Everything You Always Wanted to Know About

Neural Networks

But Were Afraid to Ask

Ilya Ezepov

Agenda

- Linear models
- Training: from GD to SGD
- Perceptron
- Deep NN
- Convolutional NN
- Modern approach to NN
- SGD hacks
- When to use NN and when to not

18. … Bxe7

White to move

19.c4

Garry Kasparov

Garry Kasparov

Deep Blue

Checkers: completely solved (2007)

Checkers: completely solved (2007)

Chess: February 1996
(first win by computer against top human)

Checkers: completely solved (2007)

Chess: February 1996
(first win by computer against top human)

Chess: November 2005
(last win by human against top computer)

Checkers: completely solved (2007)

Chess: February 1996
(first win by computer against top human)

Chess: November 2005
(last win by human against top computer)

Go: Uncracked for a long time

AlphaGo (by Google)

AlphaGo

VS

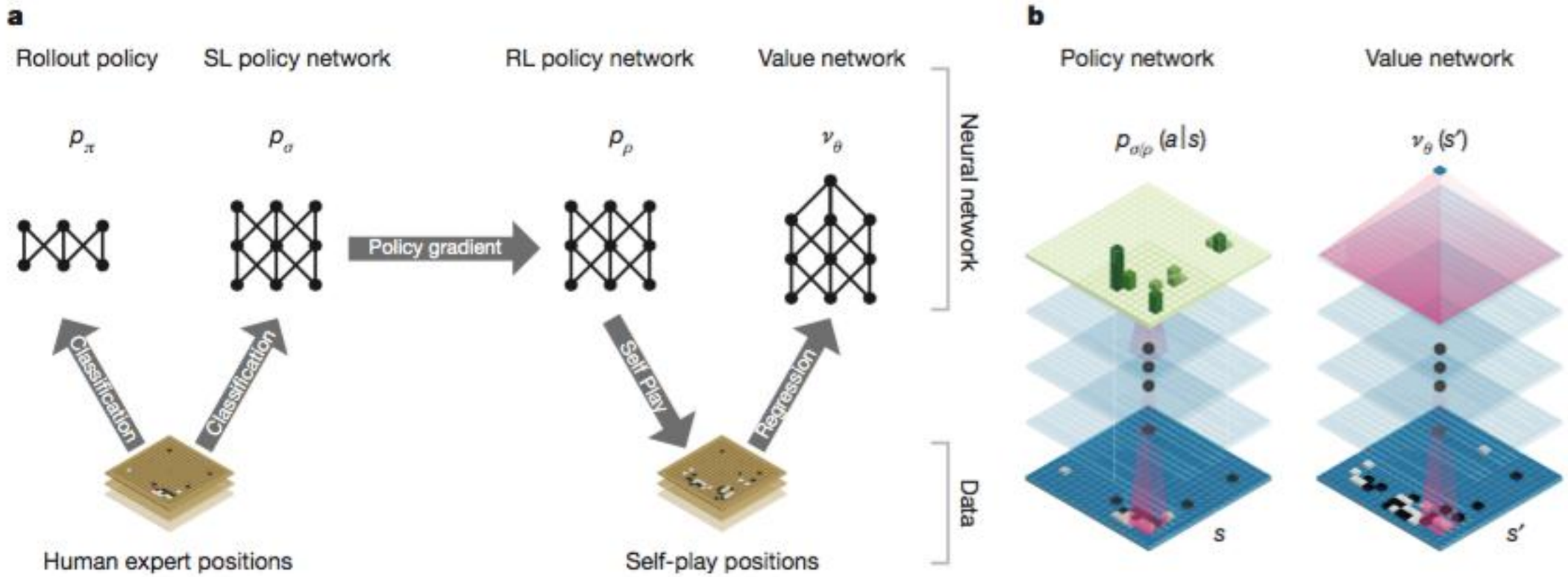
Fan Hui,
European champion

AlphaGo (by Google)

AlphaGo 5:0

Fan Hui,
European champion

It works on neural networks!



Mastering the game of Go with deep neural networks and tree search
Nature 529, 484–489 (28 January 2016) doi:10.1038/nature16961

AlphaGo (by Google)

AlphaGo ?

Lee Sedol,
Last decade top player

Linear Models

The Problem

Feature matrix

Many examples

X_1	X_2	X_3	X_4

?

Correct answer

y

The Problem

Model:

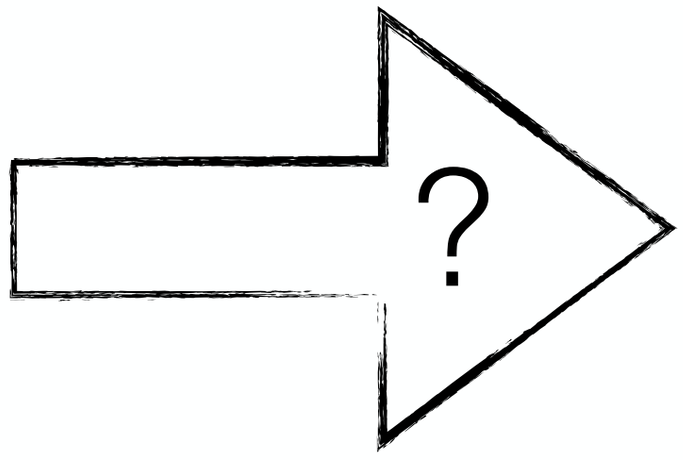
$$X_1W_1 + X_2W_2 + X_3W_3 + X_4W_4 + b = \hat{y}$$

How to find W, b?

Adding bias

Feature matrix (X)

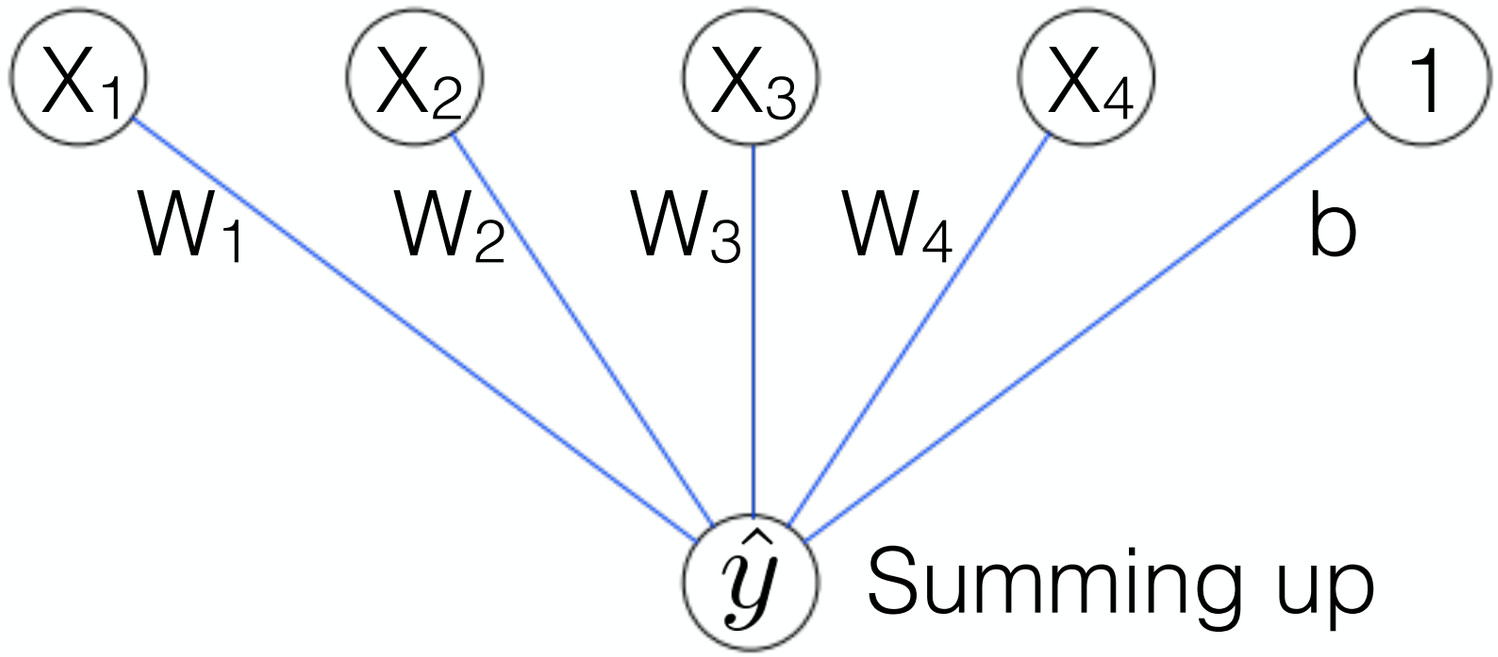
X_1	X_2	X_3	X_4	X_5
				1
				1
				1
				1
				1
				1
				1
				1
				1
				1
				1
				1



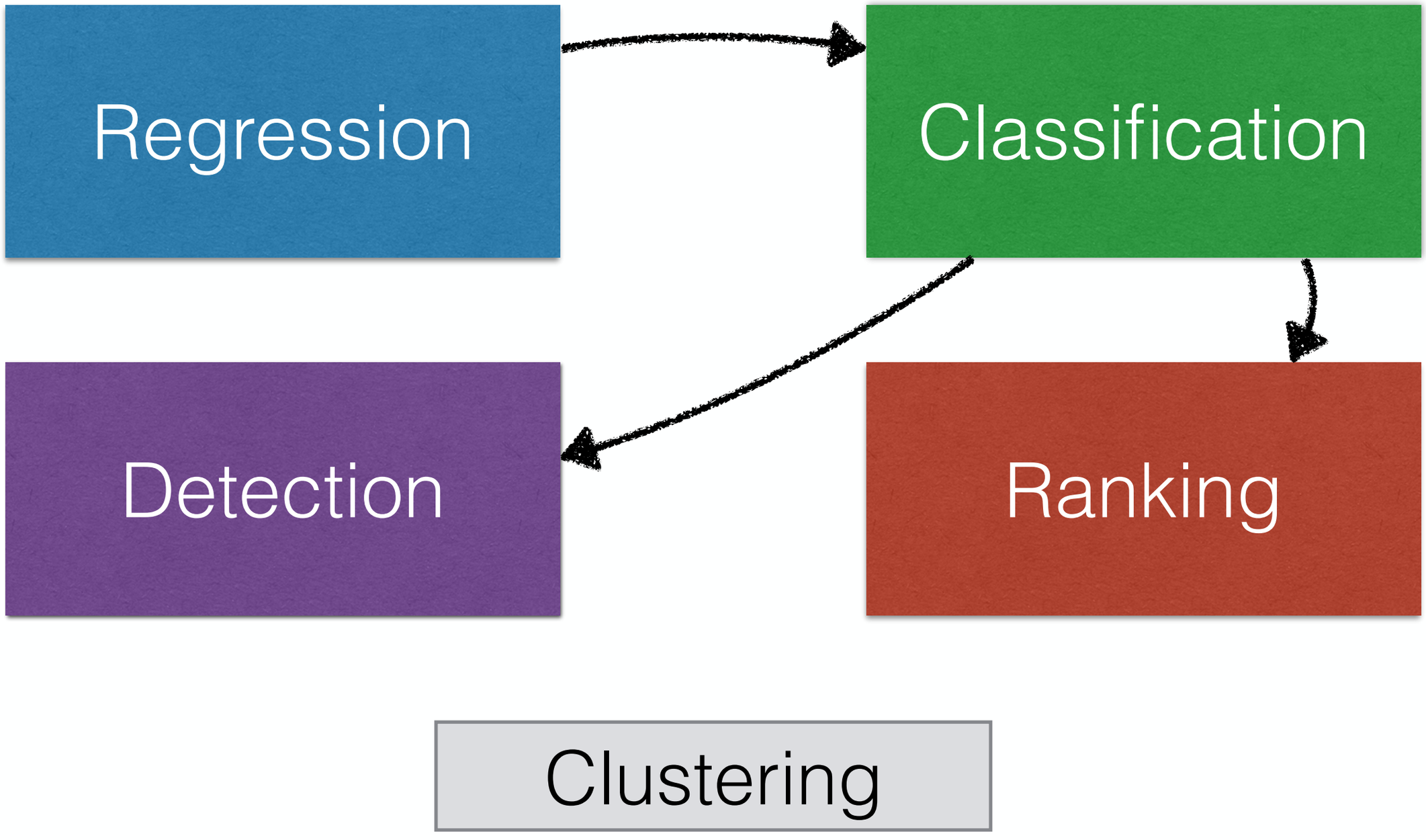
Correct answer (y)

y

Simple linear model



Machine learning
types of problems



Clustering

How to train?

- “Training” = Finding optimal W, b

How to train?

- “Training” = Finding optimal W, b
- “Optimal” = Minimize loss function (a.k.a. cost function)

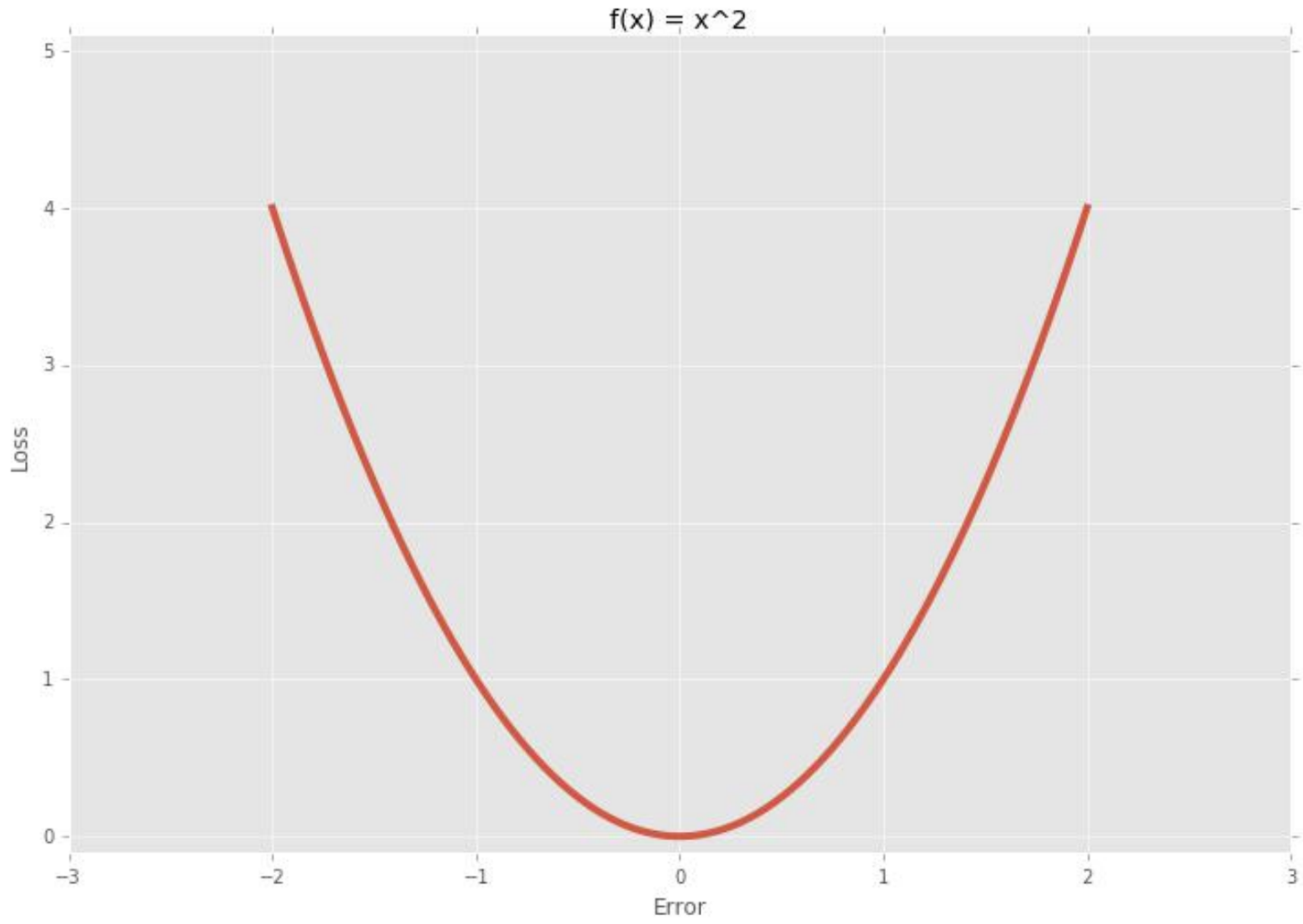
How to train?

- “Training” = Finding optimal W, b
- “Optimal” = Minimize loss function (a.k.a. cost function)
- The must-know losses:
  - MSE (mean squared error)
  - Log loss
  - Hinge loss
  - Huber loss

Loss functions

Mean Squared Error (MSE) for regression

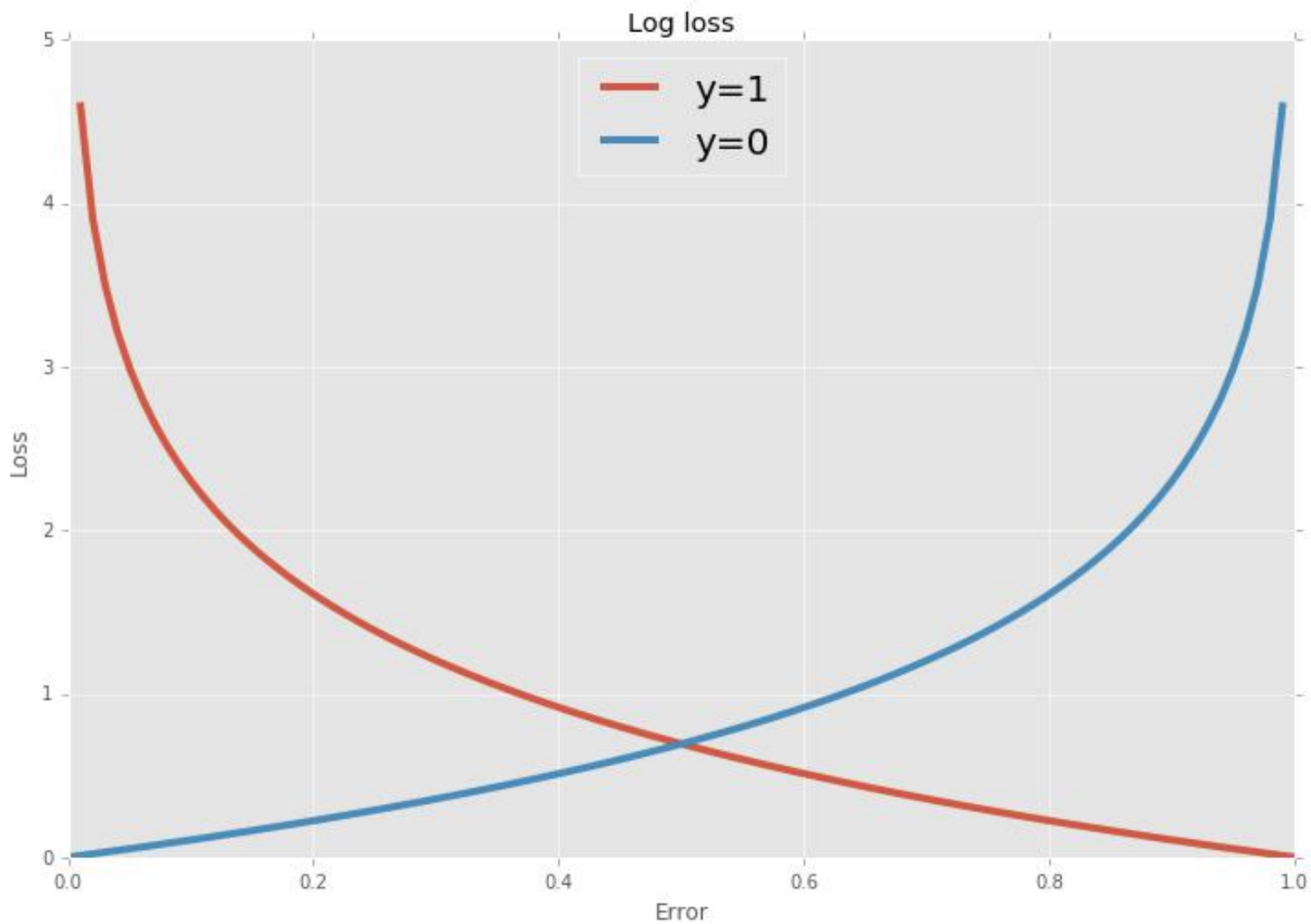
$$\frac{1}{N}\sum_{i=1}^{N}\frac{1}{2}(y_i - \hat{y}_i)^2$$



Loss functions

Logarithmic loss for classification

$$-\frac{1}{N}\sum_{i=1}^{N}(y_i \ln p_i + (1-y_i)\ln(1-p_i))$$



Explicit Solution (OLS)

$$XW = \hat{y}$$

$$\frac{1}{2}(y - XW)(y - XW)^T \to min$$

$$W = (X^T X)^{-1} X^T y$$

Explicit Solution

$$XW = \hat{y}$$

$$\frac{1}{2}(y - XW)(y - XW)^T \to min$$

$$W = (X^T X)^{-1} X^T y$$

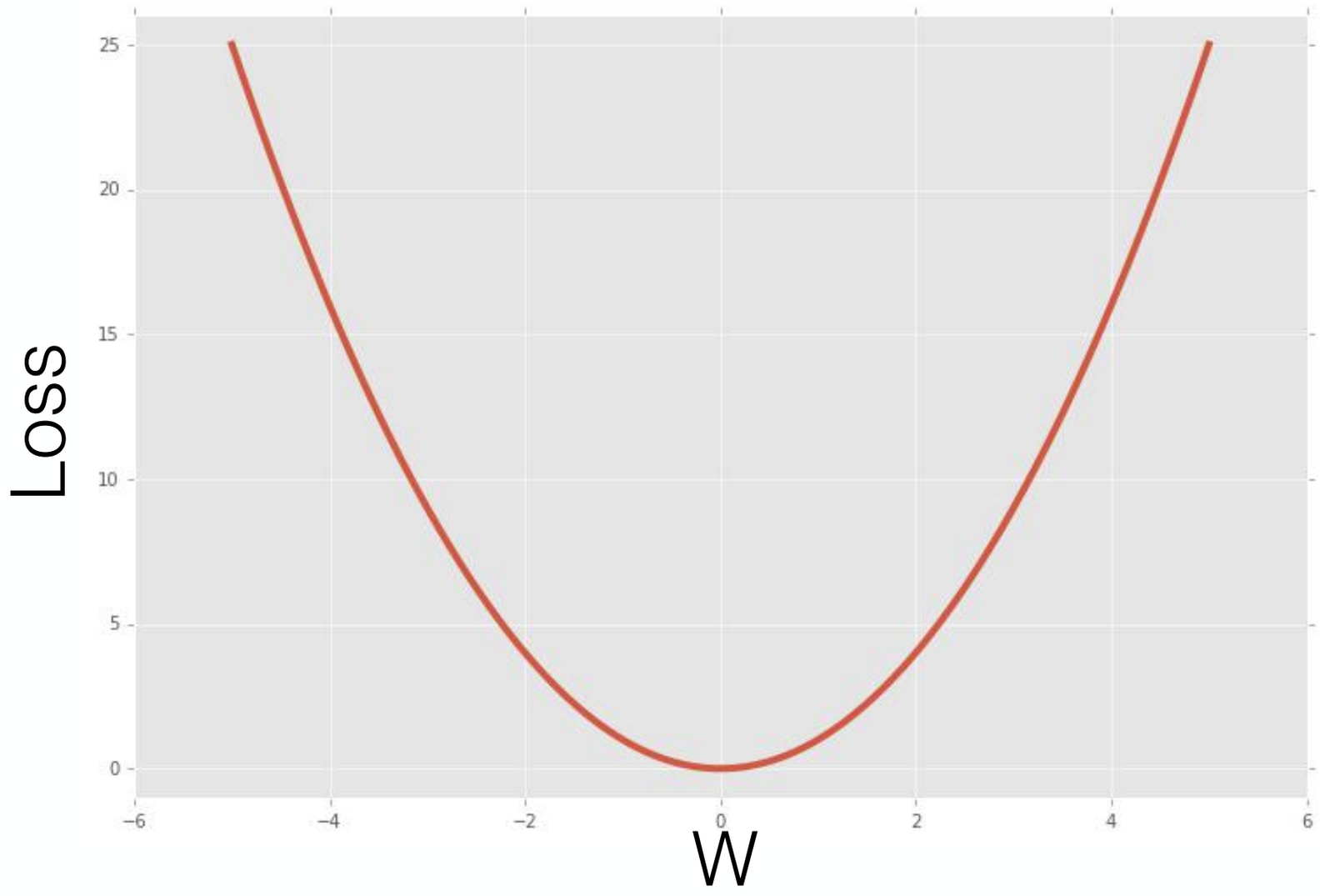
Matrix multiplication/inversion:

Coppersmith–Winograd algorithm, $O(n^{2.373})$

Gradient Descent

$$Loss(W) = \frac{1}{2}(y - XW)(y - XW)^T$$

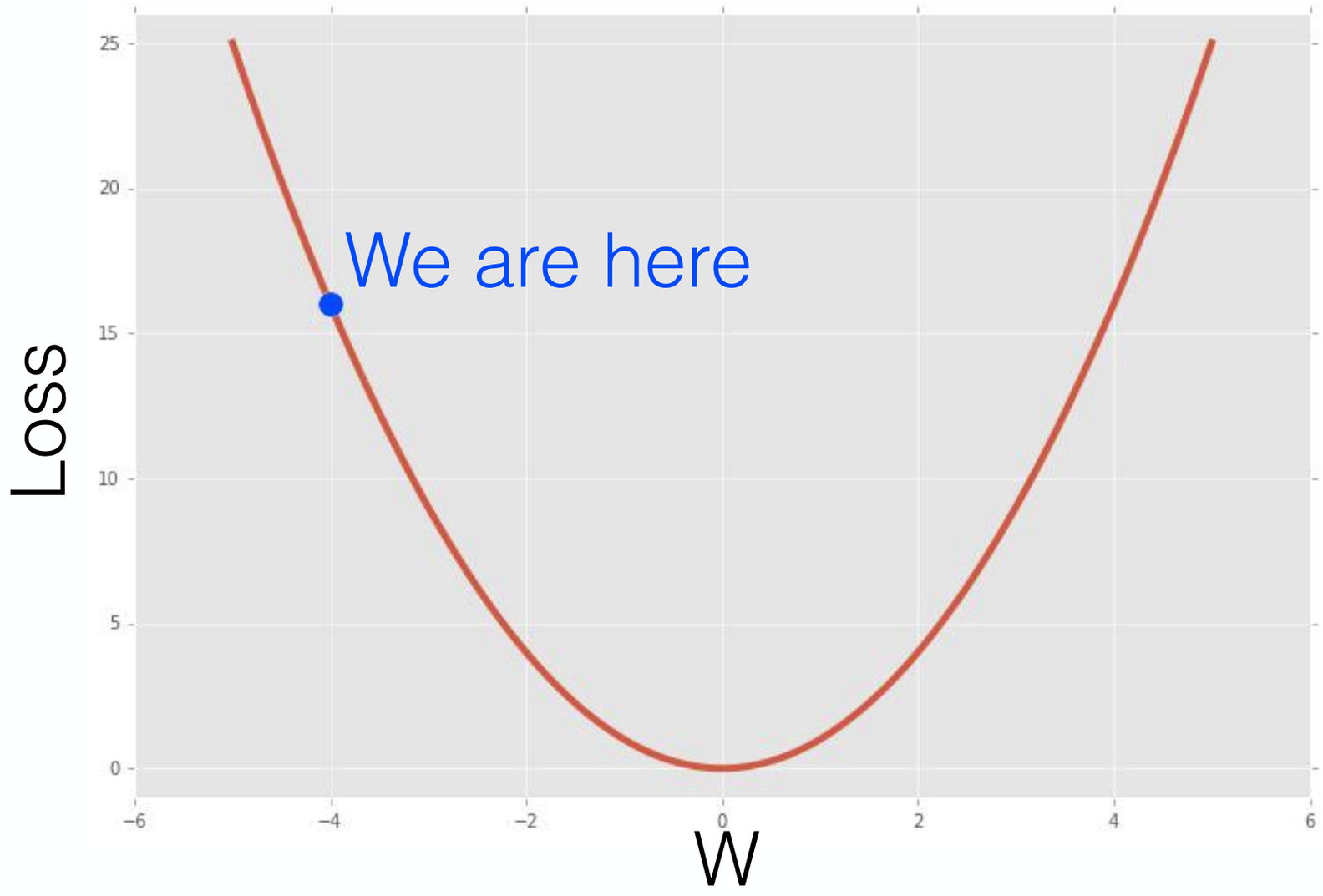
$$\frac{\partial Loss(W)}{\partial W} = ?$$



