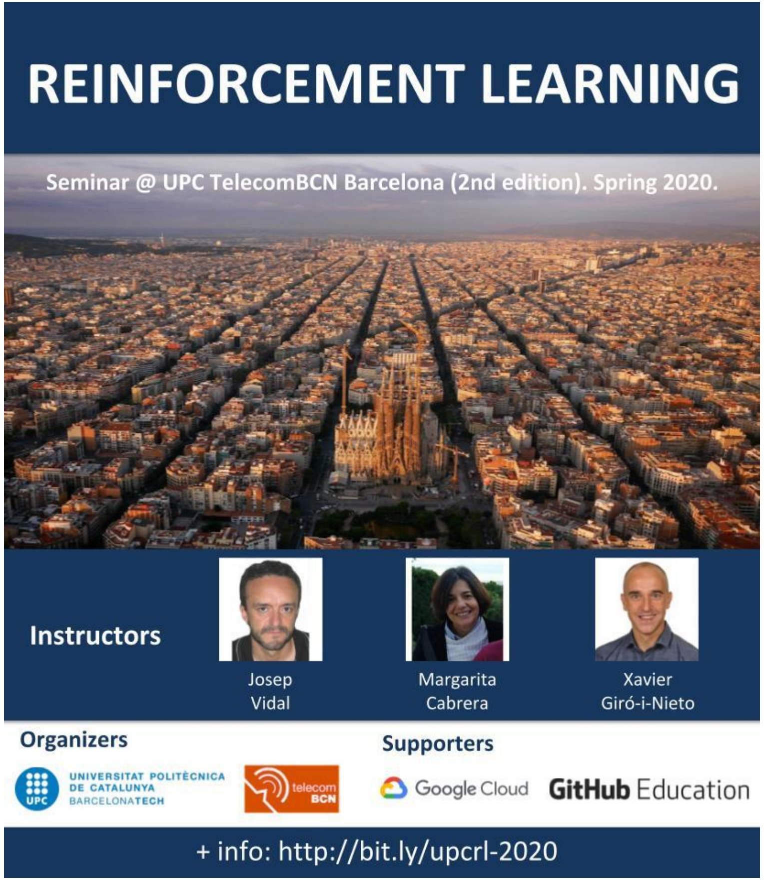


https://telecombcn-dl.github.io/mrl-2020/

MRL 2020 - Day 9 - Part 1

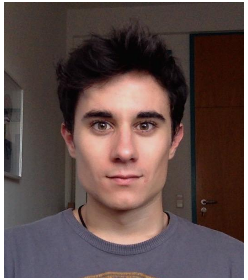
How to train your neural network

Xavier Giro-i-Nieto

@DocXavi
xavier.giro@upc.edu

Associate Professor
ETSETB TelecomBCN
Universitat Politècnica de Catalunya

Acknowledgments



Víctor Campos
victor.campos@bsc.es

PhD Candidate

Barcelona Supercomputing Center

Míriam Bellver
miriam.bellver@bsc.edu

PhD Candidate
Barcelona Supercomputing Center

Kevin McGuinness
kevin.mcguinness@dcu.ie

Research Fellow
Insight Centre for Data Analytics
Dublin City University

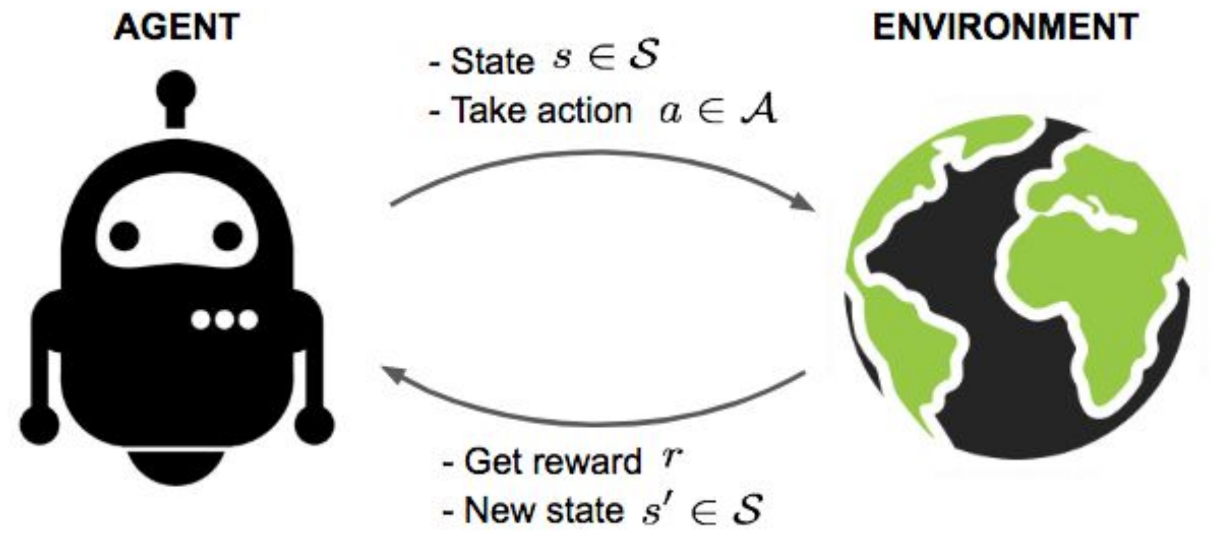
Elisa Sayrol
elisa.sayrol@upc.edu

Associate Professor
ETSETB TelecomBCN
Universitat Politècnica de Catalunya

Outline

1. RL with Neural Networks
2. Loss functions
3. Backpropagation
4. Optimizers

Reinforcement Learning (with extrinsic reward)



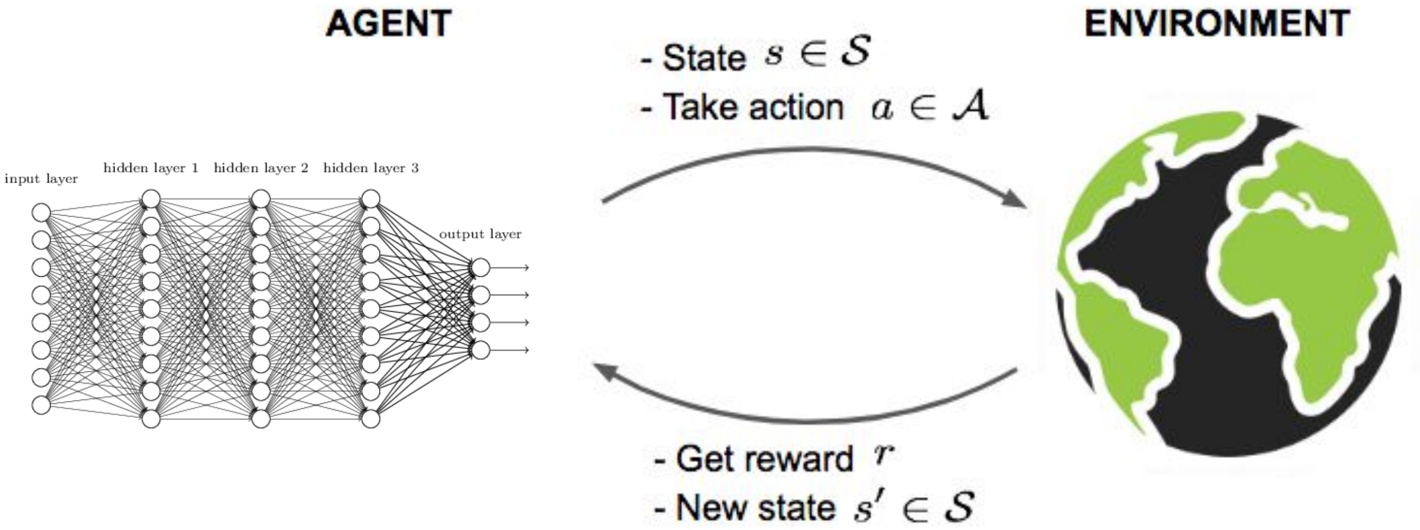
Policy π

Value function

Model
(of the Environment)

Goals of Reinforcement Learning

Reinforcement Learning with Neural Networks (NN)



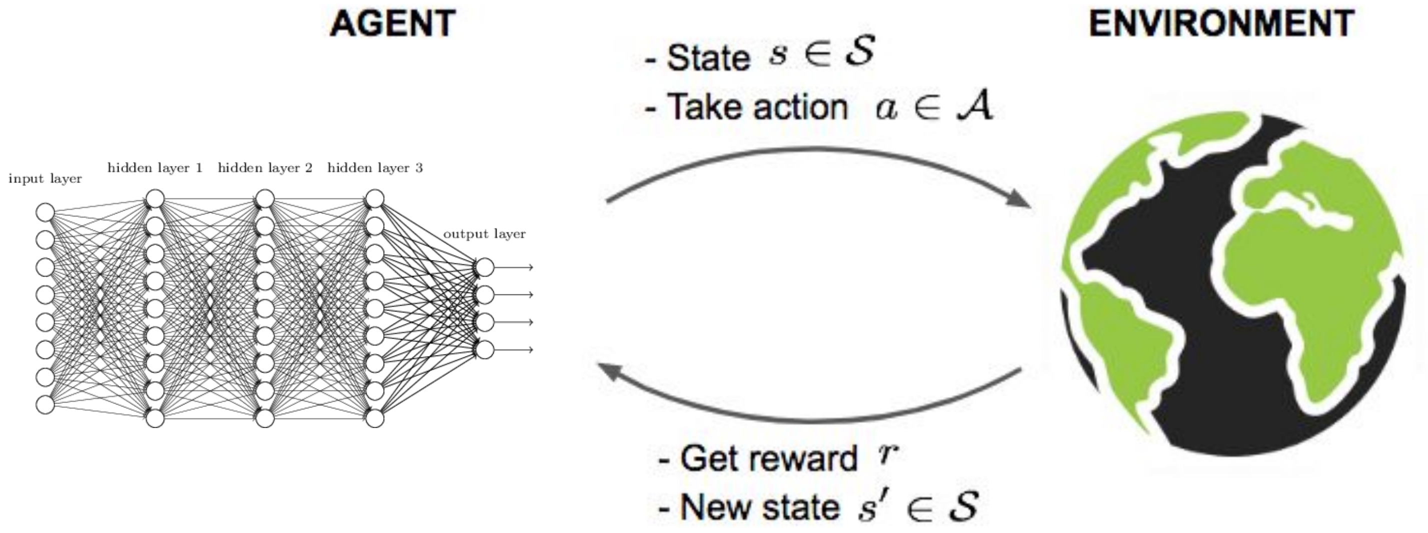
Policy π

Value function

Model
(of the Environment)

Goals of Reinforcement Learning

Policy-based RL with Neural Networks (NN)



Policy π

Value function

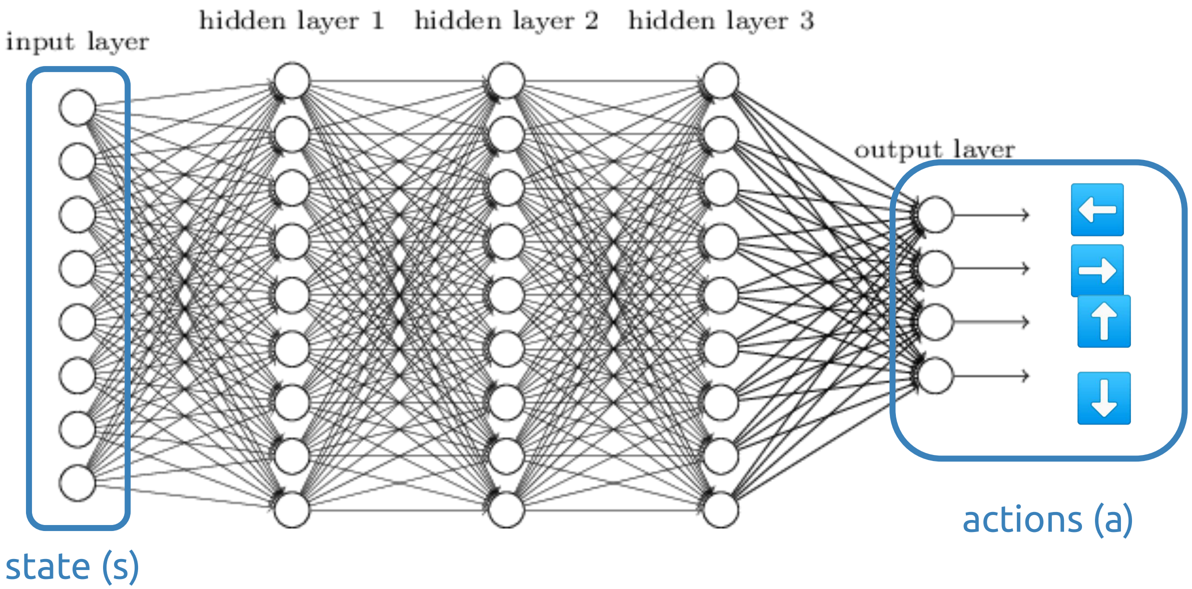
Model
(of the Environment)

Goals of Reinforcement Learning

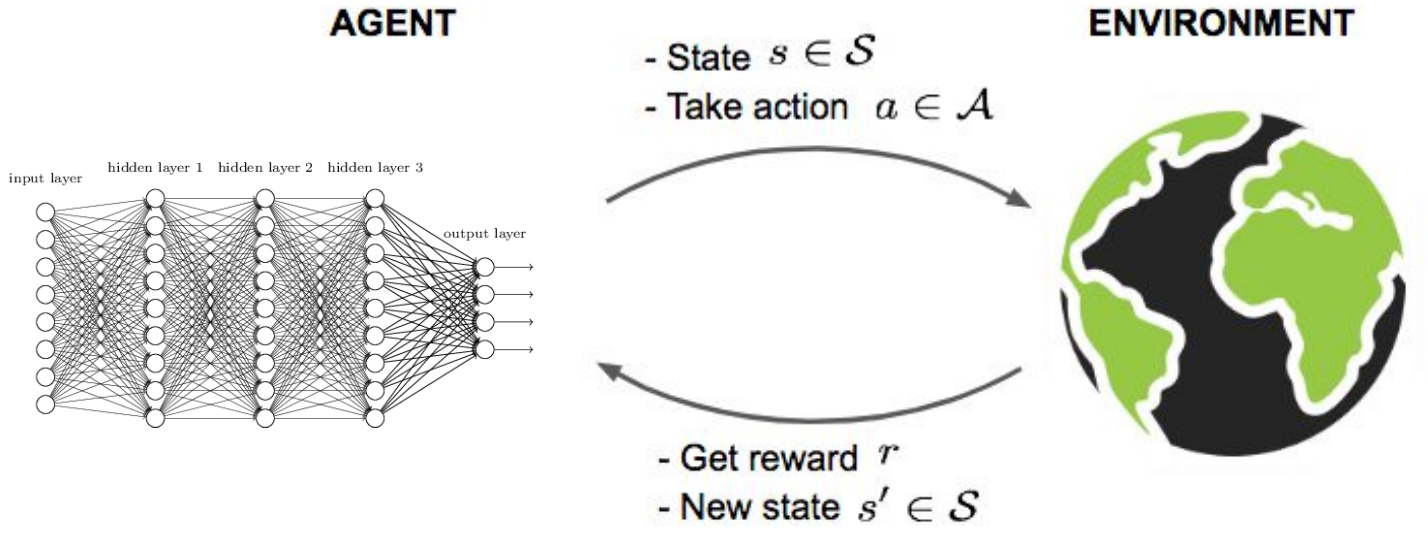
Policy-based RL with Neural Networks (NN)

The **Policy** π is a function S → A that specifies which action to take in each state:

A Multi-Layer Perceptron (MLP) can implement a **classifier** to predict the distribution of actions given a state.



