V - Optimisation with ERM

Contents:

- *Empirical risk minimization for parameter learning*
- *Gradient descent*
- *Stochastic gradient descent*
- *Backpropagation for computing gradients in neural networks*

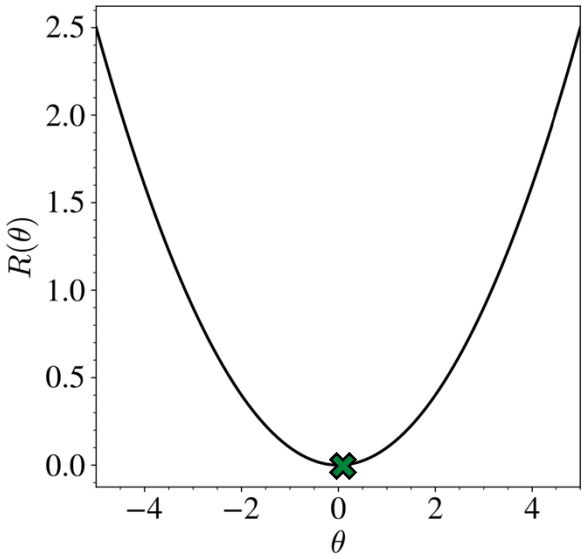
How to perform ERM in general?

- The aim of training in supervised learning is to compute

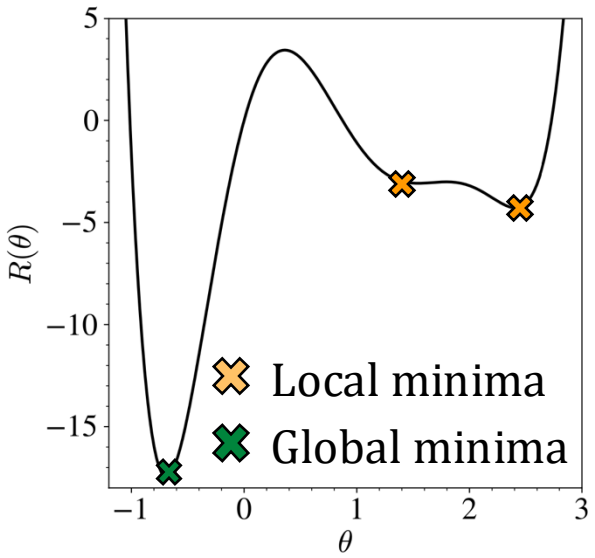
$$\widehat{\boldsymbol{\theta}} = \underset{\boldsymbol{\theta}}{\operatorname{argmin}}\, R(\boldsymbol{\theta}) = \underset{\boldsymbol{\theta}}{\operatorname{argmin}} \frac{1}{n}\sum_{i=1}^{n} \ell\left(f_{\boldsymbol{\theta}}\left(\boldsymbol{\phi}^{(i)}\right), y^{(i)}\right).$$

- In linear regression, we could directly optimise the empirical risk in closed-form, but **in general it is not possible.**
- How to perform the **Empirical Risk Minimisation** (ERM) in general?

