Institut Pasteur | C3BI

Introduction to deep learning

Christophe Zimmer

Imaging & Modeling Unit

Scope of this talk

- basic principles and terminology of deep learning
- supervised learning & classification
- convnets
- software tools

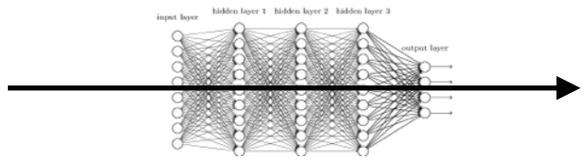
What deep learning can do

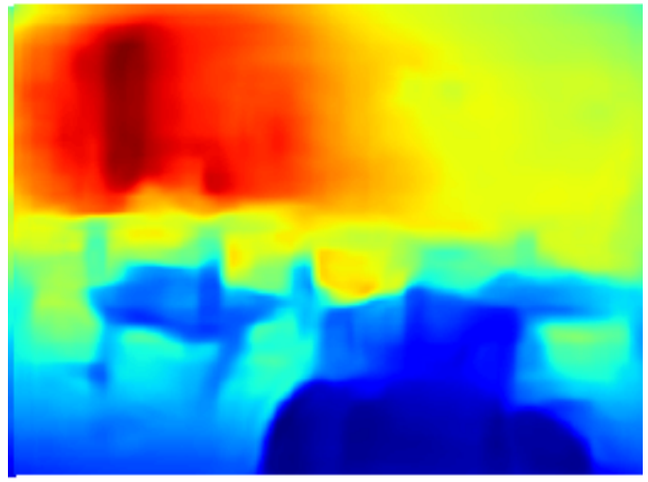
Image interpretation

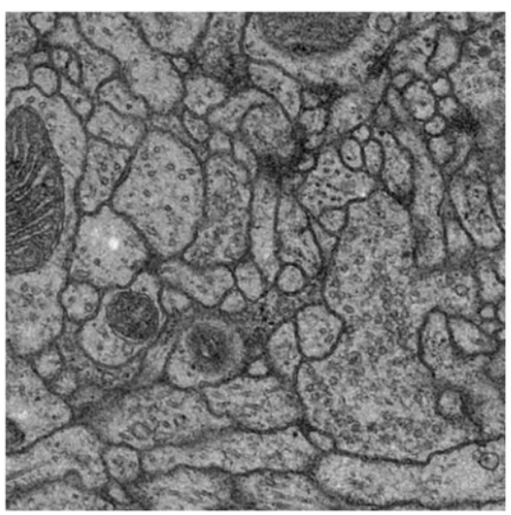
input

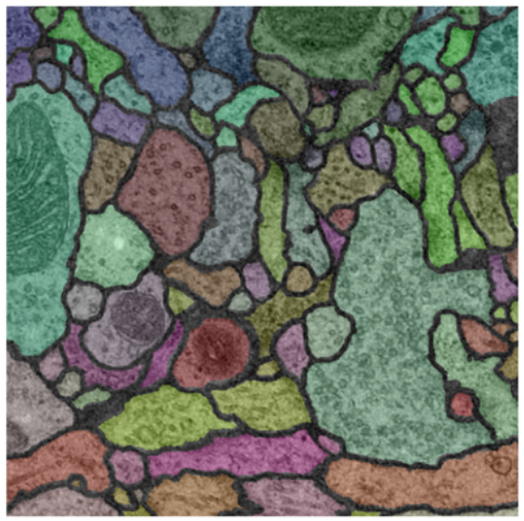
output

“A group of people sitting on a boat in the water.”







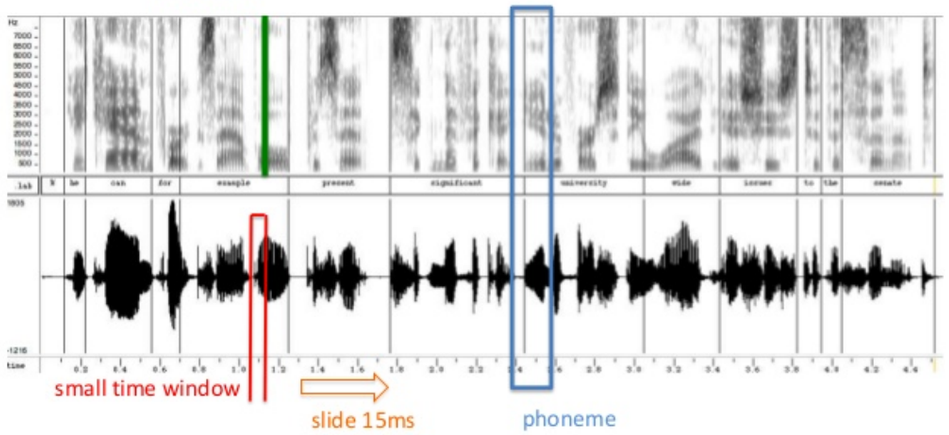


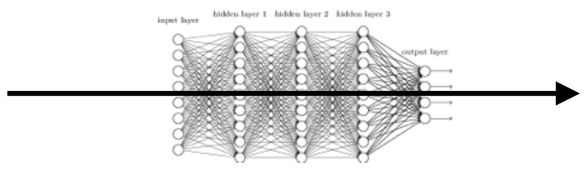
LeCun *et al Nature* 2015; Eigen *et al* 2014; Ciresan *et al* 2012

Speech recognition

input

output



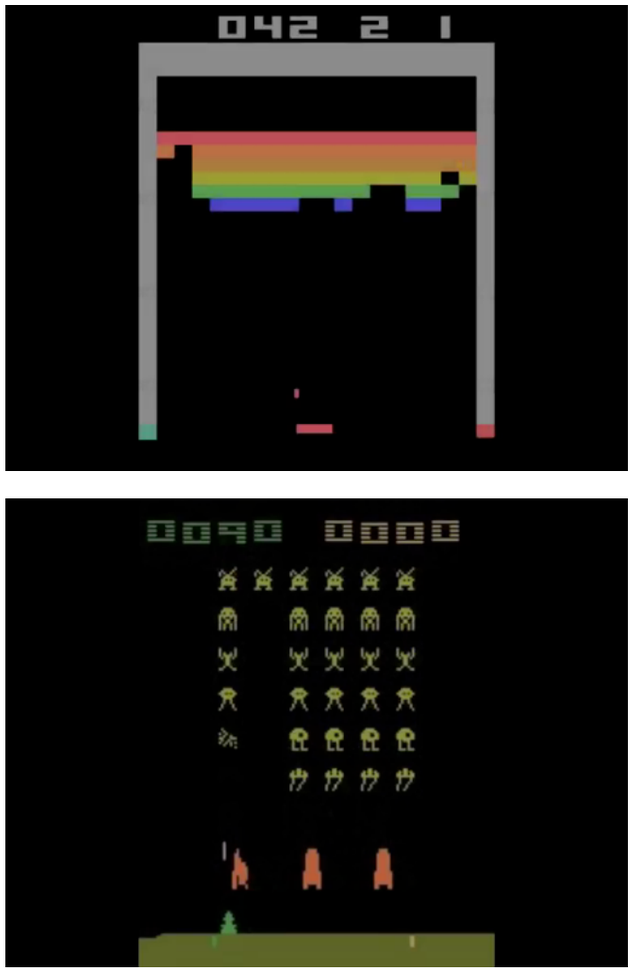


“He can for example present significant university wide issues to the senate”

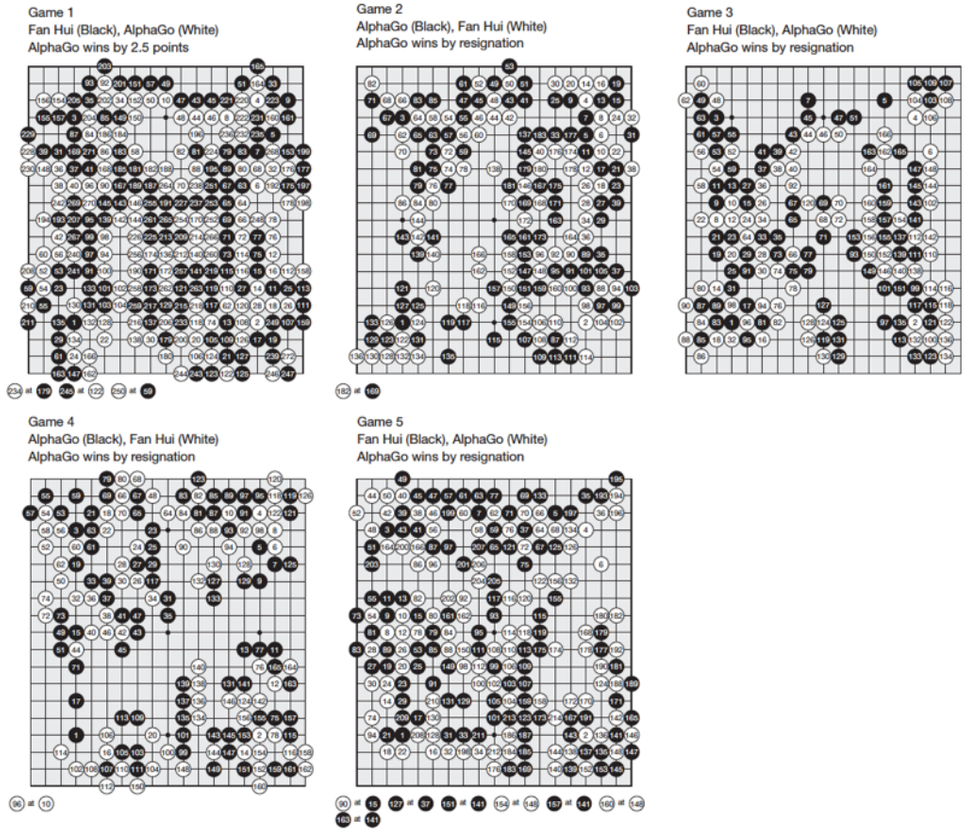
Games

deep learning + reinforcement learning (+ Monte Carlo tree search)

human/superhuman performance for 29/49 Atari games



AlphaGo vs Fan Hui: 5-0
(October 2015)



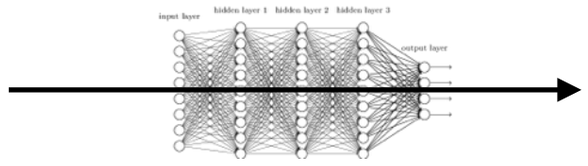
AlphaGo vs Lee Sedol: 4-1
(March 2016)

Mnih *et al Nature* 2015

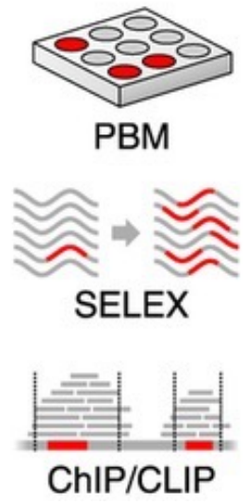
Silver *et al Nature* 2016

Regulatory genomics

input



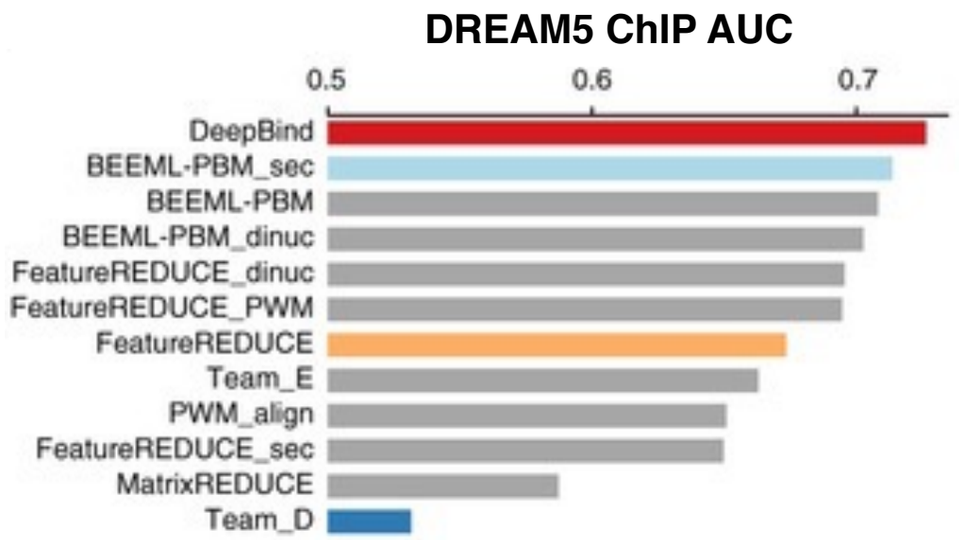
output



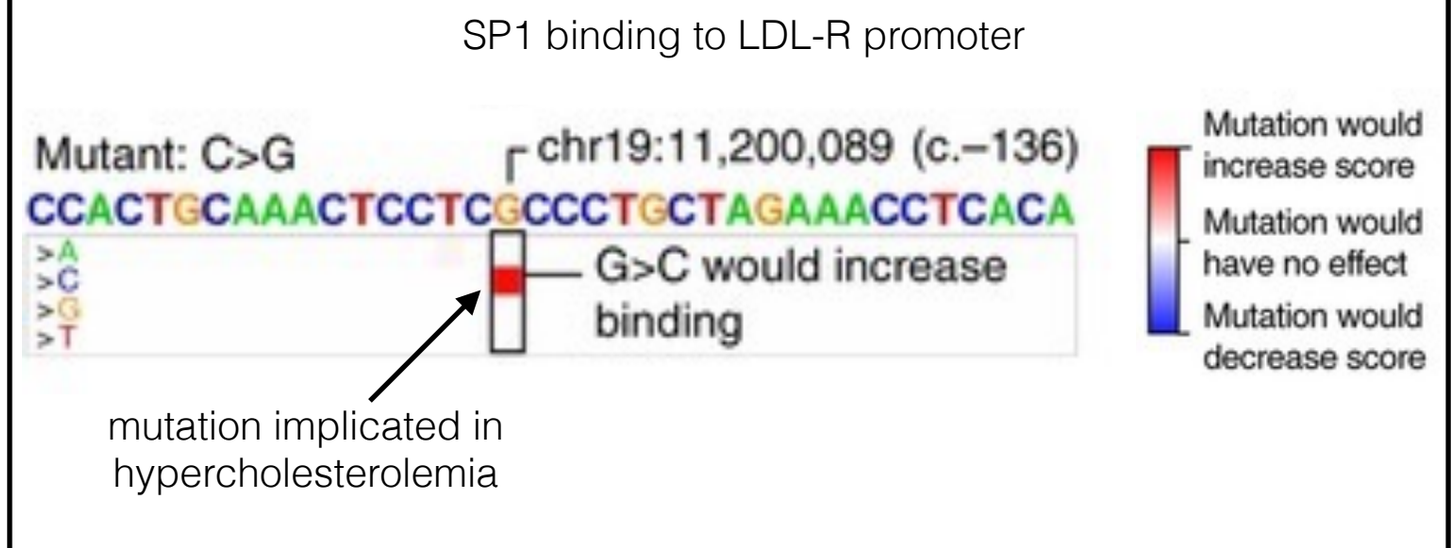
DNA-protein binding scores

- trained on 12 Tb of sequencing data from thousands of experiments

outperforms state-of-the art methods

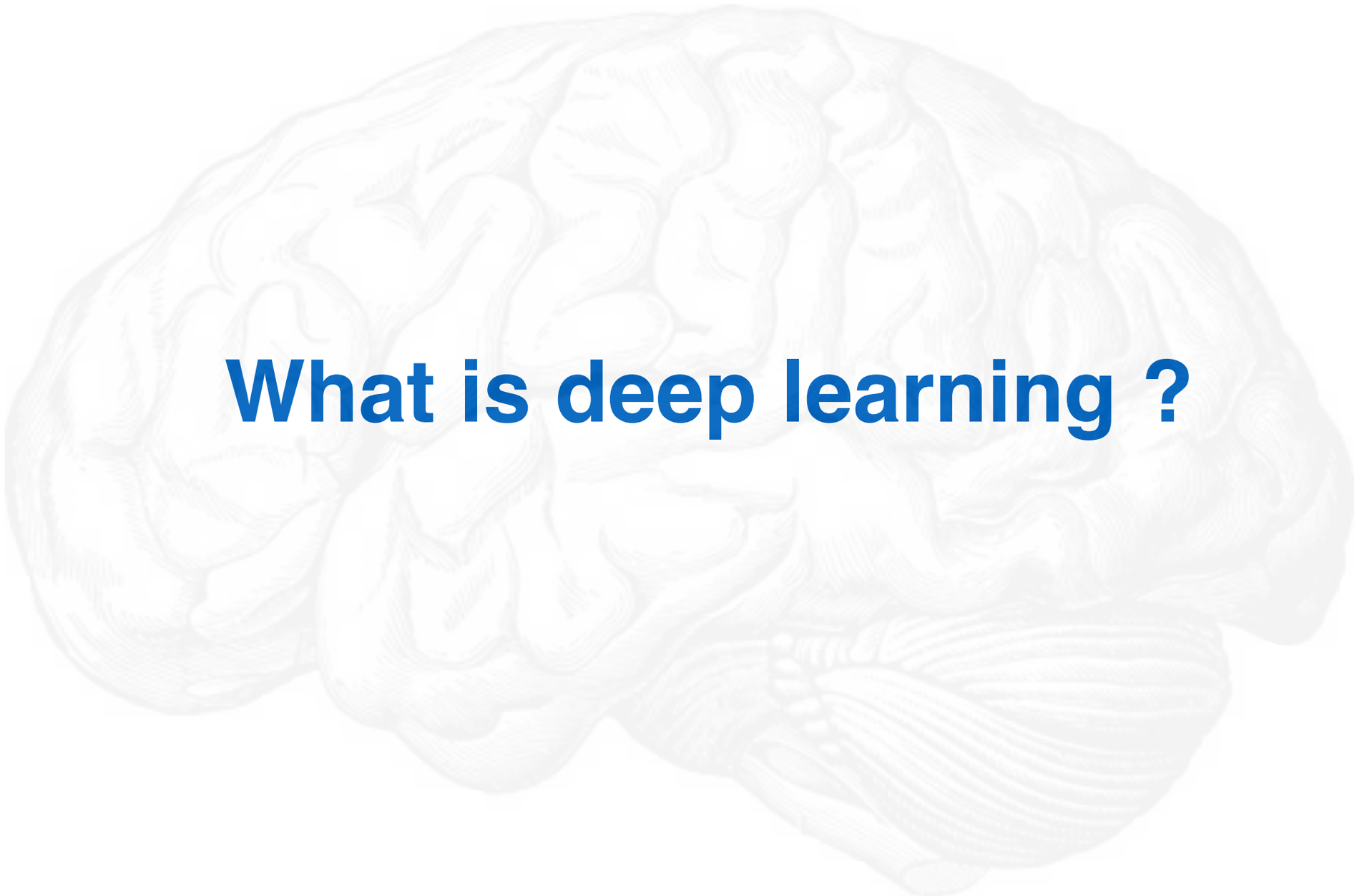


can predict damaging genetic variants



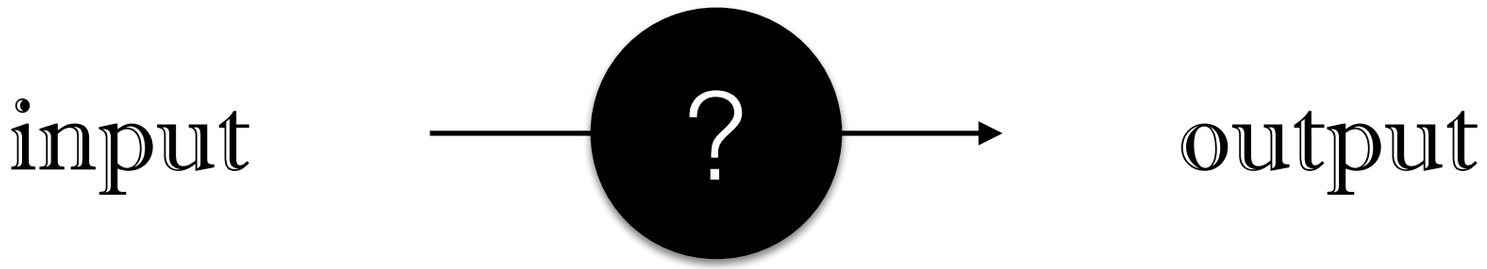
- can make predictions for mutations never seen before