Value-based RL with Neural Networks (NN)



Policy π

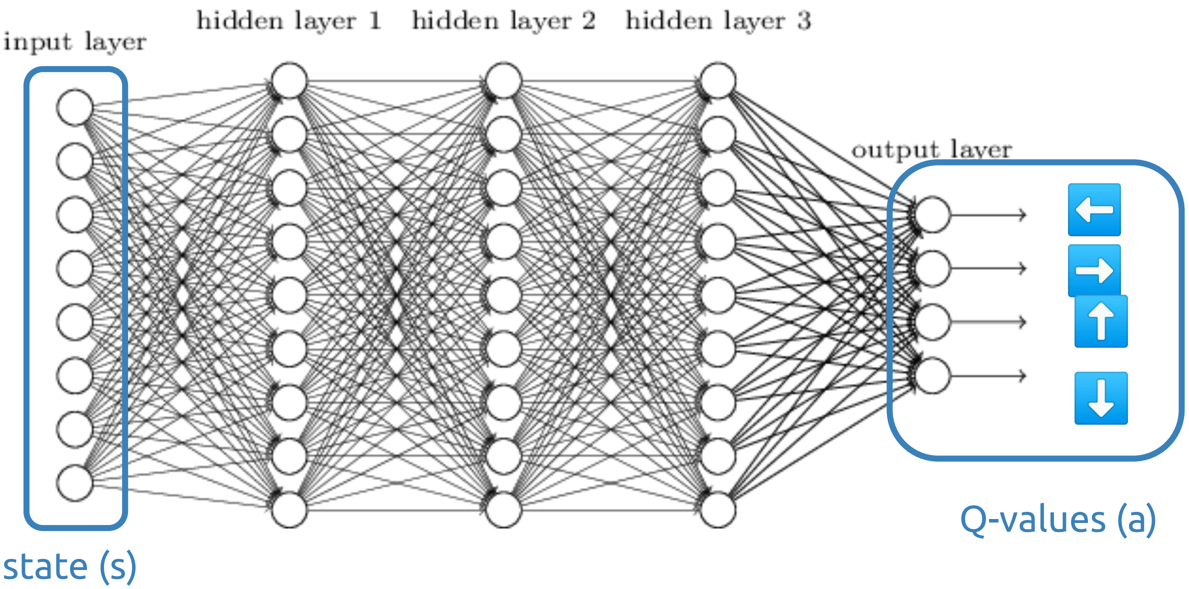
Value function

Model
(of the Environment)

Goals of Reinforcement Learning

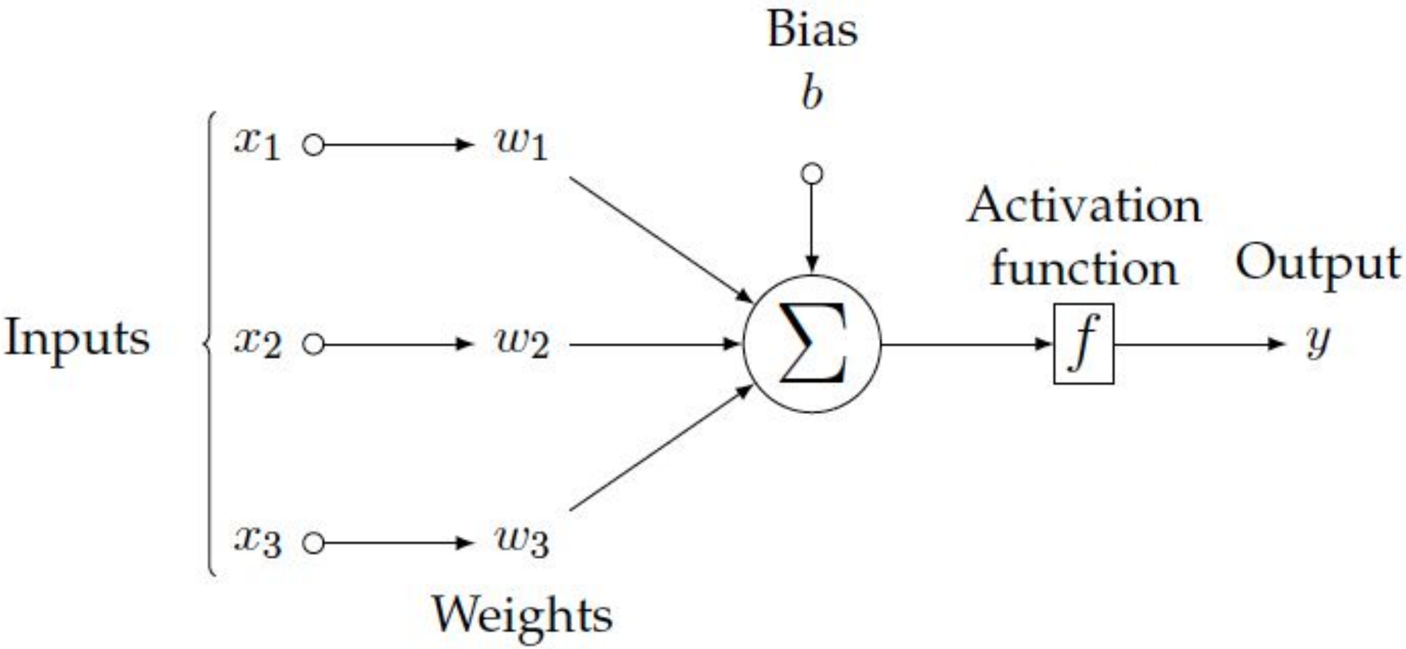
Value-based RL with Neural Networks (NN)

The **action-value $Q_\pi(s,a)$** function is the expected return for taking action a when being in state s.
A Multi-Layer Perceptron (MLP) can implement a **regressor** to predict $Q_\pi(s,a)$.



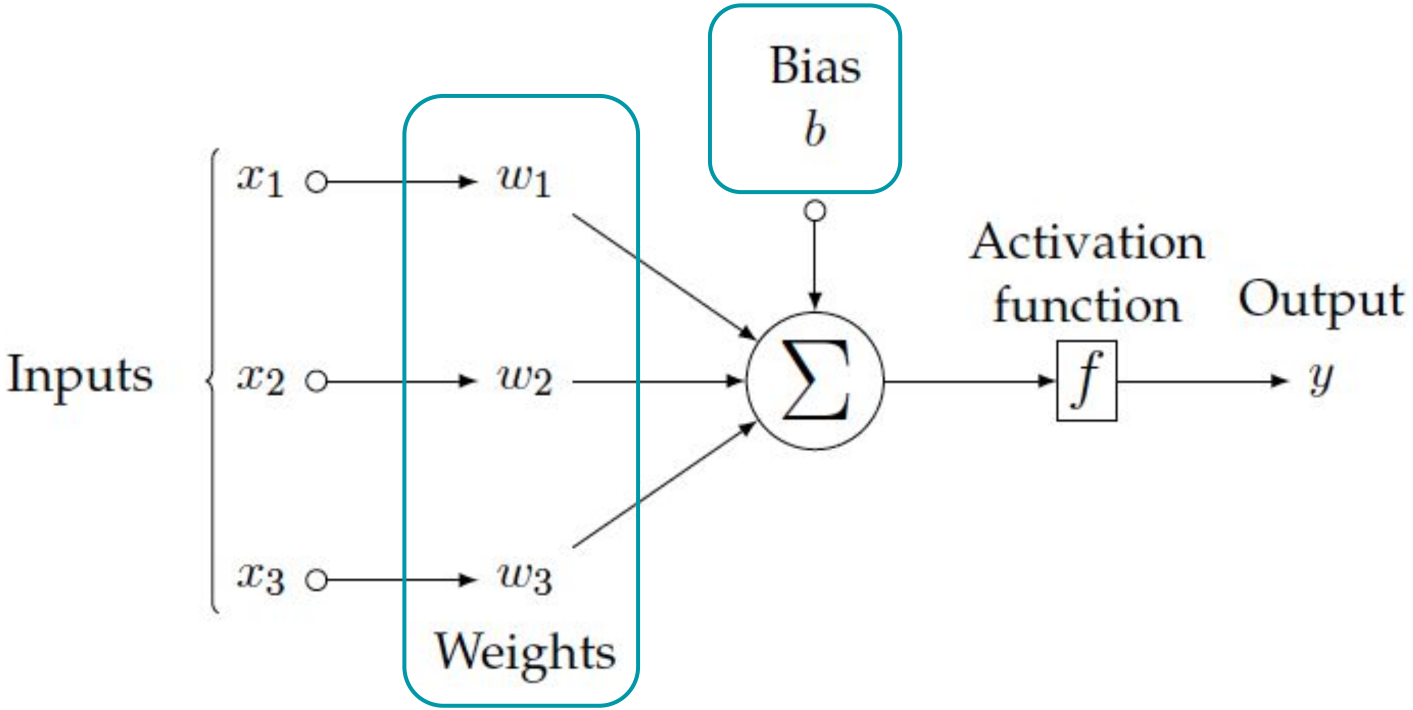
Single neuron model (perceptron)

Which components of the neurons must be estimated during training ?



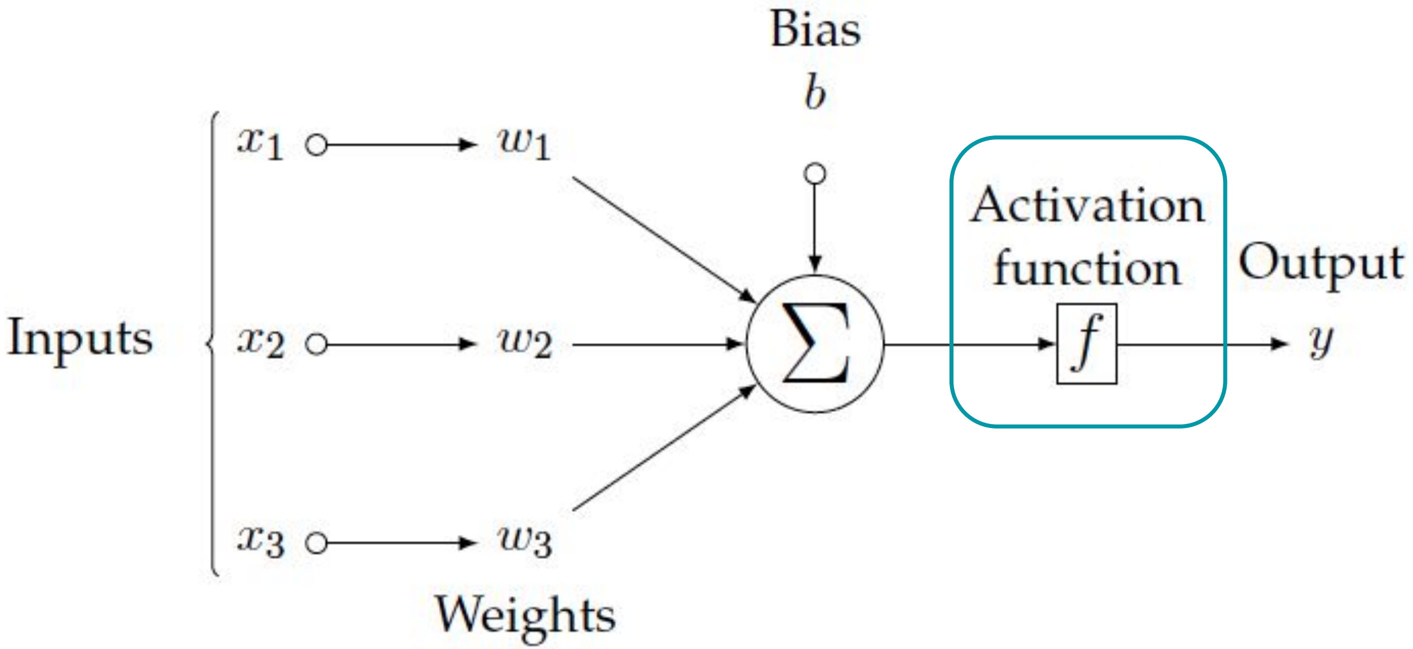
Single neuron model (perceptron)

The **weights (w_i) and bias (b)** must be estimated during training.



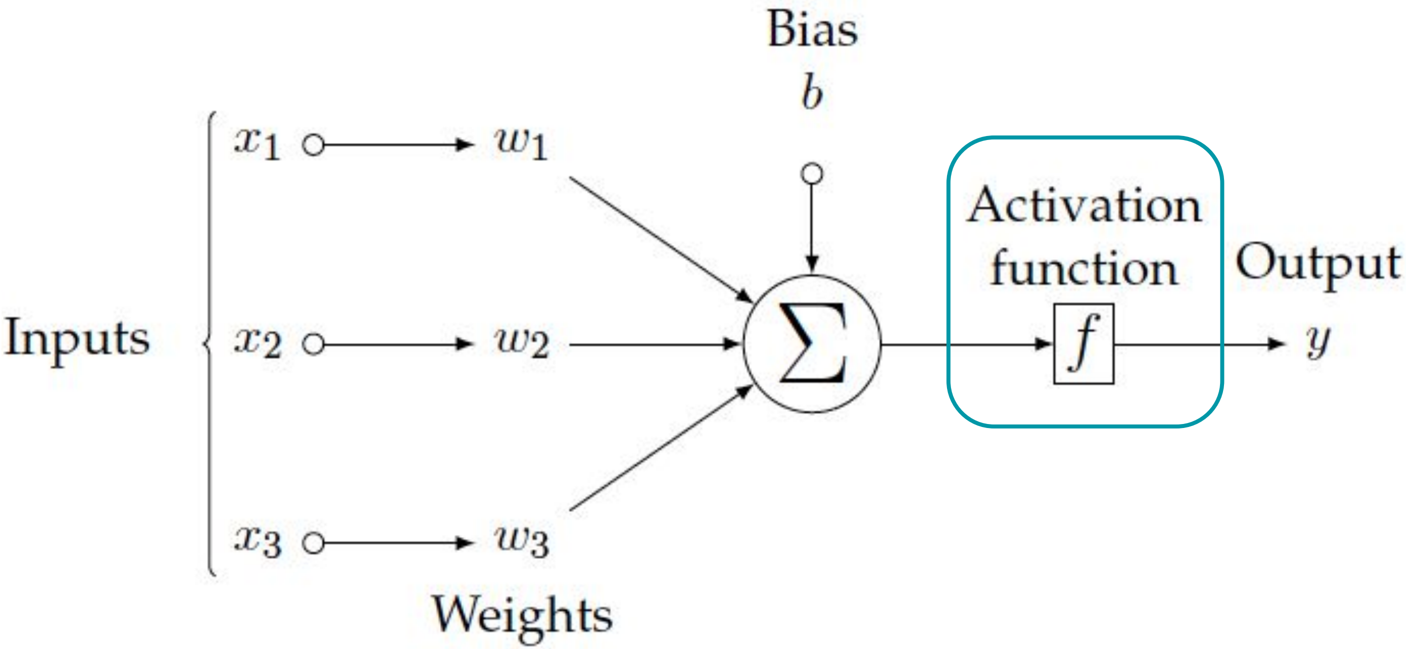
Single neuron model (perceptron)

The **activation function (f)** is a design choice.



Single neuron model (perceptron)

The **activation function (f)** is a design choice.



Single neuron model (perceptron)

Activation functions:

- They act as a **threshold**

Desirable properties

- Mostly smooth, continuous, differentiable
- Fairly linear

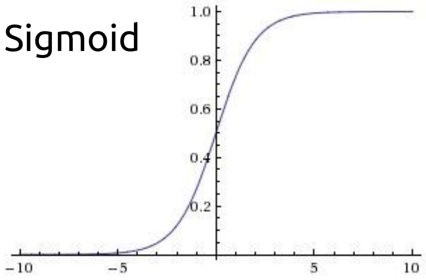
Common nonlinearities

- Sigmoid
- Tanh
- ReLU = max(0, x)

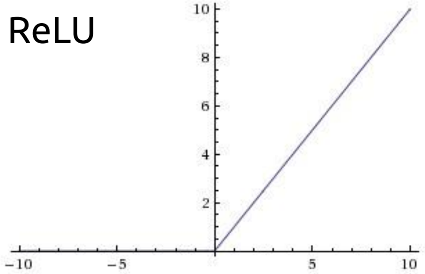
Why do we need them?

If we only use linear layers we are only able to learn linear transformations of our input.

Sigmoid

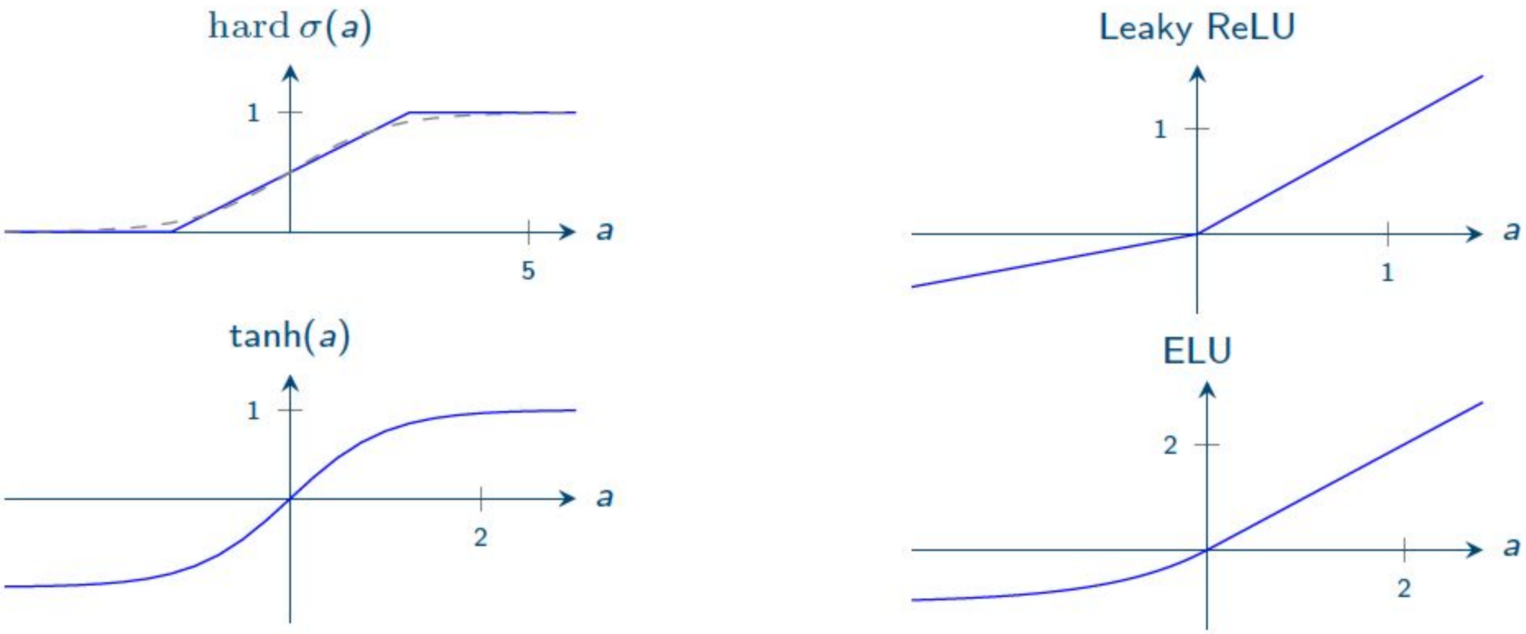


ReLU



Single neuron model (perceptron)

Other popular activation functions:



Multi-Layer Perceptron (MLP)

When each node in each layer is a linear combination of **all inputs from the previous layer** then the network is called a multilayer perceptron (MLP)

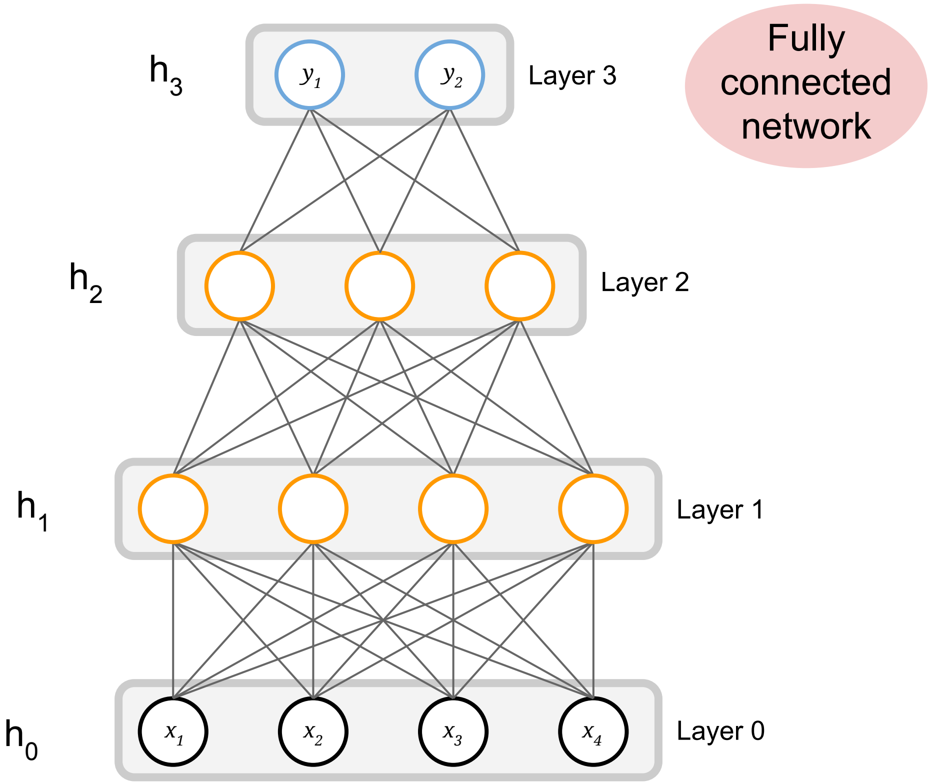
Weights can be organized into matrices.