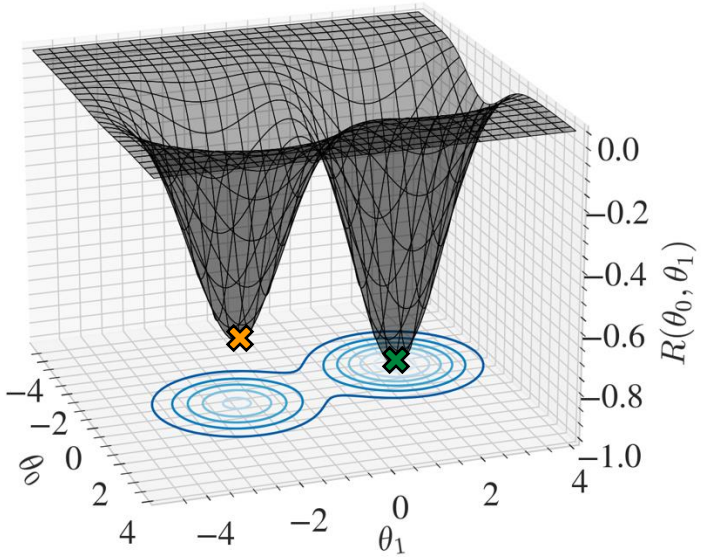
1D convex



1D non-convex

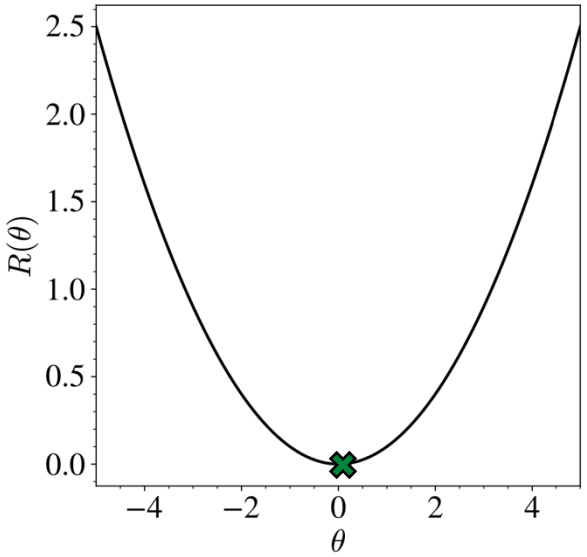


2D non-convex

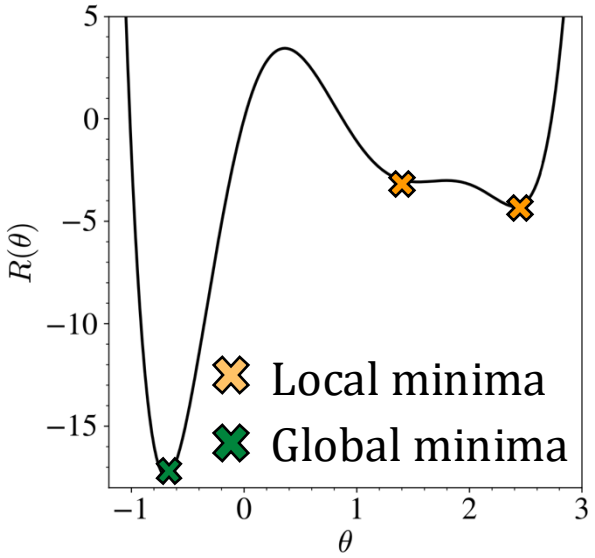


Naïve view and curse of dimensionality

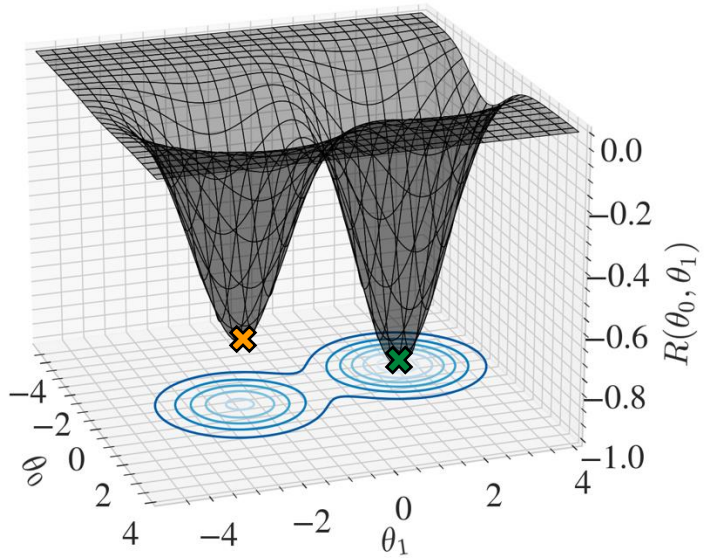
1D convex



1D non-convex



2D non-convex



- A naïve way of minimizing such functions could be to uniformly pave the parameter space and choose the one with the smallest value as being the minimum: this is **global optimization (grid search optimization)**
- This is where the **curse of dimensionality** kicks in (again). In general, we optimize models over many parameters $p \gg 1$ (imagine the pixels of an image) and all the points are far away from each other in high dimensions
- Sampling uniformly $[0, 1]^{10}$ with a step of 0.01 requires 10^{20} evaluations (think of GPT-3 and its 175 billion parameters!)

Gradient descent algorithm

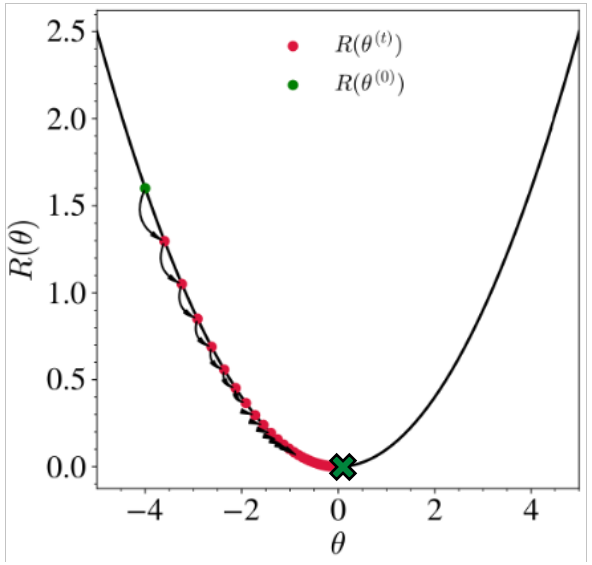
- A solution: local, directed search to navigate in the landscape
  - ↳ Numerical optimisation by **gradient descent**

Note: the superscript does not have to do with the training example here but with the time step ($\boldsymbol{\theta}$ is a parameter).

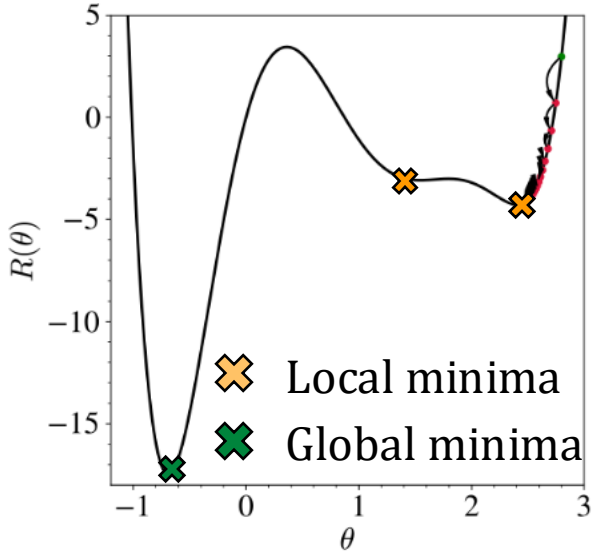
Algorithm: Gradient descent

1. Initialise $\boldsymbol{\theta}_0$ randomly
2. Compute $\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} - \eta \boldsymbol{\nabla}_{\boldsymbol{\theta}} R(\boldsymbol{\theta}^{(t)})$
3. Repeat 2 until a stopping criterion is satisfied

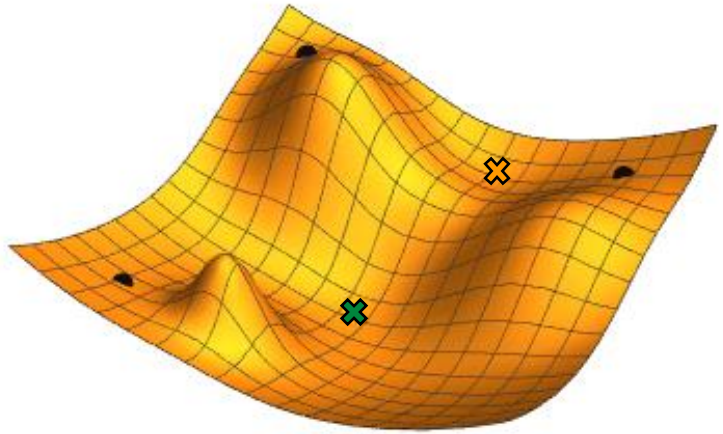
1D convex



1D non-convex



2D non-convex



Solutions in complex non-convex models depend on θ_0: it is an **hyperparameter**

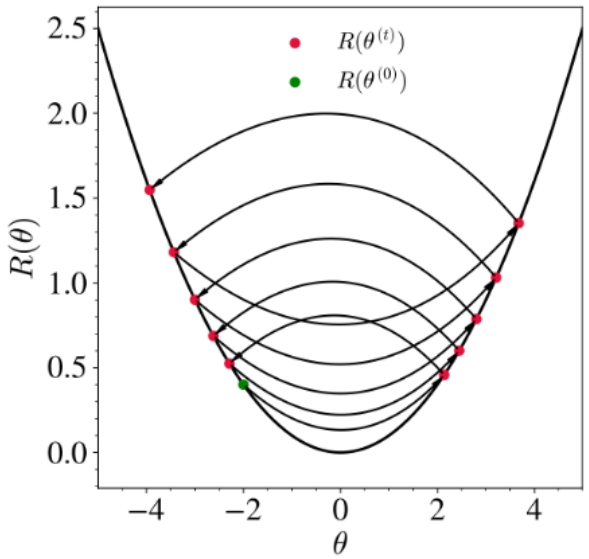
- Example of stopping criterion: $\left\|\boldsymbol{\theta}^{(t+1)} - \boldsymbol{\theta}^{(t)}\right\|_2^2 \leq \epsilon$, or fixed number of steps