Gradient Descent

$$W_t = W_{t-1} - \alpha \nabla W_{t-1}$$

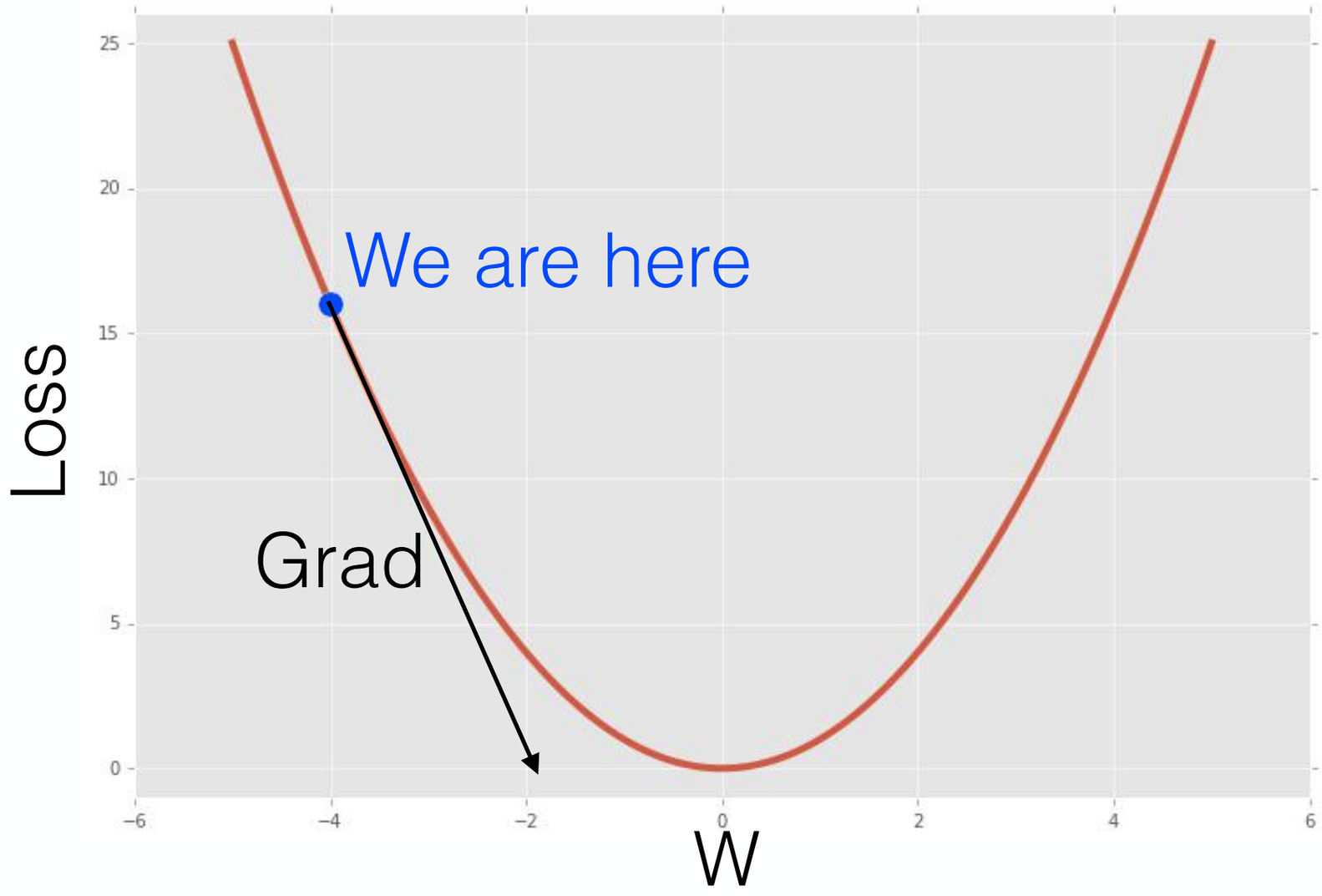
Epoch 1



Gradient Descent

$$W_t = W_{t-1} - \alpha \nabla W_{t-1}$$

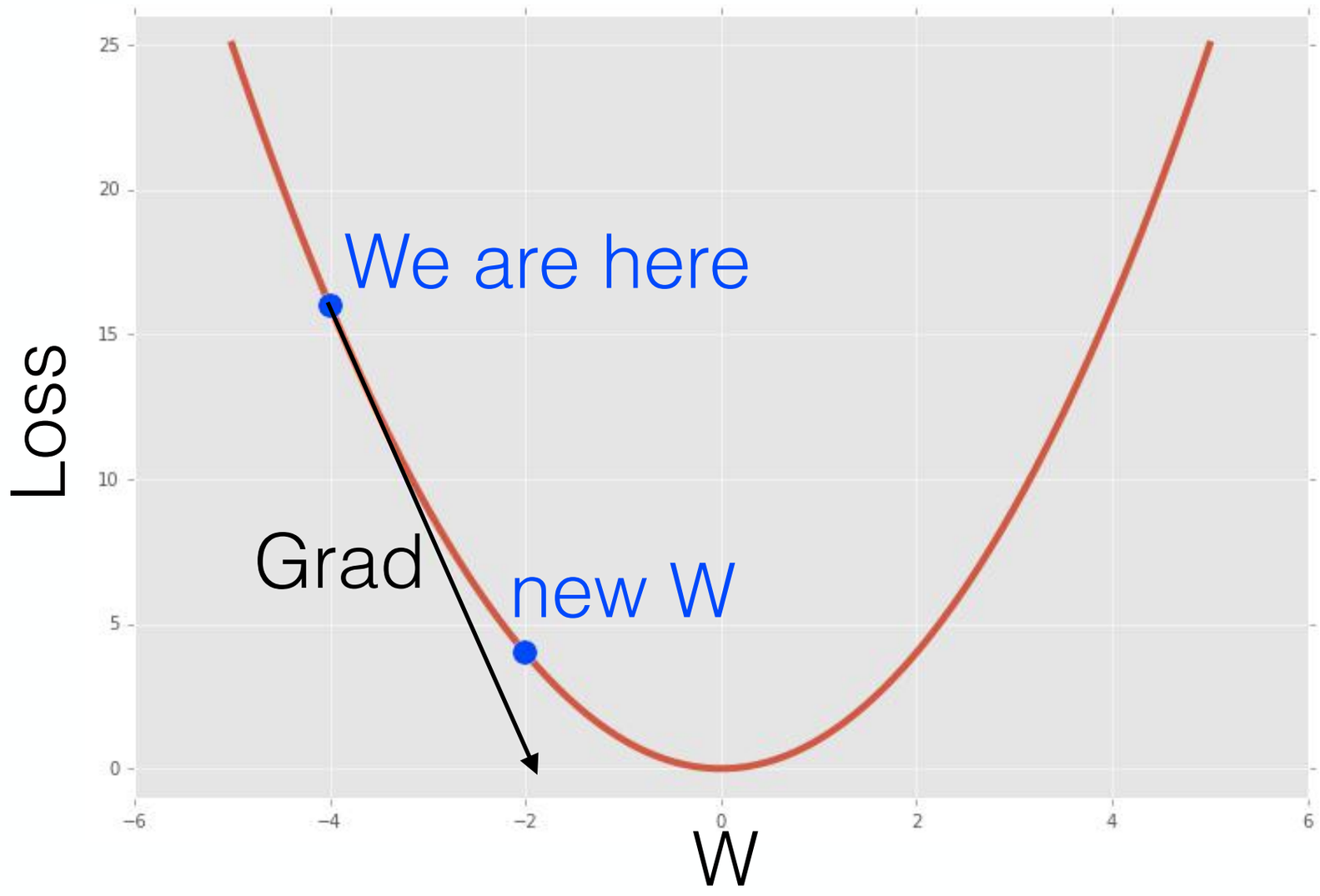
Epoch 1



Gradient Descent

$$W_t = W_{t-1} - \alpha \nabla W_{t-1}$$

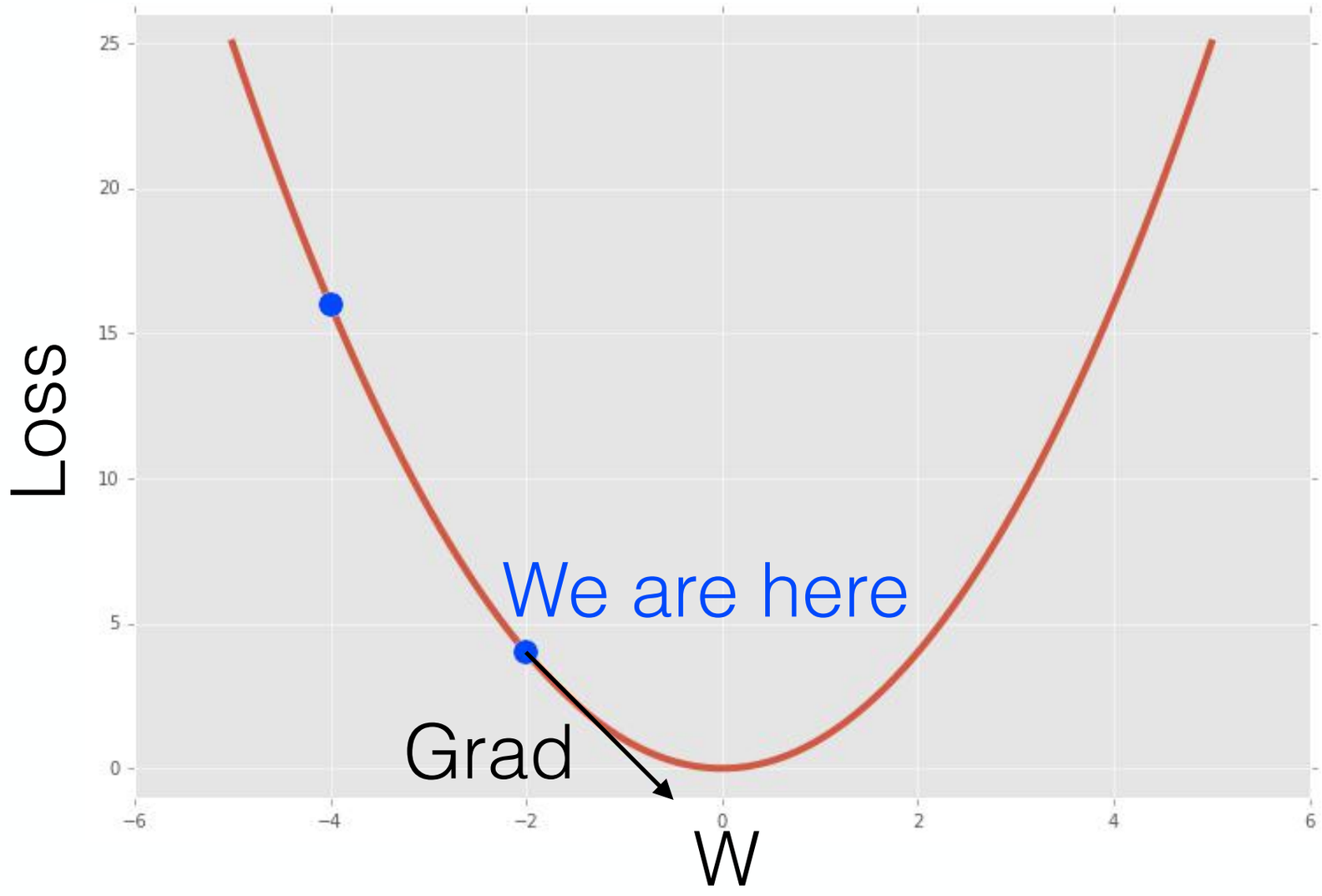
Epoch 2



Gradient Descent

$$W_t = W_{t-1} - \alpha \nabla W_{t-1}$$

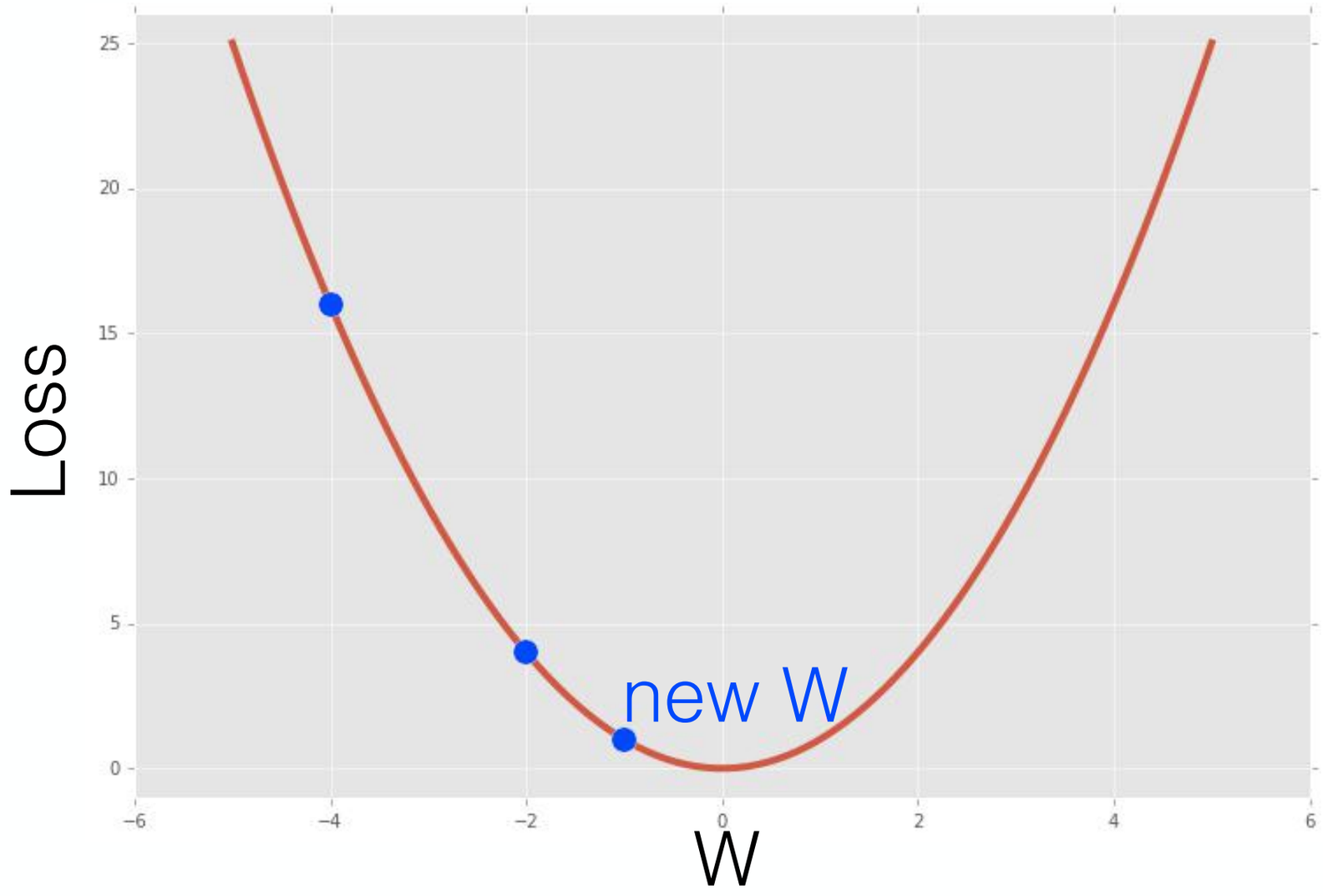
Epoch 2



Gradient Descent

$$W_t = W_{t-1} - \alpha \nabla W_{t-1}$$

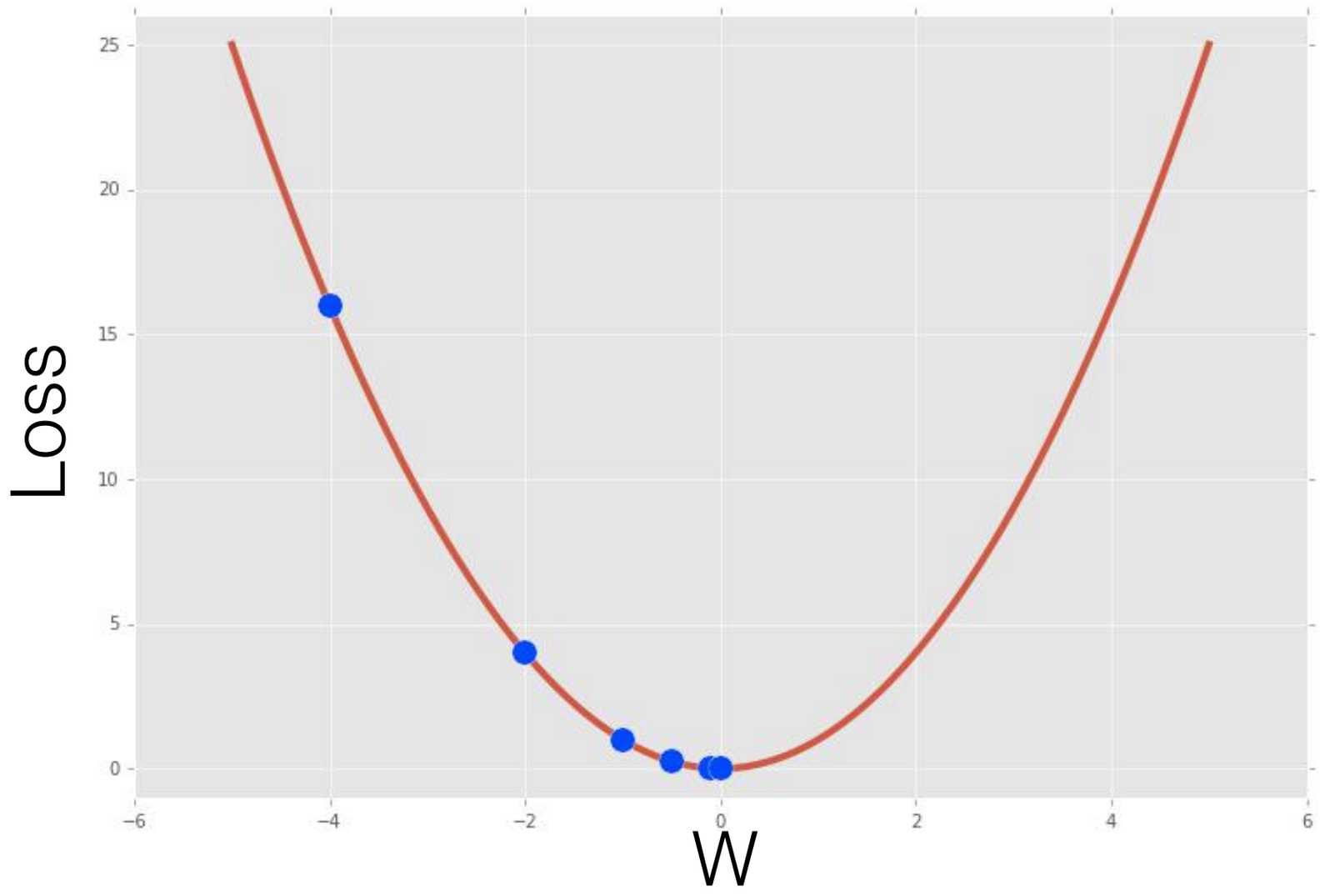
Epoch 3



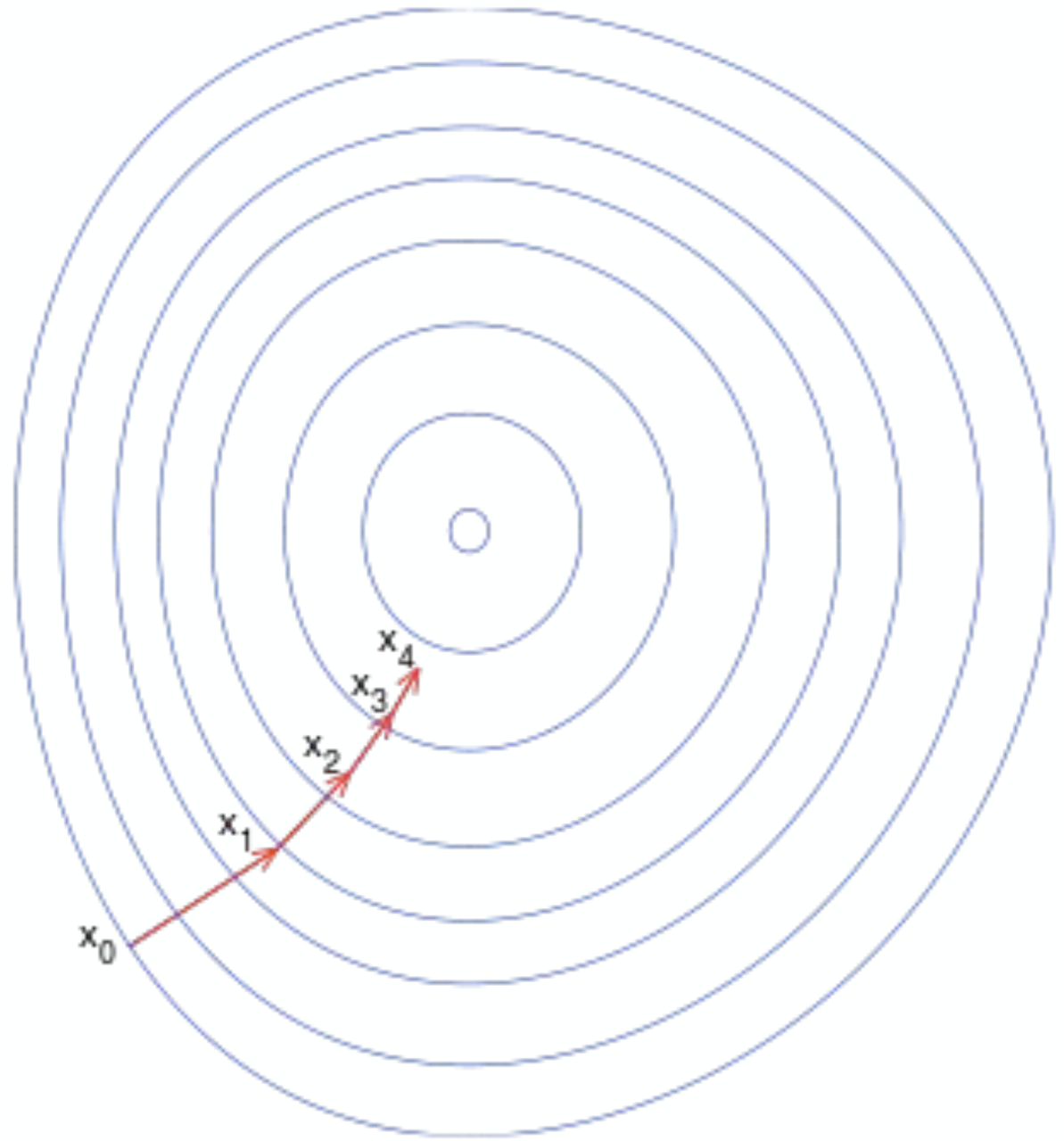
Gradient Descent

$$W_t = W_{t-1} - \alpha \nabla W_{t-1}$$

Epoch K



2d - Gradient Descent



Stochastic Gradient Descent

a.k.a. SGD

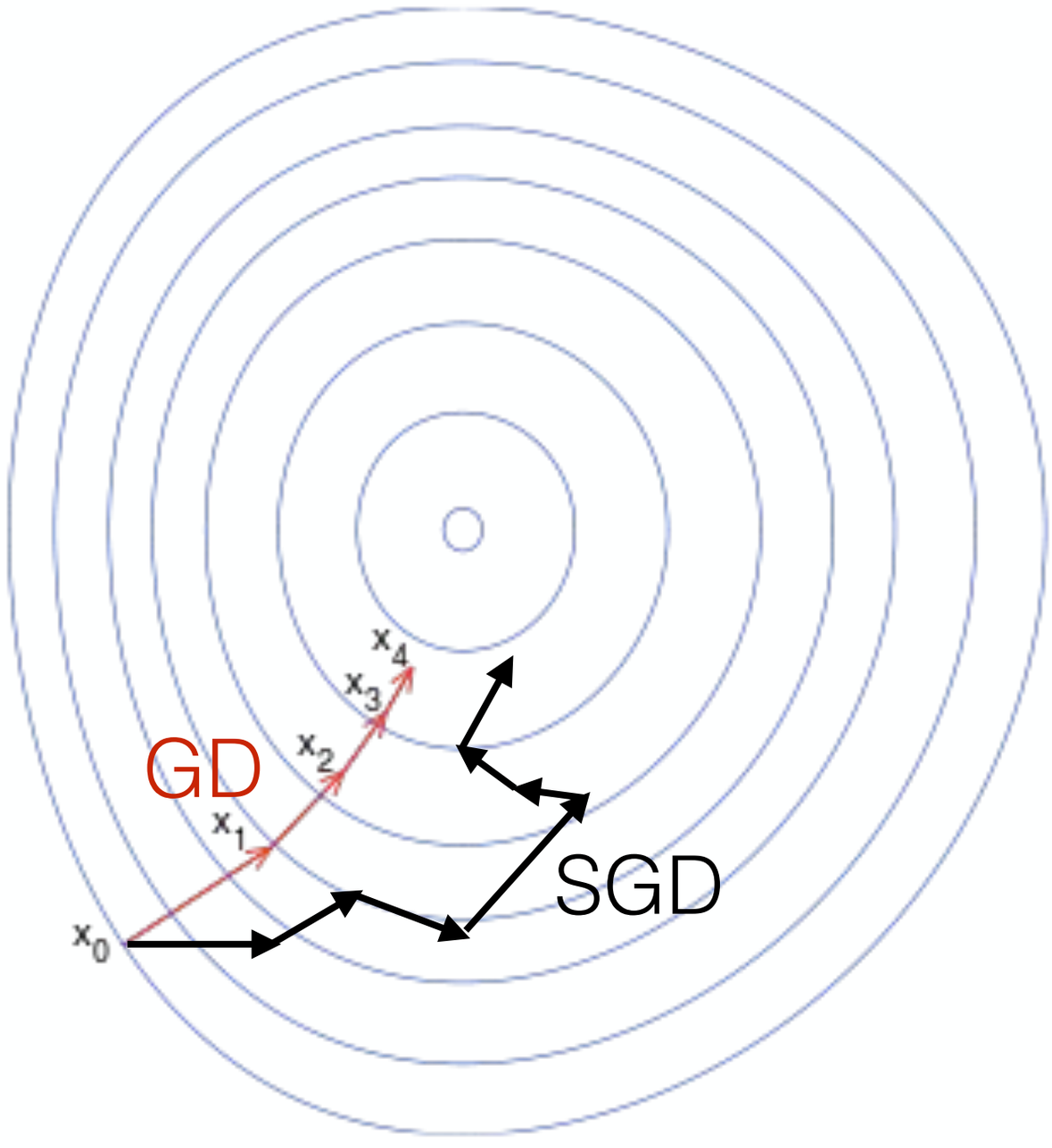
With GD you must pass through
the whole dataset to calculate one gradient!

X_1	X_2	X_3	X_4	X_5	
				1	
				1	
				1	Batch 1
				1	
				1	
				1	Batch 2
				1	
				1	
				1	
				1	Batch 3
				1	
				1	

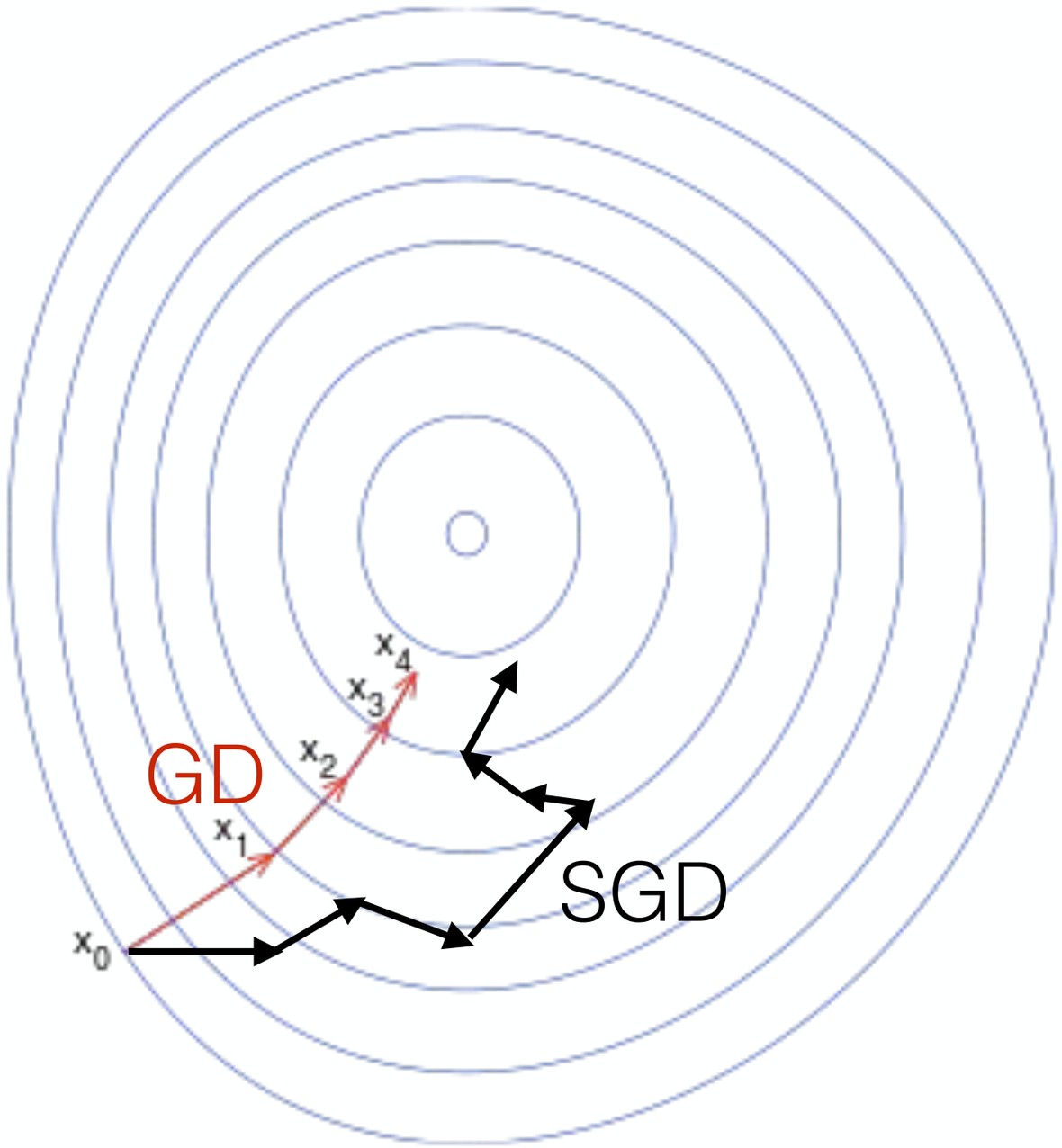
- Calculate batch gradient
- Update weights
- Repeat

That's one epoch!

GD vs SGD

