Lecture 8:
What about neural networks?

Optimization for data sciences

Rémy Sun
remy.sun@inria.fr

Course organization

Course organization

UNIVERSITÉ CÔTE D'AZUR

- Introduction to optimization
  - A few problems of interest
  - Quick mathematical refresher

- Convex problems (Following Stephen Boyd)
  - Convex sets
  - Convex functions
  - Convex problems
  - Simplex algorithm for Linear Programming

Course organization

- Duality (for convex problems)
  - Lagrangian and dual function
  - Dual problem
  - Qualification constraints
  - KKT conditions

- Newton’s Descent and Barrier methods for convex case
  - Descent for the unconstrained problems
  - Equality constrained problems
  - Interior point methods
  - Lab session!

Course organization

UNIVERSITÉ CÔTE D'AZUR

- **What about the real (neural) world?**
  - **Problem statement**
  - **Let's try to solve it!**
  - **Gradient descent with(out) convexity**
  - **Gradient descent variants**

- Backpropagation

Evaluation

- Reports on lab sessions
  - Labs on jupyter notebooks
    - Not every session
  - Explain the code done in the session
  - Summarize what is done in the practical

- Written Exam
  - Theoretical questions
  - We will do exercises in class

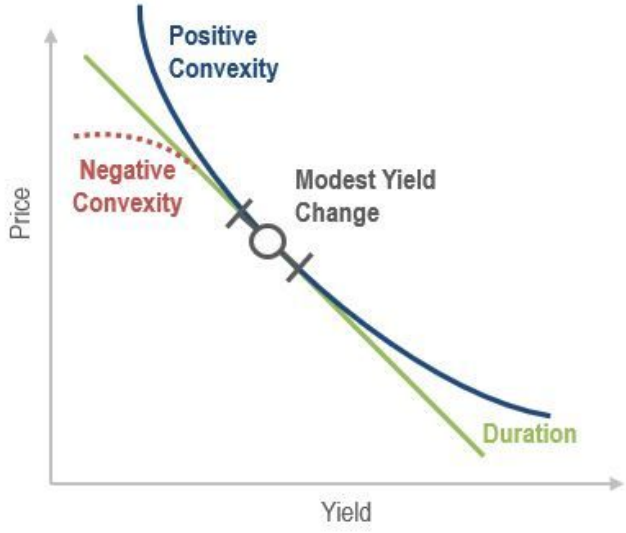
Refresher on convexity!

Easy problem

- classical view:
  - linear (zero curvature) is easy
  - nonlinear (nonzero curvature) is hard

- the classical view is **wrong**

- the correct view:
  - convex (nonnegative curvature) is easy
  - nonconvex (negative curvature) is hard



Convex set

$$x_1, x_2 \in C, \quad 0 \leq \theta \leq 1 \quad \Longrightarrow \quad \theta x_1 + (1-\theta)x_2 \in C$$

- Classic convex sets
  - Affine sets, hyperplanes, cones, balls, polyhedrons

- Convexity preserving operations
  - Intersection
  - Affine mapping
  - Perspective
  - Linear Fractional mapping

Convex functions

$$f(\theta x + (1-\theta)y) \leq \theta f(x) + (1-\theta)f(y)$$

- Classic convex functions
  - Affine, exponential, norms, max, …
- Convexity preserving operations
  - Non negative weighted sum, composition with affine
  - Pointwise maximum and supremum
  - Composition
  - Minimization
  - Perspective

Convex problems

$$\begin{array}{ll} \text{minimize} & f_0(x) \\ \text{subject to} & f_i(x) \le 0, \quad i = 1, \dots, m \\ & h_i(x) = 0, \quad i = 1, \dots, p \end{array}$$

- Convex f and linear h
  - X feasible: satisfies implicit and explicit constraints

- Quite a few classical convex problems(linear, quadratic, …)

- Easy to change variables between equivalent problems

Duality

UNIVERSITÉ CÔTE D'AZUR

$$g(\lambda, \nu) = \inf_{x \in \mathcal{D}} L(x, \lambda, \nu) = \inf_{x \in \mathcal{D}} \left(f_0(x) + \sum_{i=1}^{m} \lambda_i f_i(x) + \sum_{i=1}^{p} \nu_i h_i(x) \right)$$

- Mirror problem that is always convex!
- Gives a lower bound on solution (weak duality)
- Can give the exact solution
  - Under qualifications on constraints for convex problems
- KKT conditions can help reverse engineer a solution

Descent methods

- **descent methods** generate iterates as

$$x^{(k+1)} = x^{(k)} + t^{(k)} \Delta x^{(k)}$$

with $f(x^{(k+1)}) < f(x^{(k)})$ (hence the name)

- other notations: $x^+ = x + t\Delta x$, $x := x + t\Delta x$
- $\Delta x^{(k)}$ is the **step**, or **search direction**
- $t^{(k)} > 0$ is the **step size**, or **step length**
- from convexity, $f(x^+) < f(x)$ implies $\nabla f(x)^T \Delta x < 0$
- this means Δx is a **descent direction**

Descent methods

General descent method.

given a starting point $x \in \mathbf{dom}\, f$.
repeat

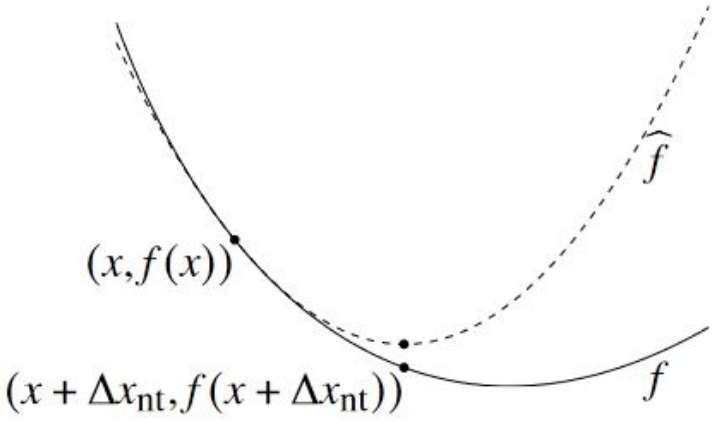
1. Determine a descent direction Δx.
2. **Line search.** Choose a step size $t > 0$.
3. **Update.** $x := x + t\Delta x$.

until stopping criterion is satisfied.

The gold standard: Newton's method

- **Newton step** is $\Delta x_{\text{nt}} = -\nabla^2 f(x)^{-1} \nabla f(x)$
- **interpretation**: $x + \Delta x_{\text{nt}}$ minimizes second order approximation

$$\widehat{f}(x+v) = f(x) + \nabla f(x)^T v + \frac{1}{2} v^T \nabla^2 f(x) v$$



Central path

- for $t > 0$, define $x^\star(t)$ as the solution of

$$\begin{array}{ll} \text{minimize} & tf_0(x) + \phi(x) \\ \text{subject to} & Ax = b \end{array}$$

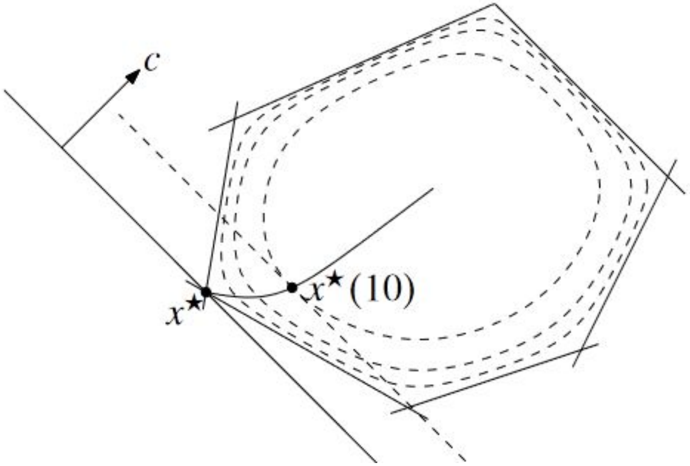
  (for now, assume $x^\star(t)$ exists and is unique for each $t > 0$)

- central path is $\{x^\star(t) \mid t > 0\}$

example: central path for an LP

$$\begin{array}{ll} \text{minimize} & c^T x \\ \text{subject to} & a_i^T x \leq b_i, \quad i = 1, \dots, 6 \end{array}$$

hyperplane $c^T x = c^T x^\star(t)$ is tangent to level curve of ϕ through $x^\star(t)$



Solvers exist!

math:

$$\begin{array}{ll} \text{minimize} & \|Ax - b\|_2^2 \\ \text{subject to} & x \geq 0 \end{array}$$

- variable is x
- A, b given
- $x \geq 0$ means $x_1 \geq 0, \ldots, x_n \geq 0$

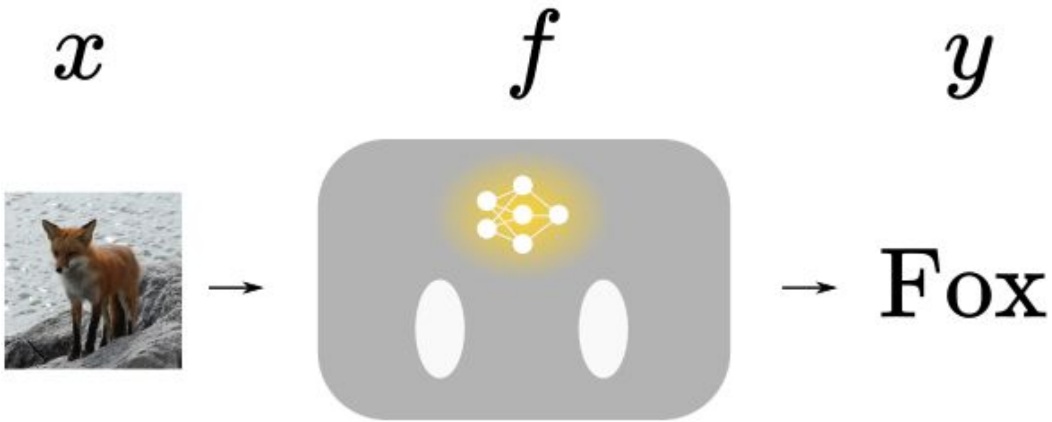
CVXPY code:

```
import cvxpy as cp

A, b = ...

x = cp.Variable(n)
obj = cp.norm2(A @ x - b)**2
constr = [x >= 0]
prob = cp.Problem(cp.Minimize(obj), constr)
prob.solve()
```

- ***You*** need to
  - Identify possibly convex problems
  - Remember ready made solvers exist
  - Use the ready made solvers
    - (and not a big neural network)

1. Statistical optimization

Problem statement: Ideal case



- Find (robot) f that classifies images well
  - Often based on neural networks

$$\forall (x, y) \in \mathcal{D}, f(x) = y$$

More formally

- Definitions
  - X set of inputs
  - Y set of labels
  - $\Omega = X \times Y$
  - $\mathcal{D}$ Distribution over Ω with probability measure p

- Find function f: X -> Y such that

$$\forall (x, y) \in \mathcal{D}, f(x) = y$$

Assessing f with a criterion

- Finding exact correspondence functions is not always the thing to do
  - No exact matching
  - Other definitions of good solutions
  - Need to use restricted function space
    - Parametric function space

$$\mathcal{F} = \{f_\theta | \theta \in \mathbb{R}^d\}$$

- Introduce an assessment of how “good” f is with a loss l so that we try to have the lowest quantity $l(f(x), y)$

Minimizing Risk

- Definitions
  - X set of inputs
  - Y set of labels
  - $\Omega = X \times Y$
  - $\mathcal{D}$ Distribution over Ω with probability measure p
  - l loss function assessing fit of f(x) to y
  - Find f in function space $\mathcal{F} = \{f_\theta | \theta \in \mathbb{R}^d\}$