A word about hyperparameters

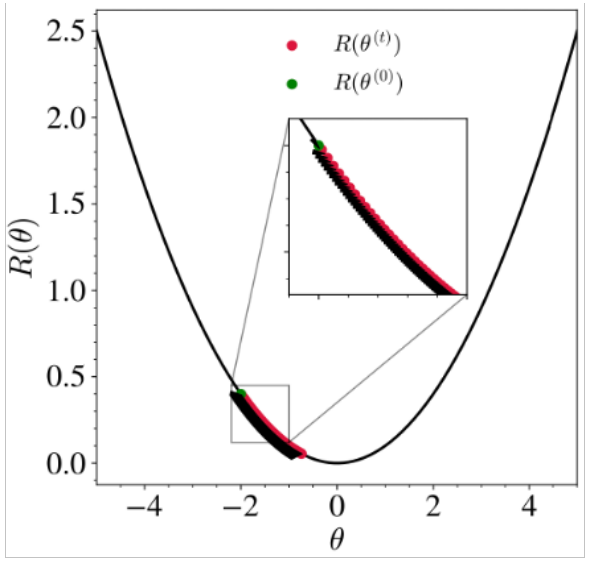
- One **hyperparameter**: the learning rate η
- A value that is too large can lead to divergence while, when too small, the computational cost explodes (+ stuck in small asperities)

Hyperparameter: parameter that is **not learned** during the optimisation. Ex: depth of tree in DTs, # of trees in RFs, learning rates, etc.

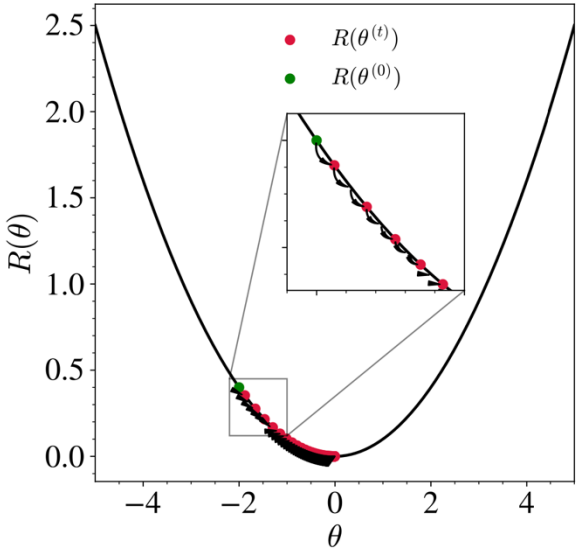
η too large



η too small



appropriate η



- Usually, we use **grid search** to find the hyperparameter performing best on a third dataset: the **validation set (or use cross-validation)**

Example: our first linear regression

- In linear regression, the empirical risk is

$$R(\boldsymbol{\theta}) = \frac{1}{n}\sum_{i=1}^{n}\left(\theta_0 + \theta_1 x_1^{(i)} - y^{(i)}\right)^2 \qquad \boldsymbol{\theta} = [\theta_0, \theta_1]^{\mathrm{T}}$$

- Let's apply the gradient descent algorithm starting from $\boldsymbol{\theta}^{(0)}$ random
- Then, we need to compute the gradient $\nabla_{\boldsymbol{\theta}} R(\boldsymbol{\theta})$

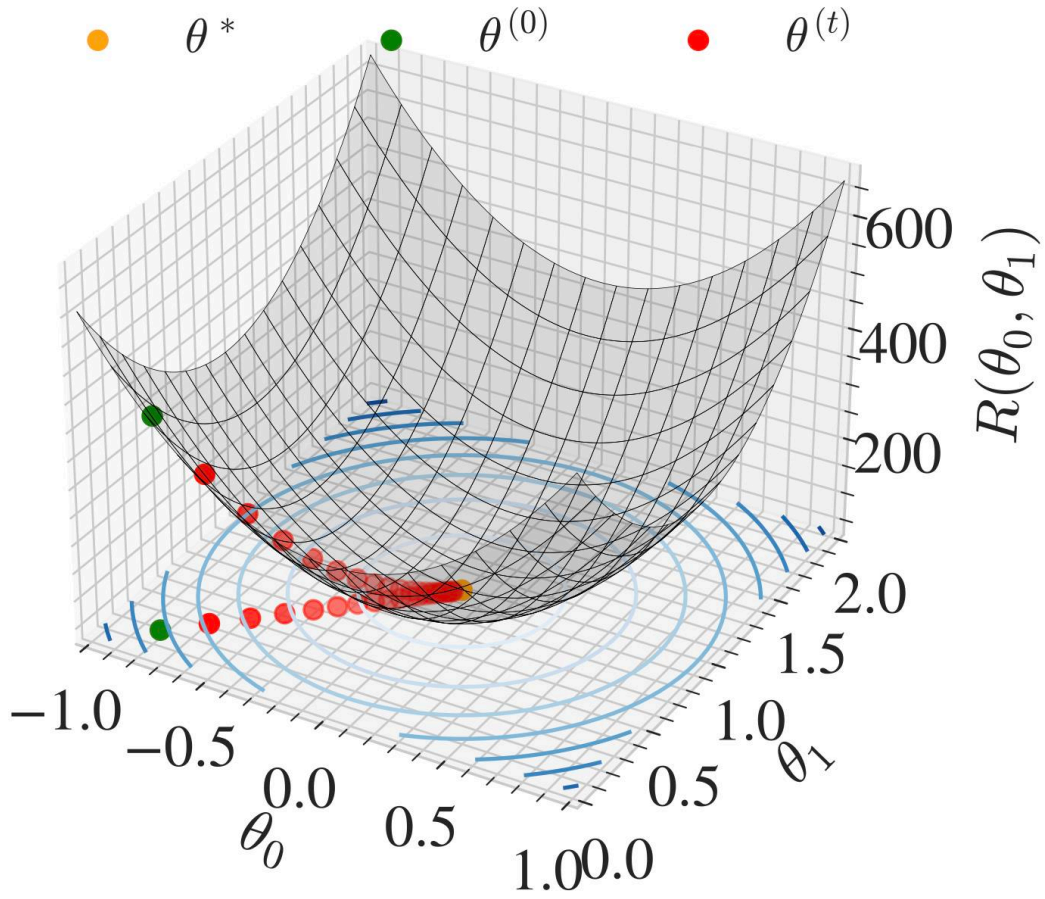
$$\frac{\partial R(\boldsymbol{\theta})}{\partial \theta_0} = \frac{2}{n}\sum_{i=1}^{n}\left(\theta_0 + \theta_1 x_1^{(i)} - y^{(i)}\right),$$

$$\frac{\partial R(\boldsymbol{\theta})}{\partial \theta_1} = \frac{2}{n}\sum_{i=1}^{n} x_1^{(i)}(\theta_0 + \theta_1 x_1^{(i)} - y^{(i)})$$