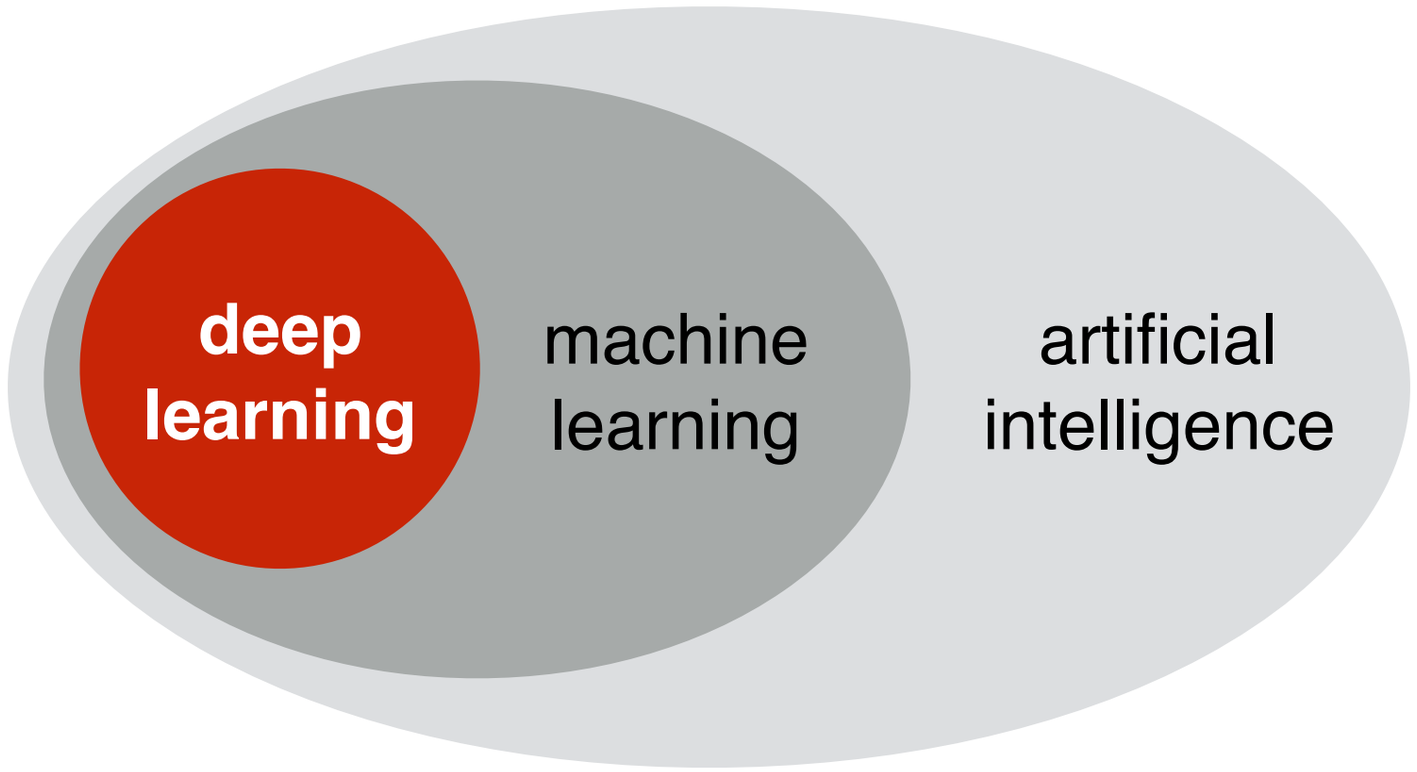
Alipanahi *et al Nat Biotech* 2015



What is deep learning ?

deep learning : a family of computational techniques using (deep) artificial neural networks to learn complex input-output relations between numerical data




deep
learning
machine
learning
artificial
intelligence

Why deep learning works *now*

- Main ideas and algorithms date back to the 80-90s (Hinton, Fukushima, LeCun, Bengio, Schmidhuber..)
- Neural nets achieved only limited success
- What changed in recent years:

1. **Faster processors**

Graphical processing Unit (GPU)

2. **Lots of data**

- YouTube: >100 h of video uploaded per minute
- Facebook: >300 M images uploaded per day

3. Improved algorithms

Why deep learning is easy

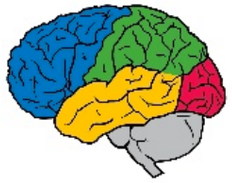
theano

Université de Montréal

Python, C

TensorFlow™

Google



Python, C++

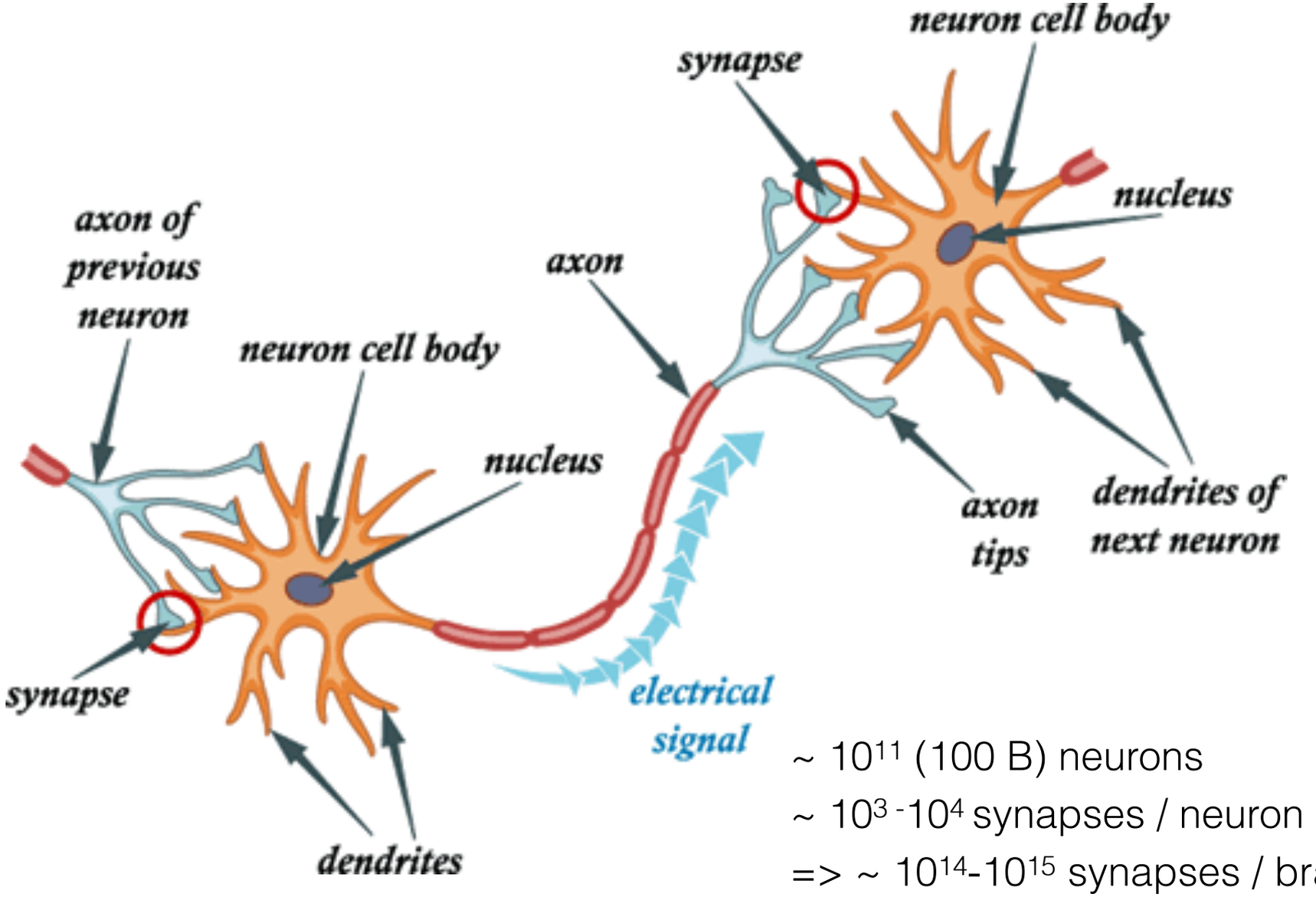
```
chz$
chz$
chz$ pip install python numpy matplotlib keras jupyter
chz$
chz$ ipython notebook
chz$
```

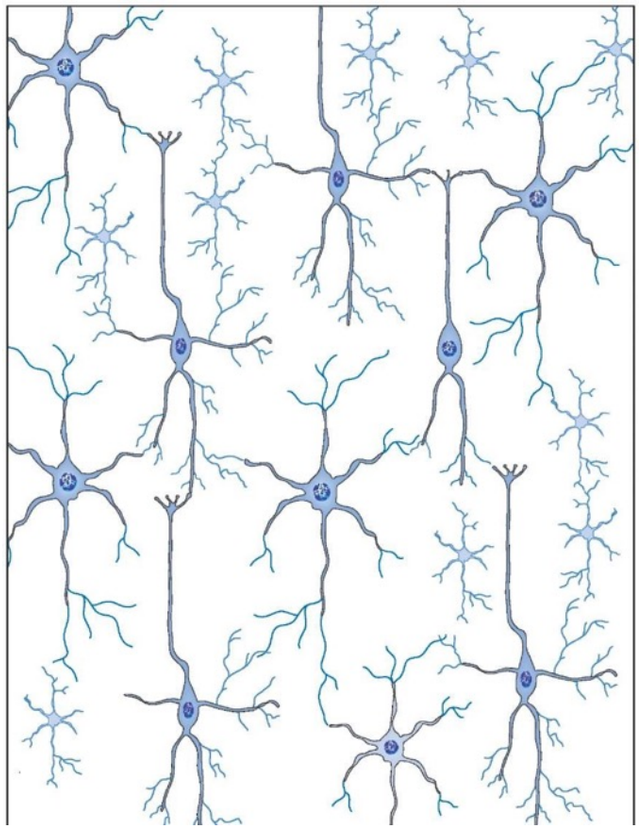
install tensorflow or theano
(instructions on their websites)

How deep learning works

- **forward propagation**

Neurons in the human brain

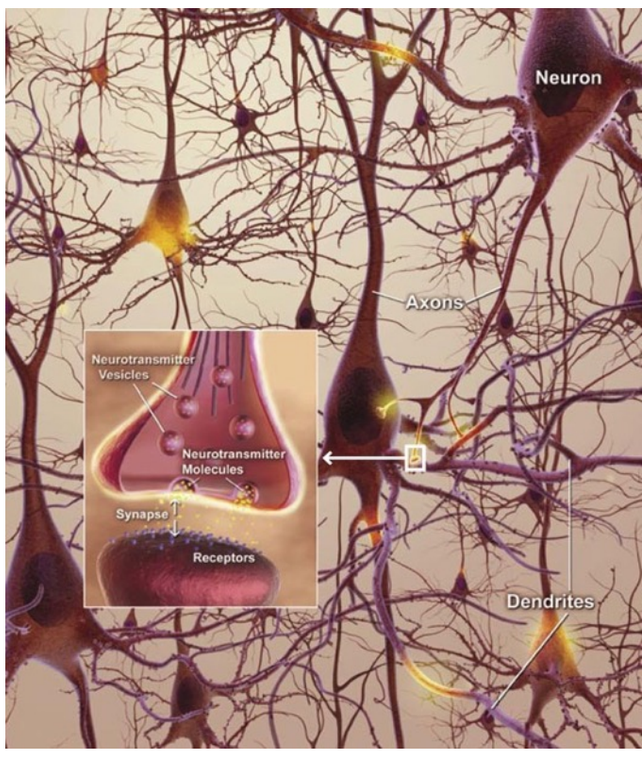




~ 10^{11} (100 B) neurons

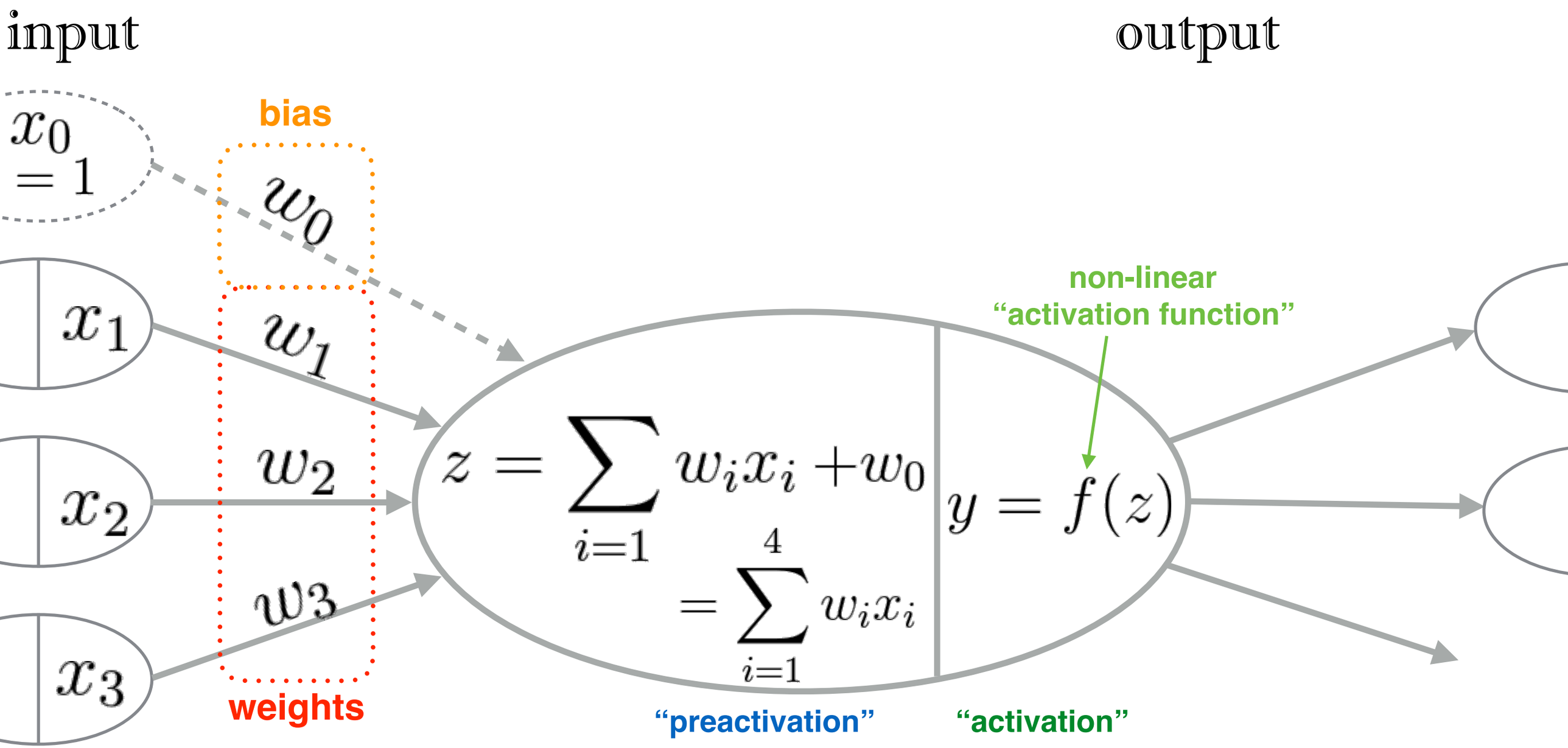
~ 10^{3} - 10^{4} synapses / neuron

=> ~ 10^{14}-10^{15} synapses / brain



- dendrites collect electrical input signals
- cell body integrates them and generates output signal
- axons carry electrical outputs to dendrites of other neurons

A single computational unit (neuron) in a neural net



a “**feature**”

weights w = synaptic strengths

w>0 : excitatory

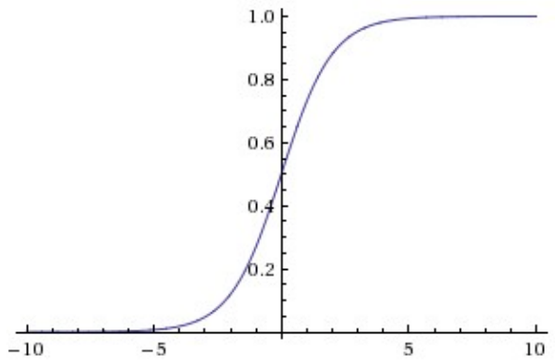
w<0 : inhibitory

weights & biases will be *learned* (see later)

Activation functions

Sigmoid:

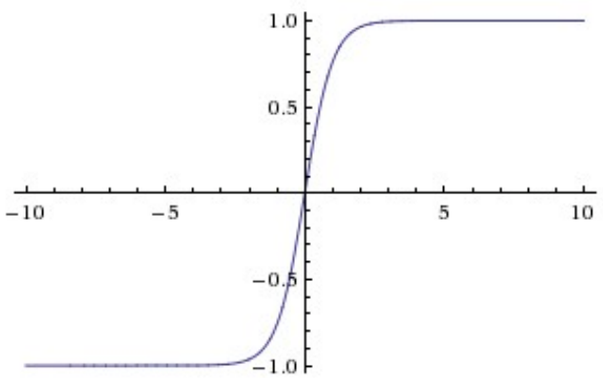
$$f(z) = \frac{1}{1 + e^{-z}}$$



- good biological interpretation
- Drawback: saturation => kills gradient & learning
- fallen out of favor

Tanh

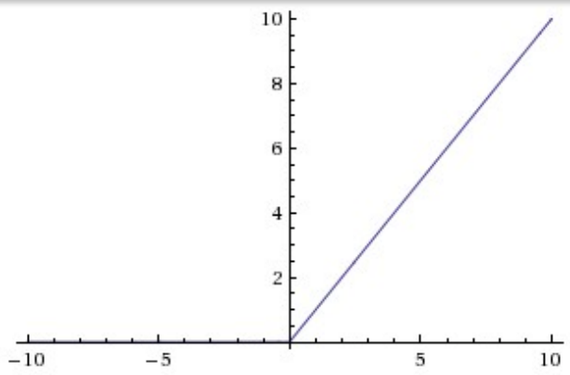
$$f(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$



- now always preferred to sigmoid

ReLU

f(x) = x if x>0
f(x) = 0 if x<=0



- accelerates learning
- computationally simple (thresholding)
- drawback: neuron can “die” during training (activation irreversibly vanishes for all training data)

Leaky ReLU

f(x) = x if x>0
f(x) = ax if x<=0 (a<<1)

- can sometimes limit dying ReLU problem
- a is a hyperparameter that can be learned