- Minimize the ***Risk*** over the distribution

$$min_\theta \mathbb{E}_{x,y \sim \mathcal{D}}[l(f_\theta(x), y)]$$

Minimizing Risk

UNIVERSITÉ CÔTE D'AZUR

- Problem: we do not know $\mathcal{D}$!
  - Solved problem otherwise…
  - Evaluating the risk requires this distribution

- Solution: Use a dataset D of (x,y) sampled from $\mathcal{D}$
  - ***<u>Empirical Risk Minimization</u>***
  - If the (x,y) are i.i.d drawn from $\mathcal{D}$ can be expressed as a mean over the dataset

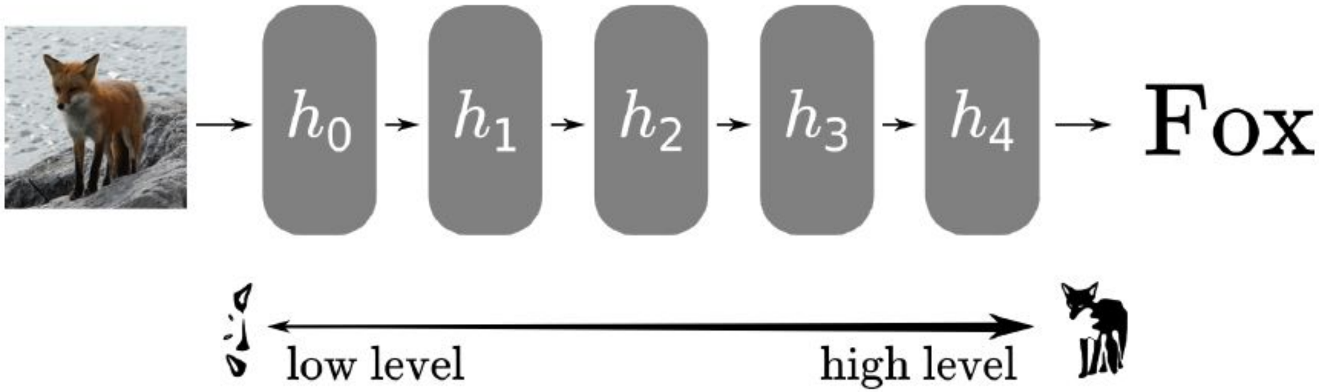
$$min_\theta \hat{\mathcal{R}}_\theta = \frac{1}{N} \sum_{i=0,\ldots,N-1} l(f_\theta(x_i), y_i)$$

Takeaway

UNIVERSITÉ CÔTE D'AZUR

- Core problem: Find function matching inputs to outputs for any (x,y) of the target distribution
- Optimize over family of parametric functions
  - Assess functions with loss criterion
- Minimize the Risk function
  - Empirical Risk Minimization in practice

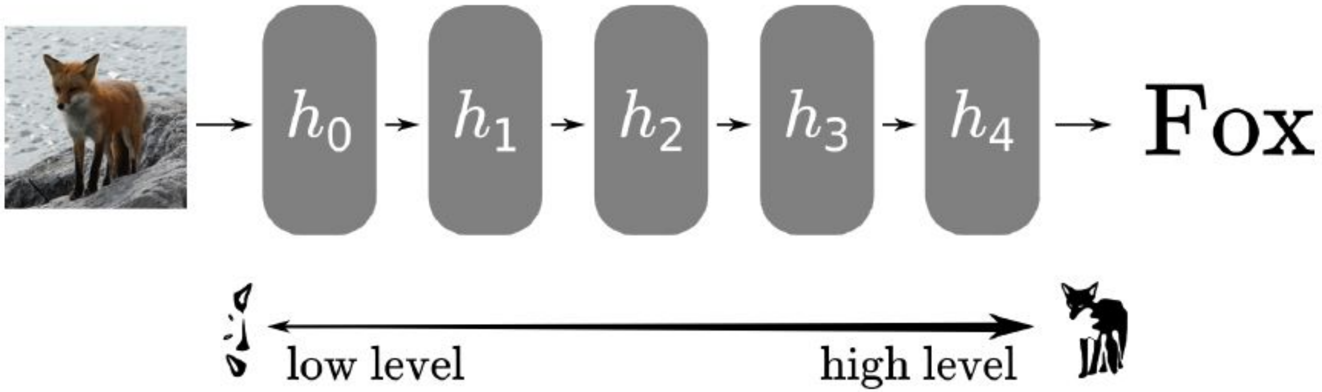
Neural network functions

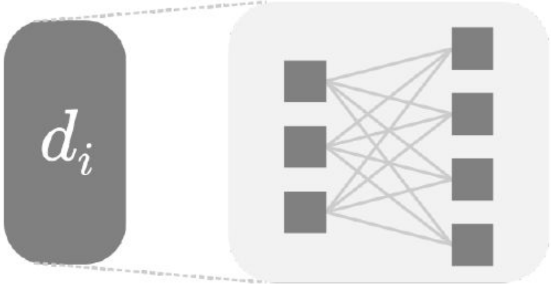


- Neural networks are sequences of simple functions

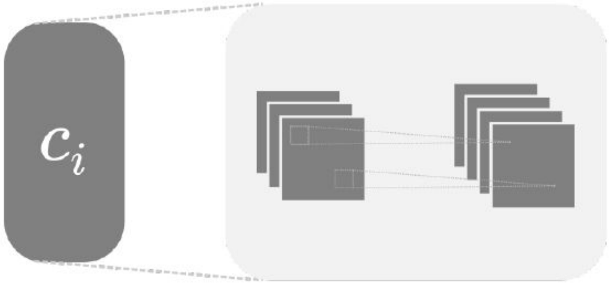
$$f_\theta = h_\theta^0 \circ h_\theta^1 \circ \cdots \circ h_\theta^{L-1}$$

Neural network functions



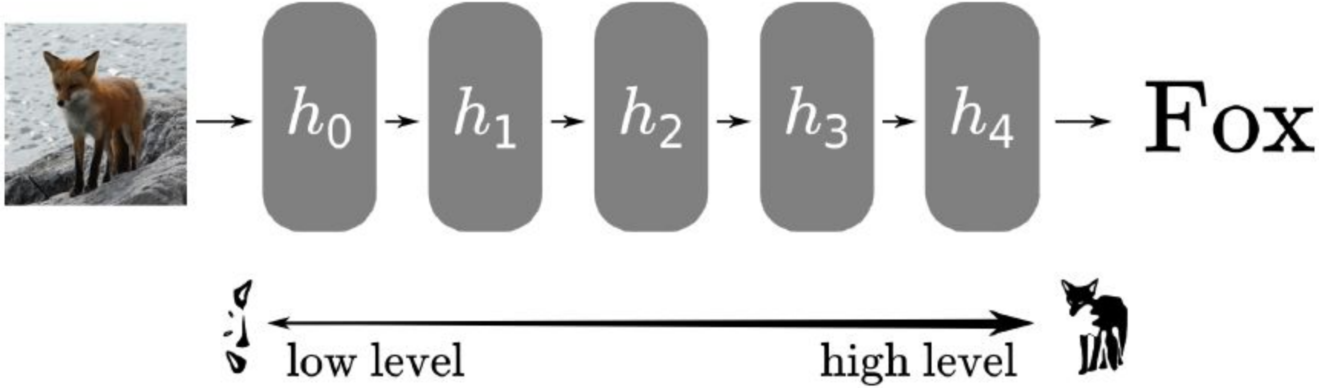


(a) Dense layer



(b) Convolutional layer

Interlude: Formalization of dense networks

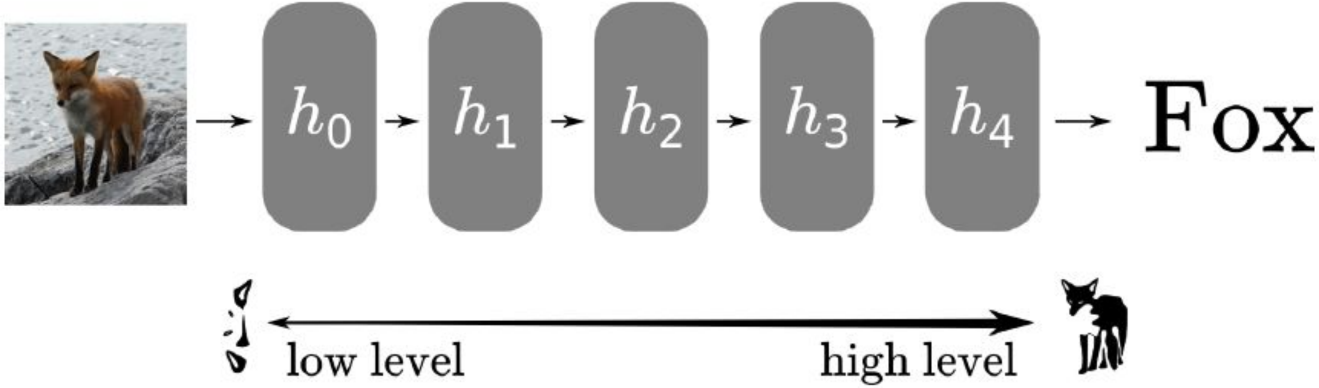


(a) Dense layer

$$d_\theta(x) = \sigma(W_\theta x^T + b_\theta)$$

$$\sigma(x) = ReLU(x) = max(0, x)$$

Interlude: Formalization of dense networks



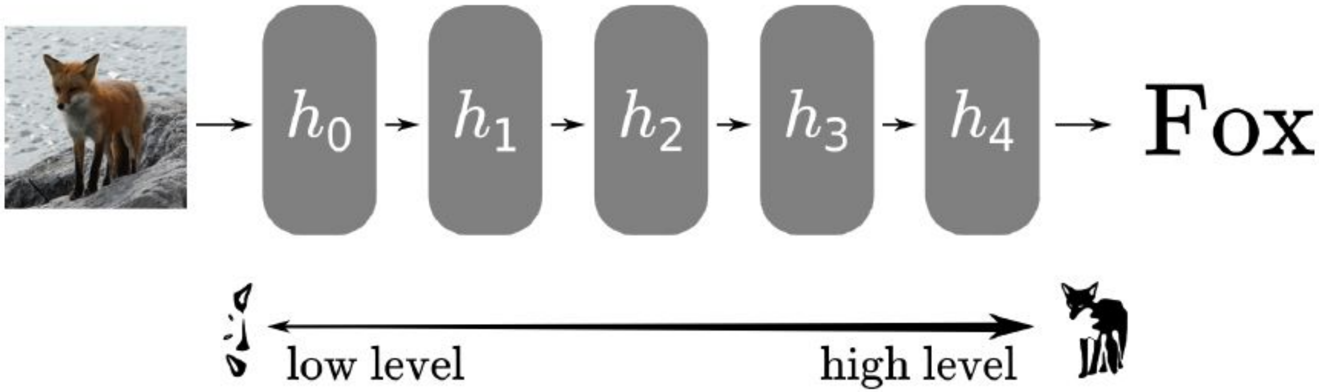
$$d_\theta(x) = \sigma(W_\theta x^T + b_\theta)$$

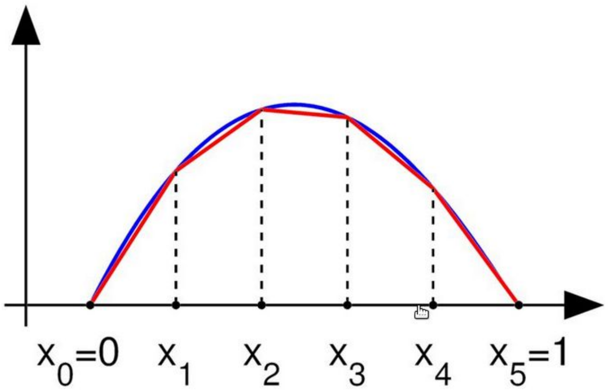
$$\sigma(x) = ReLU(x) = max(0, x)$$

<u>Piecewise linear!</u>

Individual layers are piecewise linear, composition preserves piecewise linearity

Interlude: Formalization of dense networks





- Highly expressive
  - Can fit many types of distributions

Takeaway

- Neural networks composed of simple functions
  - Typical linear layer operations
  - Non-linear activation functions

- High expressive power
  - Universal approximation with enough neurons
  - ReLU Feedforward networks are piecewise linear

2. So what do we do?

Let's analyze the problem

$$
\begin{array}{ll}
\text{minimize} & f_0(x) \\
\text{subject to} & f_i(x) \leq 0, \quad i = 1, \ldots, m \\
& h_i(x) = 0, \quad i = 1, \ldots, p
\end{array}
$$

- What is the cost function?
- What are the constraints functions?