- Therefore, the update is

$$\theta_j^{(t+1)} = \theta_j^{(t)} - \frac{2\eta}{n}\sum_{i=1}^{n} x_j^{(i)}(\theta_0 + \theta_1 x_1^{(i)} - y^{(i)})$$

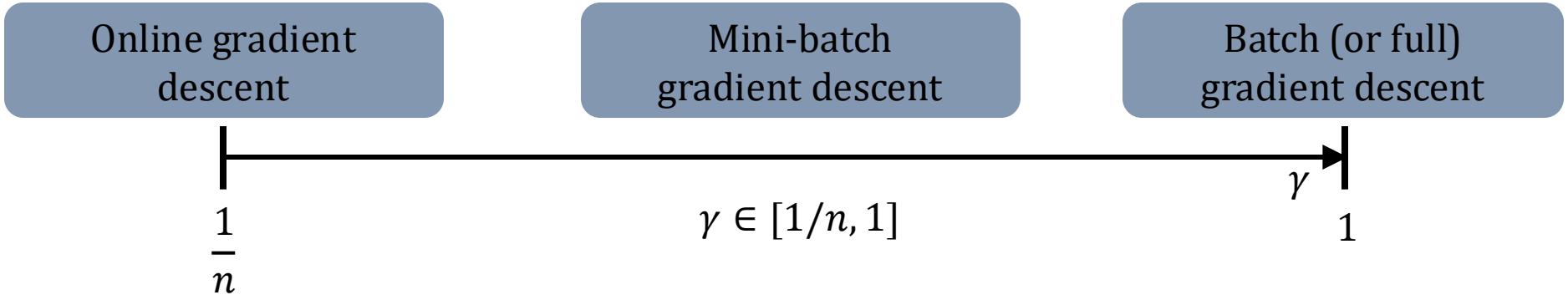
where $\forall\, i \in [1, \cdots, n]\ x_0^{(i)} = 1$



Note that I **standardized** the features. This is sometimes required when optimising some models. In GD, it allows faster convergence.

Stochastic gradient descent algorithm

- Problem of gradient descent: we **need the entire dataset** to compute $\nabla_{\boldsymbol{\theta}} R(\boldsymbol{\theta}^{(t)})$
- Solution: what about using only a fraction γ of the dataset chosen randomly?



Algorithm: Stochastic gradient descent

1. Initialise $\boldsymbol{\theta}_0$ randomly
2. For $e \in [1, \cdots, E]$
   - 2.1. Shuffle the dataset
   - 2.2. For $i \in [1, \cdots, \lfloor \gamma n \rfloor]$
     - 2.2.1. Compute $\boldsymbol{\theta}^{(i+1)} = \boldsymbol{\theta}^{(i)} - \eta \, \widehat{\nabla}_{\boldsymbol{\theta}} R(\boldsymbol{\theta}^{(i)})$

where $\widehat{\nabla} R(\boldsymbol{\theta}) = \sum_{j=i\gamma n}^{i\gamma n+\gamma n} \nabla R_j(\boldsymbol{\theta})$

- e is called an **epoch** and a set of γn training examples is called a **mini-batch**
- Usually**, SGD often converges faster than full-batch** GD
- It may however oscillate around a true minimum
- Under some technical assumptions, **SGD provides an almost sure convergence** to a local (resp. global) in non-convex (resp. convex) landscapes