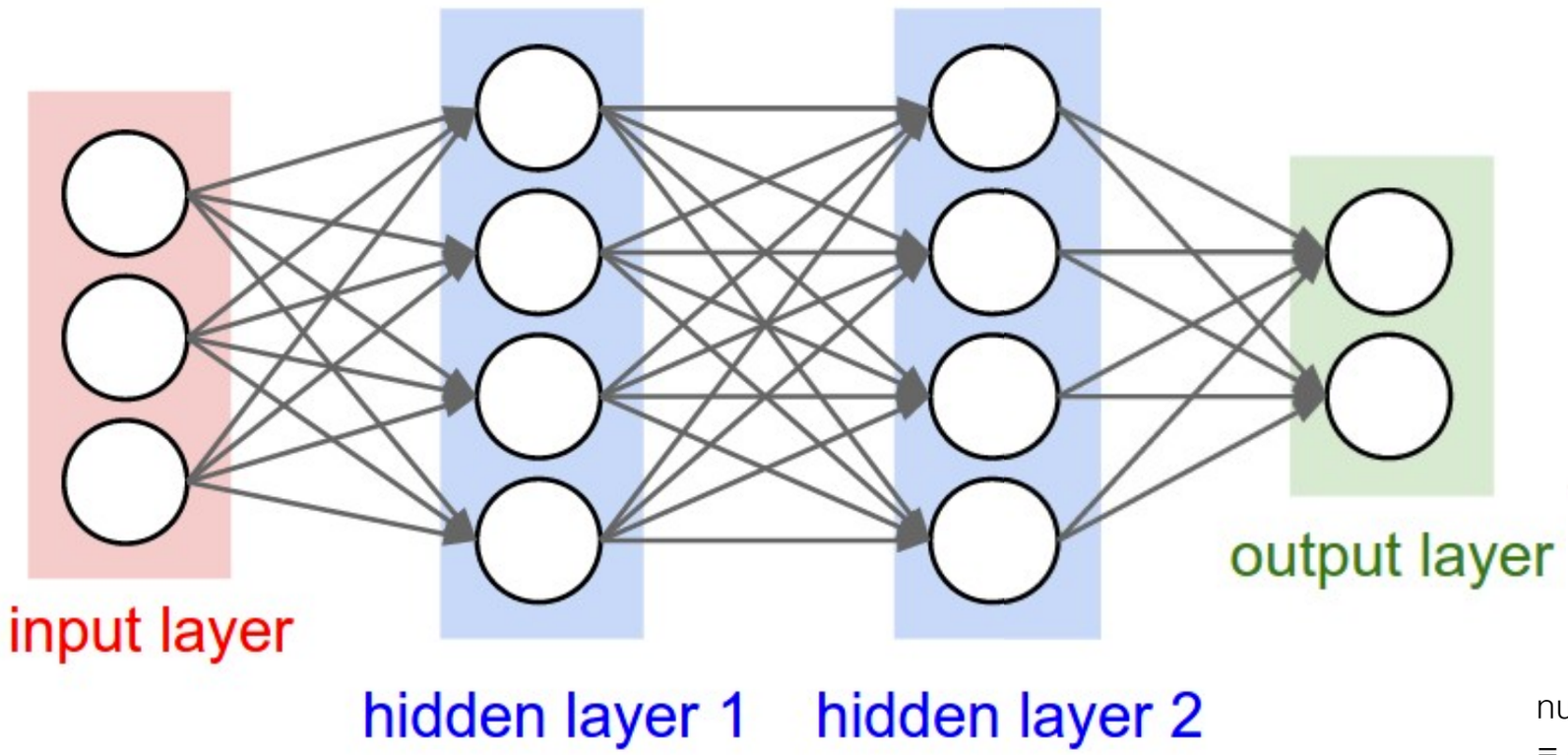
from A. Karpathy, Stanford

Neural nets

Neural net = acyclic, directed, graph
- nodes = units (neurons)
- arrows = connections (synapses)

Usually organized in layers:
- connections only between consecutive layers
- no connections within layers
- bottom layer (leftmost) = input
- top layer (rightmost) = output

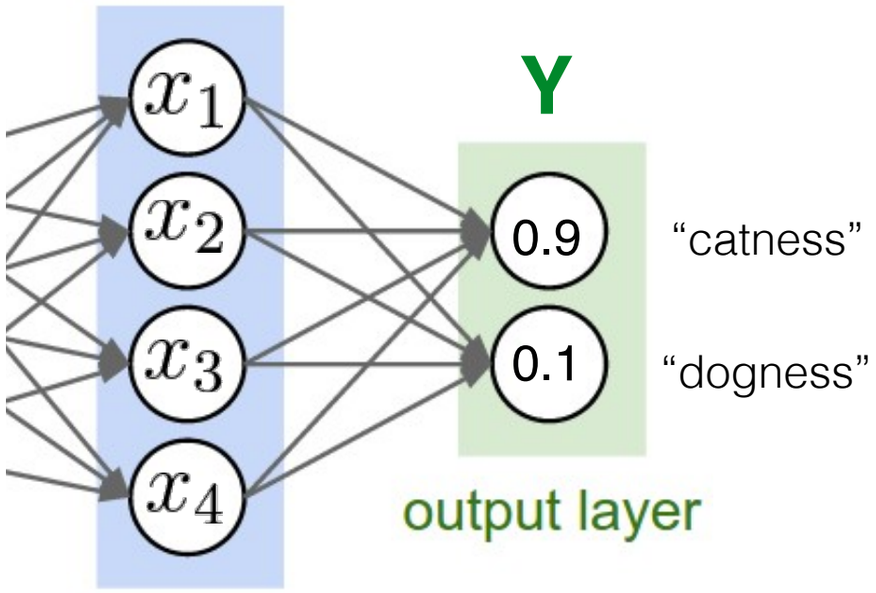
Fully connected (“dense”) layer:
each neuron is connected to all neurons of the adjacent layer.



A fully connected 3-layer neural net with 3 inputs, 2 hidden layers of 4 neurons each and an output layer of 2 neurons

number of neurons
= 4+4+2 = 10
number of weights
= 3*4+4*4+4*2 = 36
number of parameters
= 36 weights + 10 biases = 46

Softmax layer for classification



preactivations **activations**

$$z_1 = \sum_{i=1}^{4} w_{1,i} x_i \qquad y_1 = \frac{e^{z_1}}{e^{z_1} + e^{z_2}}$$

$$z_2 = \sum_{i=1}^{4} w_{2,i} x_i \qquad y_2 = \frac{e^{z_2}}{e^{z_1} + e^{z_2}}$$

Softmax for K classes:

$$y_i = \sigma(z_1, ..z_K) = \frac{e^{z_i}}{\sum_{k=1}^{K} e^{z_k}}$$

ensures probabilities: $y_i \in [0, 1]$ $\sum_{i=1}^{K} y_i = 1$

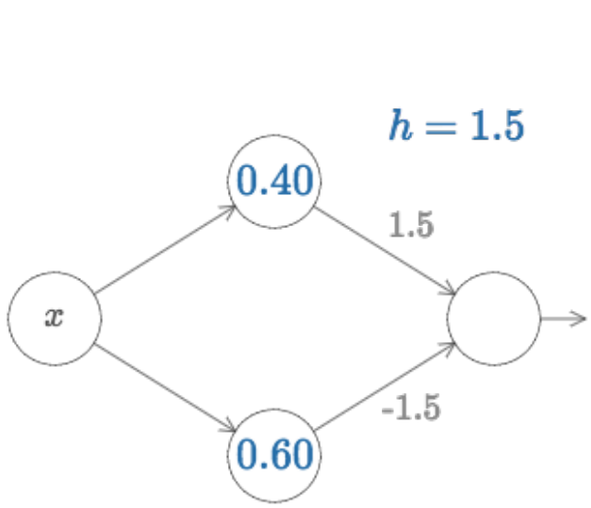
- Predicted class = max probability
- For regression (=predicting continuous variables), use other functions (e.g. y=z)

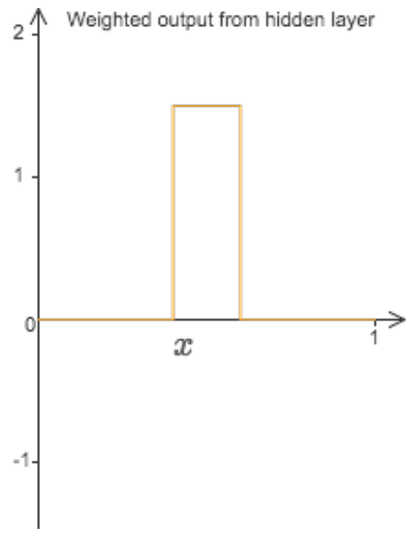
NNs are universal function approximators

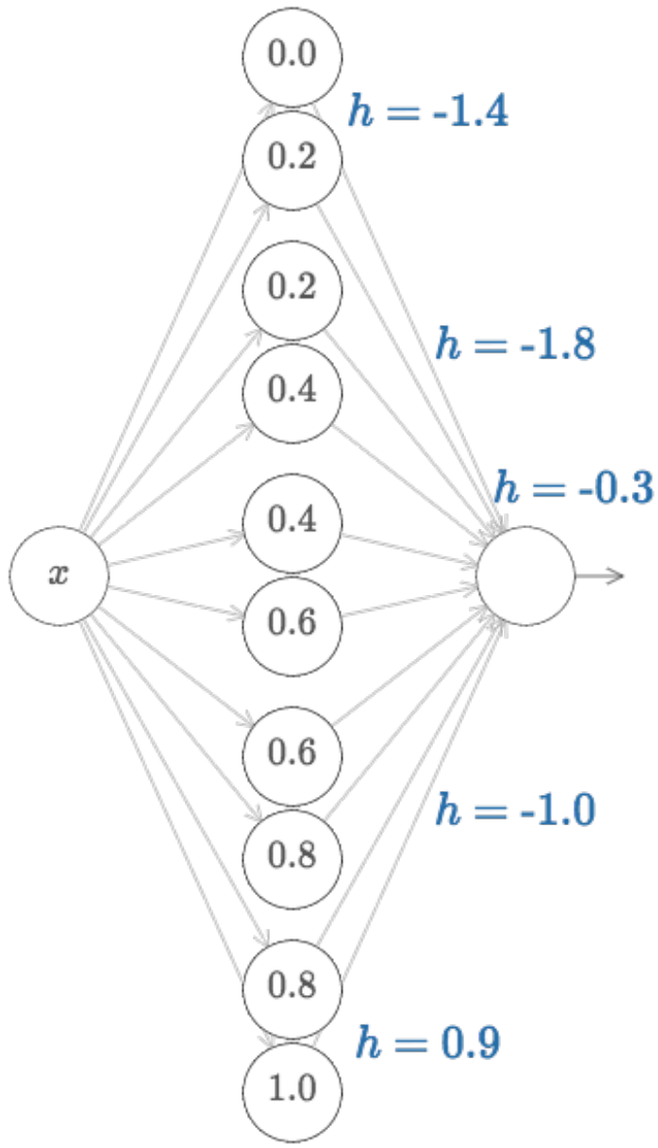
For any continuous function $\mathcal{F}$ and tolerance ϵ , there exists a NN that approximates it within the tolerance epsilon

(even with a single hidden layer)

(true regardless of the number of input and output values)







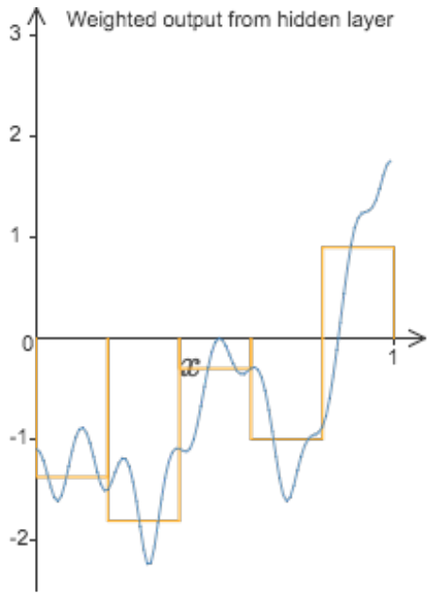
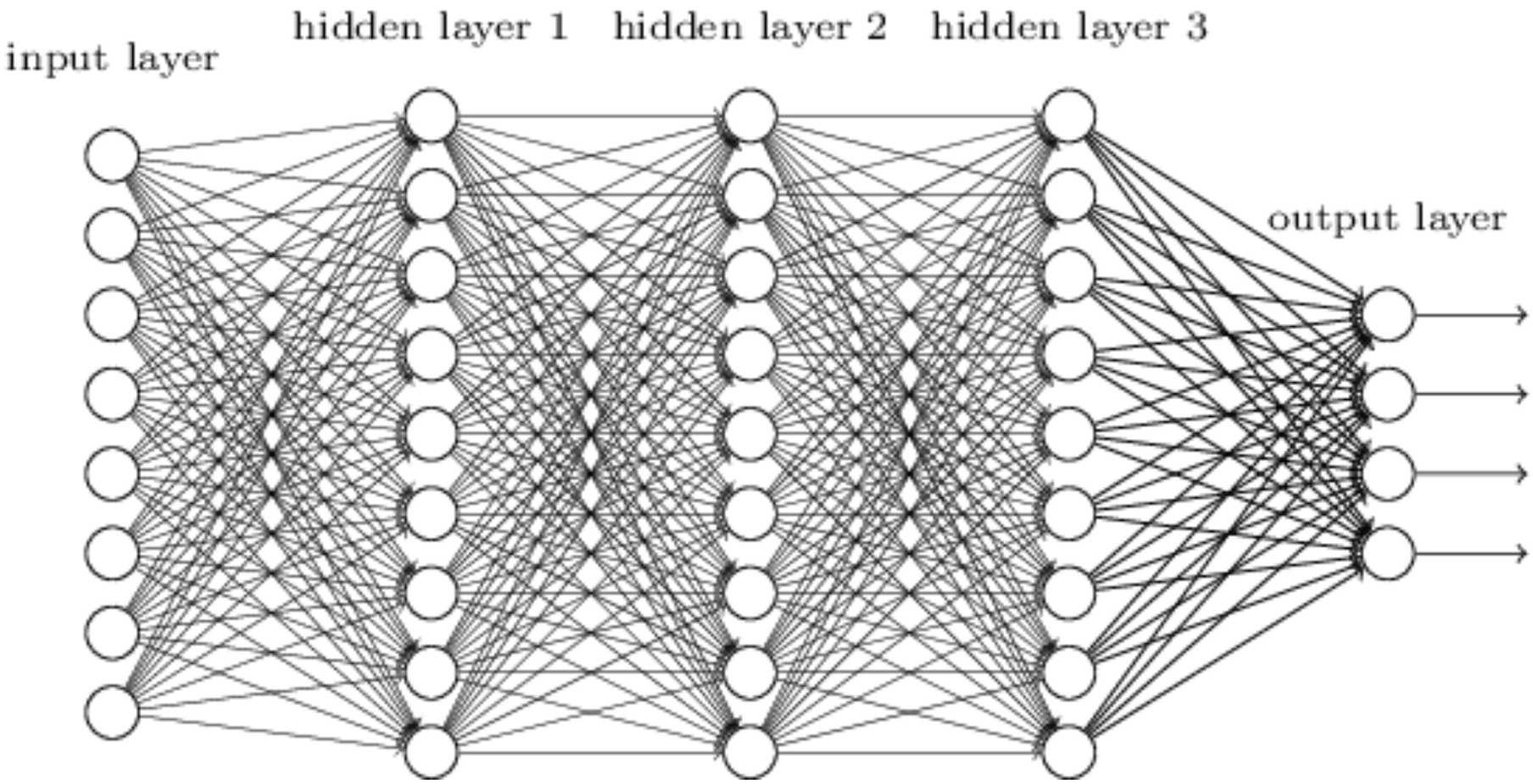


Figure credit: M. Nielsen

but it's much better to use multiple layers (next)

Deep learning uses multiple hidden layers

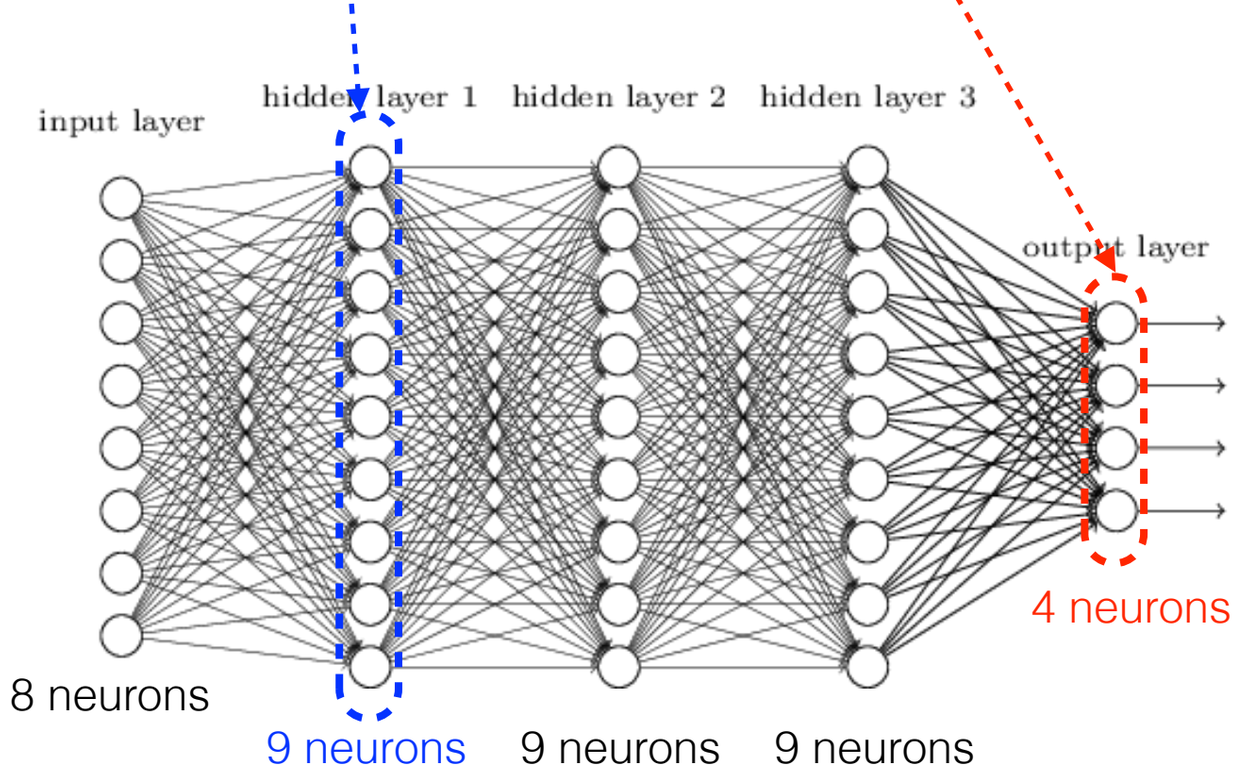


A fully connected net with 3 hidden layers

Figure credit: M. Nielsen

How to build a neural net in 7 lines

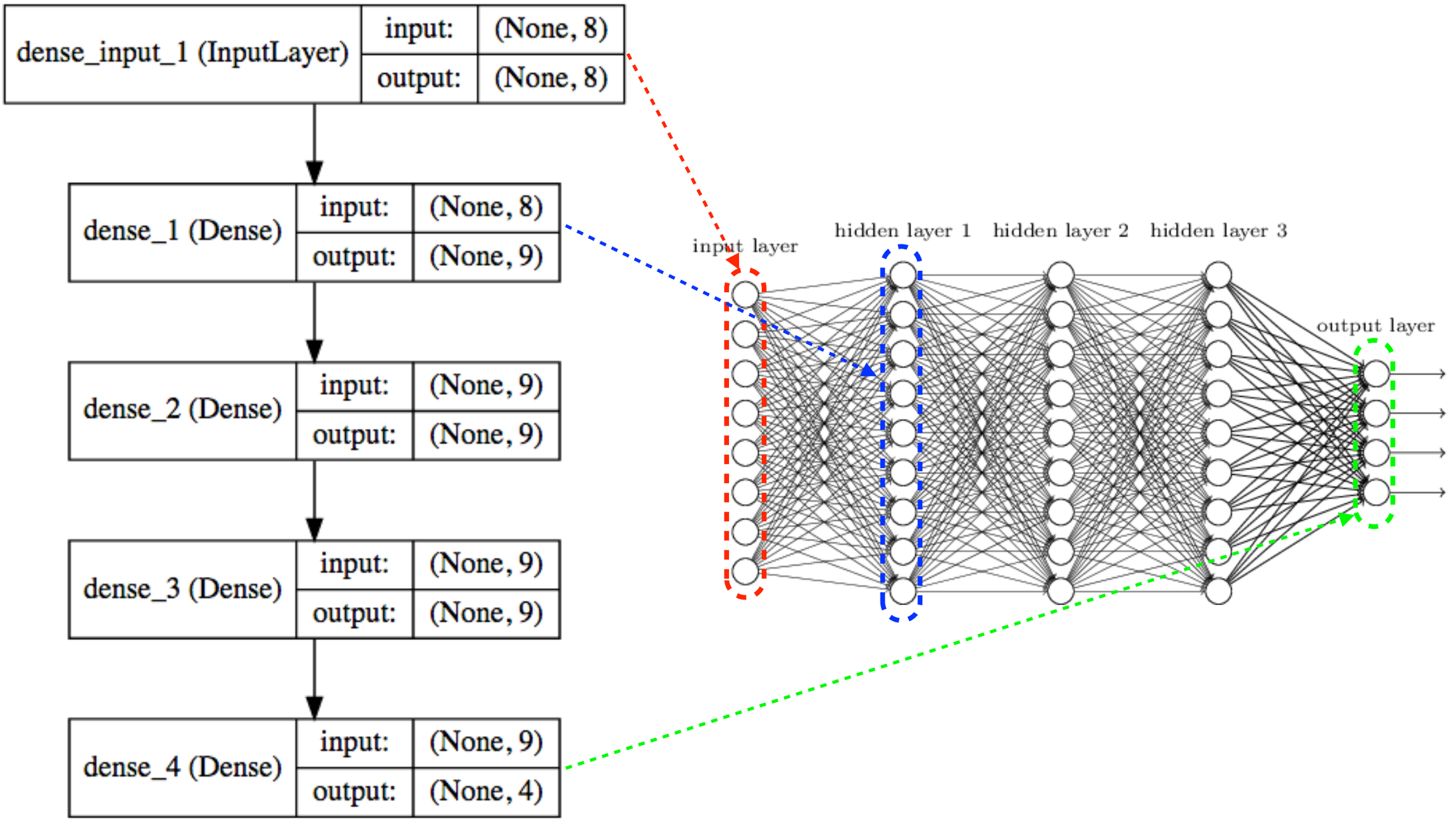
```
from keras.models import Sequential
from keras.layers import Dense, Activation

model = Sequential()
model.add(Dense(9, input_dim=8, activation='relu'))
model.add(Dense(9, activation='relu'))
model.add(Dense(9, activation='relu'))
model.add(Dense(4, activation='softmax'))
```



