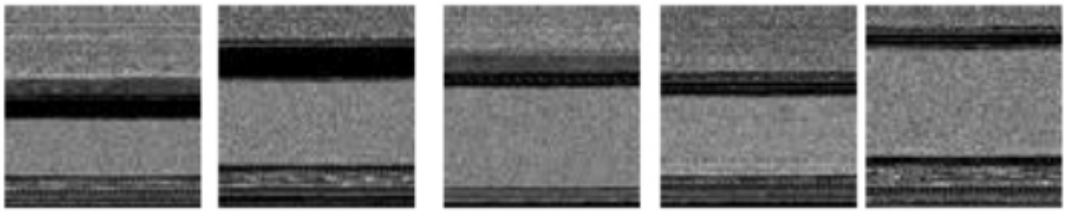
- Ensembling:
- Train several(3~15) NN independently, then *average their outputs*.
- *Multi-crop at test time*
- Predict on multiple versions of test images and average results.
  e.g. 10-crop at test time
  - Image (crops: center, top-left, top-right, bottom-left, bottom-right) + mirror-of-image (crops: center, top-left, top-right, bottom-left, bottom-right)



Case Study 1 - CNN for malware classification

David Gibert's work





Rammit samples

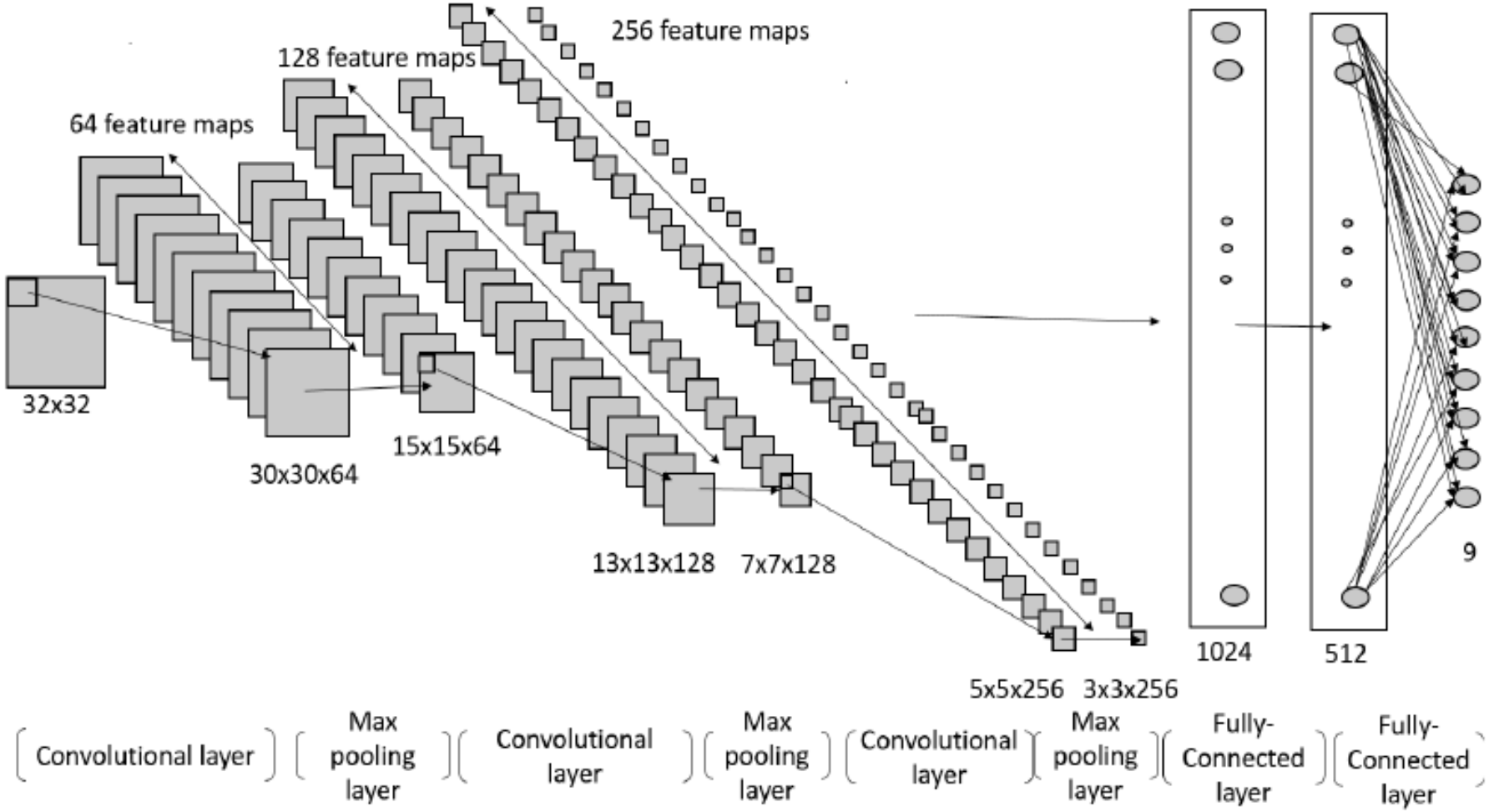
Ramnit. This type of malware is known to steal your sensitive information such as user names and passwords and it also can give access to an illegitimate user to your computer.

Kelihos_ver3 samples

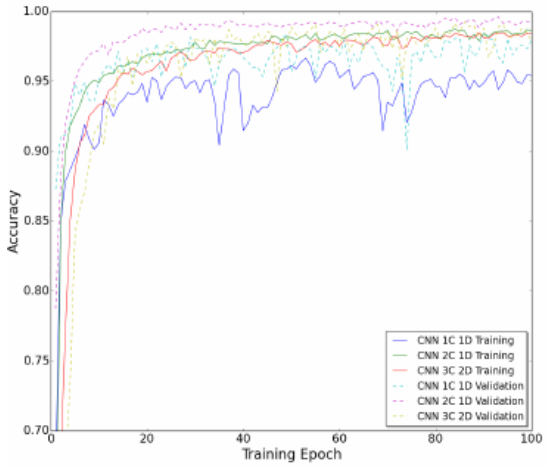
Kelihos_ver3. Kelihos botnet. Kelihos is mainly involved in spamming and theft of bitcoins. This trojan can give access and control of your computer to an illegitimate user and can also communicate with other computers about sending spam emails, run malicious programs and steal sensitive information.

A given malware binary file can be read as a vector of 8 bit unsigned integers and organized into a 2D array. This array can be visualized as a gray scale image in the range [0,255].

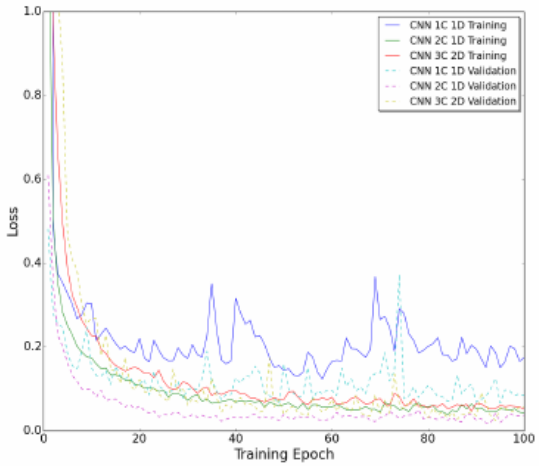
CNN for malware classification



Case Study 1 - CNN for malware classification



(a) Training & Validation accuracy



(b) Training & Validation Cross-Entropy

	Rammit	Lollipop	Kelihos_ver3	Vundo	Simda	Tracur	Kelihos_ver1	Obfuscator.ACY	Gatak
Rammit	1534	0	0	0	1	0	0	1	5
Lollipop	0	2375	0	0	0	4	0	0	98
Kelihos_ver3	0	1	2937	0	0	0	0	0	4
Vundo	0	0	0	472	0	0	2	0	1
Simda	0	0	0	1	41	0	0	0	0
Tracur	2	0	0	2	0	737	0	4	10
Kelihos_ver1	0	0	0	0	0	0	387	0	11
Obfuscator.ACY	0	0	0	0	0	1	0	1219	8
Gatak	0	0	0	0	0	2	0	0	1011

Recurrent Neural Networks

Recurrent neural network (RNN) is a type of deep learning model that is mostly used for analysis of sequential data (time series data prediction).

There are different application areas of RNN:

Speech recognition: Input: audio clip -> Output: text

Music generation: Input: integer referring to genre (or an empty set) -> Output: music

Sentiment classification: Input: text -> Output: ratings

DNA sequence analysis: Input: DNA (alphabet) -> Output: label part of the DNA sequence

Machine translation: Input: text -> Output: text translation

Video activity recognition: Input: video frames -> Output: identification of the activity

Name entity recognition: Input: sentence -> Output: identify people within it.

Why Recurrent Neural Networks

Traditional feedforward neural networks do not share features across different positions of the network.

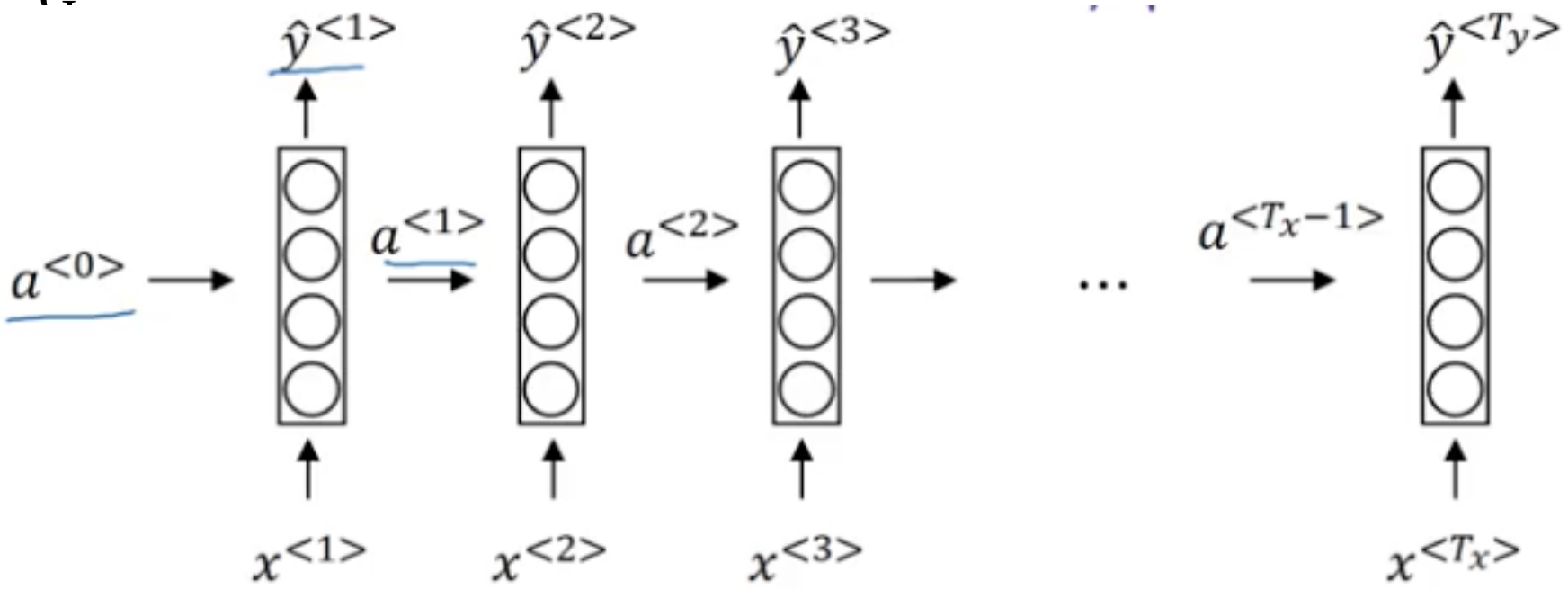
In other words, these models assume that all inputs (and outputs) are independent of each other.