Online gradient descent

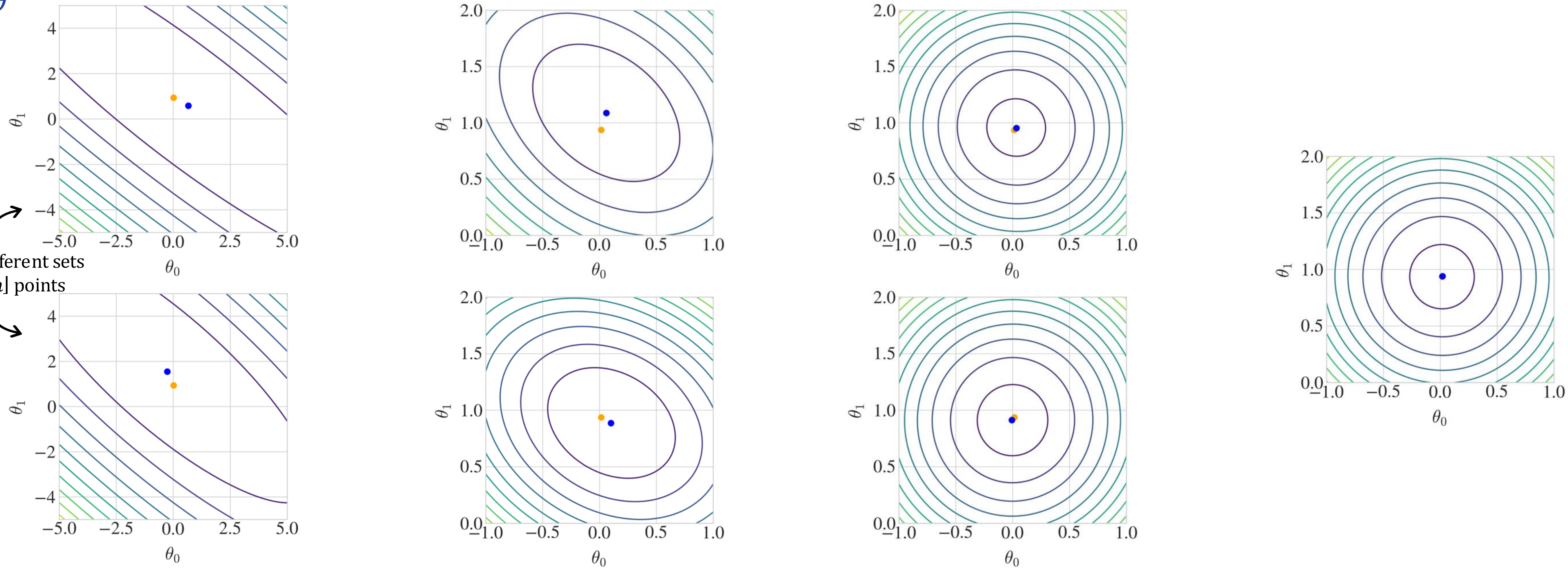
Mini-batch gradient descent

Batch (or full) gradient descent

$\frac{1}{n}$ —— 1

$\gamma \in [1/n, 1]$

θ^* $\hat{\theta}^*$



Two different sets of $\lfloor \gamma n \rfloor$ points

$\gamma = 2/n$

$\gamma = 0.1$

$\gamma = 0.5$

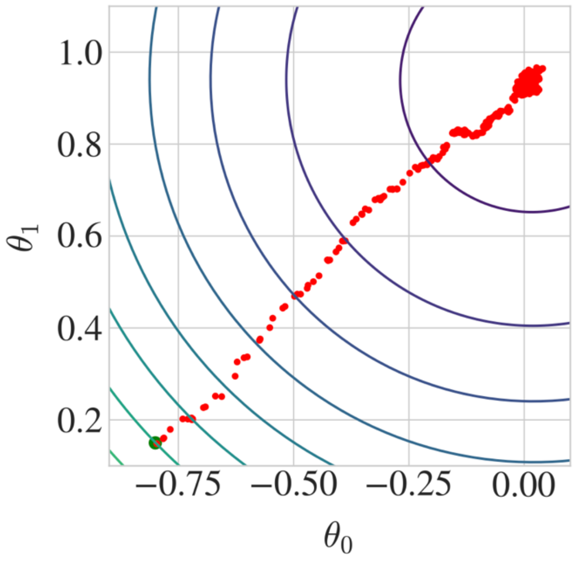
$\gamma = 1$

SGD on our linear regression

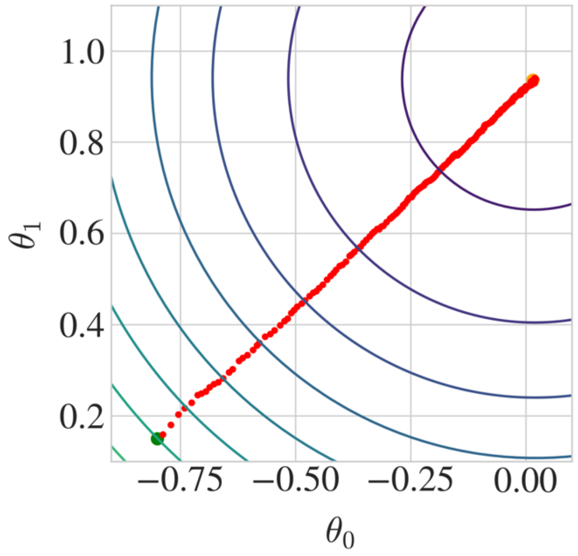
Online gradient descent | Mini-batch gradient descent | Batch (or full) gradient descent

$\frac{1}{n}$ ⟶ 1

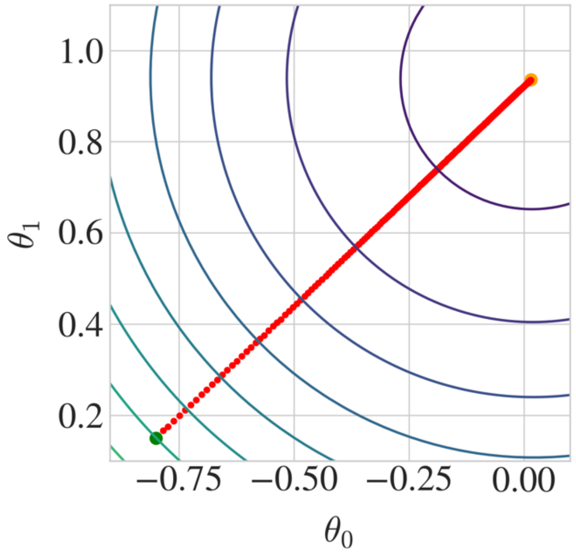
$\gamma \in [1/n, 1]$



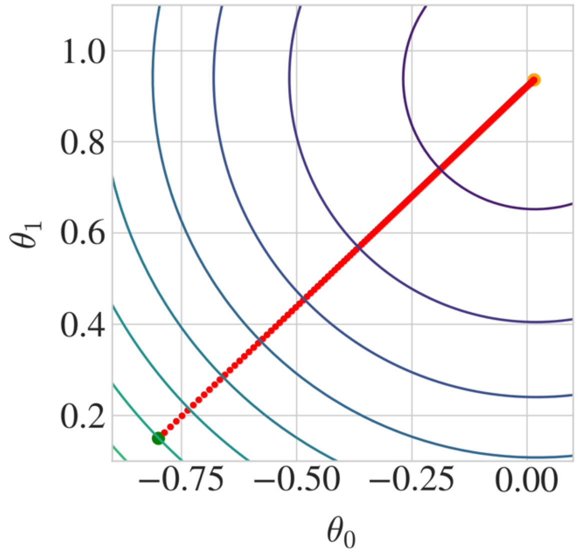
$\gamma = 2/n$



$\gamma = 0.1$



$\gamma = 0.5$



$\gamma = 1$

- The problem with neural networks is they are compositions of non-linear functions

$$y_j = s^{[L+1]}\left(\boldsymbol{W}^{[L+1]} s^{[L]}\left(\boldsymbol{W}^{[L]} \dots s^{[1]}\left(\boldsymbol{W}^{[1]}\boldsymbol{x}\right)\right)\right)$$