How to visualize a neural net in 3 lines

```
In [8]: from IPython.display import SVG
        from keras.utils.visualize_util import model_to_dot

        SVG(model_to_dot(model, show_shapes=True).create(prog='dot', format='svg'))
```

Out[8]:

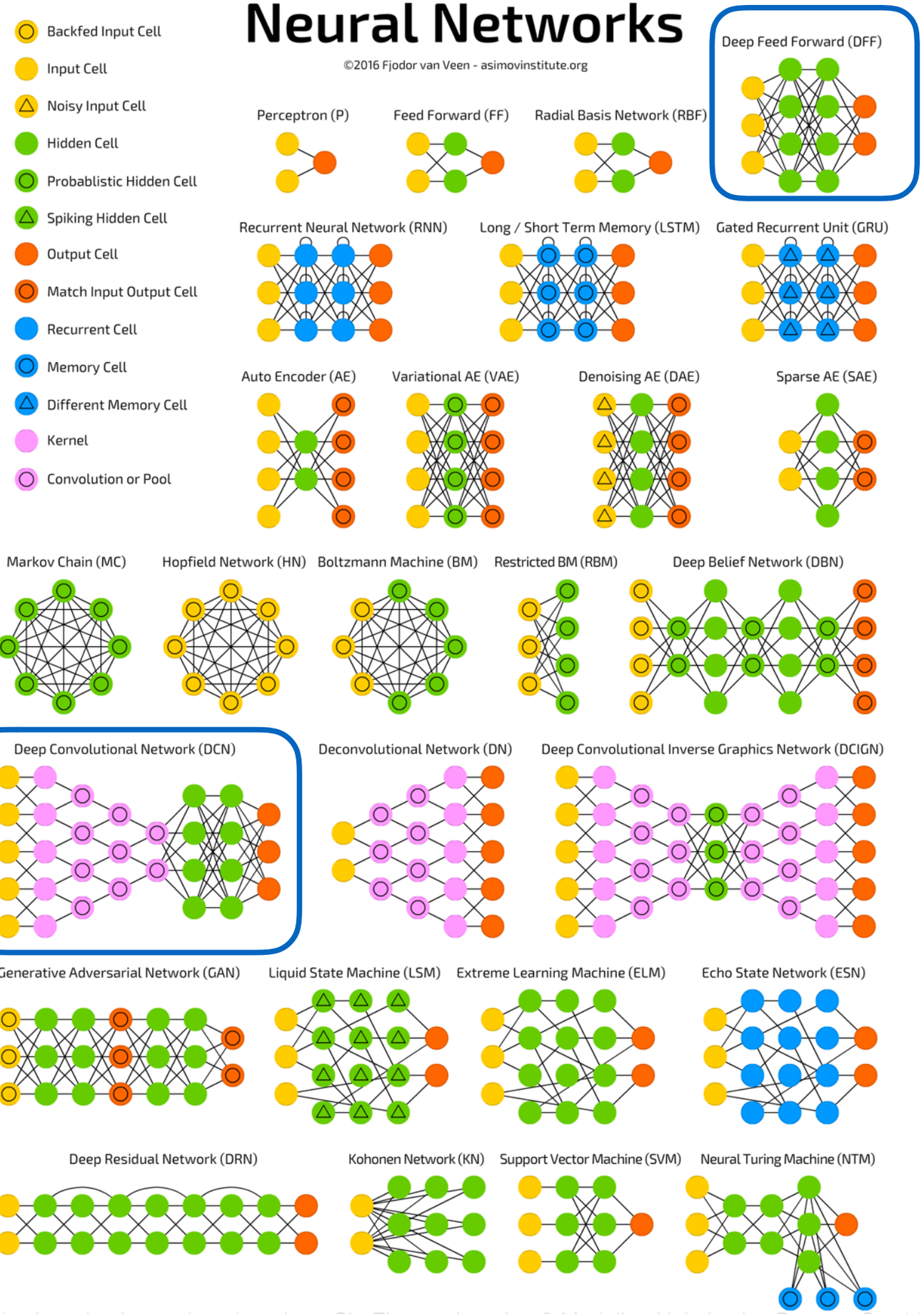


How to choose a neural net architecture ?

- use pre-existing successful architecture
- improve using random search and validation (see later)
- current trend is to go deep and slim

A mostly complete chart of

Neural Networks



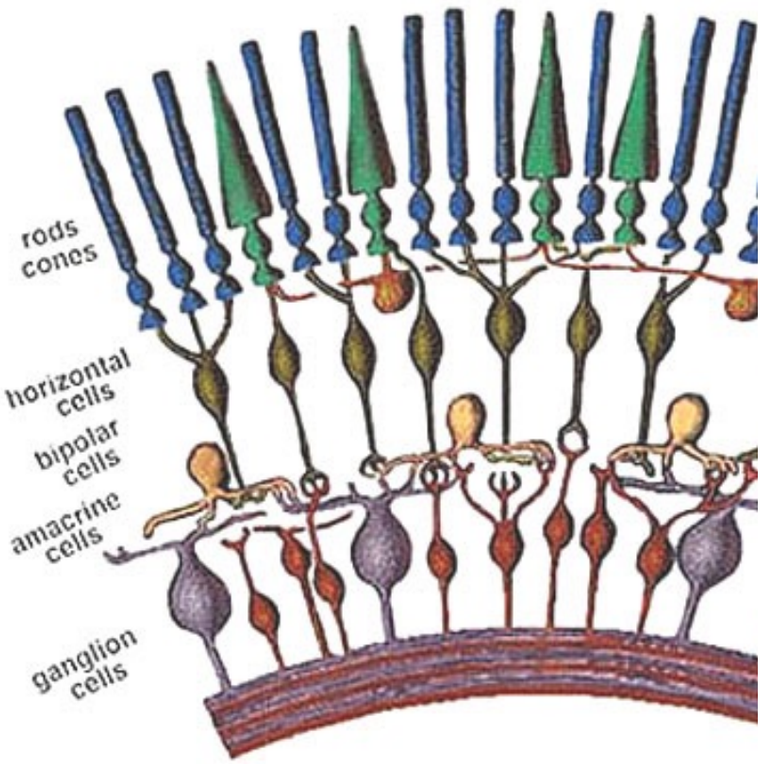
slide from Asimov Institute

Convolutional neural nets

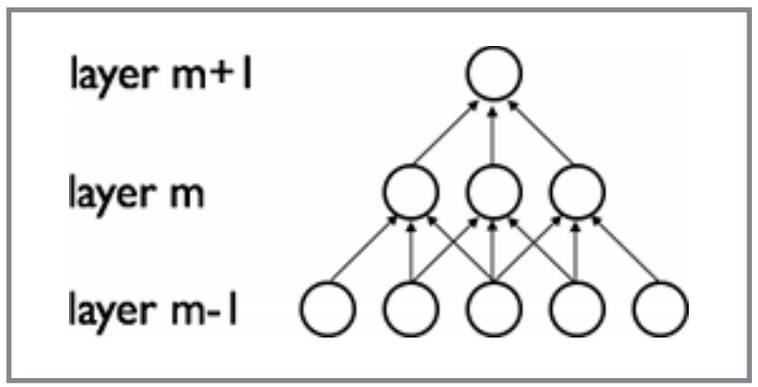
(aka convnets or CNNs)

Convnets

Special type of neural nets particularly adapted to images

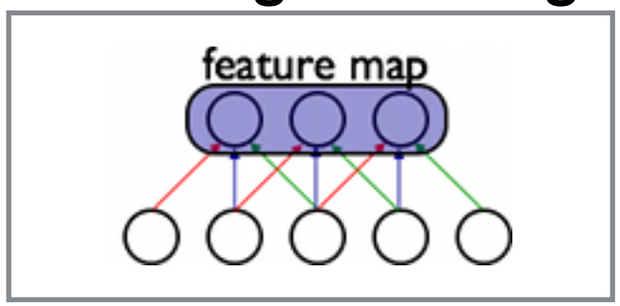


1. sparse connectivity



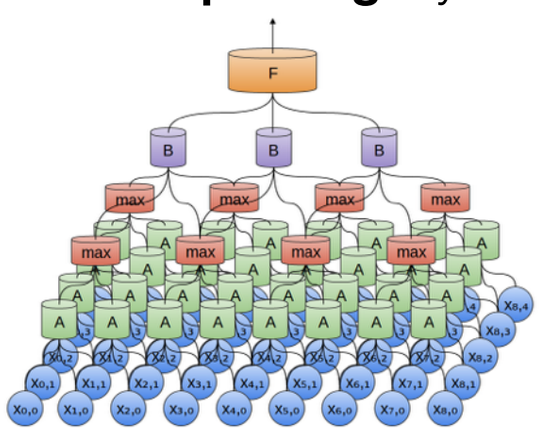
=> local receptive fields

2. weight sharing

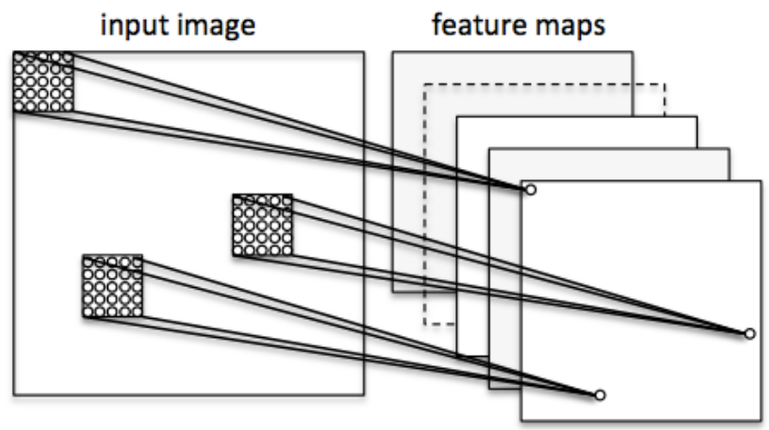


=> translational invariance

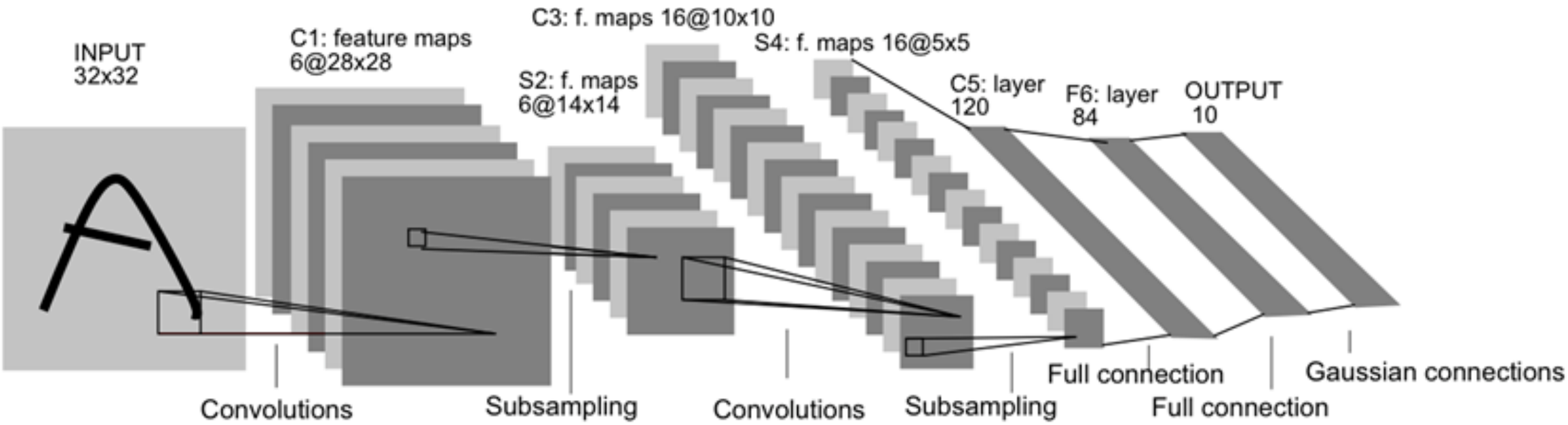
Convolutional layers are often interspersed by **max-pooling** layers



Multiple "**feature maps**" per layer



An early convnet



LeCun et al. 1989

used to recognize hand-written digits (MNIST)

→ 9

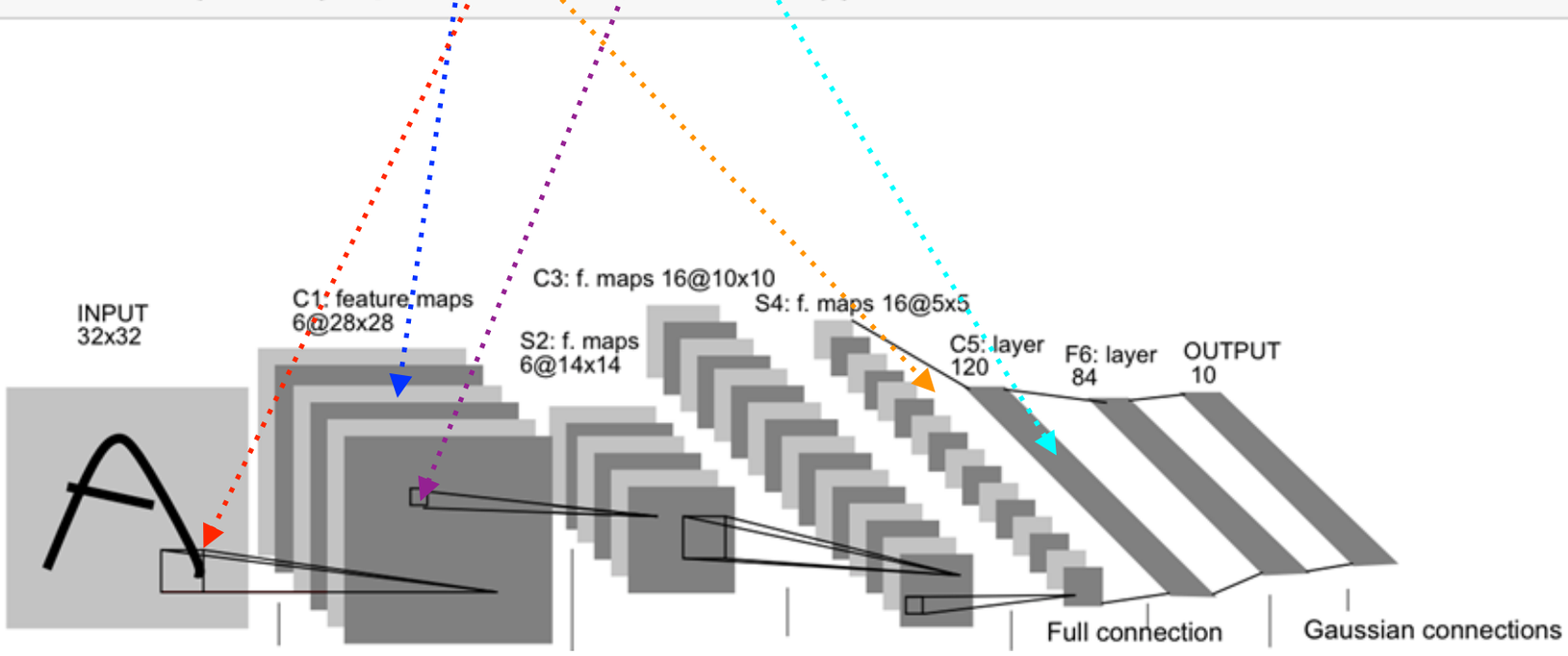
Building a convnet

number of feature maps

5x5 convolution filter

```
model = Sequential()
model.add(Convolution2D(6, 5, 5, input_shape=(1, 28, 28), activation='relu'))
model.add(MaxPooling2D(pool_size=(2,2)))
model.add(Convolution2D(16, 5, 5, activation='relu'))
model.add(MaxPooling2D(pool_size=(2,2)))
model.add(Flatten())
model.add(Dense(120, activation='relu'))
model.add(Dense(84, activation='relu'))
model.add(Dense(10, activation='softmax'))
```

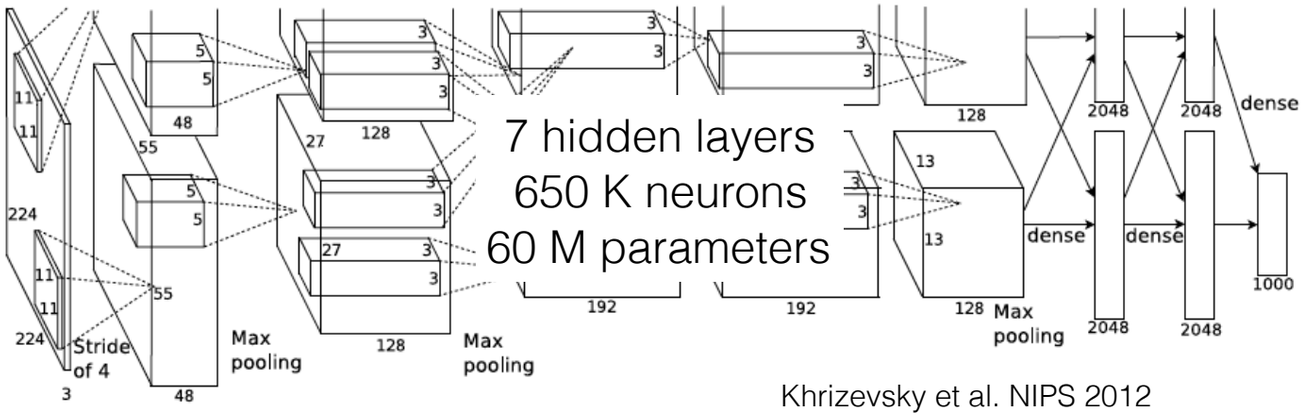
max pooling
pool size 2x2



convolution2d_input_1 (InputLayer)	input:	(None, 1, 28, 28)
	output:	(None, 1, 28, 28)
convolution2d_1 (Convolution2D)	input:	(None, 1, 28, 28)
	output:	(None, 6, 26, 26)
maxpooling2d_1 (MaxPooling2D)	input:	(None, 6, 26, 26)
	output:	(None, 6, 13, 13)
convolution2d_2 (Convolution2D)	input:	(None, 6, 13, 13)
	output:	(None, 16, 11, 11)
maxpooling2d_2 (MaxPooling2D)	input:	(None, 16, 11, 11)
	output:	(None, 16, 5, 5)
flatten_1 (Flatten)	input:	(None, 16, 5, 5)
	output:	(None, 400)
dense_5 (Dense)	input:	(None, 400)
	output:	(None, 120)
dense_6 (Dense)	input:	(None, 120)
	output:	(None, 84)
dense_7 (Dense)	input:	(None, 84)
	output:	(None, 10)

