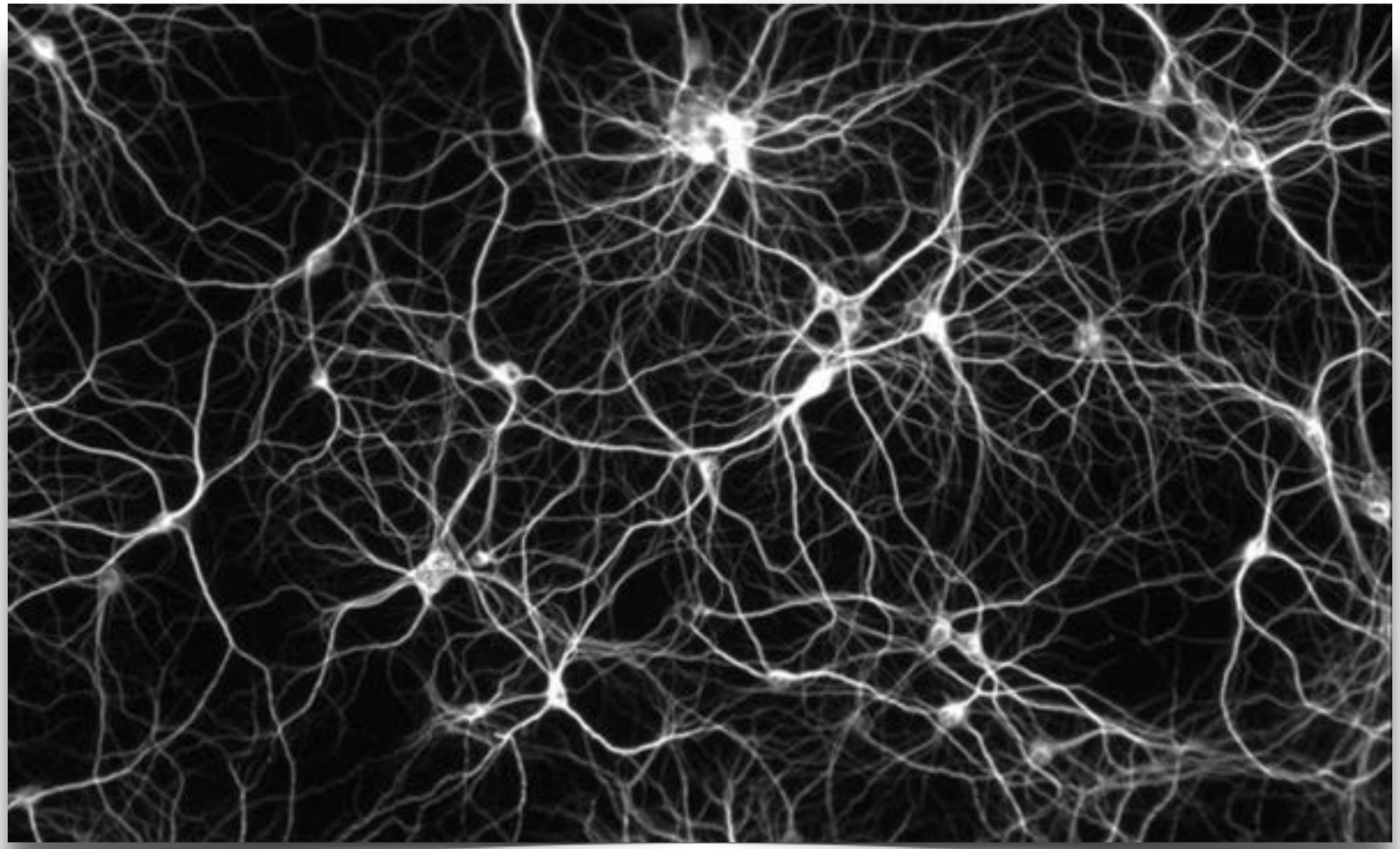
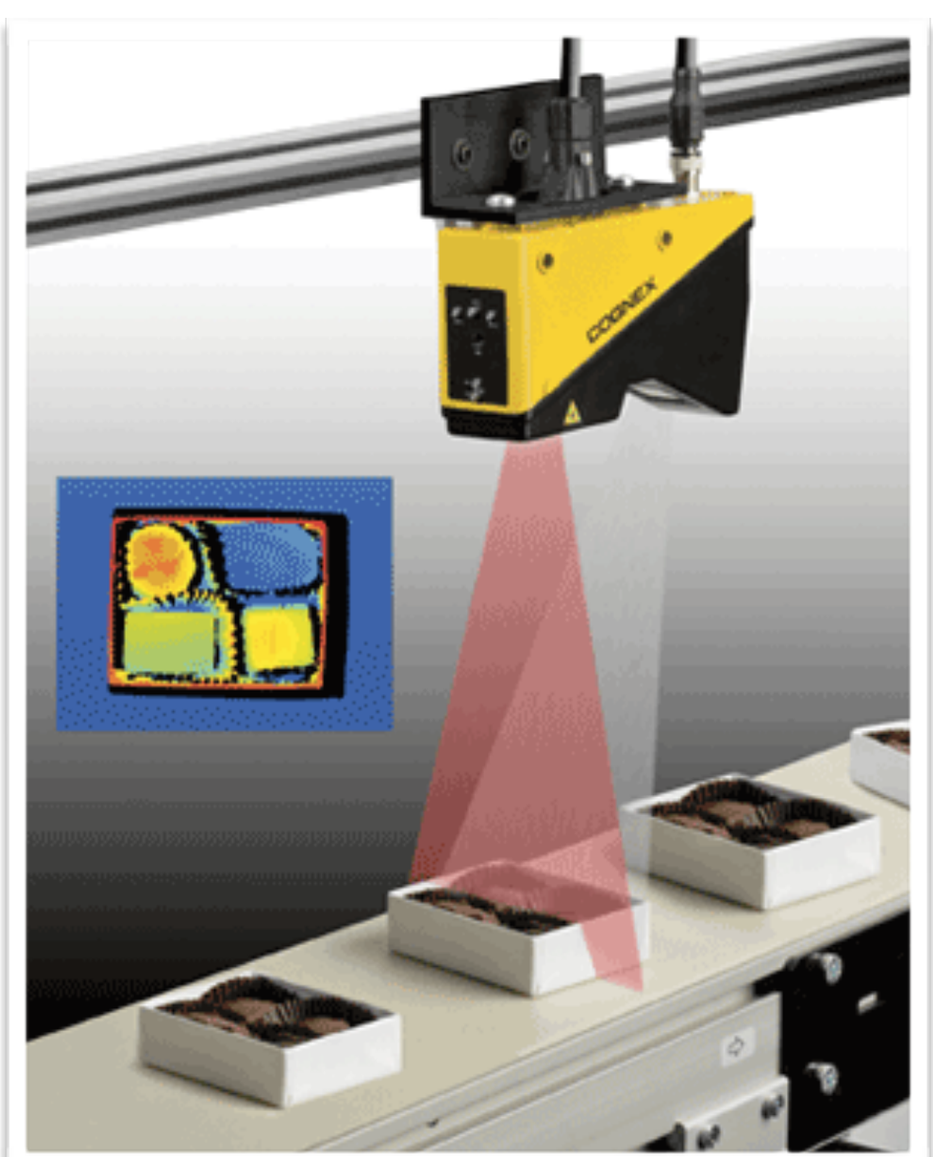


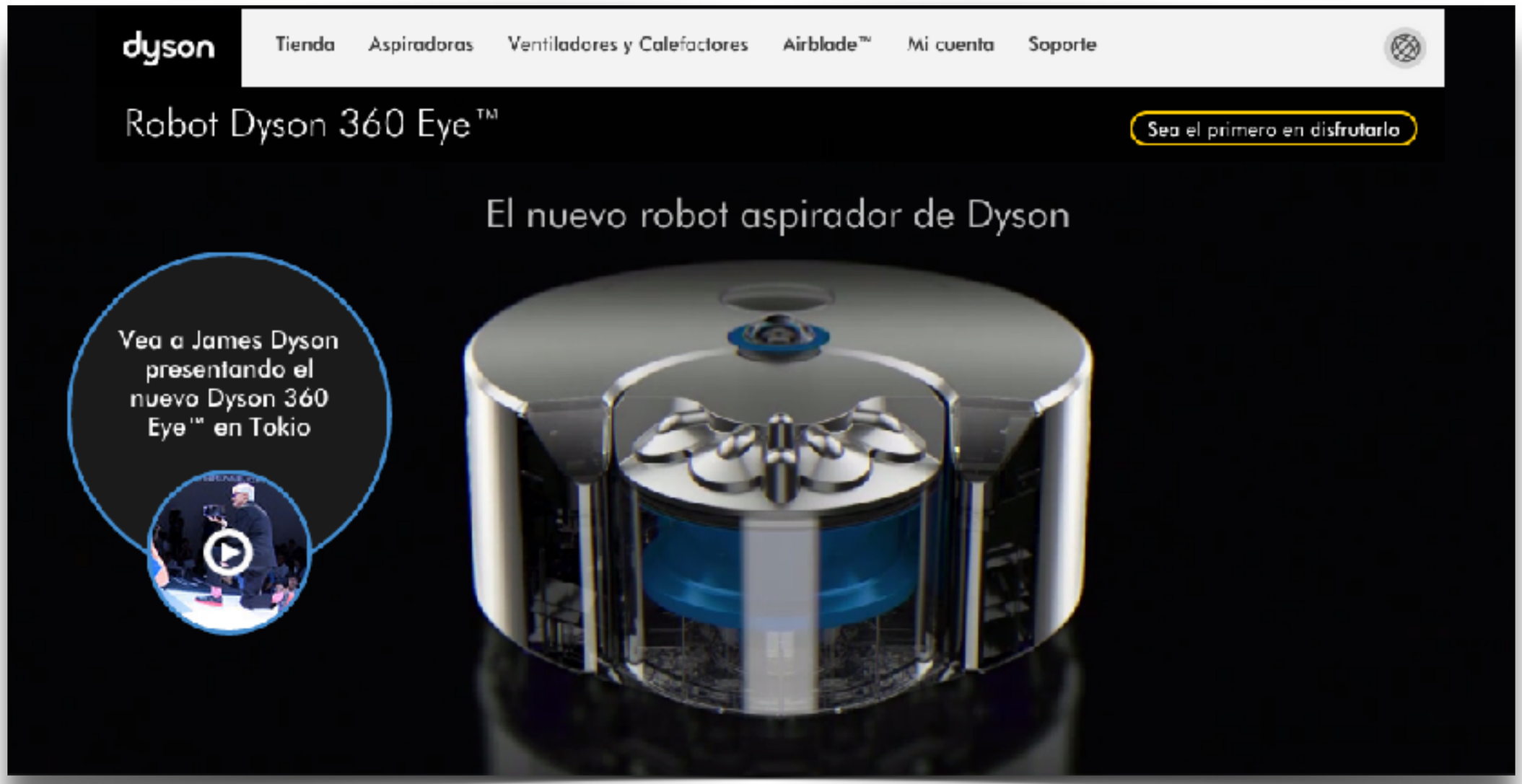
Introduction to **deep learning**

Jordi Vitrià
Universitat de Barcelona

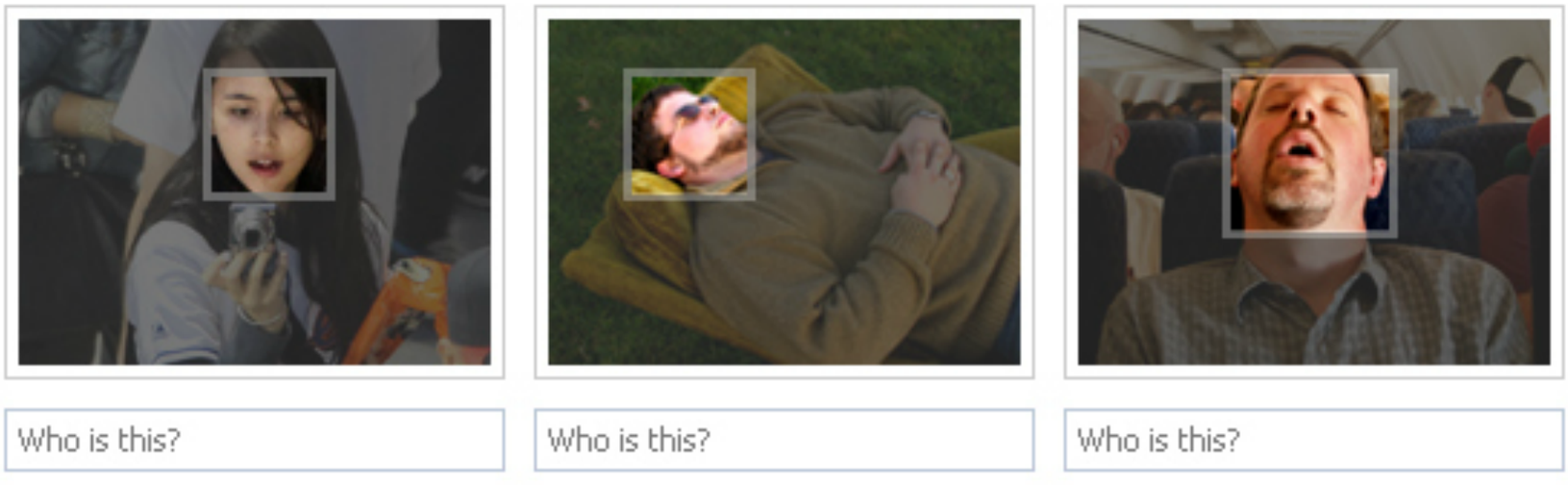
“Classical” applications of computer vision:
object classification, detection and segmentation… in controlled environments.



From 2012 deep learning has paved the way for new applications: navigation and mapping.



New applications: face recognition.



DeepFace (Facebook): Accuracy of 97.35%

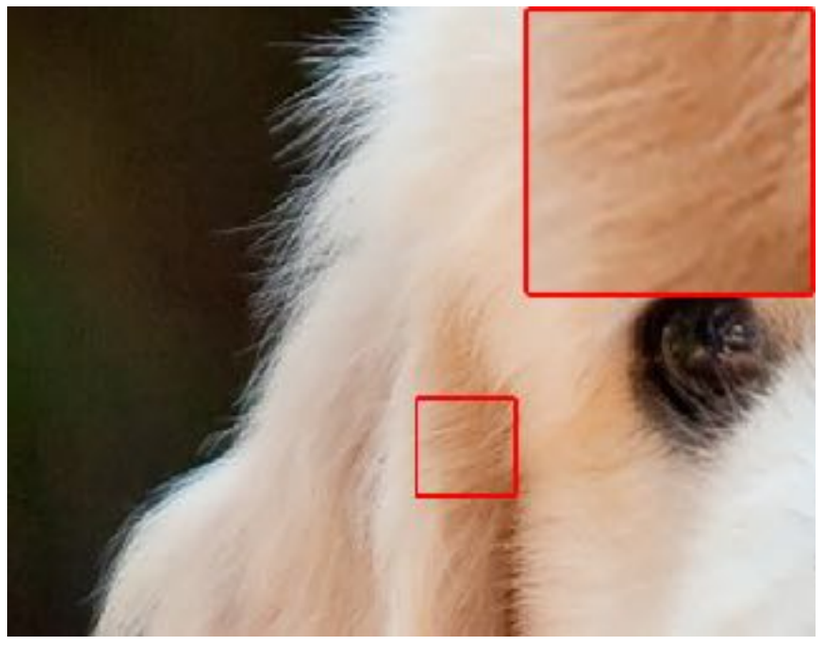
New applications: Image Upscaling (Flipboard)

http://engineering.flipboard.com/2015/05/scaling-convnets/

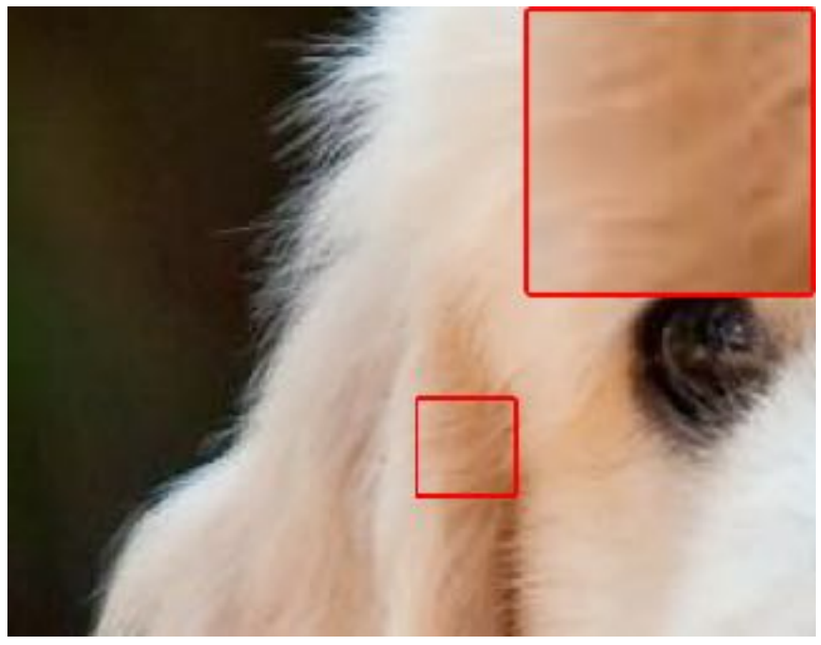
New applications: Image Upscaling (Flipboard)

http://engineering.flipboard.com/2015/05/scaling-convnets/

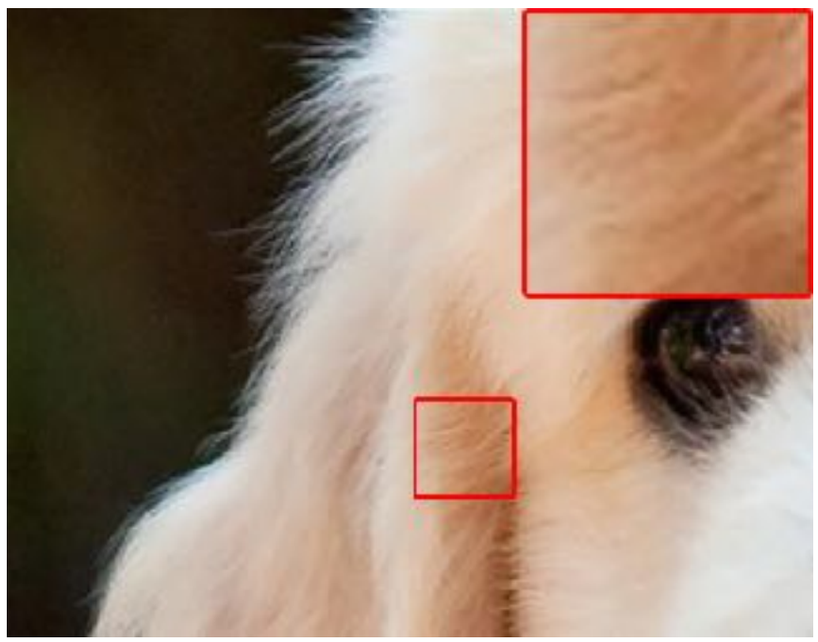
New applications: Image Upscaling (Flipboard)