Traditional neural networks require the input and output sequence lengths to be constant across all predictions.

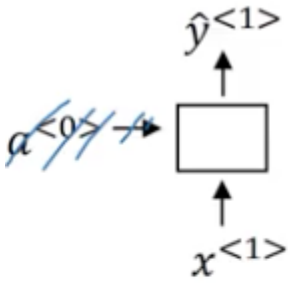
This FFNN model would not work in sequence prediction since the previous inputs are inherently important in predicting the next output.

For example, if you were predicting the next word in a stream of text, you would want to know at least a couple of words before the target word.

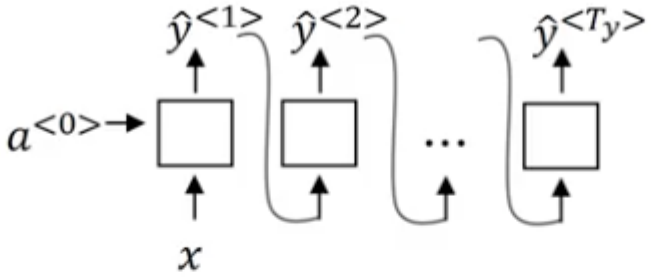
Recurrent Neural Networks



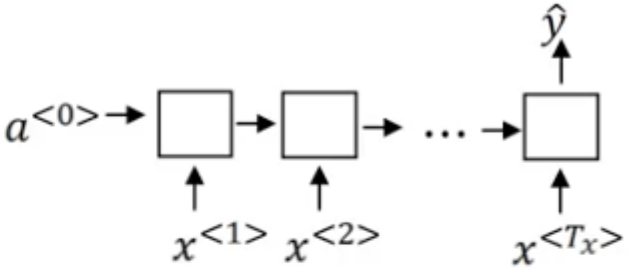
RNN Architectures



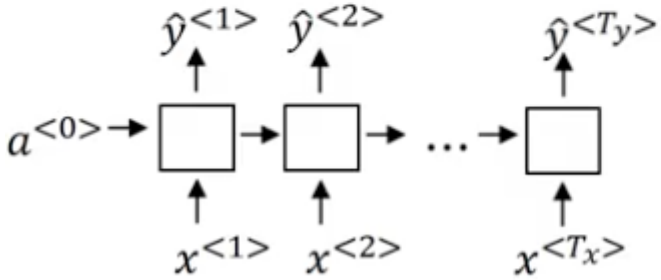
One to one



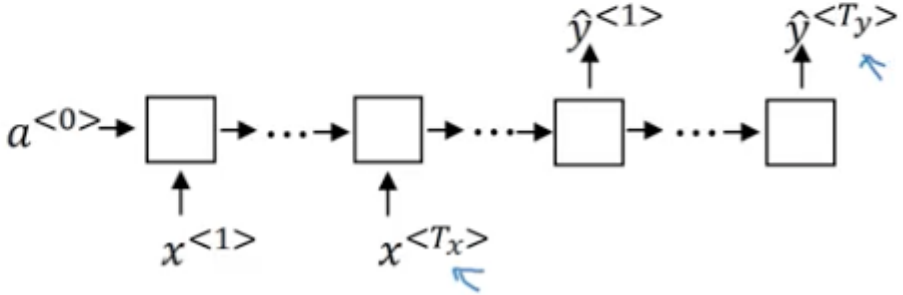
One to many



Many to one

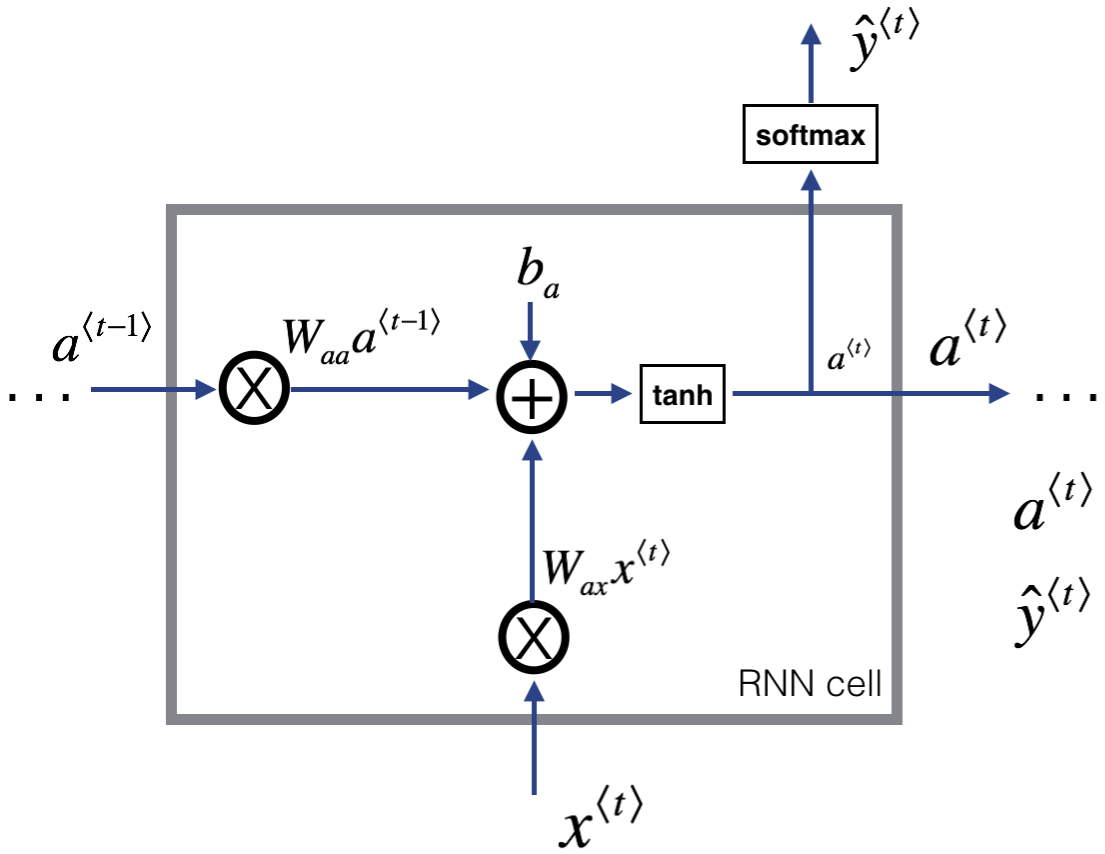


Many to many



Many to many

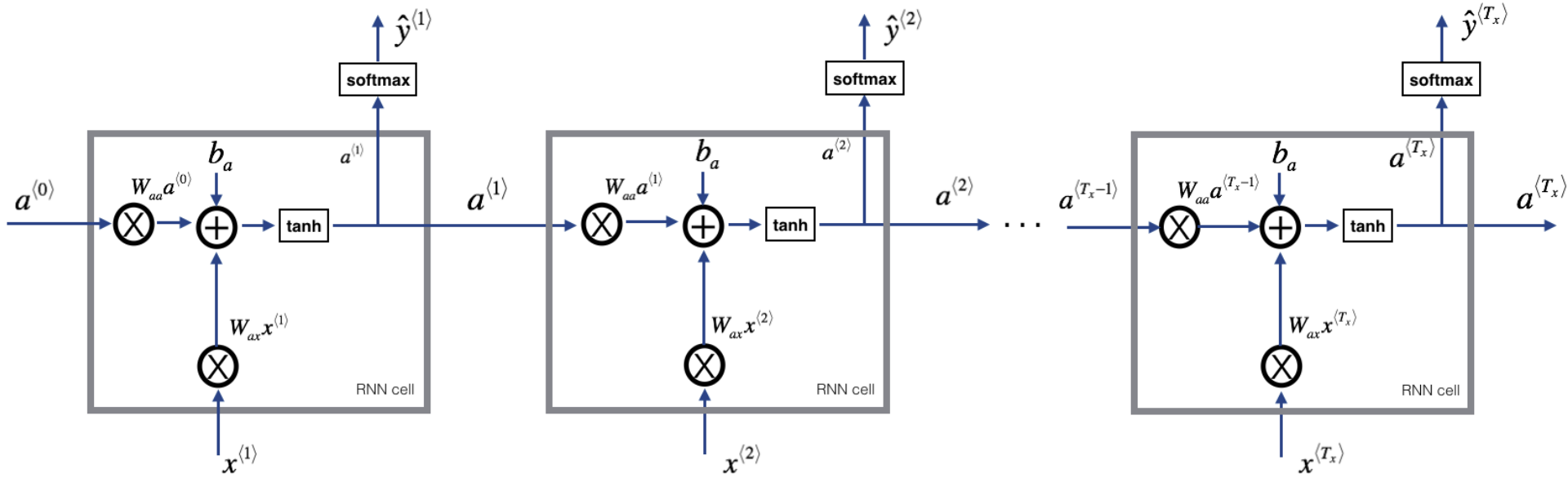
An RNN Cell



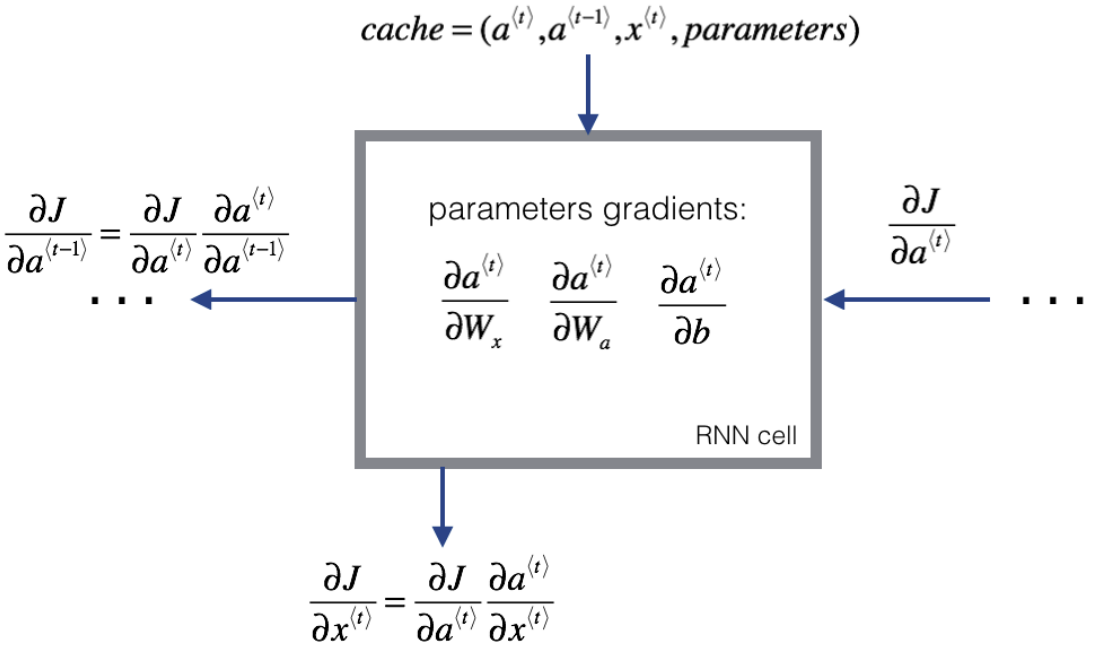
$$a^{\langle t \rangle} = \tanh(W_{ax}x^{\langle t \rangle} + W_{aa}a^{\langle t-1 \rangle} + b_a)$$

$$\hat{y}^{\langle t \rangle} = soft\max(W_{ya}a^{\langle t \rangle} + b_y)$$

RNN Forward Pass



RNN Backward Pass



$$a^{\langle t \rangle} = \tanh(W_{ax} x^{\langle t \rangle} + W_{aa} a^{\langle t-1 \rangle} + b)$$

$$\frac{\partial \tanh(x)}{\partial x} = 1 - \tanh(x)^2$$

$$\frac{\partial a^{\langle t \rangle}}{\partial W_{ax}} = (1 - \tanh(W_{ax} x^{\langle t \rangle} + W_{aa} a^{\langle t-1 \rangle} + b)^2) . x^{\langle t \rangle T}$$