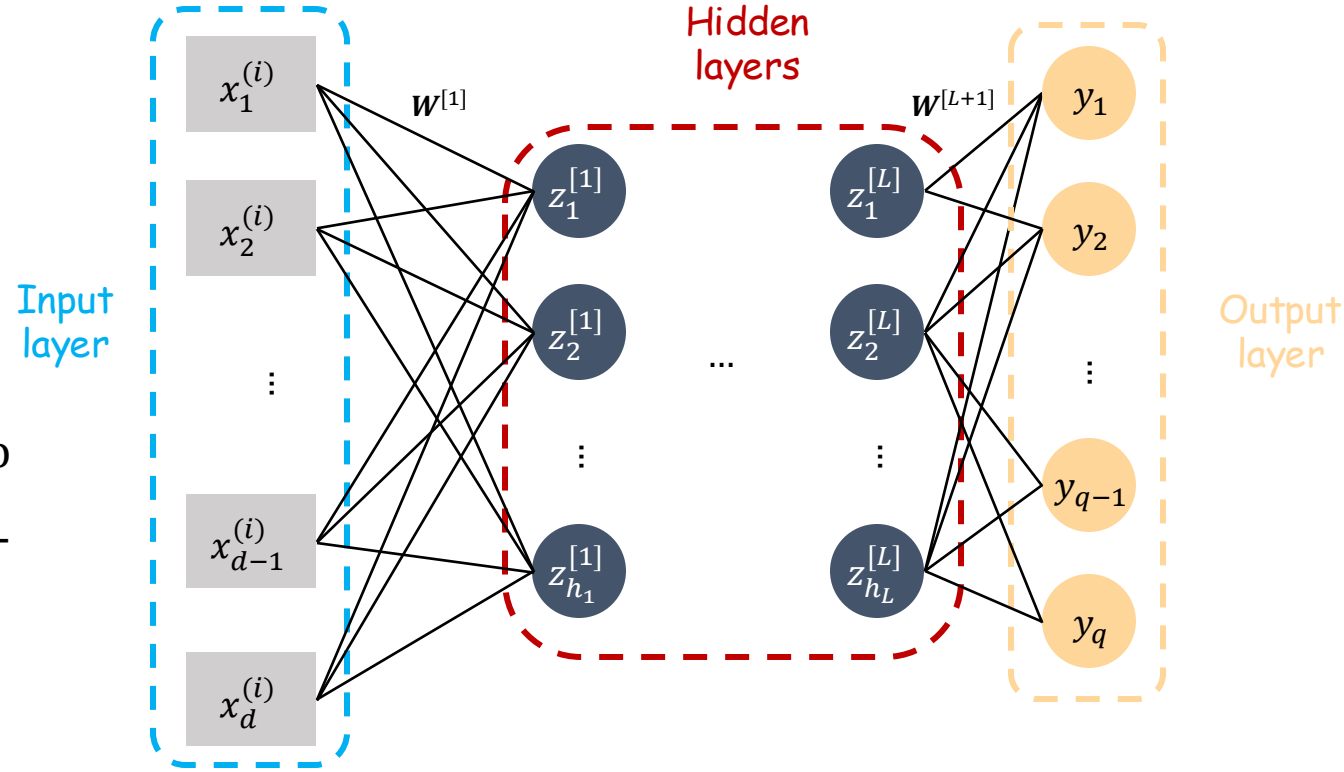
$$\boldsymbol{W}^{[l]} = \left[\boldsymbol{w}_1^{[l]}, \boldsymbol{w}_2^{[l]}, \dots, \boldsymbol{w}_{h_l}\right]^{\mathrm{T}} \in \mathbb{R}^{h_l \times (h_{l-1}+1)}$$

$$\boldsymbol{w}_j^{[l]} = \left[b_j^{[l]}, w_{1j}^{[l]}, w_{2j}^{[l]}, \dots, w_{h_l j}^{[l]}\right]$$

- We however need to optimize the cost function to obtain the "best" values of $\boldsymbol{W}$ producing the closer-to-optimal target values

- How to compute $\frac{\partial \ell(\boldsymbol{W}^{[0]},\dots,\boldsymbol{W}^{[L+1]})}{\partial w_{ij}^{[l]}}$?

- **The backpropagation of errors**: an application of the **chain rule!**



- Take the example of a fully-connected network with $d = 3$, one hidden layer, and two output neurons:

$$\hat{\boldsymbol{y}} = s\left(\boldsymbol{W}^{[2]} s\left(\boldsymbol{W}^{[1]}\boldsymbol{x}\right)\right) \quad \text{with } \boldsymbol{x} = [1, x_1, x_2, \ldots, x_d]^{\mathrm{T}}$$

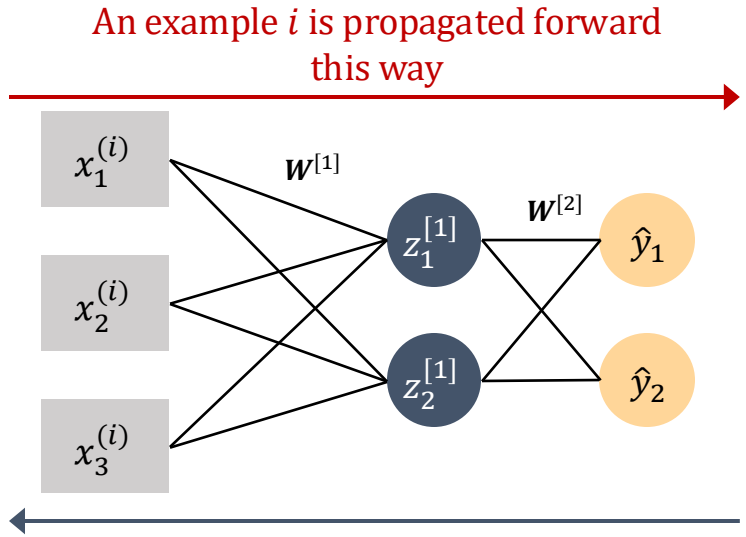
$$\boldsymbol{W}^{[1]} = \begin{bmatrix} b_1^{[1]} & w_{11}^{[1]} & w_{12}^{[1]} & w_{13}^{[1]} \\ b_2^{[1]} & w_{21}^{[1]} & w_{22}^{[1]} & w_{23}^{[1]} \end{bmatrix} \qquad \boldsymbol{W}^{[2]} = \begin{bmatrix} b_1^{[2]} & w_{11}^{[2]} & w_{12}^{[2]} \\ b_2^{[2]} & w_{21}^{[2]} & w_{22}^{[2]} \end{bmatrix}$$

- Consider the squared error loss function for each training example i

$$\ell\left(\boldsymbol{W}^{[1]}, \boldsymbol{W}^{[2]}\right) = \frac{1}{2}\sum_{j=1}^{2}\left(y_j - \hat{y}_j\right)^2$$

- From the equation of $\hat{\boldsymbol{y}}$, we see the contribution of $\boldsymbol{W}^{[2]}$ is "closer" to the output than $\boldsymbol{W}^{[1]}$

- Let's compute $\dfrac{\partial \ell\left(\boldsymbol{W}^{[1]}, \boldsymbol{W}^{[2]}\right)}{\partial w_{11}^{[2]}}$