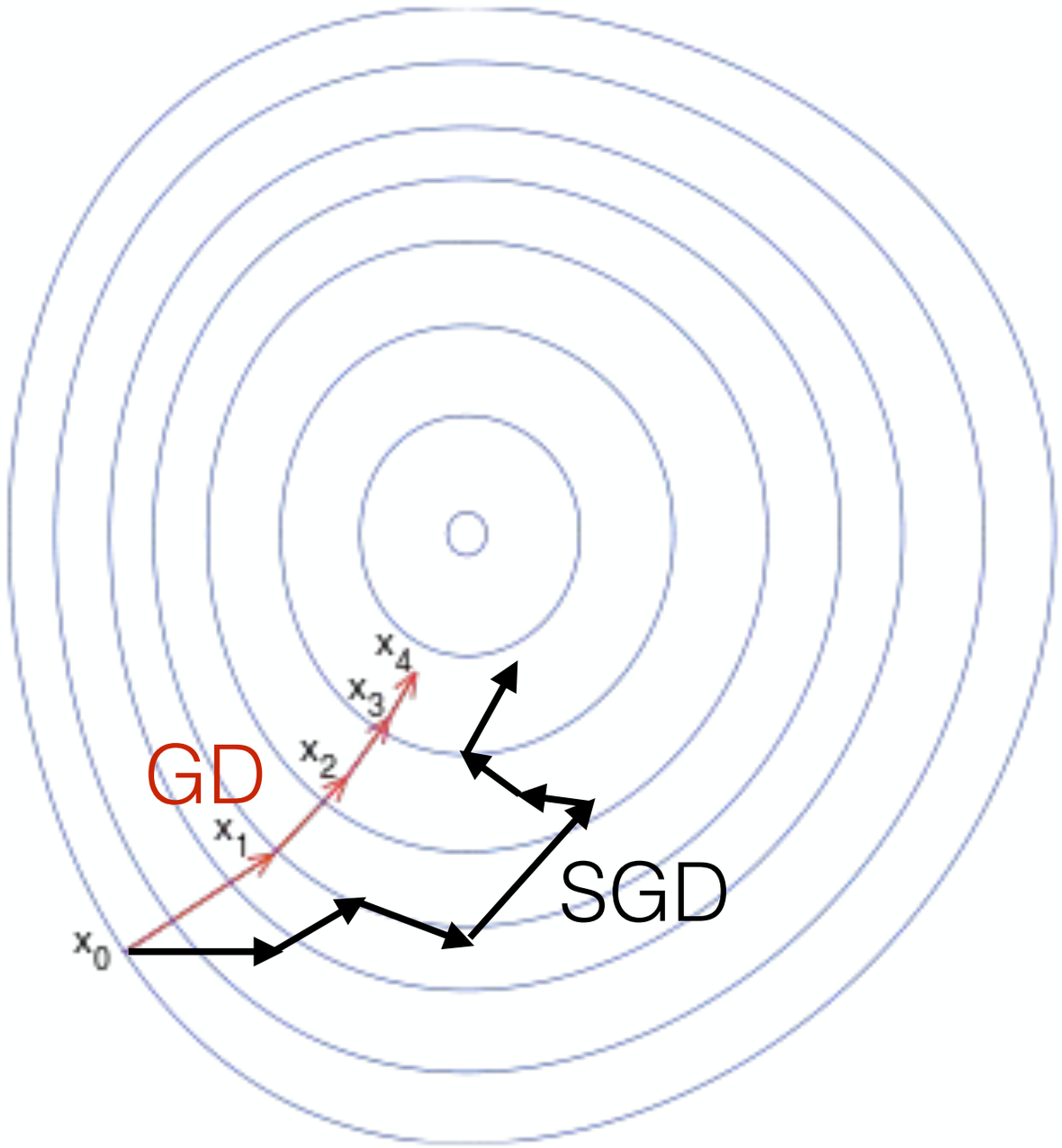


GD vs SGD



- (as GD) No guarantee about global minimum
- (as GD) No guarantee that solution would be found in finite time
- (as GD) No guarantee about convergence at all
- No guarantee about moving in correct direction

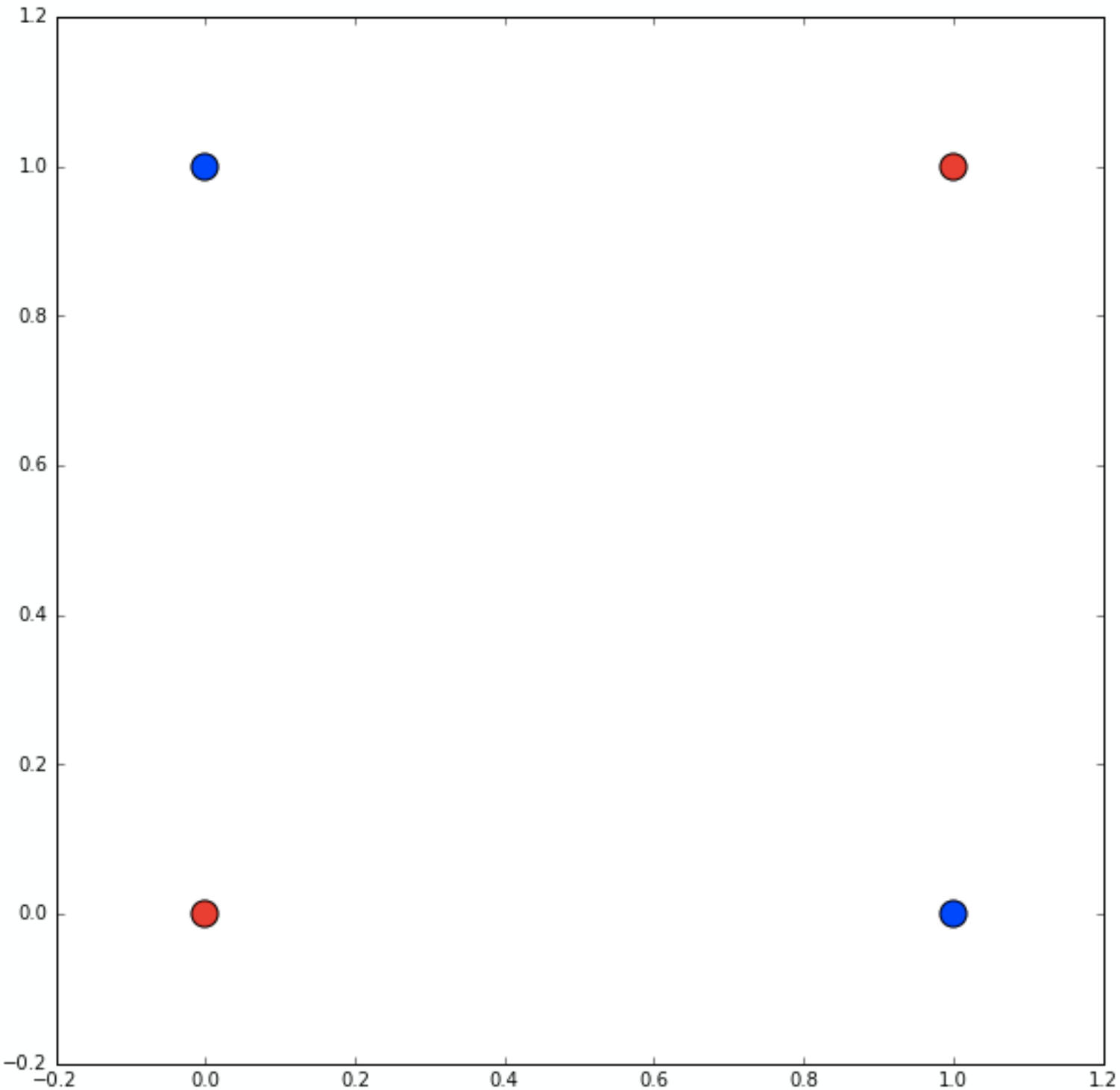
GD vs SGD



- (as GD) No guarantee about global minimum
- (as GD) No guarantee that solution would be found in finite time
- (as GD) No guarantee about convergence at all
- No guarantee about moving in correct direction

GD: O(n)
SGD: O(1)

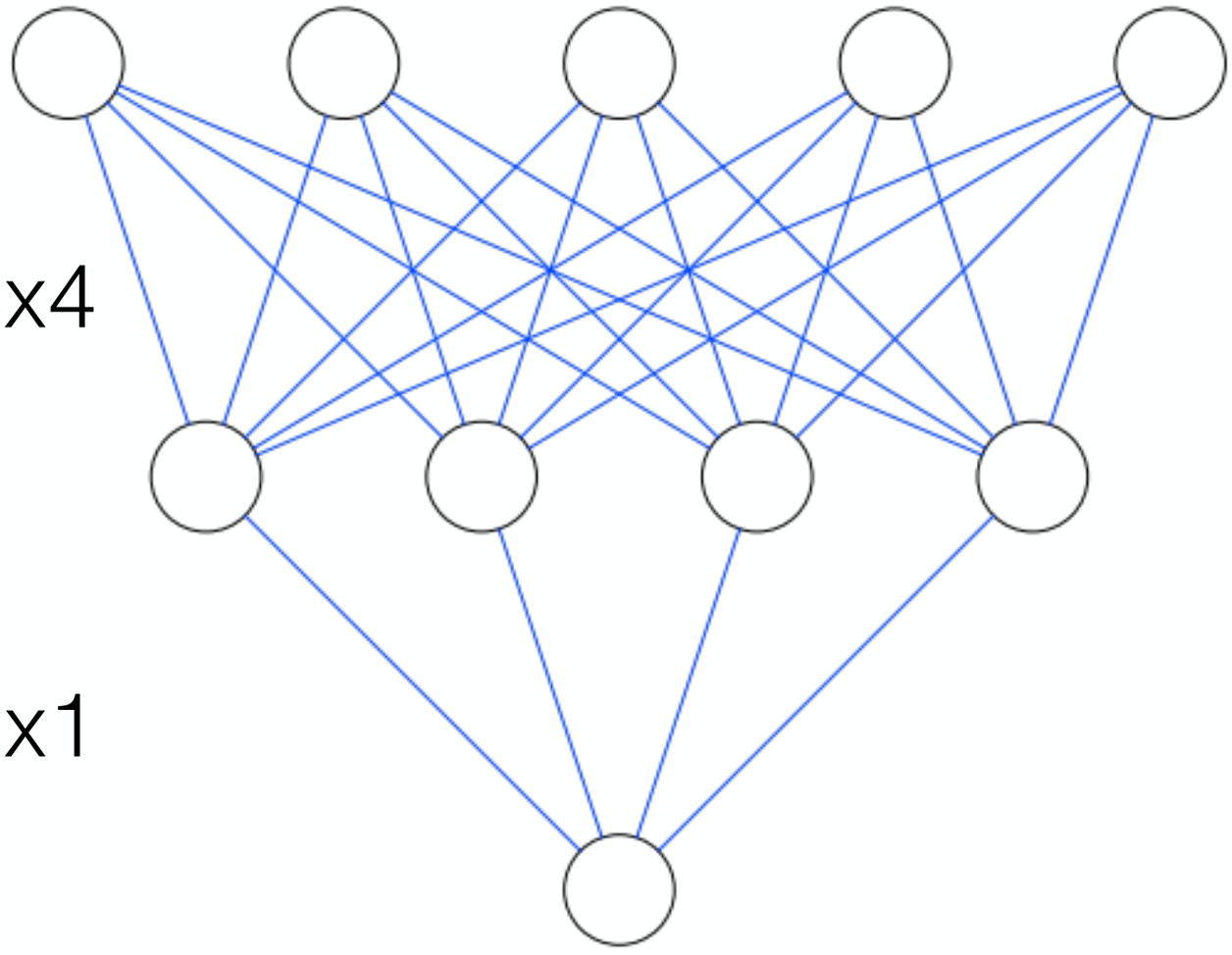
XOR problem



Perceptron

Perceptron model

Published in 1957, implemented in 1960



Input layer

W_1 - matrix 5x4

Hidden layer

W_2 - matrix 4x1

Output

Output calculation

Linear model: $\hat{y} = XW$

Output calculation

Linear model: $\hat{y} = XW$

Hidden: $h = XW_1$

Perceptron:

Output calculation

Linear model: $\hat{y} = XW$

Perceptron:

Hidden: $h = XW_1$

Output: $\hat{y} = hW_2$

Output calculation

Linear model: $\hat{y} = XW$

Hidden: $h = XW_1$

Perceptron:

Output: $\hat{y} = hW_2$

Linear again! damn.

