CSC_4AI07_TP
"Introduction to Deep Learning"

Geoffroy Peeters

contact: geoffroy.peeters@telecom-paris.fr
Télécom-Paris, IP-Paris, France

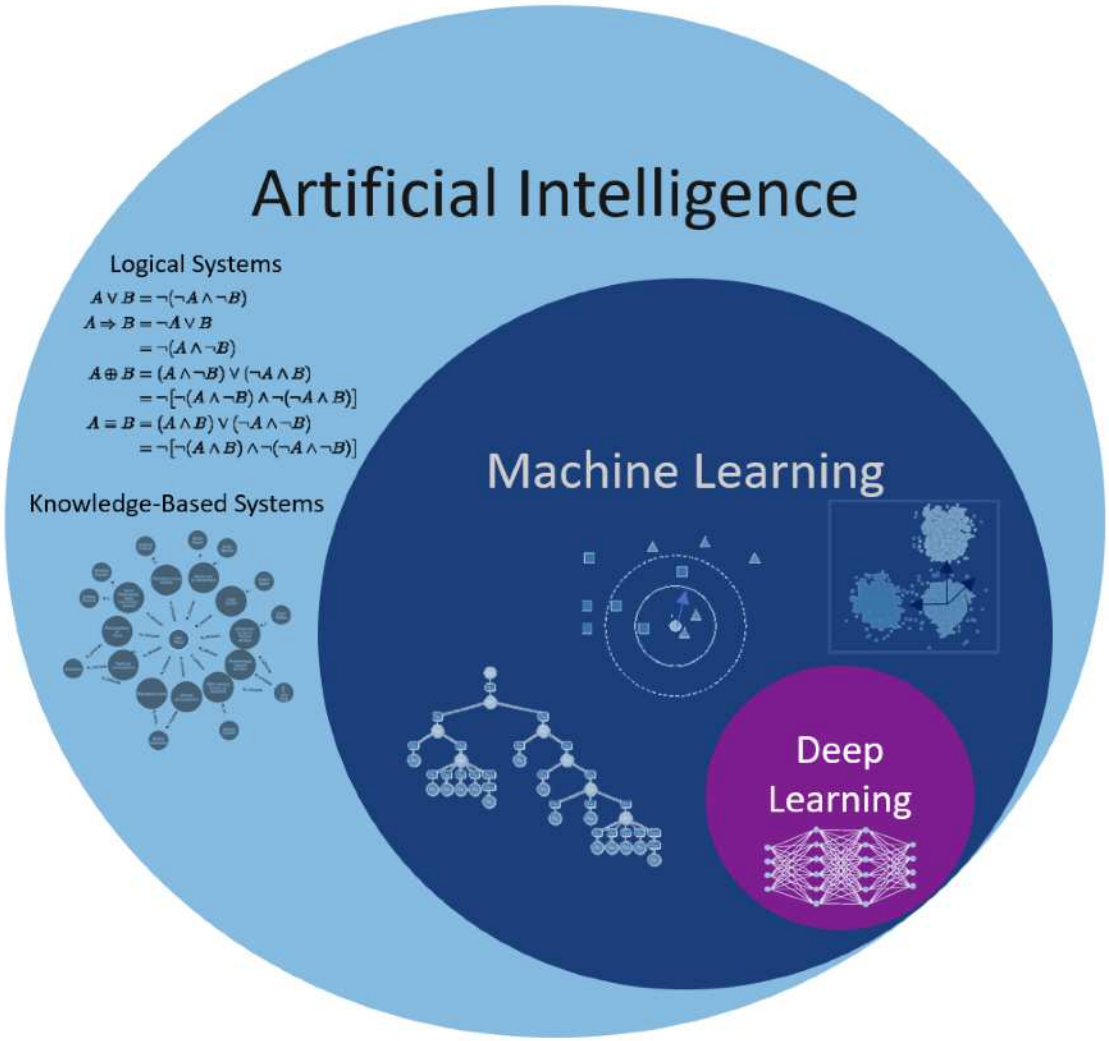
Planning

Date		Topic	Room	Speaker
30/04/2025 AM	Wednesday	Lecture MLP	Thévenin	PEETERS Geoffroy
14/05/2025 AM	Wednesday	Lab MLP + Quizz	1A...	PEETERS Geoffroy et al.
21/05/2025 AM	Wednesday	Lecture CNN	Thévenin	Stepan Alaniz
28/05/2025 AM	Wednesday	Lab CNN + Quizz	1A...	Stepan Alaniz et al.