Forward pass computes

$$\mathbf{h}_0 = \mathbf{x}$$

$$\mathbf{h}^{(t)} = g(W^{(t)}\mathbf{h}^{(t-1)} + \mathbf{b}^{(t)})$$

$$f(\mathbf{x}) = \mathbf{h}^{(L)}$$



Slide: Kevin McGuinness (DCU)

Multi-Layer Perceptron (MLP)

W_1

w_{11}	w_{12}	w_{13}	w_{14}
w_{21}	w_{22}	w_{23}	w_{24}
w_{31}	w_{32}	w_{33}	w_{34}
w_{41}	w_{42}	w_{43}	w_{44}

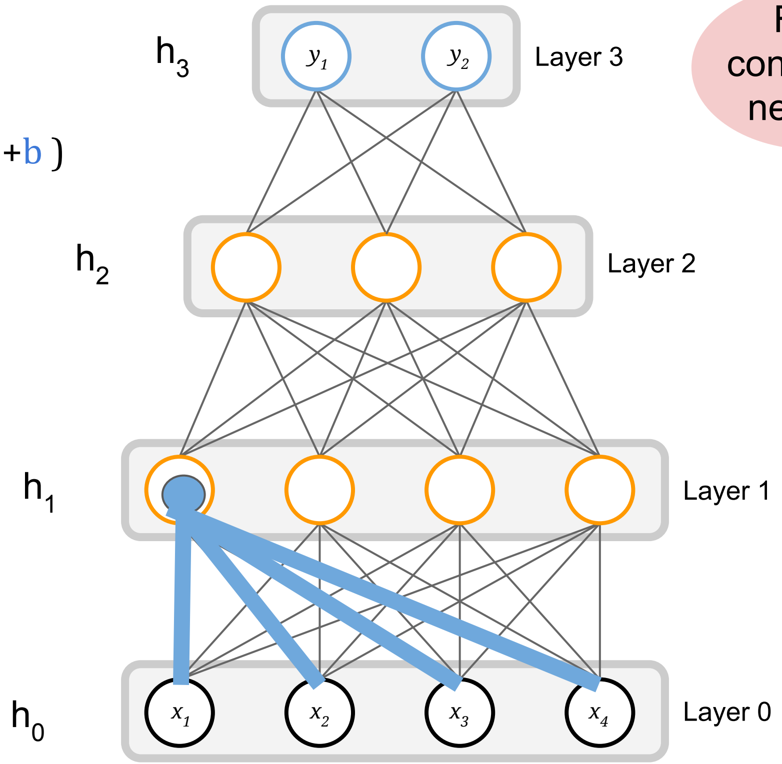
h_0

x_1
x_2
x_3
x_4

b_1

b_1
b_2
b_3
b_4

$h_{11} = g(\mathbf{wx} + b)$



Fully connected network

Forward pass computes

$$\mathbf{h}_0 = \mathbf{x}$$

$$\mathbf{h}^{(t)} = g(W^{(t)}\mathbf{h}^{(t-1)} + \mathbf{b}^{(t)})$$

$$f(\mathbf{x}) = \mathbf{h}^{(L)}$$

Slide: Kevin McGuinness (DCU)

Multi-Layer Perceptron (MLP)

W_1

w_{11}	w_{12}	w_{13}	w_{14}
w_{21}	w_{22}	w_{23}	w_{24}
w_{31}	w_{32}	w_{33}	w_{34}
w_{41}	w_{42}	w_{43}	w_{44}

h_0

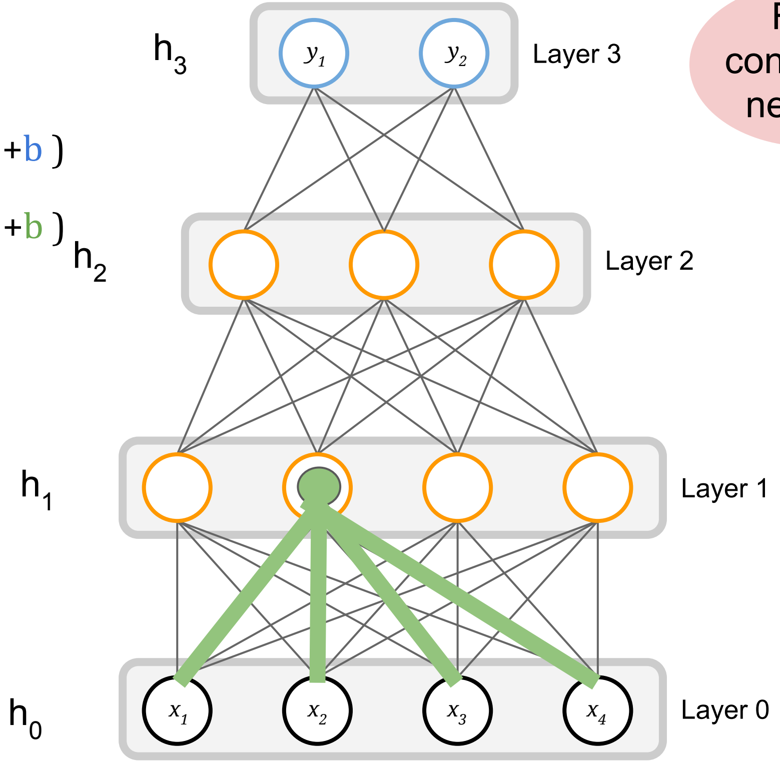
x_1
x_2
x_3
x_4

b_1

b_1
b_2
b_3
b_4

$h_{11} = g(\mathbf{w}\mathbf{x} + b)$

$h_{12} = g(\mathbf{w}\mathbf{x} + b)$



Fully connected network

Forward pass computes

$$\mathbf{h}_0 = \mathbf{x}$$

$$\mathbf{h}^{(t)} = g(W^{(t)}\mathbf{h}^{(t-1)} + \mathbf{b}^{(t)})$$

$$f(\mathbf{x}) = \mathbf{h}^{(L)}$$

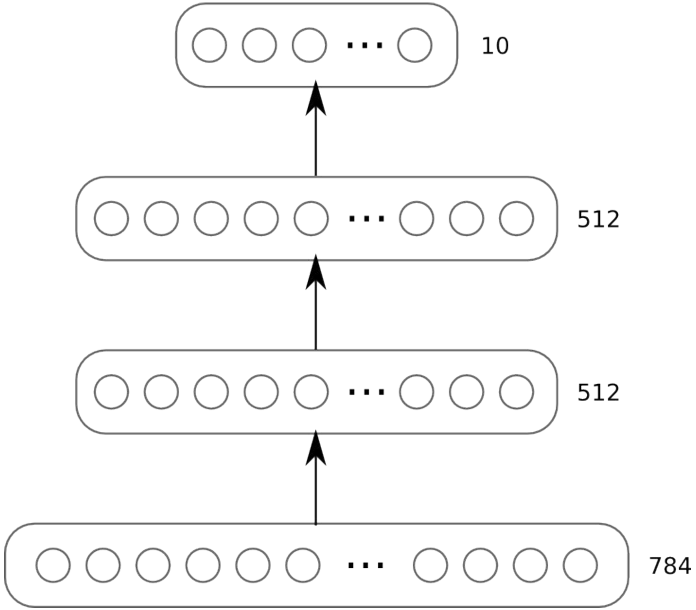
Slide: Kevin McGuinness (DCU)

Multi-Layer Perceptron (MLP)

How many parameters contains the following MLP ?

Model

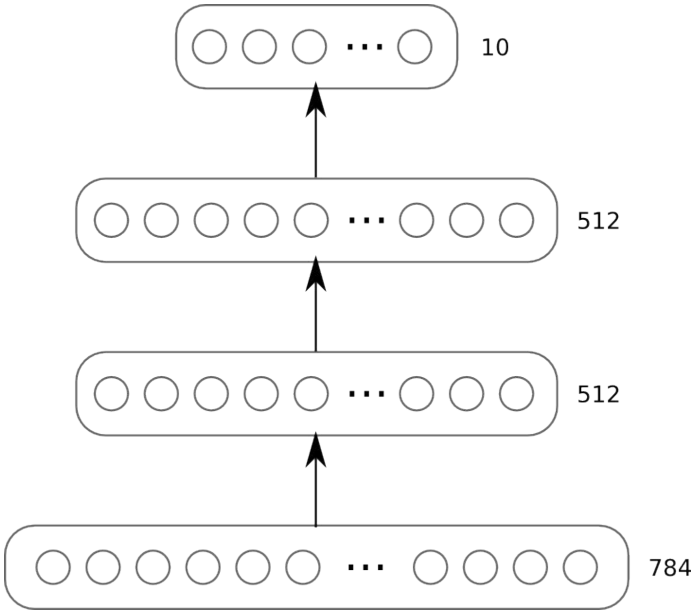
- 3 layer neural network (2 hidden layers)
- Tanh units (activation function)
- 512-512-10



Multi-Layer Perceptron (MLP)

How many parameters contains the following MLP ?

Layer	#Weights	#Biases	Total
1	784 x 512	512	401,920
2	512 x 512	512	262,656
3	512 x 10	10	5,130
			669,706

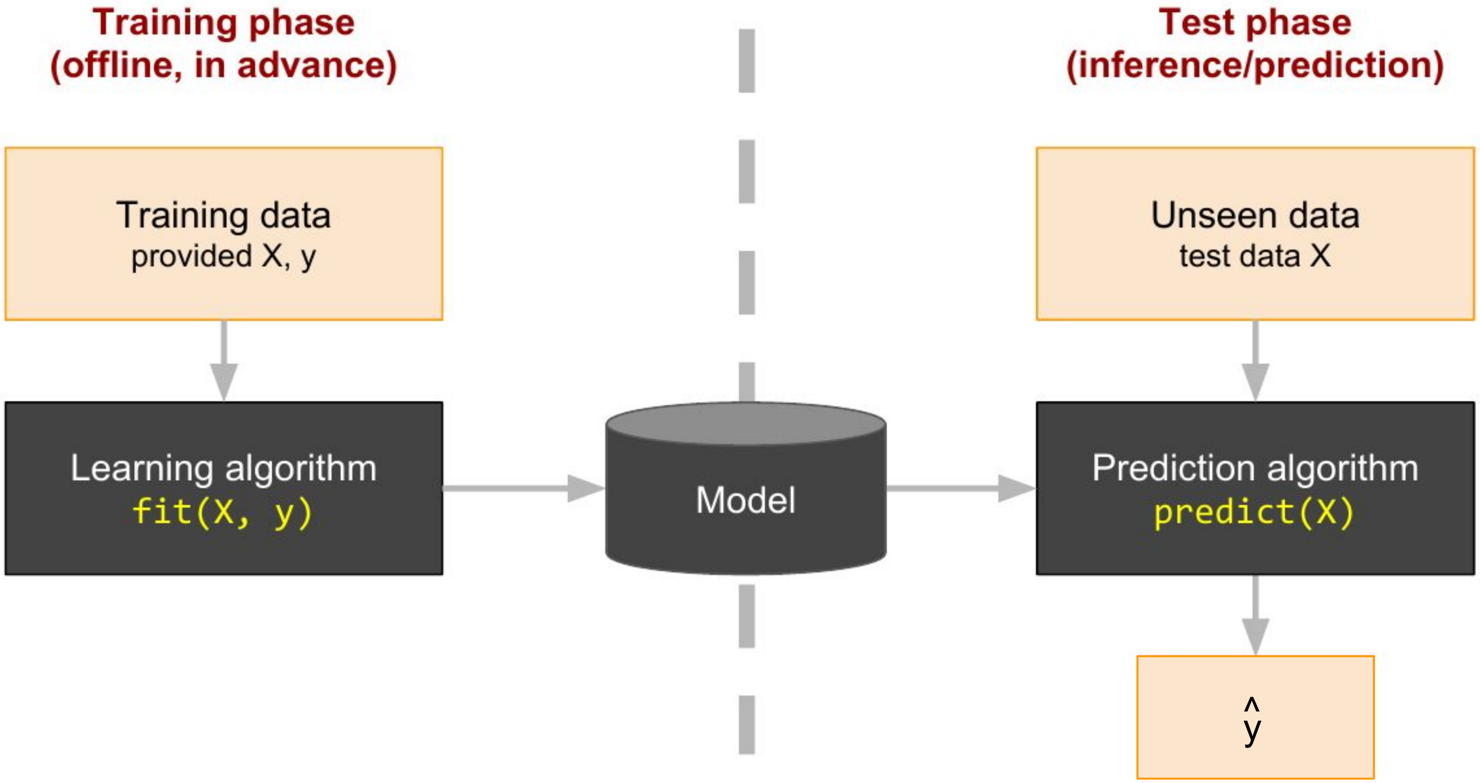


Multi-Layer Perceptron (MLP)

Outline

1. RL with Neural Networks
2. Loss functions
3. Backpropagation
4. Optimizers

Black box abstraction of supervised learning



Slide: Kevin McGuinness (DCU)

Loss function - L(y, ŷ)

loss function

= **cost** function

= **objective** function

= **error** function

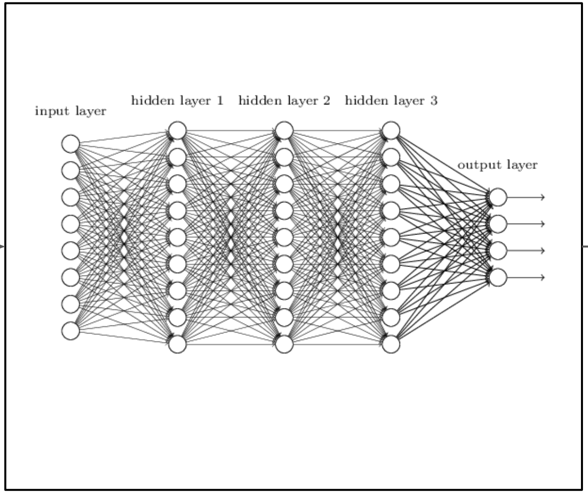
Loss function - L(y, ŷ)

How good does our network with the training data?

Neural Network

input

x



output

$f_\theta(x)$

labels (ground truth)

input

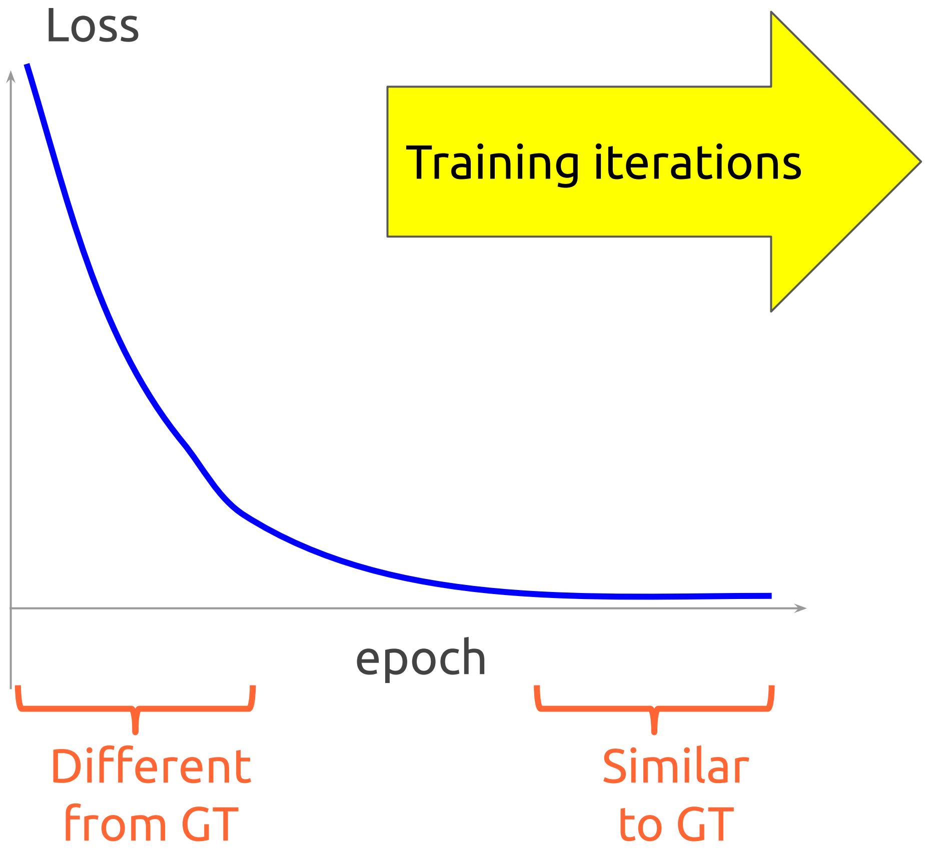
$$\mathcal{L}(w) = distance(f_\theta(x), y)$$

error

parameters (weights, biases)

Loss function - L(y, ŷ)

The Loss value should decrease when the more the NN output matches the ground truth (GT).