But: $\hat{y} = hW_2 = XW_1W_2 = XA$

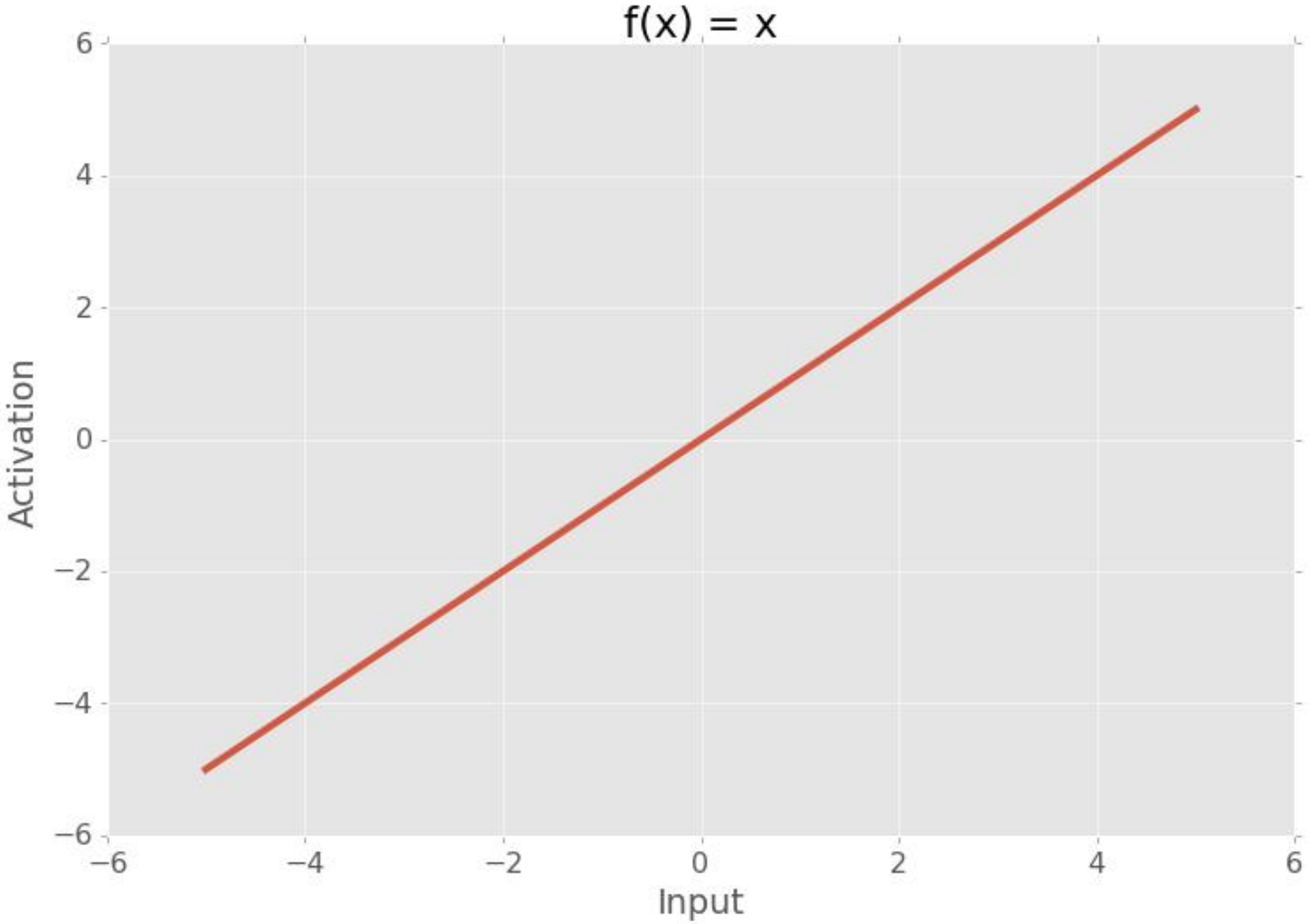
Idea

Add some non-linear activation in the hidden layer

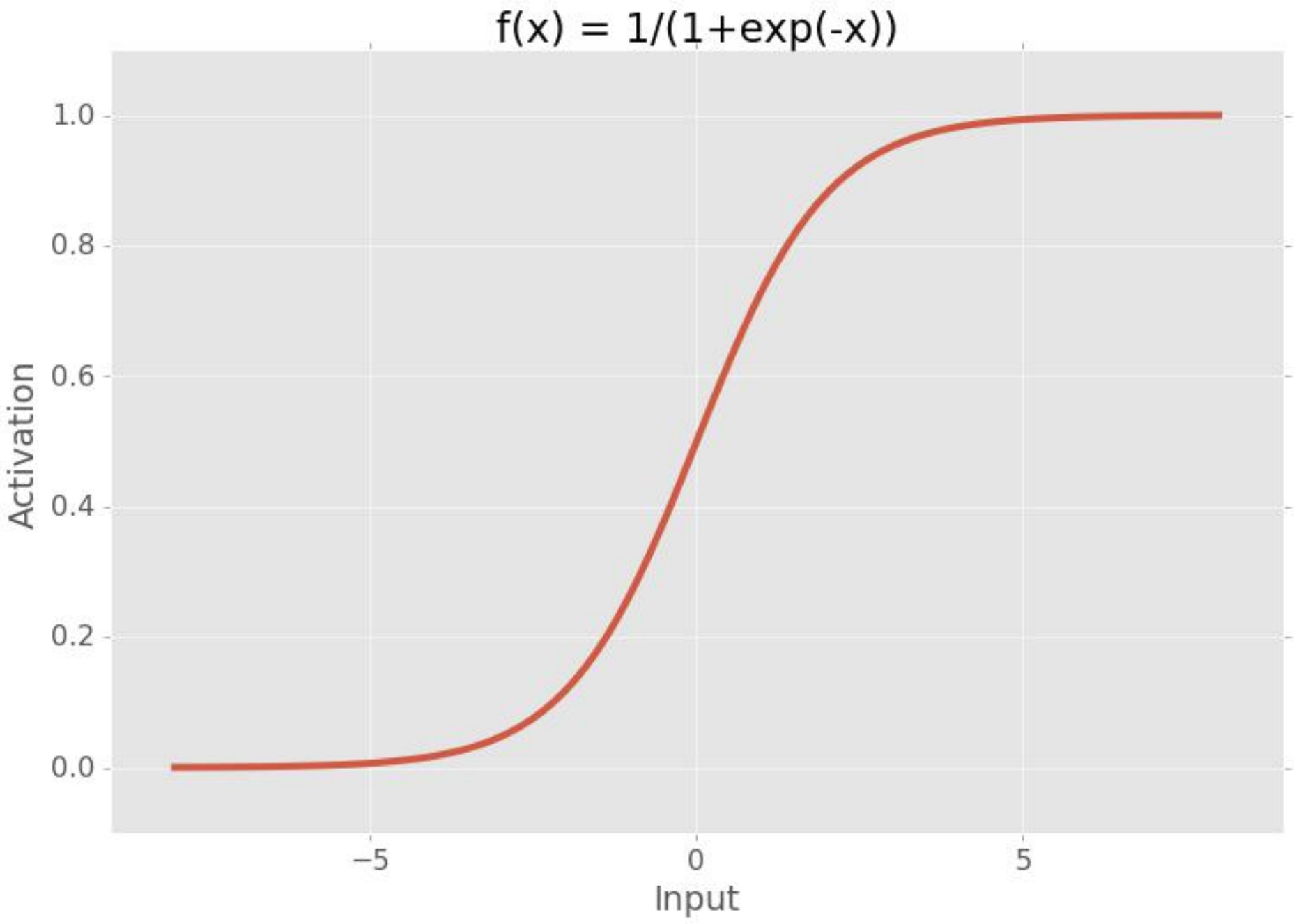
$$h = f(XW_1)$$

$$\hat{y} = hW_2 = f(XW_1)W_2 \neq XA$$

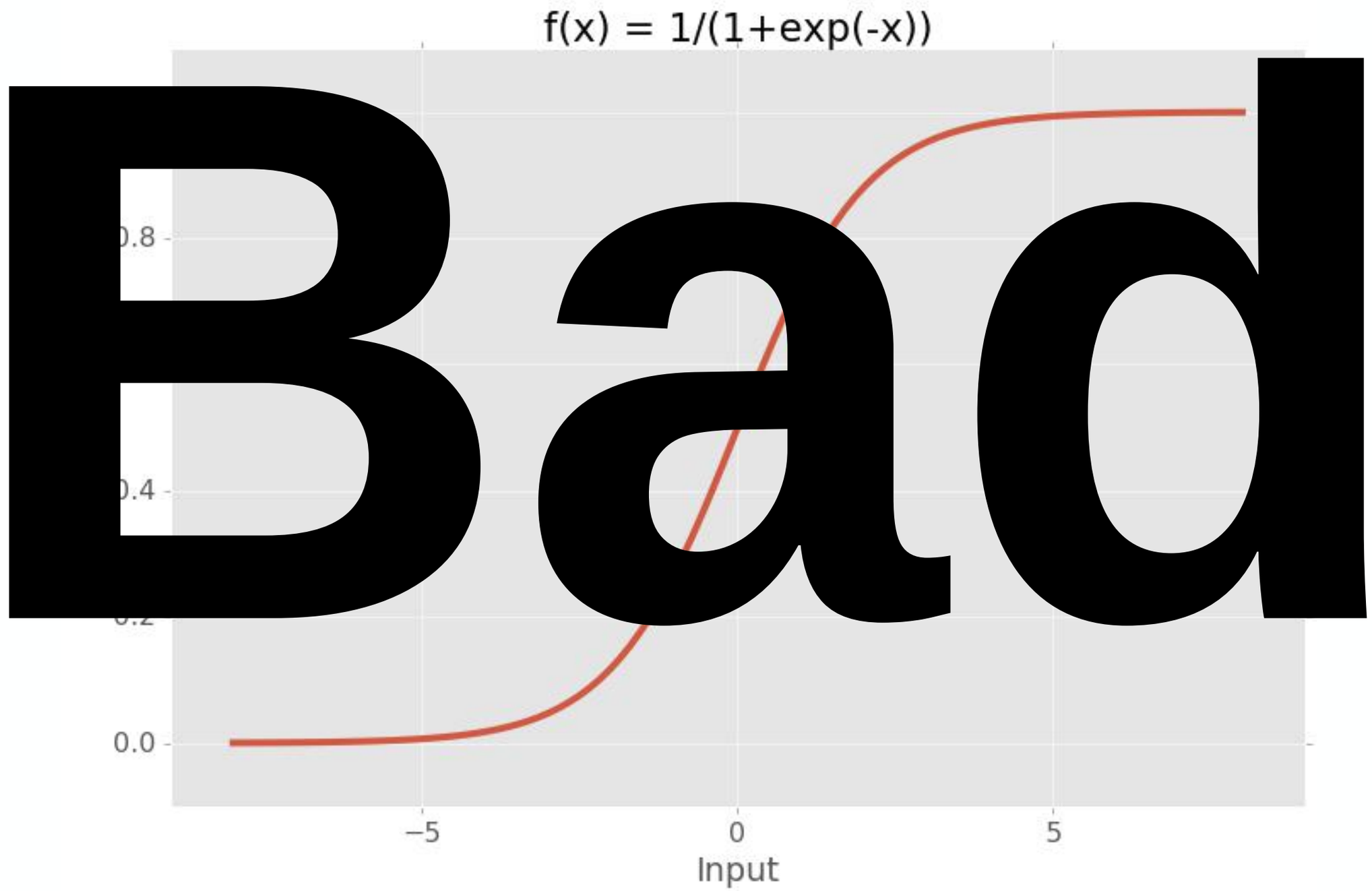
Activation function



Sigmoid

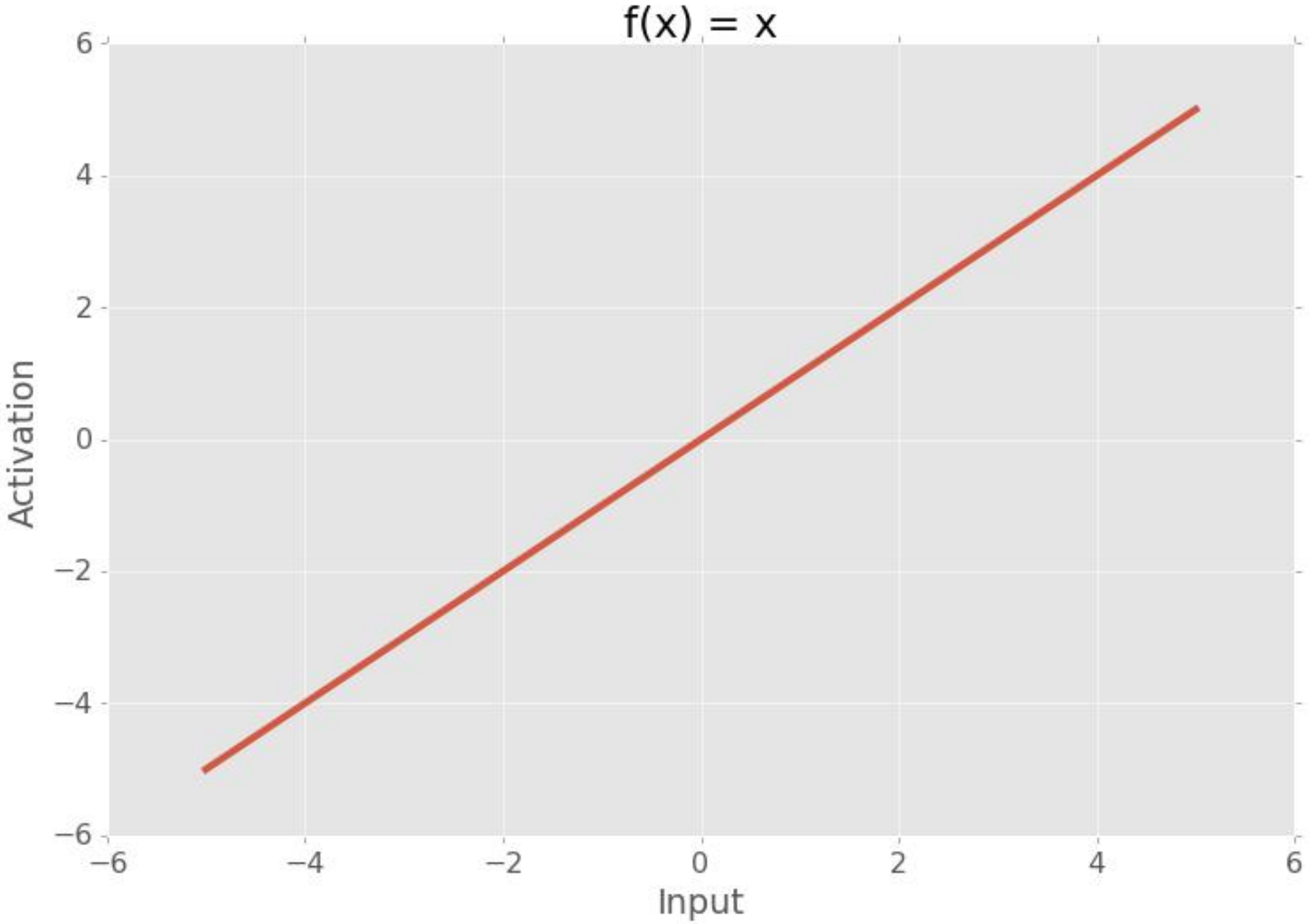


Sigmoid

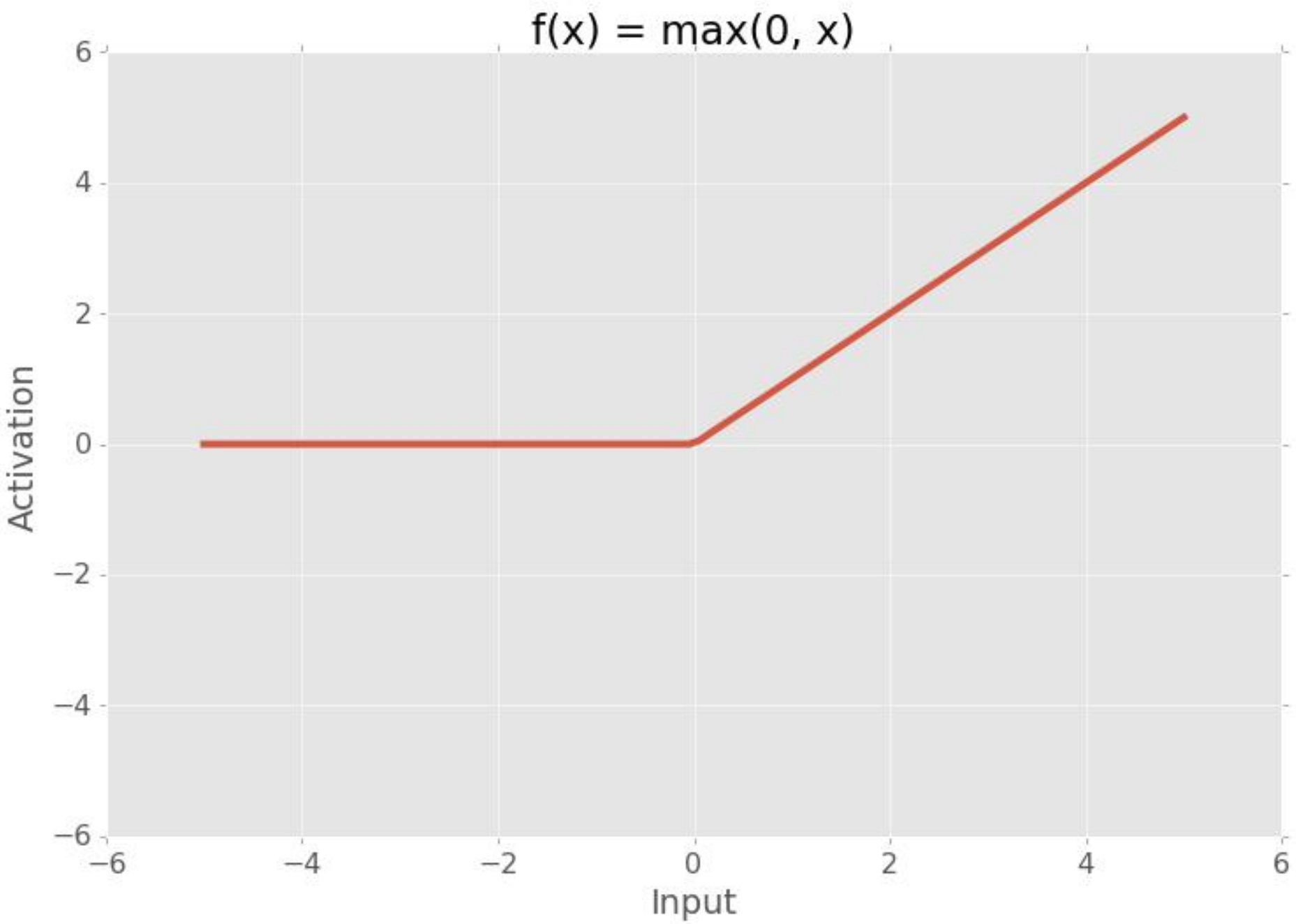


Bad

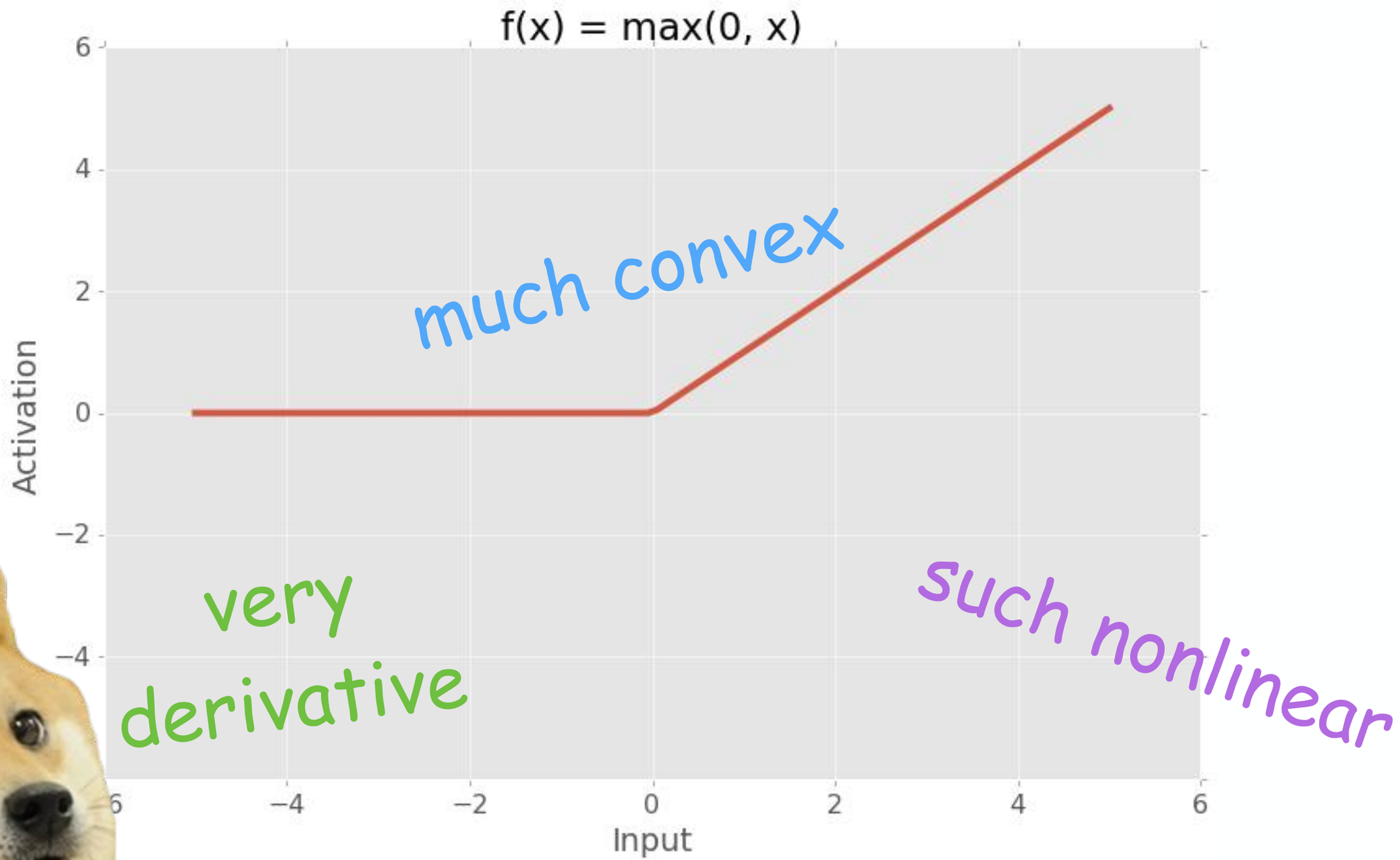
From linear activation …



… to nonlinear



ReLU


f(x) = max(0, x)
Activation
Input
6
4
2
0
−2
−4
−6
−4
−2
0
2
4
6
much convex
very
derivative
such nonlinear

How to train perceptron?

$$Loss(W1, W2) = \frac{1}{2}(y - f(XW_1)W_2)^2$$

The key point is chain rule:

$$f(g(x))' = f'(g(x)) * g(x)'$$

How to train perceptron?

$$Loss(W1, W2) = \frac{1}{2}(y - f(XW_1)W_2)^2$$

The key point is chain rule:

$$f(g(x))' = f'(g(x)) * g(x)'$$

$$\frac{\partial Loss}{\partial W_2} = -(y - f(XW_1)W_2)f(XW_1)$$

$$\frac{\partial Loss}{\partial W_1} = -(y - f(XW_1)W_2)W_2 f'(XW_1)W_1$$

The Perceptron Convergence Theorem

Frank Rosenblatt, 1965

- Perceptron can solve **any** problem
- Perceptron will converge in finite time
- Result is independent of initial weights

Great. Why to implement anything else?

Going Deeper

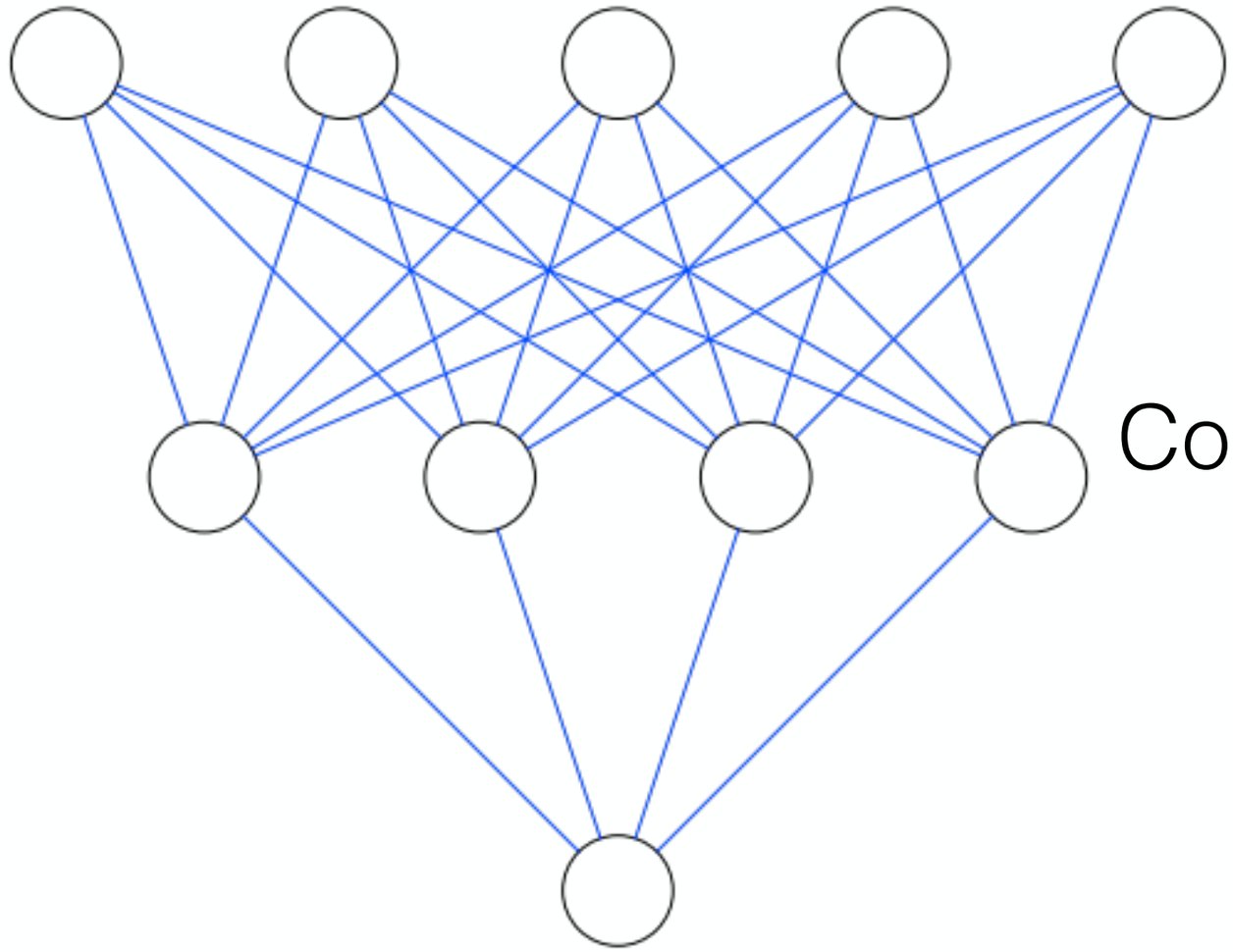
Motivation: computational cost

Wide enough perceptron can solve any problem, but the cost will be very high

It’s cheaper to go deeper than wider

Motivation

Input features



Complex features,
e.g. $X_1 + X_2$

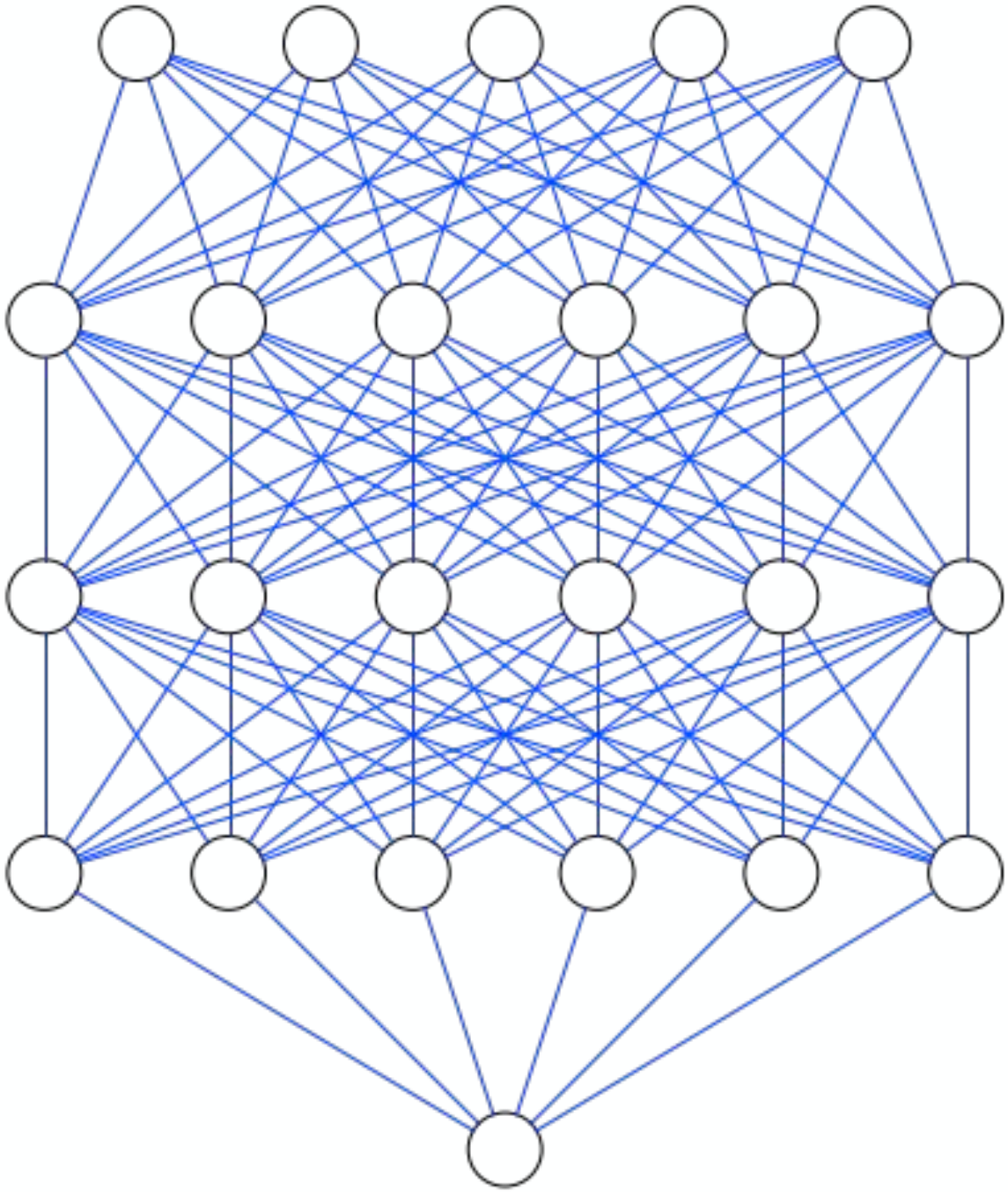
Linear model on complex features

Input features

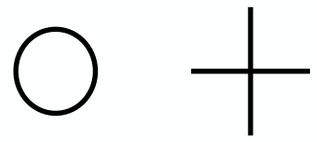
Complex features

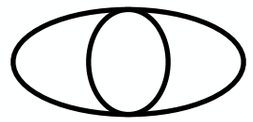
Very
Complex features

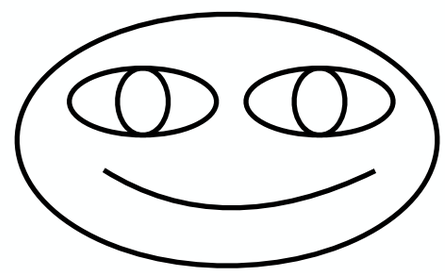
Extremely
Complex features



E.g.







Face recognition system

How to train?

Forward propagate your data:

$$X \Rightarrow XW_1 \Rightarrow f(XW_1) \Rightarrow f(XW_1)W_2 \Rightarrow ... \Rightarrow \hat{y}$$

How to train?

Forward propagate your data:

$$X \Rightarrow XW_1 \Rightarrow f(XW_1) \Rightarrow f(XW_1)W_2 \Rightarrow ... \Rightarrow \hat{y}$$

Calculate loss:

$$Loss = \frac{1}{2}(y - \hat{y})^2$$

How to train?

Forward propagate your data:

$$X \Rightarrow XW_1 \Rightarrow f(XW_1) \Rightarrow f(XW_1)W_2 \Rightarrow ... \Rightarrow \hat{y}$$

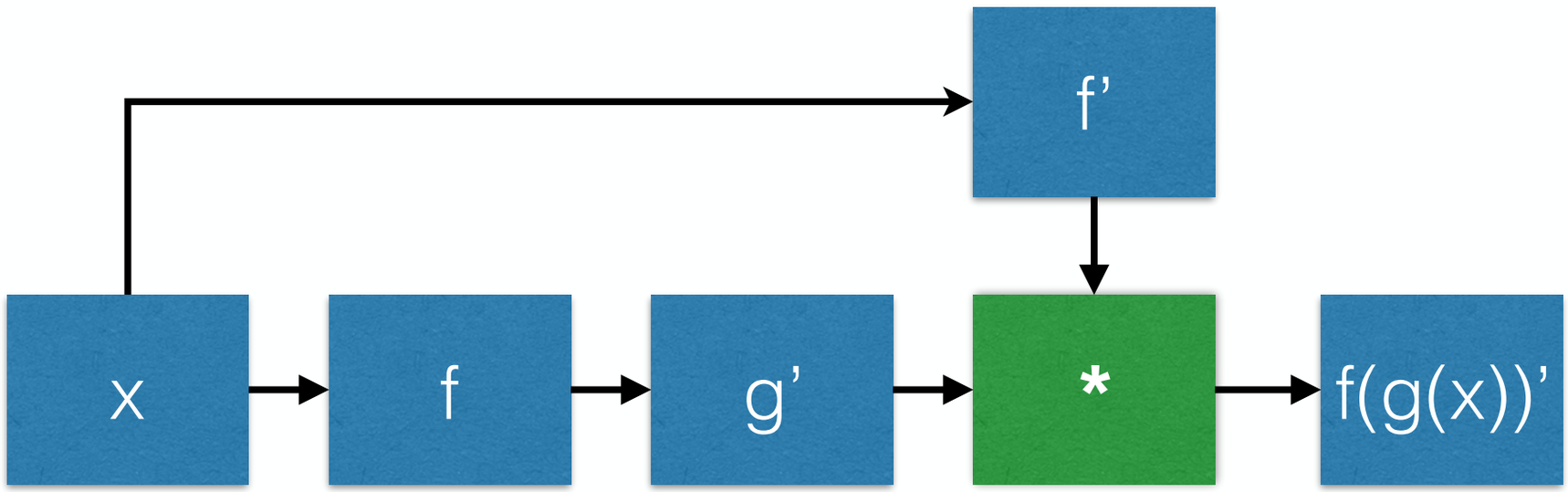
Calculate loss:

$$Loss = \frac{1}{2}(y - \hat{y})^2$$

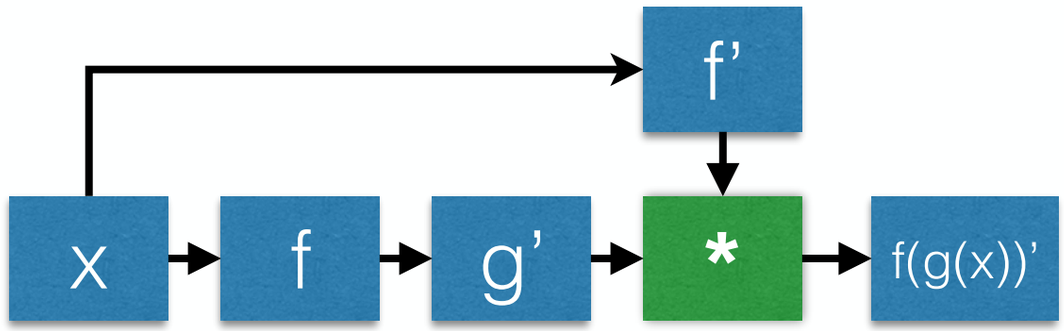
Calculate derivatives:

As always, chain rule

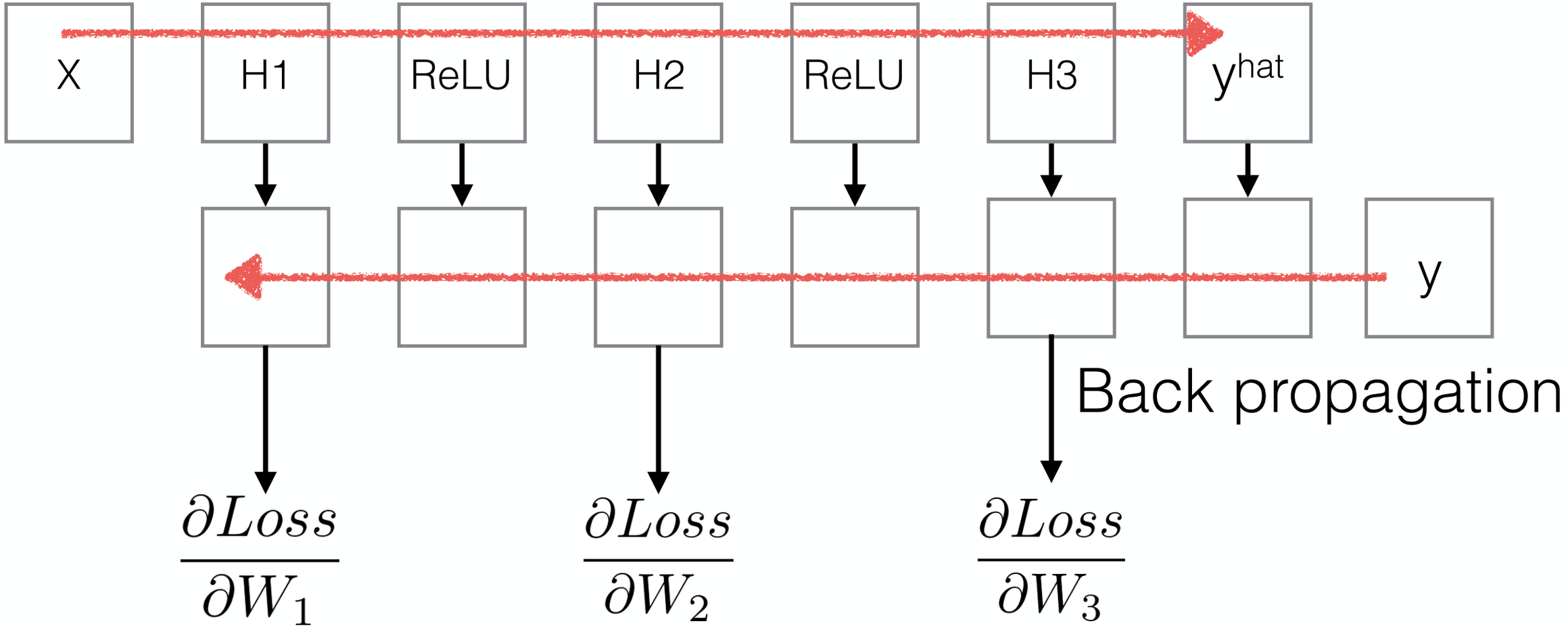
So much of chain rule



So much of chain rule

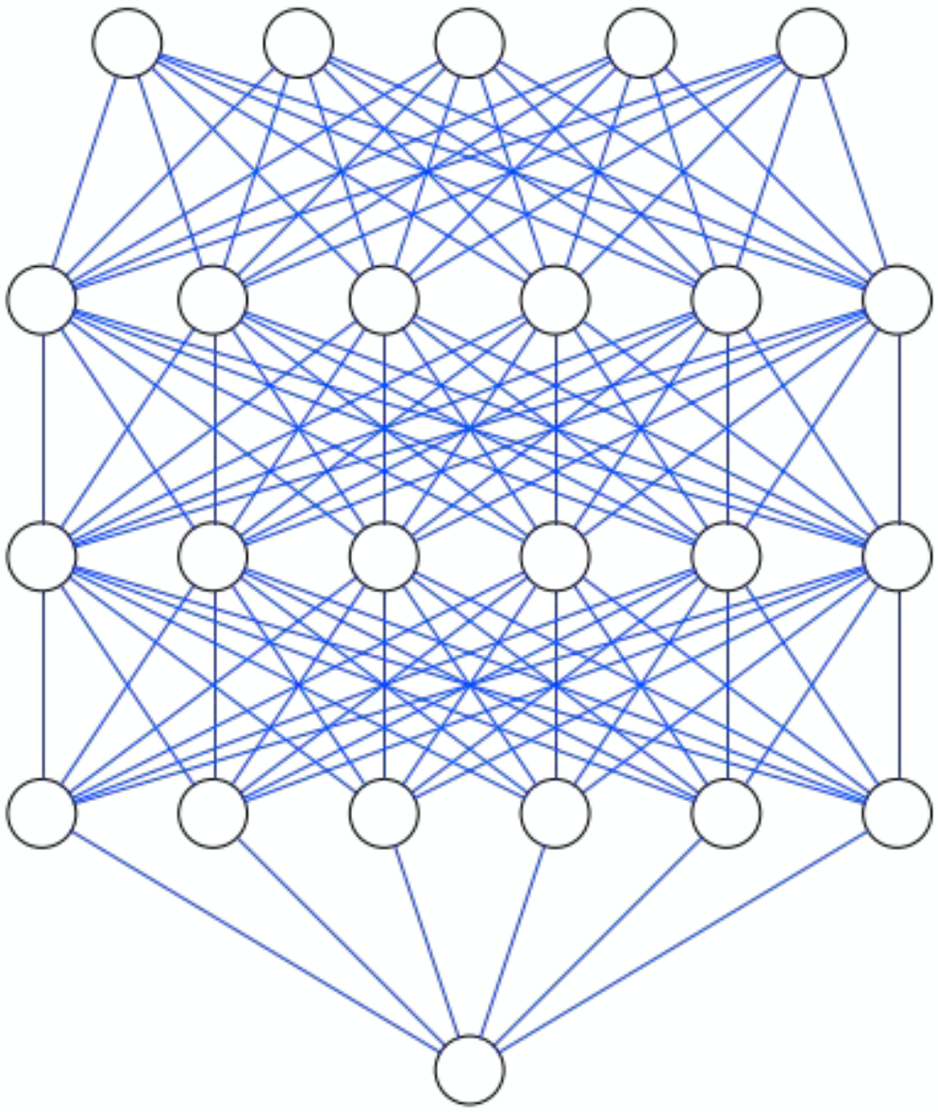


Forward propagation



Modern visualisation

Every layer is just a function $f(X)$

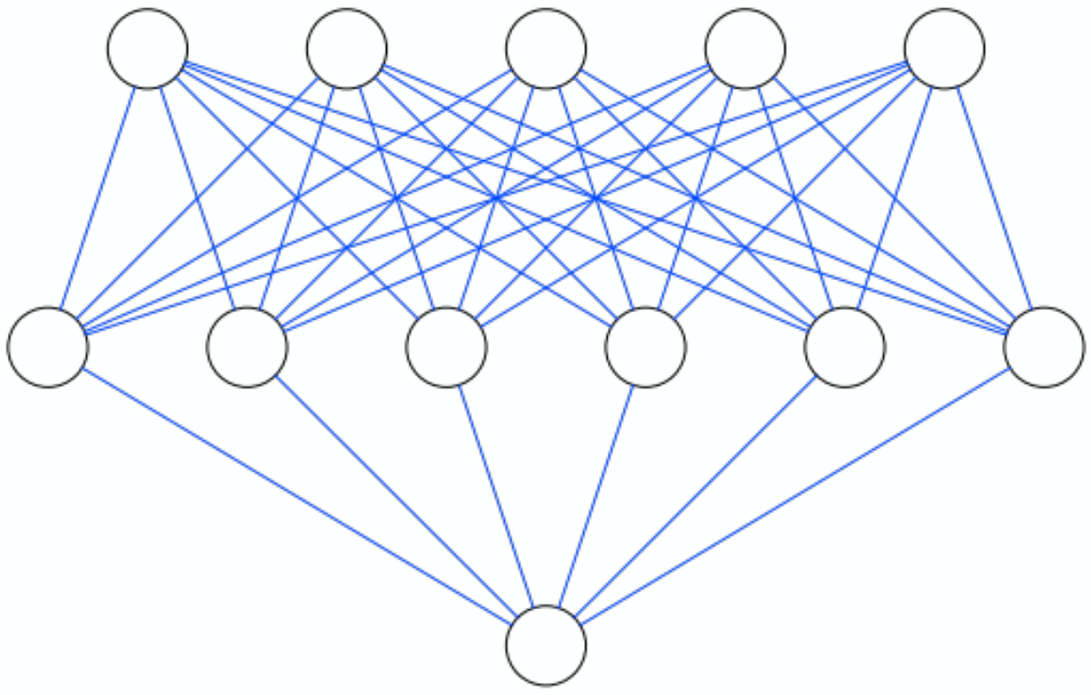


Input layer	
Fully connected layer (FC)	$XW + b$
ReLU	$max(0, X)$
FC	$XW + b$
ReLU	$max(0, X)$
FC	$XW + b$
Output	

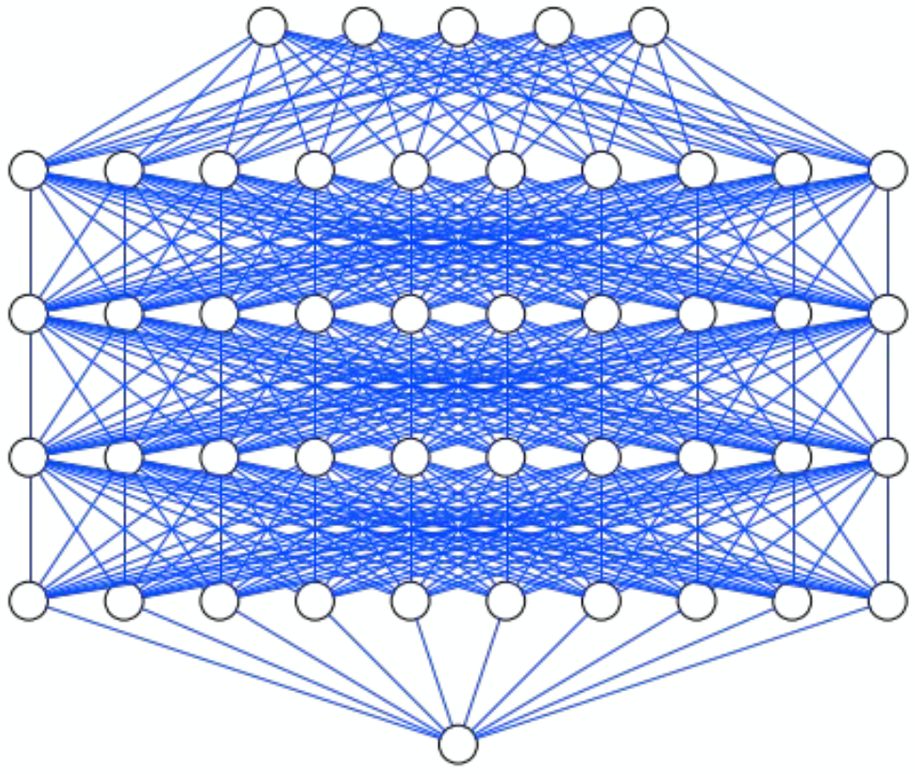
Non-convex

- With great number of weights comes great number of local minimums. Correct?

- In which case it should be the problem: in the shallow or deep network?



36 weights



360 weights

Non-convex

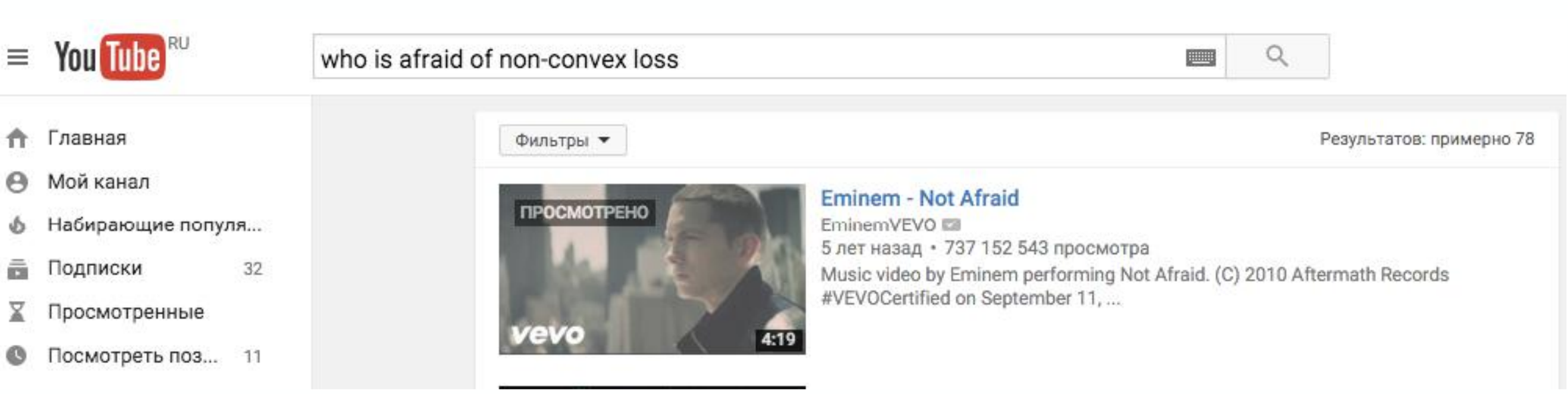
- Great number of dimensions allow you to pass by any “mountains” in the loss function!

- Yann LeCun “Who is afraid of non-convex loss functions”

Non-convex

- Great number of dimensions allow you to pass by any “mountains” in the loss function!

- Yann LeCun “Who is afraid of non-convex loss functions”

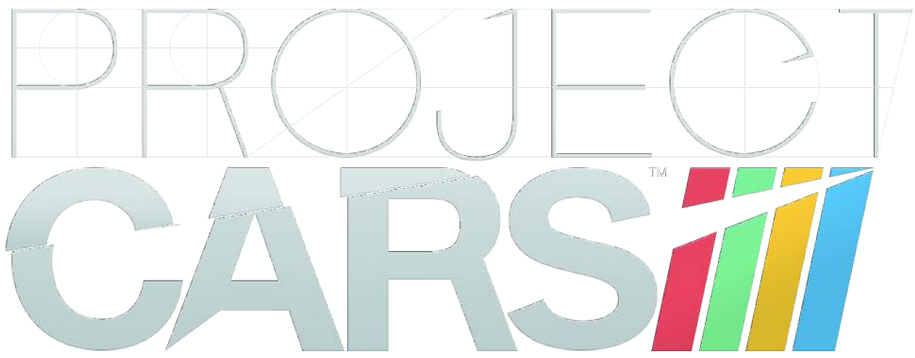


Why is it possible?

I mean, perceptron was crated in 1960

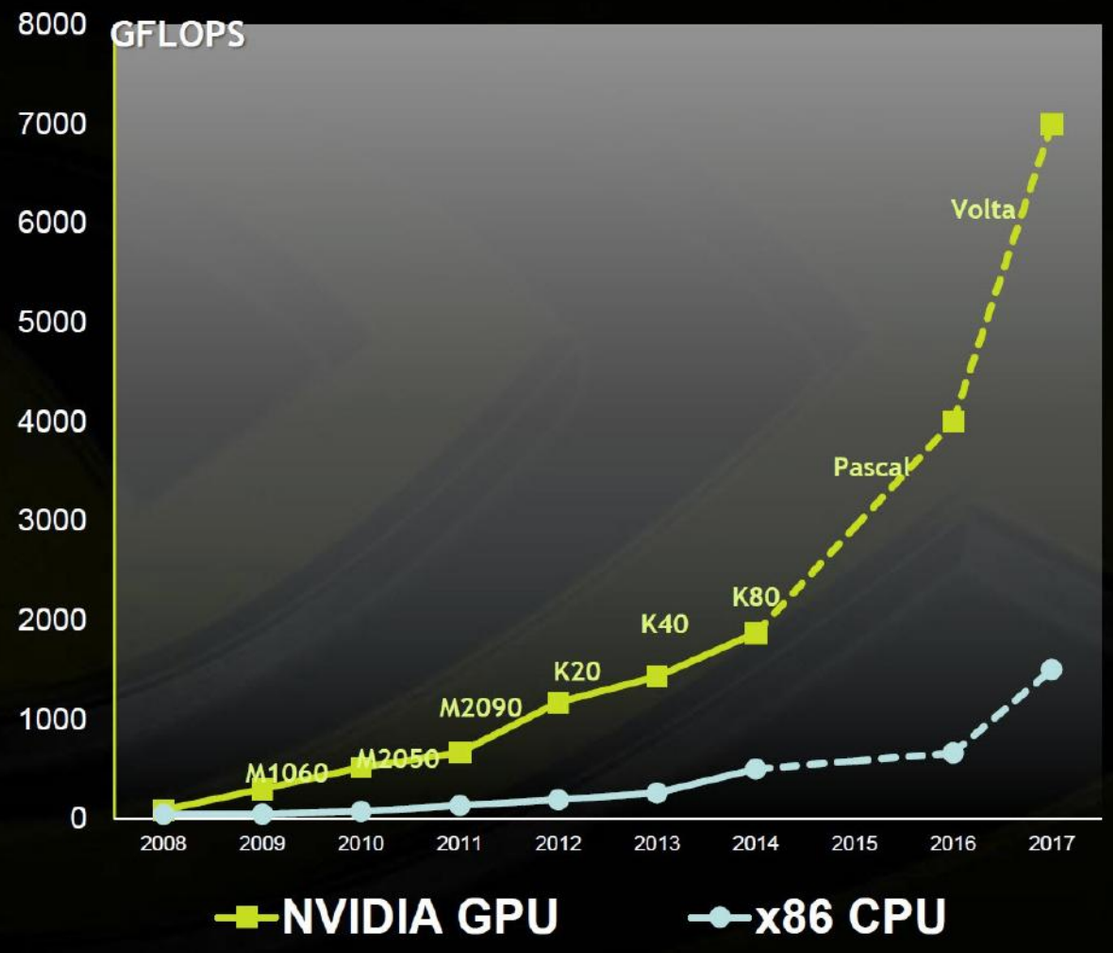
Deep neural nets became popular only in mid-2000s

Why is it possible?

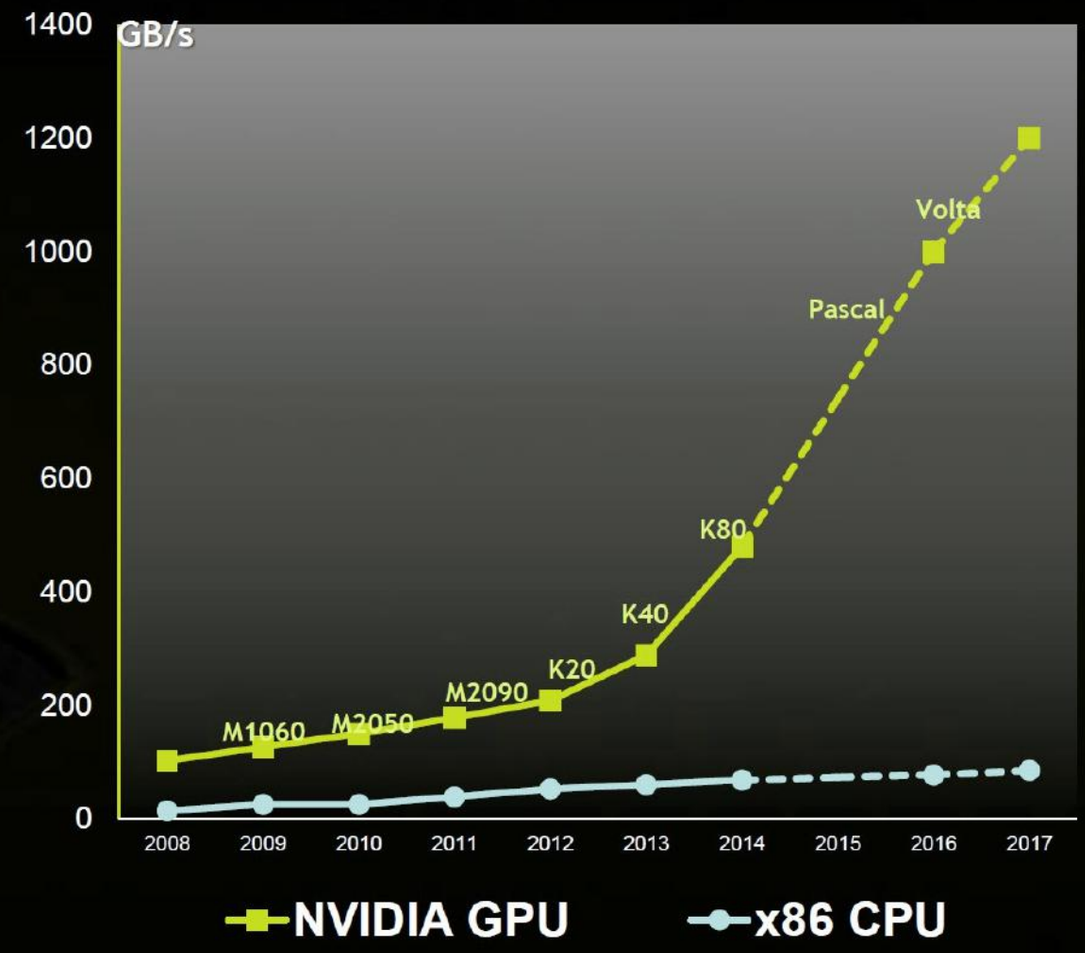


GPU Motivation (I): Performance Trends

Peak Double Precision FLOPS



Peak Memory Bandwidth



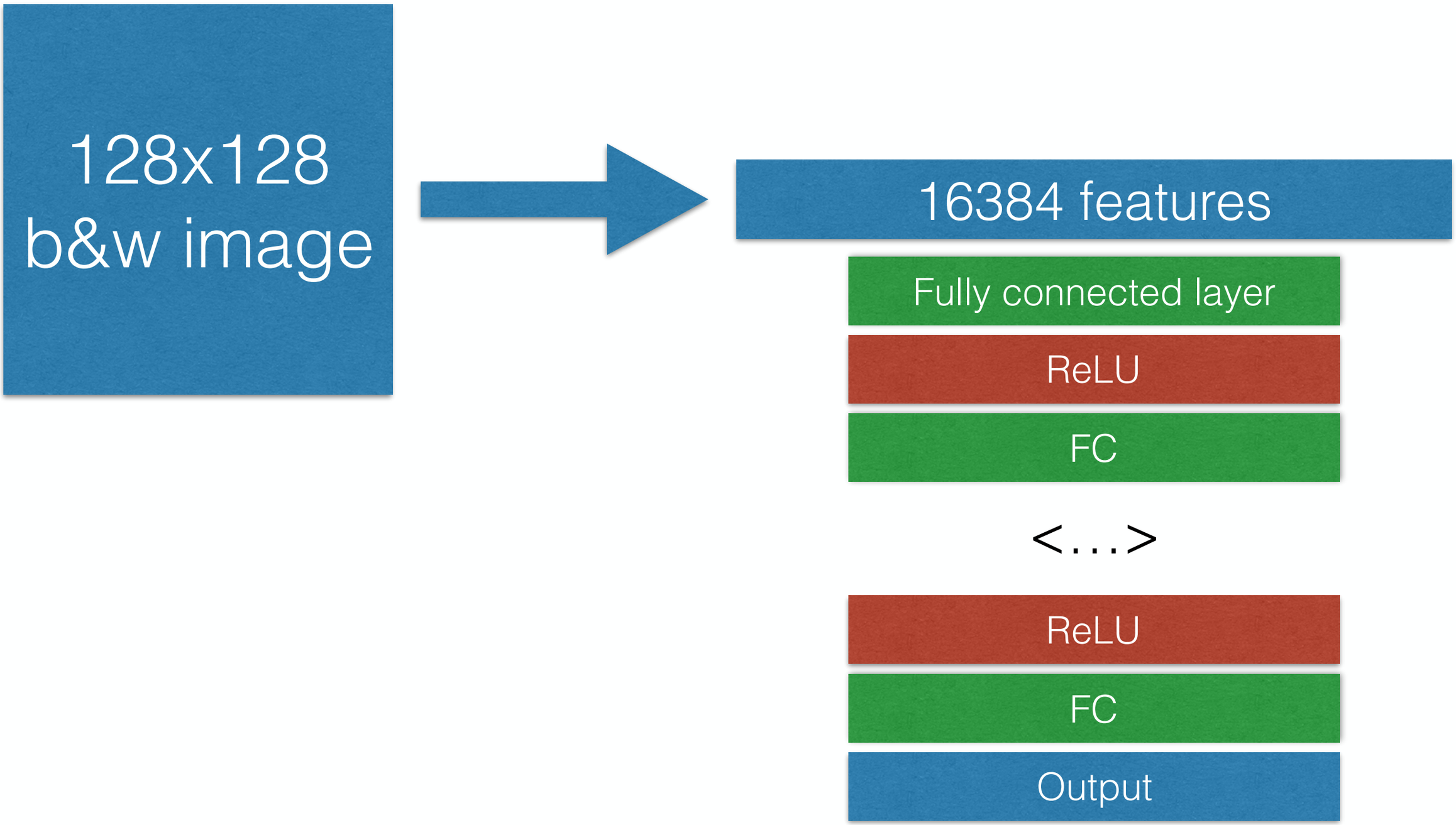
Convolution NN

Image recognition

Image recognition

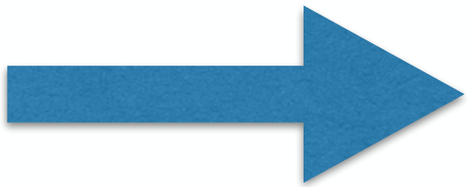
Image recognition — 2

Pixel remembering



Pixel remembering

128x128
b&w image



128x128 = 16384 features

Fully connected layer

ReLU

FC

<…>

ReLU

FC

Output

- A lot of weights!
- Information about relative position is lost
- No translation invariance

Transitional invariance

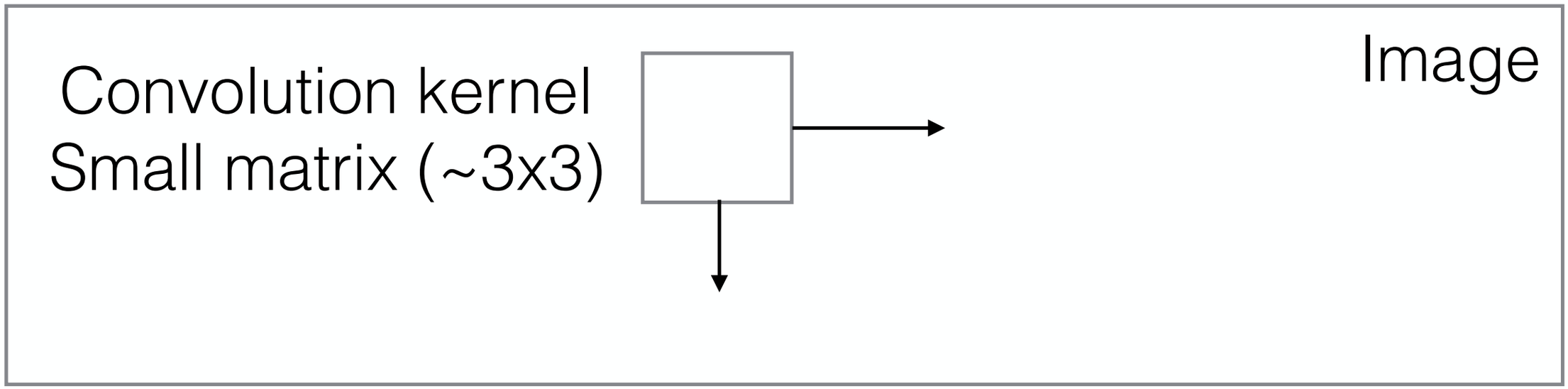
Transitional invariance

Color invariance

Weight sharing

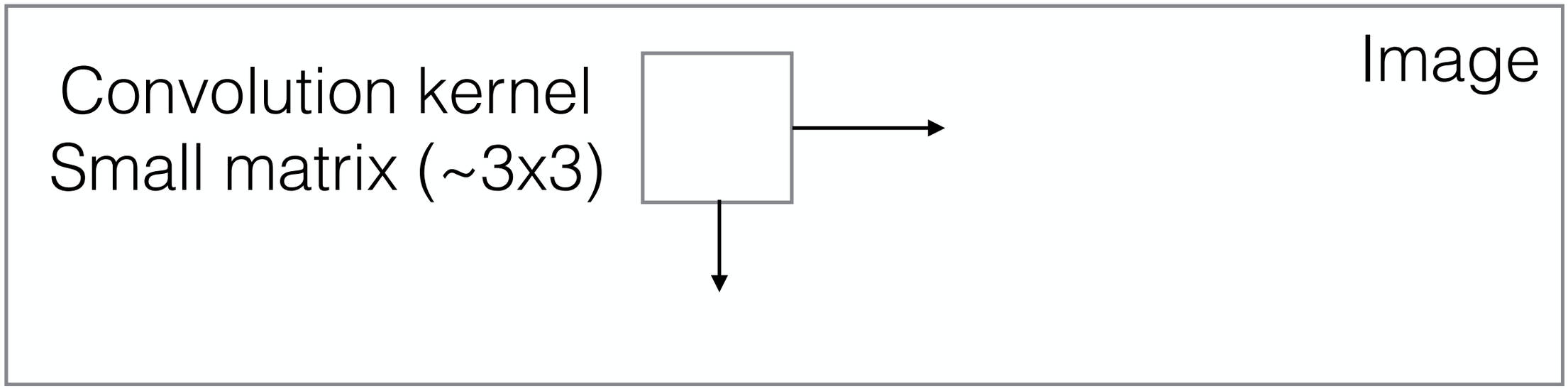
They all should have
the same weight

Convolution



In every point of image we multiply kernel on the pixels intensity and sum up.