- Is this convex? Why?

Let’s analyze the problem

$$
\begin{array}{ll}
\text{minimize} & f_0(x) \\
\text{subject to} & f_i(x) \leq 0, \quad i = 1, \ldots, m \\
& h_i(x) = 0, \quad i = 1, \ldots, p
\end{array}
$$

- **Cost function given by neural network and loss**
- What are the constraints functions?

- Is this convex? Why?

Let’s analyze the problem

$$\begin{array}{ll} \text{minimize} & f_0(x) \\ \text{subject to} & f_i(x) \leq 0, \quad i = 1, \ldots, m \\ & h_i(x) = 0, \quad i = 1, \ldots, p \end{array}$$

- Cost function given by neural network and loss
- **No constraints!**
    - **Easy unconstrained problem!**

- Is this convex? Why?

(Non-convex) Unconstrained problem

$$\begin{array}{ll} \text{minimize} & f_0(x) \\ \text{subject to} & f_i(x) \le 0, \quad i = 1, \ldots, m \\ & h_i(x) = 0, \quad i = 1, \ldots, p \end{array}$$

- Cost function is given by neural network and loss
- No constraints!
  - Easy unconstrained problem!

- **Not convex, for a number of reasons.**

Let’s do Newton descent!

- What do we need?
  - Step size
    - Evaluating is expensive… Backtrack? Fixed?
  - Gradient
    - Could get pretty expensive too…
  - Hessian
    - This is getting very very expensive.
    - And we need to invert it too?!

Let’s do discount descent: Gradient descent

- What do we need?
    - Step size
        - Fixed step size
    - Gradient
        - Could get pretty expensive too…
        - Obtained through backpropagation!
    - No Hessian!

Does that work?!

- Gradient descent
  - OK solver for convex problems
  - Guaranteed to converge to the global minimum on convex problems

- What changes on neural networks?
  - Still guaranteed to converge under conditions
  - But not much more…

Why did we spend 7 weeks on convexity?!

- Easy solved problems
    - Need to be able to recognize them
    - Available fast solvers

- Convexity still gives us some intuition
    - We are using a convex optimization method
    - Optimizer “proofs” tend to be on convex case
        - For intuition! Because non-convex is hard…

3. Gradient descent with(out) convexity

Gradient descent

- general descent method with $\Delta x = -\nabla f(x)$

---

given a starting point $x \in \textbf{dom} f$.
repeat
1. $\Delta x := -\nabla f(x)$.
2. **Line search.** Choose step size t via exact or backtracking line search.
3. **Update.** $x := x + t\Delta x$.

until stopping criterion is satisfied.

- stopping criterion usually of the form $\|\nabla f(x)\|_2 \leq \epsilon$
- convergence result: for strongly convex f,

$$f(x^{(k)}) - p^\star \leq c^k (f(x^{(0)}) - p^\star)$$

  $c \in (0, 1)$ depends on m, $x^{(0)}$, line search type
- very simple, but can be very slow

What can we say?

UNIVERSITÉ CÔTE D'AZUR

- Simplest setting
  - Fixed step size
  - Descent direction if opposite of gradient

- What assumptions can we make on f?
  - Convexity is out of the question!
  - There **is** a minimum?
    - Does not guarantee convergence!

A more reasonable assumption

- We cannot assume convexity
- We can assume a minimum
  - Otherwise there is nothing to find
  - But it is not sufficient

A more reasonable assumption

UNIVERSITÉ CÔTE D'AZUR

- We cannot assume convexity

- We can assume a minimum
  - Otherwise there is nothing to find
  - But it is not sufficient

- **Maybe neural networks change slowly?**
  - **Smoothness**

A more reasonable assumption

UNIVERSITÉ CÔTE D'AZUR

- We cannot assume convexity
- We can assume a minimum
  - Otherwise there is nothing to find
  - But it is not sufficient
- **Maybe neural networks change slowly?**
  - **Smoothness**

L-Lipschitz continuity

$$|f(u) - f(w)| \leq L \, \|u - w\| \, .$$

- Function is L-Lipschitz (continuous)
  - For a given norm
  - Inequality condition
- L quantifies how fast f moves
  - Small L means regular function

L-Lipschitz smoothness

$$||\nabla f(x) - \nabla f(y)|| \leq L||x - y||.$$

- Gradient is L-Lipschitz
- Fairly reasonable assumption
  - Network components usually respect this
  - Can be very large
- Can even be optimized for (Lipschitz networks)

Bound on difference in values of L-smooth

UNIVERSITÉ CÔTE D'AZUR

- L-smooth network f
  - L>0
  - For the given norm

- x, y inputs in the domain of f

- Upper bound on the difference f(x) and f(y)

$$f(y) \leq f(x) + \langle \nabla f(x), y - x \rangle + \frac{L}{2} ||y - x||^2.$$

The descent sequence is decreasing

- L-smooth network f
- X_t elements of a gradient descent sequence
- Descent step $\frac{1}{L}$
- The sequence is decreasing
  - With minimal decrement

$$f(x_{t+1}) \leq f(x_t) - \frac{1}{2L}||\nabla f(x_t)||^2.$$

Conclusions? It converges, no convexity!

- 2 direct consequences
  - The sequence converges
    - Decreasing
    - Bounded below by existing minimum
  - The gradient sequence converges to 0

Conclusions? It converges, no convexity!

- 2 direct consequences
  - The sequence converges
    - Decreasing
    - Bounded below by existing minimum
  - The gradient sequence converges to 0

- **We won? We won!**

Conclusions? It converges, no convexity!

UNIVERSITÉ CÔTE D'AZUR

- 2 direct consequences
  - The sequence converges
    - Decreasing
    - Bounded below by existing minimum
  - The gradient sequence converges to 0

- Not so fast…

Conclusions? It converges, no convexity!

- 2 direct consequences
  - The sequence converges
    - Decreasing
    - Bounded below by existing minimum
  - The gradient sequence converges to 0
- Not so fast...
  - **Convergence is weak (no guaranteed rate)**

Conclusions? It converges, no convexity!

UNIVERSITÉ CÔTE D'AZUR

- 2 direct consequences
  - The sequence converges
    - Decreasing
    - Bounded below by existing minimum
  - The gradient sequence converges to 0

- Not so fast…
  - No convergence rate and L can be very big…
  - **It does not converge to the minimum…**

With convexity

- L-smooth convex network f

- Descent step $\frac{1}{L}$

- Sequence converges to global minimum
- **And** we know the convergence rate!

$$f(x_t) - f(x_*) \leq \frac{2L||x_0 - x_*||^2}{t+4}.$$

Takeaway

- Under reasonable L-smooth assumption
  - Gradient descent converges!
    - To something
    - With a minuscule fixed step size

- General deep learning heuristics
  - Most local minimums are similarly good/bad
  - Big networks have very few really bad mins

4. Gradients and optimizers

Approximation: Gradient Descent

- Iteratively make steps of size η to minimize risk

$$\theta^{t+1} := \theta^t - \eta \nabla_\theta \hat{\mathcal{R}}_\theta$$

- Elementwise form:

$$\theta_i^{t+1} := \theta_i^t - \eta \frac{\partial \hat{\mathcal{R}}_\theta}{\partial \theta_i}$$

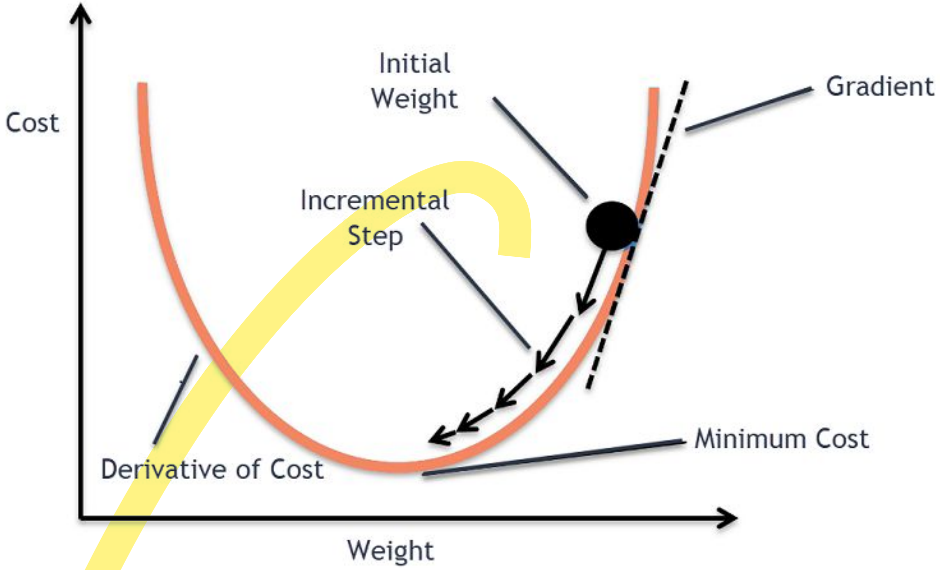


Image from Analytics Vidhya

Interlude: A few nice things to know

- Guaranteed to converge under certain conditions
  - Lipschitz gradient gives nice upper bounds
  - Adaptive gradient steps can offer guarantees
    - Steps traditionally fixed

- No guarantee to find a global optimum
  - Quite unlikely
  - Gravitates towards Local optimum

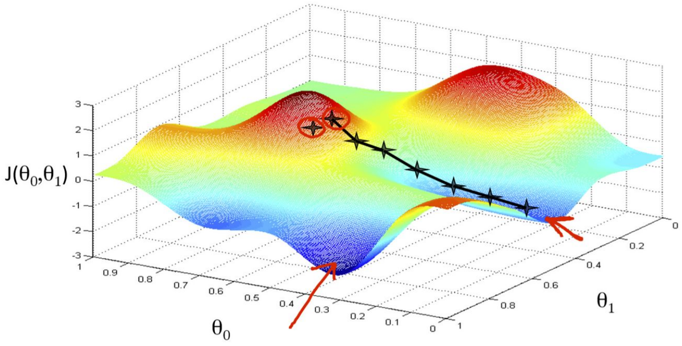


Image from Towards Data Science

In practice: Stochastic Gradient Descent

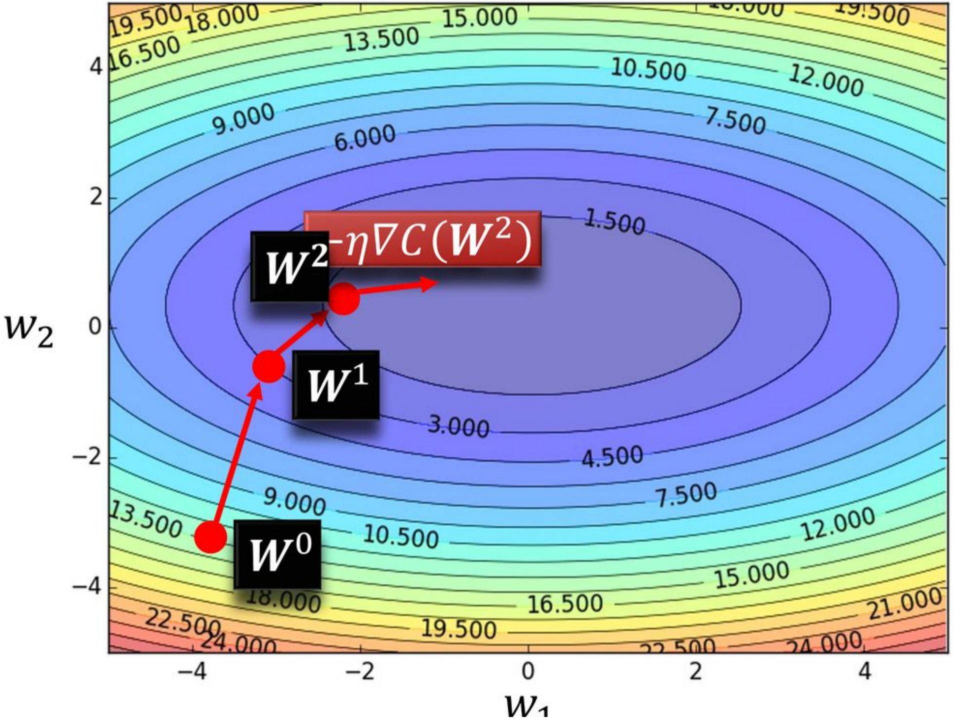
- Deep learning deals in large-scale
  - Big datasets
  - Big models
  - GD can get expensive!