Loss functions for regression

Regression: the network predicts **continuous**, **numeric** variables

- Example: Price of a house

$$y = \mathbf{w}^T \cdot \mathbf{x} + b = \ w1 \cdot x1 + w2 \cdot x2 + w3 \cdot x3 + \ldots + wM \cdot xM + b$$

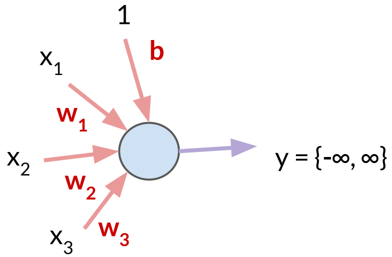
Which loss function would you choose for this problem ?

Loss function - L(y, ŷ)

The **loss function** assesses the performance of our model by comparing its predictions (ŷ) to an expected value (y), typically coming from annotations.

Example: the predicted price (ŷ) and one actually paid (y) could be compared with the Euclidean distance (also referred as L2 distance or Mean Square Error - MSE):

$$y = w_1 \cdot x_1 + w_2 \cdot x_2 + w_3 \cdot x_3 + b = \mathbf{w}^{\mathbf{T}} \cdot \mathbf{x} + b$$



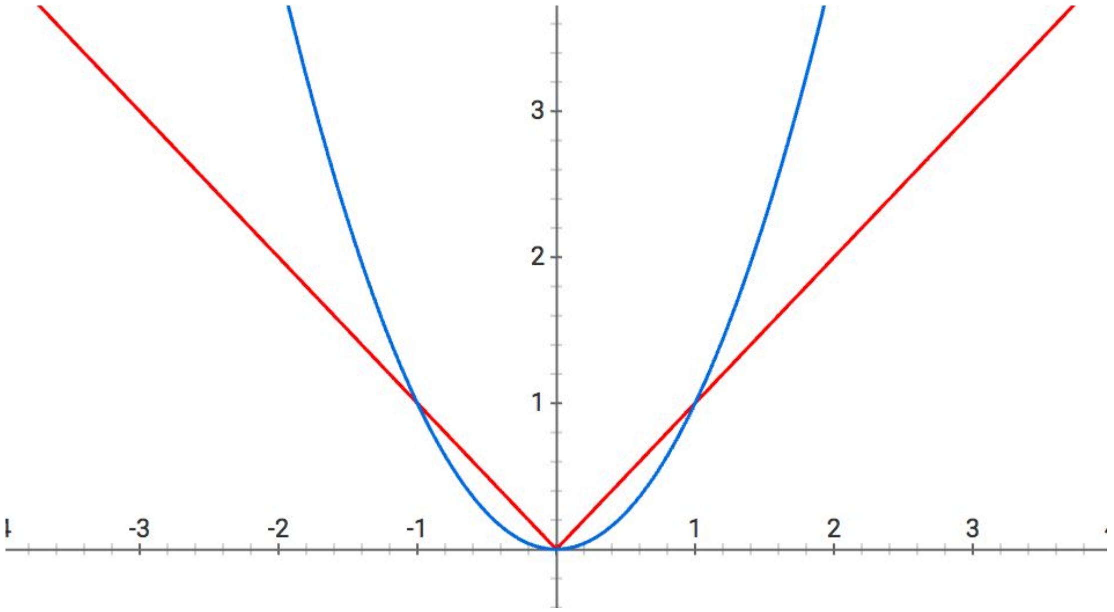
Loss functions for regression

L1 Loss

$$\mathcal{L} = \frac{1}{n}\sum_{i=1}^{n} |y_i - f_\theta(x_i)|$$

L2 Loss

$$\mathcal{L} = \frac{1}{n}\sum_{i=1}^{n} (y_i - f_\theta(x_i))^2$$



Which of these two losses is lighter to compute ?

Figure: Amod Kolwalkar, “Regularization in Machine Learning and Deep Learning” (2019)

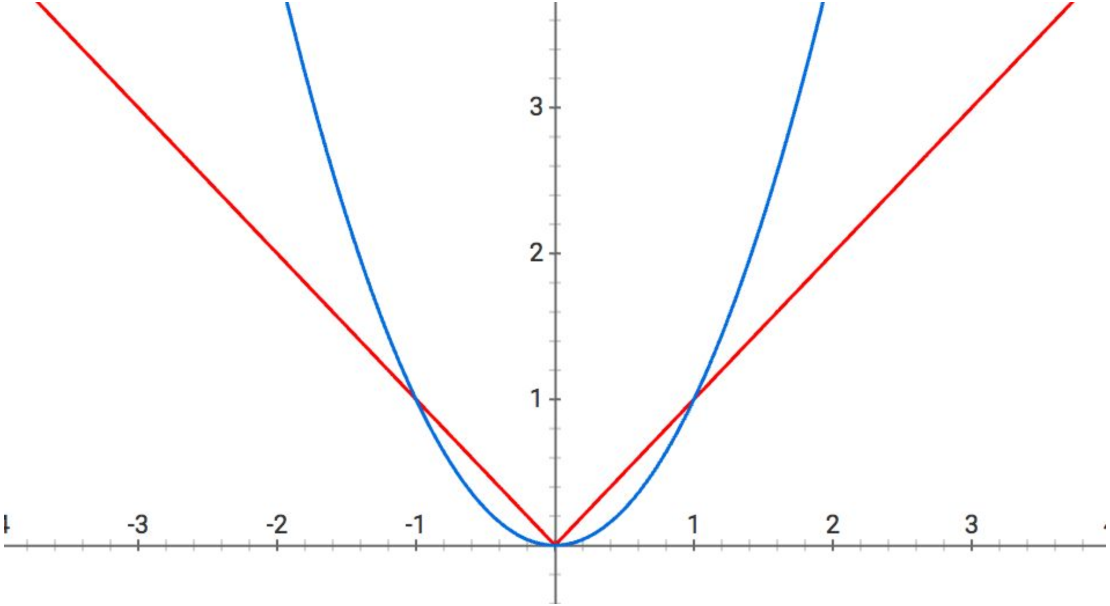
Loss functions for regression

L1 Loss

$$\mathcal{L} = \frac{1}{n}\sum_{i=1}^{n} |y_i - f_\theta(x_i)|$$

L2 Loss

$$\mathcal{L} = \frac{1}{n}\sum_{i=1}^{n} (y_i - f_\theta(x_i))^2$$

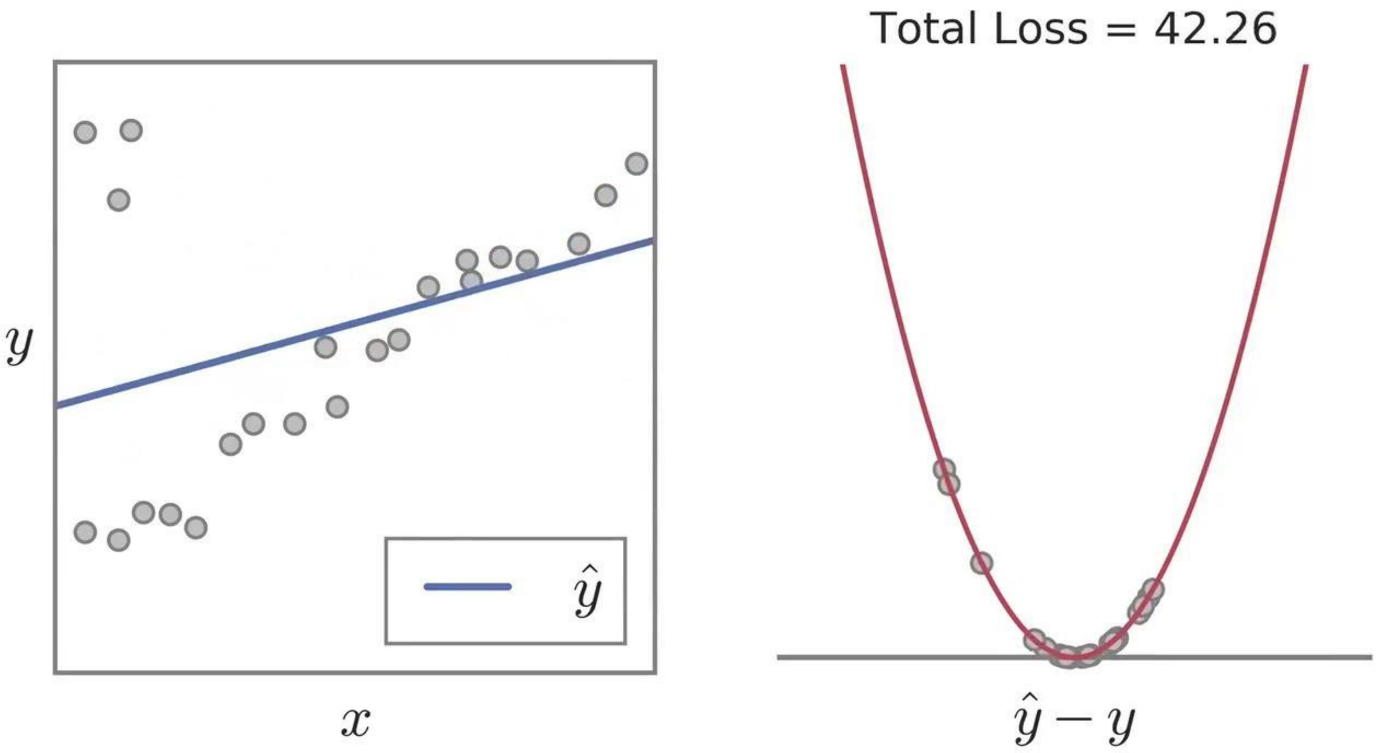


Which of these two losses is more sensitive to outliers ($|y_i - f_\theta(x_i)| >> 1$) ?

Figure: Amod Kolwalkar, "Regularization in Machine Learning and Deep Learning" (2019)

Loss functions for regression

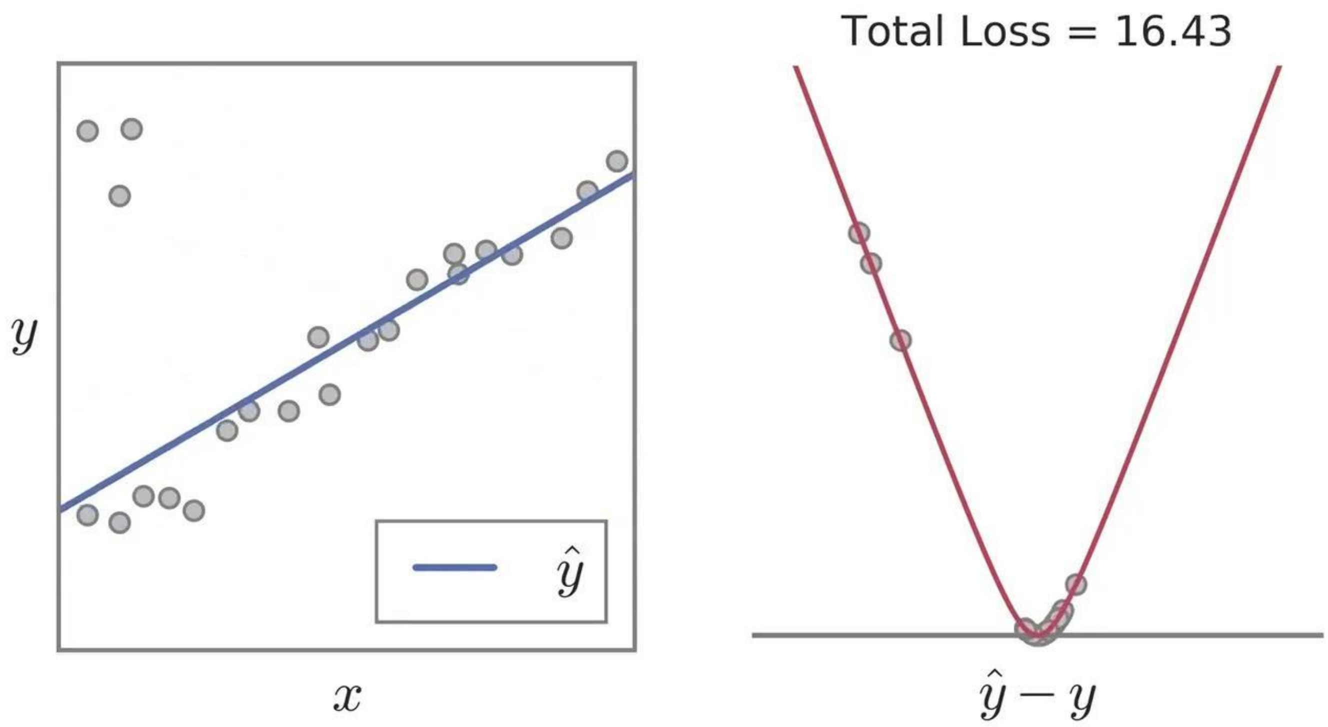
<u>L2 loss</u> has problems when there are outliers in the data



Source: Barron, J. T. A general and adaptive robust loss function. CVPR 2019 [slides] [Tweet by Ankur Handa]

Loss functions for regression

Pseudo-Huber loss (Charbonnier, Smooth L_1) is generally more resilient to outliers than L2.



Source: Barron, J. T. A general and adaptive robust loss function. CVPR 2019 [slides] [Tweet by Ankur Handa]

Outline

1. RL with Neural Networks
2. Loss functions
3. **Backpropagation**
4. Optimizers

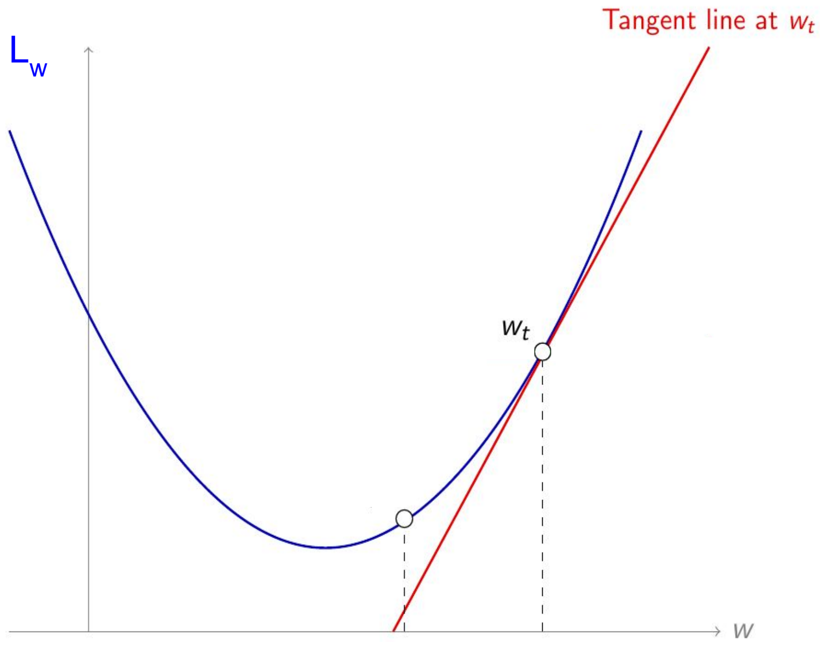
Backpropagation

Discussion: Consider the single parameter model...

$$\hat{y} = x \cdot w$$

...and that, given a pair $(y, \hat{y})$, we would like to update the current w_t value to a new w_{t+1} based on the loss function L_w.

(a) Would you increase or decrease w_t?
(b) What operation could indicate which way to go?
(c) How much would you increase or decrease w_t?



Gradient Descent (GD)

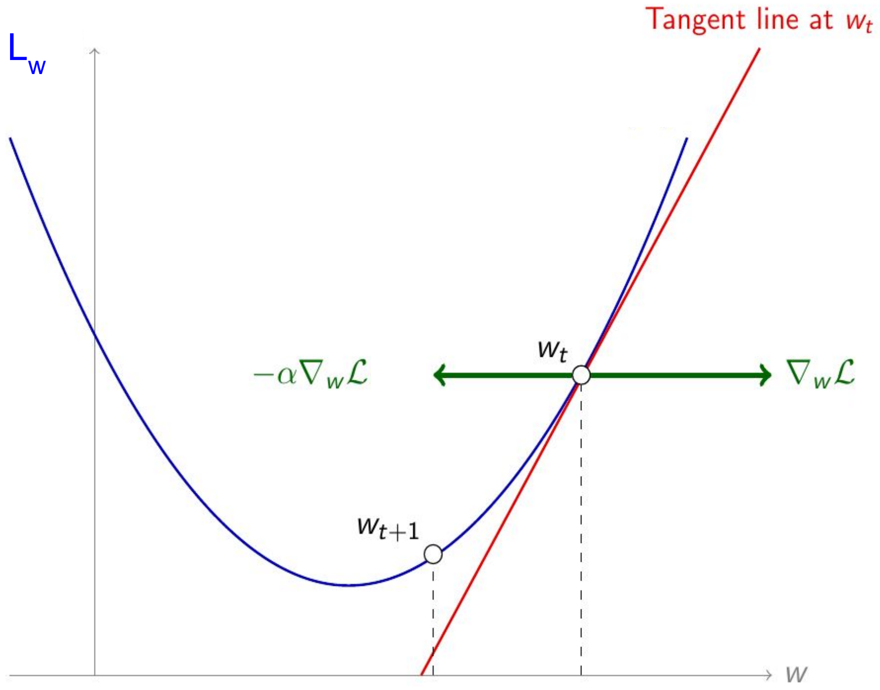
<u>Motivation for this lecture</u>:

if we had a way to estimate the gradient of the loss (∇L) with respect to the parameter(s), we could use gradient descent to optimize them.