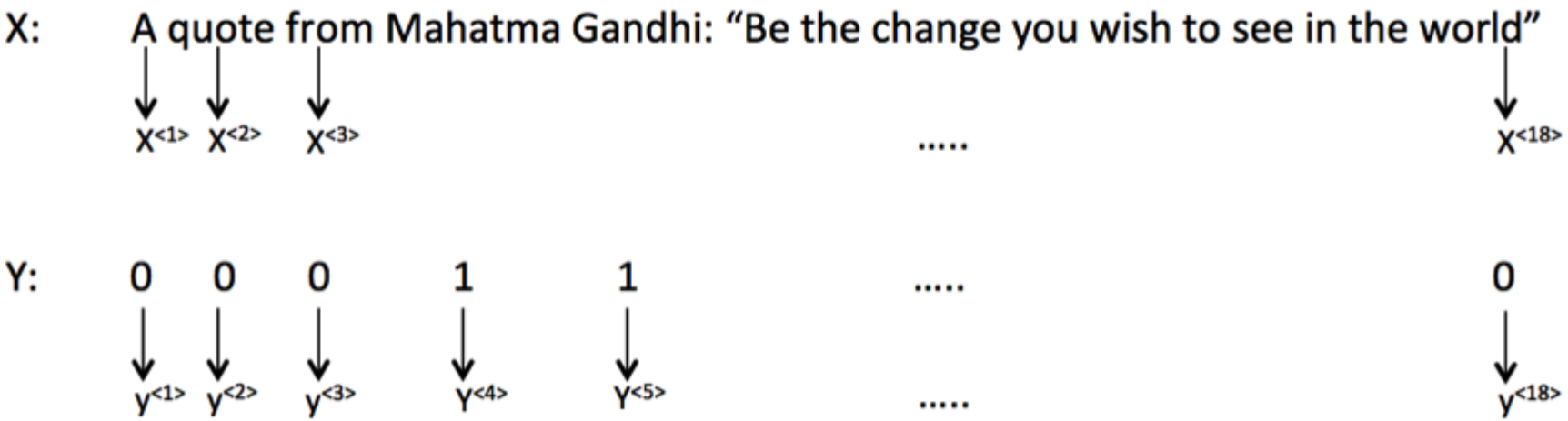
$$\frac{\partial a^{\langle t \rangle}}{\partial W_{aa}} = (1 - \tanh(W_{ax} x^{\langle t \rangle} + W_{aa} a^{\langle t-1 \rangle} + b)^2) . a^{\langle t-1 \rangle T}$$

$$\frac{\partial a^{\langle t \rangle}}{\partial b} = \sum_{batch} (1 - \tanh(W_{ax} x^{\langle t \rangle} + W_{aa} a^{\langle t-1 \rangle} + b)^2)$$

$$\frac{\partial a^{\langle t \rangle}}{\partial x^{\langle t \rangle}} = W_{ax}{}^T . (1 - \tanh(W_{ax} x^{\langle t \rangle} + W_{aa} a^{\langle t-1 \rangle} + b)^2)$$

$$\frac{\partial a^{\langle t \rangle}}{\partial a^{\langle t-1 \rangle}} = W_{aa}{}^T . (1 - \tanh(W_{ax} x^{\langle t-1 \rangle} + W_{aa} a^{\langle t-1 \rangle} + b)^2)$$

Name entity recognition



$x^{<t>}$: t-th element in the input sequence
$x^{(i)<t>}$: t-th element in the input sequence of training examples i
T_x : length of the input sequence
T_y : length of the output sequence
$T_x^{(i)}$: input sequence lenght for training example i
$y^{<t>}$: t-th element in the output sequence
$y^{(i)<t>}$: t-th element in the output sequence of training examples i

Name Entity Recognition

To represent words in a sentence, we come up with a vocabulary (dictionary) that lists all the words and assigns a sequential number to each one.

You can find online dictionaries already prepared for you, which contain 100,000 words.

If a word is not in the vocabulary, you can assign it to the <UNK> (“unknown”) token.

Finally, we use a one-hot representation for each word as a vector of zeros and a one corresponding to the position of the word in the vocabulary list.

Name Entity Recognition

To learn the mapping from X to Y, we might use a standard neural network, where we feed the x<1>, x<2>, … x<t> to obtain y<1>, y<2>, … y<t>.

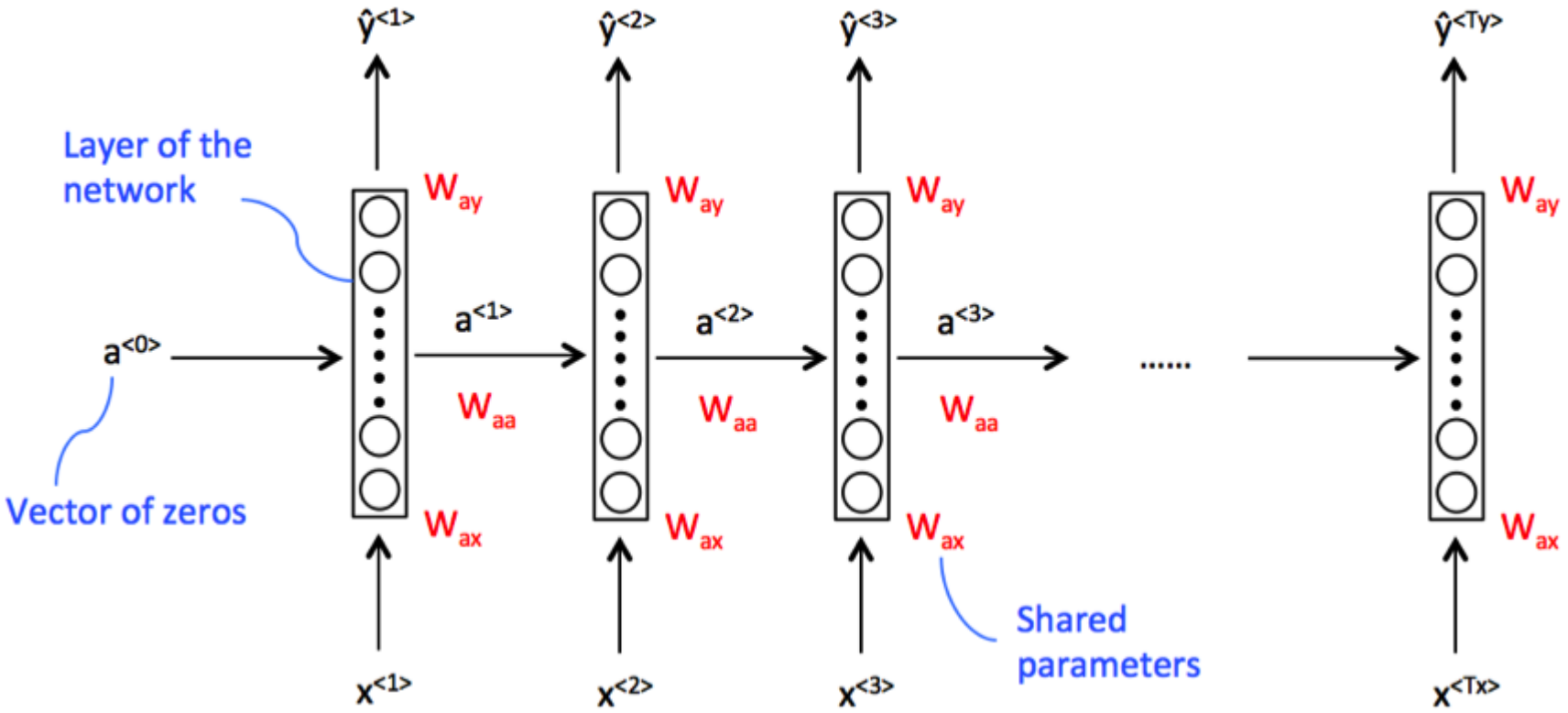
This doesn’t work well. A couple of problems are present:

The inputs and outputs for the different examples can be of different lengths. Each input is a sentence, so it’s fair to imagine that most of the training examples are sentences of different length.

The network doesn’t share features learned from different positions of the text. It doesn’t generalize well.

RNN best for applications dealing with sequence model

The RNN scans through the data from left to right and the parameters used for each time step (Wax) are shared. The horizontal connections are governed by Waa parameters, which are the same for every time step. The Way's are the parameters that govern the output predictions.



RNN best for applications dealing with sequence model

$a^{<0>} = \vec{0}$ (a vector of zeros)

$a^{<t>} = g(W_{aa}a^{<t-1>} + W_{ax}x^{<t>} + b_a)$

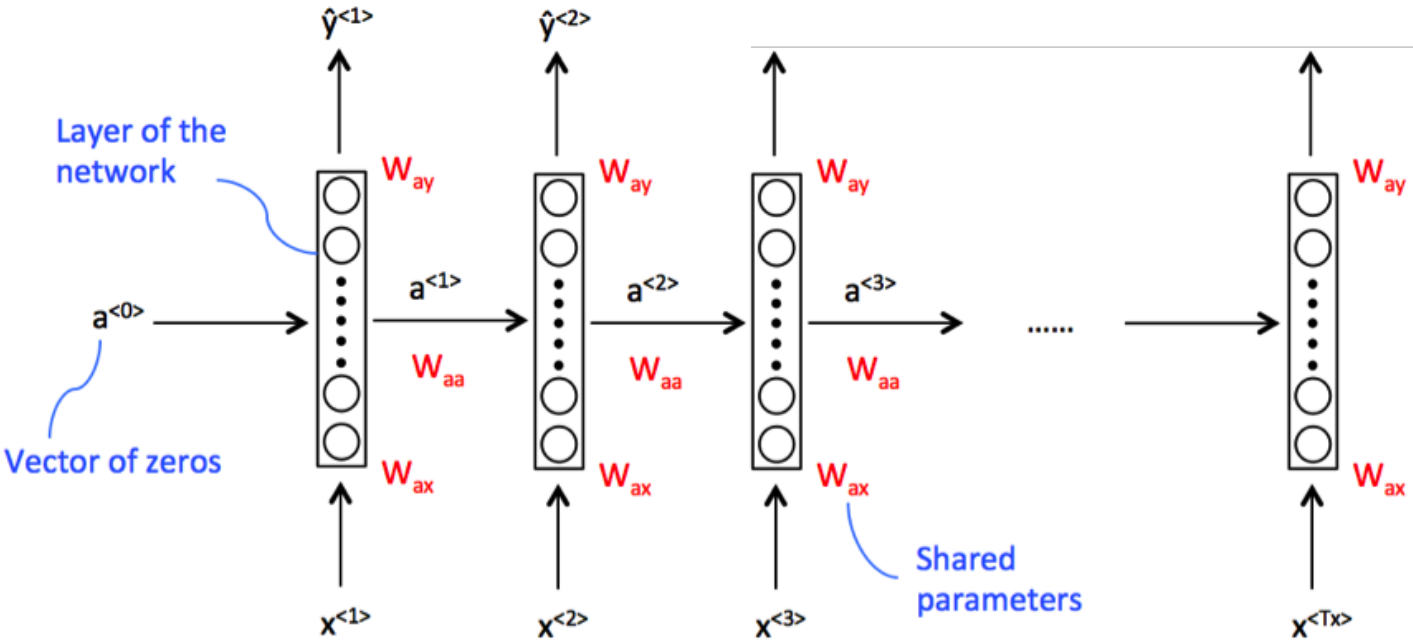
$\hat{y}^{<t>} = g(W_{ya}a^{<t>} + b_y)$