- Stochastic Gradient Descent
  - Work on small batches of data B instead of D

$$\theta^{t+1} := \theta^{t} - \eta \nabla_{\theta} \hat{R_{\theta}}(B)$$

  - Noisier process

Remember SGD?



Randomly pick a starting point $\boldsymbol{W}^0$

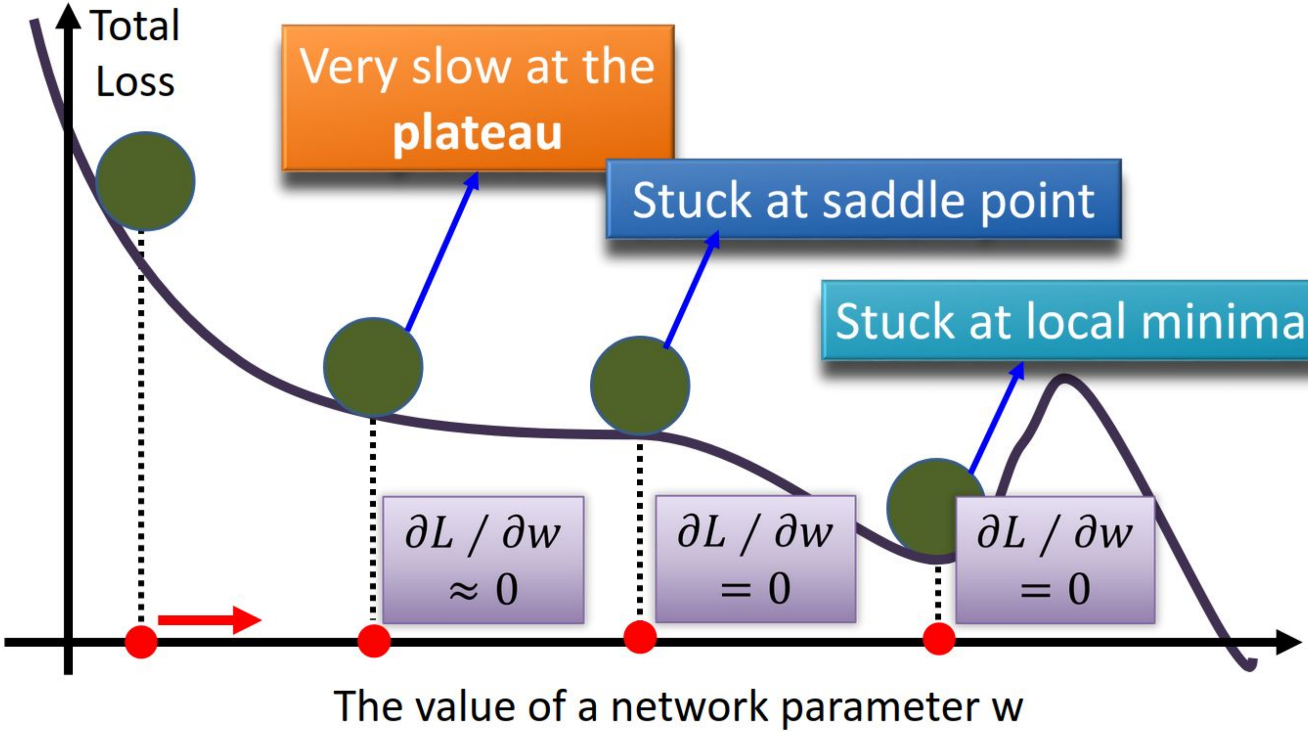
Compute the negative gradient at $\boldsymbol{W}^0$

$\Rightarrow -\nabla C(\boldsymbol{W}^0)$

Times the learning rate η

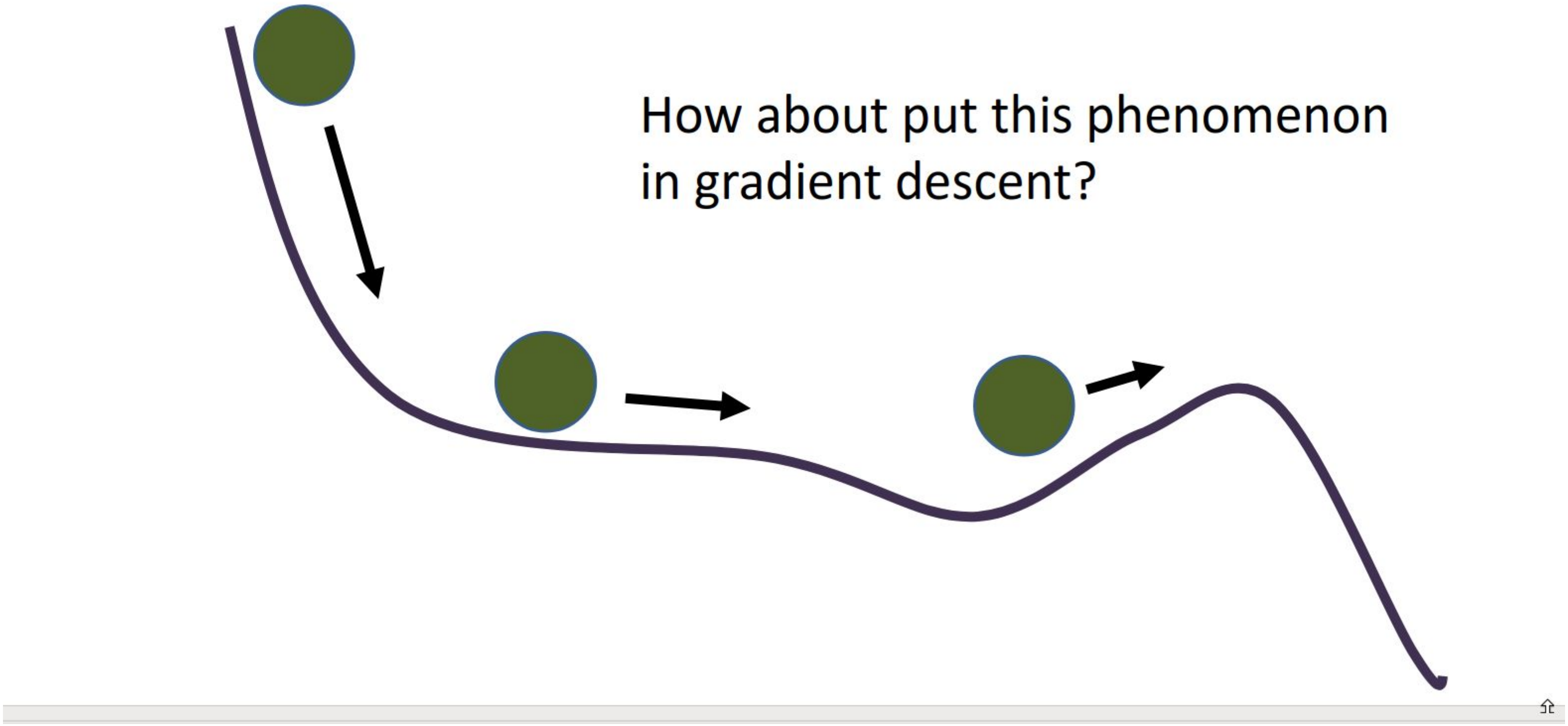
$\Rightarrow -\eta\nabla C(\boldsymbol{W}^0)$

What is happening in a NN

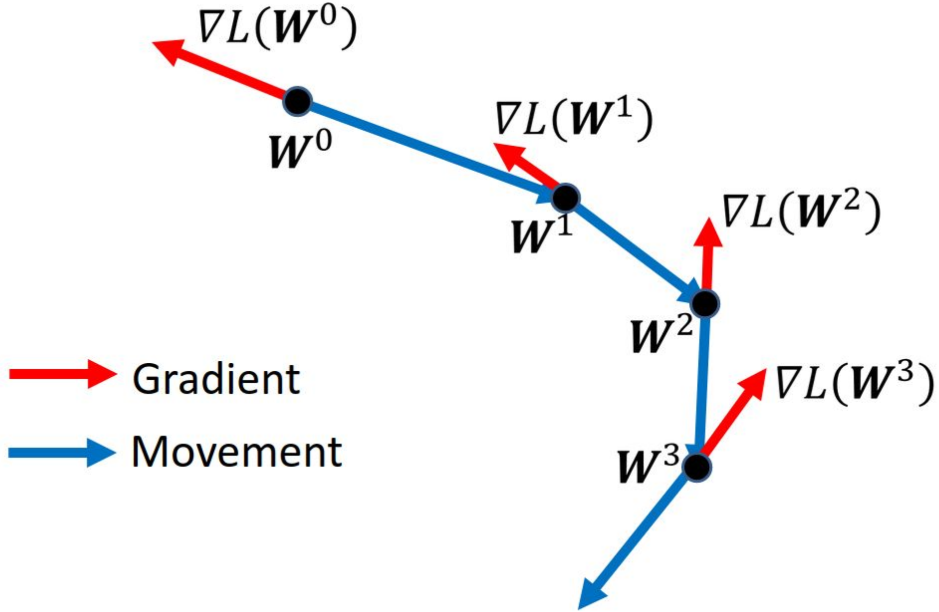


Momentum

Cannot we build “momentum” as go?