Descend
(minus sign)

$$w_{t+1} \leftarrow w_t - \alpha \nabla \mathcal{L}_w(w_t)$$

Learning rate (LR)

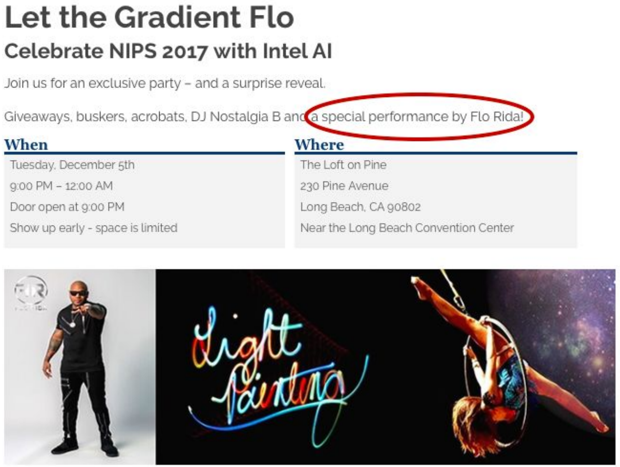


Gradient Descent (GD)

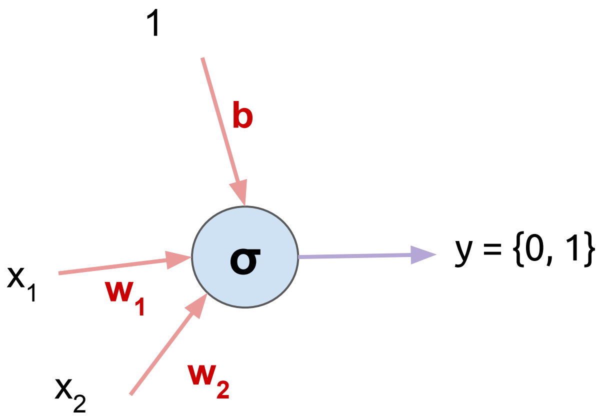
Backpropagation will allow us to compute the **gradients of the loss function** with respect to:

- all model parameters (**w & b**) - final goal during training
- input/intermediate data - visualization & interpretability purposes.

Gradients will **“flow”** from the output of the model towards the input (“back”).

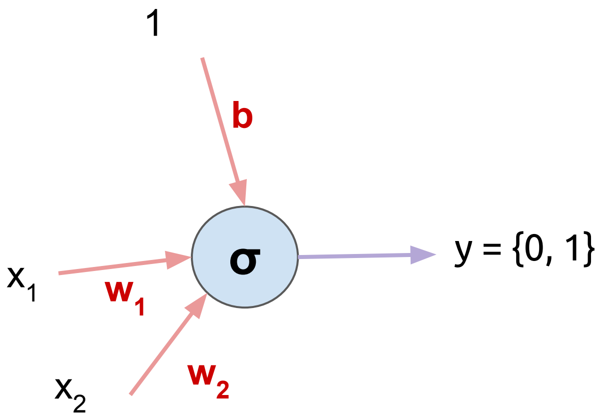


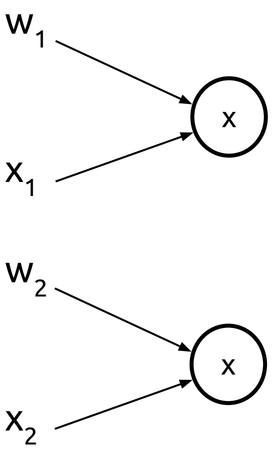
Computational graph of a simple perceptron



<u>Question</u>: What is the order of operations of this perceptron with a sigmoid activation?

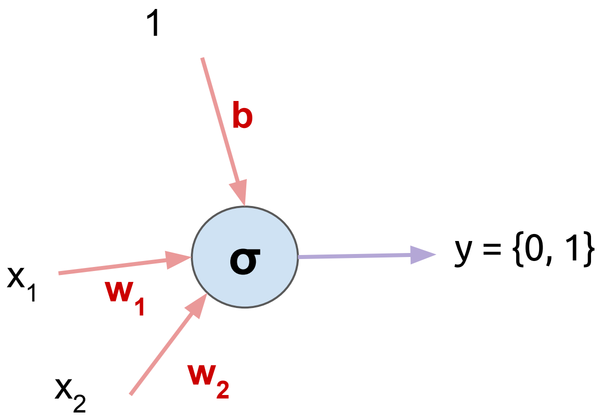
Computational graph of a perceptron

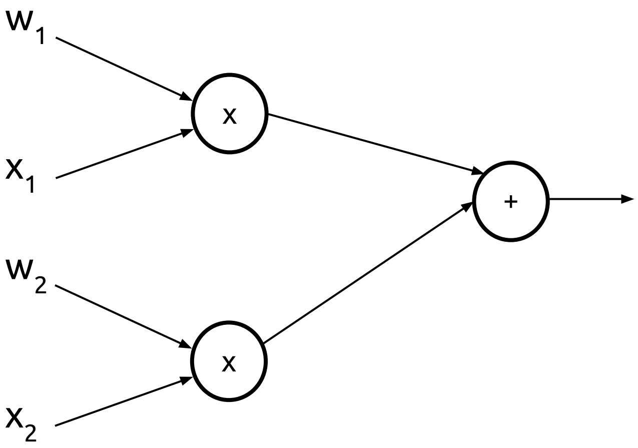




Example adapted from [“CS231n: Convolutional Neural Networks for Visual Recognition”](), Stanford University.

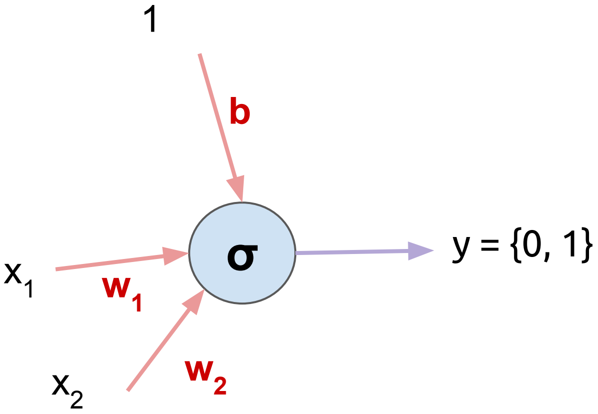
Computational graph of a perceptron

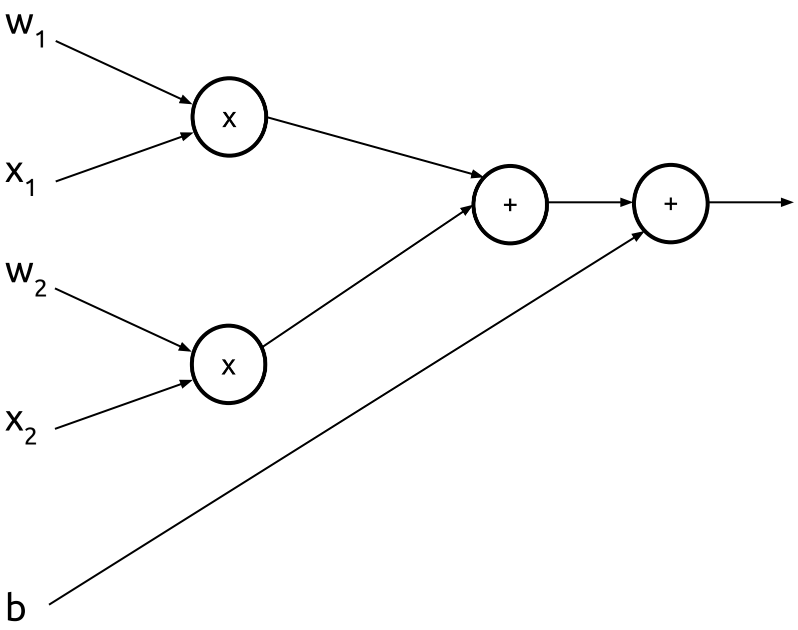




Example adapted from [“CS231n: Convolutional Neural Networks for Visual Recognition”](), Stanford University.

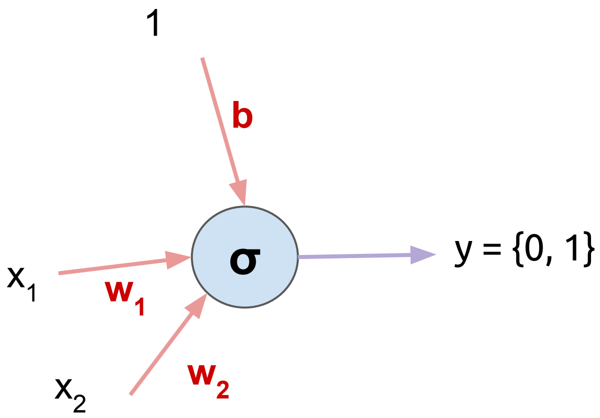
Computational graph of a perceptron

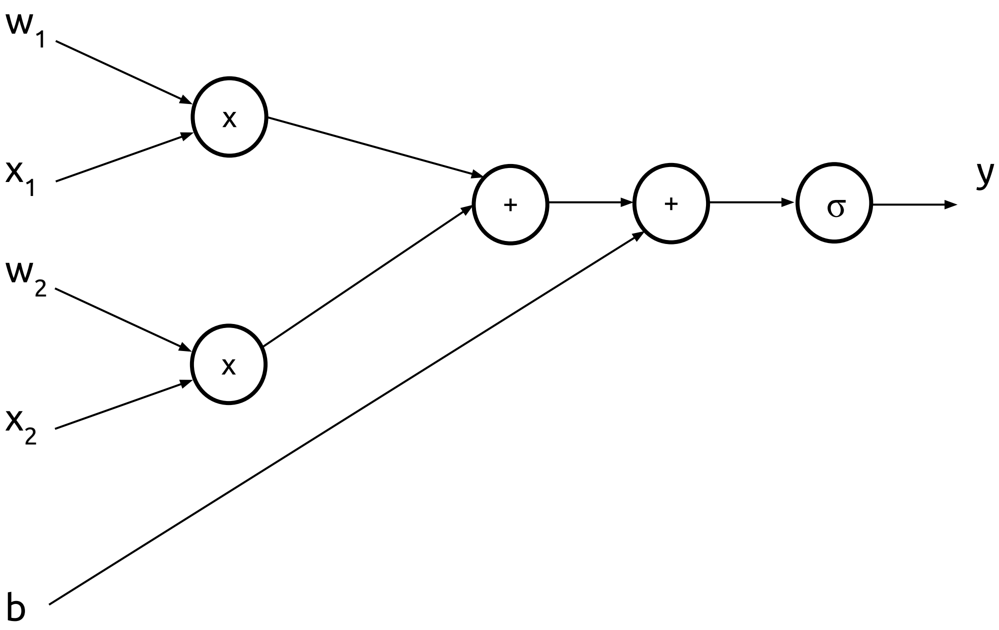




Example adapted from [“CS231n: Convolutional Neural Networks for Visual Recognition”](), Stanford University.

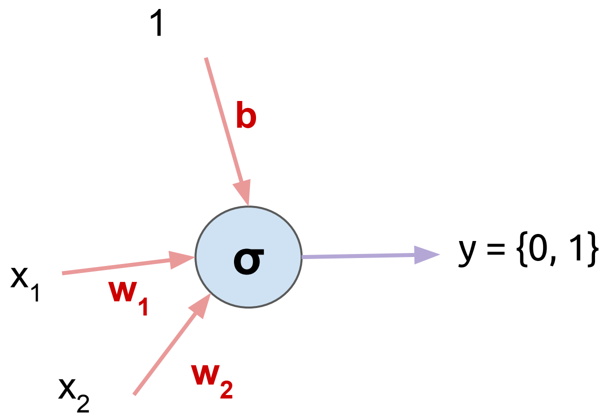
Computational graph of a perceptron

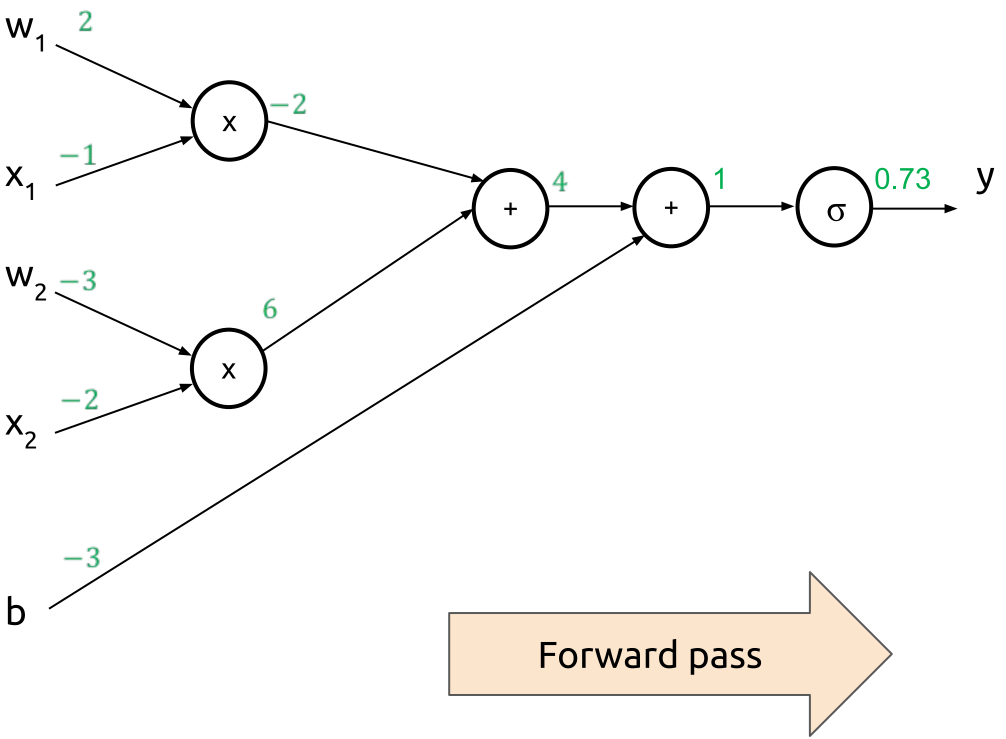




Example adapted from "CS231n: Convolutional Neural Networks for Visual Recognition", Stanford University.

Computational graph of a perceptron





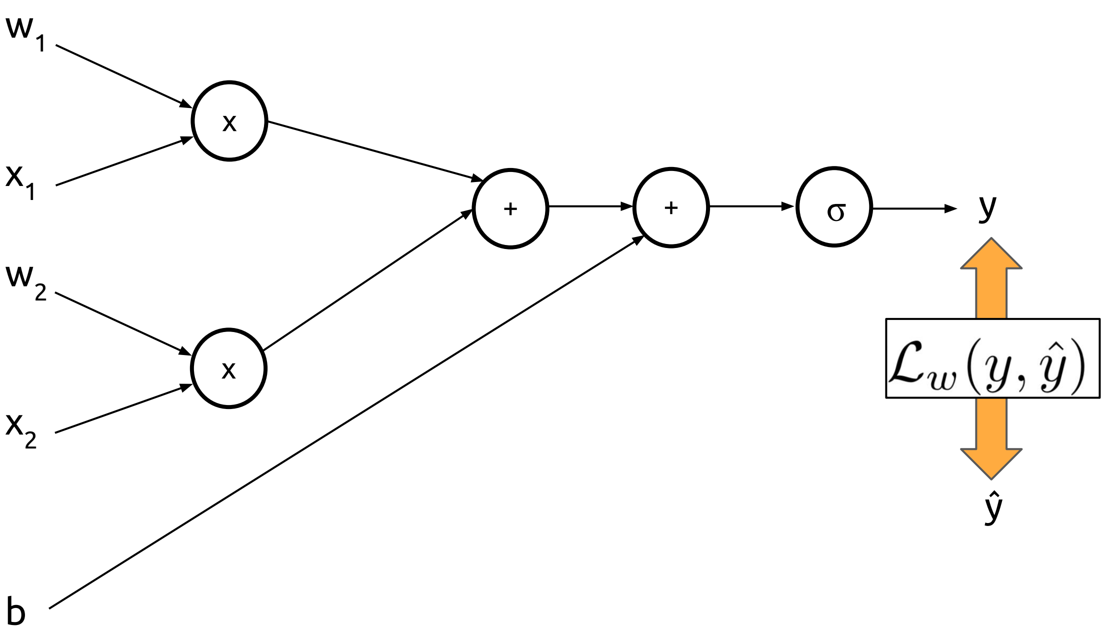
Computational graph of a perceptron

Challenge: How to compute the gradient of the loss function with respect to w_1, w_2 or b?

$$\frac{\partial \mathcal{L}(y, \hat{y})}{\partial w_1} = ?$$

$$\frac{\partial \mathcal{L}(y, \hat{y})}{\partial w_2} = ?$$

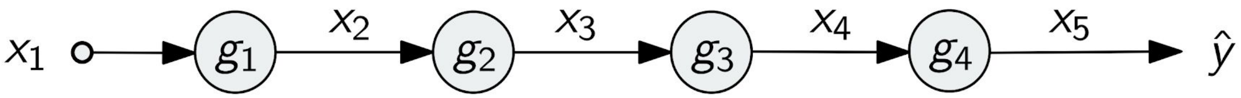
$$\frac{\partial \mathcal{L}(y, \hat{y})}{\partial b} = ?$$



$\mathcal{L}_w(y, \hat{y})$

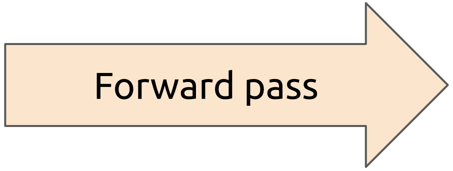
Gradients from composition (chain rule)

$$\hat{y} = g_4(g_3(g_2(g_1(x_1))))$$



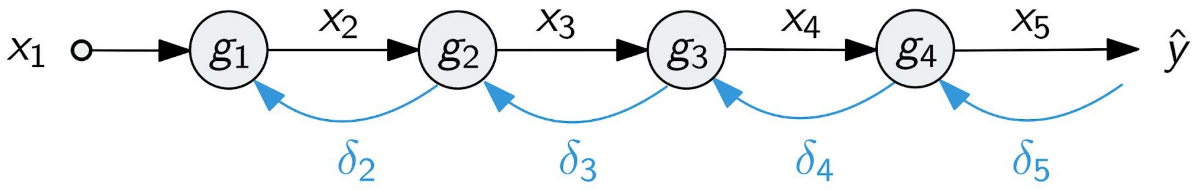
Decompose into steps (**forward propagation**):

$$\begin{aligned} x_2 &= g_1(x_1) \\ x_3 &= g_2(x_2) \\ x_4 &= g_3(x_3) \\ \hat{y} = x_5 &= g_4(x_4) \end{aligned}$$



Gradients from composition (chain rule)

$$\hat{y} = g_4(g_3(g_2(g_1(x_1))))$$



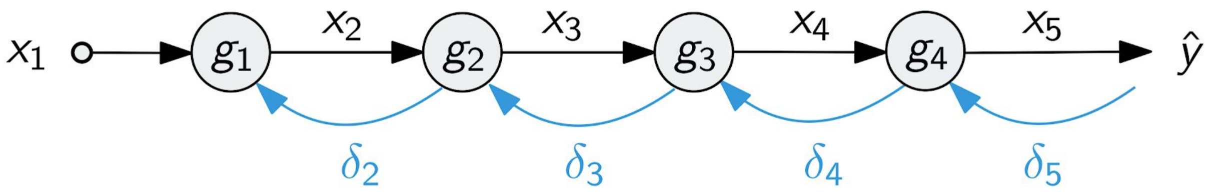
Want to find $\frac{\partial \hat{y}}{\partial x_1}$. Chain rule:

$$\frac{\partial \hat{y}}{\partial x_1} = \frac{\partial \hat{y}}{\partial x_4} \frac{\partial x_4}{\partial x_3} \frac{\partial x_3}{\partial x_2} \frac{\partial x_2}{\partial x_1}$$

How does a variation ("difference") on the input affect the prediction ?