Matrix W_a as: $[W_{aa} \mid W_{ax}]$

$a^{<t>} = g(W_a[a^{<t-1>}, x^{<t>}] + b_a)$

$\hat{y}^{<t>} = g(W_y a^{<t>} + b_y)$

Vectors stacked together as: $\begin{bmatrix} a^{<t-1>} \\ - - - \\ x^{<t>} \end{bmatrix}$

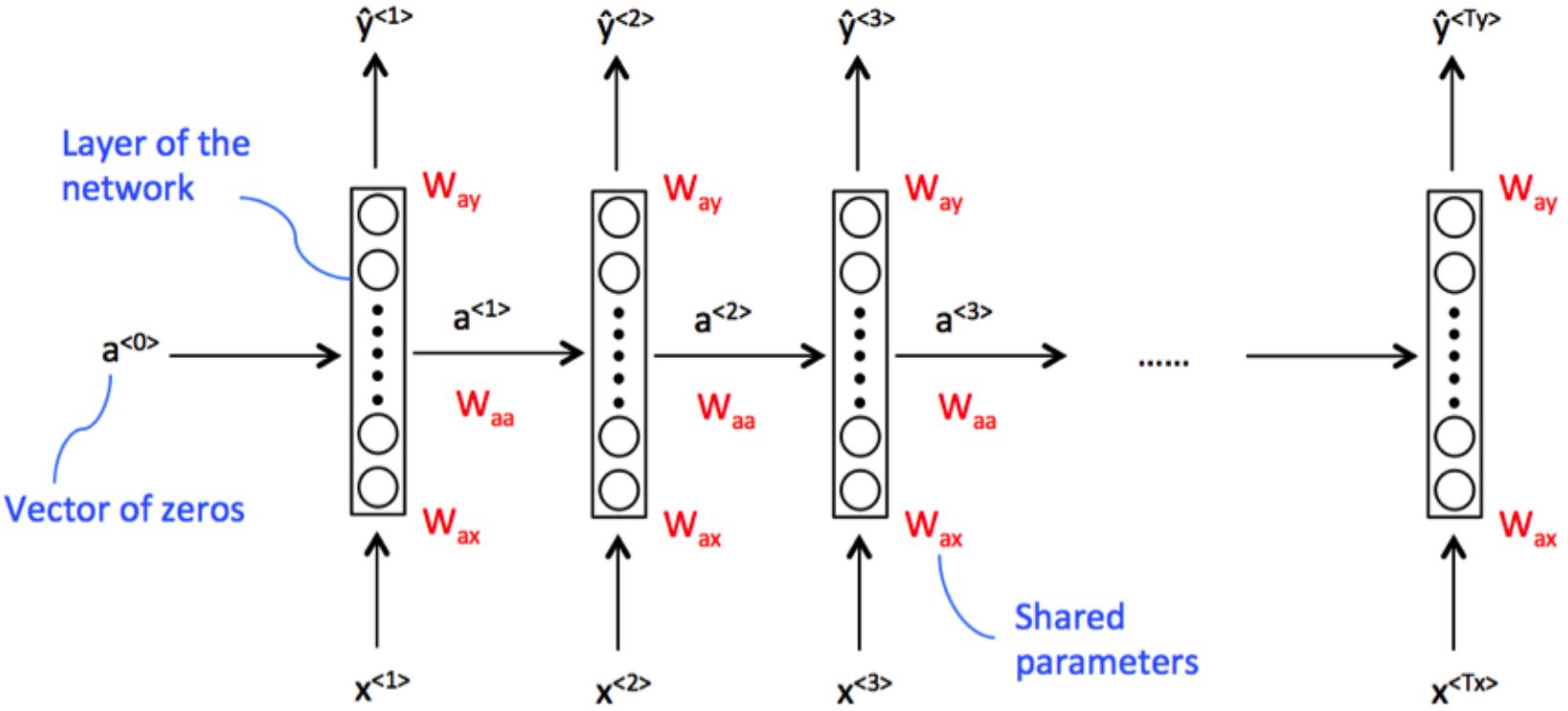


RNN best for applications dealing with sequence model

For each layer of the network, we will calculate the loss, and then sum the losses up to obtain the entire loss for the sequence.

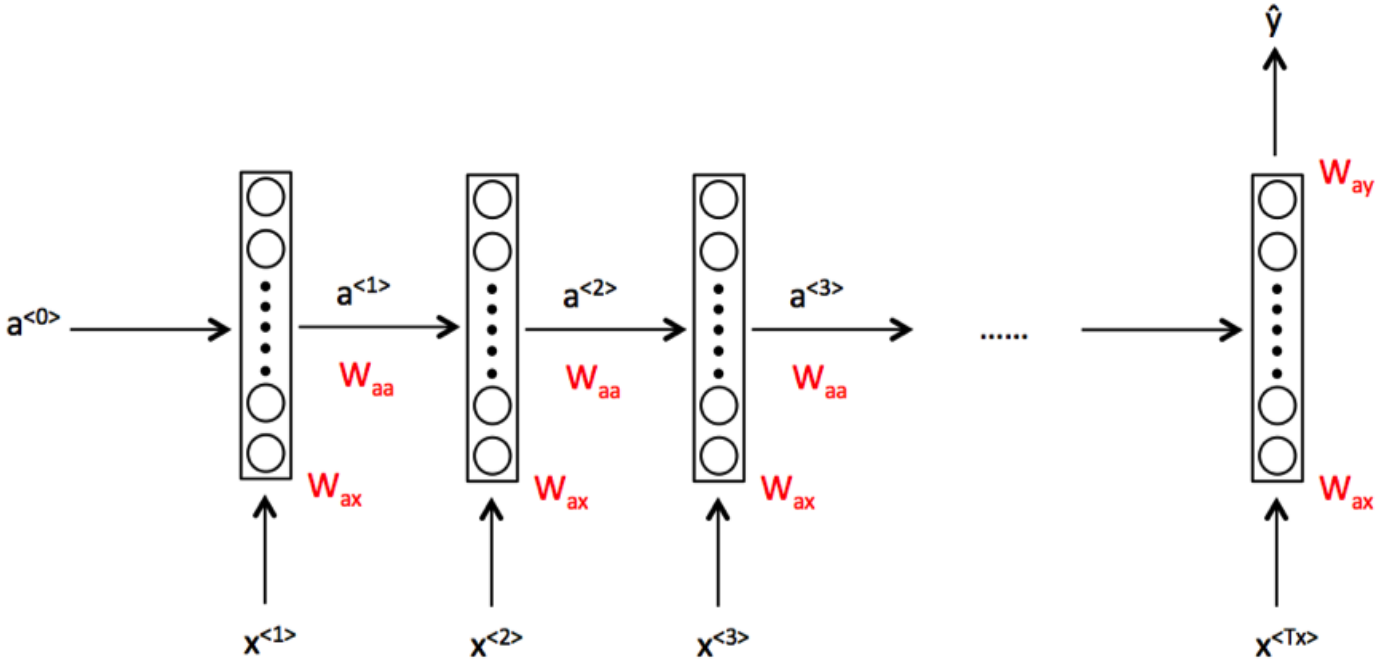
After that, we can compute the backpropagation, which in this network is called “backpropagation through time” as we run back through the sequence.

This is the “many-to-many” architecture where Tx = Ty.



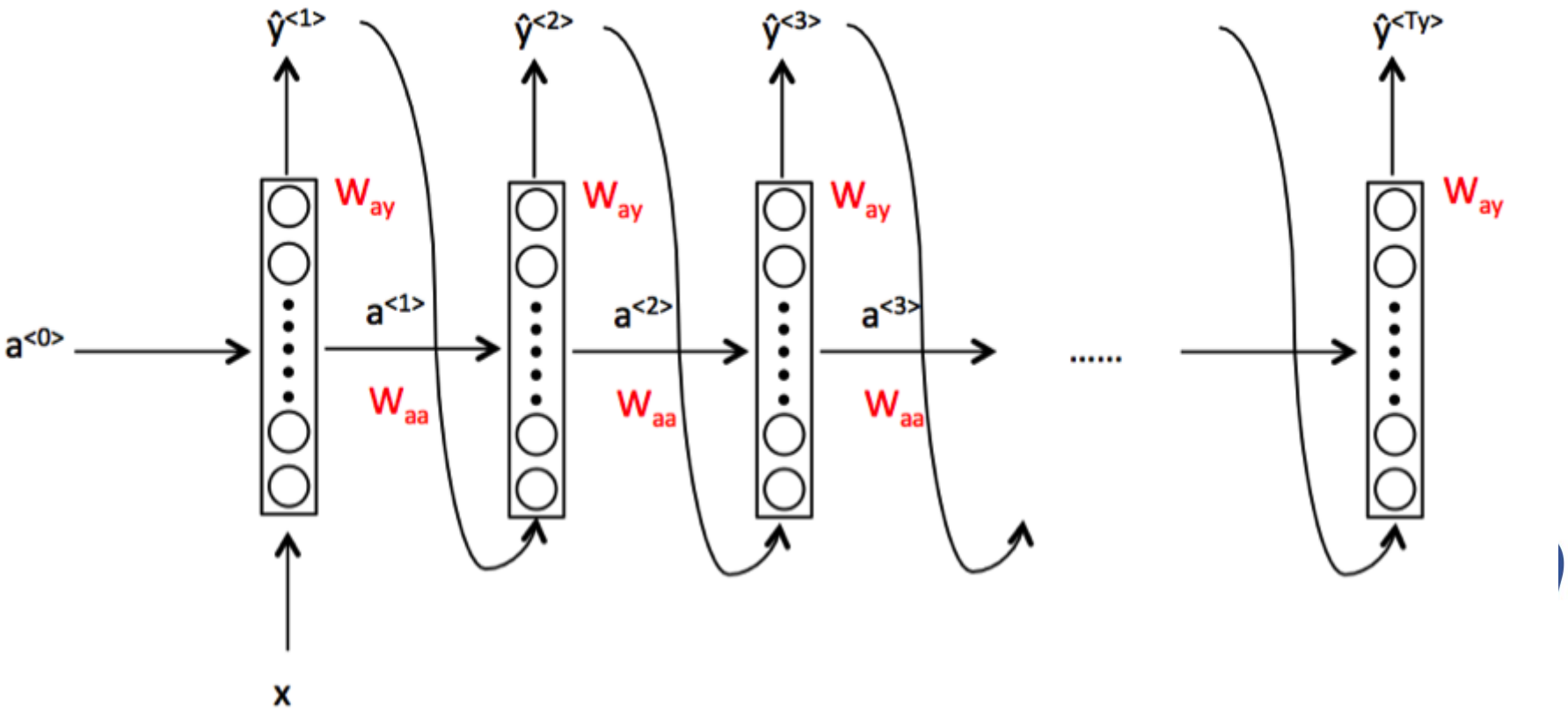
Other RNN Architectures

An application such as sentiment classification we end up in a situation where Ty=1, so our RNN is of the “many-to-one” type, and the architecture is:



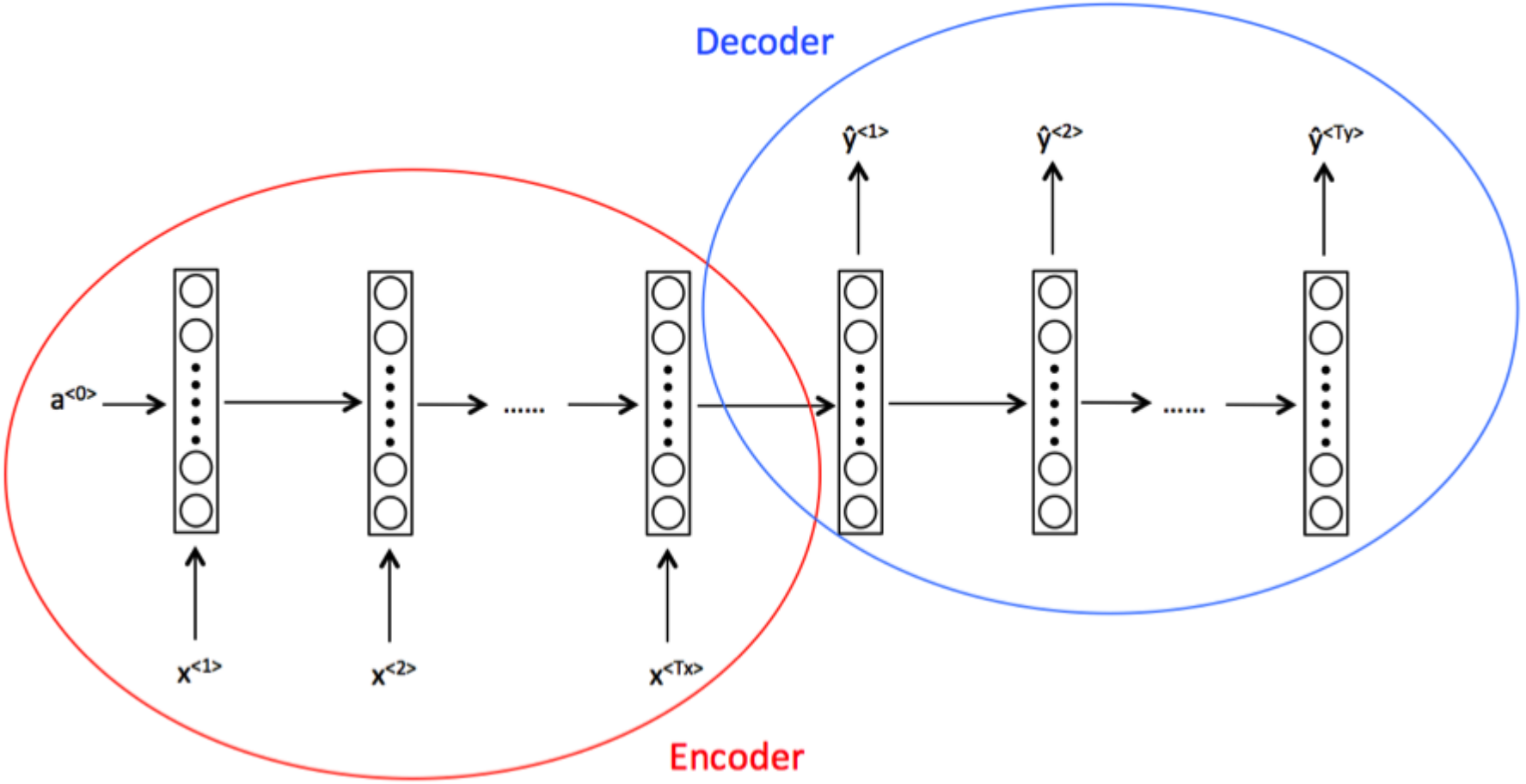
Other RNN Architectures

For an application as music generation, we have a “one-to-many” architecture, as the input can be an integer related to a genre, while the output is a piece of music:



Other RNN Architectures

In machine translation, when the input sequence length is different than the output sequence (Tx ≠ Ty), the architecture of the RNN reflects a “many-to-many” relationship.



RNN Language Modeling

Language models and sequence generation - Language models make predictions by estimating the probability of the next word given the words that precede it.

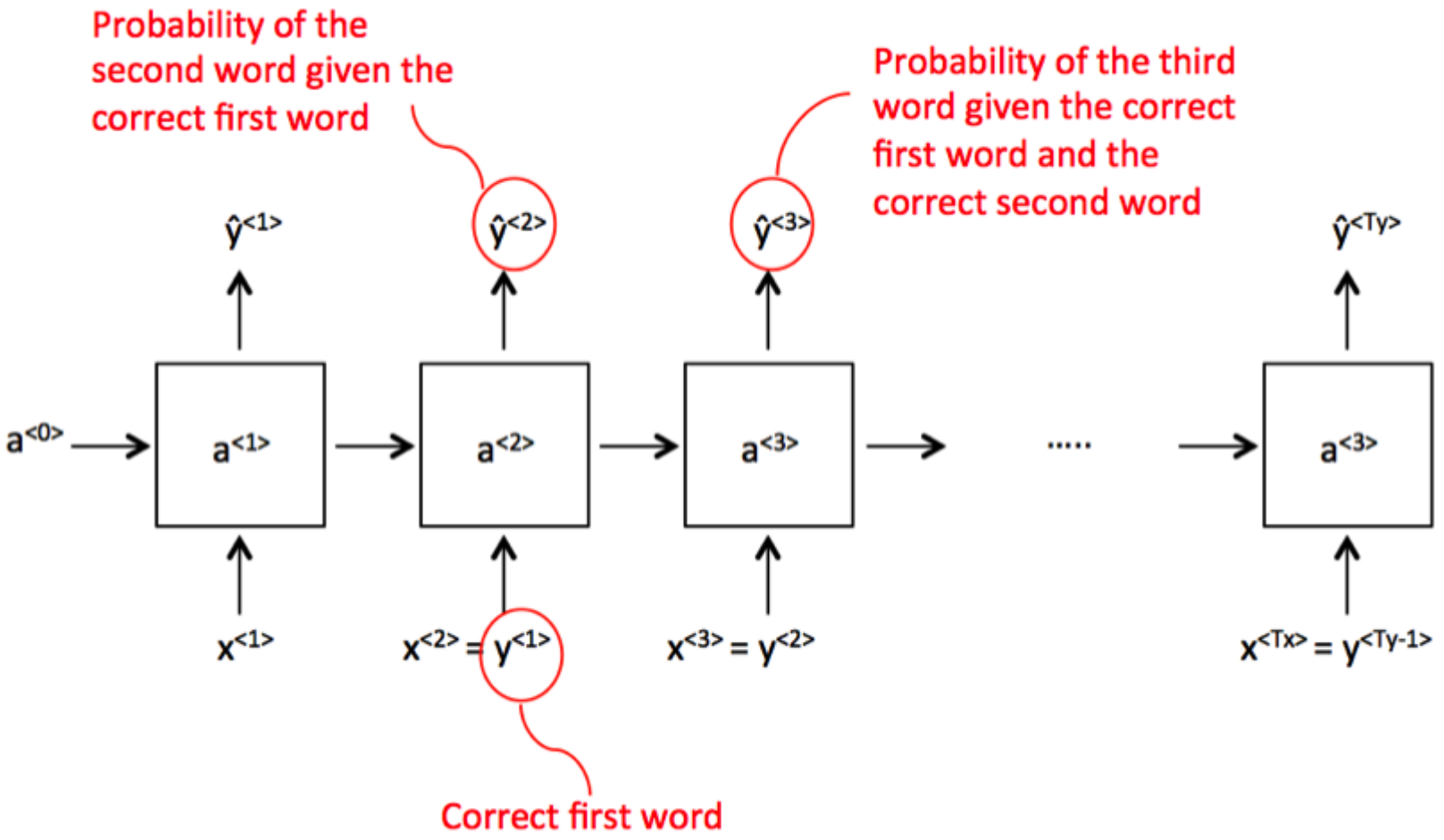
After you've trained a language model, the conditional distributions (what a language model does is to estimate the probability of the particular sequence of words that it will output.) you've estimated may be used to sample novel sequences.

RNN Language Modeling

Steps to build the model:

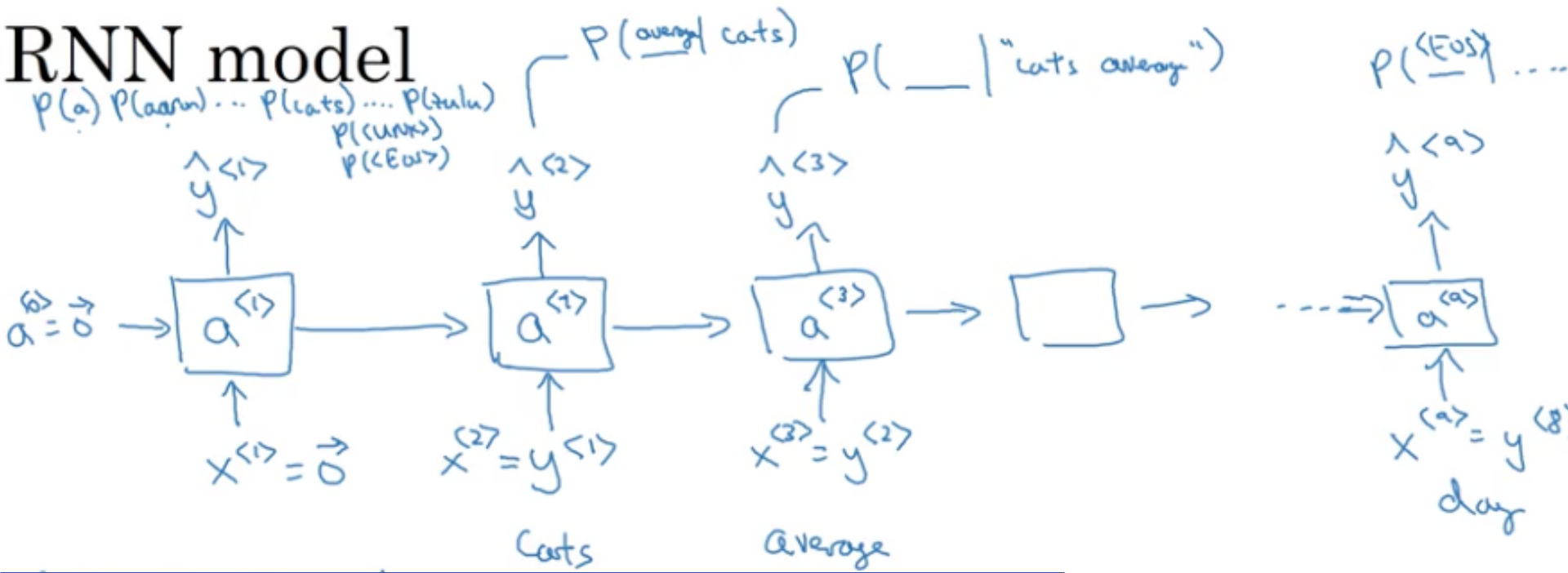
- Take a training set: a large corpus of English (or whichever language suits you) text.
- Tokenize the input sentences. (This is what we did before when we assigned a token (a number) to each word from the vocabulary). Remember, if a word does not exist within the dictionary we can always replace it with the <UNK> token.
- Map each word to a one-hot vector of indices.
- Model when the sentence ends by adding an extra token called <EOS>.
- Build an RNN to model the chance of these different sequences.