Vanilla SGD



Start at position $\boldsymbol{W}^0$

Compute gradient at $\boldsymbol{W}^0$

Move to $\boldsymbol{W}^1 = \boldsymbol{W}^0 - \eta \nabla L(\boldsymbol{W}^0)$

Compute gradient at $\boldsymbol{W}^1$

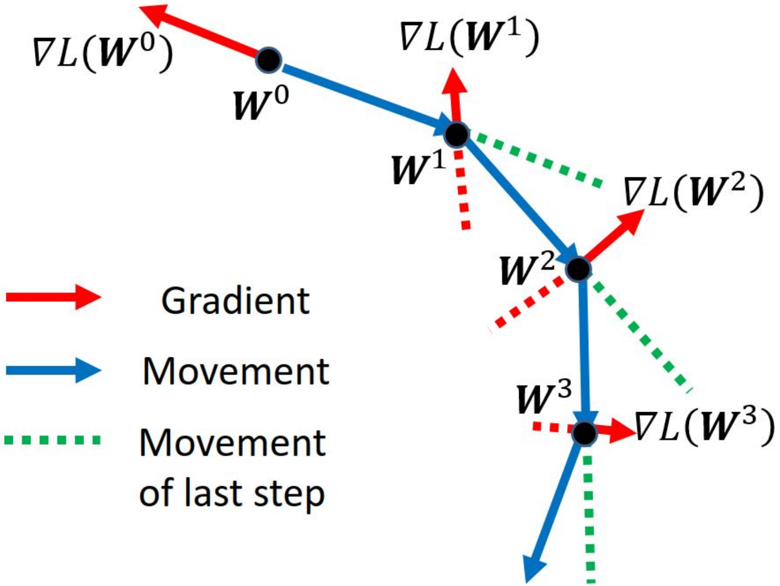
Move to $\boldsymbol{W}^2 = \boldsymbol{W}^1 - \eta \nabla L(\boldsymbol{W}^1)$

⋮

Stop until $\nabla L(\boldsymbol{W}^t) \approx 0$

Momentum SGD

Movement: movement of last step minus gradient at present



Start at point $\boldsymbol{W}^0$

Movement $v^0=0$

Compute gradient at $\boldsymbol{W}^0$

Movement $v^1 = \lambda v^0 - \eta \nabla L(\boldsymbol{W}^0)$

Move to $\boldsymbol{W}^1 = \boldsymbol{W}^0 + v^1$

Compute gradient at $\boldsymbol{W}^1$

Movement $v^2 = \lambda v^1 - \eta \nabla L(\boldsymbol{W}^1)$

Move to $\boldsymbol{W}^2 = \boldsymbol{W}^1 + v^2$

Movement not just based on gradient, but previous movement.

Momentum SGD

Movement: movement of last step minus gradient at present

v^i is actually the weighted sum of all the previous gradient: $\nabla L(\boldsymbol{W}^0), \nabla L(\boldsymbol{W}^1), \ldots \nabla L(\boldsymbol{W}^{i-1})$

$v^0 = 0$

$v^1 = -\eta \nabla L(\boldsymbol{W}^0)$

$v^2 = -\lambda \eta \nabla L(\boldsymbol{W}^0) - \eta \nabla L(\boldsymbol{W}^1)$

$\vdots$

Start at point $\boldsymbol{W}^0$

Movement $v^0=0$

Compute gradient at $\boldsymbol{W}^0$

Movement $v^1 = \lambda v^0 - \eta \nabla L(\boldsymbol{W}^0)$

Move to $\boldsymbol{W}^1 = \boldsymbol{W}^0 + v^1$

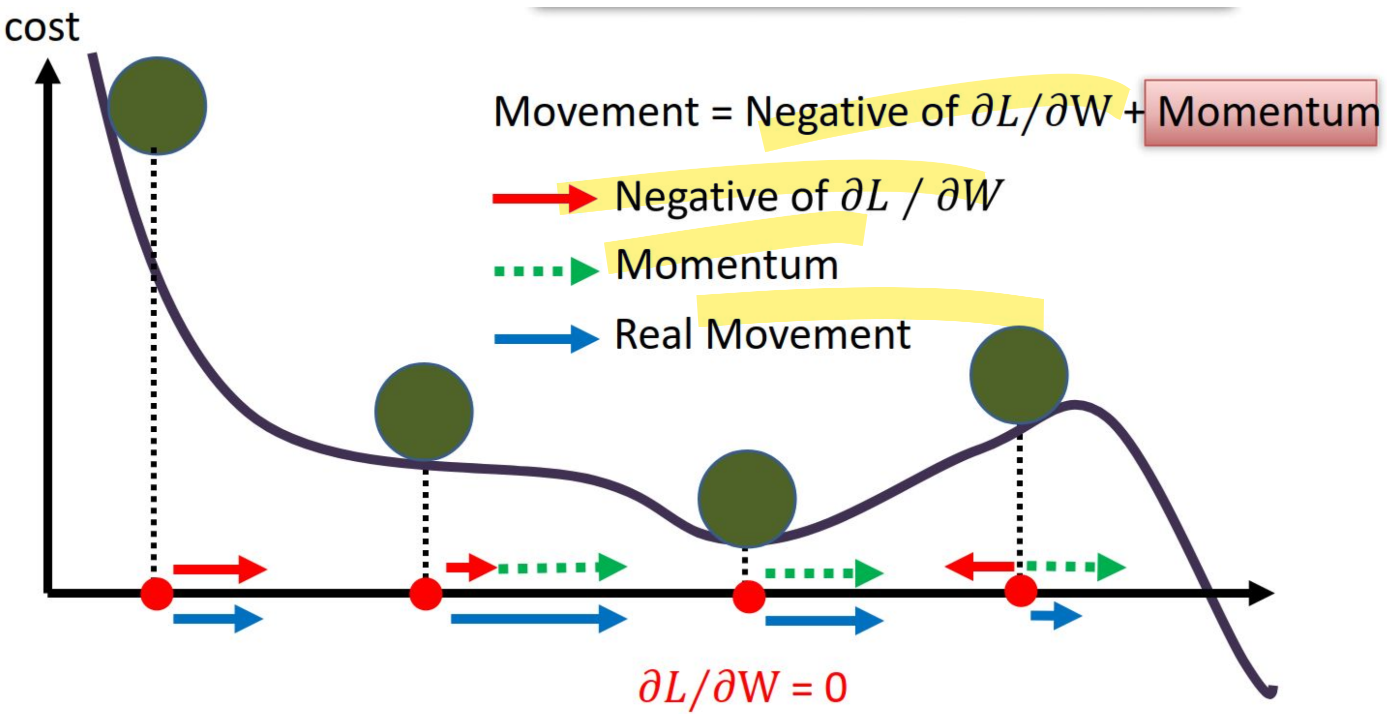
Compute gradient at $\boldsymbol{W}^1$

Movement $v^2 = \lambda v^1 - \eta \nabla L(\boldsymbol{W}^1)$

Move to $\boldsymbol{W}^2 = \boldsymbol{W}^1 + v^2$

Movement not just based on gradient, but previous movement

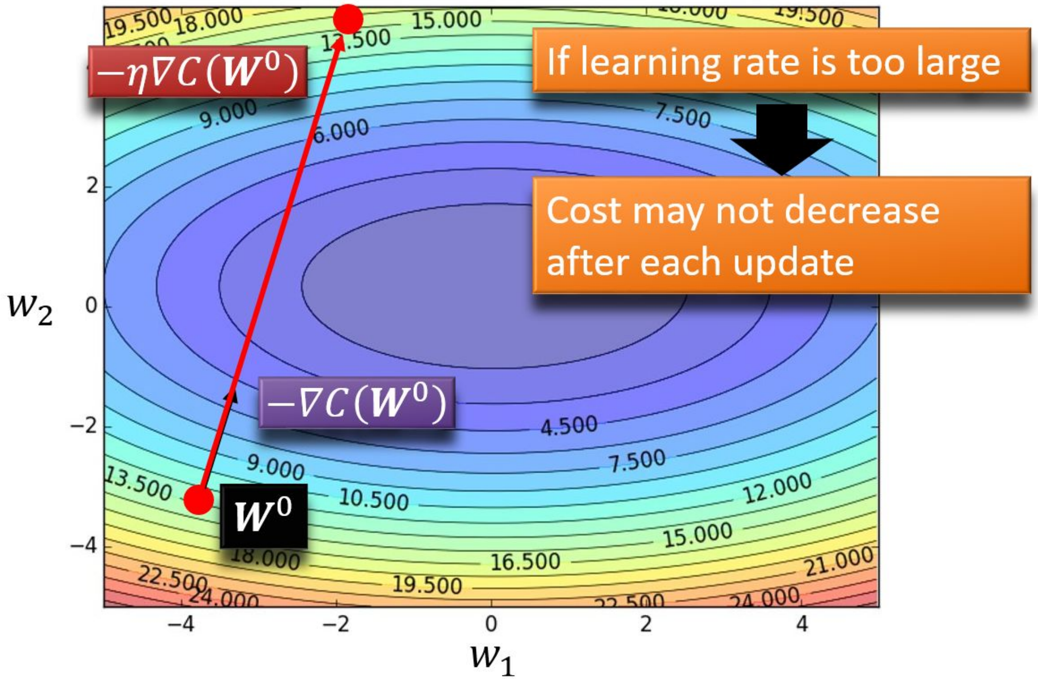
Momentum SGD



Adapting the learning rate

- ***Adaptive Learning Rate***

Set the learning rate η carefully



Adapting the learning rate

- Popular & Simple Idea: Reduce the learning rate by some factor every few epochs.
  - At the beginning, we are far from the destination, so we use larger learning rate
  - After several epochs, we are close to the destination, so we reduce the learning rate
  - E.g. 1/t decay: $\eta^t = \eta / \sqrt{t+1}$
- Learning rate cannot be one-size-fits-all
  - Giving different parameters different learning rates

Adapting the learning rate

UNIVERSITÉ CÔTE D'AZUR

Original Gradient Descent

$$\boldsymbol{W}^t \leftarrow \boldsymbol{W}^{t-1} - \eta \nabla C(\boldsymbol{W}^{t-1})$$

Each parameter w are considered separately

$$\boldsymbol{W}^{t+1} \leftarrow \boldsymbol{W}^t - \boxed{\eta_{\boldsymbol{W}}}\, \underline{g^t} \qquad \underline{g^t} = \frac{\partial C(\boldsymbol{W}^t)}{\partial \boldsymbol{W}}$$

Parameter dependent learning rate

$$\eta_{\boldsymbol{W}} = \frac{\eta}{\sqrt{\boxed{\sum_{i=0}^{t} (g^i)^2}}}$$

constant

g^i is $\partial L \,/\, \partial w$ obtained at the i-th update

Summation of the square of the previous derivatives

Adapting the learning rate

$$\eta_w = \boxed{\frac{\eta}{\sqrt{\sum_{i=0}^{t} (g^i)^2}}}$$

- ***Adagrad***

w_1	g^0
	0.1

Learning rate:

$$\frac{\eta}{\sqrt{0.1^2}} = \frac{\eta}{0.1}$$

$$\frac{\eta}{\sqrt{0.1^2 + 0.2^2}} = \frac{\eta}{0.22}$$

w_2	g^0
	20.0

Learning rate:

$$\frac{\eta}{\sqrt{20^2}} = \frac{\eta}{20}$$

$$\frac{\eta}{\sqrt{20^2 + 10^2}} = \frac{\eta}{22}$$

Observation: 1. Learning rate is smaller and smaller for all parameters

2. Smaller derivatives, larger learning rate, and vice versa

RMSProp

UNIVERSITÉ CÔTE D'AZUR

$$w^1 \leftarrow w^0 - \frac{\eta}{\sigma^0} g^0 \qquad \sigma^0 = g^0$$

$$w^2 \leftarrow w^1 - \frac{\eta}{\sigma^1} g^1 \qquad \sigma^1 = \sqrt{\alpha(\sigma^0)^2 + (1-\alpha)(g^1)^2}$$

$$w^3 \leftarrow w^2 - \frac{\eta}{\sigma^2} g^2 \qquad \sigma^2 = \sqrt{\alpha(\sigma^1)^2 + (1-\alpha)(g^2)^2}$$

$$\vdots$$

$$w^{t+1} \leftarrow w^t - \frac{\eta}{\sigma^t} g^t \qquad \sigma^t = \sqrt{\alpha(\sigma^{t-1})^2 + (1-\alpha)(g^t)^2}$$