Backward pass

Gradients from composition (chain rule)

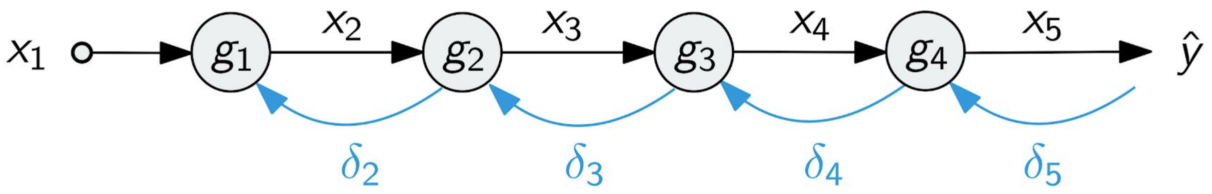


Decompose into steps again. Let $\delta_k = \frac{\partial \hat{y}}{\partial x_k}$. **Backpropagation**:

$$\delta_5 = \frac{\partial \hat{y}}{\partial x_5} = 1$$

A variation in x_5 directly affects on ŷ with a 1:1 factor.

Gradients from composition (chain rule)



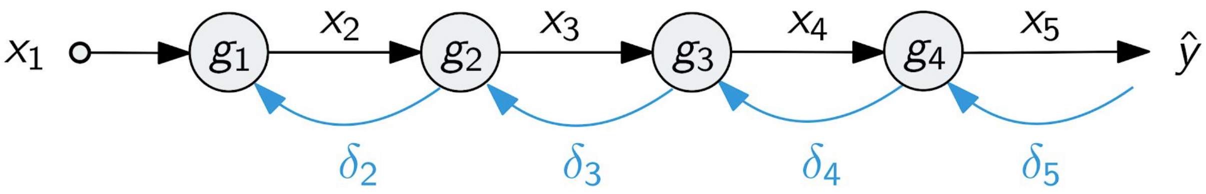
Decompose into steps again. Let $\delta_k = \frac{\partial \hat{y}}{\partial x_k}$. **Backpropagation**:

How does a variation on x_4 affect the predicted $\hat{y}$?

$$\delta_5 = \frac{\partial \hat{y}}{\partial x_5} = 1$$

$$\delta_4 = \frac{\partial \hat{y}}{\partial x_4} = \frac{\partial \hat{y}}{\partial x_5} \frac{\partial x_5}{\partial x_4} = \delta_5 g_4'(x_4)$$

Gradients from composition (chain rule)



Decompose into steps again. Let $\delta_k = \frac{\partial \hat{y}}{\partial x_k}$. **Backpropagation**:

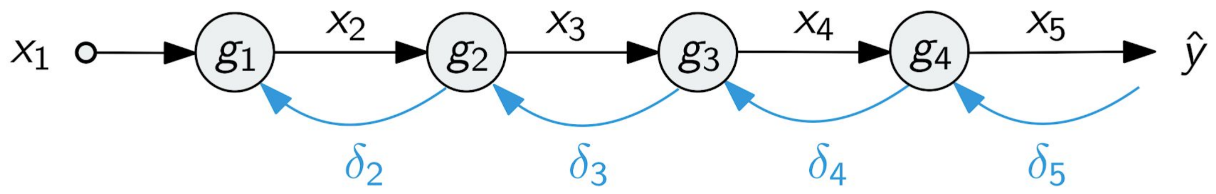
$$\delta_5 = \frac{\partial \hat{y}}{\partial x_5} = 1$$

$$\delta_4 = \frac{\partial \hat{y}}{\partial x_4} = \frac{\partial \hat{y}}{\partial x_5}\frac{\partial x_5}{\partial x_4} = \delta_5 g_4'(x_4)$$

How does a variation on x_4 affect the predicted $\hat{y}$?

It corresponds to how a variation of x_5 affects $\hat{y}$...

Gradients from composition (chain rule)



Decompose into steps again. Let $\delta_k = \frac{\partial \hat{y}}{\partial x_k}$. **Backpropagation**:

$$\delta_5 = \frac{\partial \hat{y}}{\partial x_5} = 1$$

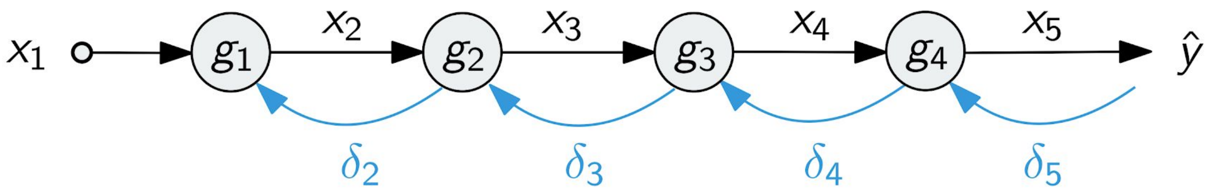
$$\delta_4 = \frac{\partial \hat{y}}{\partial x_4} = \frac{\partial \hat{y}}{\partial x_5} \frac{\partial x_5}{\partial x_4} = \delta_5 g_4'(x_4)$$

How does a variation on x_4 affect the predicted $\hat{y}$?

It corresponds to how a variation of x_5 affects $\hat{y}$ ...

...**multiplied** by how a variation near the input x_4 affects the output $g_4(x_4)$.

Gradients from composition (chain rule)



The same reasoning can be iteratively applied until reaching $\dfrac{\partial \hat{y}}{\partial x_1}$:

$$\delta_5 = \frac{\partial \hat{y}}{\partial x_5} = 1$$

$$\delta_4 = \frac{\partial \hat{y}}{\partial x_4} = \frac{\partial \hat{y}}{\partial x_5}\frac{\partial x_5}{\partial x_4} = \delta_5 g_4'(x_4)$$

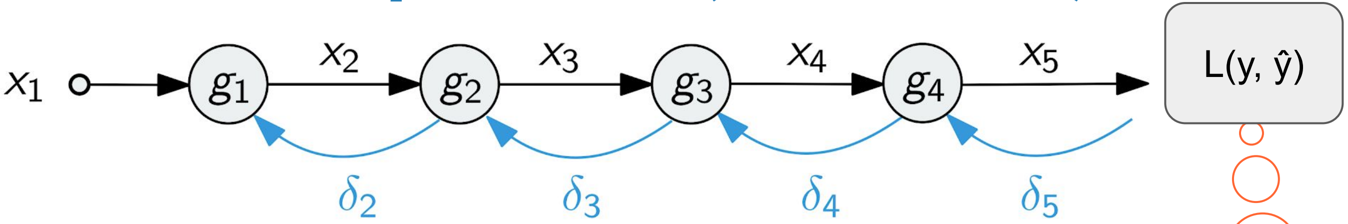
$$\delta_3 = \frac{\partial \hat{y}}{\partial x_3} = \frac{\partial \hat{y}}{\partial x_4}\frac{\partial x_4}{\partial x_3} = \delta_4 g_3'(x_3)$$

$$\delta_2 = \frac{\partial \hat{y}}{\partial x_2} = \frac{\partial \hat{y}}{\partial x_3}\frac{\partial x_3}{\partial x_2} = \delta_3 g_2'(x_2)$$

$$\delta_1 = \boxed{\frac{\partial \hat{y}}{\partial x_1}} = \frac{\partial \hat{y}}{\partial x_2}\frac{\partial x_2}{\partial x_1} = \delta_2 g_1'(x_1)$$

Backward pass

Gradients from composition (chain rule)



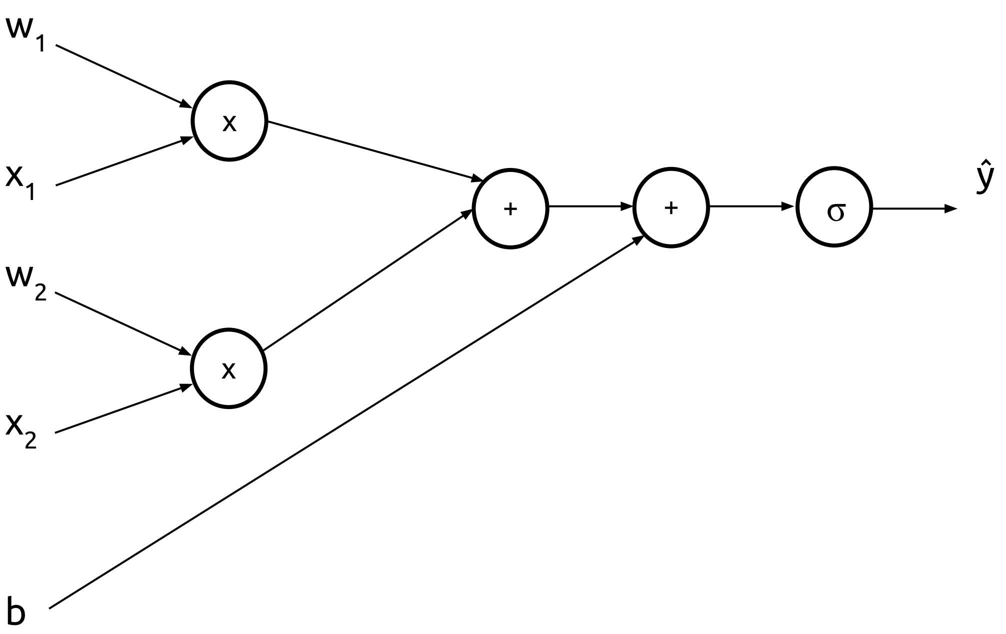
When training NN, we will actually compute the derivative over the loss function with respect the weights and biases

Backward pass

Gradients from composition (chain rule)

Question: What are the derivatives of the function involved in the computational graph of a perceptron ?

- SUM (+) $\frac{\partial(a+b)}{\partial a}$
- PRODUCT (x) $\frac{\partial(a \cdot b)}{\partial a}$
- SIGMOID (σ) $\frac{\partial \sigma(x)}{\partial x}$



Gradient weights for sigmoid σ

$$\frac{\partial \sigma(x)}{\partial x} = \frac{\partial}{\partial x}\left(\frac{1}{1+e^{-x}}\right) \overset{(*)}{=} \frac{-1}{(1+e^{-x})^2}\frac{\partial(1+e^{-x})}{\partial x} = \frac{-1}{(1+e^{-x})^2}\frac{\partial(e^{-x})}{\partial x}$$

(*) $f(x) = \frac{g(x)}{h(x)} \qquad f'(x) = \frac{g'(x)h(x) - g(x)h'(x)}{h^2}$

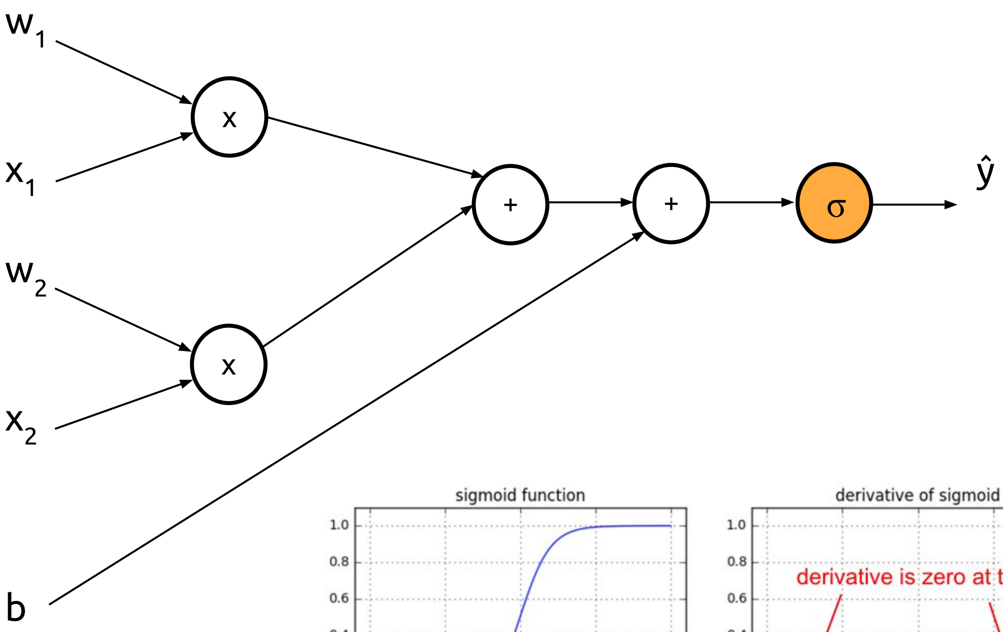
$$\frac{\partial \sigma(x)}{\partial x} = \frac{-1}{(1+e^{-x})^2}(-e^{-x}) = \frac{e^{-x}}{(1+e^{-x})^2}$$

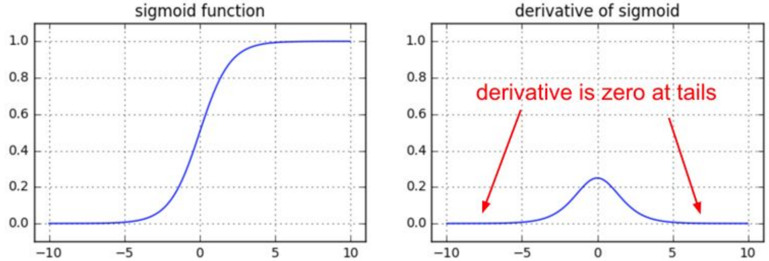
...which can be re-arranged as...

$$\frac{\partial \sigma(x)}{\partial x} = \frac{e^{-x}}{(1+e^{-x})^2} = \frac{e^{-x}}{(1+e^{-x})}\frac{1}{(1+e^{-x})}$$

$$\frac{\partial \sigma(x)}{\partial x} = \left(\frac{1+e^{-x}}{1+e^{-x}} - \frac{1}{1+e^{-x}}\right)\sigma(x)$$

$$\frac{\partial \sigma(x)}{\partial x} = (1-\sigma(x))\,\sigma(x)$$





Even more details: Arunava, [“Derivative of the Sigmoid function”]() (2018)

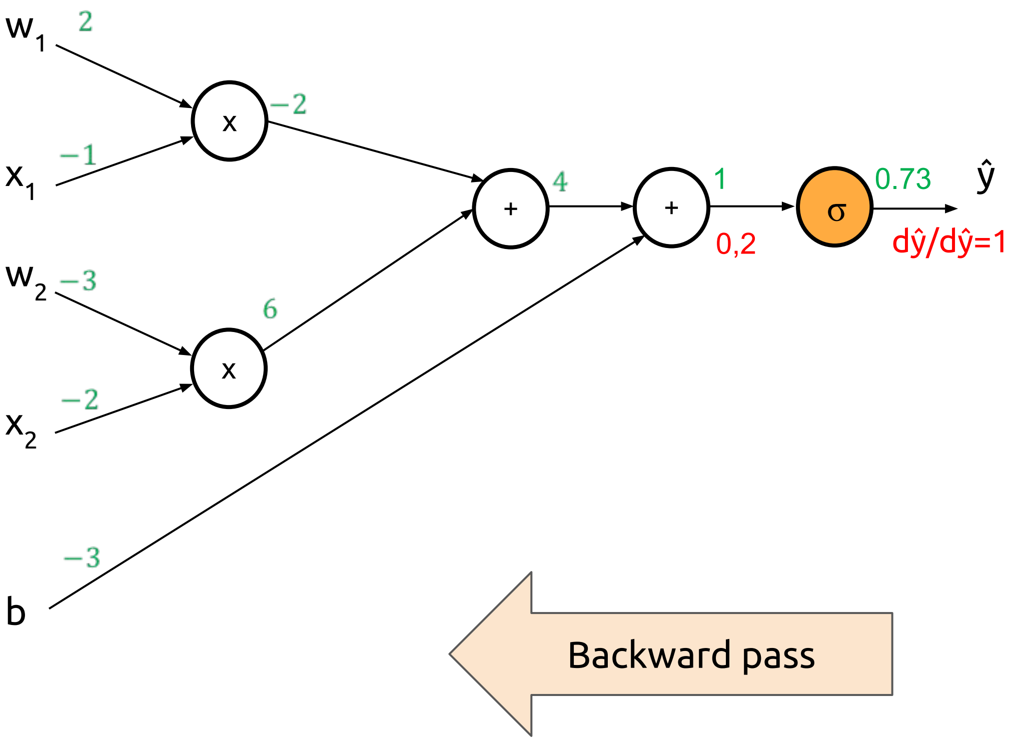
Figure: [Andrej Karpathy]()

Gradient backpropagation in a perceptron

$$\frac{\partial \sigma(x)}{\partial x} = (1 - \sigma(x))\,\sigma(x)$$

```python
import math

dot=1

# sigmoid function
f = 1.0 / (1 + math.exp(-dot))

# gradient on dot variable,
ddot = (1 - f) * f

print(ddot)
```

```
0.19661193324148185
```



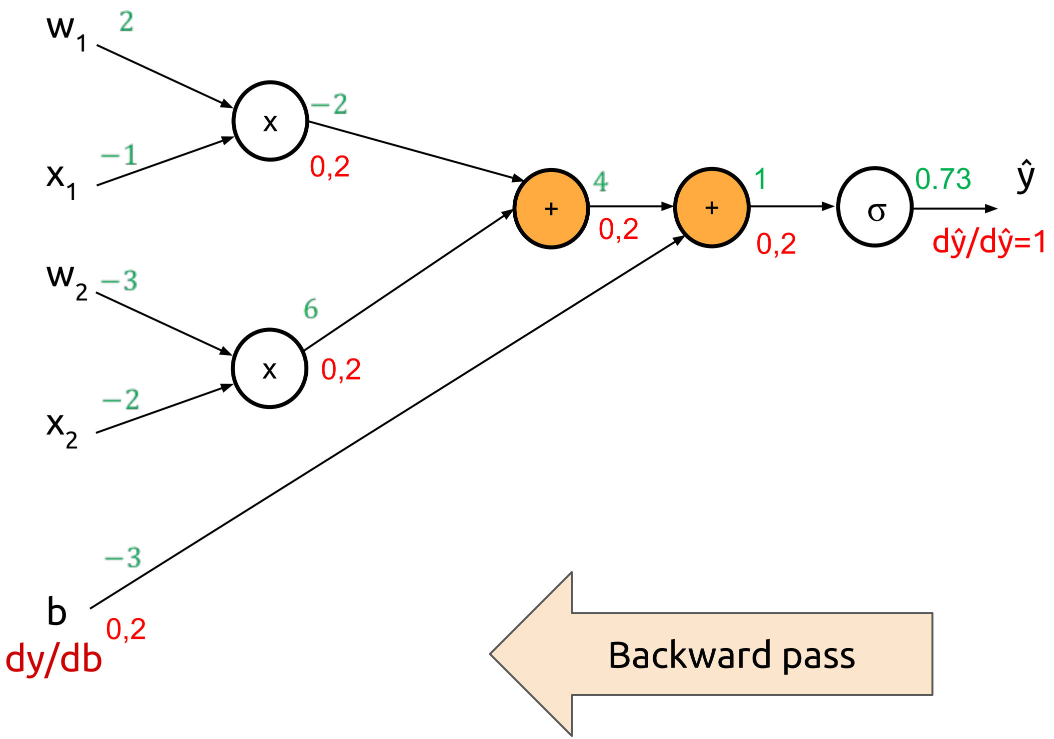
Example extracted from Andrej Karpathy's notes for CS231n from Stanford University.