04/06/2025 AM	Wednesday	Lecture VAE	Thévenin	Arthur Leclaire
11/06/2025 AM	Wednesday	Lecture RNN	Thévenin	PEETERS Geoffroy
18/06/2025 AM	Wednesday	Lab RNN + Quizz	1A ...	PEETERS Geoffroy et al.
25/06/2025	Wednesday	Examen	Thévenin/ Estaunié	

https://ecampus.paris-saclay.fr/course/view.php?id=22120

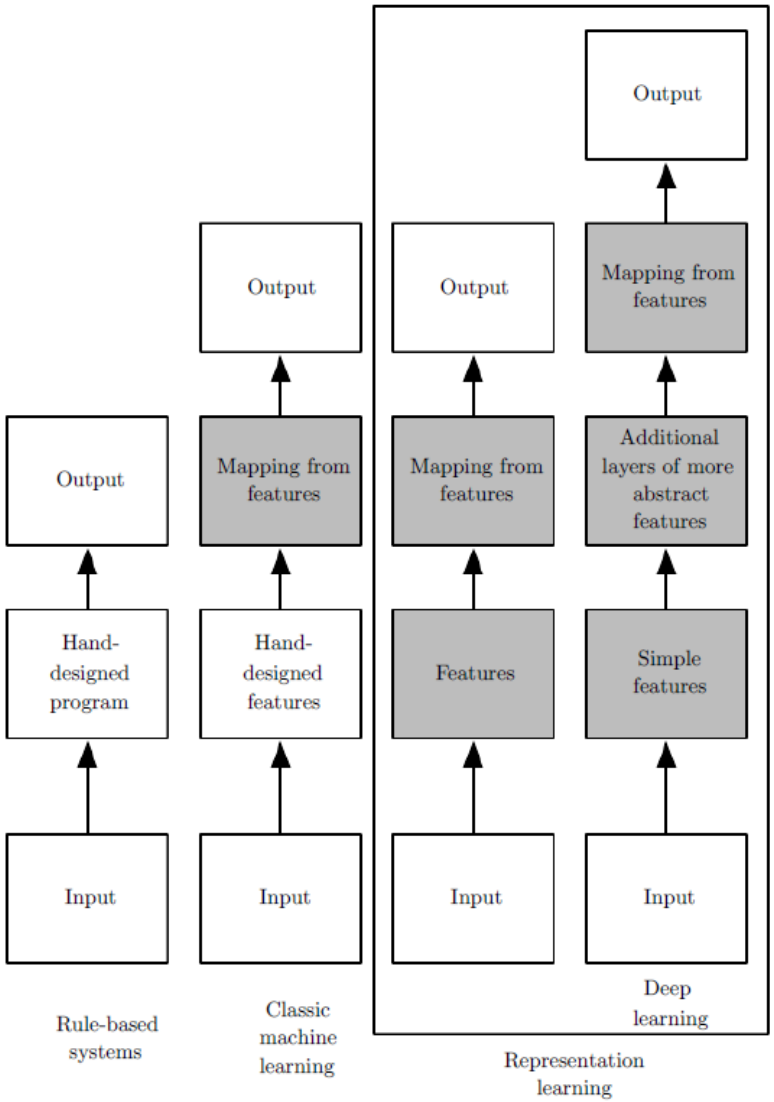
Deep Learning and Neural Networks : History

Deep learning (a subset of machine learning)