Root Mean Square of the gradients
with previous gradients being decayed

Adam: Momentum + RMSProp

$$\nu_t = \beta_1 * \nu_{t-1} - (1 - \beta_1) * g_t$$
$$s_t = \beta_2 * s_{t-1} - (1 - \beta_2) * g_t^2$$

$$\Delta\omega_t = -\eta \frac{\nu_t}{\sqrt{s_t + \epsilon}} * g_t$$

$$\omega_{t+1} = \omega_t + \Delta\omega_t$$

η : *Initial Learning rate*

g_t : *Gradient at time t along* ω^j

ν_t : *Exponential Average of gradients along* ω_j

s_t : *Exponential Average of squares of gradients along* ω_j

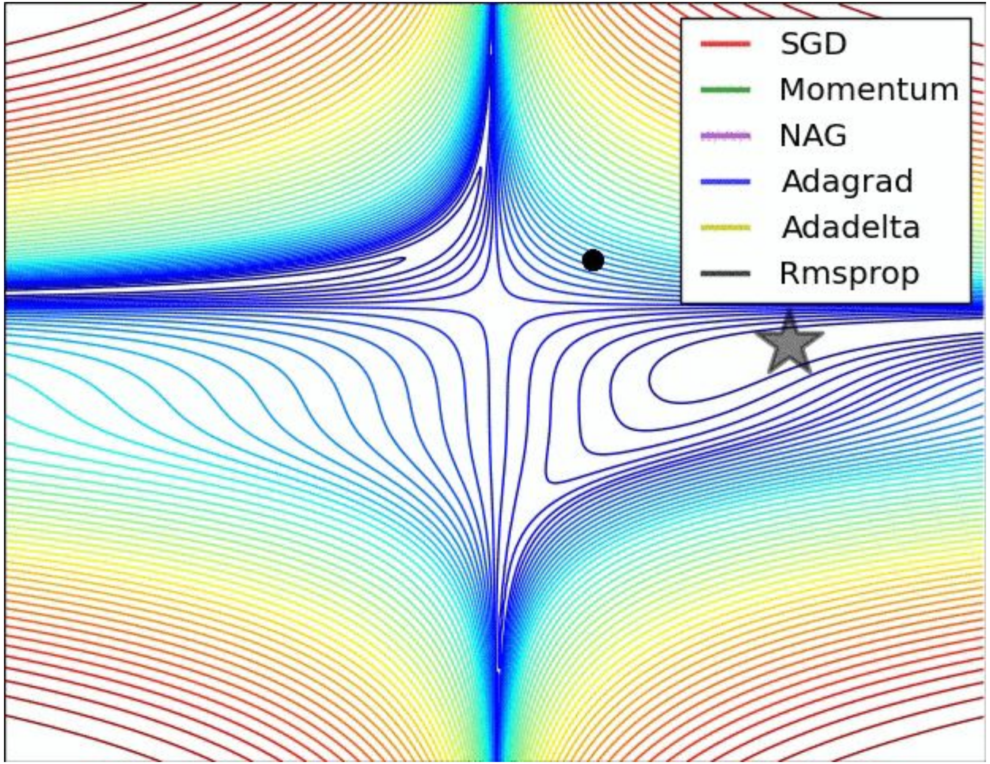
β_1, β_2 : *Hyperparameters*

1,059

Optimizer toolbox

- Adagrad [John Duchi, JMLR'11]
- RMSprop
  - https://www.youtube.com/watch?v=O3sxAc4hxZU
- Adadelta [Matthew D. Zeiler, arXiv'12]
- Adam [Diederik P. Kingma, ICLR'15] = RMSprop + Momentum
- AdaSecant [Caglar Gulcehre, arXiv'14]
- "No more pesky learning rates" [Tom Schaul, arXiv'12]
- Nadam
  - http://cs229.stanford.edu/proj2015/054_report.pdf

Visualization



A word on learning rate and batch size

- Small batch = Good generalization
  - Because of noisiness
    - Kind of but not really

- Perfectly possible to use large batches
  - Need to scale learning rate (or not, if saturated)
  - If SGD
    - xN batch -> x N learning rate
  - If adaptive (Adam, …)
    - Less clear, xN B -> x N or x sqrt(N) lr
    - An issue of Hessian eigenvalues

Takeaway

UNIVERSITÉ CÔTE D'AZUR

- Momentum can help get over small “hills”
  - Can avoid very shallow minima
- Adapt the learning rate (Step, cosine, exp, …)
  - Get smaller learning rates to go further “into” the minimum
- Per parameter adaptations (Adam, …)
  - Move differently for different parameters
- **Look at the torch.optim optimizer**

5. Backprop

We need the gradient…

$$\theta^{t+1} := \theta^t - \eta \nabla_\theta \hat{\mathcal{R}_\theta(B)}$$

- Requires finding the risk gradient wrt parameters

$$\nabla_\theta \hat{\mathcal{R}_\theta(B)} = \frac{1}{\#B} \sum_{k=0,\dots,B-1} \nabla_\theta l(f_\theta(x_k), y_k)$$

- Boils down to computing gradients for one sample

$$\nabla_\theta l(f_\theta(x), y)$$

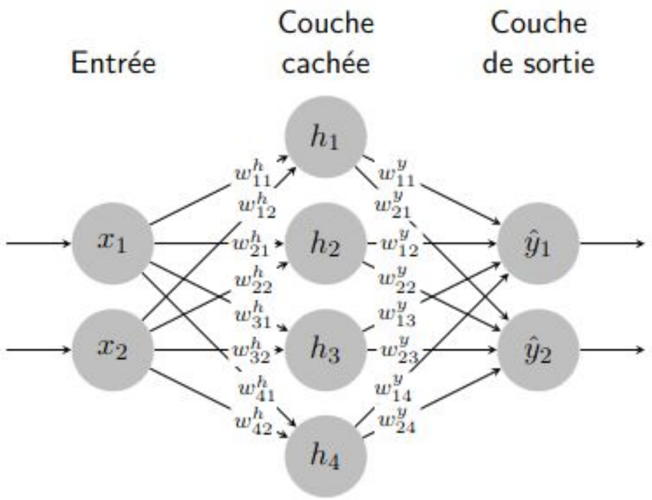
Backpropagation (Informal)

$$l := l(f_\theta(x), y)$$

- Networks are complex but made of simple parts!
  - Simple gradients of component functions
  - Chain-rule allows decomposition into simple gradients $\frac{\partial l}{\partial w} = \frac{\partial l}{\partial a}\frac{\partial a}{\partial w}$

- Need to store intermediate activations "a" to evaluate partial derivatives $\frac{\partial a}{\partial w}$

Example: Tanh MLP

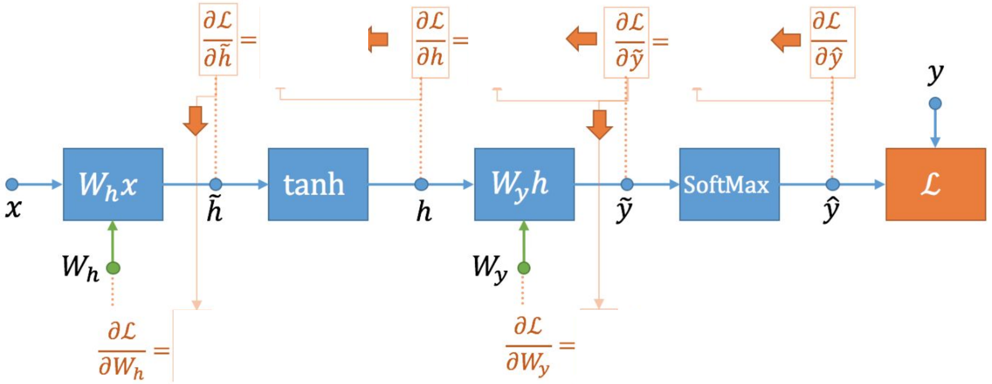


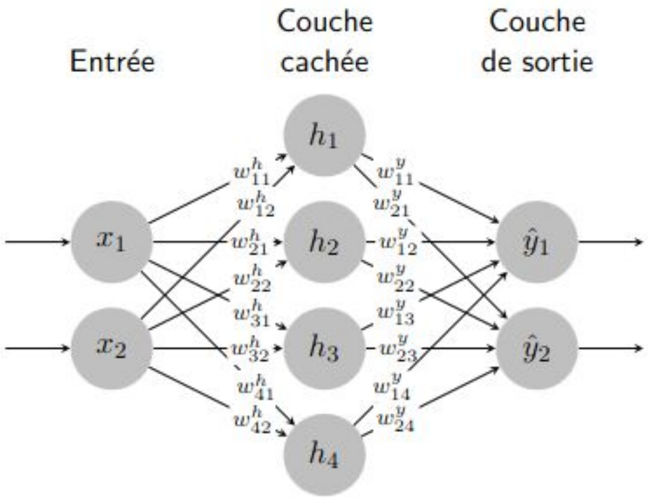
- Simple 1 hidden layer MLP
  - 2 inputs
  - 2 outputs
  - 4 hidden activations
- Classification problem
  - Outputs probabilities
  - Cross-entropy loss

$$l_{CE}(\hat{y}, y) = - \sum_{i=0}^{\#Classes-1} y_i \log(\hat{y}_i)$$

Example: Tanh MLP

$$l_{CE}(\hat{y}, y) = -\sum_{i=0}^{\#Classes-1} y_i \log(\hat{y}_i)$$

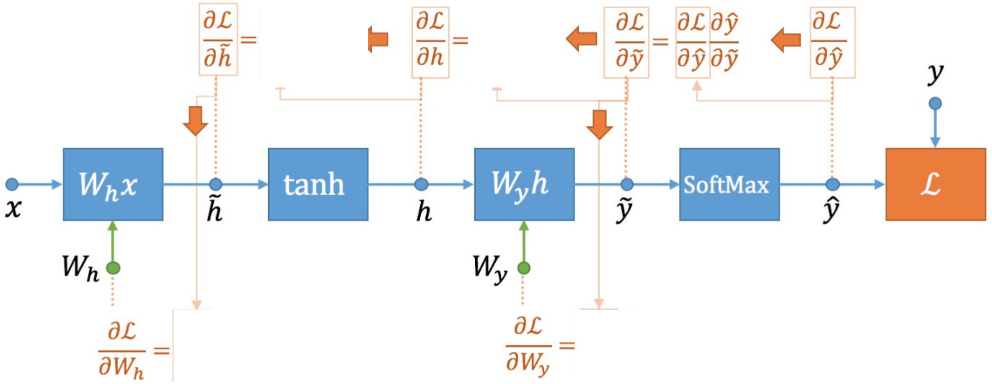


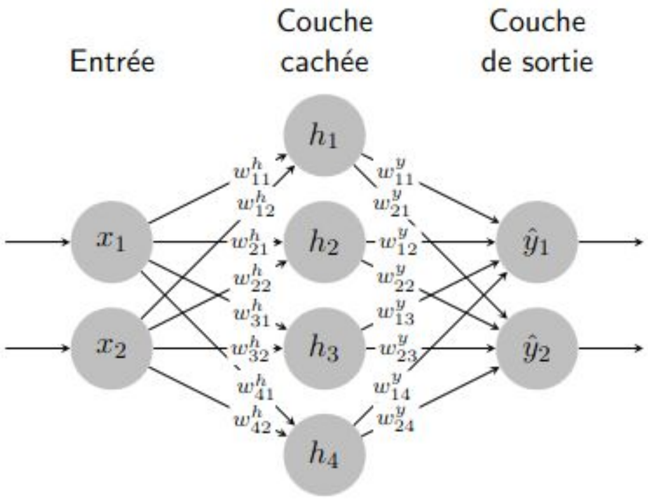


$$\begin{cases} \tilde{h}_i = \sum_{j=1}^{n_x} W^h_{i,j}\; x_j + b^h_i \\ h_i = \tanh(\tilde{h}_i) \\ \tilde{y}_i = \sum_{j=1}^{n_h} W^y_{i,j}\; h_j + b^y_i \\ \hat{y}_i = \mathrm{SoftMax}(\tilde{y}_i) = \dfrac{e^{\tilde{y}_i}}{\sum_{j=1}^{n_y} e^{\tilde{y}_j}} \end{cases}$$