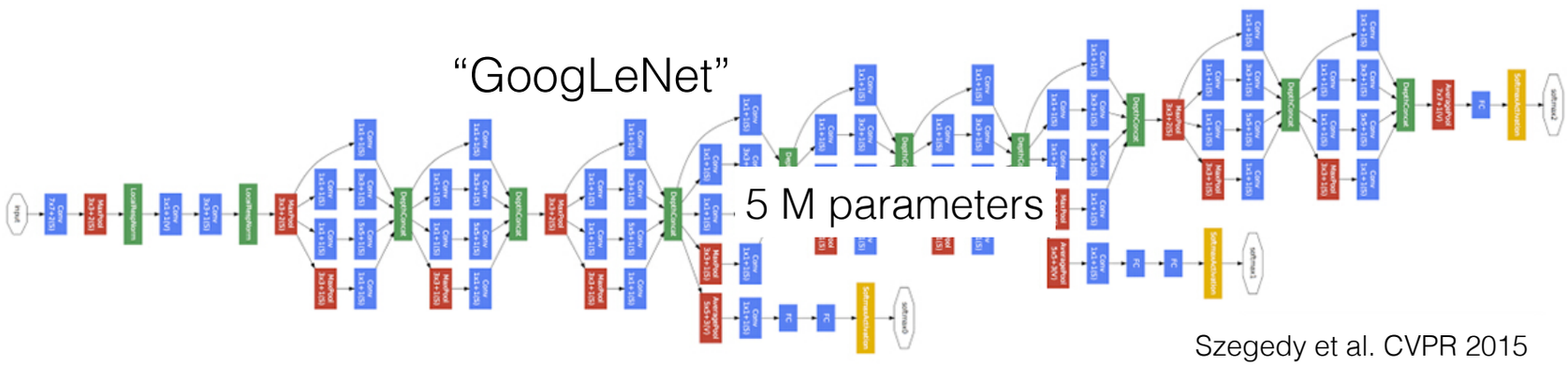
More recent convnets

"AlexNet"



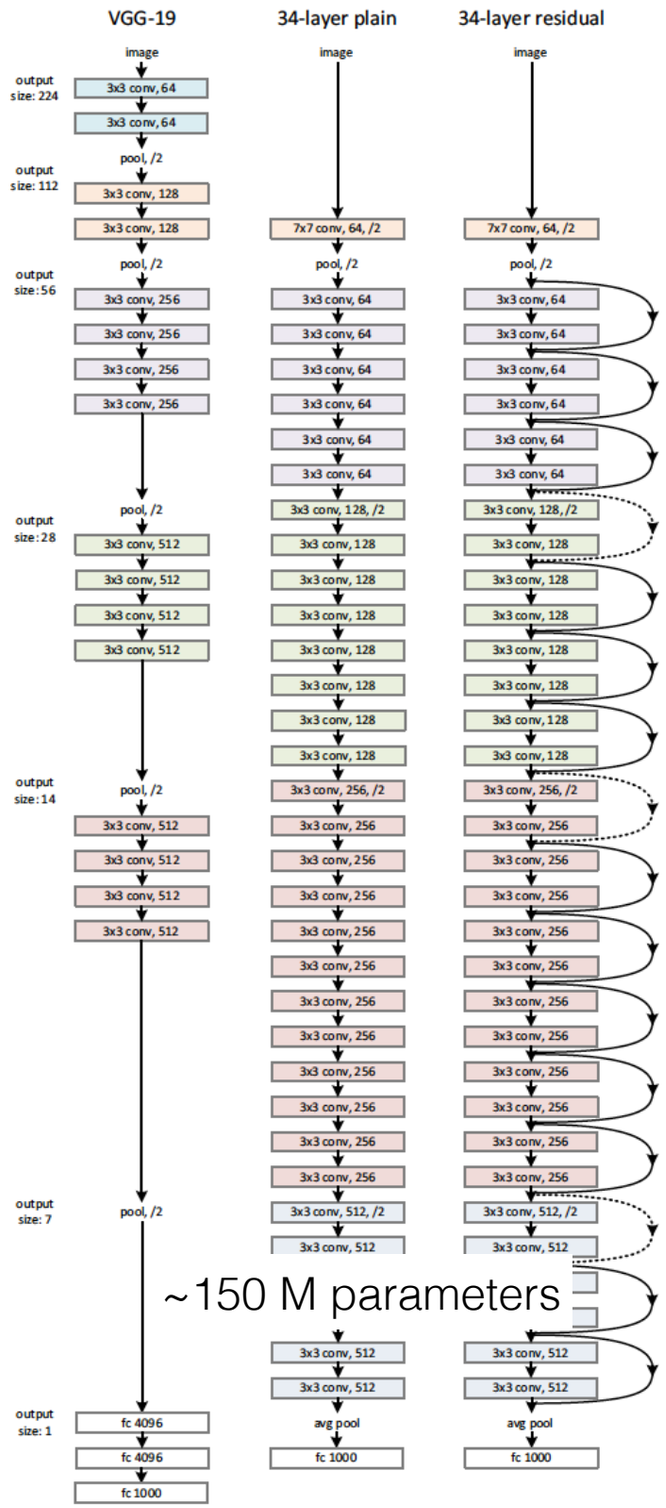
Khrizevsky et al. NIPS 2012

"GoogLeNet"



Szegedy et al. CVPR 2015

"VGGNet"



Simonyan & Zisserman ICLR 2015

- Trend towards deeper networks without increasing the number of parameters
- Training deeper nets is harder
- But lots of trained models can be downloaded

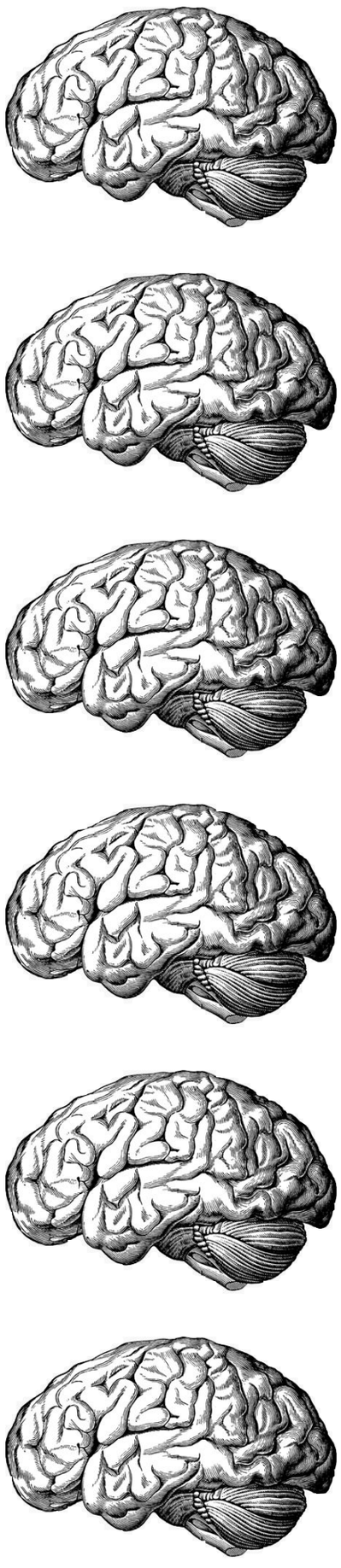
so far we assumed that the parameters (weights + biases) were *given*

but these parameters must be *learned*

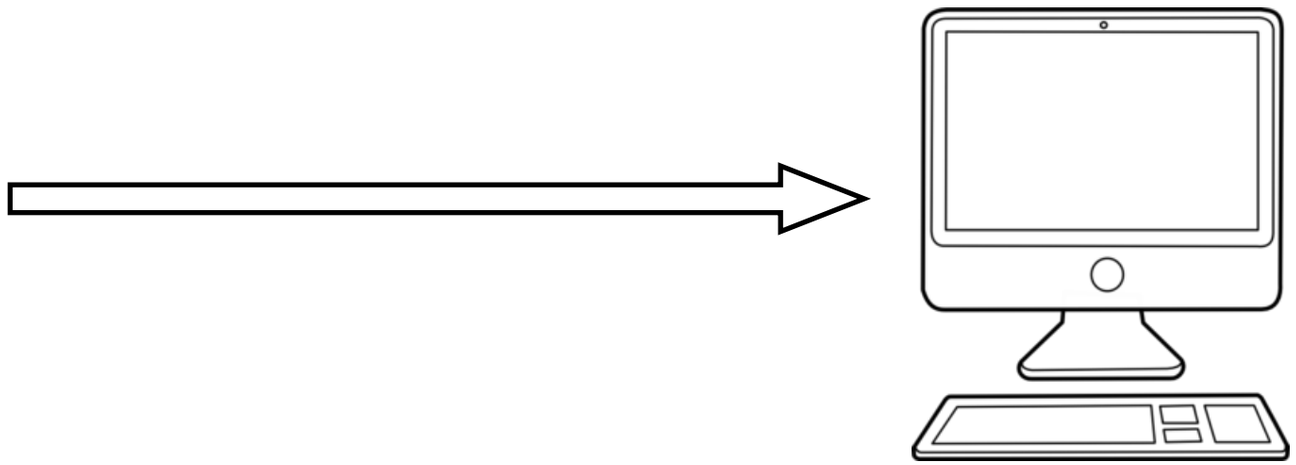
How deep learning works

- forward propagation
- **training a neural net**

Supervised learning



Neural nets learn from humans



Supervised learning

Training

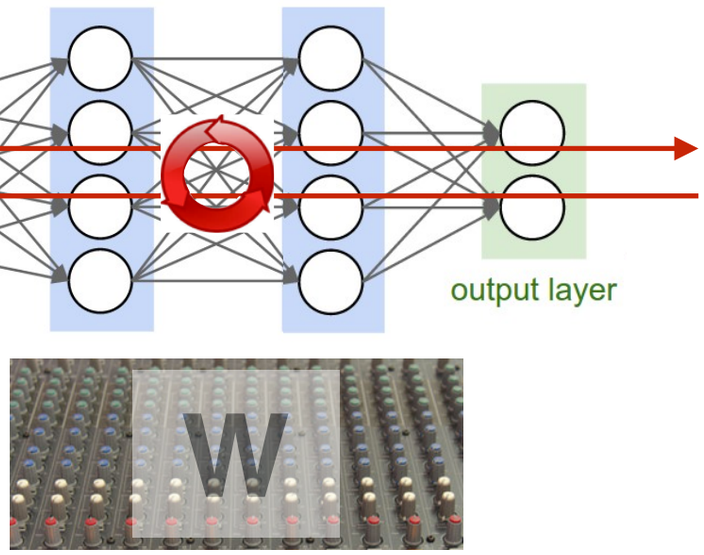
input

images with known labels

$X^{(1)}$=

$X^{(n)}$=

n~10^3-10^6



weights are tuned to minimize error

output

labels

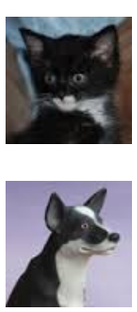
predicted catness (machine)	true catness (human)
0.9	$Y^{(1)}$= 1
0.2	0
0.1	0
0.2	0
0.4	1
0.9	1
0.8	$Y^{(n)}$= 1

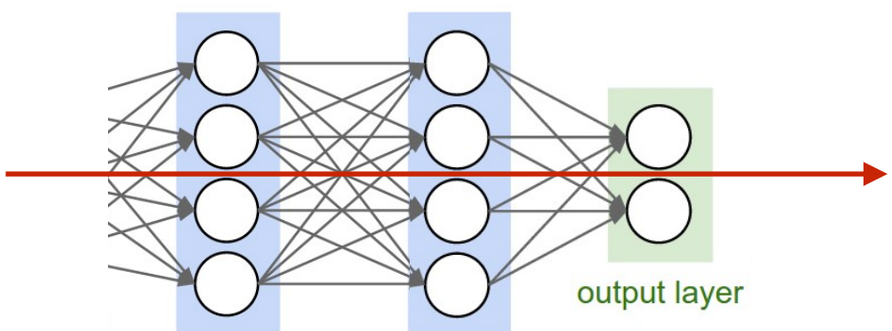
mean squared error =
$[(0.9-1)^2 +(0.2-0)^2 +(0.1-0)^2 +(0.2-0)^2 +(0.4-1)^2 +(0.9-1)^2 +(0.8-1)^2]/7$

number of possible 50x50 8-bit images = 256^{2500}

Testing

images without labels





predicted labels

predicted catness (machine)

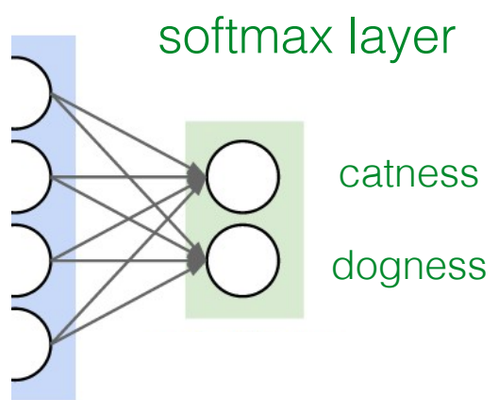
0.9 ⇨ “cat”

0.1 ⇨ “dog”

Cross-entropy

- We want to minimize the classification error
- "**Loss**" $J(W)$
- Simple loss: mean square error

$$J_i(W) = (Y_{i,1} - \mathfrak{F}_1(X_i; W))^2 + (Y_{i,2} - \mathfrak{F}_2(X_i; W))^2$$



- For classification, prefer **cross-entropy**:

$$J_i(W) = -Y_{i,1} \ln \mathfrak{F}_1(X_i; W) - (1 - Y_{i,1}) \ln (1 - \mathfrak{F}_1(X_i; W))$$

predicted catness

true catness

loss for image i

$$-Y_{i,2} \ln \mathfrak{F}_2(X_i; W) - (1 - Y_{i,2}) \ln (1 - \mathfrak{F}_2(X_i; W))$$

predicted dogness

true dogness

Total loss: $$J(W) = \frac{1}{N} \sum_{i=1}^{N} J_i(W)$$

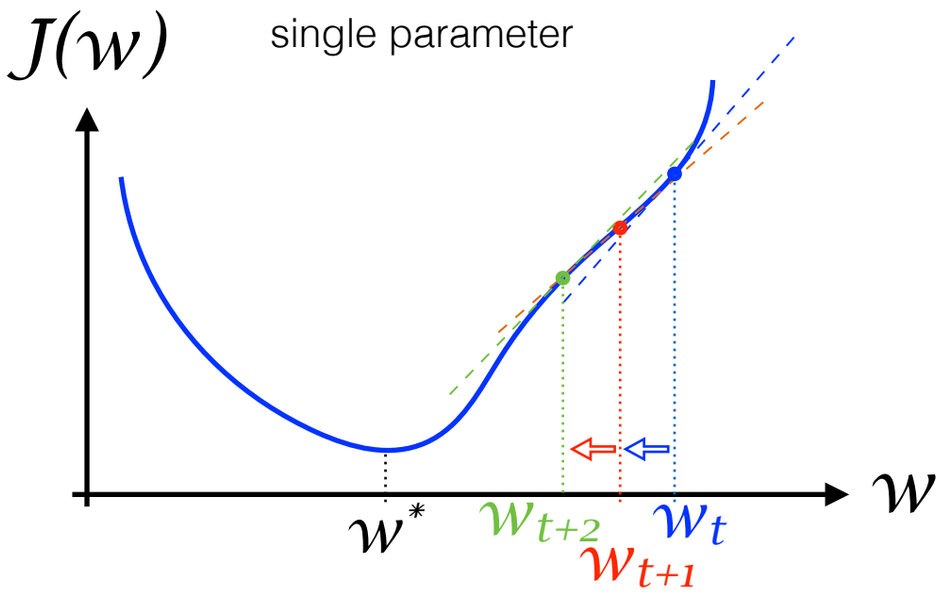
mean over the training examples

Gradient descent

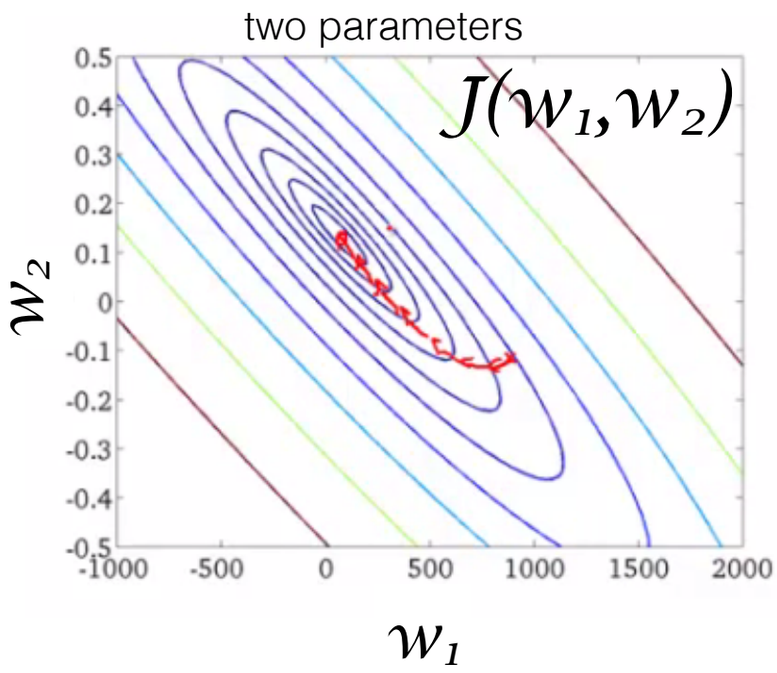
- We want the parameters *W* that minimize the loss *J* $W^* = \arg\min_W J(W)$
- Cannot be solved analytically
- We use **gradient descent**:

1. initialize parameters *W* (e.g. randomly) ← **use random values or start from a pre-trained network**
2. compute gradient of loss *J* relative to parameters in *W* $\nabla_W J(W)$
3. change parameters in direction of negative gradient $W := W - \alpha \nabla_W J(W)$
4. repeat from step 2 until convergence (or exhaustion)