Original



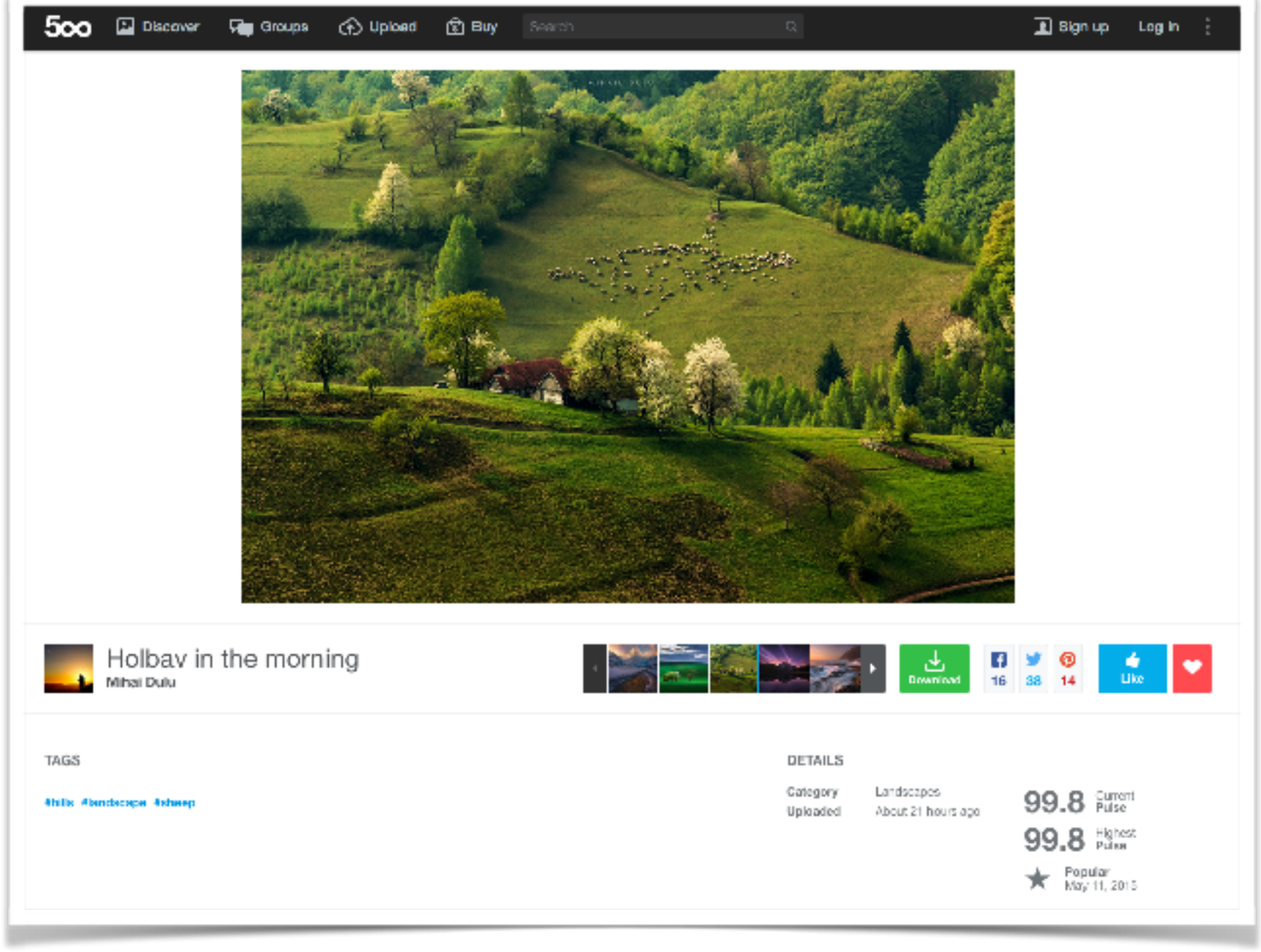
Bicubic



Model

http://engineering.flipboard.com/2015/05/scaling-convnets/

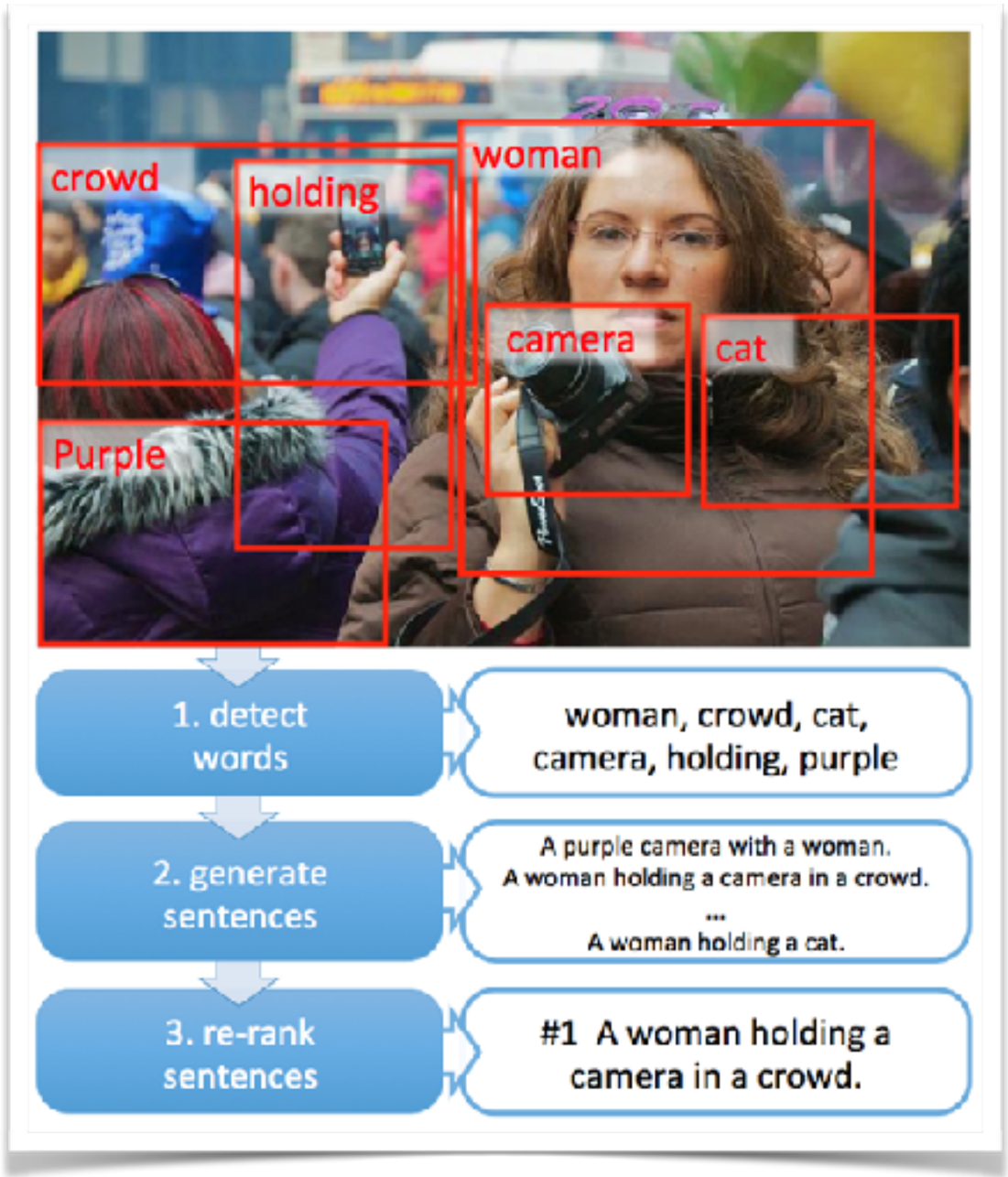
New applications: Non visual data prediction



What is Pulse?

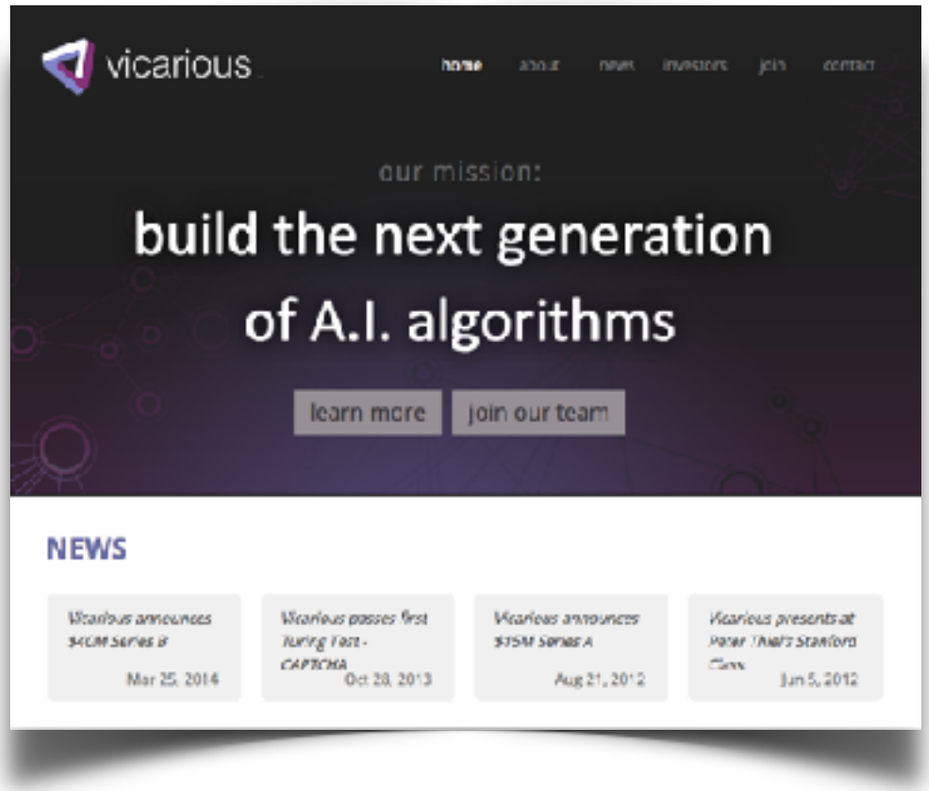
Pulse is a score out of 100 points that measures how **popular** a photo is. Pulse is calculated by an algorithm, which is unique to 500px and is based on votes (Likes & Favorites) on your photo from the community. The Pulse algorithm was designed to promote daily exposure of new photographs and photographers. It is not necessarily a measure of photograph's quality.

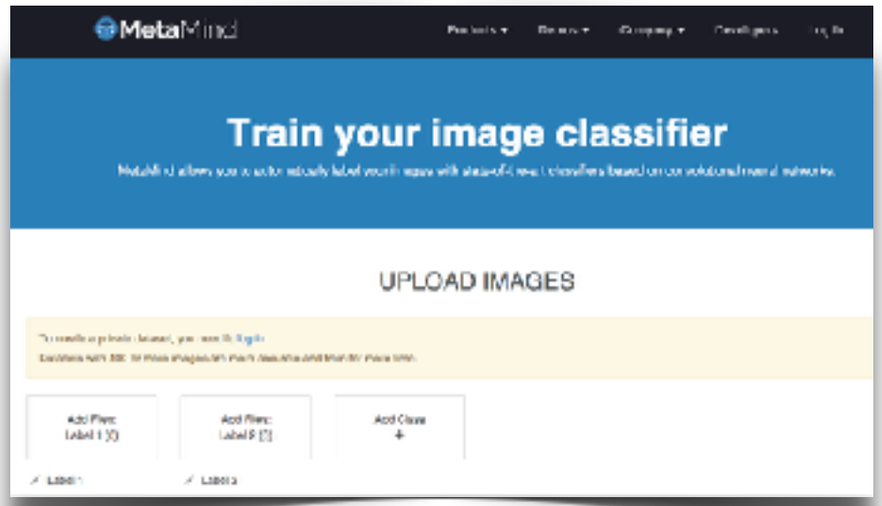
New applications: Automatic Image Captioning

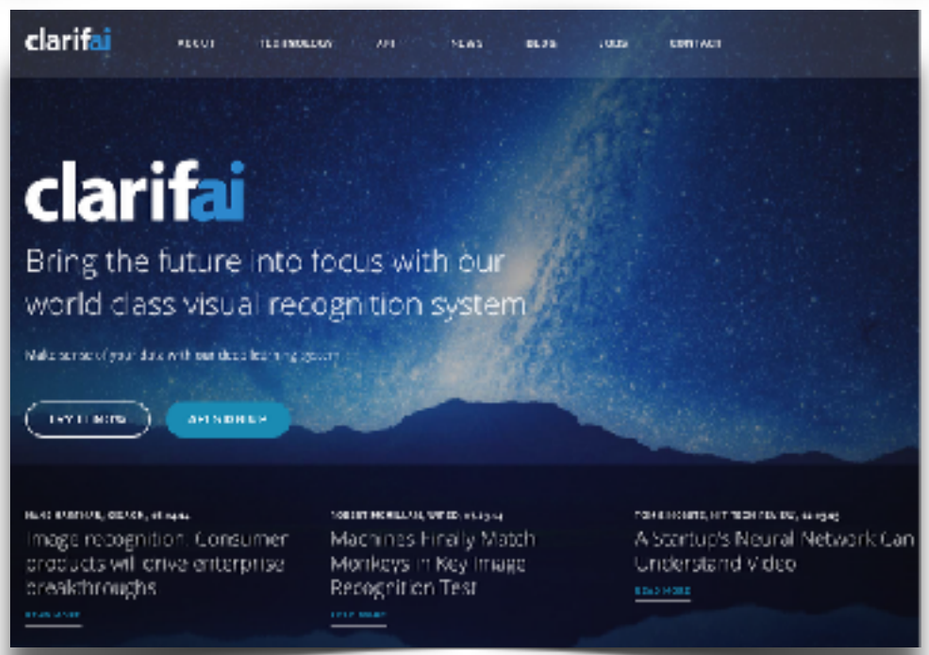


http://blogs.technet.com/b/machinelearning/archive/2014/11/18/rapid-progress-in-automatic-image-captioning.aspx

Deep learning hype/bubble?







Why Deep Learning?


It’s funny!
It’s not rocket science!
It’s powerful!

- In 1943, neurophysiologist **Warren McCulloch** and mathematician **Walter Pitts** wrote a paper on how neurons might work. In order to describe how neurons in the brain might work, they modeled a simple neural network using electrical circuits.

- In 1949, Donald **Hebb** wrote *The Organization of Behavior*, a work which pointed out the fact that neural pathways are strengthened each time they are used, a concept fundamentally essential to the ways in which humans learn. If two nerves fire at the same time, he argued, the connection between them is enhanced.

- In 1957 **Frank Rosenblatt** attempted to build a kind of mechanical brain called the Perceptron, which was billed as "a machine which senses, recognizes, remembers, and responds like the human mind".


- In 1962, **Widrow & Hoff** developed a learning procedure that examines the value before the weight adjusts it (i.e. 0 or 1) according to the rule: Weight Change = (Pre-Weight line value) * (Error / (Number of Inputs)). It is based on the idea that while one active perceptron may have a big error, one can adjust the weight values to distribute it across the network, or at least to adjacent perceptrons.


- A critical book written in 1969 by **Marvin Minsky** and his collaborator **Seymour Papert** showed that Rosenblatt's original system was painfully limited, literally blind to some simple logical functions like "exclusive-or" (As in, you can have the cake or the pie, but not both). What had become known as the field of "neural networks" all but disappeared.

First neural network winter is coming

- In 1982, interest in the field was renewed. **John Hopfield** of Caltech presented a paper to the National Academy of Sciences. His approach was to create more useful machines by using bidirectional lines. Previously, the connections between neurons was only one way.

- In 1986, the problem was how to extend the Widrow-Hoff rule to multiple layers. Three independent groups of researchers, which included **David E. Rumelhart, Geoffrey E. Hinton** and **Ronald J. Williams**, came up with similar ideas which are now called back-propagation networks because it distributes pattern recognition errors throughout the network.

- From 1986 to mid 90's new developments arised: convolutional neural networks (**Y.LeCun**), unsupervised learning (**Y.Bengio**), RBM (**G.Hinton**), etc. But, by this point **new machine learning methods** had begun to also emerge, and people were again beginning to be skeptical of neural nets since they seemed so intuition-based and since computers were still barely able to meet their computational needs.

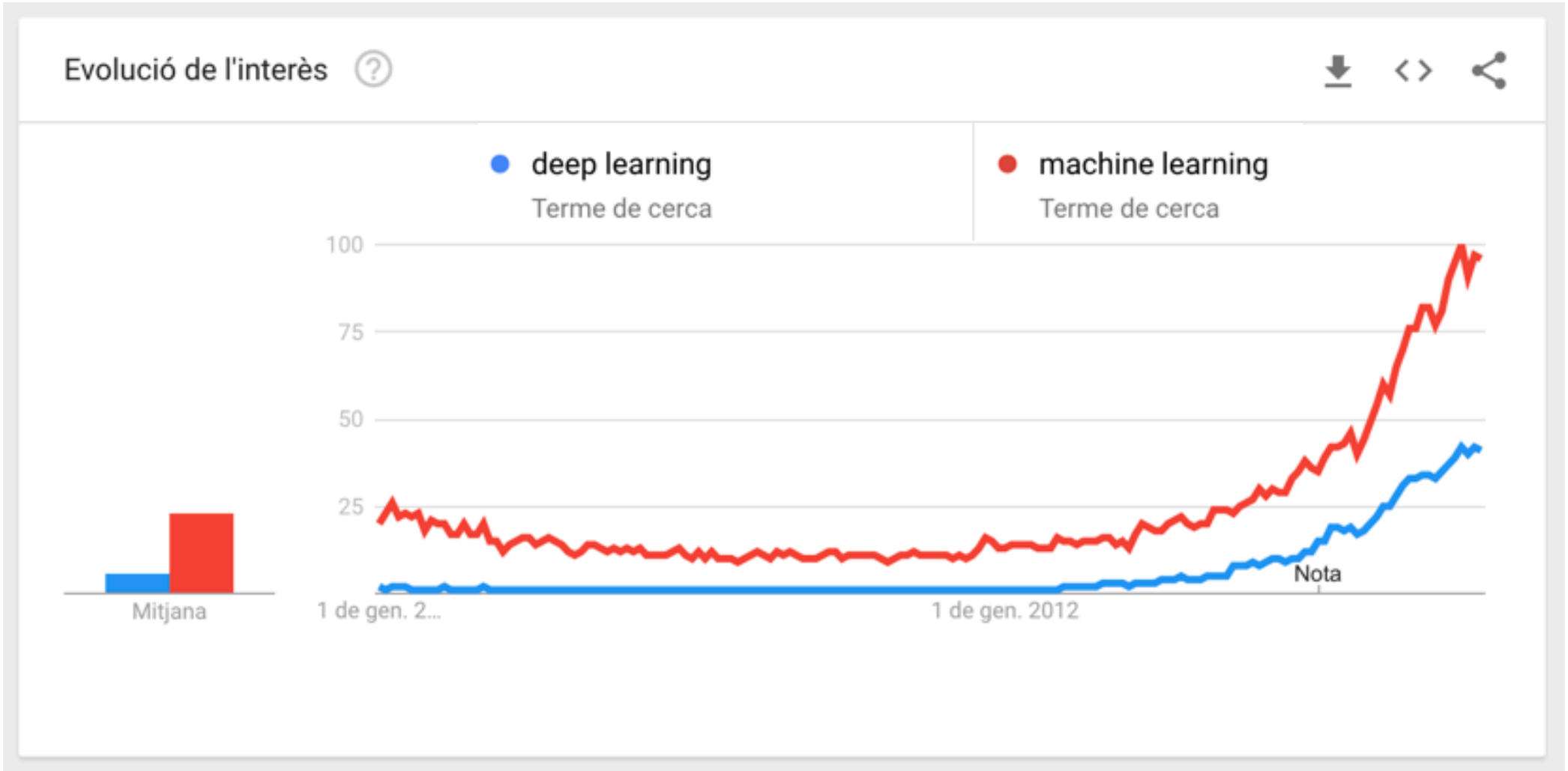
Second neural network winter is coming

- With the ascent of Support Vector Machines and the failure of backpropagation, the early 2000s were a dark time for neural net research.