Convolution



In every point of image we multiply kernel on the pixels intensity and sum up.

0	0	1	1	0
0	1	0	1	0
0	0	1	1	0
0	1	0	0	0
0	1	1	1	0

Image

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

Kernel

2	2	-2
1	0	-1
2	0	-2

Result:
Feature map (5-2)x(5-2)

Convolutional layer

Only 45 weights!

Input

128x128
b&w image

Layer

5 different
3x3 kernels

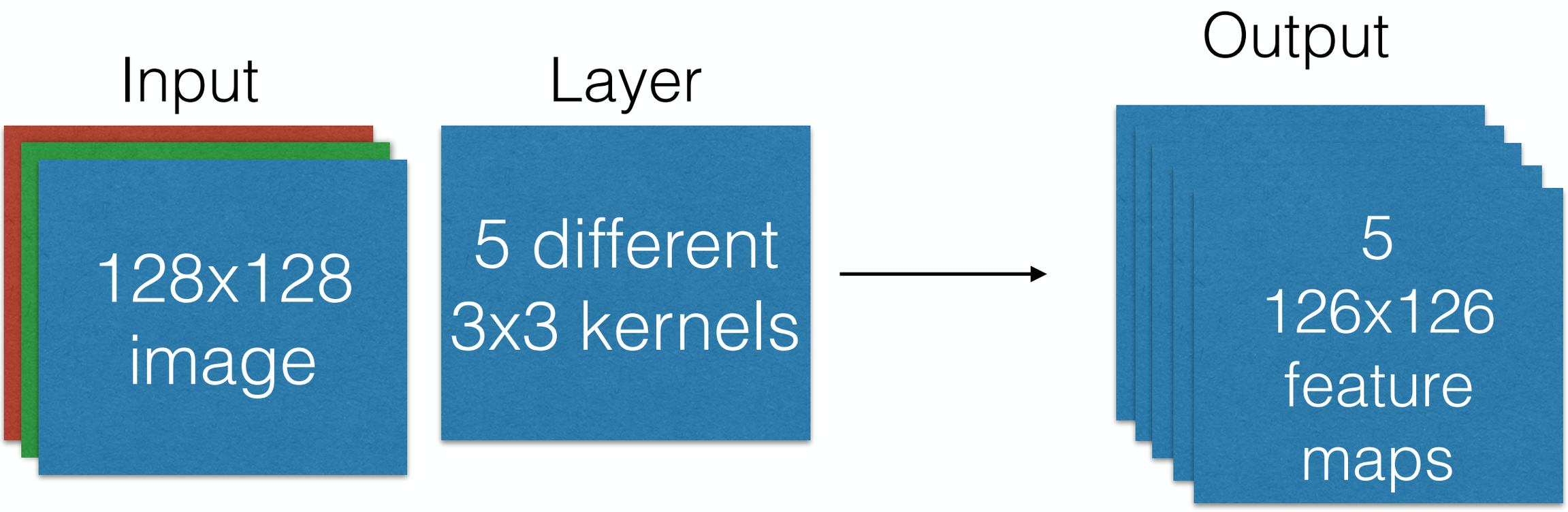
Output

5
126x126
feature
maps

May be considered as encoding 1-channel image
to 5-channel image

Convolutional layer

Only 135 weights!



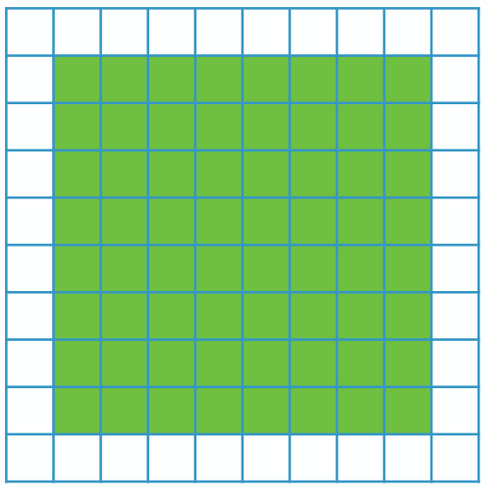
May be considered as encoding 3-channel image to 5-channel image

Reduce the output

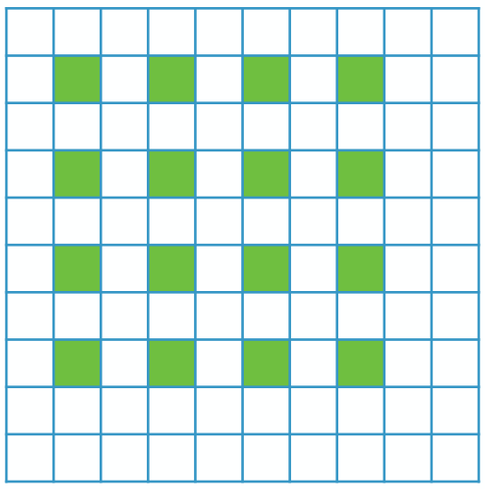
5x126x126 output from convolution is pretty big

Strategy 1: Strides

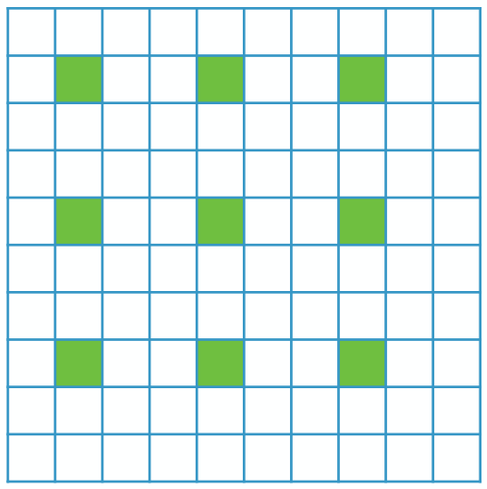
We calculated convolution
at green points



Stride = 1



Stride = 2



Stride = 3

Very aggressive

Reduce the output

5x126x126 output from convolution is pretty big

Strategy 2: Pooling

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

Feature Map

Pool size=2

Max Pooling →

4	7	9
1	6	7
4	5	3

Average Pooling →

1	4	4,8
0,8	2,5	2,8
2,5	1	0,8

Typical DNN

Input	3-channel image	3x128x128	49152
Convolution	6 kernels, 5x5	6x124x124	92256
MaxPool	Pool size = 2	6x62x62	19220
Convolution	12 kernels, 3x3	12x60x60	43200
MaxPool	Pool size = 2	12x30x30	43200
Convolution	20 kernels, 3x3	20x28x28	10800
MaxPool	Pool size = 2	20x14x14	3920
Convolution	40 kernels, 3x3	40x12x12	5760
MaxPool	Pool size = 2	40x6x6	1440
FC	512 hidden neurones		
ReLU			
FC	256 hidden neurones		
ReLU			
FC	1 neuron		

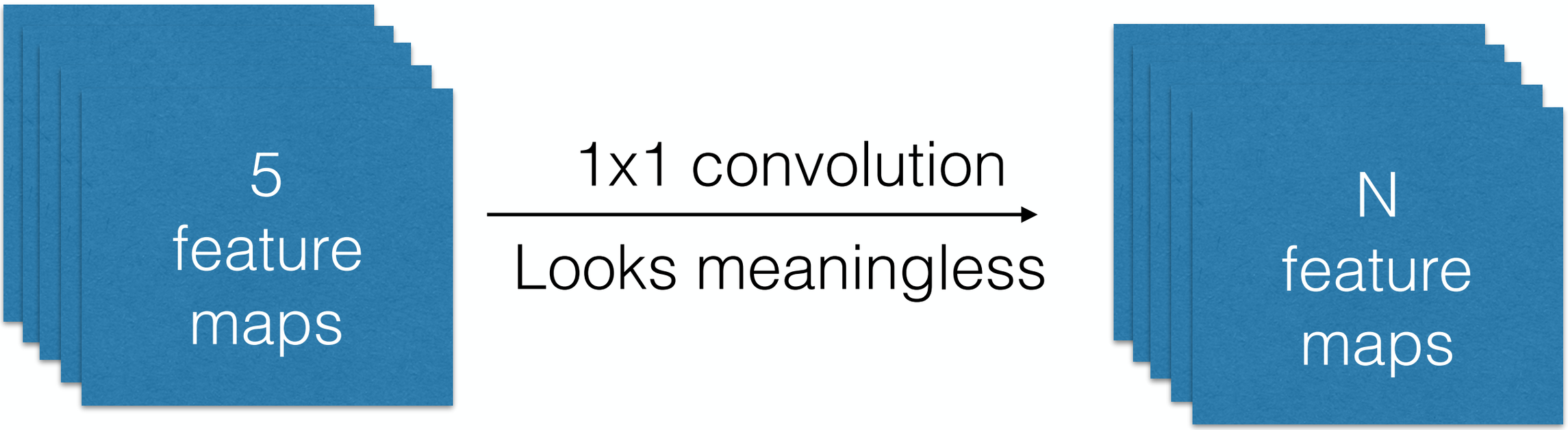
Typical DNN

Input	3-channel image	3x128x128	49152
Convolution	6 kernels, 5x5	6x124x124	92256
MaxPool	Pool size = 2	6x62x62	19220
Convolution	12 kernels, 3x3	12x60x60	43200
MaxPool	Pool size = 2	12x30x30	43200
Convolution	20 kernels, 3x3	20x28x28	10800
MaxPool	Pool size = 2	20x14x14	3920
Convolution	40 kernels, 3x3	40x12x12	5760
MaxPool	Pool size = 2	40x6x6	1440
FC	512 hidden neurones		
ReLU			
FC	256 hidden neurones		
ReLU			
FC	1 neuron		

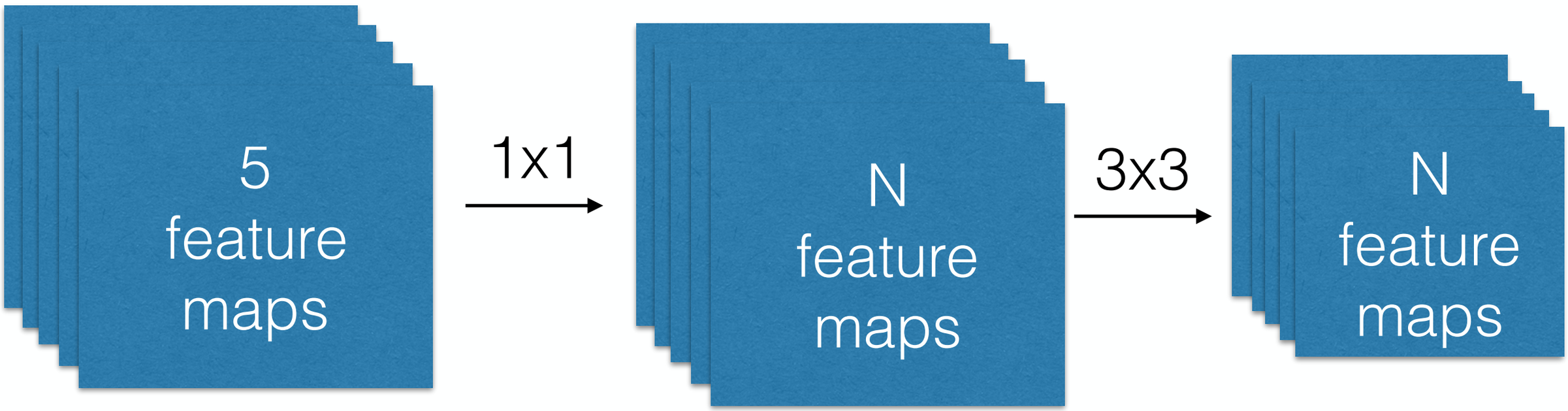
ReLU

You can insert non-linearity!

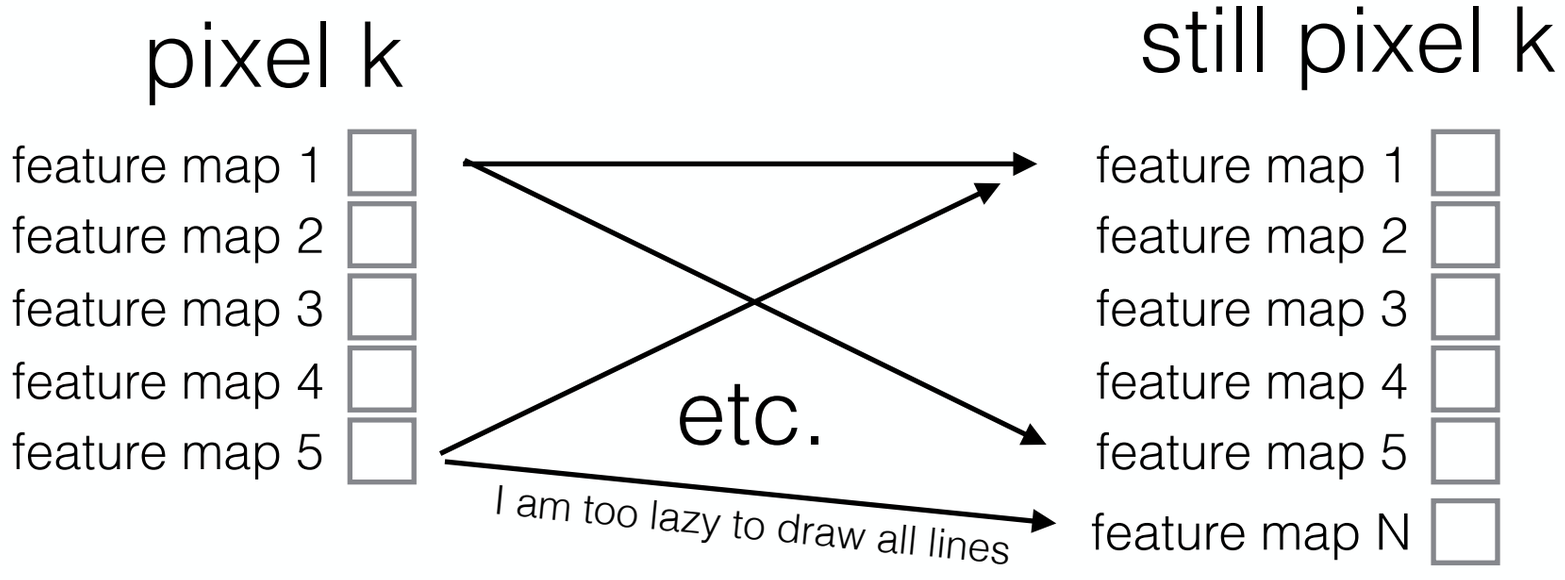
1x1 convolution



1x1 convolution



Now it's deep even on convolution!



We need more data

Rule of thumb:

at least 10 points per one degree of freedom

Google images use DNN with 37M weights =>

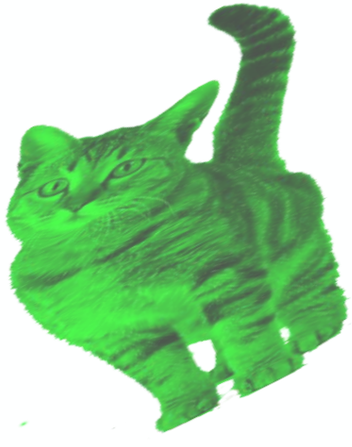
at least 370M images to train

Data augmentation

Cat

Data augmentation

Cat



Still a cat — we can learn on them!

Overfitting

Thousands of parameters may (and will) cause overfitting

- Data augmentation if possible
- L2 penalty (add sum of weights squares to loss)
- Dropout

Dropout

Introduced in 2014

In terms of layers as functions:

$$f_{train}(x) = \begin{cases} 0, & \text{with probability } p \\ x, & \text{with probability } 1-p \end{cases}$$

$$f_{test}(x) = (1-p)x$$

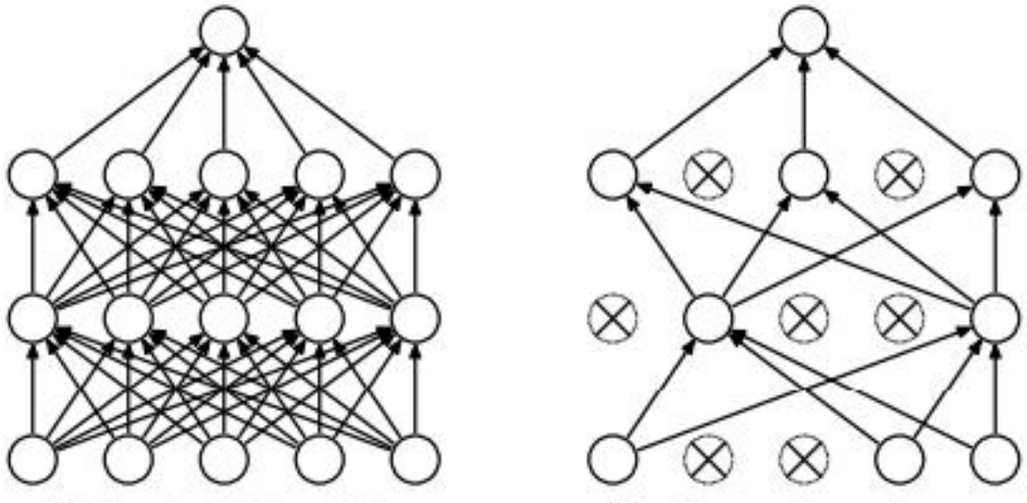
Dropout

Introduced in 2014

In terms of layers as functions:

$$f_{train}(x) = \begin{cases} 0, & \text{with probability } p \\ x, & \text{with probability } 1-p \end{cases}$$

$$f_{test}(x) = (1-p)x$$



(a) Standard Neural Net

(b) After applying dropout.

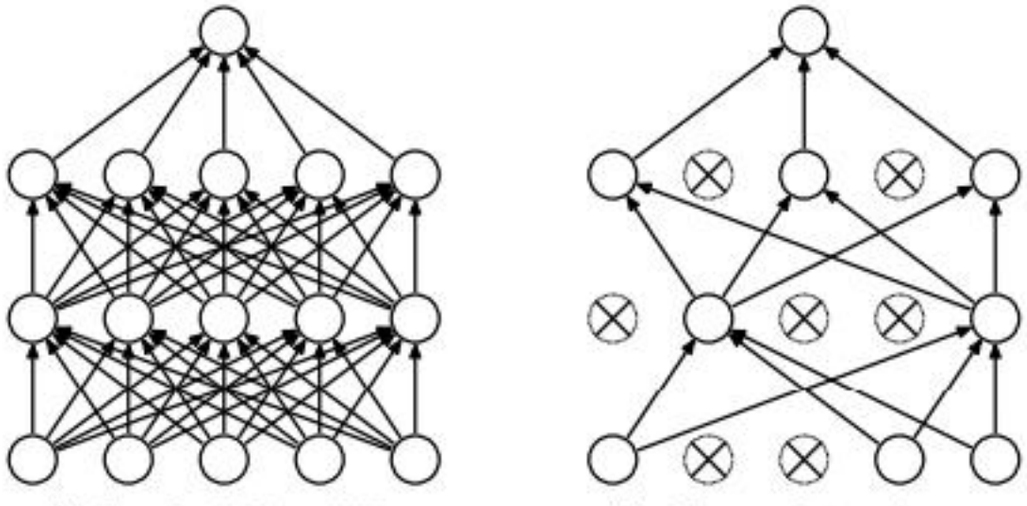
Dropout

Introduced in 2014

In terms of layers as functions:

$$f_{train}(x) = \begin{cases} 0, & \text{with probability } p \\ x, & \text{with probability } 1-p \end{cases}$$

$$f_{test}(x) = (1-p)x$$



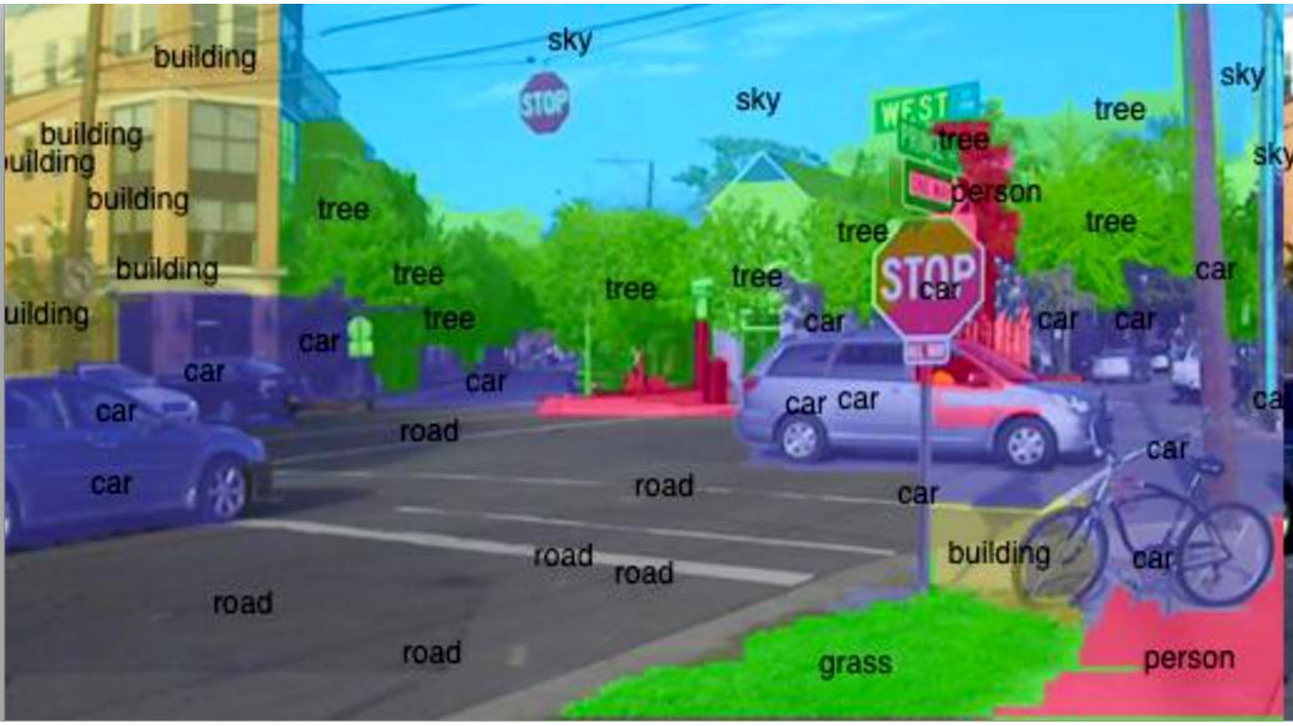
(a) Standard Neural Net

(b) After applying dropout.