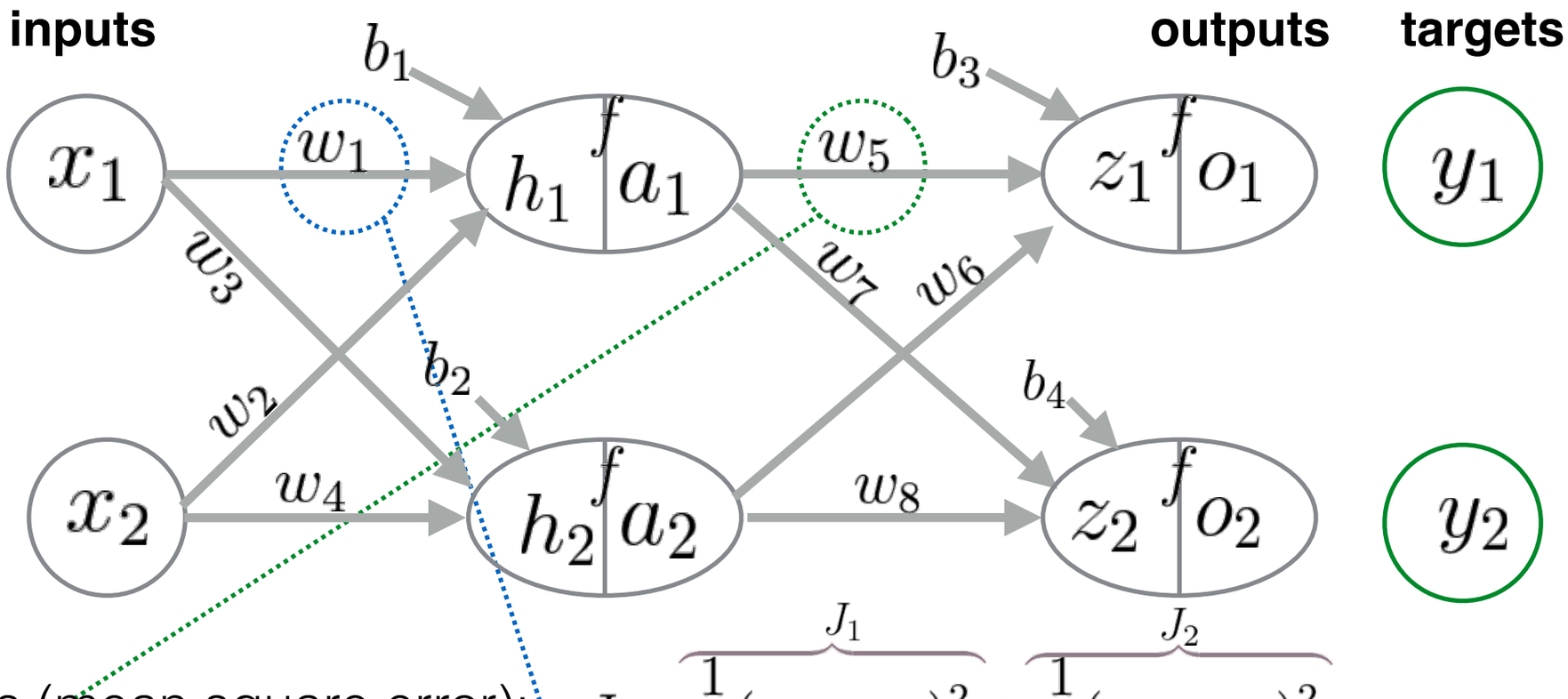
learning rate (a hyperparameter)



$$w := w - \alpha \frac{\mathrm{d}}{\mathrm{d}w} J(w)$$



Gradients are computed by backpropagation



Loss (mean square error): $J = \overbrace{\frac{1}{2}(o_1 - y_1)^2}^{J_1} + \overbrace{\frac{1}{2}(o_2 - y_2)^2}^{J_2}$

$$\frac{\partial J}{\partial w_5} = \frac{\partial J}{\partial o_1}\frac{\partial o_1}{\partial z_1}\frac{\partial z_1}{\partial w_5}$$

$o_1 - y_1$ $f'(z_1)$ a_1

$$\frac{\partial J}{\partial w_1} = \frac{\partial J}{\partial a_1}\frac{\partial a_1}{\partial h_1}\frac{\partial h_1}{\partial w_1} x_1$$

$f'(h_1)$

$$\frac{\partial J_1}{\partial a_1} + \frac{\partial J_2}{\partial a_1} = \frac{\partial J_1}{\partial o_1}\frac{\partial o_1}{\partial z_1}\frac{\partial z_1}{\partial a_1} + \frac{\partial J_2}{\partial o_2}\frac{\partial o_2}{\partial z_2}\frac{\partial z_2}{\partial a_1}$$

$o_1 - y_1$ $f'(z_1)$ w_5 $o_2 - y_2$ $f'(z_2)$ w_7

$$\Rightarrow \frac{\partial J}{\partial w_1} = \left[(o_1 - y_1) f'(z_1) w_5 + (o_2 - y_2) f'(z_2) w_7\right] f'(h_1) x_1$$

Stochastic gradient descent

Full loss: $J(W)=\frac{1}{N}\sum_{i=1}^{N}J_i(W)$ $=\frac{1}{N}\left[\sum_{i=1}^{100}J_i(W)+\sum_{i=101}^{200}J_i(W)+\ldots+\sum_{i=N-99}^{N}J_i(W)\right]$

mean over N training examples

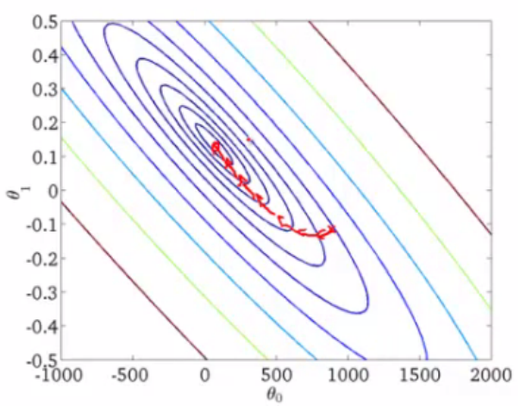
loss for image i

batch gradient descent

uses *all* examples at each iteration

$$W:=W-\alpha\nabla J(W)$$

- usually too expensive
- can fall in bad minimum



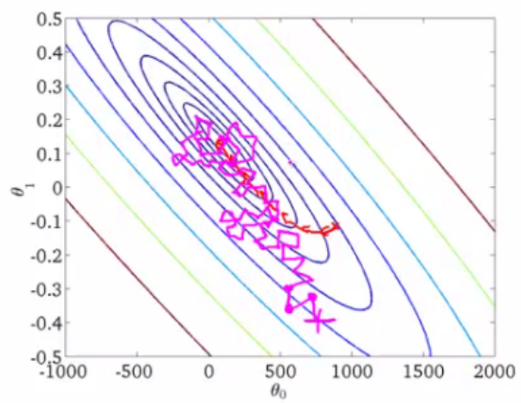
stochastic gradient descent

uses *a single* example at each iteration

- randomly reshuffle the *N* examples
- for i=1..*N* do:

$$W:=W-\alpha\nabla J_i(W)$$

- repeat



one epoch

mini-batch gradient descent

uses *m*=100 examples at each iteration

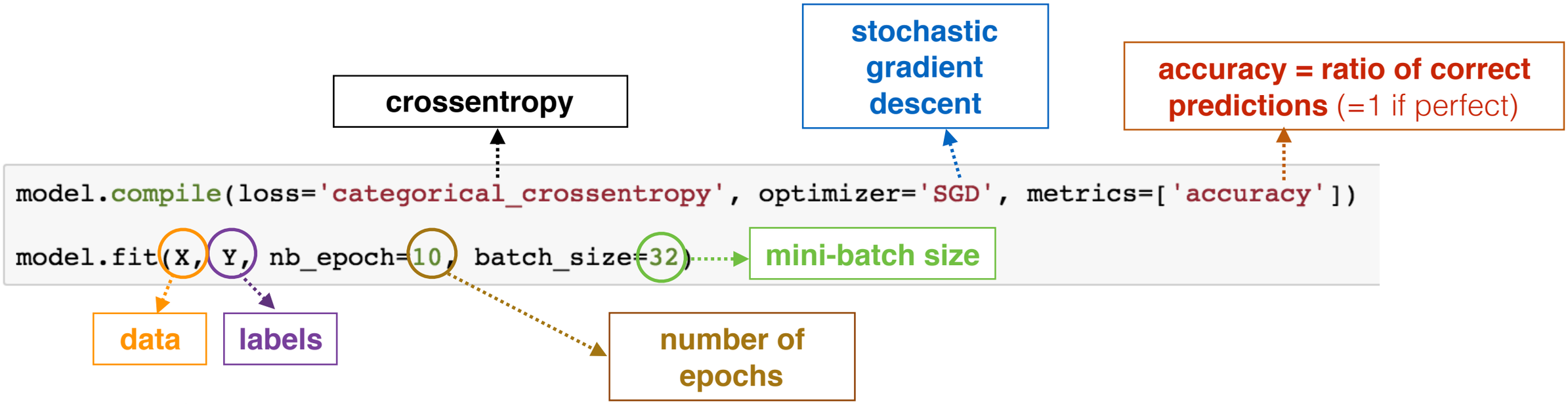
- randomly reshuffle the N examples
- for *k*=1,2,..,*N*/100 do:

$$W:=W-\frac{\alpha}{100}\nabla\left[\sum_{i=100(k-1)+1}^{100k}J_i(W)\right]$$

- repeat

- most widely used
- m~10-100 (hyperparameter)
- depends on memory

How to train a neural net in 2 lines

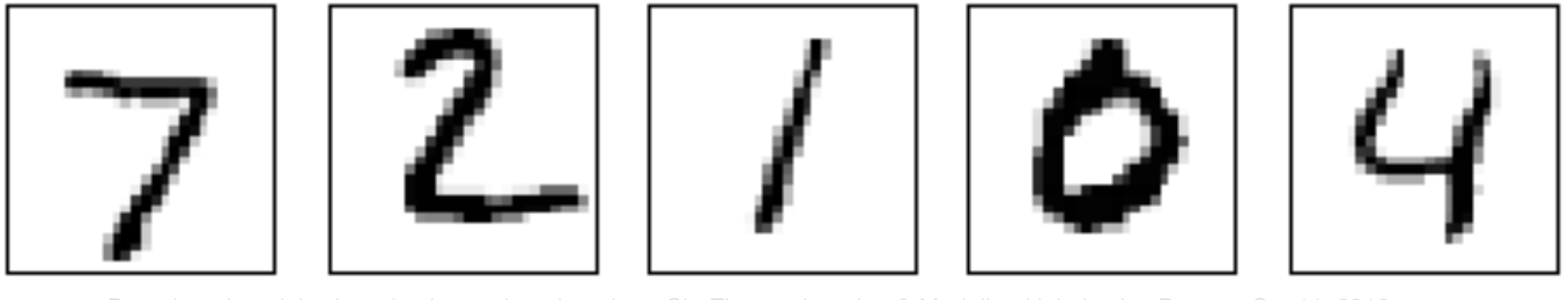



```
Epoch 1/10
1000/1000 [==============================] - 0s - loss: 2.8494 - acc: 0.1240
Epoch 2/10
1000/1000 [==============================] - 0s - loss: 2.8390 - acc: 0.2010
Epoch 3/10
1000/1000 [==============================] - 0s - loss: 2.8344 - acc: 0.3360
Epoch 4/10
1000/1000 [==============================] - 0s - loss: 2.8327 - acc: 0.3940
Epoch 5/10
1000/1000 [==============================] - 0s - loss: 2.8318 - acc: 0.4210
Epoch 6/10
1000/1000 [==============================] - 0s - loss: 2.8312 - acc: 0.4550
Epoch 7/10
1000/1000 [==============================] - 0s - loss: 2.8308 - acc: 0.4590
Epoch 8/10
1000/1000 [==============================] - 0s - loss: 2.8305 - acc: 0.4860
Epoch 9/10
1000/1000 [==============================] - 0s - loss: 2.8303 - acc: 0.4950
Epoch 10/10
1000/1000 [==============================] - 0s - loss: 2.8301 - acc: 0.5020
```

The MNIST data set

- 70,000 images of handwritten digits
- size=28x28 (784 pixels)
- 10 classes
- 7,000 images per class

Goal:
predict digits Y
from images X



Import MNIST data

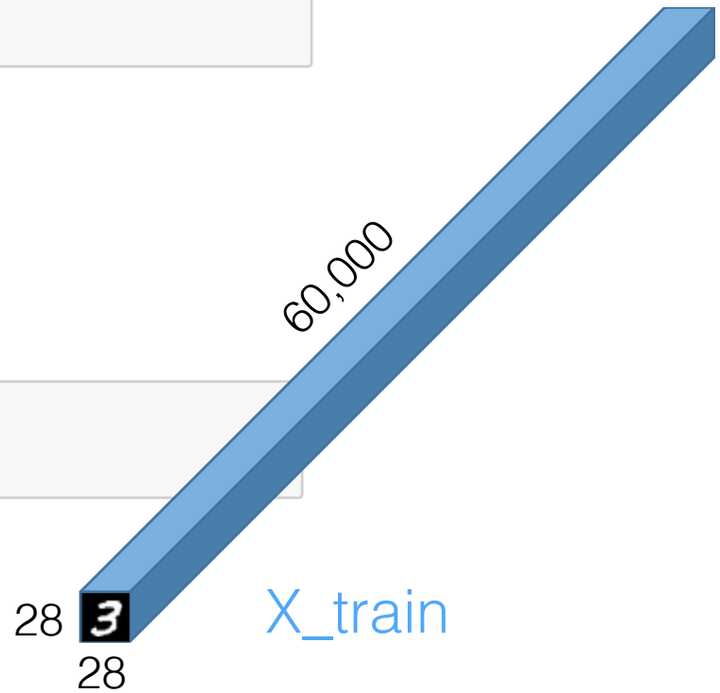
```
from keras.datasets import mnist
(X_train, y_train), (X_test, y_test) = mnist.load_data()
print("done importing")
```

```
done importing
```

training data

test data
(see later)

X = images
y = labels (0,1,..9)

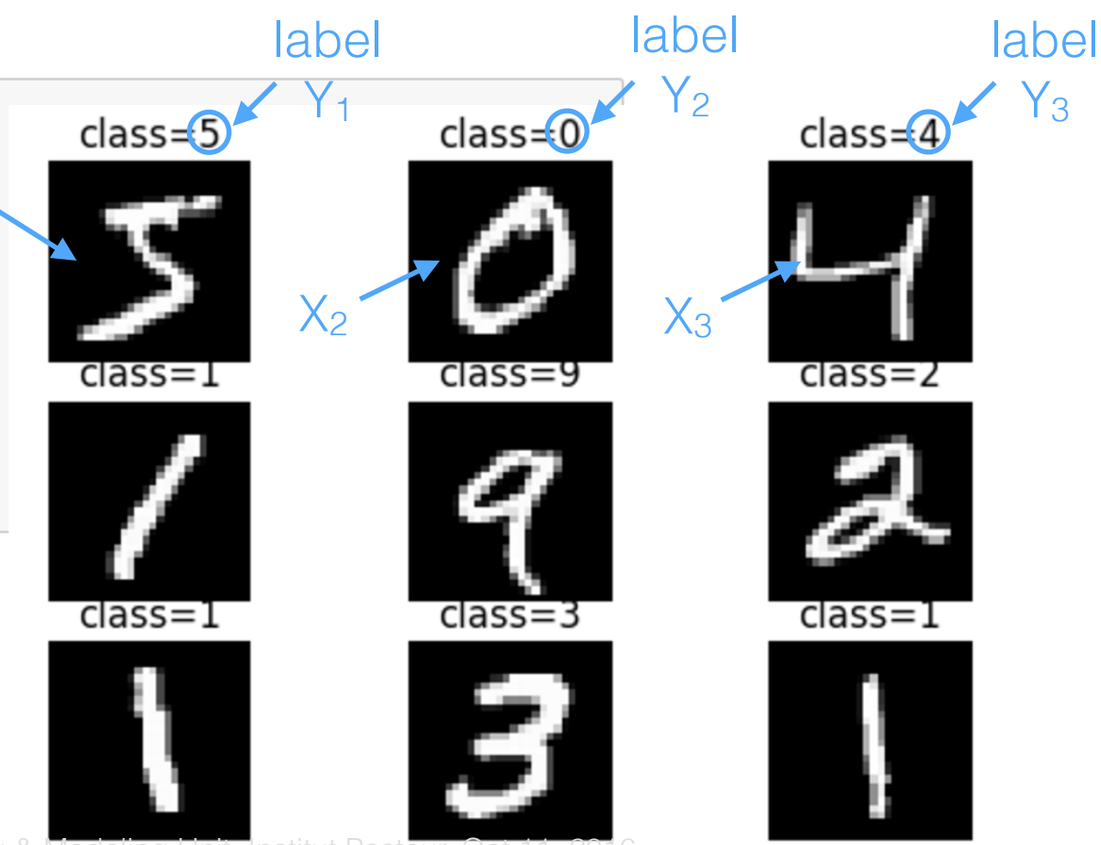



```
print(X_train.shape)
print(y_train.shape)
```

```
(60000, 28, 28)
(60000,)
```

Show some training images & labels

```
import matplotlib.pyplot as plt

%matplotlib inline

for i in range(9):
    plt.subplot(3,3,i+1)
    plt.imshow(X_train[i], cmap='gray', interpolation='nearest')
    plt.title("class={}".format(y_train[i]))
    plt.axis('off')
```



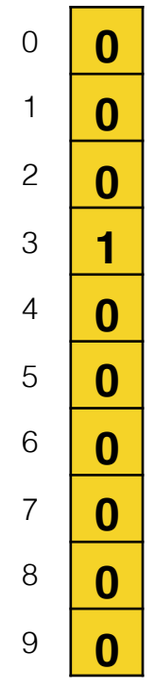
Some minor preprocessing

```
X_train = X_train.reshape(X_train.shape[0], 1, 28, 28)

X_train = X_train.astype('float32')

X_train /= 255
```

Make labels “one-hot”

```
from keras.utils import np_utils

print(y_train[0])

Y_train = np_utils.to_categorical(y_train, nb_classes)

print(Y_train[0])
```

```
5
[ 0.  0.  0.  0.  0.  1.  0.  0.  0.  0.]
```

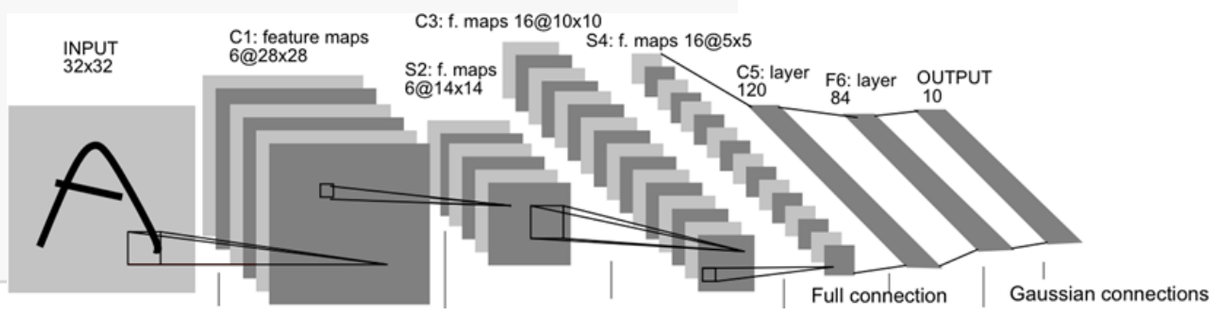


“one-hot” vector corresponding to the digit “3”

this is the ideal output for an image showing the number three, like this one:

Define convnet architecture

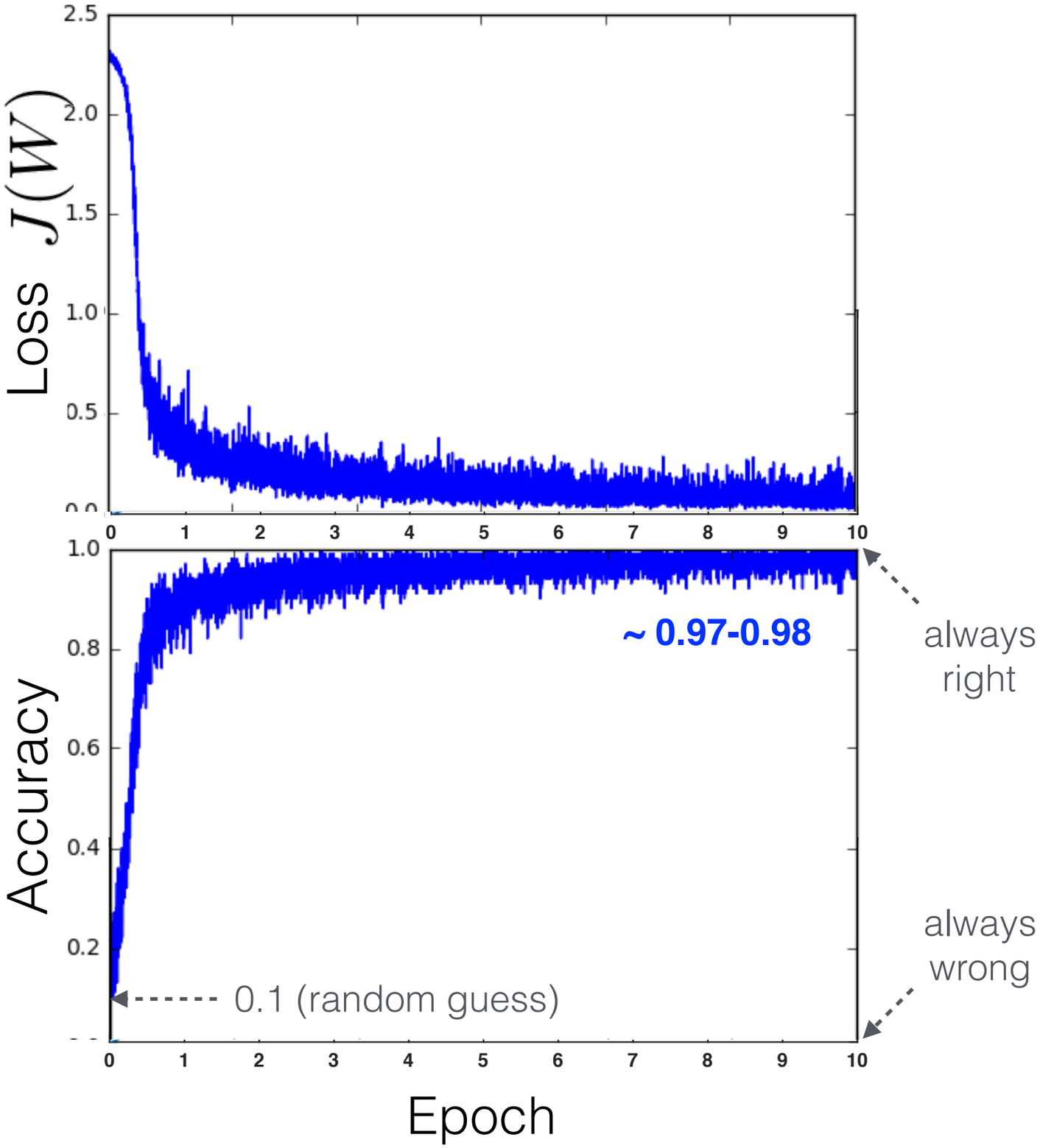
```
model = Sequential()
model.add(Convolution2D(6, 5, 5, input_shape=(1, 28, 28), activation='relu'))
model.add(MaxPooling2D(pool_size=(2,2)))
model.add(Convolution2D(16, 5, 5, activation='relu'))
model.add(MaxPooling2D(pool_size=(2,2)))
model.add(Flatten())
model.add(Dense(120, activation='relu'))
model.add(Dense(84, activation='relu'))
model.add(Dense(10, activation='softmax'))
```



Compile and train

```
model.compile(loss='categorical_crossentropy', optimizer='sgd', metrics=['accuracy'])

model.fit(X_train, Y_train,  batch_size=100, nb_epoch=10, callbacks=[plotting_callback])
```

Monitoring learning: training loss



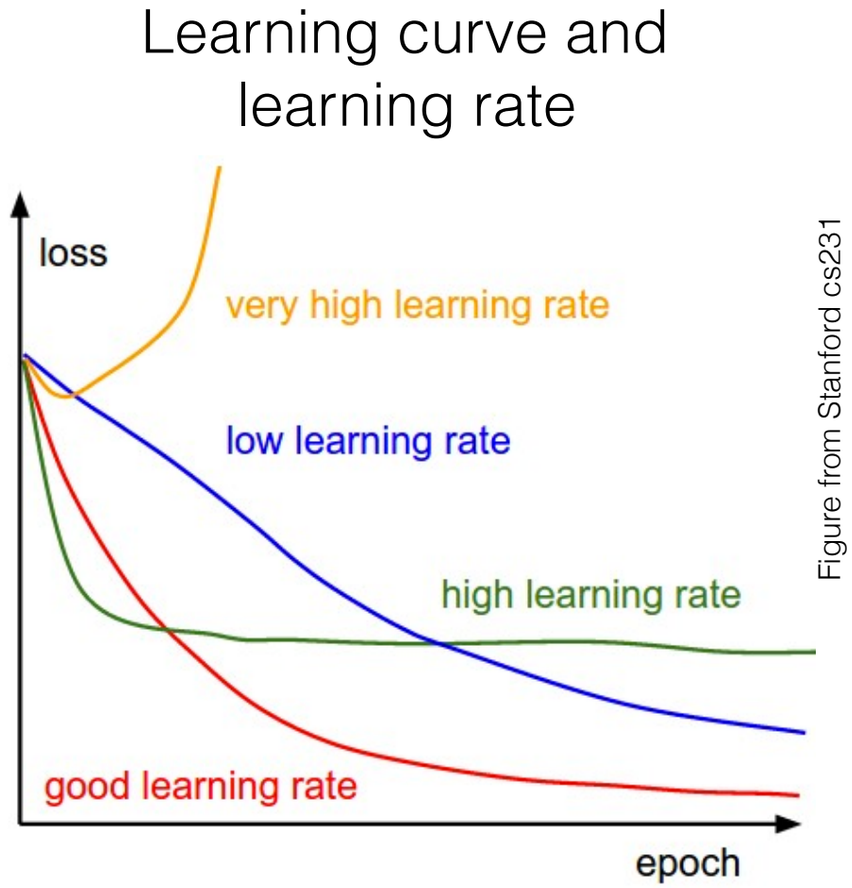
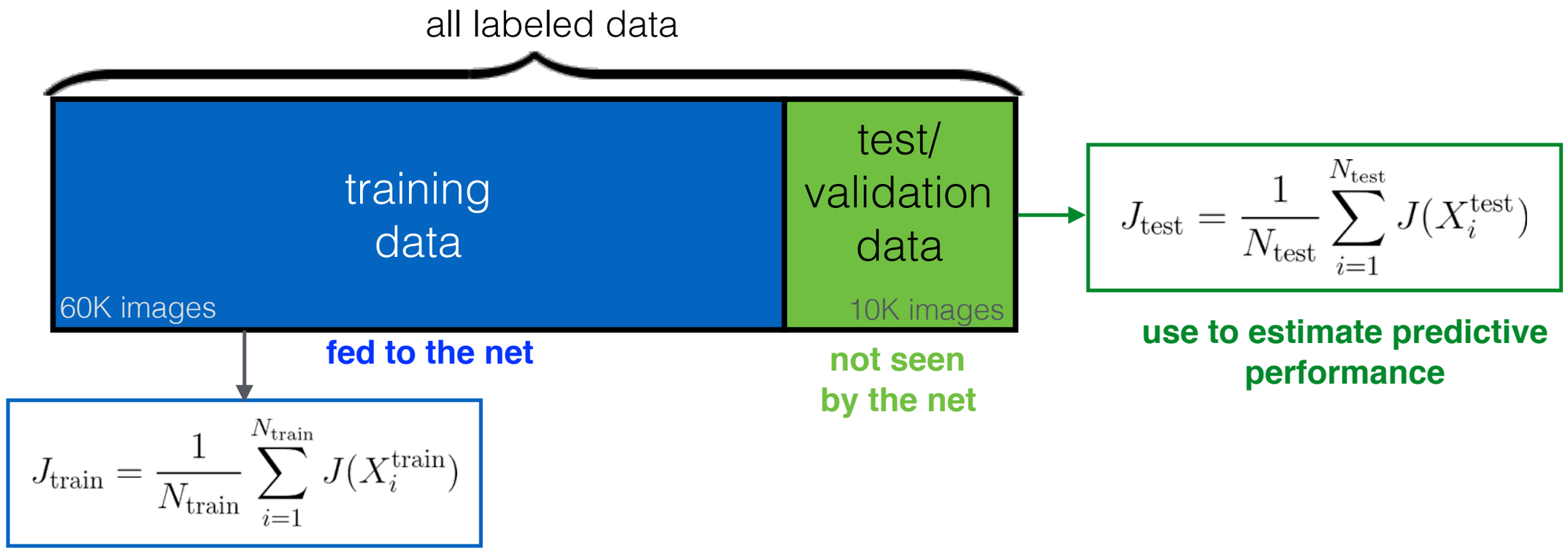


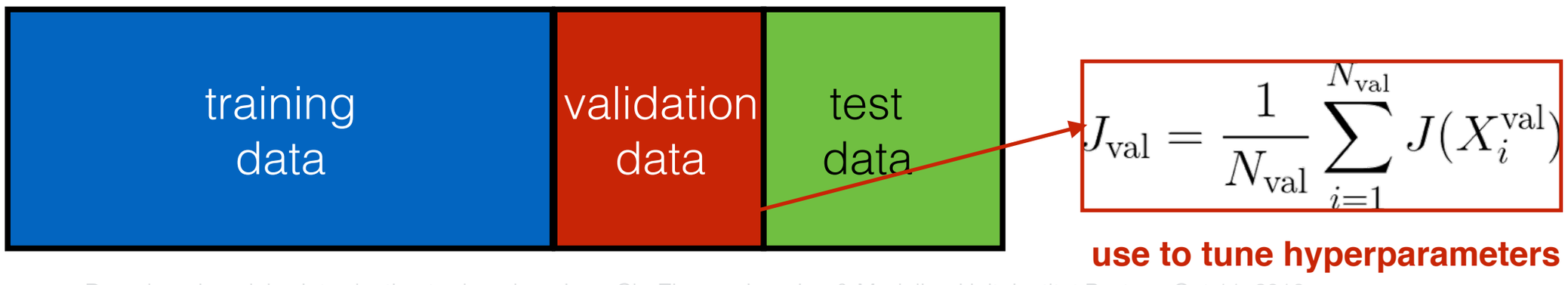
Figure from Stanford cs231

- When to stop ?
- The longer the better ?

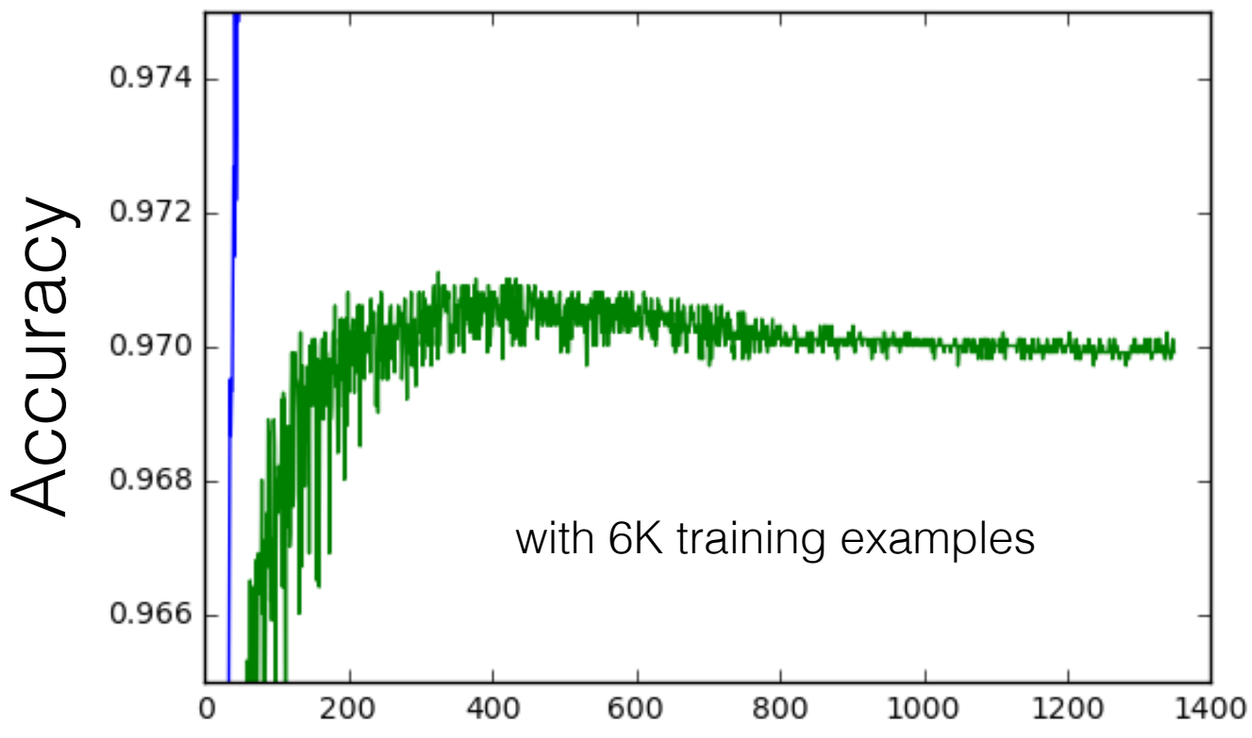
Monitoring learning: validation loss

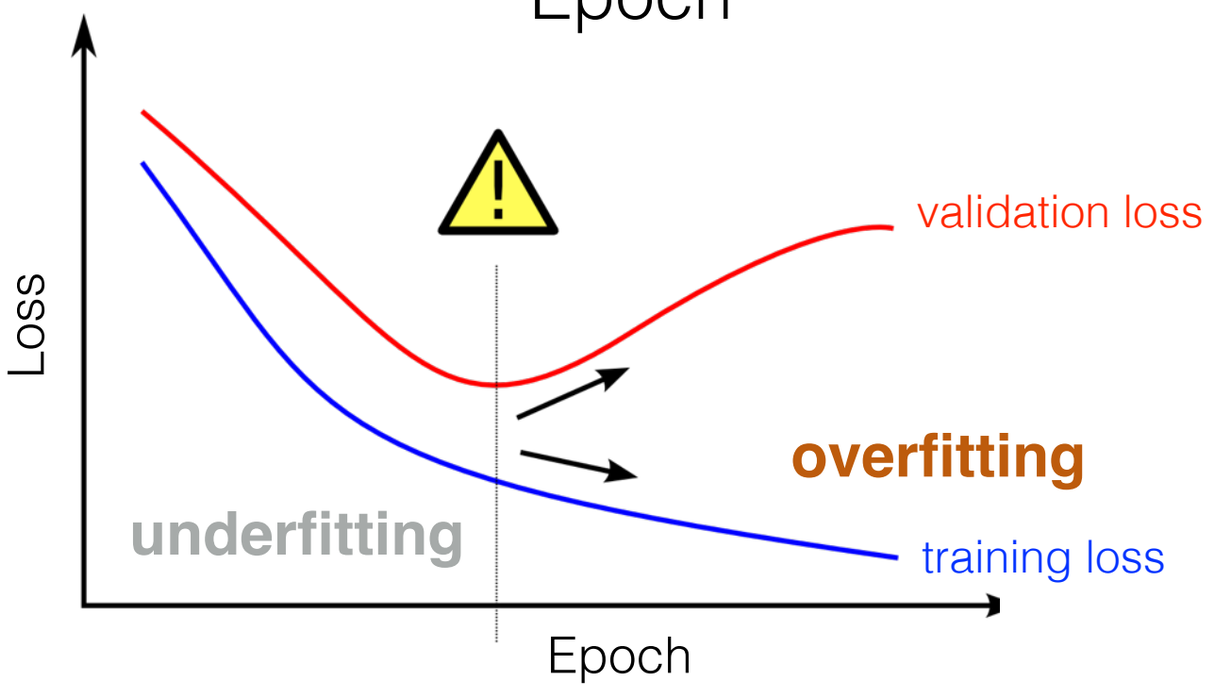
- The loss we used so far measures the performance on the *training data*
- We want correct predictions on new, *unlabelled* data
- We can measure this predictive power on *labelled* data *not* used for training





Monitoring learning: validation loss

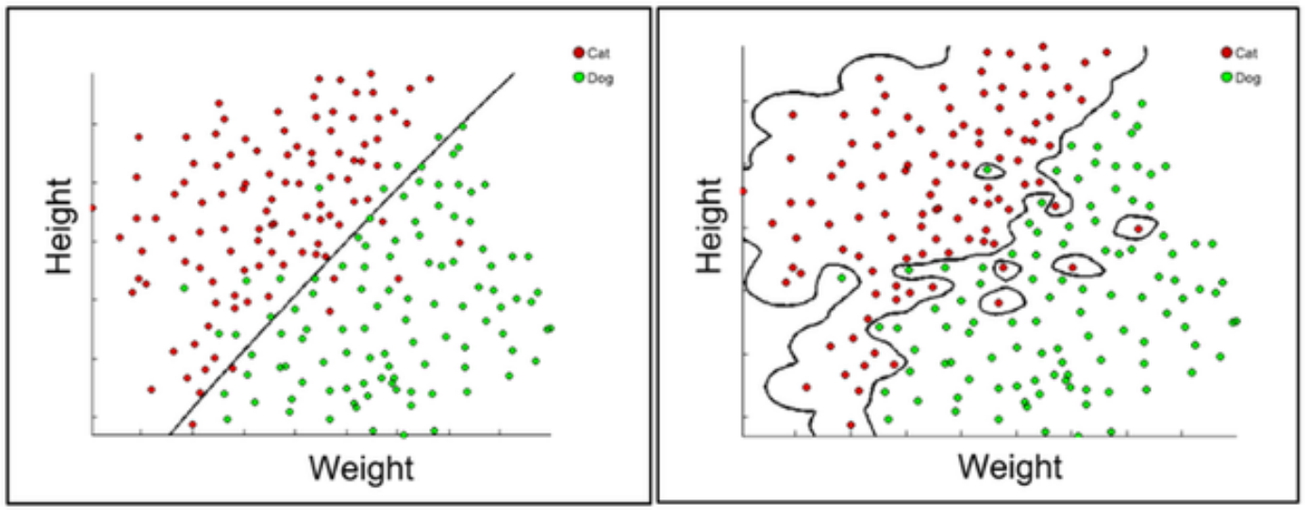




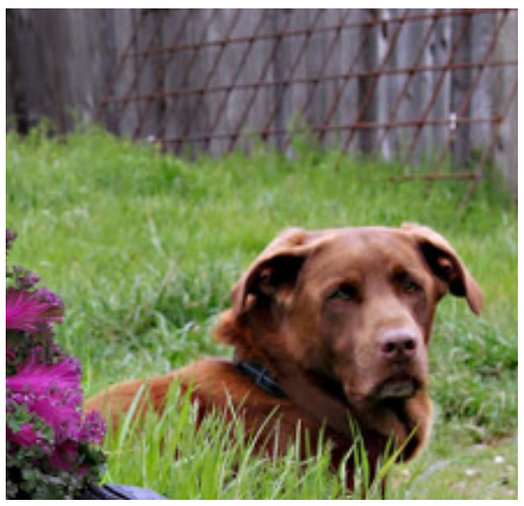
Overfitting:
model works very well on the training data but does not generalize well and performs poorly on unseen data (even if similar)

Overfitting

Predict if cat or dog from height and size



Predict if cat or dog from image



an overfitted image classification model might use forks to predict cats or violet flowers to predict dogs

Fighting under- and overfitting

Diagnostic

Underfitting (high bias)

- J_{val} keeps decreasing with epochs or flattens at high level
- J_{val} and J_{train} are similar
- J_{train} increases with more training data (N_{train}) and approaches J_{val}

Overfitting (high variance)

- J_{val} increases with epochs
- Low J_{train} and high J_{val} : large gap
- J_{val} decreases with more data (N_{train})

Remedies

- Train more (use GPUs)
- Increase learning rate ?
- Better initialization
- Better optimizer
- Make net bigger (deeper)
- Diminish regularization penalty

- Early stopping
- Get more data ! (use **data augmentation**)
- Add **regularization penalty**
- **Dropout**
- (Don't make the net smaller)

next slides

Data augmentation

input output

original image

“cats”

“cats”

x100

transformed images

rotations,

shifts,

flips,

shear,

zooms

…

“cats”

“cats”

“cats”