- Then, what every researcher must dream of actually happened: G.Hinton, S.Osindero, and Y.W.Teh published a paper in 2006 that was seen as a breakthrough, a breakthrough significant enough to rekindle interest in neural nets: *A fast learning algorithm for* ***deep*** *belief nets.*

- After that, following Moore's law, computers got dozens of times faster (GPUs) since the slow days of the 90s, making learning with large datasets and many layers much more tractable.

Neural Networks Reborn



Google Trends

Roll over image to zoom in

NVIDIA

NVIDIA Jetson TK1 Development Kit

33 customer reviews | 24 answered questions

List Price: ~~$199.99~~

Price: $170.77 + $49.57 Shipping & Import Fees Deposit to Spain Details

You Save: $29.22 (15%)

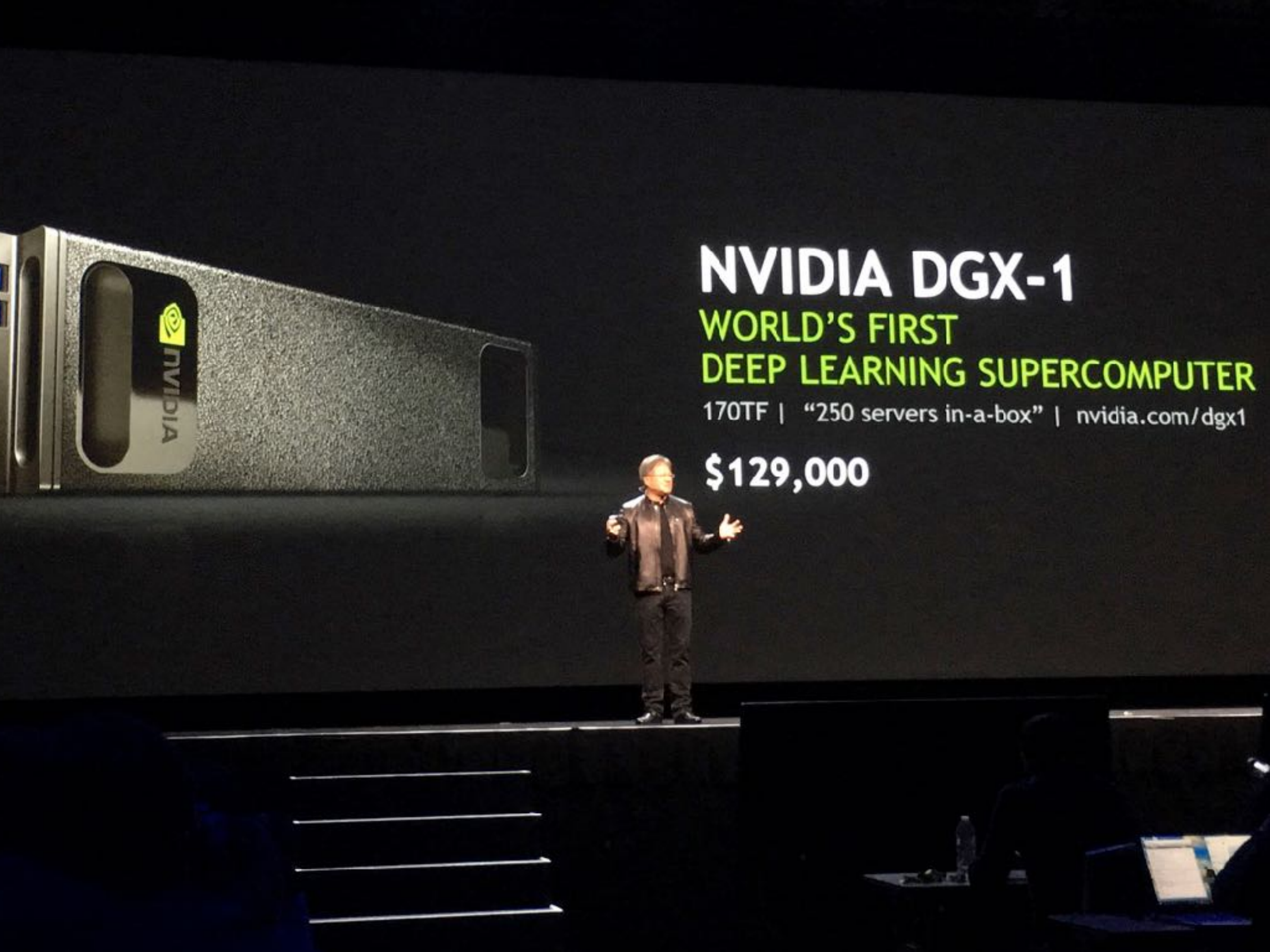
Item is eligible: No interest if paid in full within 6 months with the Amazon.com Store Card.

In Stock.

This item ships to **Barcelona, Spain**. **Want it Monday, Feb. 12?** Order within **5 hrs 25 mins** and choose **AmazonGlobal Priority Shipping** at checkout. Learn more

Ships from and sold by Amazon.com. Gift-wrap available.

- NVIDIA Kepler GPU with 192 CUDA cores
- NVIDIA 4-Plus-1 quad-core ARM Cortex-A15 CPU
- 2 GB memory, 16 GB eMMC
- Gigabit Ethernet, USB 3.0, SD/MMC, miniPCIe
- HDMI 1.4, SATA, Line out/Mic in, RS232 serial port
- Expansion ports for additional display, GPIOs, and high-bandwidth camera interface


NVIDIA
NVIDIA DGX-1
WORLD'S FIRST
DEEP LEARNING SUPERCOMPUTER
170TF | "250 servers in-a-box" | nvidia.com/dgx1
$129,000

STANDARD MACHINE LEARNING

DECISION

Loss Function
Batch Optimization

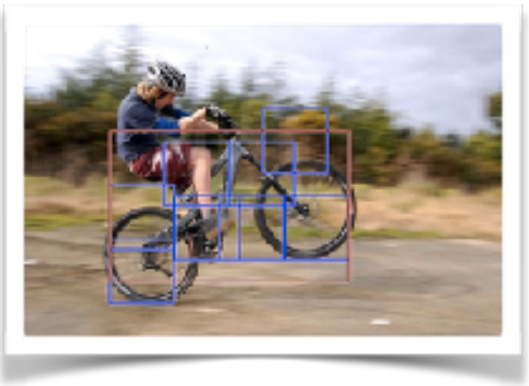
Trainable classifier

Feature Library
Custom Features

Hand-engineered features

Raw Data

Example



A few classes +
Hinge Loss (SVM) +
Sequential Minimal Optimization +
HOG +
Deformable Part Models

DEEP LEARNING

Backpropagation + Tricks

DECISION

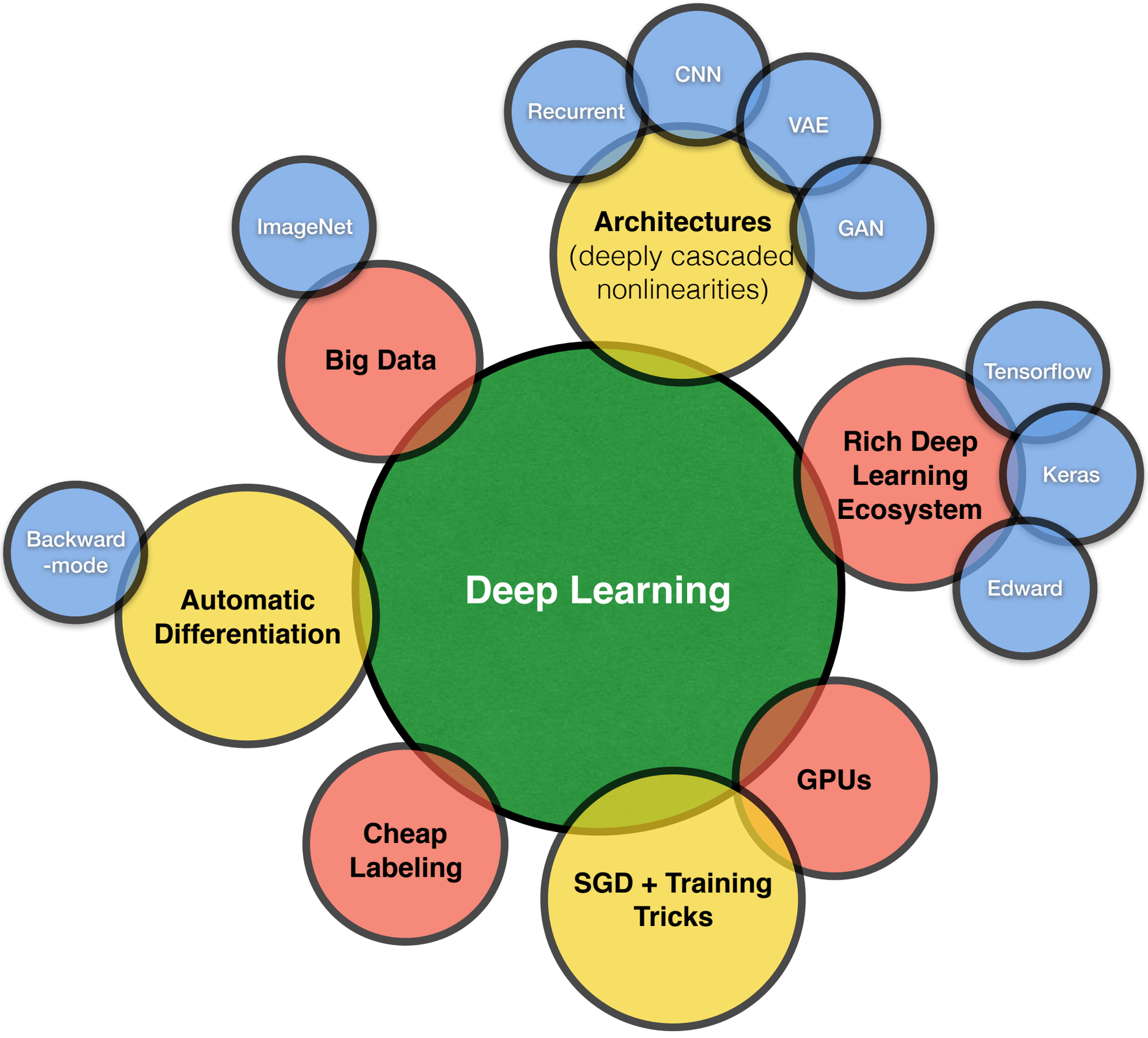
Trainable classifier

+

Trainable features

Raw Data

Loss Function
Network Architecture
Stochastic Gradient Descend


Recurrent
CNN
VAE
GAN
Architectures
(deeply cascaded
nonlinearities)
ImageNet
Big Data
Tensorflow
Keras
Rich Deep
Learning
Ecosystem
Edward
Backward
-mode
Automatic
Differentiation
Deep Learning
GPUs
Cheap
Labeling
SGD + Training
Tricks

Definitions

- **Neural Networks (NN)** is a beautiful biologically-inspired programming paradigm which enables a computer to learn from observational data.

- **Deep Learning (DL)** is a powerful set of techniques (tricks?) for learning in neural networks.

- NN and DL currently provide the best solutions to many problems in image recognition, speech recognition, and natural language processing.

Basic learning setup with one function

Training data: a set of $(x^{(m)}, y^{(m)})$ pairs.
Learn a function $f_w : x \to y$ to predict on new inputs x.

1. Choose a model function family f_w.

2. Optimize parameters w.

High capacity models

- How to find the parameters of the function?

- We can use optimization techniques (minimizing a function, the **loss function**, that measures the discrepancy between the outcomes of a model and the desired outcomes).

- To optimize, we must **compute the derivative of every parameter with respect to the loss function**.

- But we have (possibly) **millions of parameters** and the loss function is a (possibly) large composition of functions…

Automatic Differentiation

```
import autograd.numpy as np   # Thinly-wrapped version of Numpy
from autograd import grad

def taylor_sine(x):  # Taylor approximation to sine function
    ans = currterm = x
    i = 0
    while np.abs(currterm) > 0.001:
        currterm = -currterm * x**2 / ((2 * i + 3) * (2 * i + 2))
        ans = ans + currterm
        i += 1
    return ans

grad_sine = grad(taylor_sine)
print "Gradient of sin(pi) is", grad_sine(np.pi)
```

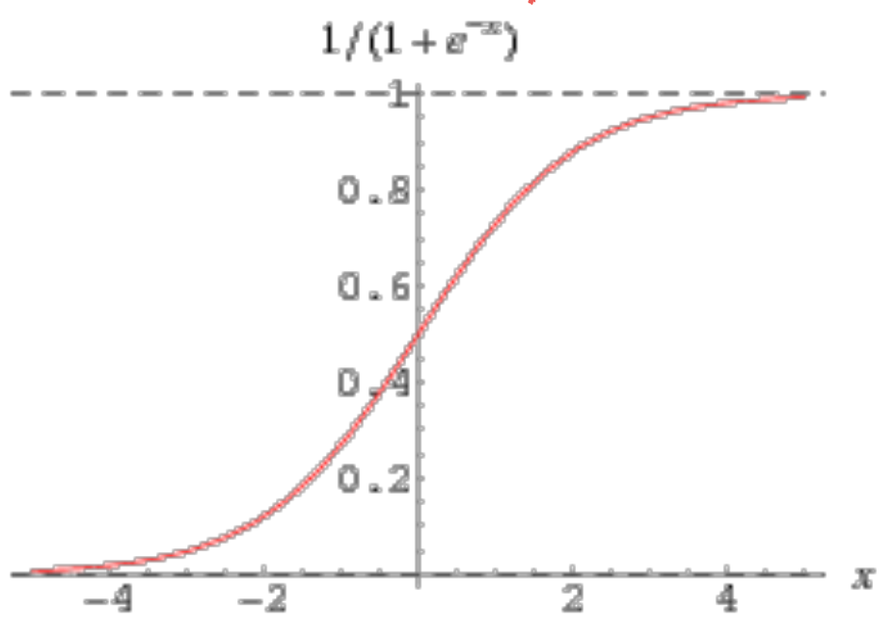
SGD-based logistic regression

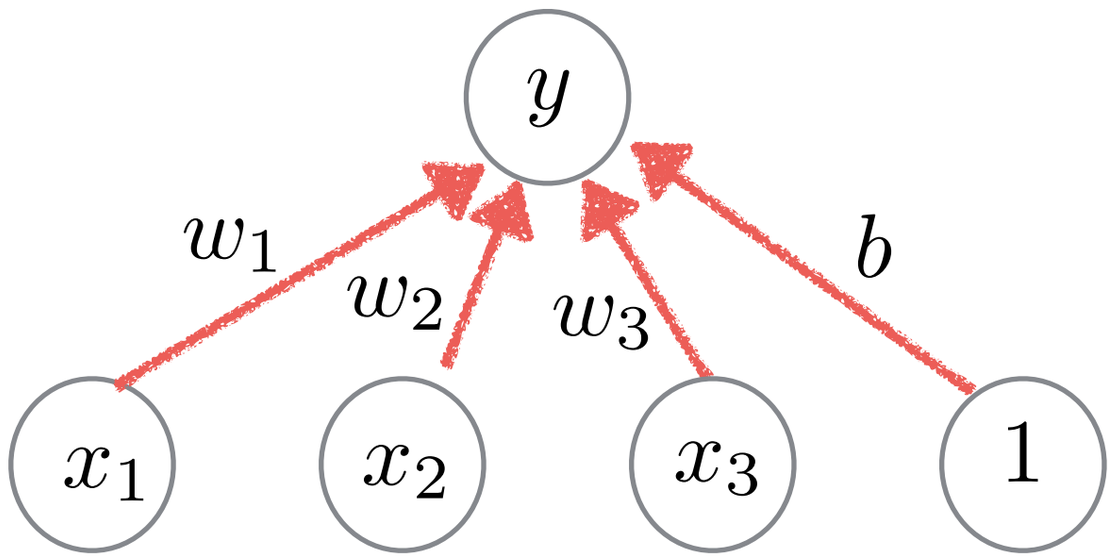
```
import autograd.numpy as np
from autograd import grad

def sigmoid(x):
    return 0.5*(np.tanh(x) + 1)

def logistic_predictions(weights, inputs):
    # Outputs probability of a label being true according to logistic model.
    return sigmoid(np.dot(inputs, weights))

def training_loss(weights):
    # Training loss is the negative log-likelihood of the training labels.
    preds = logistic_predictions(weights, inputs)
    label_probabilities = preds * targets + (1 - preds) * (1 - targets)
    return -np.sum(np.log(label_probabilities))

# Build a toy dataset.
inputs = np.array([[0.52, 1.12,  0.77],
                   [0.88, -1.08, 0.15],
                   [0.52, 0.06, -1.30],
                   [0.74, -2.49, 1.39]])
targets = np.array([True, True, False, True])

# Define a function that returns gradients of training loss using autograd.
training_gradient_fun = grad(training_loss)

# Optimize weights using gradient descent.
weights = np.array([0.0, 0.0, 0.0])
print "Initial loss:", training_loss(weights)
for i in xrange(100):
    weights -= training_gradient_fun(weights) * 0.01

print  "Trained loss:", training_loss(weights)
```