- Using the chain rule

$$\frac{\partial \ell(\boldsymbol{W}^{[1]}, \boldsymbol{W}^{[2]})}{\partial w_{11}^{[2]}} = \frac{\partial \ell}{\partial u_1^{[2]}} \times \frac{\partial u_1^{[2]}}{\partial w_{11}^{[2]}}$$

where $u_1^{[2]}$ is the pre-activation of the unit 1 in layer 2 (here output layer)

$$u_1^{[2]} = w_{11}^{[2]} z_1^{[1]} + w_{21}^{[2]} z_2^{[1]} + b_1^{[2]}$$

- The second term is then easy to compute as $\frac{\partial u_1^{[2]}}{\partial w_{11}^{[2]}} = z_1^{[1]}$
- For the first term, use the chain rule again leads to

$$\delta_1^{[2]} = \frac{\partial \ell}{\partial u_1^{[2]}} = \frac{\partial \ell}{\partial \hat{y}_1} \frac{\partial \hat{y}_1}{\partial u_1^{[2]}}$$

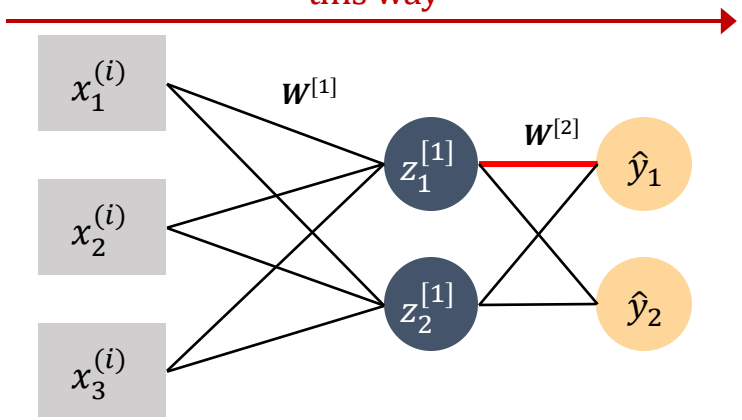
with $\hat{y}_1 = s\left(u_1^{[2]}\right) = s(w_{11}^{[2]} z_1^{[1]} + w_{21}^{[2]} z_2^{[1]} + b_1^{[2]})$

Finally,

$$\frac{\partial \ell(\boldsymbol{W}^{[1]}, \boldsymbol{W}^{[2]})}{\partial w_{11}^{[2]}} = -(y_1 - \hat{y}_1) s'\left(u_1^{[2]}\right) z_1^{[1]}$$

$$\ell(\boldsymbol{W}^{[1]}, \boldsymbol{W}^{[2]}) = \frac{1}{2} \sum_{j=1}^{2} (y_j - \hat{y}_j)^2$$

An example i is propagated forward this way



The gradient is propagated layer by layer from output to input

$$\hat{\boldsymbol{y}} = s\left(\boldsymbol{W}^{[2]} s(\boldsymbol{W}^{[1]} \boldsymbol{x})\right)$$

$$\boldsymbol{W}^{[1]} = \begin{bmatrix} b_1^{[1]} & w_{11}^{[1]} & w_{12}^{[1]} & w_{13}^{[1]} \\ b_2^{[1]} & w_{21}^{[1]} & w_{22}^{[1]} & w_{23}^{[1]} \end{bmatrix}$$

$$\boldsymbol{W}^{[2]} = \begin{bmatrix} b_1^{[2]} & w_{11}^{[2]} & w_{12}^{[2]} \\ b_2^{[2]} & w_{21}^{[2]} & w_{22}^{[2]} \end{bmatrix}$$

- You can proceed the same for all the weights linked to the output layer
- What about parameters in the hidden layers?
- Let's compute

$$\frac{\partial \ell(\boldsymbol{W}^{[1]}, \boldsymbol{W}^{[2]})}{\partial w_{11}^{[1]}} = \frac{\partial \ell}{\partial u_1^{[1]}} \times \frac{\partial u_1^{[1]}}{\partial w_{11}^{[1]}}$$

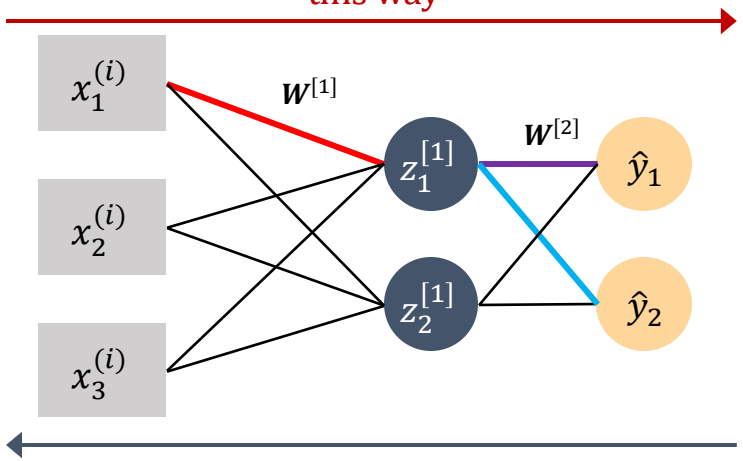
- The second term is still easy to compute
- For the first term, we have $u_1^{[1]} = w_{11}^{[1]} x_1 + w_{21}^{[1]} x_2 + b_1^{[1]}$ and observe that there are two paths to reach the weight $w_{11}^{[1]}$:

$$\delta_1^{[1]} = \frac{\partial \ell}{\partial u_1^{[2]}} \frac{\partial u_1^{[2]}}{\partial u_1^{[1]}} + \frac{\partial \ell}{\partial u_2^{[2]}} \frac{\partial u_2^{[2]}}{\partial u_1^{[1]}}$$

Already computed in the previous layer (hence the name "Backpropagation")

$$\ell(\boldsymbol{W}^{[1]}, \boldsymbol{W}^{[2]}) = \frac{1}{2} \sum_{j=1}^{2} (y_j - \hat{y}_j)^2$$

An example i is propagated forward this way



The gradient is propagated layer by layer from output to input

$$\hat{\boldsymbol{y}} = s\left(\boldsymbol{W}^{[2]} s(\boldsymbol{W}^{[1]} \boldsymbol{x})\right)$$

$$\boldsymbol{W}^{[1]} = \begin{bmatrix} b_1^{[1]} & w_{11}^{[1]} & w_{12}^{[1]} & w_{13}^{[1]} \\ b_2^{[1]} & w_{21}^{[1]} & w_{22}^{[1]} & w_{23}^{[1]} \end{bmatrix}$$

$$\boldsymbol{W}^{[2]} = \begin{bmatrix} b_1^{[2]} & w_{11}^{[2]} & w_{12}^{[2]} \\ b_2^{[2]} & w_{21}^{[2]} & w_{22}^{[2]} \end{bmatrix}$$

$$\delta_1^{[1]} = \frac{\partial \ell}{\partial u_1^{[2]}} \frac{\partial u_1^{[2]}}{\partial u_1^{[1]}} + \frac{\partial \ell}{\partial u_2^{[2]}} \frac{\partial u_2^{[2]}}{\partial u_1^{[1]}}$$