Gradient backpropagation in a perceptron

SUM

$$\frac{\partial(a+b)}{\partial a} = 1$$

$$\frac{\partial(a+b)}{\partial b} = 1$$



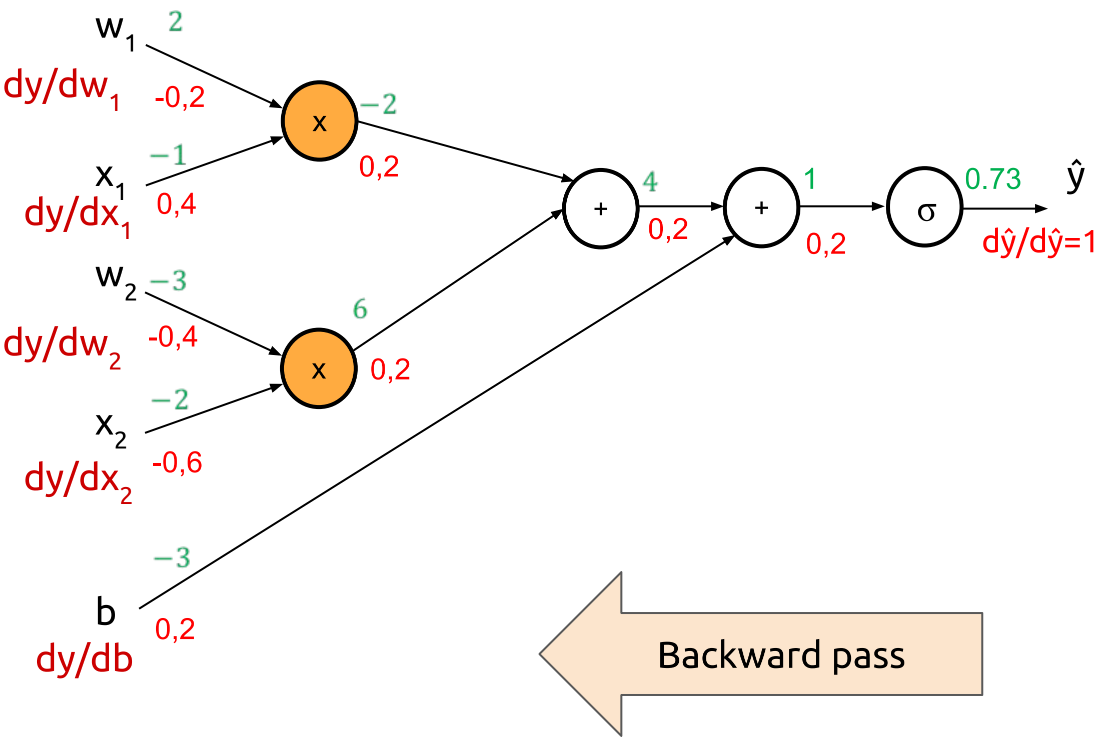
Example extracted from Andrej Karpathy's notes for CS231n from Stanford University.

Gradient backpropagation in a perceptron

Product

$$\frac{\partial(a \cdot b)}{\partial a} = b$$

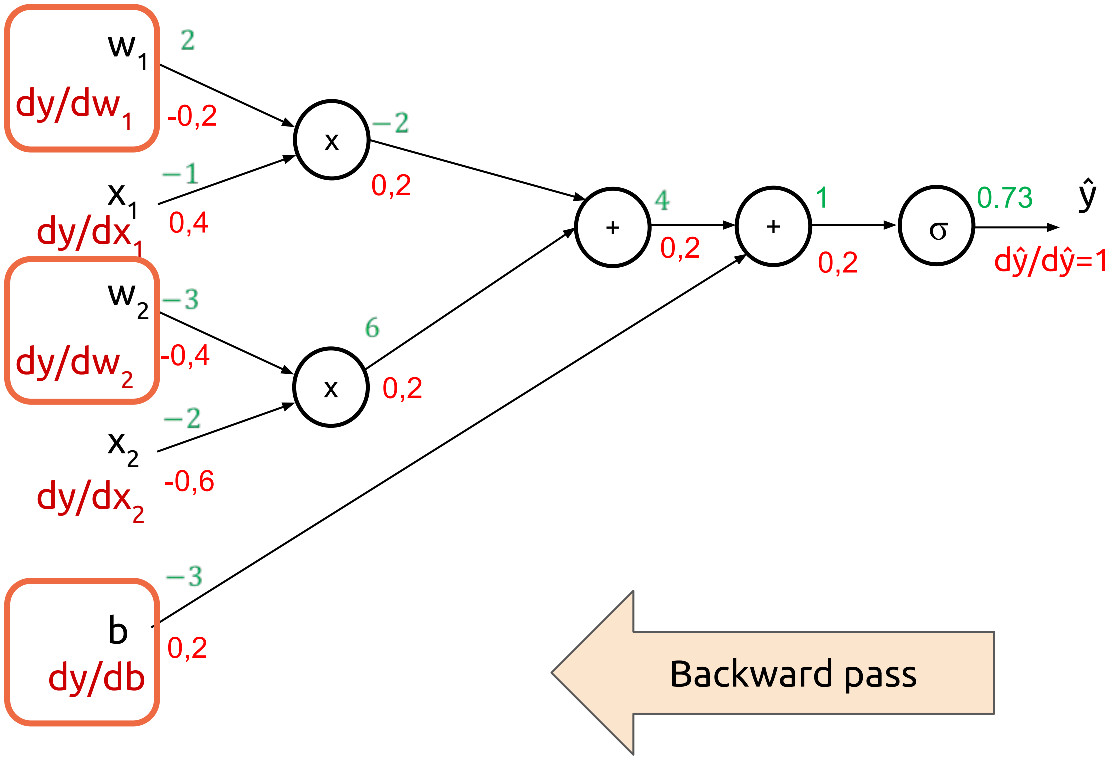
$$\frac{\partial(a \cdot b)}{\partial b} = a$$



Example extracted from Andrej Karpathy's notes for CS231n from Stanford University.

Gradient backpropagation in a perceptron

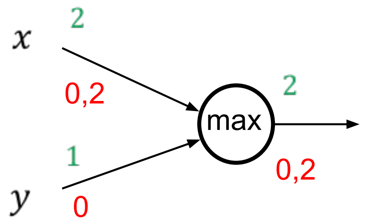
Normally, we will be interested only on the weights (w_i) and biases (b), not the inputs (x_i). The weights are the parameters to learn in our models.



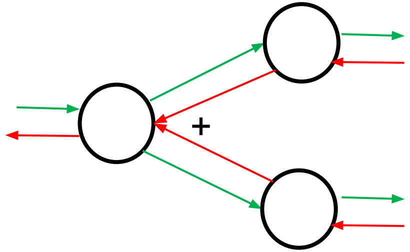
Example extracted from [Andrej Karpathy's notes]for CS231n from Stanford University.

(bonus) Gradients weights for MAX & SPLIT

Max: Routes the gradient only to the higher input branch (not sensitive to the lower branches).



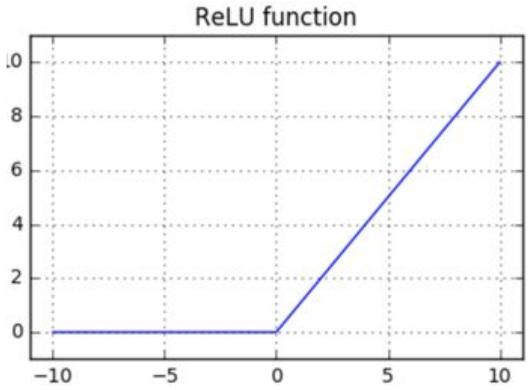
Split: Branches that split in the forward pass and merge in the backward pass, add gradients

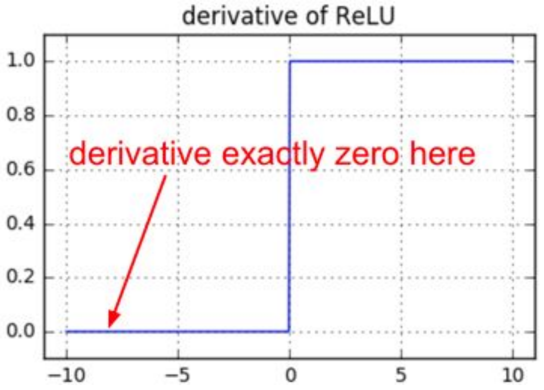


(bonus) Gradient weights for ReLU

$$ReLU(x) = \left\{ \begin{array}{ll} x & \text{if } x \geq 0 \\ 0 & \text{if } x < 0 \end{array} \right\}$$

$$\frac{\partial ReLU(x)}{\partial x} = u(x) = \left\{ \begin{array}{ll} 1 & \text{if } x > 0 \\ 0 & \text{if } x < 0 \end{array} \right\}$$

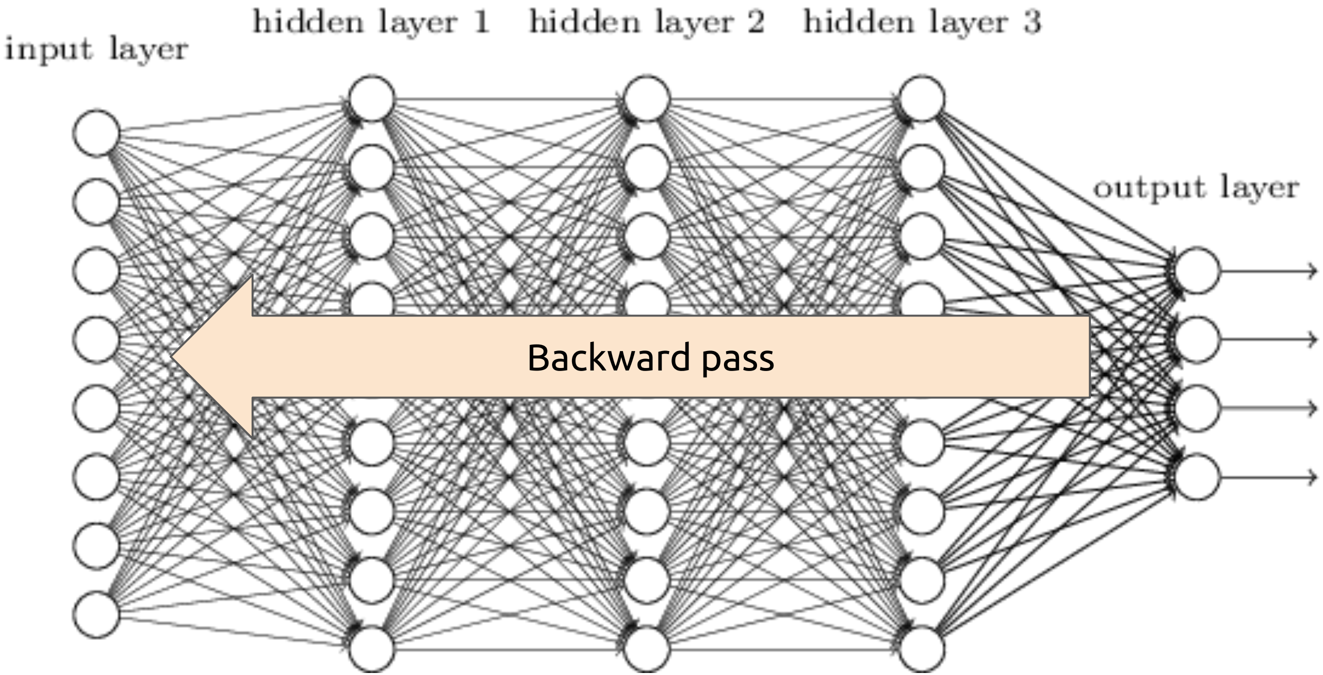




Figures: [Andrej Karpathy]()

Backpropagation across layers

Gradients can flow across stacked layers of neurons to estimate their parameters.



Backpropagation & RL

Greg Brockman @gdb

Seguint

For differentiable problems, there's backpropagation. For everything else, there's RL.

Tradueix el tuit

18:11 - 31 de gen. de 2019

Yann LeCun @ylecun

Seguint

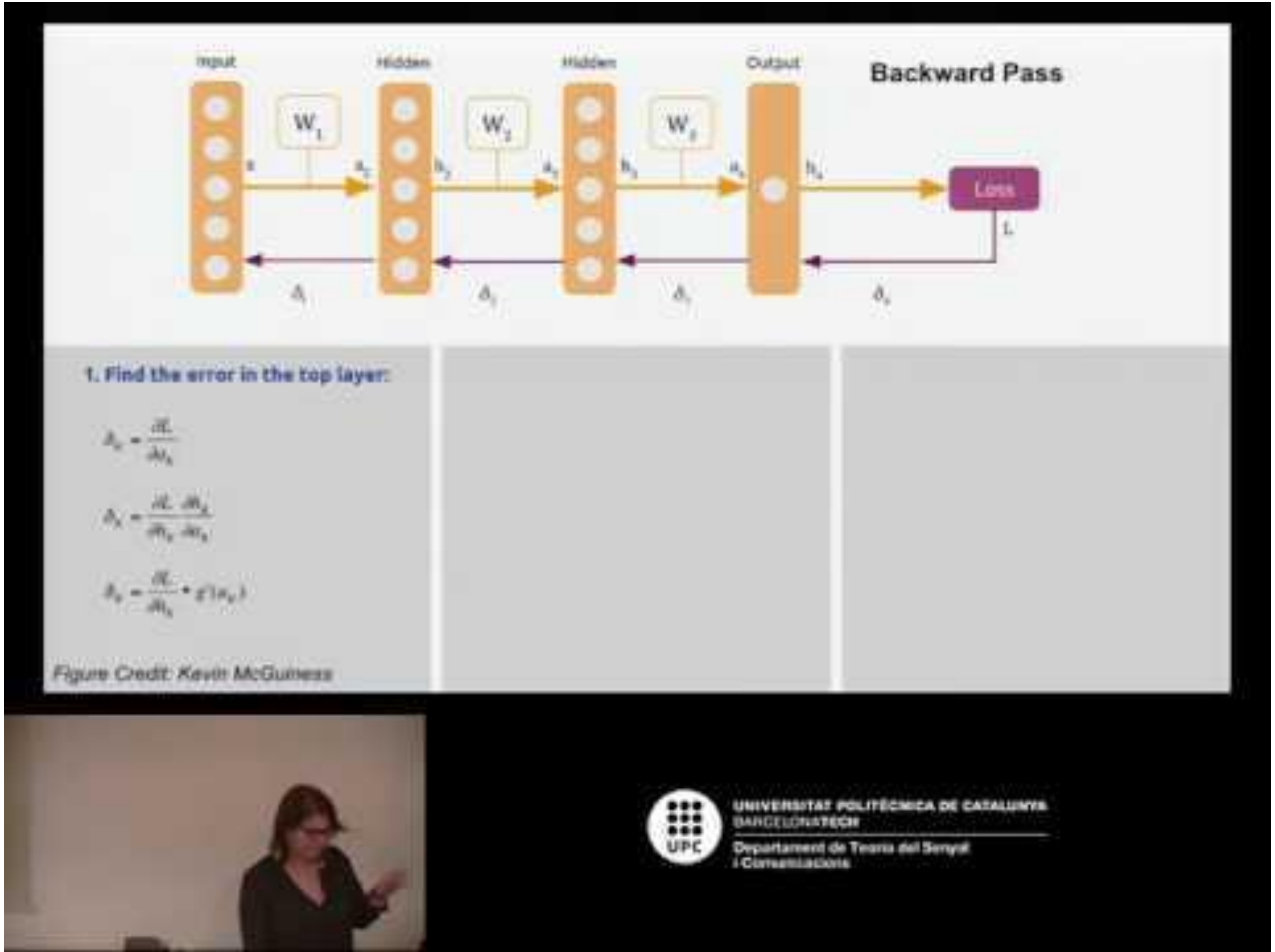
En resposta a @gdb

Not quite right.
A more accurate statement would be "for everything else, there is gradient-free (zeroth-order) optimization."
RL is when there is a sequential decision process and what you see depends on previous actions you took.