Neural Networks

1-Layer Net function model: $f(x) = \sigma(w^T \cdot x + b)$

Non linear function. F.e. sigmoid





Neural Networks

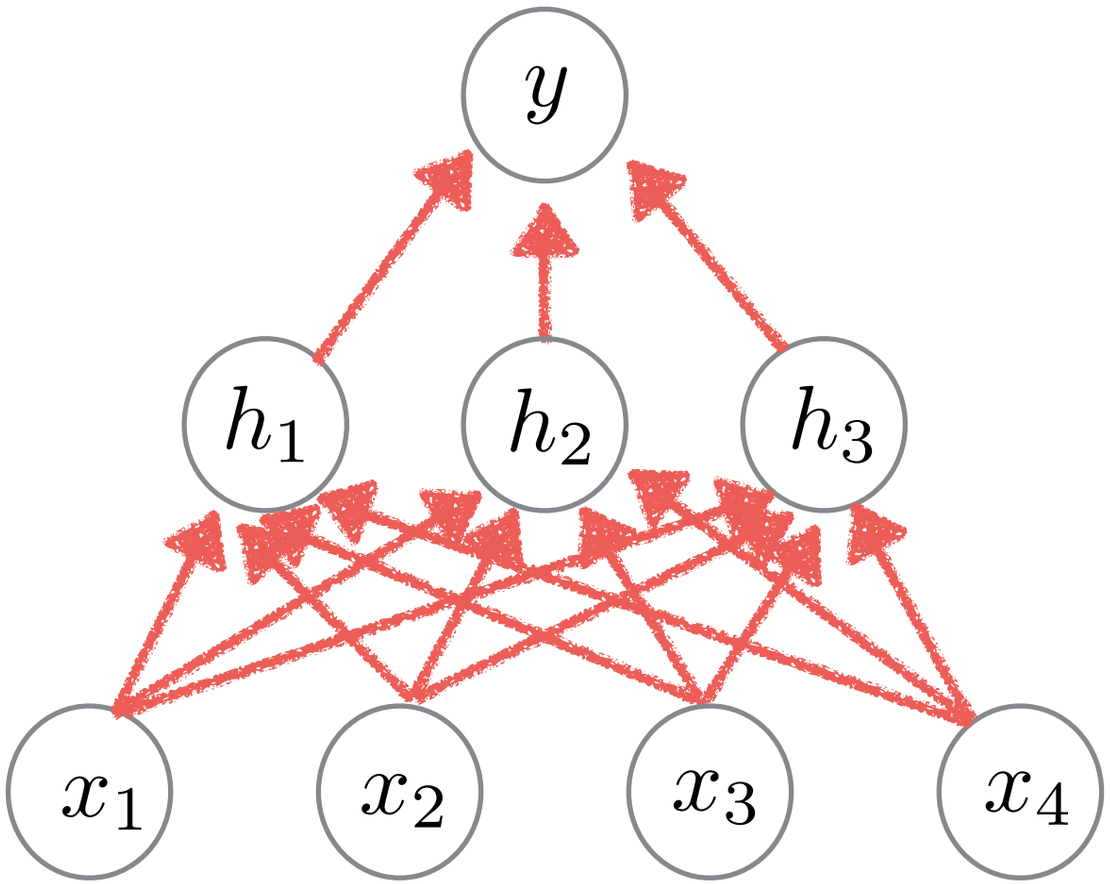
1-Layer Net function model: $f(x) = \sigma(w^T \cdot x + b)$

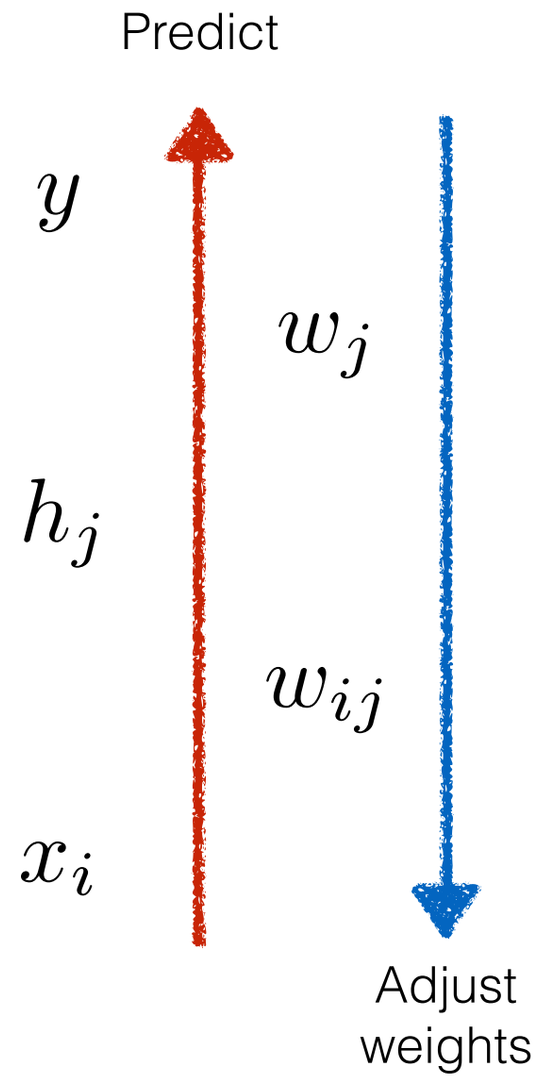
Training:

1. Assume a Squared-Error (or Cross Entropy) Loss $Loss(w) = \frac{1}{2}\sum_m (\sigma(w^T x^{(m)}) - y^{(m)})^2$
2. Initialize w
3. Compute $\nabla_w Loss = \sum_m Error^{(m)} * \sigma'(w^T x^{(m)}) * x^m$
4. $w \rightarrow w - \gamma(\nabla_w Loss)$
5. Repeat steps 3-4 until some condition satisfied

1-layer nets model linear hyperplanes. It is a trainable system but it is not powerful.

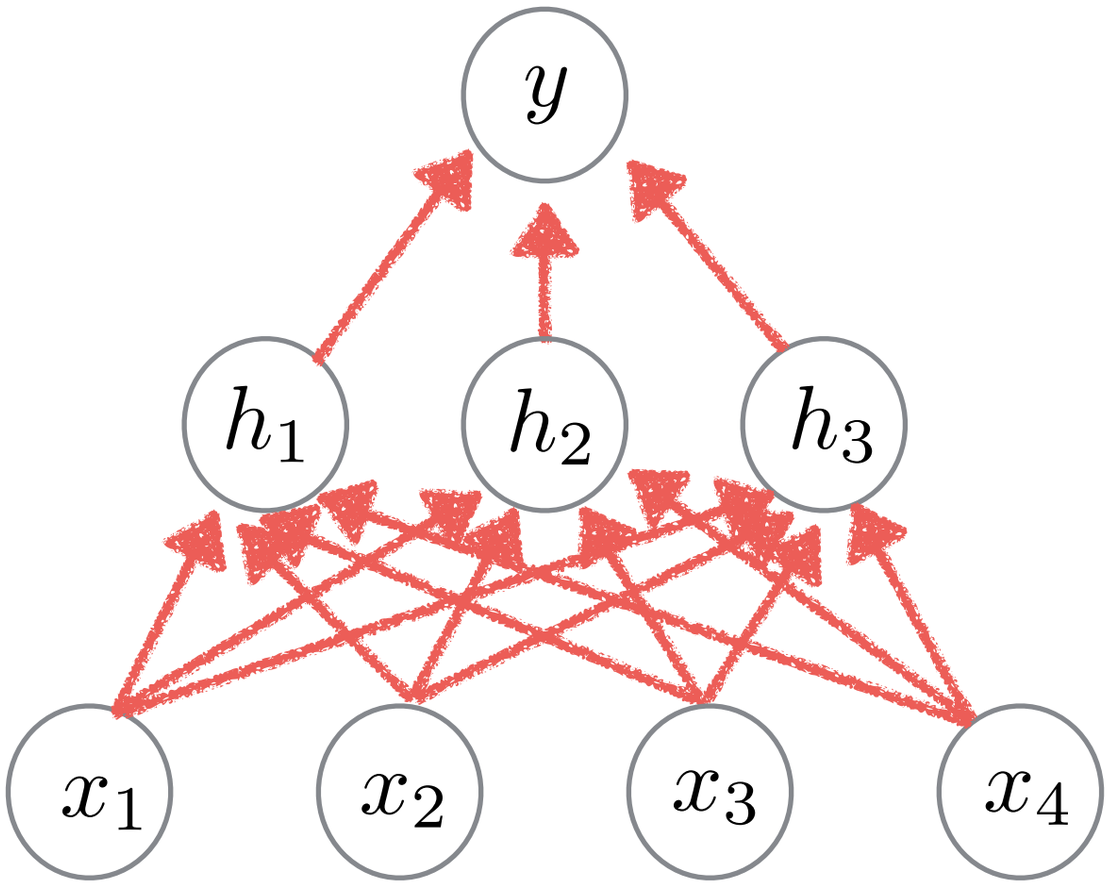
Training a two layer by **backpropagation**

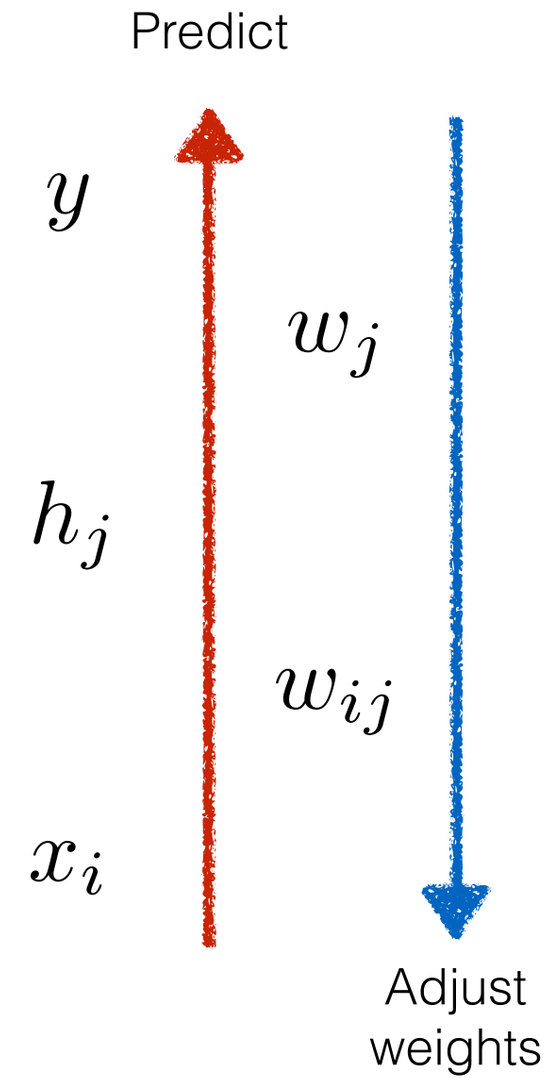




$$f(x) = \sigma(\sum_j w_j \cdot h_j) = \sigma(\sum_j w_j \cdot \sigma(\sum_i w_{ij} x_i))$$

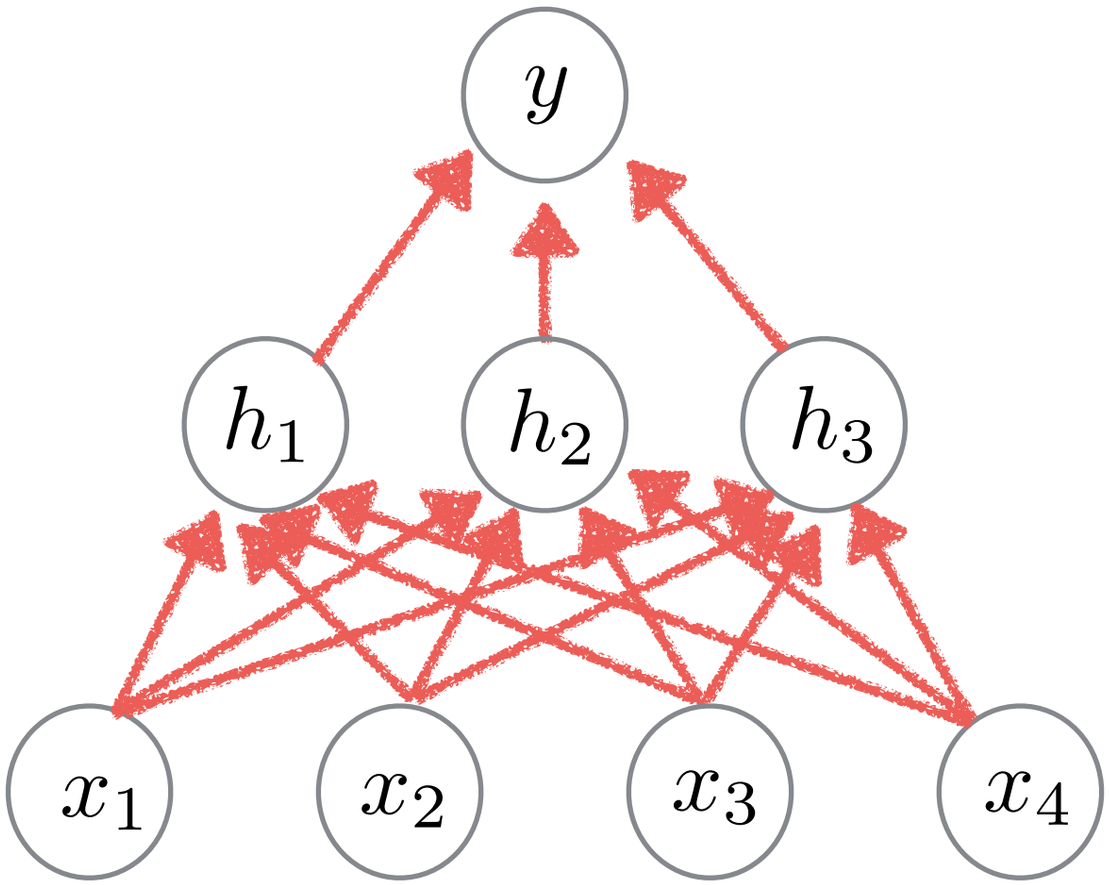
Training a two layer by backpropagation

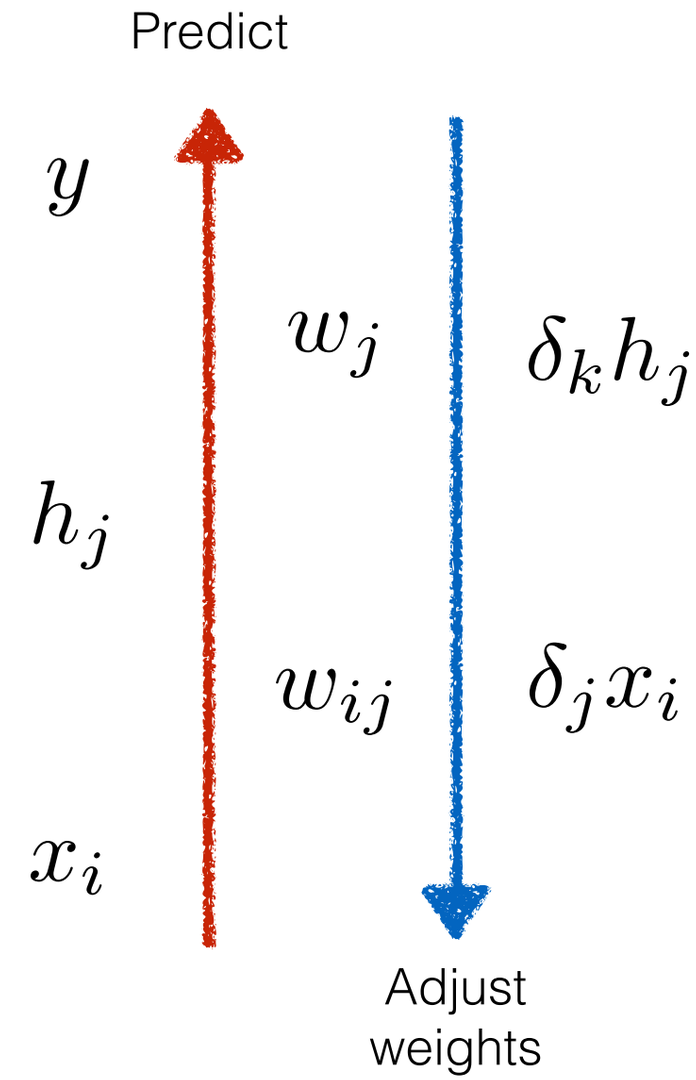




1. For each sample, compute $f(x^{(m)}) = \sigma(\sum_j w_j \cdot \sigma(\sum_i w_{ij} x_i^{(m)}))$.
2. If $f(x^{(m)}) \neq y^{(m)}$, back-propagate error and adjust weights $\{w_{ij}, w_j\}$

Training a two layer by backpropagation



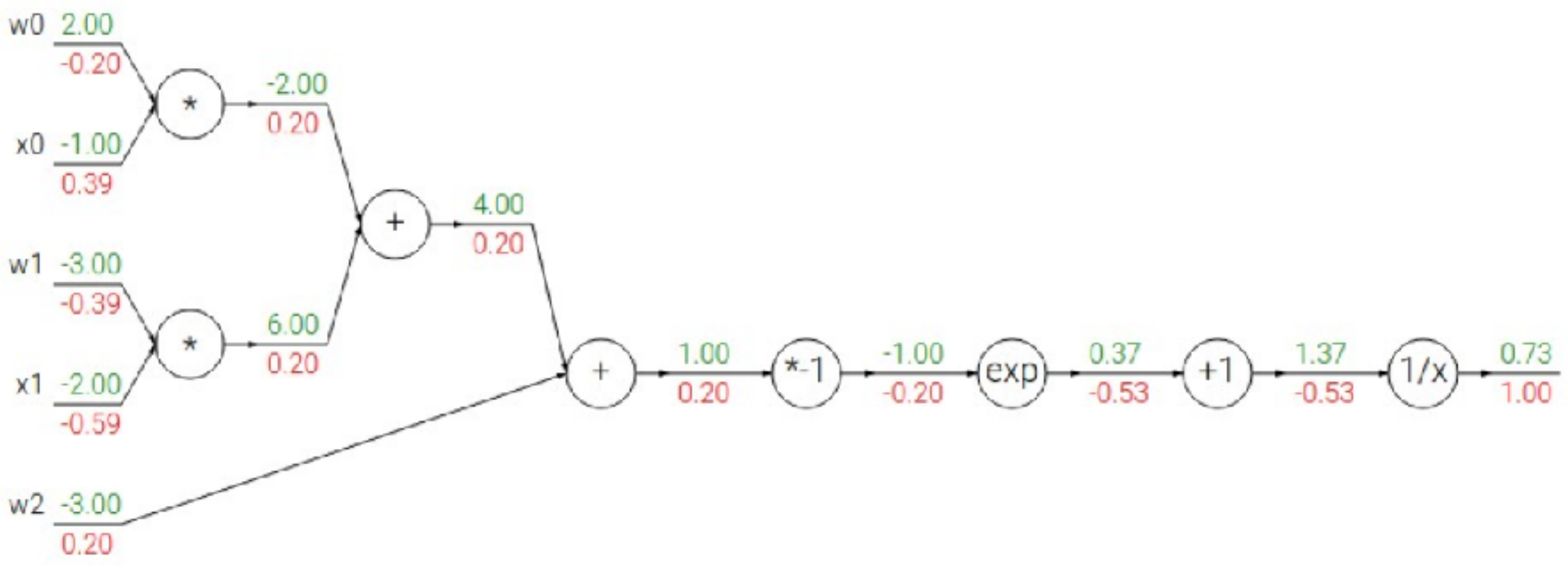


All updates involve some scaled error from output * input feature.

2-layer nets are universal function approximators (given infinite hidden nodes). It is a powerful system but when the number of hidden nodes increases it is not trainable.