Example: Tanh MLP

$$l_{CE}(\hat{y}, y) = - \sum_{i=0}^{\#Classes-1} y_i \log(\hat{y}_i)$$

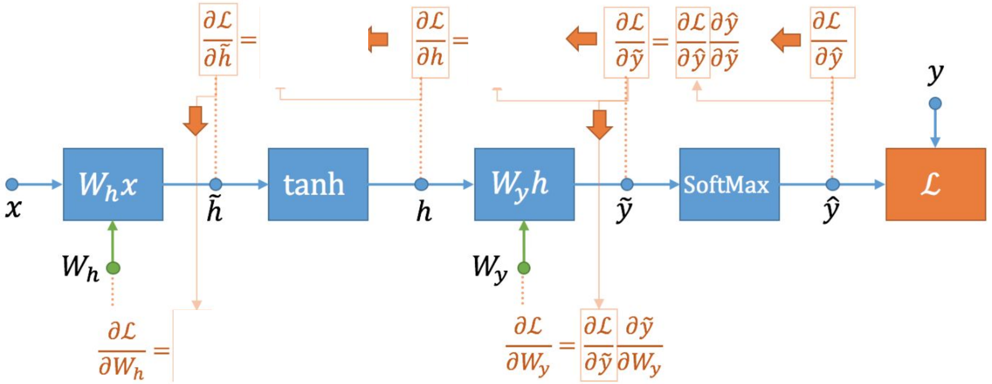


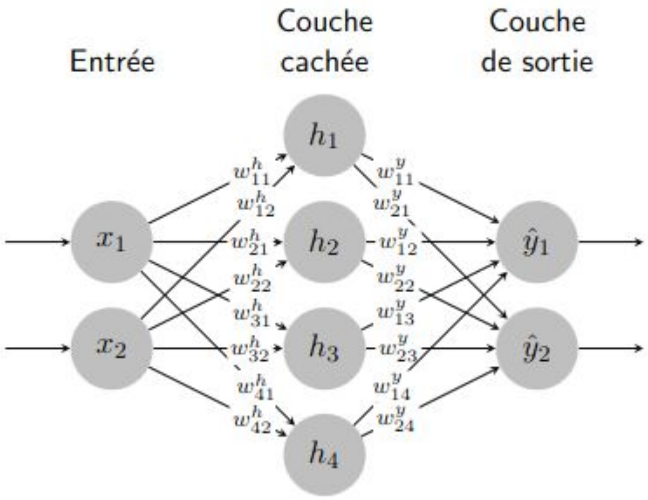


$$
\begin{cases}
\tilde{h}_i = \sum_{j=1}^{n_x} W^h_{i,j} \; x_j + b^h_i \\
h_i = \tanh(\tilde{h}_i) \\
\tilde{y}_i = \sum_{j=1}^{n_h} W^y_{i,j} \; h_j + b^y_i \\
\hat{y}_i = \mathrm{SoftMax}(\tilde{y}_i) = \dfrac{e^{\tilde{y}_i}}{\sum_{j=1}^{n_y} e^{\tilde{y}_j}}
\end{cases}
$$

Example: Tanh MLP

$$l_{CE}(\hat{y}, y) = - \sum_{i=0}^{\#Classes-1} y_i \log(\hat{y}_i)$$

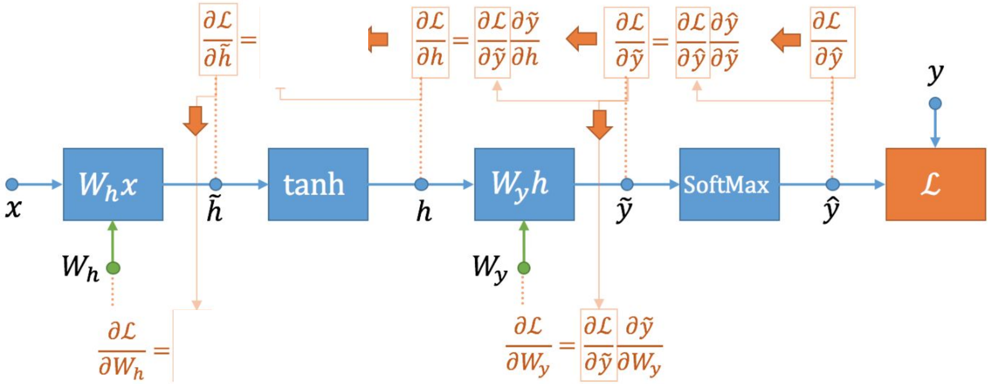


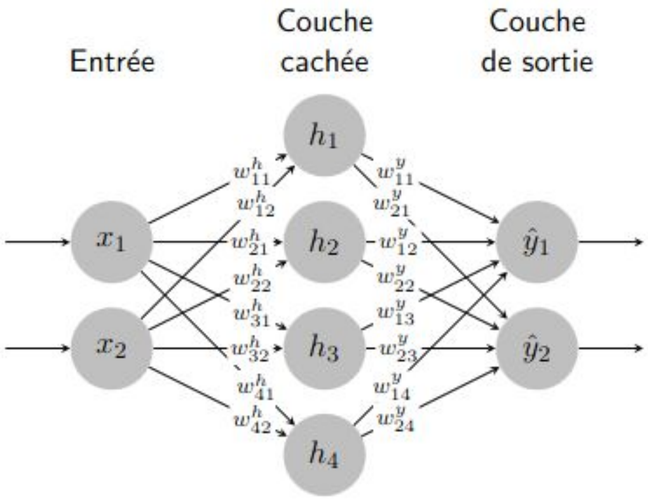


$$\begin{cases} \tilde{h}_i = \sum_{j=1}^{n_x} W^h_{i,j} \; x_j + b^h_i \\ h_i = \tanh(\tilde{h}_i) \\ \tilde{y}_i = \sum_{j=1}^{n_h} W^y_{i,j} \; h_j + b^y_i \\ \hat{y}_i = \mathrm{SoftMax}(\tilde{y}_i) = \dfrac{e^{\tilde{y}_i}}{\sum_{j=1}^{n_y} e^{\tilde{y}_j}} \end{cases}$$

Example: Tanh MLP

$$l_{CE}(\hat{y}, y) = -\sum_{i=0}^{\#Classes-1} y_i \log(\hat{y}_i)$$

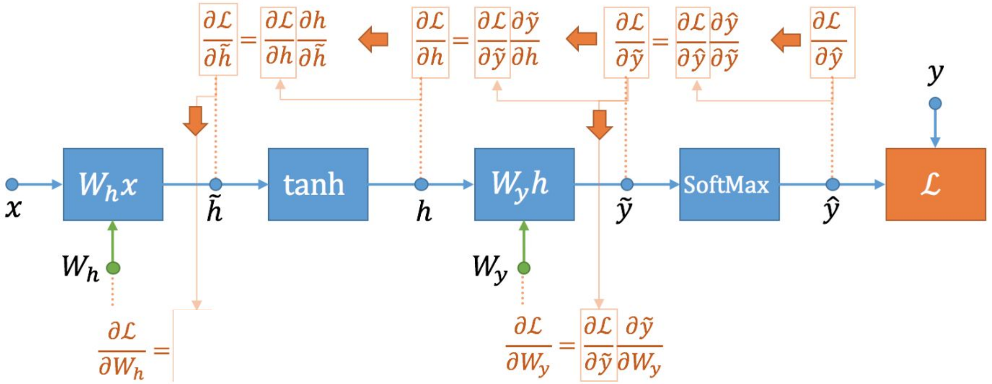


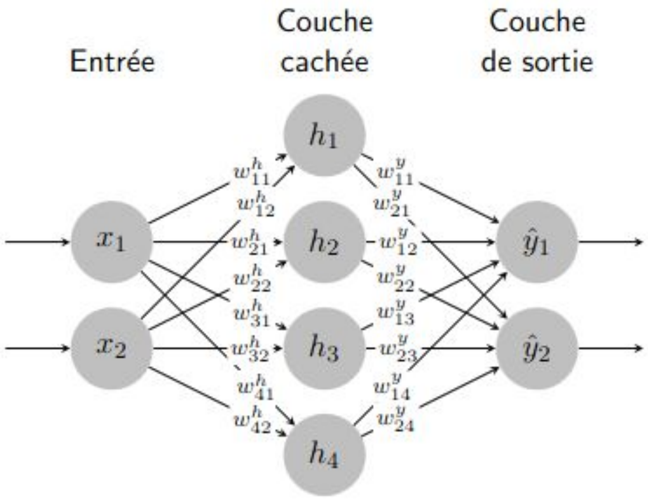


$$\begin{cases} \tilde{h}_i = \sum_{j=1}^{n_x} W^h_{i,j}\ x_j + b^h_i \\ h_i = \tanh(\tilde{h}_i) \\ \tilde{y}_i = \sum_{j=1}^{n_h} W^y_{i,j}\ h_j + b^y_i \\ \hat{y}_i = \mathrm{SoftMax}(\tilde{y}_i) = \dfrac{e^{\tilde{y}_i}}{\sum_{j=1}^{n_y} e^{\tilde{y}_j}} \end{cases}$$

Example: Tanh MLP

$$l_{CE}(\hat{y}, y) = -\sum_{i=0}^{\#Classes-1} y_i \log(\hat{y}_i)$$

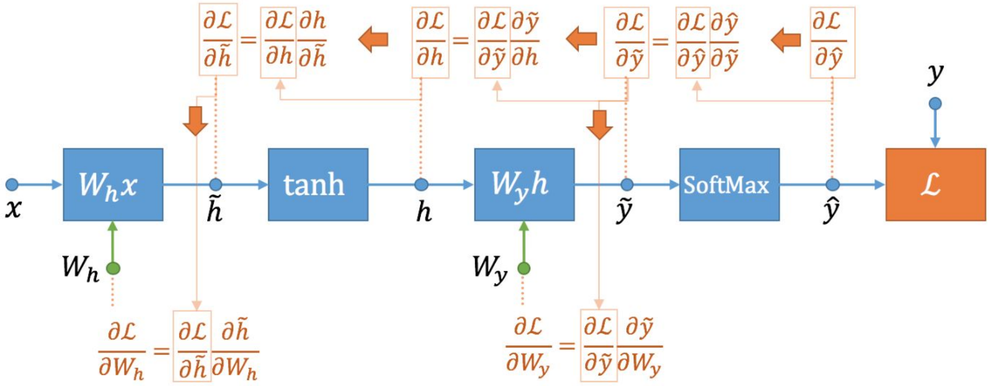


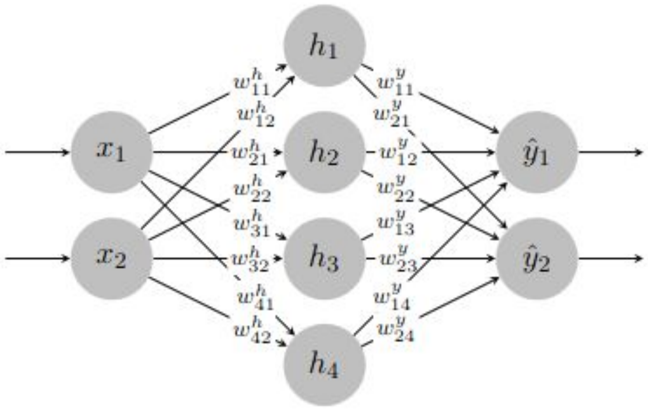


$$\begin{cases} \tilde{h}_i = \sum_{j=1}^{n_x} W^h_{i,j}\ x_j + b^h_i \\ h_i = \tanh(\tilde{h}_i) \\ \tilde{y}_i = \sum_{j=1}^{n_h} W^y_{i,j}\ h_j + b^y_i \\ \hat{y}_i = \text{SoftMax}(\tilde{y}_i) = \dfrac{e^{\tilde{y}_i}}{\sum_{j=1}^{n_y} e^{\tilde{y}_j}} \end{cases}$$