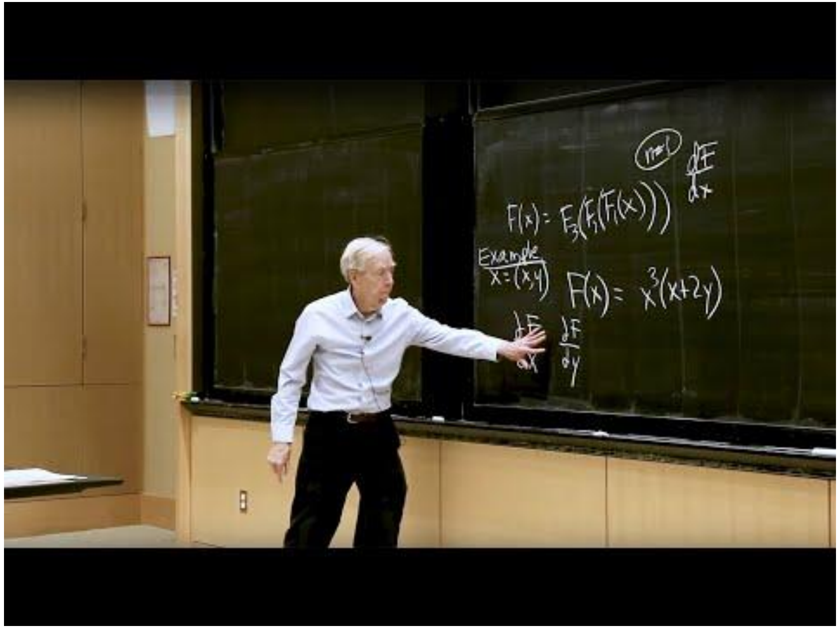
Tradueix el tuit

2:38 - 1 de febr. de 2019

Backpropagation: Learn more



Backpropagation: Learn more



Gilbert Strang, “27. Backpropagation: Find Partial Derivatives”. MIT 18.065 (2018)

Creative Commons, “Yoshua Bengio Extra Footage 1: Brainstorm with students” (2018)

Outline

1. RL with Neural Networks
2. Loss functions
3. Backpropagation
4. **Optimizers**

Gradient Descent (GD)

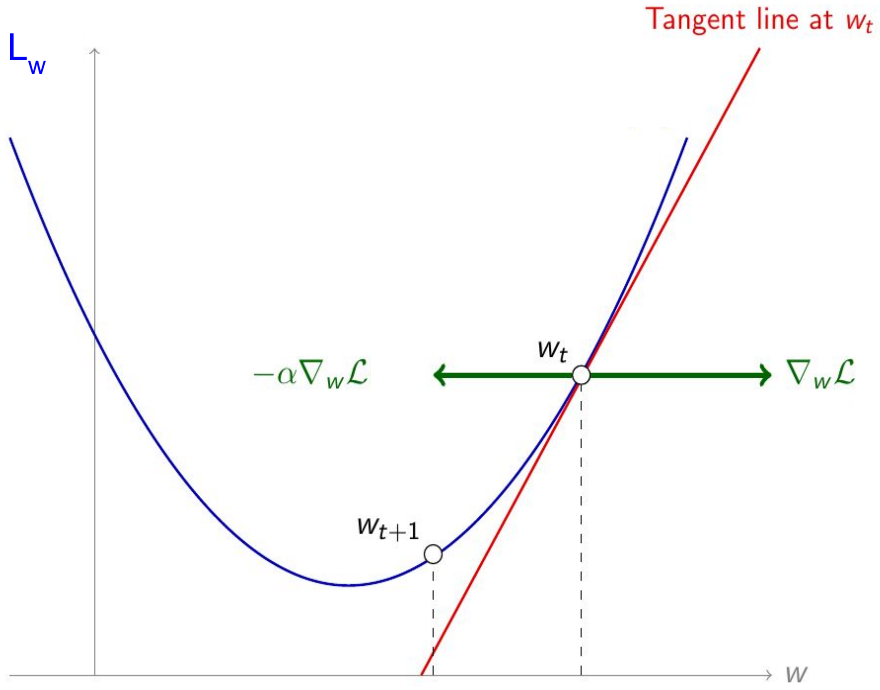
Motivation for this lecture:

if we had a way to estimate the gradient of the loss (∇L) with respect to the parameter(s), we could use gradient descent to optimize them.

Descend
(minus sign)

$$w_{t+1} \leftarrow w_t - \alpha \nabla \mathcal{L}_w(w_t)$$

Learning rate (LR)



Gradient descent (GD)

Computing the gradient for the full dataset at each step is slow

- Especially if the dataset is large!

For most losses we care about, the total loss can be expressed as a sum (or average) of losses on the individual examples

$$\mathcal{L} = \frac{1}{N}\sum_{i=1}^{N} L(\mathbf{y}_i, \hat{\mathbf{y}}_i)$$

The gradient is the average of the gradients on individual examples

$$\nabla\mathcal{L} = \frac{1}{N}\sum_{i=1}^{N} \nabla L(\mathbf{y}_i, \hat{\mathbf{y}}_i)$$

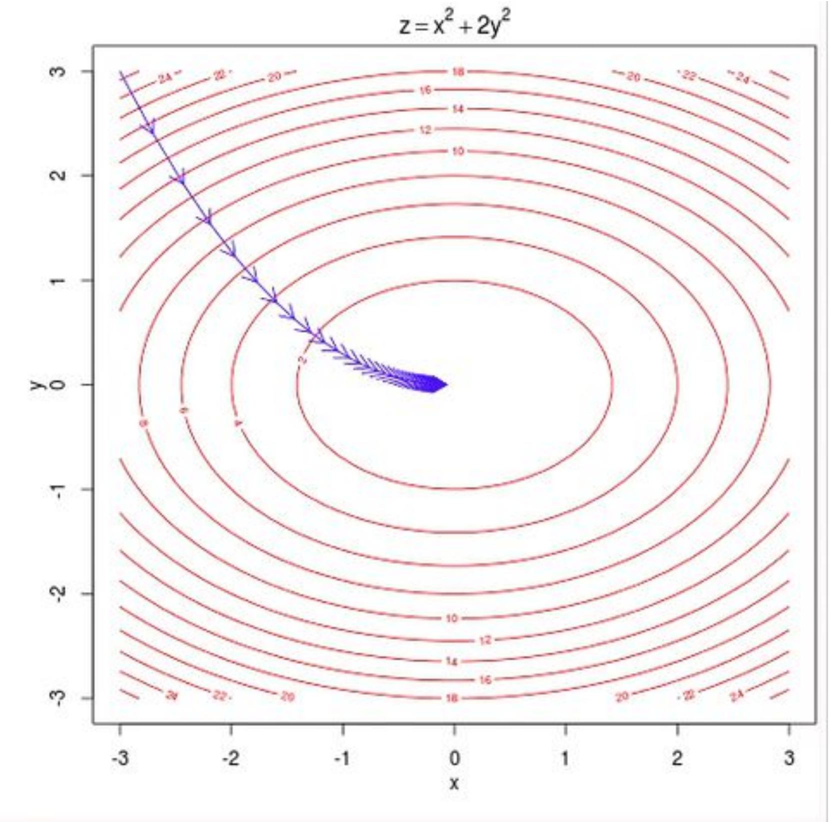
Stochastic gradient descent (SGD)

SGD: estimate the gradient using a subset of the examples

- Pick a **single random training example**
- **Estimate** a (noisy) loss on this single training example (the *stochastic* gradient)
- Compute gradient wrt. this loss
- Take a step of gradient descent using the estimated loss

Slide: Kevin McGuinness (DCU)

Stochastic gradient descent



Slide: Kevin McGuinness (DCU)

Stochastic Gradient Descent (SGD)

SGD Advantages

- Very fast (only need to compute gradient on single example)
- Memory efficient (does not need the full dataset to compute gradient)
- Online (don’t need full dataset at each step)

SGD Disadvantages

- Gradient is very noisy, may not always point in correct direction
- Convergence can be slower

In practice: Mini-batch SGD

- Estimate gradient on small batch of training examples (say 50)
- Known as **mini-batch stochastic gradient descent**

Slide: Kevin McGuinness (DCU)

Vanilla mini-batch SGD

$$\theta_t = \theta_{t-1} - \alpha \underbrace{\nabla_\theta \mathcal{L}(\theta_{t-1})}_{\text{Evaluated on a mini-batch}}$$

Momentum

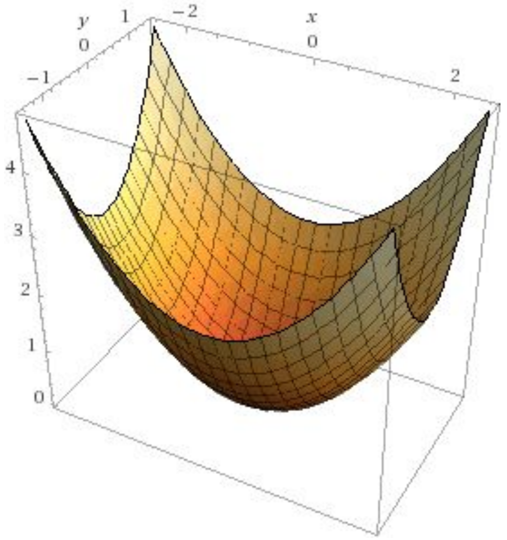
Velocity

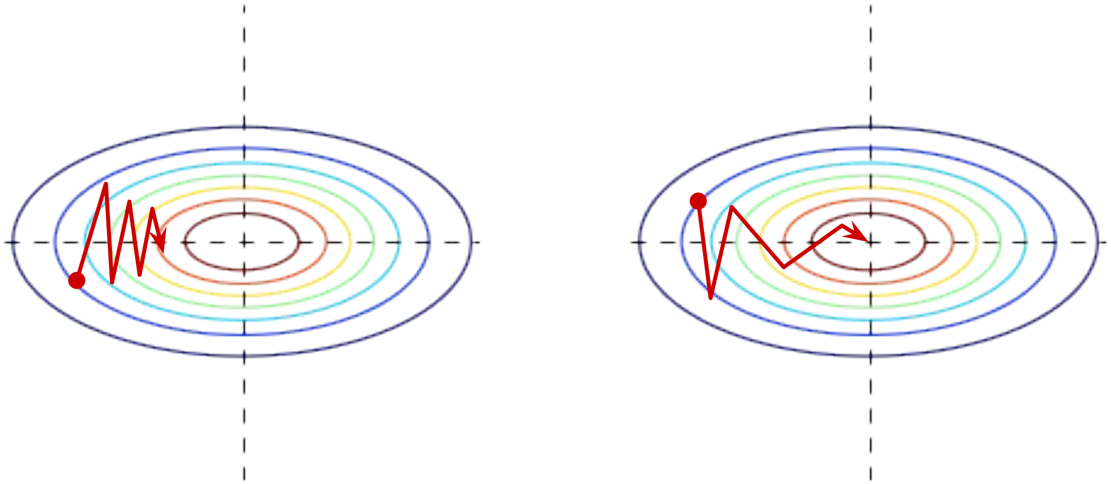
$$v_t = \gamma v_{t-1} + \alpha \nabla_\theta \mathcal{L}(\theta_{t-1})$$

$$\theta_t = \theta_{t-1} - v_t$$

2x memory for parameters!

Momentum





$$v_t = \gamma v_{t-1} + \alpha \nabla_\theta \mathcal{L}(\theta_{t-1})$$

$$\theta_t = \theta_{t-1} - v_t$$

2x memory for parameters!

Adagrad

Adapts the learning rate for each of the parameters based on sizes of previous updates.

- Scales updates to be larger for parameters that are updated less
- Scales updates to be smaller for parameters that are updated more

Store **sum of squares** of gradients so far in diagonal of matrix G_t

$$G_t = \sum_{i=0}^{t} \text{diag}(\nabla \mathcal{L}(\theta)_i)^2$$

Gradient of loss at timestep *i*

Update rule: $\theta_t = \theta_{t-1} - \alpha G_t^{-\frac{1}{2}} \nabla \mathcal{L}(\theta_{t-1})$

Duchi et al. **Adaptive Subgradient Methods for Online Learning and Stochastic Optimization**. JMRL 2011

RMSProp

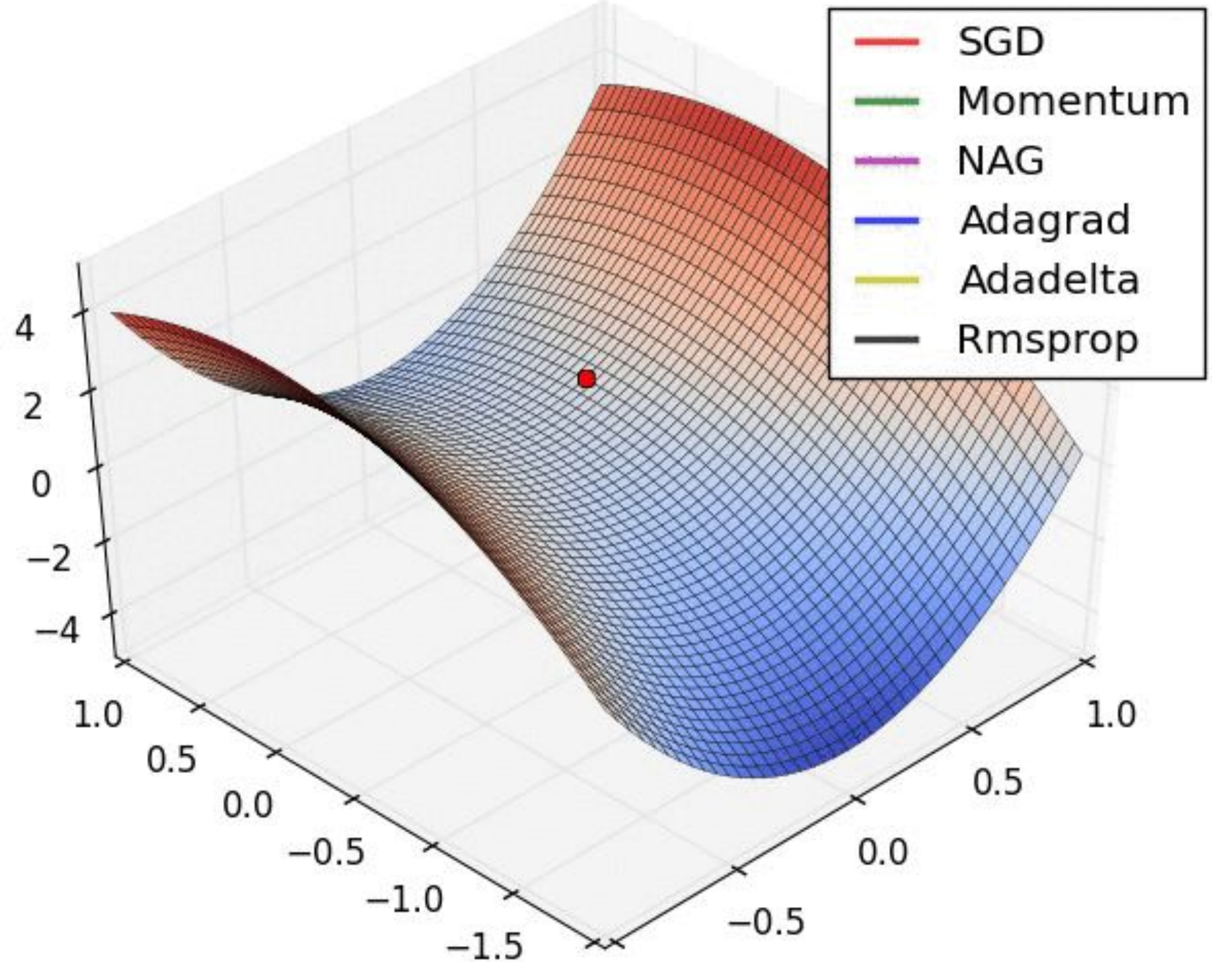
Modification of Adagrad to address aggressively decaying learning rate.

Instead of storing sum of squares of gradient over all time steps so far, use a **decayed moving average** of sum of squares of gradients

$$G_t = \gamma G_{t-1} + (1-\gamma)\mathrm{diag}(\nabla \mathcal{L}(\theta))^2$$

Update rule: $$\theta_t = \theta_{t-1} - \alpha G_t^{-\frac{1}{2}} \nabla \mathcal{L}(\theta_{t-1})$$

Geoff Hinton, Unpublished



Images credit: Alec Radford.

Adam

Combines momentum and RMSProp

Keep decaying average of both first-order moment of gradient (momentum) and second-order moment (like RMSProp)

First-order: $v_t = \gamma_1 v_{t-1} + (1 - \gamma_1) \nabla \mathcal{L}(\theta_{t-1})$

Second-order: $G_t = \gamma_2 G_{t-1} + (1 - \gamma_2) \mathrm{diag}(\nabla \mathcal{L}(\theta))^2$

Update rule: $\theta_t = \theta_{t-1} - \alpha G_t^{-\frac{1}{2}} v_t$

3x memory!

Kingma et al. **Adam: a Method for Stochastic Optimization**. ICLR 2015

Summary

We need an algorithm to find **good weight configurations**.

This is an unconstrained continuous **optimization problem**.

We can use standard iterative optimization methods like **gradient descent**.

To use gradient descent, we need a way to find the **gradient of the loss with respect to the parameters** (weights and biases) of the network.

Error **backpropagation** is an efficient algorithm for finding these gradients.

Basically an application of the multivariate **chain rule** and **dynamic programming**.

In practice, computing the full gradient is expensive. Backpropagation is typically used with **stochastic gradient descent**.

Slide: Kevin McGuinness (DCU)

Outline

1. RL with Neural Networks
2. Loss functions
3. Backpropagation
4. Optimizers

Undergradese

What undergrads ask vs. what they're REALLY asking