Backpropagation

$$f(w,x) = \frac{1}{1 + e^{-(w_0 x_0 + w_1 x_1 + w_2)}}$$

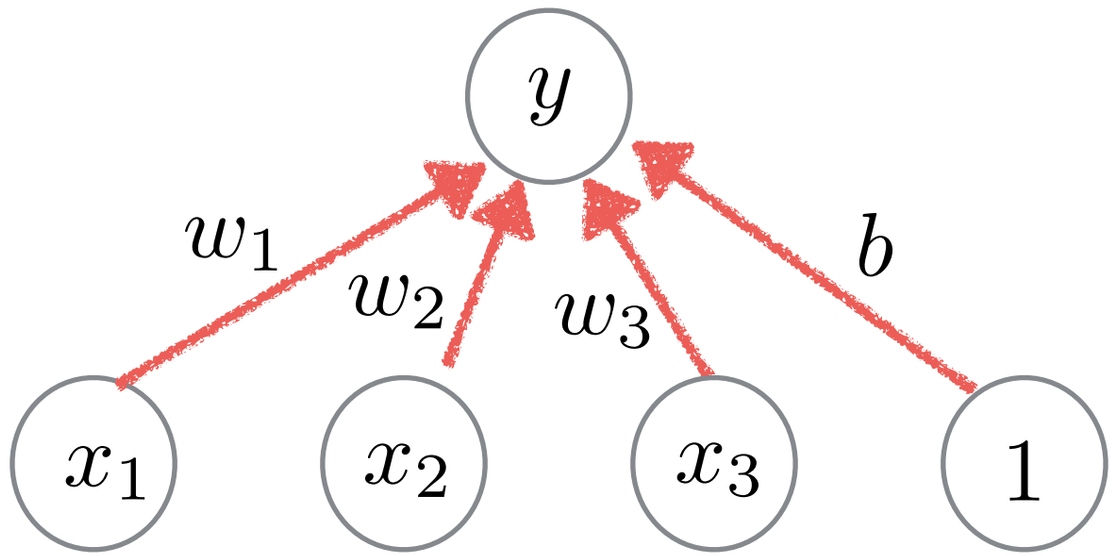


Backpropagation

We can structure our code in **layers**, where you can derive the local gradients, then chain the gradients during backpropagation.

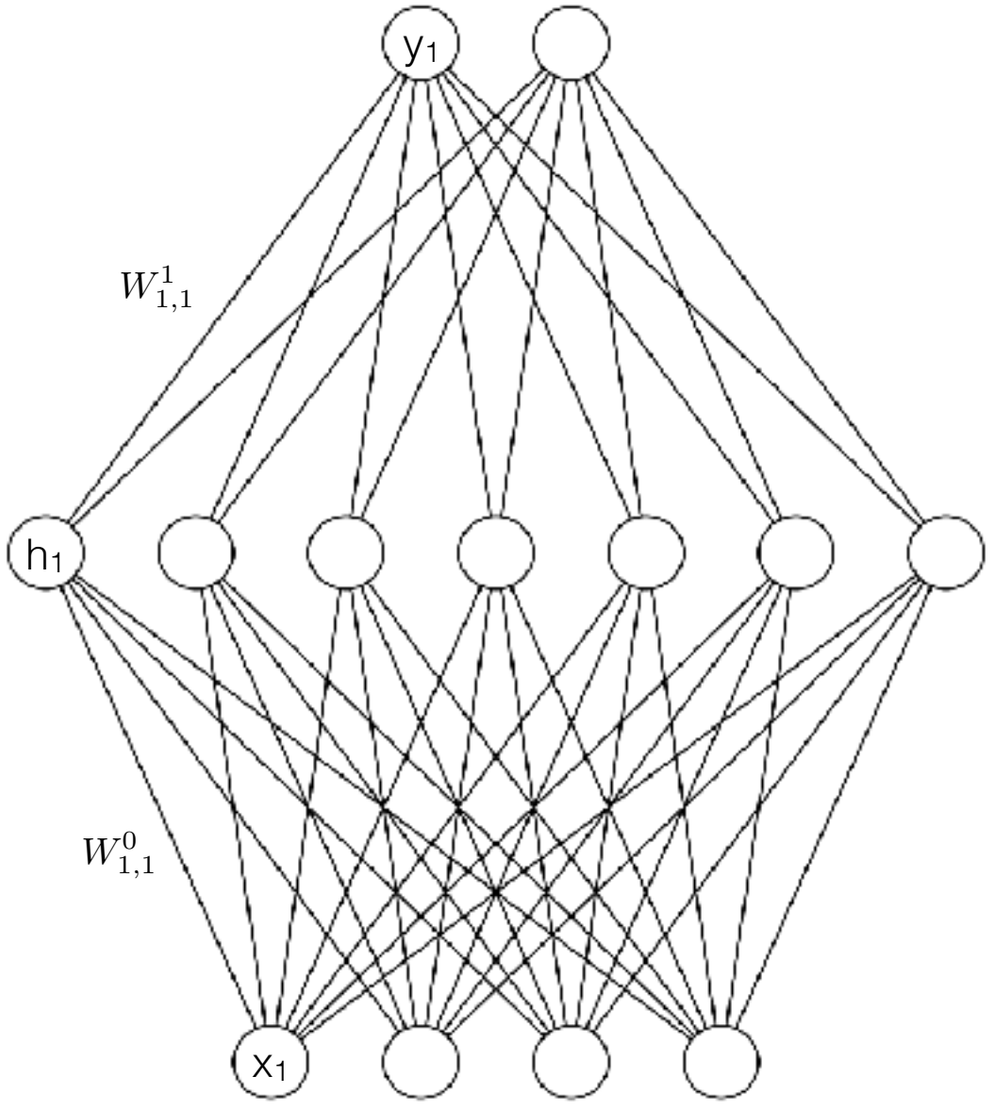
1-layer neural net model

$$f(x) = \sigma(w^T \cdot x + b)$$



Graphical Representation

2-layer neural net model



Output Layer
(y1, y2)

Weights W1

$y_i = \sigma(\sum_{j=1}^{7} h_i * W^1_{i,j})$

Hidden Layer
(h1, h2, h3, h4, h5, h6 h7)

Weights W0

$h_i = \sigma(\sum_{j=1}^{7} x_i * W^0_{i,j})$

Input Layer
(x1, x2, x3, x4)

Deep Models

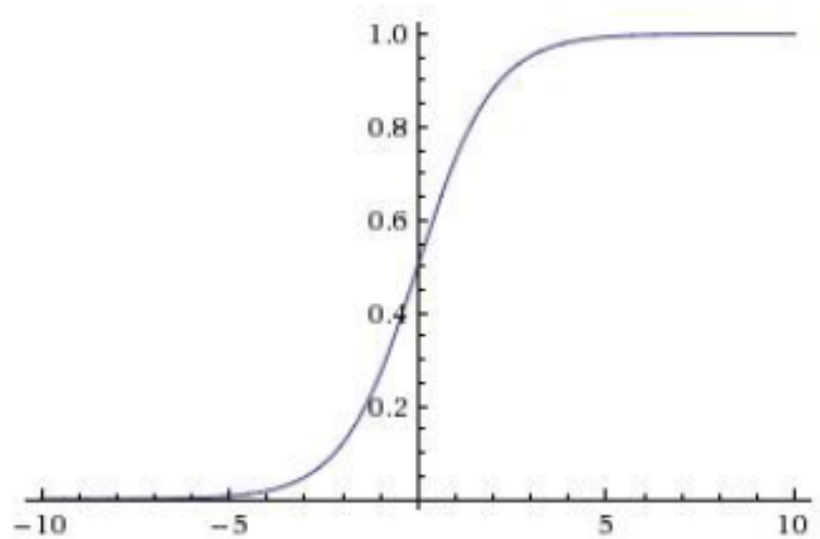
[1]Inception 5 (GoogLeNet)

Inception 7a

[1]Going Deeper with Convolutions, [C. Szegedy et al, CVPR 2015]


TRIPS
TRICKS

Activation functions



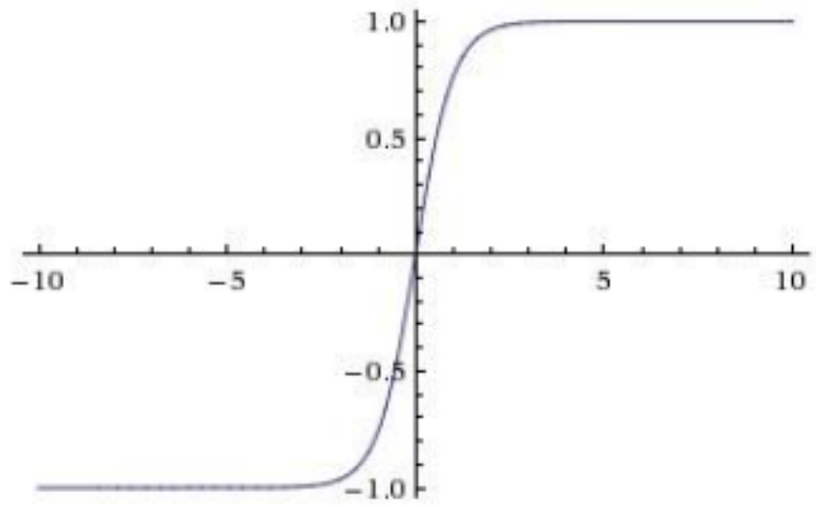
Sigmoid

- Squashes numbers to range [0,1].
- Historically popular since they have nice interpretation as a saturating “firing rate” of a neuron.

2 BIG problems:

- Saturated neurons “kill” the gradients.
- Sigmoid outputs are not zero centered.

Activation functions



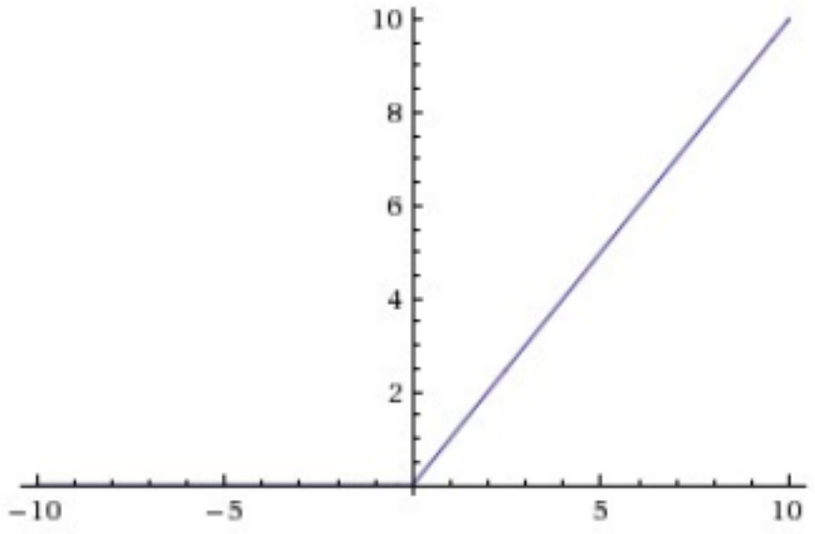
tanh(x)

- Squashes numbers to range [-1,1].
- Zero centered.

1 BIG problem:

- Saturated neurons “kill” the gradients.

Activation functions



ReLU

- Does not saturate.
- Computationally efficient.
- Fast convergence.

1 small problem:

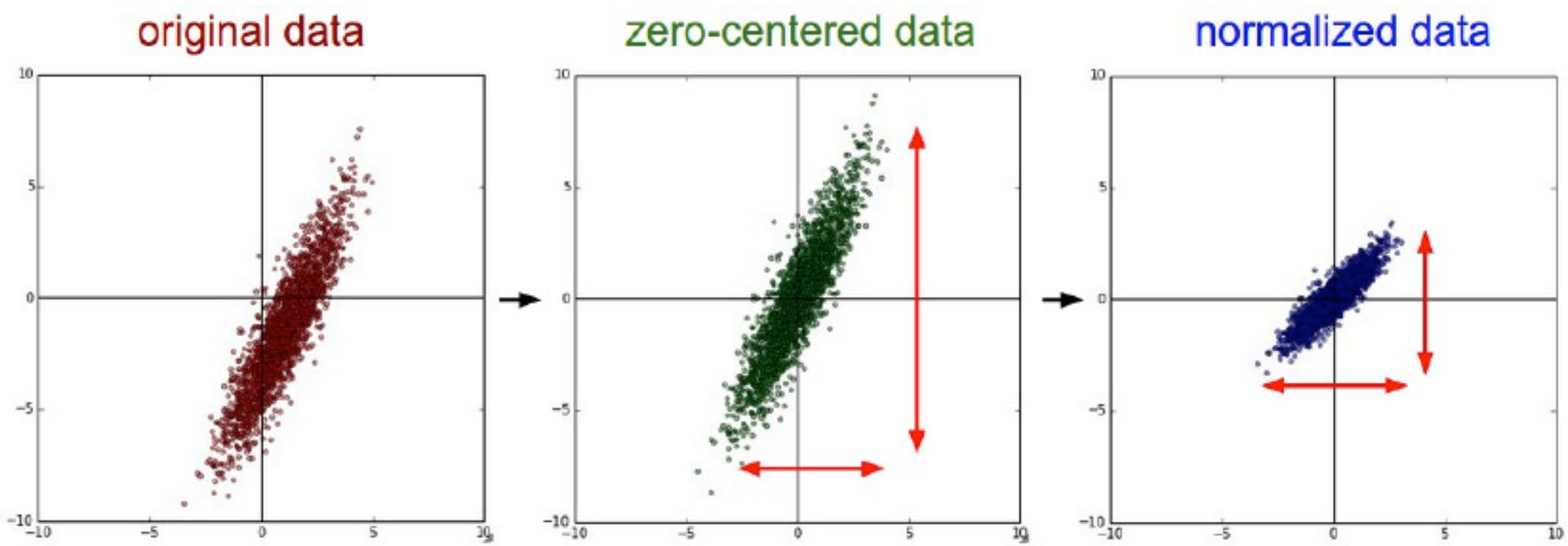
- Dead ReLU ($x < 0$).

Activation functions

- Use ReLU. Be careful with your learning rates
- Try out tanh but don’t expect much
- Never use sigmoid

Training

Step 1: Preprocess the data




```
X -= np.mean(X, axis = 0)
```

```
X /= np.std(X, axis = 0)
```

Training

Step 2: Initialize well

- Set weights to small random numbers.
- Set biases to zero.

Training

Step 3: Use stochastic gradient descent.

Strategies to build powerful decision functions

Given a dictionary of non linear decision functions $g_1, \ldots, g_n$, we can build a non linear decision function by combination or by composition:

1. Combination $\quad f(x) = a_1 g_1(x) + \cdots + a_n g_n(x)$

2. Composition $\quad f(x) = g_1(g_2(\ldots(g_n(x))\ldots))$

Deep Learning is based on the composition of some differentiable functions that produce a **trainable** and **powerful** system.

Given same number of units, a deeper architecture is more expressive than a shallow one.

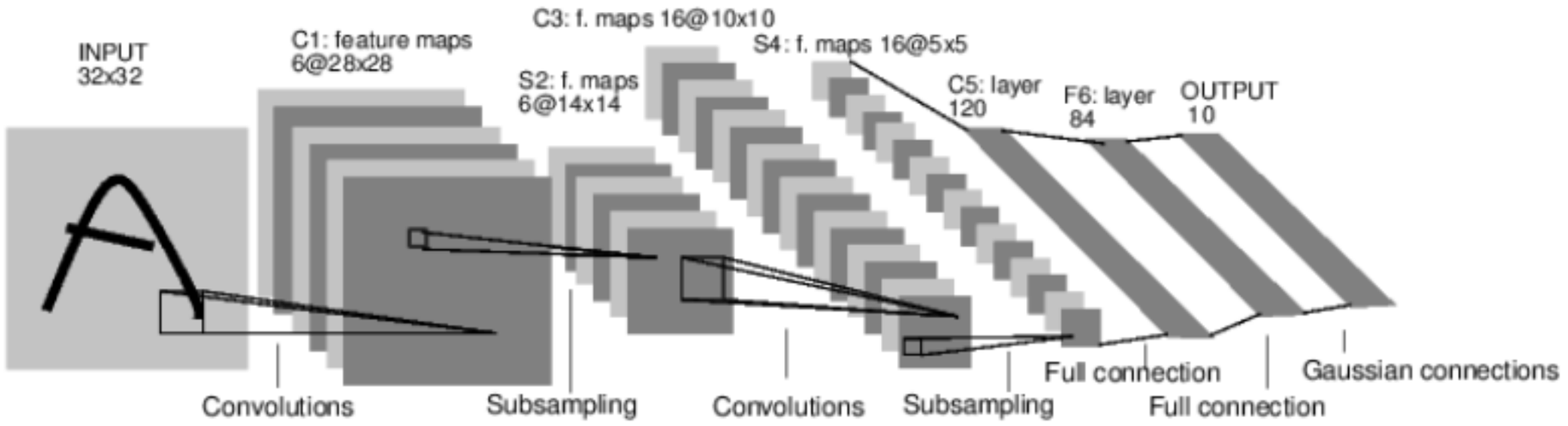
1-layer nets model linear hyperplanes

2-layer nets are universal function approximators (given infinite hidden nodes)

3-layer nets are universal function approximators with fewer nodes/weights

A DNN with two hidden layers and a modest number of hidden units can sort N N-bit numbers.