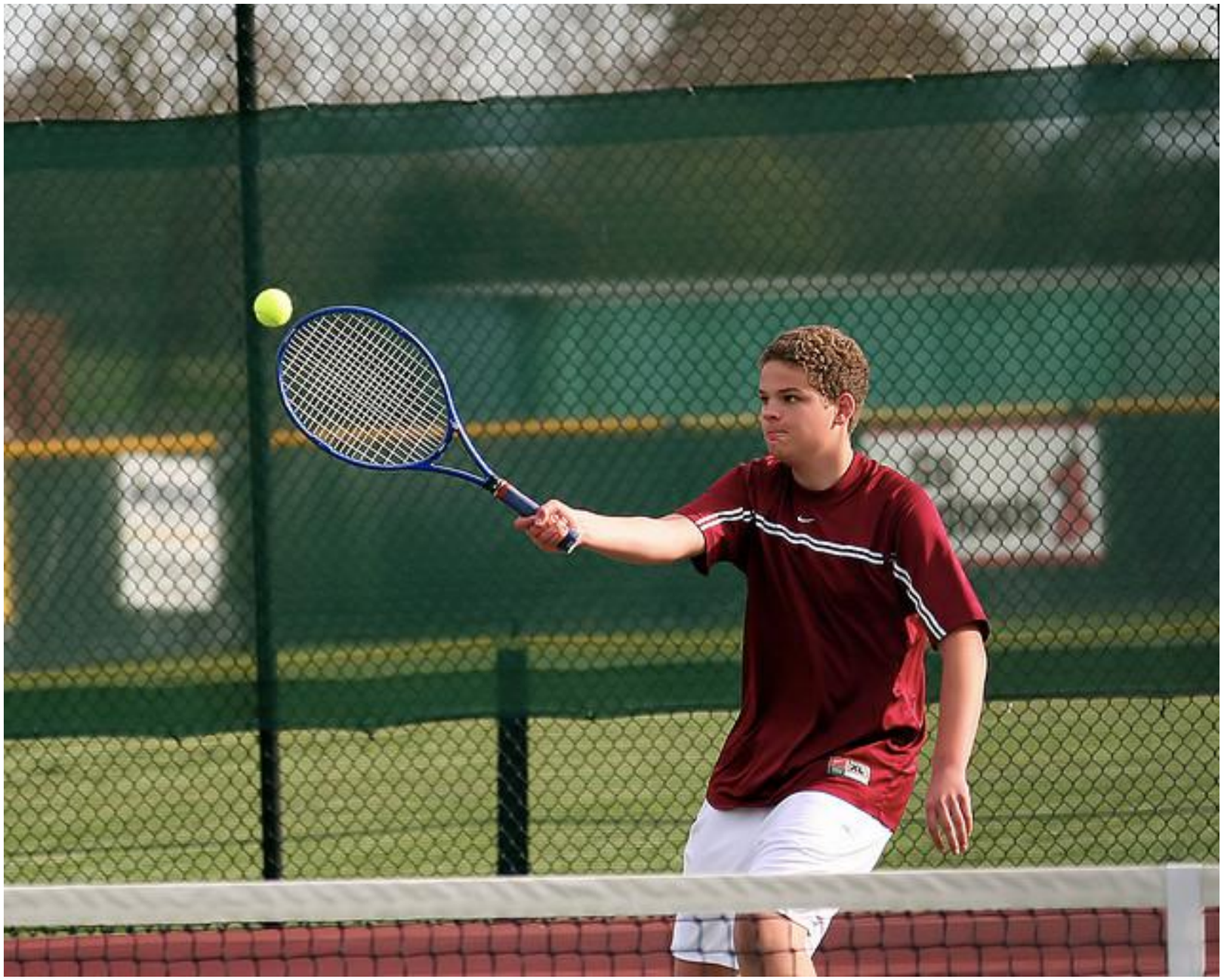
p=0.5 is good choice

Dropout: A Simple Way to Prevent Neural Networks from Overfitting / Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, Ruslan Salakhutdinov; 15(Jun): 1929–1958, 2014.

DNN performance



DNN performance



a man is playing tennis on a tennis court

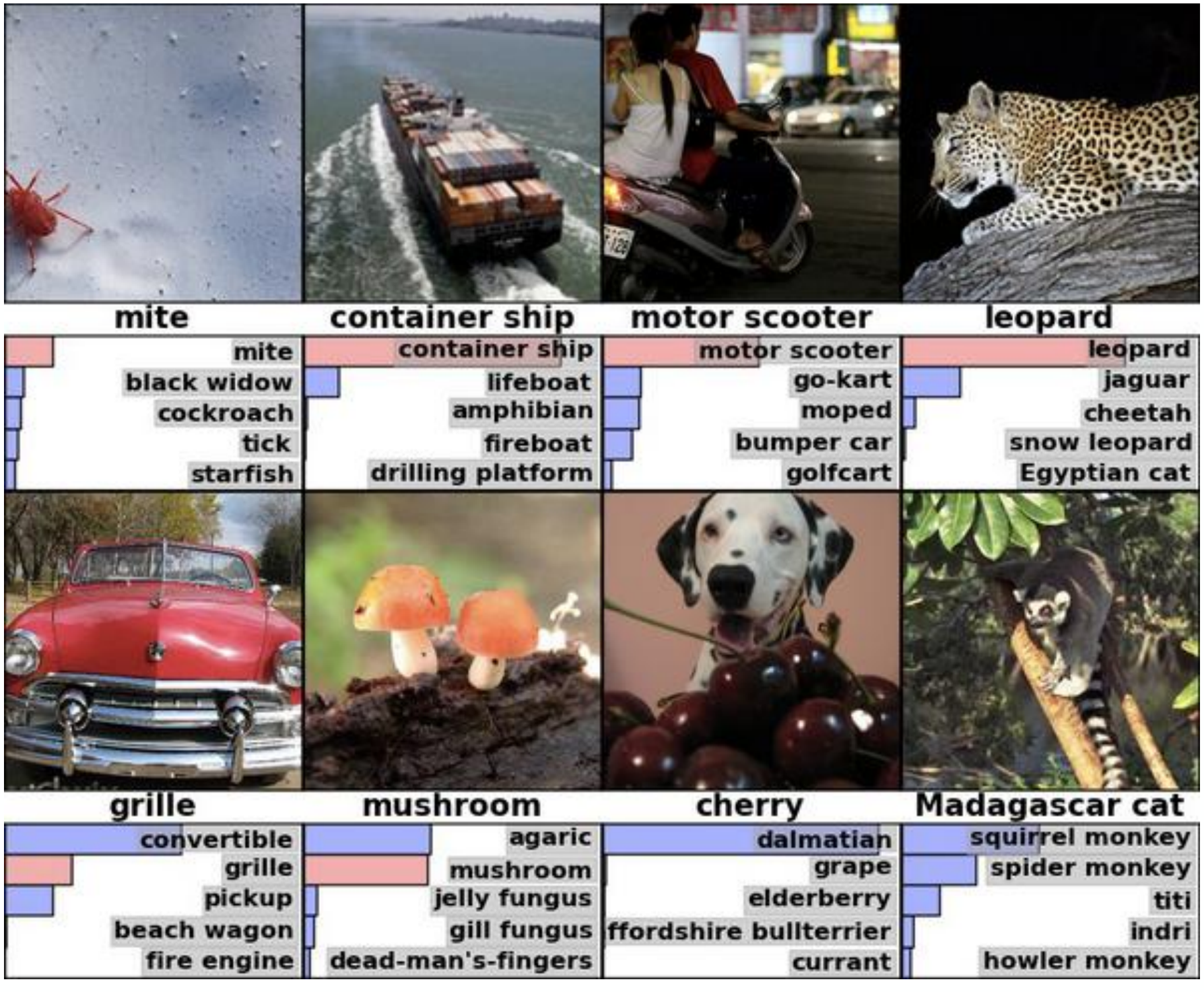
DNN performance

a man sitting on a bus with a dog

DNN performance

a bunch of bananas are hanging from a ceiling

DNN performance



DNN performance

The Great Wave off Kanagawa

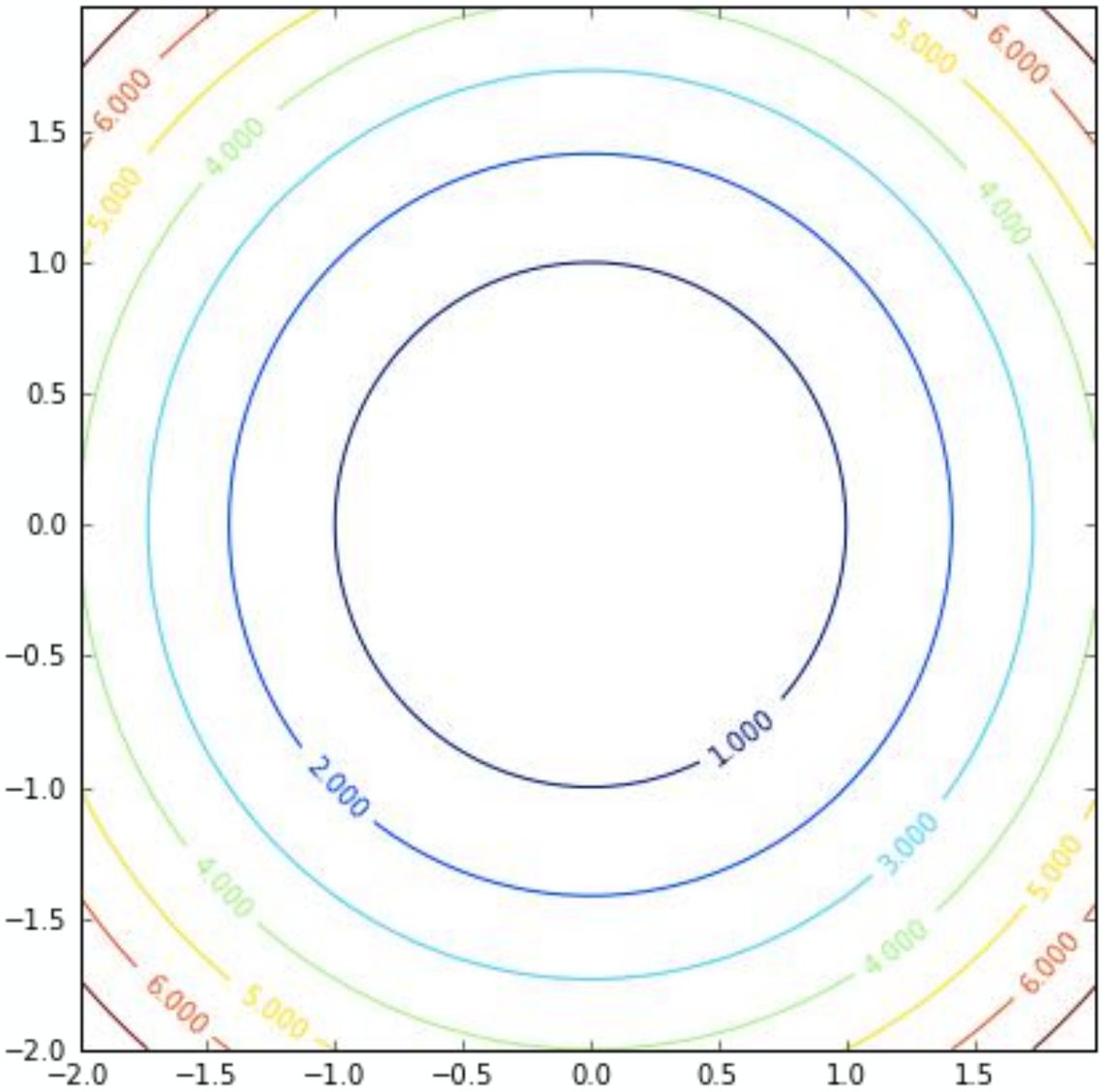
+

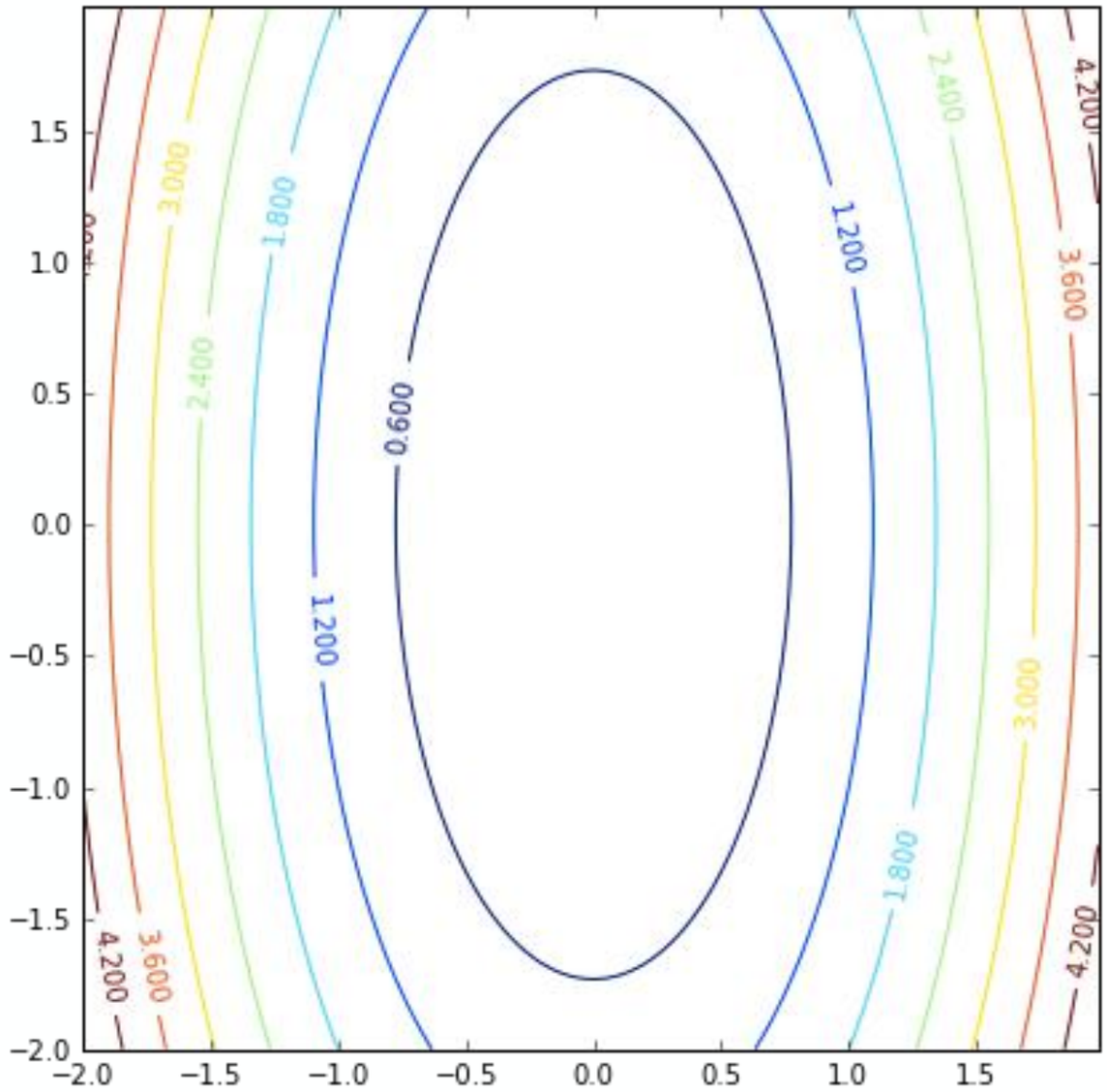
DNN performance

deepart.io

SGD hacks

Standardization





Which one is better for SGD (GD)?

Batch norm

Introduced in 2015

Instead of batch X use $\frac{X - \mu_X}{\sigma_X}$

Batch norm

Introduced in 2015

Instead of batch X use $\frac{X - \mu_X}{\sigma_X}$

It's a layer too! $f(X) = \frac{X - \mu_X}{\sigma_X}$

Really speeds up learning!

AdaGrad

$$W_t = W_{t-1} - \alpha \frac{\nabla W_{t-1}}{\sqrt{G}}$$

Remember total update of every feature and scale updates to prevent jittering

Use momentum

Add momentum to your SGD path

$$\nabla W_t = \nabla W_t + \lambda \nabla W_{t-1}$$

Use ~0.9 momentum rate

Nesterov acceleration

$$\nabla W_t = \nabla(W_t + \lambda \nabla W_{t-1}) + \lambda \nabla W_{t-1}$$

Point: Nesterov is better than everything

Weights initialization

W is gaussian, b is constant (or uniform [-1;1])

There are a lot of ways to select best properties of gaussian distribution for W

One of the best is:

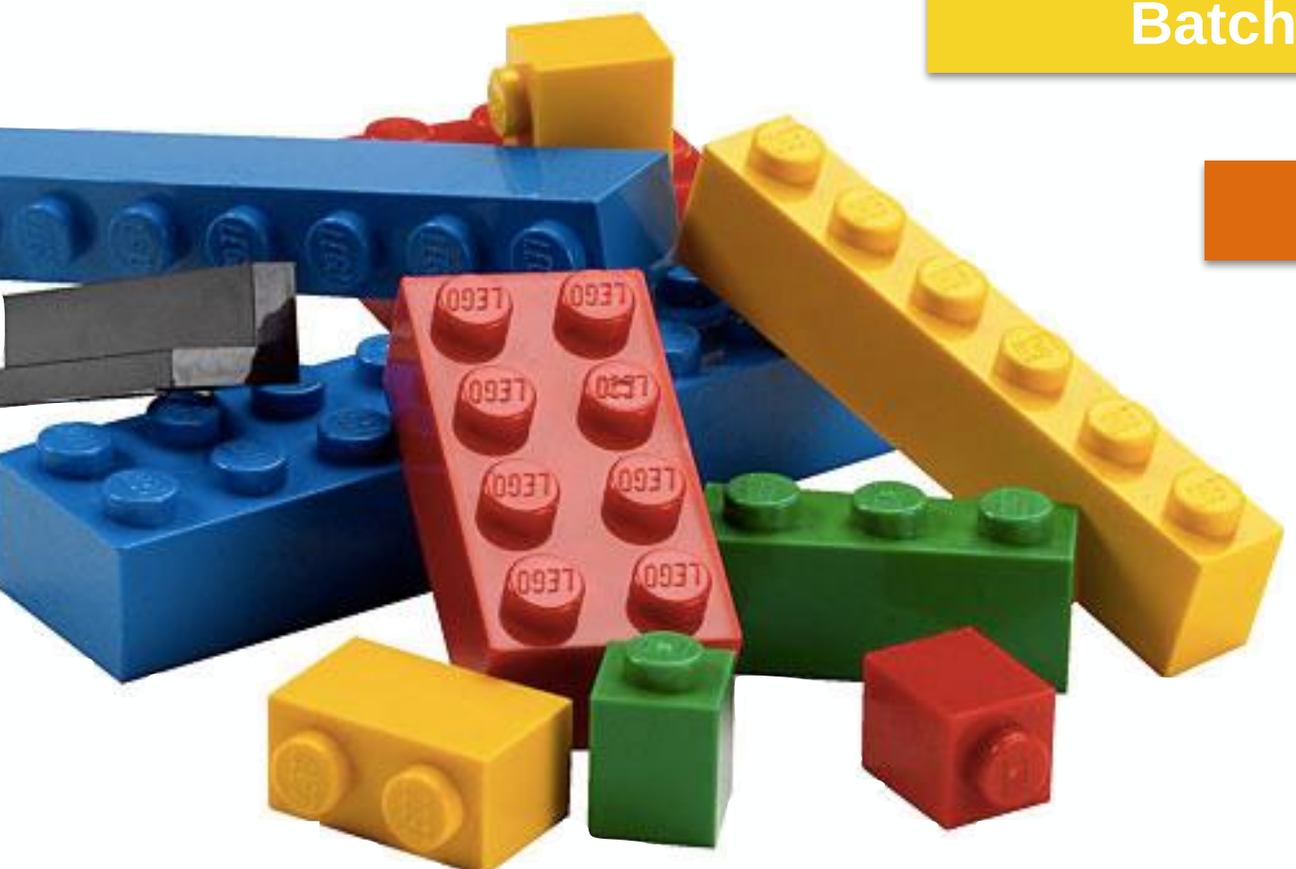
$$W \sim N\left(0, \frac{1}{\sqrt{\#\text{ input neurones}}}\right)$$

Lego

FC

ReLU

Dropout



Batch Norm

MaxPool

Convolution

1x1 Convolution

When to use?

- There are a lot of data
- The data is persistent
- There is some real structure in the data
- All types of images
- You need to complex interaction
- Features are strongly correlated

What to do next?

- Write your own Neural Network?

What to do next?

- Write your own Neural Network?
- Take Deep Learning course @ udacity

What to do next?

- Write your own Neural Network?
- Take Deep Learning course @ udacity
- Take a look at low-level frameworks: especially **TensorFlow**

What to do next?

- Write your own Neural Network?
- Take Deep Learning course @ udacity
- Take a look at low-level frameworks: especially **TensorFlow**
- Take a look at high-level **Keras**

What to do next?

- Write your own Neural Network?
- Take Deep Learning course @ udacity
- Take a look at low-level frameworks:
  especially **TensorFlow**
- Take a look at high-level **Keras**
- Just watch Yann LeCun lectures

What to do next?

- Write your own Neural Network?
- Take Deep Learning course @ udacity
- Take a look at low-level frameworks:
  especially **TensorFlow**
- Take a look at high-level **Keras**
- Just watch Yann LeCun lectures
- Experiment! (buy GPU before)

What to do next?

- Write your own Neural Network?
- Take Deep Learning course @ udacity
- Take a look at low-level frameworks:
  especially **TensorFlow**
- Take a look at high-level **Keras**
- Just watch Yann LeCun lectures
- Experiment! (buy GPU before)
- **Decrease your learning rate!**

“Deep learning is shallow next to the depth of your greatness.”

–Kaggle