Already computed in the previous layer (hence the name "Backpropagation")

- To compute $\frac{\partial u_1^{[2]}}{\partial u_1^{[1]}}$ and $\frac{\partial u_2^{[2]}}{\partial u_1^{[1]}}$, we can use the same chain rule again giving

$$\frac{\partial u_1^{[2]}}{\partial u_1^{[1]}} = \frac{\partial u_1^{[2]}}{\partial z_1^{[1]}} \frac{\partial z_1^{[1]}}{\partial u_1^{[1]}} = w_{11}^{[2]} s'\left(u_1^{[1]}\right) \qquad \frac{\partial u_2^{[2]}}{\partial u_1^{[1]}} = \frac{\partial u_2^{[2]}}{\partial z_1^{[1]}} \frac{\partial z_1^{[1]}}{\partial u_1^{[1]}} = w_{21}^{[2]} s'\left(u_1^{[1]}\right)$$

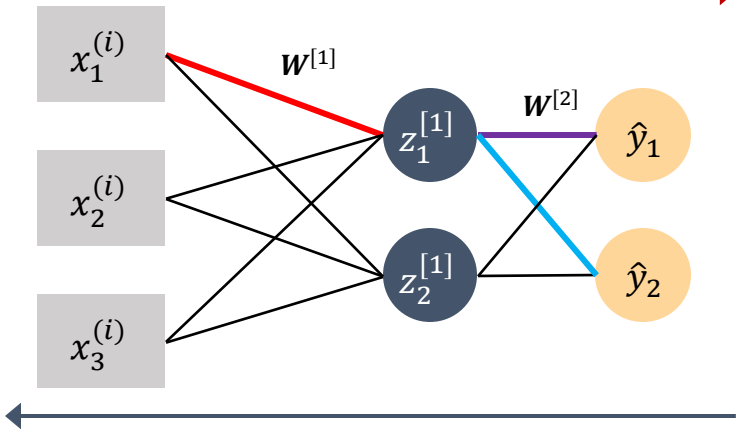
- Hence,

$$\frac{\partial \ell\left(\boldsymbol{W}^{[1]}, \boldsymbol{W}^{[2]}\right)}{\partial w_{11}^{[1]}} = \left(\delta_1^{[2]} w_{11}^{[2]} + \delta_2^{[2]} w_{21}^{[2]}\right) s'\left(u_1^{[1]}\right) x_1$$

- Where everything is known!

$$\ell\left(\boldsymbol{W}^{[1]}, \boldsymbol{W}^{[2]}\right) = \frac{1}{2} \sum_{j=1}^{2} \left(y_j - \hat{y}_j\right)^2$$

An example i is propagated forward this way



The gradient is propagated layer by layer from output to input

$$\hat{\boldsymbol{y}} = s\left(\boldsymbol{W}^{[2]} s\left(\boldsymbol{W}^{[1]} \boldsymbol{x}\right)\right)$$

$$\boldsymbol{W}^{[1]} = \begin{bmatrix} b_1^{[1]} & w_{11}^{[1]} & w_{12}^{[1]} & w_{13}^{[1]} \\ b_2^{[1]} & w_{21}^{[1]} & w_{22}^{[1]} & w_{23}^{[1]} \end{bmatrix}$$

$$\boldsymbol{W}^{[2]} = \begin{bmatrix} b_1^{[2]} & w_{11}^{[2]} & w_{12}^{[2]} \\ b_2^{[2]} & w_{21}^{[2]} & w_{22}^{[2]} \end{bmatrix}$$

Backprop: summary and conclusions

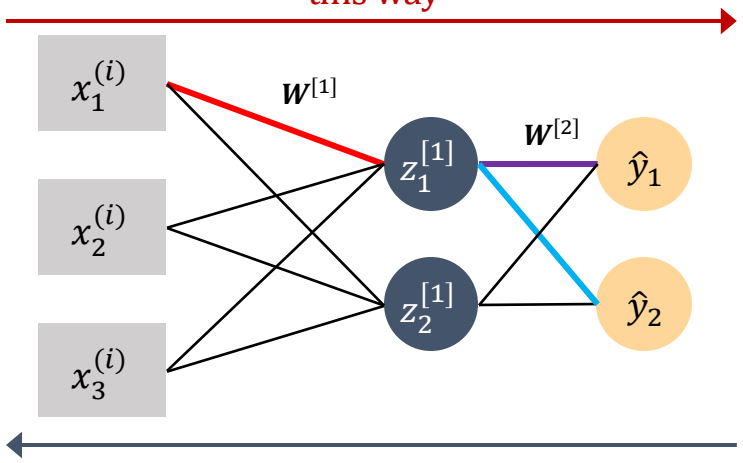
1. Writing those equations **propagating the errors from output to input recursively** for both weights and biases leads to the **backpropagation algorithm.**

2. **Together with the SGD algorithm they allow to train efficiently networks with possibly many parameters.**

3. Modern programming tools implement an automatic version of this algorithm called ***automatic differentiation*** generating the code for backward equations from forward propagation declaration.

$$\ell(\boldsymbol{W}^{[1]}, \boldsymbol{W}^{[2]}) = \frac{1}{2}\sum_{j=1}^{2}(y_j - \hat{y}_j)^2$$

An example i is propagated forward this way



The gradient is propagated layer by layer from output to input

$$\hat{\boldsymbol{y}} = s\left(\boldsymbol{W}^{[2]}s(\boldsymbol{W}^{[1]}\boldsymbol{x})\right)$$

$$\boldsymbol{W}^{[1]} = \begin{bmatrix} b_1^{[1]} & w_{11}^{[1]} & w_{12}^{[1]} & w_{13}^{[1]} \\ b_2^{[1]} & w_{21}^{[1]} & w_{22}^{[1]} & w_{23}^{[1]} \end{bmatrix}$$

$$\boldsymbol{W}^{[2]} = \begin{bmatrix} b_1^{[2]} & w_{11}^{[2]} & w_{12}^{[2]} \\ b_2^{[2]} & w_{21}^{[2]} & w_{22}^{[2]} \end{bmatrix}$$