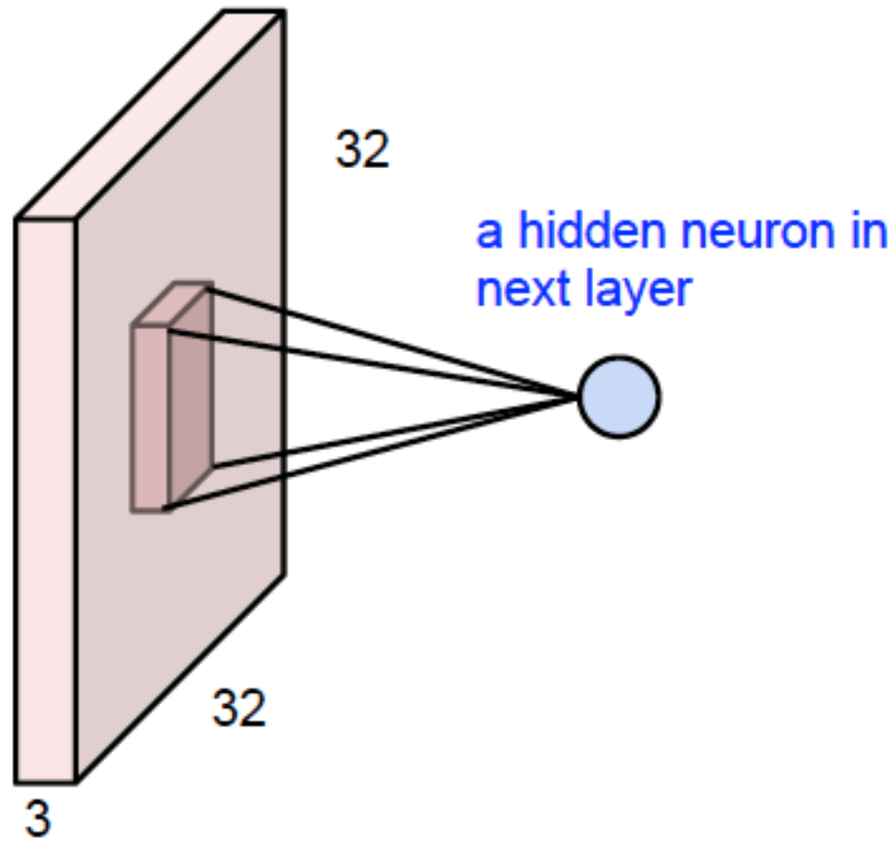
Convolutional neural networks



LeNet, 1980

Convolutional neural networks

They are just neural networks but with **local connectivity and shared weights.**

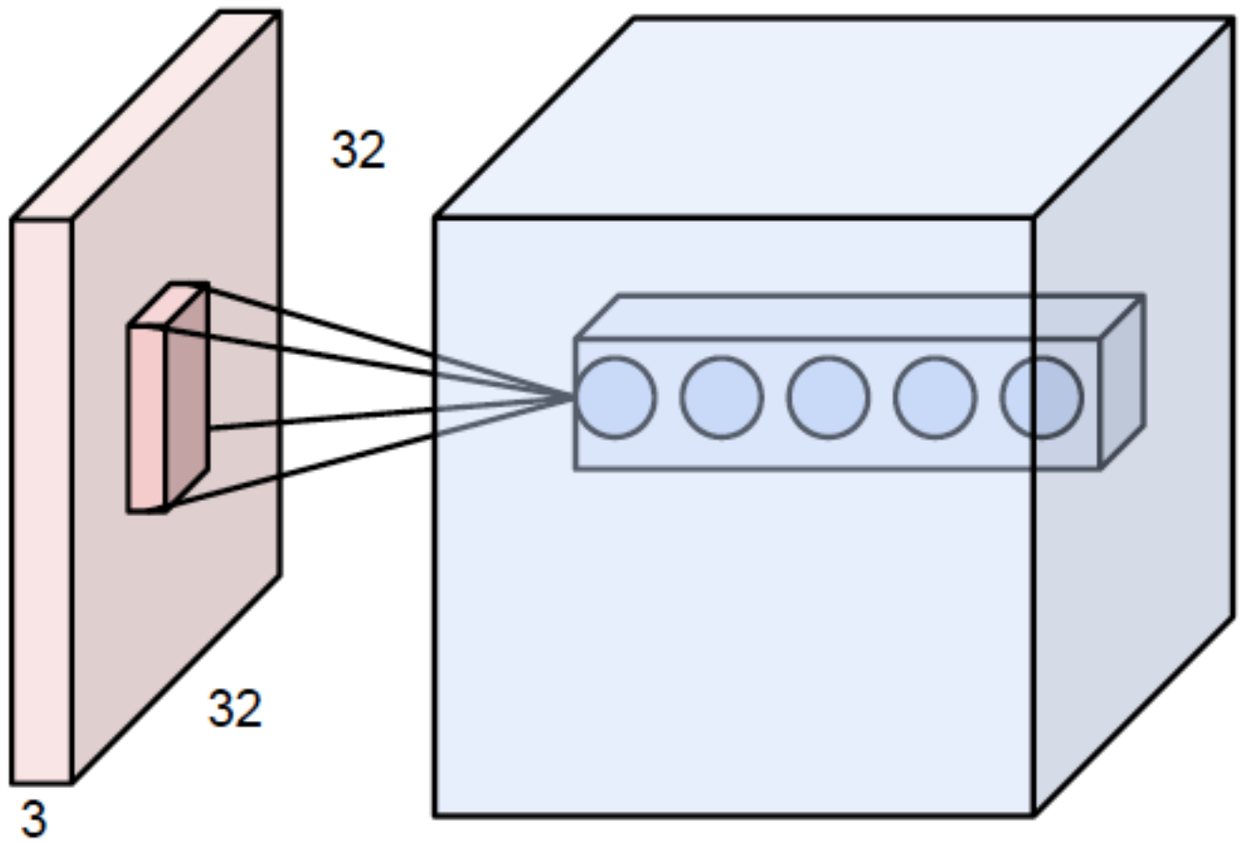


Full connectivity: **32x32x3 weights**

Local connectivity: one neuron will connect to, e.g. 5x5x3 chunk and only have **5x5x3 weights**.

Convolutional neural networks

They are just neural networks but with **local connectivity and shared weights.**



Multiple neurons all looking at the same region of the input volume, stacked along depth.

Convolutional neural networks

Input [32 x 32 x3], 30 neurons (bands) with receptive fields 5x5:
=> Output volume: [32 x 32 x 30] (= 30720 neurons)

Each neuron has 5*5*3 (=75) weights
=> Number of weights in such layer: 30720 * 75 ~ 3 million

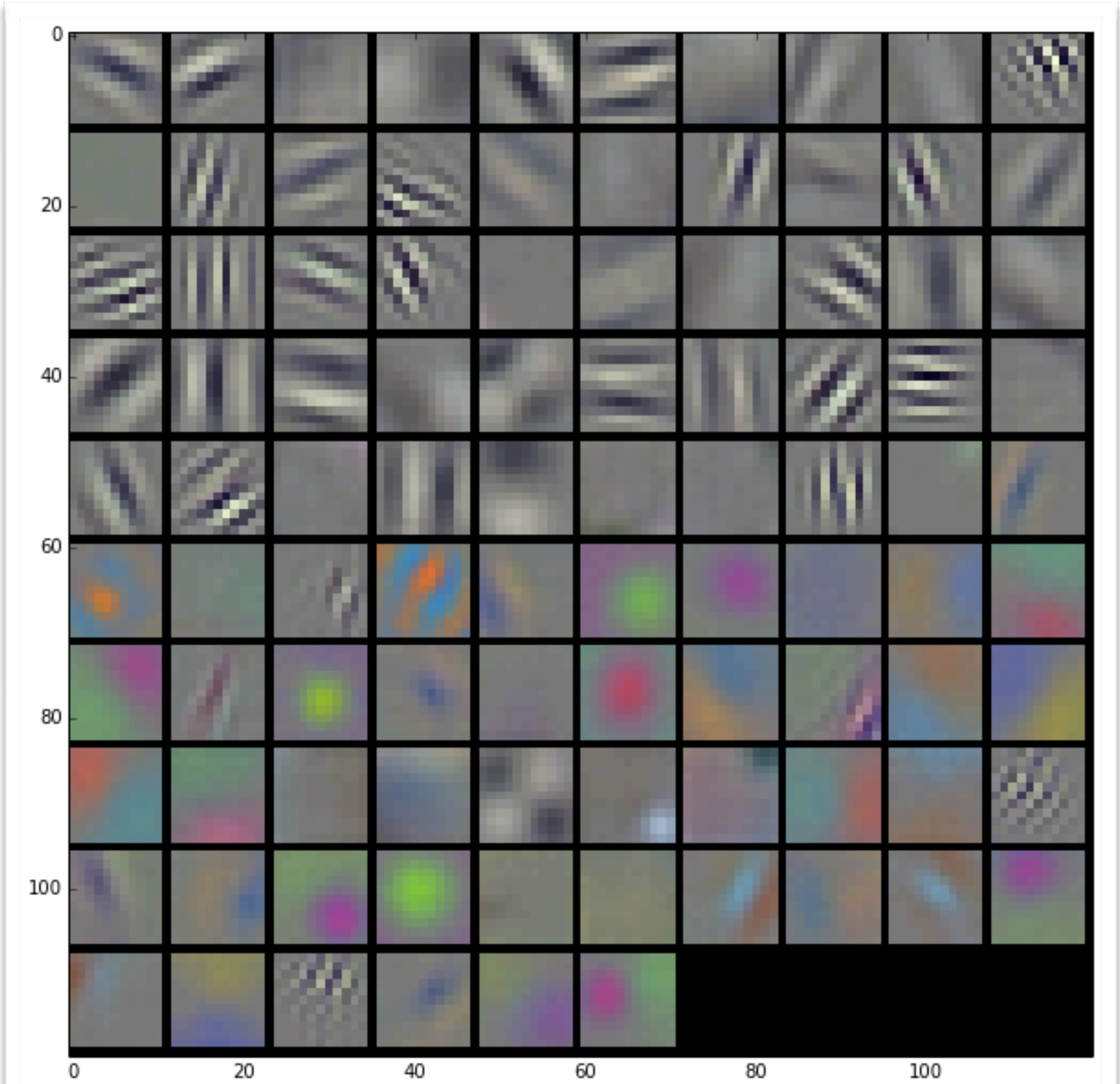
Convolutional neural networks

PARAMETER SHARING: lets not learn the same thing across all spatial locations - > 30 different (spatially invariant) filters
instead of 30 specific filters at each location.

Before:
#weights in such layer: (32*32*30) * 75 = 3 million

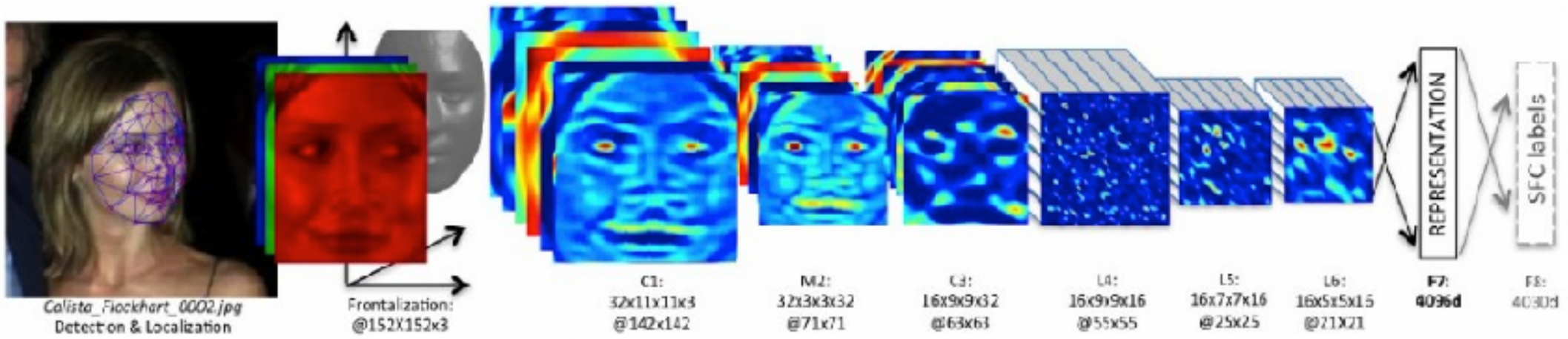
Now: (parameter sharing)
#weights in the layer: 30 * 75 = 2250.

Convolutional neural networks



Convolutional neural networks

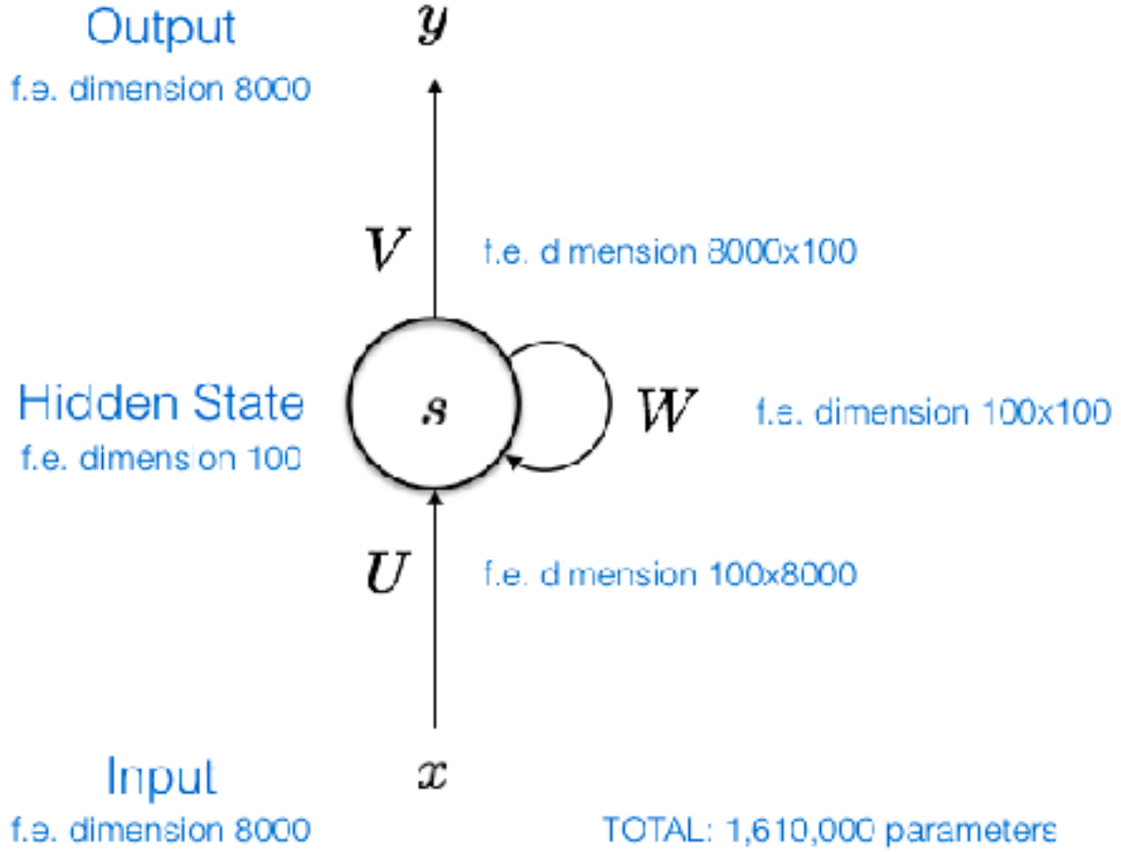
Sometimes it’s not a good idea to share the parameters



Recurrent neural layer model

Variable-size inputs + sequences

$$\mathbf{s}_t = \tanh\left(U\mathbf{x}_t + W\mathbf{s}_{t-1}\right)$$
$$\mathbf{y}_t = V\mathbf{s}_t$$



Recurrent neural layer model

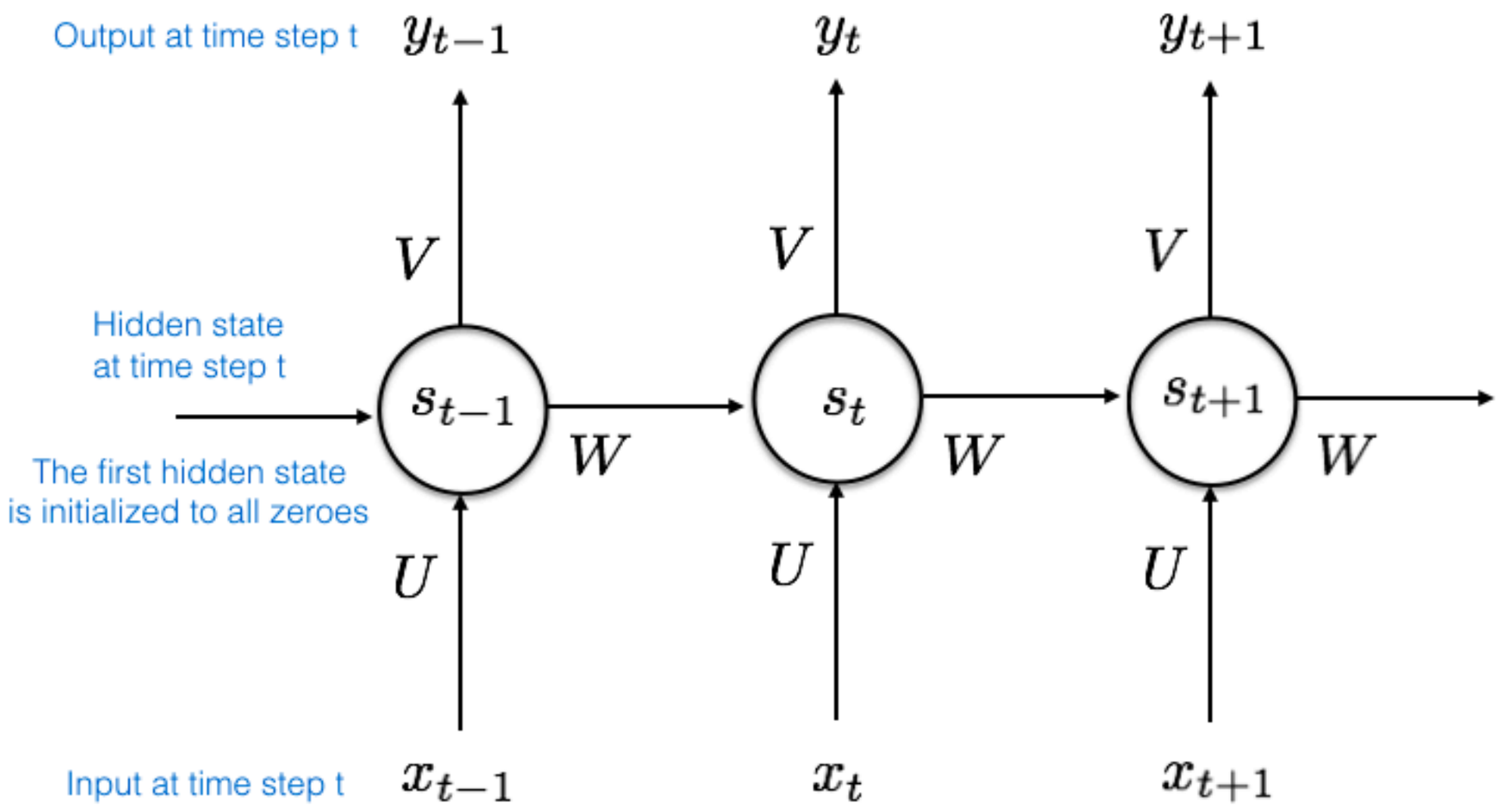
Variable-size inputs + sequences

$$\mathbf{s}_t = \tanh\left(U\mathbf{x}_t + W\mathbf{s}_{t-1}\right)$$
$$\mathbf{y}_t = V\mathbf{s}_t$$

```
class RNN:
    #...
    def step(self,x):
        self.h = np.tanh(np.dot(self.W_ss, self.h) +
                         np.dot(self.W_xs, self.x))
        y = np.dot(self.W_sy, self.h)
        return y
    #...
```