RNN Language Modeling

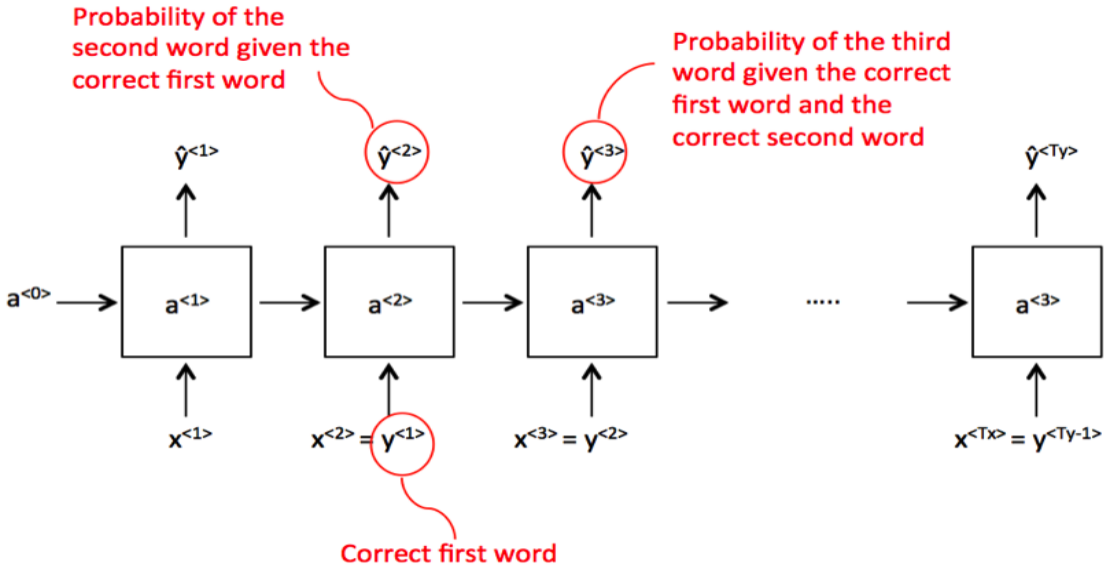


RNN Language Modeling

Language models and sequence generation - Language models make predictions by estimating the probability of the next word given the words that precede it. After you've trained a language model, you'll get the probability of entire sequences



Language Modeling



We see that a<1> makes a softmax prediction to try to figure out what is the probability of the first word ŷ<1>. Then in the second step, a<2> makes a softmax prediction given the correct first word (y<1>). All of the following steps make a prediction based on the correct words that come before them.

After training, it can predict, given an initial set of words, what' s the chance of the next word. So, given a new sentence (y<1>, y<2>, y<3>) it can tell the probability of this sentence:

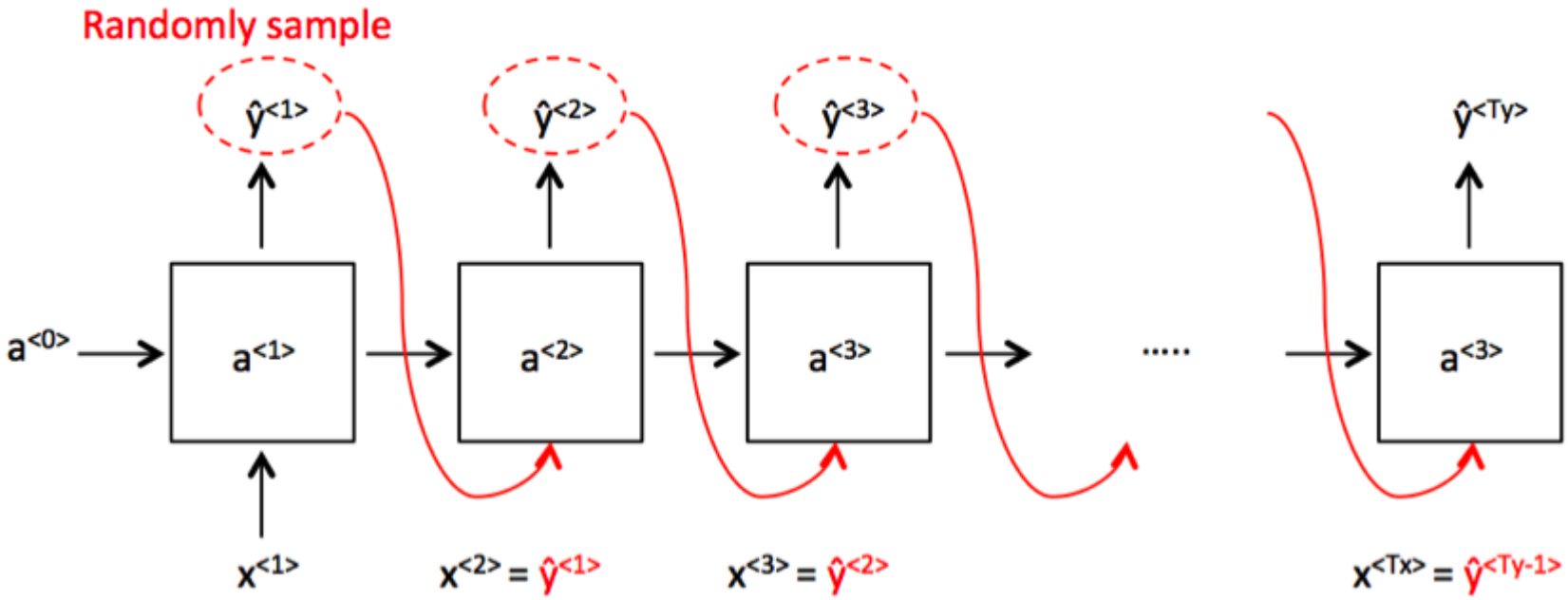
$$P(y^{<1>}, y^{<2>}, y^{<3>}) = P(y^{<1>}) * P(y^{<2>} \mid y^{<1>}) * P(y^{<3>} \mid y^{<1>}, y^{<2>})$$

Now that we have trained our RNN we can use it to generate novel sequences.

Language Modeling

A trained model can then generate any random sequence of words! Essentially, the input of each step, instead of being the y from the previous step will be a random sample from the previous step distribution.

So far we have built RNN on a word level, meaning the vocabulary is composed of words. We can also build a character level RNN, where the vocabulary is comprised of the individual character of the alphabet.



Vanishing gradients with RNNs

Throughout the previous examples, you might have noticed that the output ŷ was mainly influenced by the values in the sequence close to it. On the other hand, there are situations when some sentences have long dependencies, meaning some words within the sentence are related to other ones much earlier in the sequence.

Think about a sentence where you have a subject, followed by many words, and then finally we have the verb, which is depending on the earlier subject.

Basic RNNs are not good at capturing these long-term dependencies.

It's like what we have seen in a deep neural network, where the network has a difficult time propagating back to affect the weights of earlier layers.

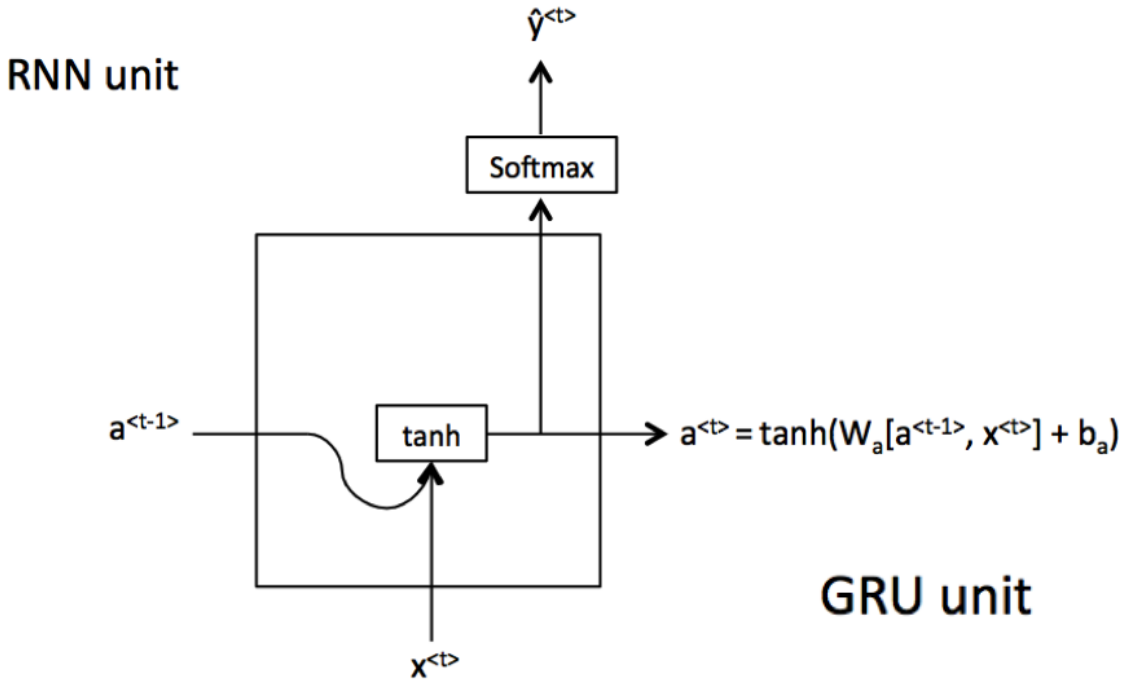
Exploding gradients in an RNN are rare, but when they happen, they can be catastrophic, as parameters get very large. So, in a way, it's kind of easy to spot when this is happening and fixing the situation.