Example: Tanh MLP

$$l_{CE}(\hat{y}, y) = -\sum_{i=0}^{\#Classes-1} y_i \log(\hat{y}_i)$$



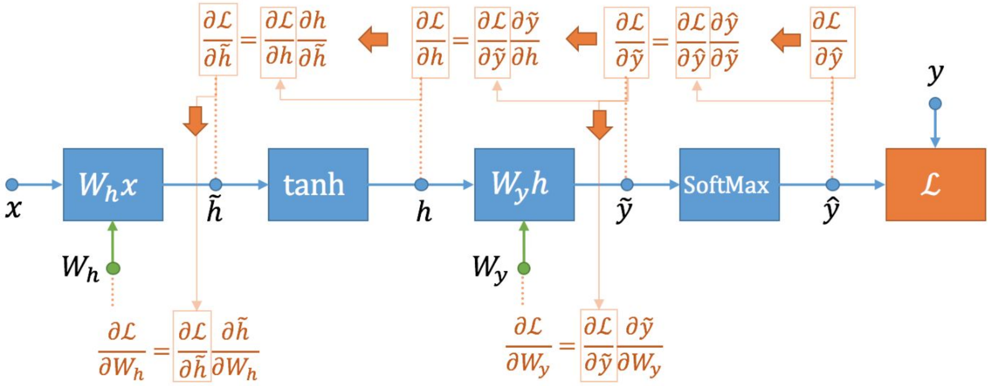


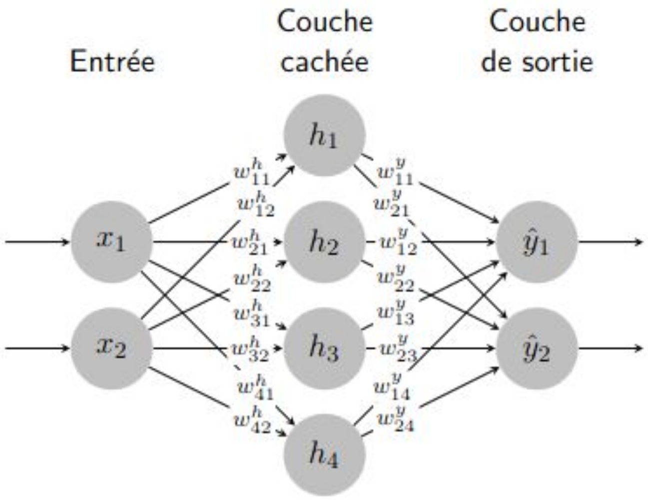
$$\begin{cases} \tilde{h}_i = \sum_{j=1}^{n_x} W^h_{i,j}\; x_j + b^h_i \\ h_i = \tanh(\tilde{h}_i) \\ \tilde{y}_i = \sum_{j=1}^{n_h} W^y_{i,j}\; h_j + b^y_i \\ \hat{y}_i = \mathrm{SoftMax}(\tilde{y}_i) = \dfrac{e^{\tilde{y}_i}}{\sum_{j=1}^{n_y} e^{\tilde{y}_j}} \end{cases}$$

$$\begin{cases} \delta^y_i = \dfrac{\partial \ell}{\partial \tilde{y}_i} = \\ \dfrac{\partial \ell}{\partial W^y_{i,j}} = \\ \dfrac{\partial \ell}{\partial b^y_i} = \end{cases}$$

Example: Tanh MLP

$$l_{CE}(\hat{y}, y) = - \sum_{i=0}^{\#Classes-1} y_i \log(\hat{y}_i)$$



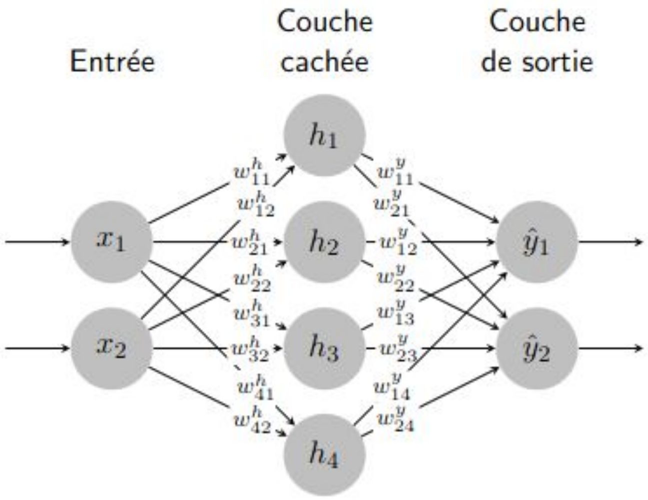


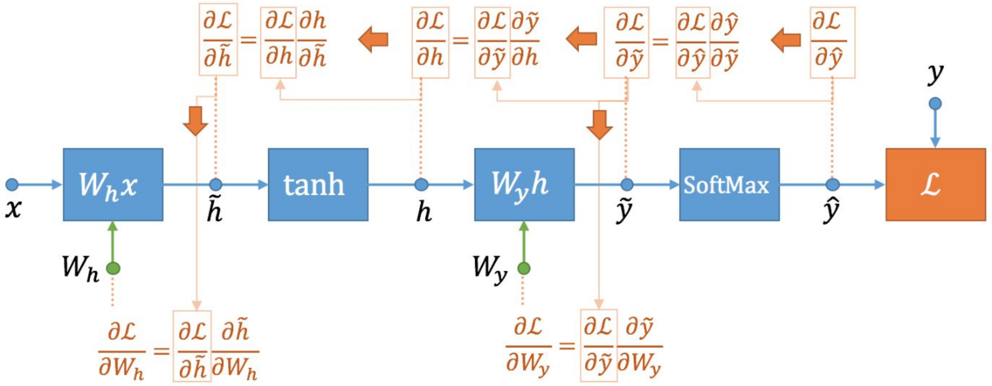
$$\begin{cases} \tilde{h}_i = \sum_{j=1}^{n_x} W^h_{i,j}\ x_j + b^h_i \\ h_i = \tanh(\tilde{h}_i) \\ \tilde{y}_i = \sum_{j=1}^{n_h} W^y_{i,j}\ h_j + b^y_i \\ \hat{y}_i = \mathrm{SoftMax}(\tilde{y}_i) = \dfrac{e^{\tilde{y}_i}}{\sum_{j=1}^{n_y} e^{\tilde{y}_j}} \end{cases}$$

$$\begin{cases} \delta^y_i = \dfrac{\partial \ell}{\partial \tilde{y}_i} = \hat{y}_i - y_i \\ \dfrac{\partial \ell}{\partial W^y_{i,j}} = \delta^y_i\ h_j \\ \dfrac{\partial \ell}{\partial b^y_i} = \delta^y_i \end{cases}$$

Example: Tanh MLP

$$l_{CE}(\hat{y}, y) = -\sum_{i=0}^{\#Classes-1} y_i \log(\hat{y}_i)$$



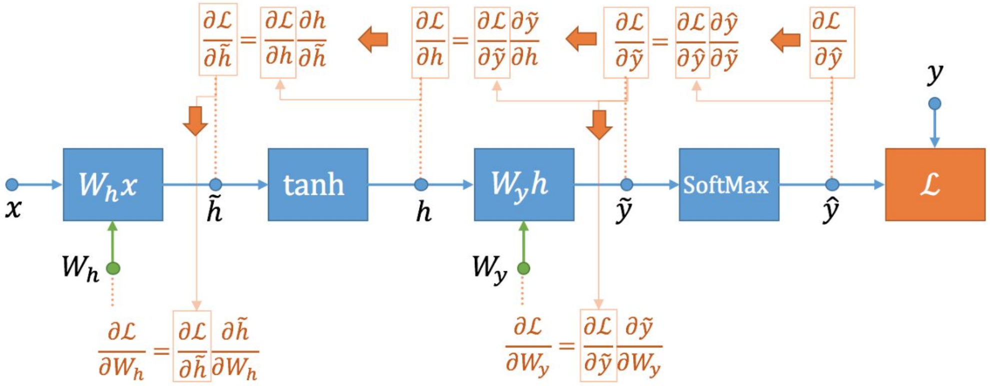


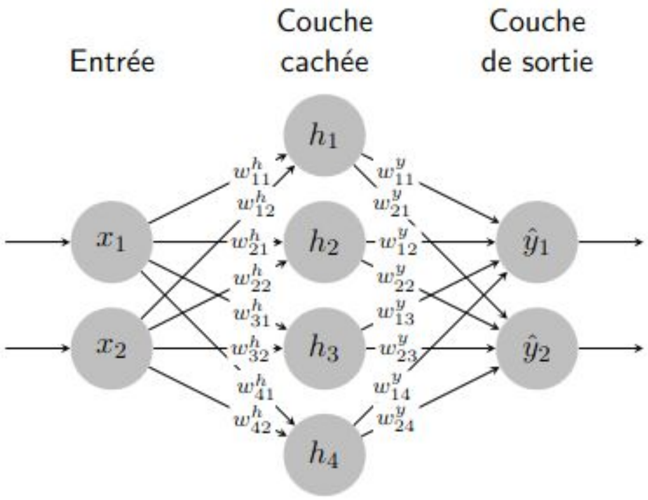
$$\begin{cases} \tilde{h}_i = \sum_{j=1}^{n_x} W^h_{i,j}\ x_j + b^h_i \\ h_i = \tanh(\tilde{h}_i) \\ \tilde{y}_i = \sum_{j=1}^{n_h} W^y_{i,j}\ h_j + b^y_i \\ \hat{y}_i = \mathrm{SoftMax}(\tilde{y}_i) = \dfrac{e^{\tilde{y}_i}}{\sum_{j=1}^{n_y} e^{\tilde{y}_j}} \end{cases}$$

$$\begin{cases} \delta^y_i = \dfrac{\partial \ell}{\partial \tilde{y}_i} = \hat{y}_i - y_i \\ \dfrac{\partial \ell}{\partial W^y_{i,j}} = \delta^y_i\ h_j \\ \dfrac{\partial \ell}{\partial b^y_i} = \delta^y_i \\ \delta^h_i = \dfrac{\partial \ell}{\partial \tilde{h}_i} = \\ \dfrac{\partial \ell}{\partial W^h_{i,j}} = \\ \dfrac{\partial \ell}{\partial b^h_i} = \end{cases}$$

Example: Tanh MLP

$$l_{CE}(\hat{y}, y) = -\sum_{i=0}^{\#Classes-1} y_i \log(\hat{y}_i)$$





$$\begin{cases} \tilde{h}_i = \sum_{j=1}^{n_x} W^h_{i,j}\ x_j + b^h_i \\ h_i = \tanh(\tilde{h}_i) \\ \tilde{y}_i = \sum_{j=1}^{n_h} W^y_{i,j}\ h_j + b^y_i \\ \hat{y}_i = \mathrm{SoftMax}(\tilde{y}_i) = \dfrac{e^{\tilde{y}_i}}{\sum_{j=1}^{n_y} e^{\tilde{y}_j}} \end{cases}$$

$$\begin{cases} \delta^y_i = \dfrac{\partial \ell}{\partial \tilde{y}_i} = \hat{y}_i - y_i \\ \dfrac{\partial \ell}{\partial W^y_{i,j}} = \delta^y_i\ h_j \\ \dfrac{\partial \ell}{\partial b^y_i} = \delta^y_i \\ \delta^h_i = \dfrac{\partial \ell}{\partial \tilde{h}_i} = (1 - h_i^2) \sum_{j=1}^{n_y} \delta^y_j W^y_{j,i} \\ \dfrac{\partial \ell}{\partial W^h_{i,j}} = \delta^h_i\ x_j \\ \dfrac{\partial \ell}{\partial b^h_i} = \delta^h_i \end{cases}$$

Takeaway

- Core problem: Find gradient updates
- Gradient can be computed efficiently with backpropagation
  - Chain rule starting from the "end" of the network
  - Keep network activation to evaluate gradients
  - Simple layers mean simple gradient blocks