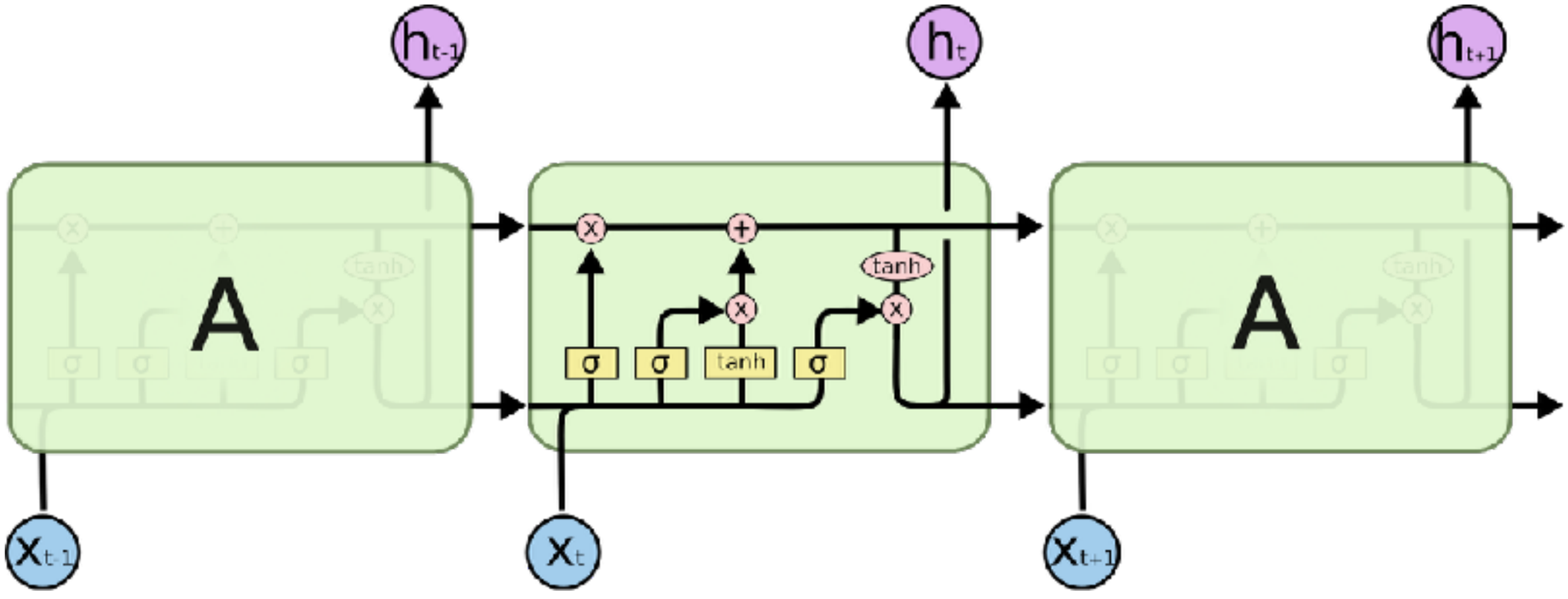
Recurrent neural layer model

Variable-size inputs + sequences

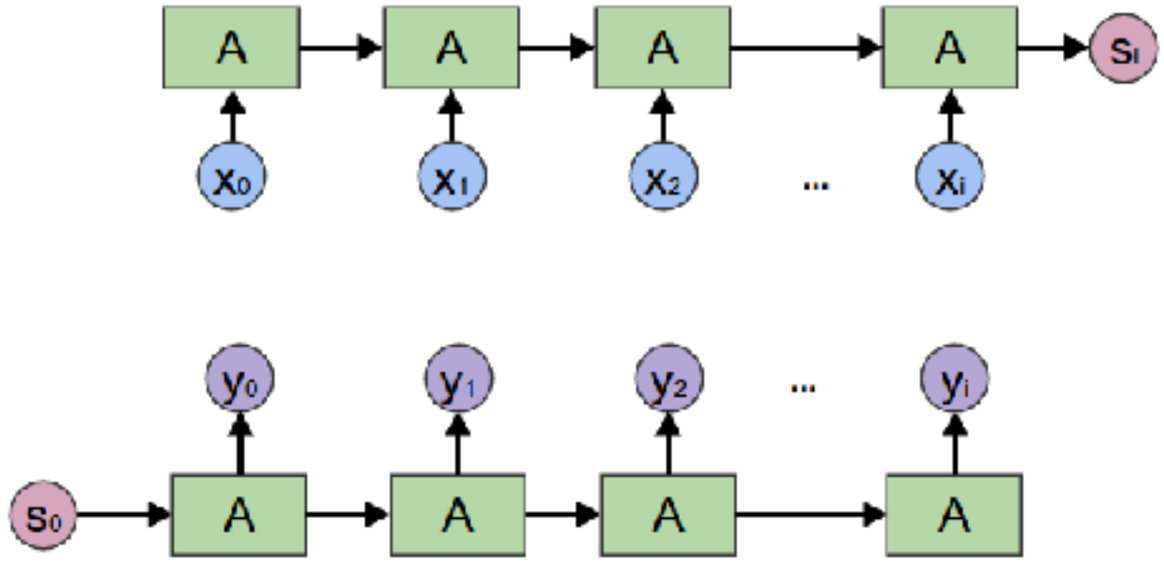


Recurrent neural layer model

Variable-size inputs + sequences

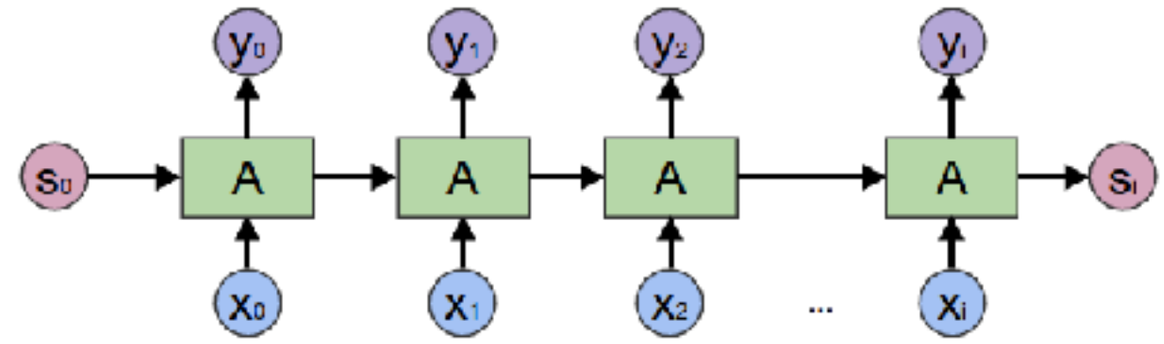


Recurrent neural layer model

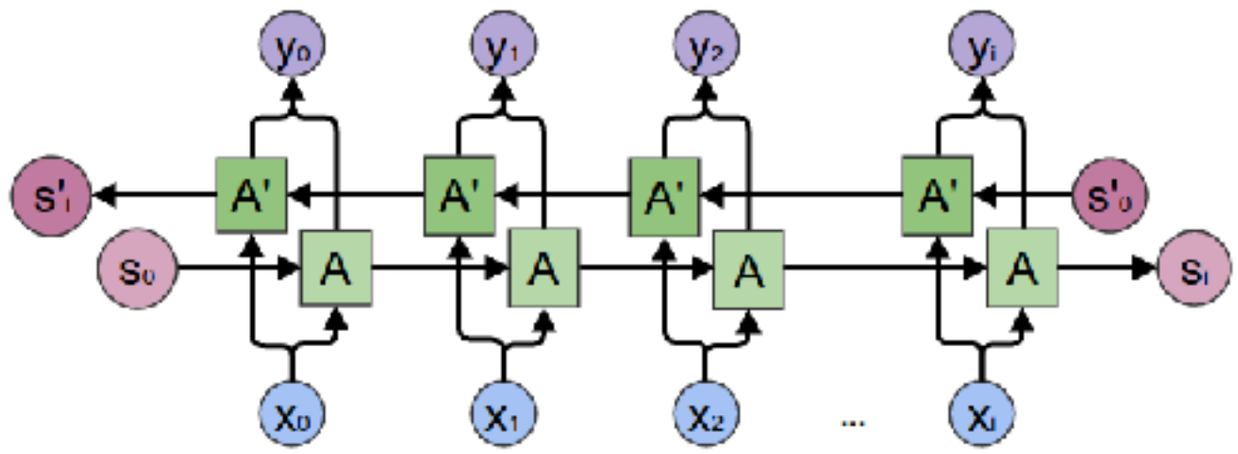


Encoder

Decoder

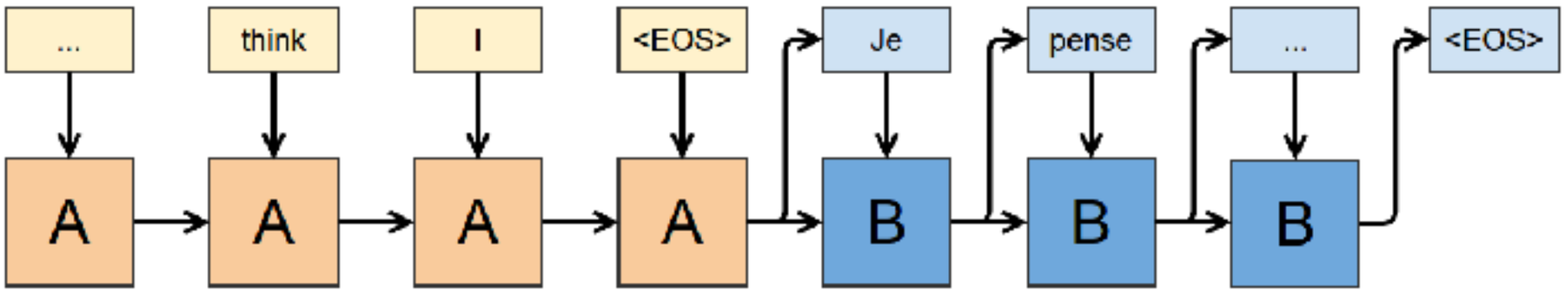


Encoder-Decoder



Bidirectional

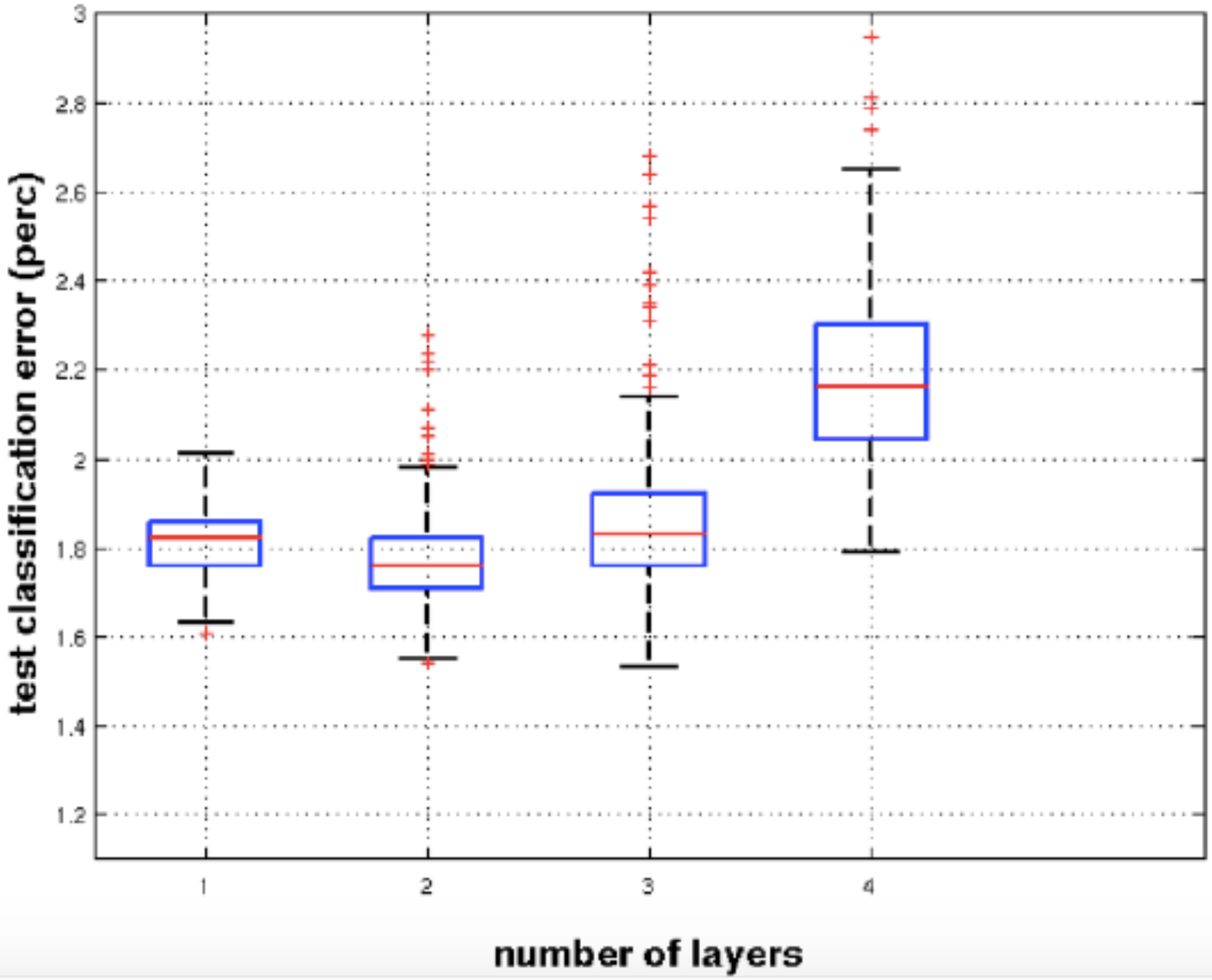
Recurrent neural layer model



But deep networks are hard to train….

Vanishing gradient problem: δ_j may vanish after repeated multiplication.

- MNIST digit classification task; 400 trials (random seed)
- Each layer: initialize w_{ij} by uniform$[-1/\sqrt{(FanIn)}, 1/\sqrt{(FanIn)}]$
- Although $L+1$ layers is more expressive, worse error than L layers



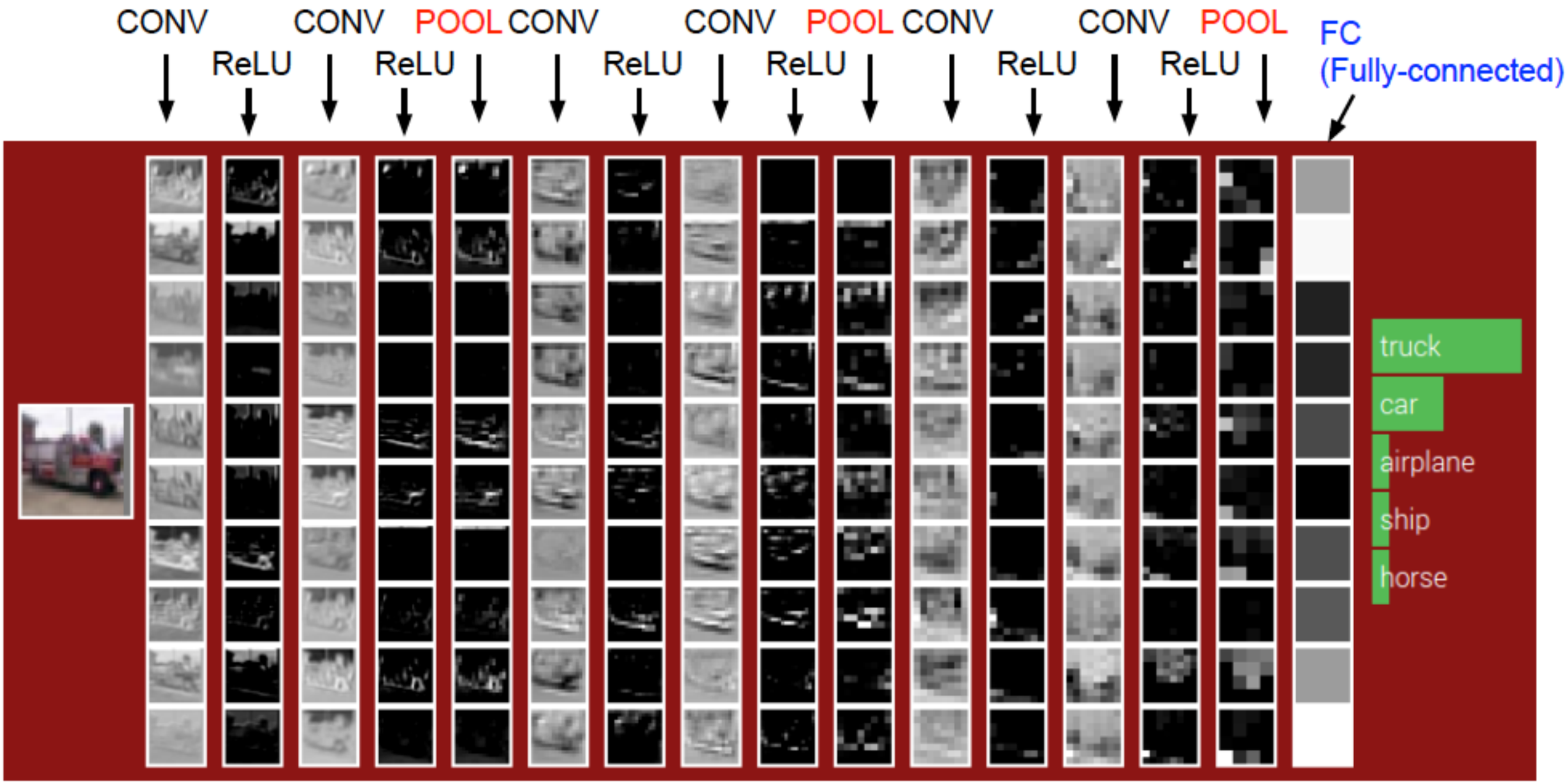
What changed since the 80's?

- **Computers were slow**. So the neural networks of past were tiny. And tiny neural networks cannot achieve very high performance on anything. In other words, small neural networks are not powerful.

- **Datasets were small**. So even if it was somehow magically possible to train LDNNs, there were no large datasets that had enough information to constrain their numerous parameters. So failure was inevitable.

- **Nobody knew how to train deep nets**. The current best object recognition networks have between 20 and 100 successive layers of convolutions. A 2 layer neural network cannot do anything good on object recognition. Yet back in the day everyone was very sure that deep nets cannot be trained with SGD, since that would've been too good to be true!

Convolutional neural networks



Modern CNNs:

- use filter sizes of 3x3 (maybe even 2x2 or 1x1!)
- use pooling sizes of 2x2 (maybe even less - e.g. fractional pooling!)
- stride 1
- very deep

Learning

The success of Deep Learning hinges on a very fortunate fact: that well-tuned and carefully-initialized **stochastic gradient descent** (SGD) can train LDNNs on problems that occur in practice. It is not a trivial fact since the training error of a neural network as a function of its weights is **highly non-convex**.

The problem of training neural networks is NP-hard, and in fact there exists a family of datasets such that the problem of finding the best neural network with three hidden units is NP-hard. And yet, SGD just solves it in practice. This is the main pillar of deep learning.

Learning

Climbing the correlation mountain

We can say fairly confidently that successful LDNN training relies on the “easy” **correlation** in the data, which allows learning to bootstrap itself towards the more “complicated” correlations in the data.

Neural networks start their learning process by noticing the most “blatant” correlations between the input and the output, and once they notice them they introduce several hidden units to detect them, which enables the neural network to see more complicated correlations.

Training deep networks: practical advice

Get the data: Make sure that you have a high-quality dataset of input-output examples that is large, representative, and has relatively clean labels. Learning is completely impossible without such a dataset.