- transformations change inputs
- without changing outputs
- cheap increase of training data
- (of course less good than adding *real* data)

feeding the multitude miracle, Matthew 14:13-14:21

Regularization penalties

$$J(W) = \underbrace{J_{\text{error}}\left(\mathfrak{F}(X_i; W), Y\right)}_{\text{measures how well the training data are predicted}} + \underbrace{\lambda_1 \sum_{l=1}^{N_{\text{layers}}} \sum_{k,j} \left|W_{k,j}^{l}\right| + \lambda_2 \sum_{l=1}^{N_{\text{layers}}} \sum_{k,j} \left(W_{k,j}^{l}\right)^2}_{\textbf{data independent penalty on the weights}}$$

e.g. cross-entropy

L1 regularization:

encourages sparsity

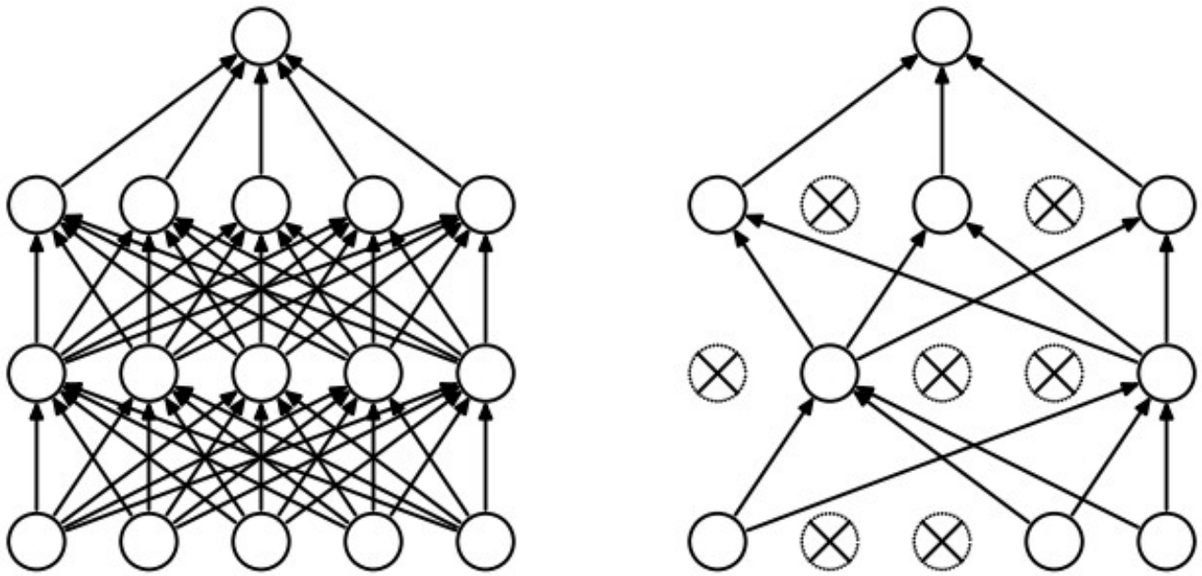
L2 regularization:

discourages very high weights, makes weights more diffuse

λ_1, λ_2 **are hyperparameter**

Dropout

- randomly inactivate neurons with probability *p* at each iteration
- at test time, use “mean net” with weights multiplied by *p*



(a) Standard Neural Net (b) After applying dropout.

p is a hyperparameter

here p=0.5

Srivastava *et al. J Mach Res* 2014

- usually beats other forms of regularization
- very effective

```
model.add(Convolution2D(16, 5, 5, activation='relu'))
model.add(Dropout(0.2))
```

Complete code to train a convnet on MNIST data (1/4)

Import MNIST data

```
from keras.datasets import mnist

(X_train, y_train), (X_test, y_test) = mnist.load_data()

print('X_train shape:', X_train.shape)

print(X_train.shape[0], 'train samples')

print(X_test.shape[0], 'train samples')
```

```
('X_train shape:', (60000, 28, 28))
(60000, 'train samples')
(10000, 'train samples')
```

Reshape and normalized images

```
X_train = X_train.reshape(X_train.shape[0], 1, 28, 28)
X_train = X_train.astype('float32')
X_train /= 255

X_test = X_test.reshape(X_test.shape[0], 1, 28, 28)
X_test = X_test.astype('float32')
X_test /= 255
```

Make labels one-hot

```
from keras.utils import np_utils

print(y_train[0])

Y_train = np_utils.to_categorical(y_train, 10)
print(Y_train[0])

Y_test = np_utils.to_categorical(y_test, 10)
```

```
5
[ 0.  0.  0.  0.  0.  1.  0.  0.  0.  0.]
```

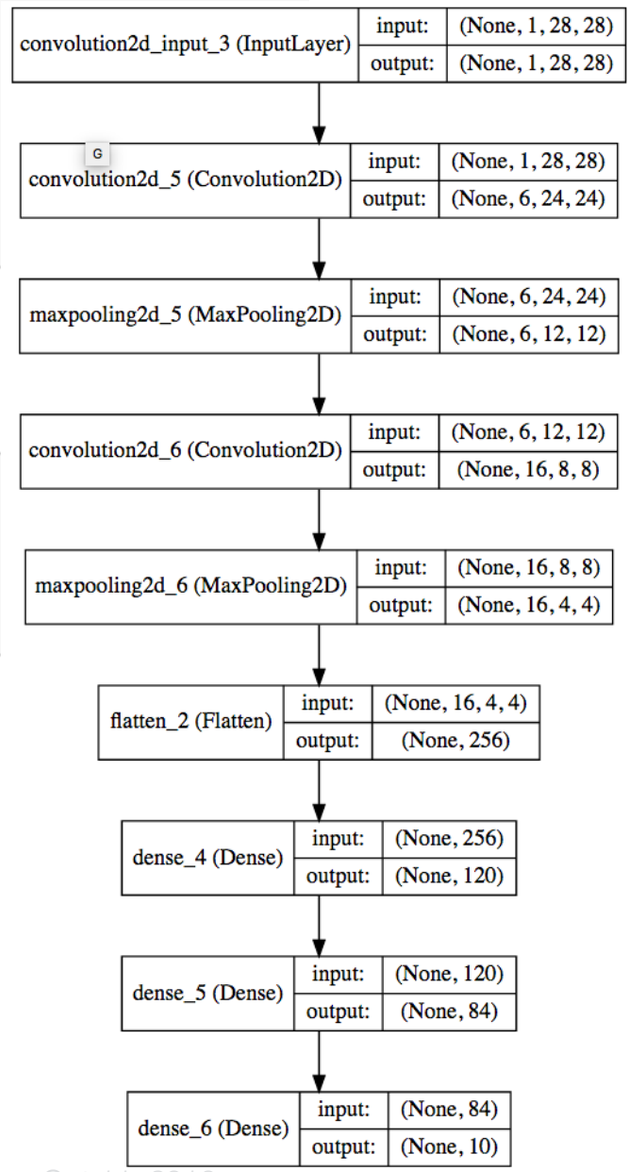
Complete code to train a convnet on MNIST data (2/4)

Define convnet (LeNet-5-like)

```
from keras.models import Sequential
from keras.layers import Convolution2D, MaxPooling2D
from keras.layers import Dense, Dropout, Activation, Flatten

model = Sequential()
model.add(Convolution2D(6, 5, 5, input_shape=(1, 28, 28), activation='relu', dropout=0.5))
model.add(MaxPooling2D(pool_size=(2,2)))
model.add(Convolution2D(16, 5, 5, activation='relu', dropout=0.5))
model.add(MaxPooling2D(pool_size=(2,2)))
model.add(Flatten())
model.add(Dense(120, activation='relu', dropout=0.5))
model.add(Dense(84, activation='relu', dropout=0.5))
model.add(Dense(10, activation='softmax'))
```

Visualize network

```
from IPython.display import SVG
from keras.utils.visualize_util import model_to_dot

SVG(model_to_dot(model, show_shapes=True).create(prog='dot', format='svg'))
```



Complete code to train a convnet on MNIST data (3/4)

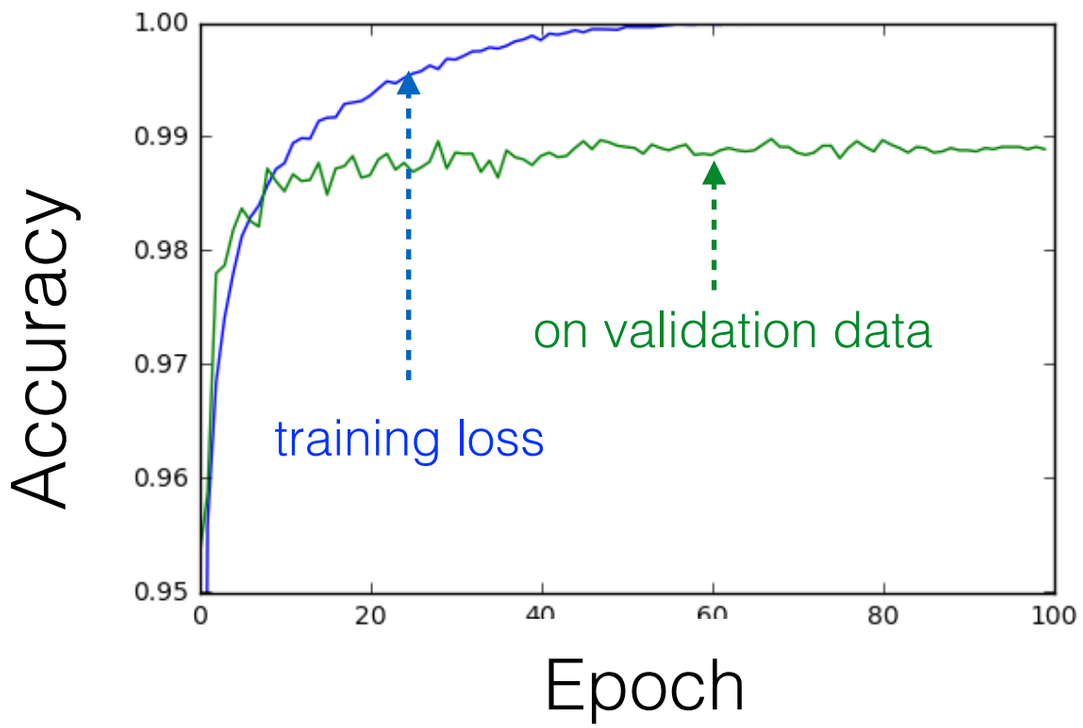
Compile and train the network

```
model.compile(loss='categorical_crossentropy', optimizer='sgd', metrics=['accuracy'])

model.fit(X_train, Y_train, validation_data=(X_test, Y_test), batch_size=32, nb_epoch=100)
```

```
Epoch 1/100
60000/60000 [==============================] - 10s - loss: 0.4856 - acc: 0.8530 - val_loss: 0.1491 - val_acc: 0.9528
Epoch 2/100
60000/60000 [==============================] - 10s - loss: 0.1434 - acc: 0.9559 - val_loss: 0.1262 - val_acc: 0.9587
Epoch 3/100
60000/60000 [==============================] - 10s - loss: 0.1019 - acc: 0.9684 - val_loss: 0.0739 - val_acc: 0.9780
Epoch 4/100
60000/60000 [==============================] - 10s - loss: 0.0823 - acc: 0.9741 - val_loss: 0.0675 - val_acc: 0.9787
```

...

```
Epoch 98/100
60000/60000 [==============================] - 10s - loss: 4.4440e-04 - acc: 1.0000 - val_loss: 0.0598 - val_acc: 0.9889
Epoch 99/100
60000/60000 [==============================] - 10s - loss: 4.4169e-04 - acc: 1.0000 - val_loss: 0.0598 - val_acc: 0.9891
Epoch 100/100
60000/60000 [==============================] - 10s - loss: 4.3478e-04 - acc: 1.0000 - val_loss: 0.0598 - val_acc: 0.9889
```



accuracy on training data

accuracy on validation data

state of the art accuracy = 99.77 % (committees of 30 CNNs)

Complete code to train a convnet on MNIST data (4/4)

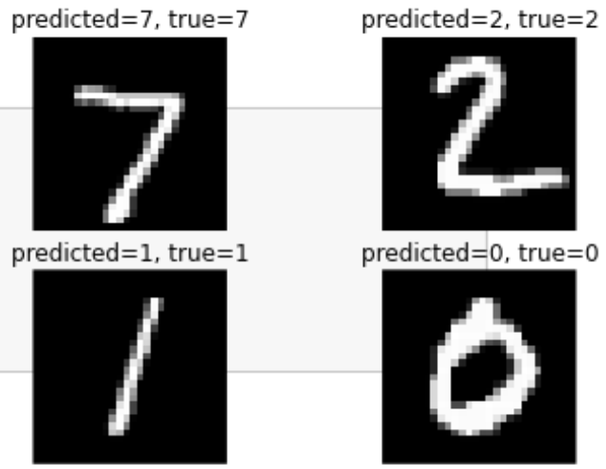
Find correct predictions and mistakes

```
predicted_classes = model.predict_classes(X_test)

import numpy as np

correct_indices = np.nonzero(predicted_classes == y_test)[0]
incorrect_indices = np.nonzero(predicted_classes != y_test)[0]
```

```
 9984/10000 [============================>.] - ETA: 0s
```

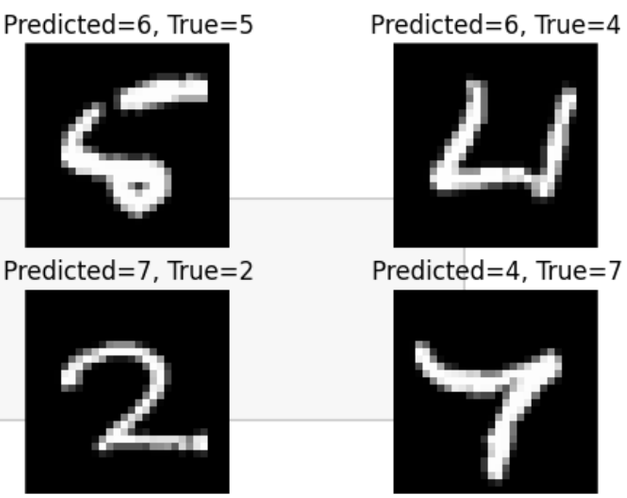
Inspect some correct predictions

```
for i, correct in enumerate(correct_indices[:4]):
    plt.subplot(2,2,i+1)
    plt.imshow(X_test[correct].reshape(28,28), cmap='gray', interpolation='none')
    plt.title("predicted={}, true={}".format(predicted_classes[correct], y_test[correct]))
    plt.axis("off")
```



Inspect wrong predictions

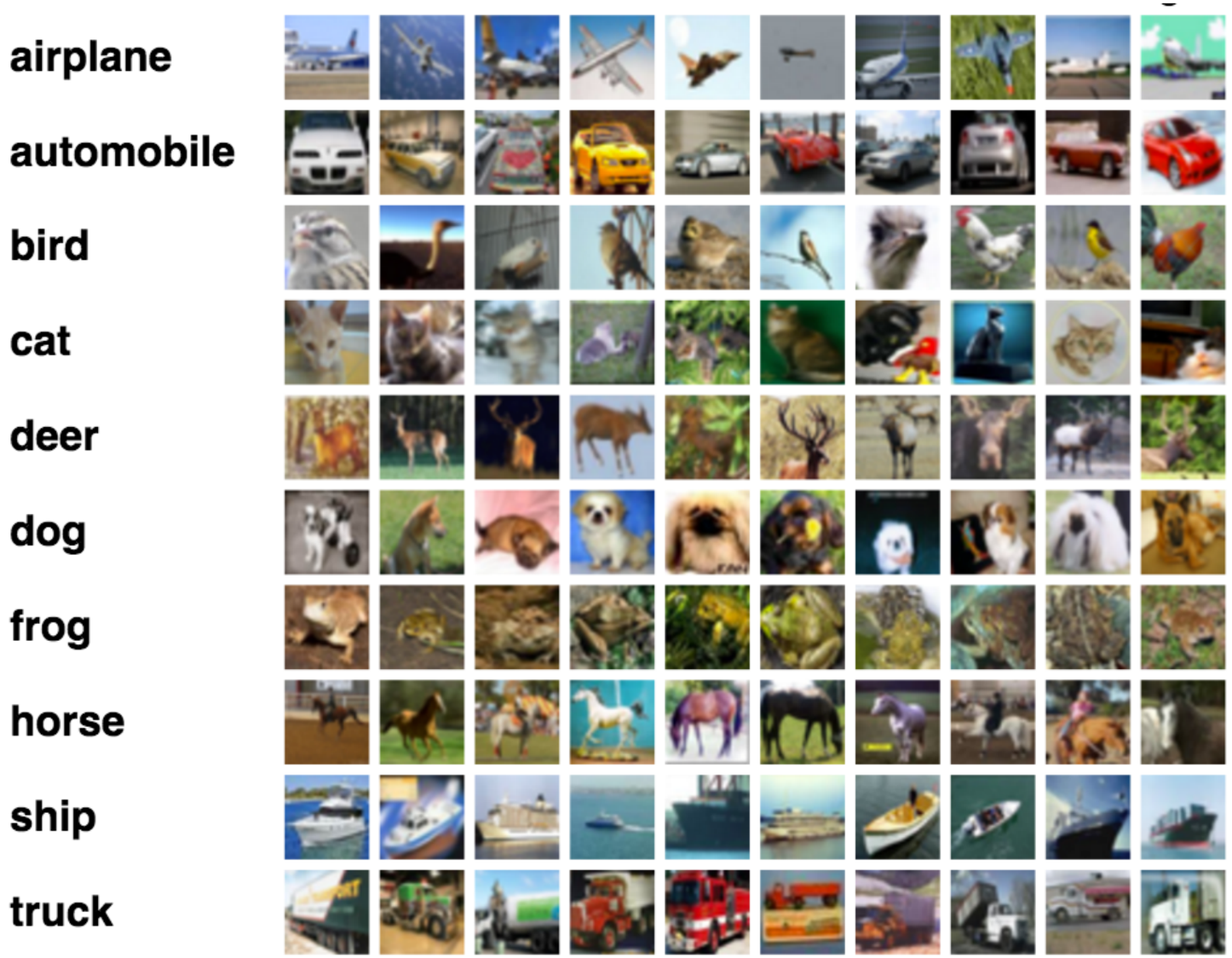
```
for i, incorrect in enumerate(incorrect_indices[:4]):
    plt.subplot(2,2,i+1)
    plt.imshow(X_test[incorrect].reshape(28,28), cmap='gray', interpolation='none')
    plt.title("Predicted={}, True={}".format(predicted_classes[incorrect], y_test[incorrect]))
    plt.axis("off")
```



The CIFAR-10 data set

- 60,000 RGB images
- size=32x32x3 (3072 values)
- 10 classes
- 6,000 images per class

- Training set: 5,000 images per class
- Test set: 1,000 images per class



can get >80% test accuracy on a laptop in 18h out of the box (w/o babysitting)

(state-of-the art: >92%)

Summary

- Deep learning achieves breakthroughs in multiple domains
- Basic principles of deep learning
  - Artificial neurons compute take inputs from other neurons, compute a linear combination with weights and biases, and pass the result to a non-linear function as output
  - Neurons are connected into networks. Neural nets are universal function approximators
  - Convnets have sparse connections and shared weights
  - Parameters of neural nets are learned iteratively from training data using backpropagation and (stochastic) gradient-descent
  - In supervised learning, a neural net is trained to predict outputs from human labels
  - Overfitting and underfitting can be diagnosed and treated
- Free and simple tools make deep learning widely accessible

Deep learning word cloud

labels

neural net

stochastic gradient descent

preactivation

weights

epoch

loss

convnet

activation

hidden layer

ReLu

backprop

hyperparameter

data augmentation

learning curve

regularization

overfitting

dropout

neuron

Resnet

underfitting

learning rate

cross-entropy

softmax

autoencoder

Adagrad

sigmoid

model

training data

catness

AI

LSTM

autoencoder

validation data

regression

dogness

RNN

memory nets

Boltzmann machines

neural Turing machines

generative adversarial nets

transfer learning

learning to learn

teacher forcing

Some deep learning resources

- Coursera “Machine learning” by A. Ng
- Coursera “Neural networks for machine learning” by G. Hinton (2013)
- Collège de France lectures by Y. LeCun
- Udacity course “Deep Learning” by Google
- MIT book “Deep learning“ by Y. Bengio et al. (www.deeplearningbook.org)
- many more free resources available online