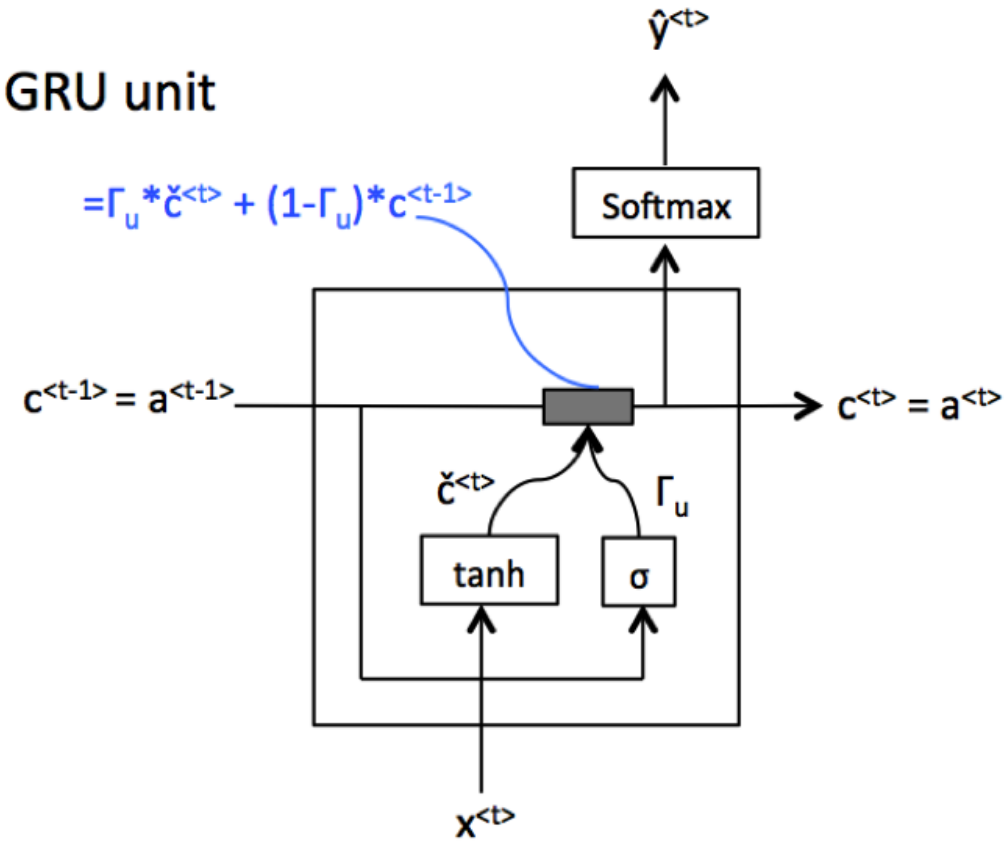
The solution is to apply "gradient clipping".

Therefore, take a look at the gradient vectors, and if it's getting bigger than a set threshold, you can rescale the gradients.

Let's focus on the most difficult problem: vanishing gradients. When the network "forgets" what happened earlier and does not propagate dependencies across the whole sentence.

Gated Recurrent Unit
GRU





GRU

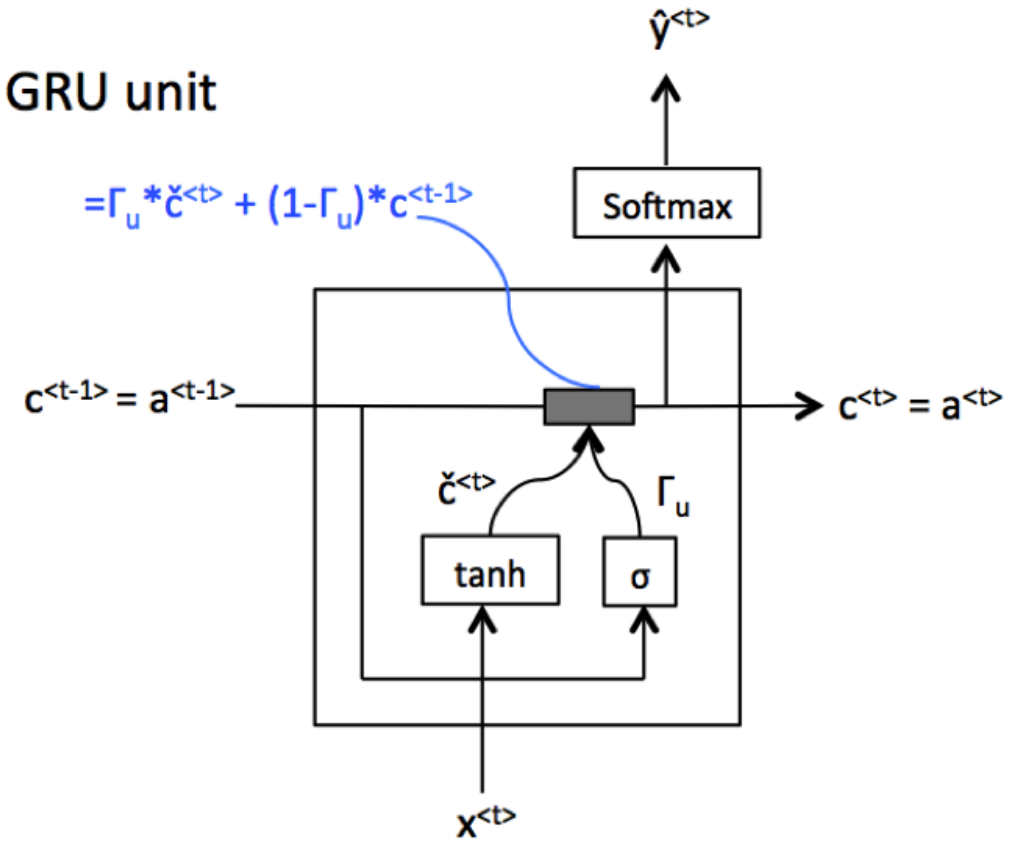
The GRUn unit has a new variable called c, which is a ”memory cell” that provides a bit of memory to remember words even further along the sentence.

At every step, this memory cell c is overwritten by č, computed using the activation function tanh of Wc.

The purpose of č is to replace c through the use c a gate Γu, which takes a value between 0 anc 1, and it basically decides whether or not we update c with č.

When Γu = 0 then c<t> = c<t-1> . Therefore the value of c<t> is maintained across many tim steps. This allows overcoming the problem c vanishing gradients.

On the other hand, when Γu = 1 then č <t> = c<t>.



Gated Recurrent Unit (GRU)

When Γu = 0 then c<t> = c<t-1> . Therefore the value of c<t> is maintained across many time steps. This allows overcoming the problem of vanishing gradients.

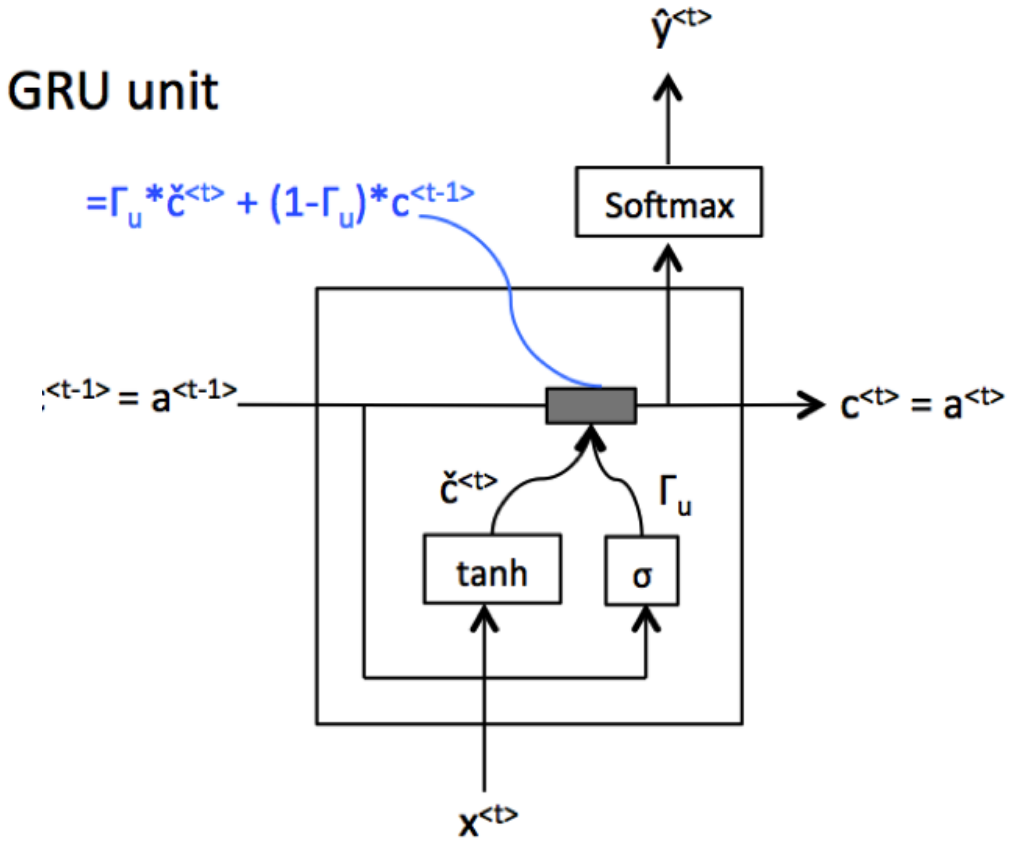
On the other hand, when Γu = 1 then č <t> = c<t>.

The equations that govern this unit are:

$$\check{c}^{<t>} = \tanh\left(W_c[c^{<t-1>}, x^{<t>}] + b_c\right)$$

$$\Gamma_u = \sigma(W_u[c^{<t-1>}, x^{<t>}] + b_u)$$

$$c^{<t>} = \Gamma_u * \check{c}^{<t>} + (1 - \Gamma_u) * c^{<t-1>}$$



Long Short Term Memory (LSTM)

LSTM is another, even more powerful, unit to learn very long-range connections in a sequence. The unit consists of three gates: the "forget", "update" and "output" gate.

The forget gate plays the role of (1- Γu) that we saw in the GRU unit.

Additionally, this time, c<t> is different than a<t>.
These are the equations that govern the unit, followed by the visual architecture:

$$\check{c}^{<t>} = \tanh\left(W_c[a^{<t-1>}, x^{<t>}] + b_c\right)$$

$$\text{update gate: } \Gamma_u = \sigma(W_u[a^{<t-1>}, x^{<t>}] + b_u)$$

$$\text{forget gate: } \Gamma_f = \sigma(W_f[a^{<t-1>}, x^{<t>}] + b_f)$$

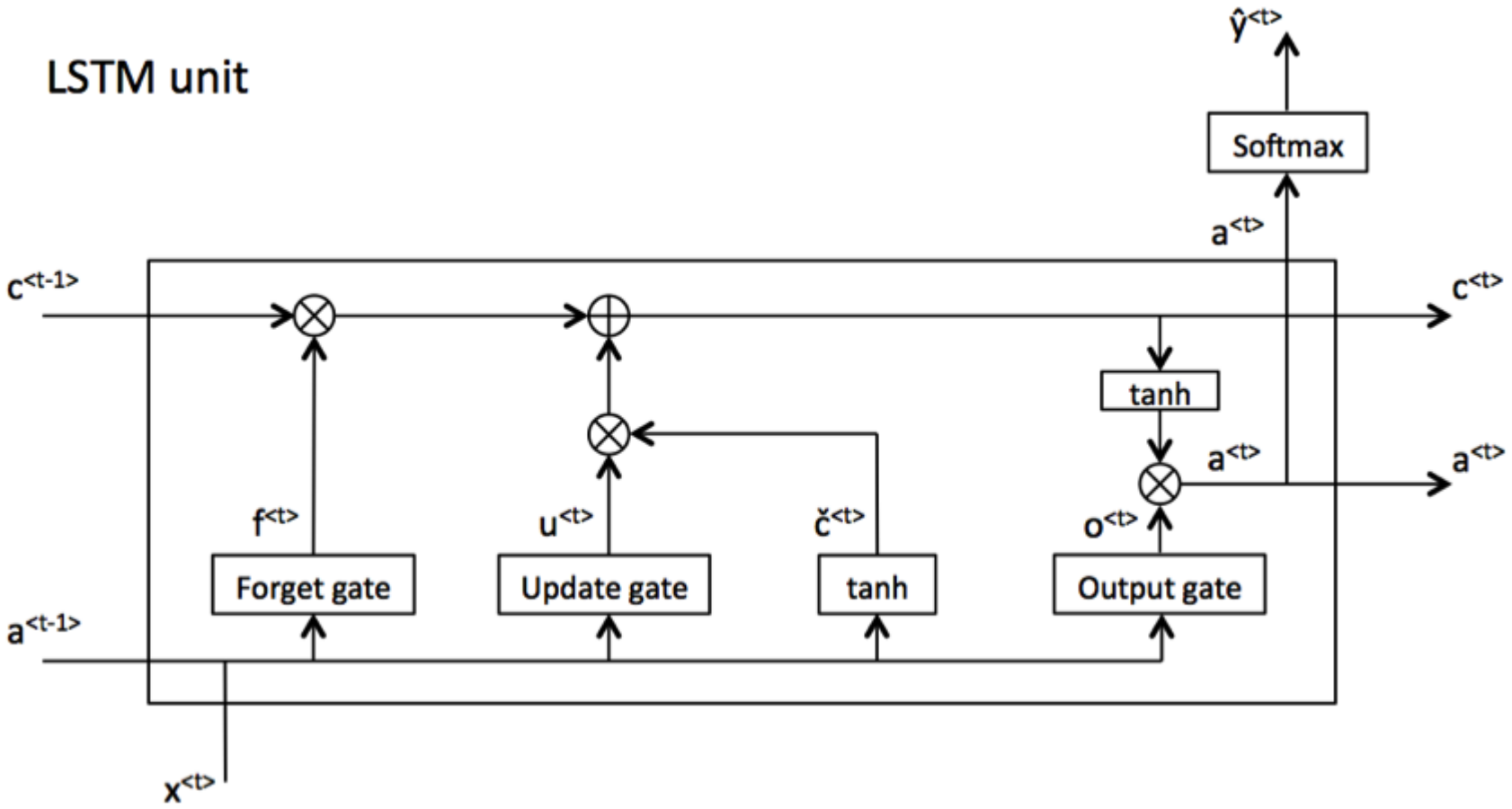
$$\text{output gate: } \Gamma_o = \sigma(W_o[a^{<t-1>}, x^{<t>}] + b_o)$$

$$c^{<t>} = \Gamma_u * \check{c}^{<t>} + \Gamma_f * c^{<t-1>}$$

$$a^{<t>} = \Gamma_o * \tanh\left(c^{<t>}\right)$$

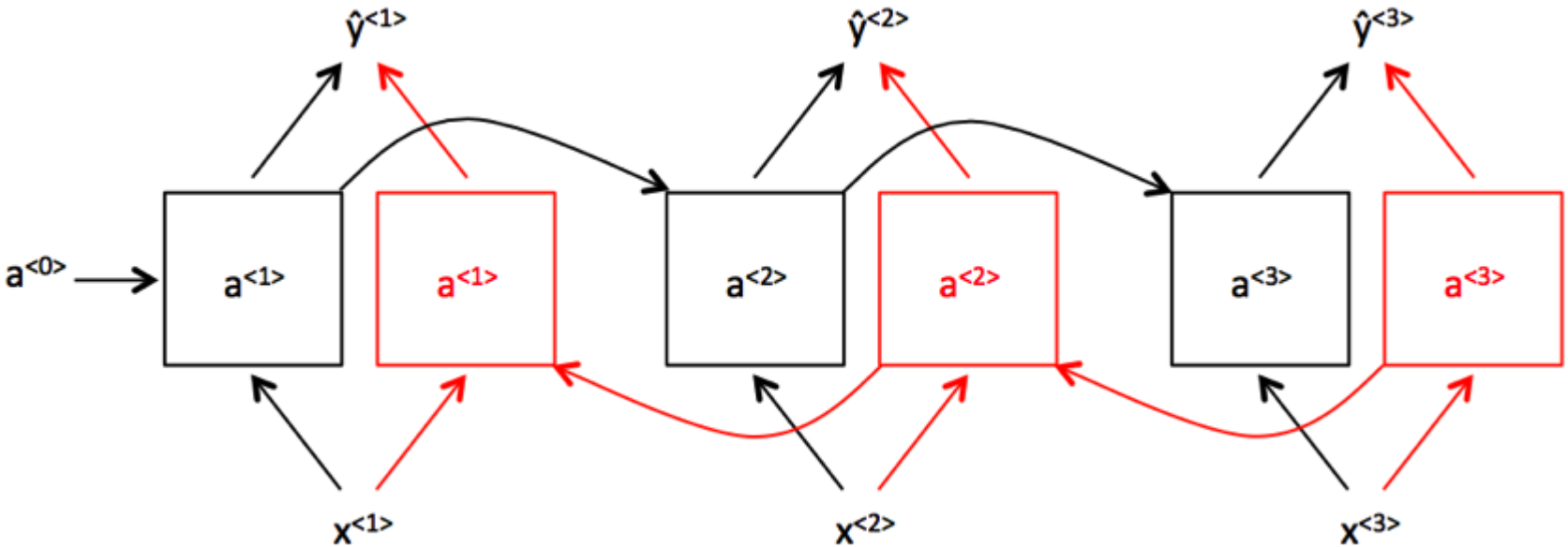
LSTM

An LSTM unit is more powerful and flexible than a GRU unit, and it's the more proven choice. GRU units, instead, are more recent and more straightforward, so it's easier to build bigger models with them.



Bidirectional LSTM (BLSTM)

These networks take not only information from earlier in the sequence, but also from later on. They are basically like a patient listener. They “listen” to the whole sentence before making a prediction, which is nice, generally speaking. Not a lot of humans can do that!



Case Study 2 – CNN for Classification of Malware Disassembly Files

– David Gibert

The extracted features from the disassembled files are usually n-grams.

An n-gram is a contiguous sequence of n items from a sentence.

In this case, those items are opcodes extracted from the disassembled files.

Instead, represent opcodes as word embeddings that captures the relationship among the words (here, opcodes).

Language models here learn vector representation of words, with the hypothesis that words that occur in the same contexts tend to share semantic meaning.

Skip-Gram model tries to predict each context word from its target word.

input to the model is w_i and the output is $w_{i-2}, w_{i-1}, w_{i+1}, w_{i+2}$

Case Study 2 – CNN for Classification of Malware Disassembly Files

– David Gibert

- First, build a vocabulary of words from the malware training samples (665 opcodes)
- Represent each word (opcode like *push*) as a one-hot vector.
- The output layer depends on the window size, w (neighbourhood size)
- If w=1 (one word each on left & right of target word), then network output is a 2D vector – probs. Of words on the left and probs of words on the right
- Hidden layer dimension = VxE (vocabulary size x embeding size)
- Opcodes whose vector representations are most similar
- to the opcode "push" are:

  1. pop
  2. insertps
  3. fucomp

  which makes sense because tons of push instructions in malware files are followed by the pop instruction or viceversa and are the two opcodes most used.
- Use CNN on the word embeddings to classify the malware

Case 3 - IoT Malware Hunting Using RNN – HamidHaddad et al

Internet of Things (IoT) devices are increasingly deployed in different industries and for different purposes (e.g. sensing/collecting of environmental data in both civilian and military settings).

They are a valuable attack target, such as malware designed to compromise specific IoT devices.

In this work Recurrent Neural Network (RNN) was used in detecting IoT malware. Specifically, theirvapproach uses RNN to analyze ARM-based IoT applications' execution operation

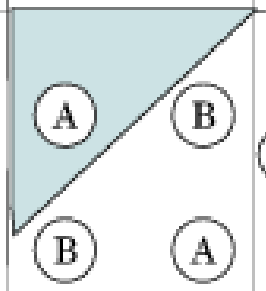
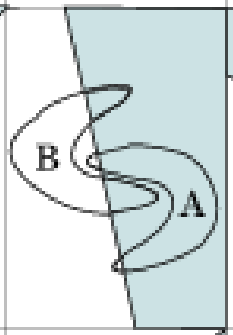
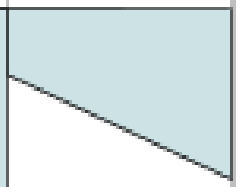
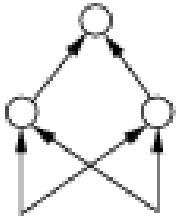
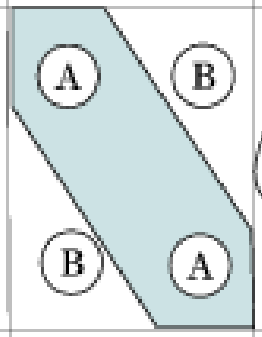
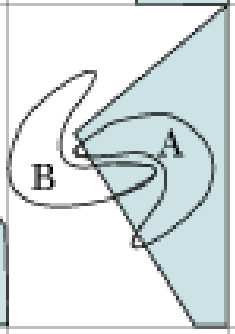
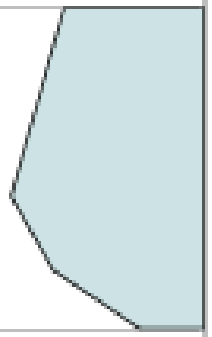
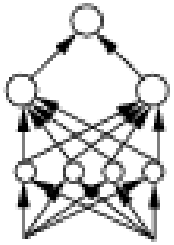
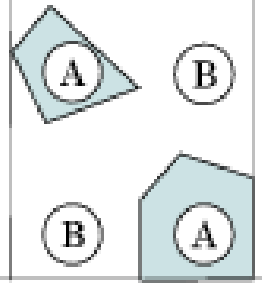
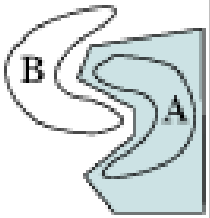
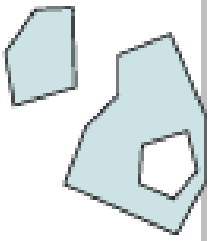
codes (OpCodes).

IoT application dataset comprising 281 malware and 270 benign ware, evaluated on 100 new IoT malware samples (i.e. not previously exposed to the model) with three different Long Short Term Memory (LSTM) configurations.

Obtained an accuracy of 98.18%

Thank you

Geometric Interpretation of Perceptron Learning

Structure	Types of decision regions	Exclusive OR problem	Classes with meshed regions	Most general region shapes	
Single-layer 	Half Plane (Bounded by hyperplane)				(a)
Two-layer 	Convex (Open or closed regions)				(b)
Three-layer 	Arbitrary (Complexity limited by number of neurons)				(c)

Training

- Generalization: network's performance on a set of test patterns it has never seen before (lower than on training set)
- Training set used to let ANN capture features in data or mapping
- Initial large drop in error is due to learning, but subsequent slow reduction is due to:
    1. Network memorization (too many training cycles used)
    2. Overfitting (too many hidden nodes)

  (network learns individual training examples and loses generalization ability)