Training deep networks: practical advice

Preprocess the data:

- It is essential to center the data so that its **mean is zero and so that the variance of each of its dimensions is one**. Sometimes, when the input dimension varies by orders of magnitude, it is better to take the $log(1+x)$ of that dimension.

- This is the case because the weights are updated by the formula: change in $w_{ij} \propto x_i(\delta L/\delta y_i)$

  (w denotes the weights from layer x to layer y, and L is the loss function). If the average value of the x's is large (say, 100), then the weight updates will be very large and correlated, which makes learning bad and slow. Keeping things zero-mean and with small variance simply makes everything work much better.

Training deep networks: practical advice

Use minibatches. Modern computers cannot be efficient if you process one training case at a time. It is vastly more efficient to train the network on minibatches of 128 examples, because doing so will result in massively greater throughput.

It would actually be nice to use minibatches of size 1, and they would probably result in improved performance and lower overfitting; but the benefit of doing so is outweighed the massive computational gains provided by minibatches. But don't use very large minibatches because they tend to work less well and overfit more. So the practical recommendation is: **use the smaller minibatch that runs efficiently on your machine.**

Training deep networks: practical advice

Gradient normalization: Divide the gradient by minibatch size. This is a good idea because of the following pleasant property: you won't need to change the learning rate (not too much, anyway), if you double the minibatch size (or halve it).

Training deep networks: practical advice

Learning rate schedule: Start with a normal-sized learning rate (LR) and reduce it towards the end.

- A typical value of the LR is 0.1.
- Use a validation set to decide when to lower the learning rate and when to stop training (e.g., when error on the validation set starts to increase).
- A practical suggestion for a learning rate schedule: if you see that you stopped making progress on the validation set, divide the LR by 2 (or by 5), and keep going. Eventually, the LR will become very small, at which point you will stop your training.
- Worry about the Learning Rate. One useful idea is to monitor the ratio between the update norm and the weight norm. This ratio should be at around 10^{-3}. If it is much smaller then learning will probably be too slow, and if it is much larger then learning will be unstable and will probably fail.

Training deep networks: practical advice

Weight initialization. Worry about the random initialization of the weights at the start of learning.

- If you are lazy, it is usually enough to do something like 0.02 * randn(num_params).
- If it doesn't work well (say your neural network architecture is unusual and/or very deep), then you should initialize each weight matrix with the init_scale / sqrt(layer_width) * randn. In this case init_scale should be set to 0.1 or 1, or something like that.

Training deep networks: practical advice

Fun story: researchers believed, for many years, that SGD cannot train deep neural networks from random initializations.

Every time they would try it, it wouldn't work. Embarrassingly, they did not succeed because they used the "small random weights" for the initialization, which works great for shallow nets but simply doesn't work for deep nets at all. When the nets are deep, the many weight matrices all multiply each other, so the effect of a suboptimal scale is amplified.

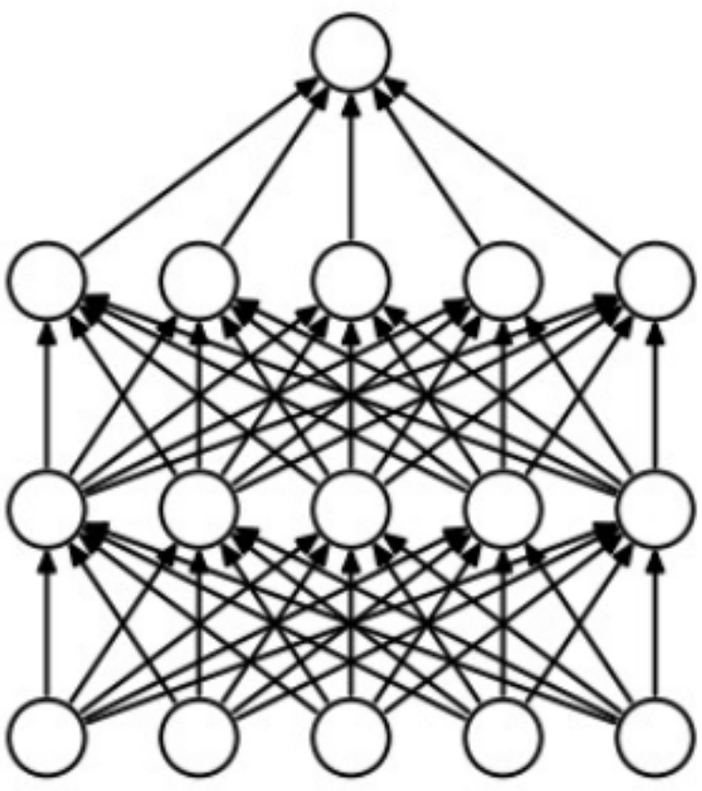
Training deep networks: practical advice

Data augmentation: be creative, and find ways to algorithmically increase the number of training cases that are in your disposal.

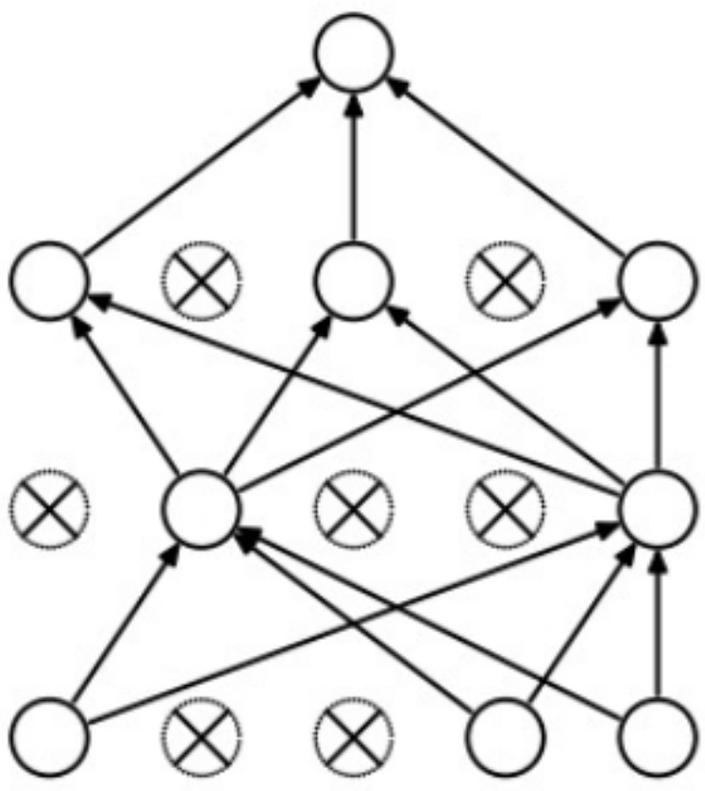
If you have images, then you should translate and rotate them; if you have speech, you should combine clean speech with all types of random noise; etc. Data augmentation is an art (unless you're dealing with images). Use common sense.

Training deep networks: practical advice

Dropout. Dropout provides an easy way to improve performance. It's trivial to implement and there's little reason to not do it.



(a) Standard Neural Net



(b) After applying dropout.

Training deep networks: practical advice

Dropout. Dropout provides an easy way to improve performance. It's trivial to implement and there's little reason to not do it.

Remember to tune the dropout probability, and to not forget to turn off Dropout and to multiply the weights by (namely by 1-dropout probability) at test time. Also, be sure to train the network for longer. Unlike normal training, where the validation error often starts increasing after prolonged training, dropout nets keep getting better and better the longer you train them. So be patient.

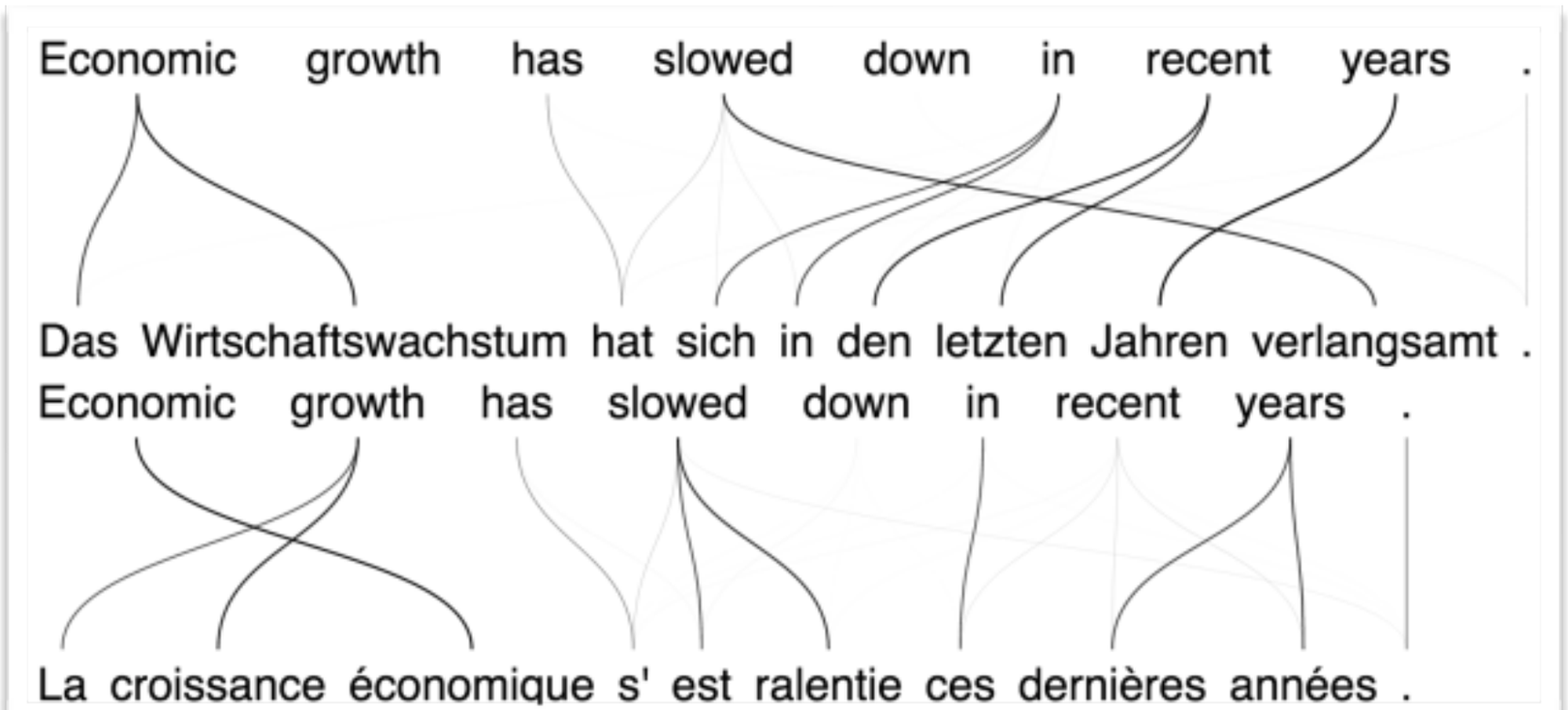
Training deep networks: practical advice

Ensembling. Train 10 neural networks and average their predictions. It's a fairly trivial technique that results in easy, sizeable performance improvements.

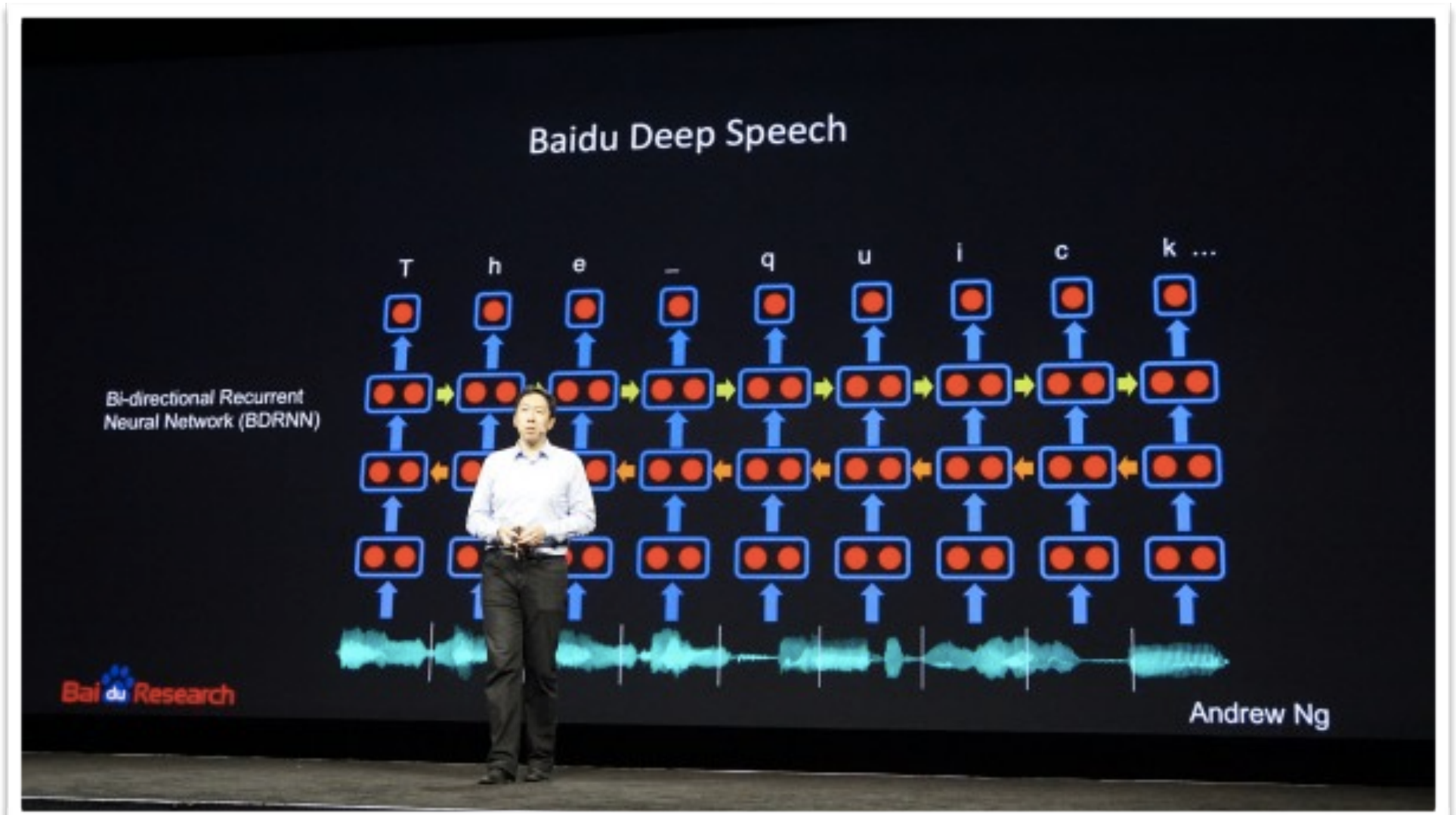
One may be mystified as to why averaging helps so much, but there is a simple reason for the effectiveness of averaging. Suppose that two classifiers have an error rate of 70%. Then, when they agree they are right. But when they disagree, one of them is often right, so now the average prediction will place much more weight on the correct answer. The effect will be especially strong whenever the network is confident when it's right and unconfident when it's wrong.

If you mix all these ingredients you can…

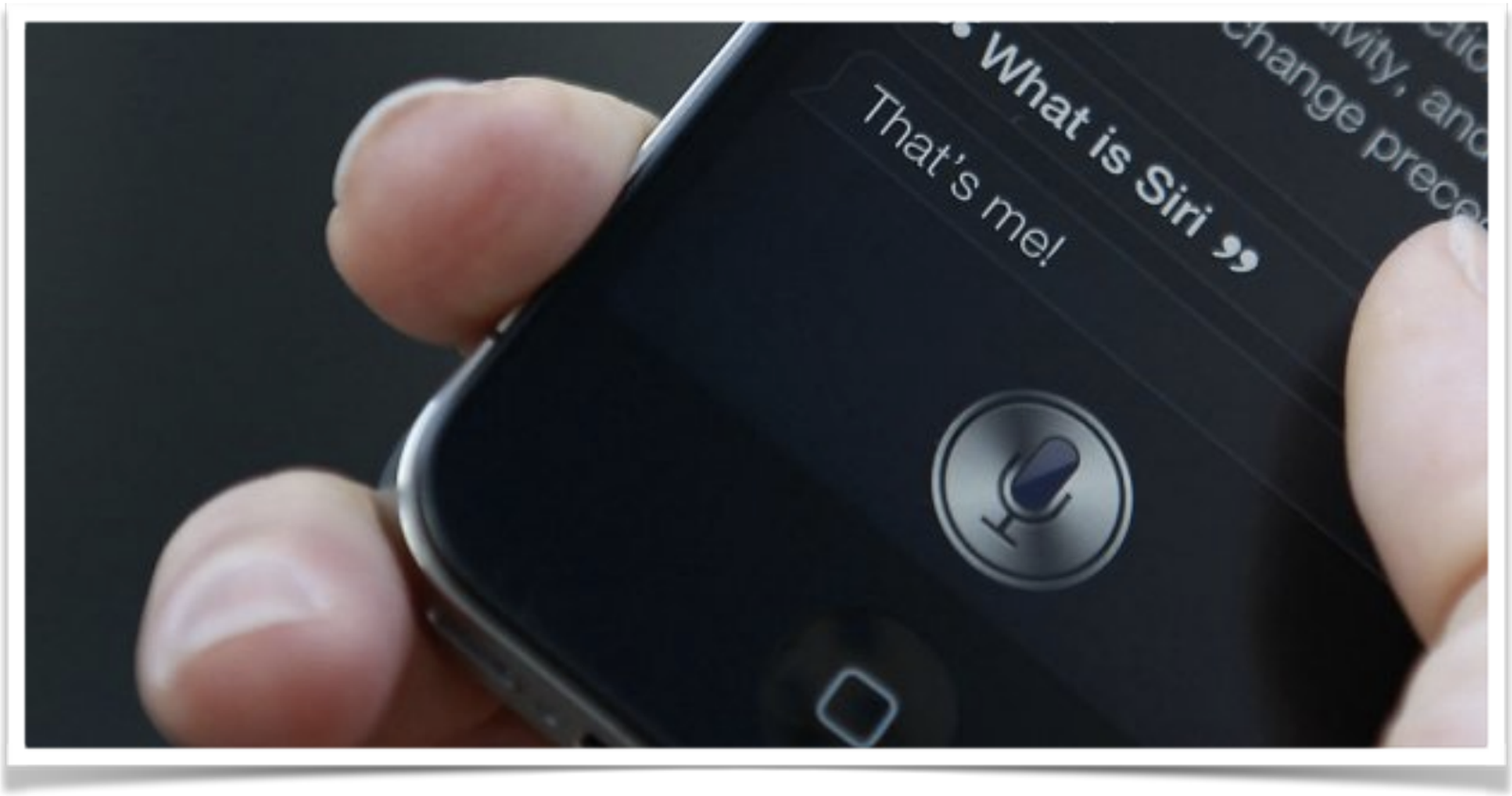
If you mix all these ingredients you can…



If you mix all these ingredients you can…



If you mix all these ingredients you can…



If you mix all these ingredients you can…