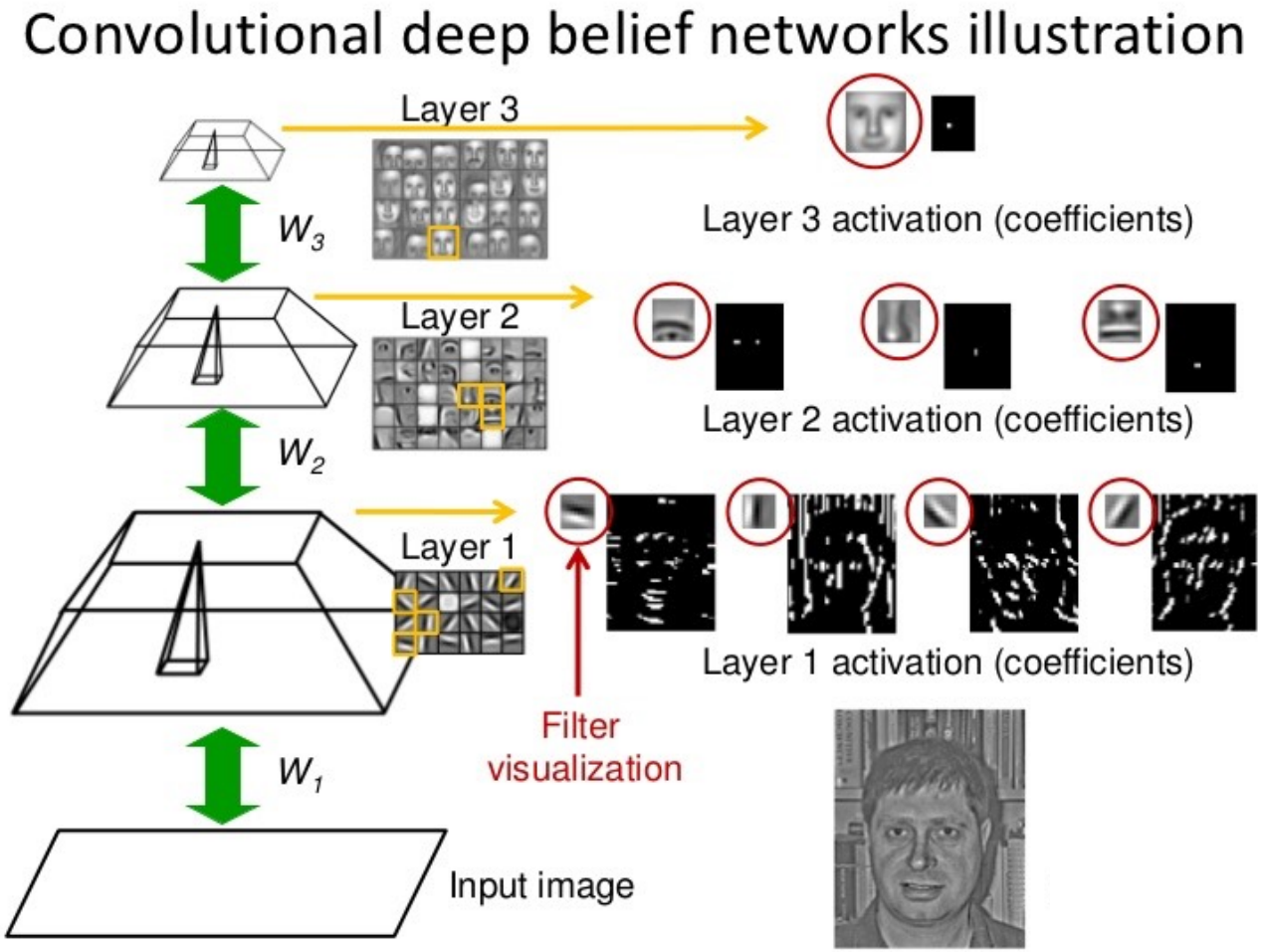
From http://canvart.club/atmose-11_21_13.html

Deep learning : learning hierarchical representations



From Deep Learning by Ian Goodfellow and Yoshua Bengio and Aaron Courville

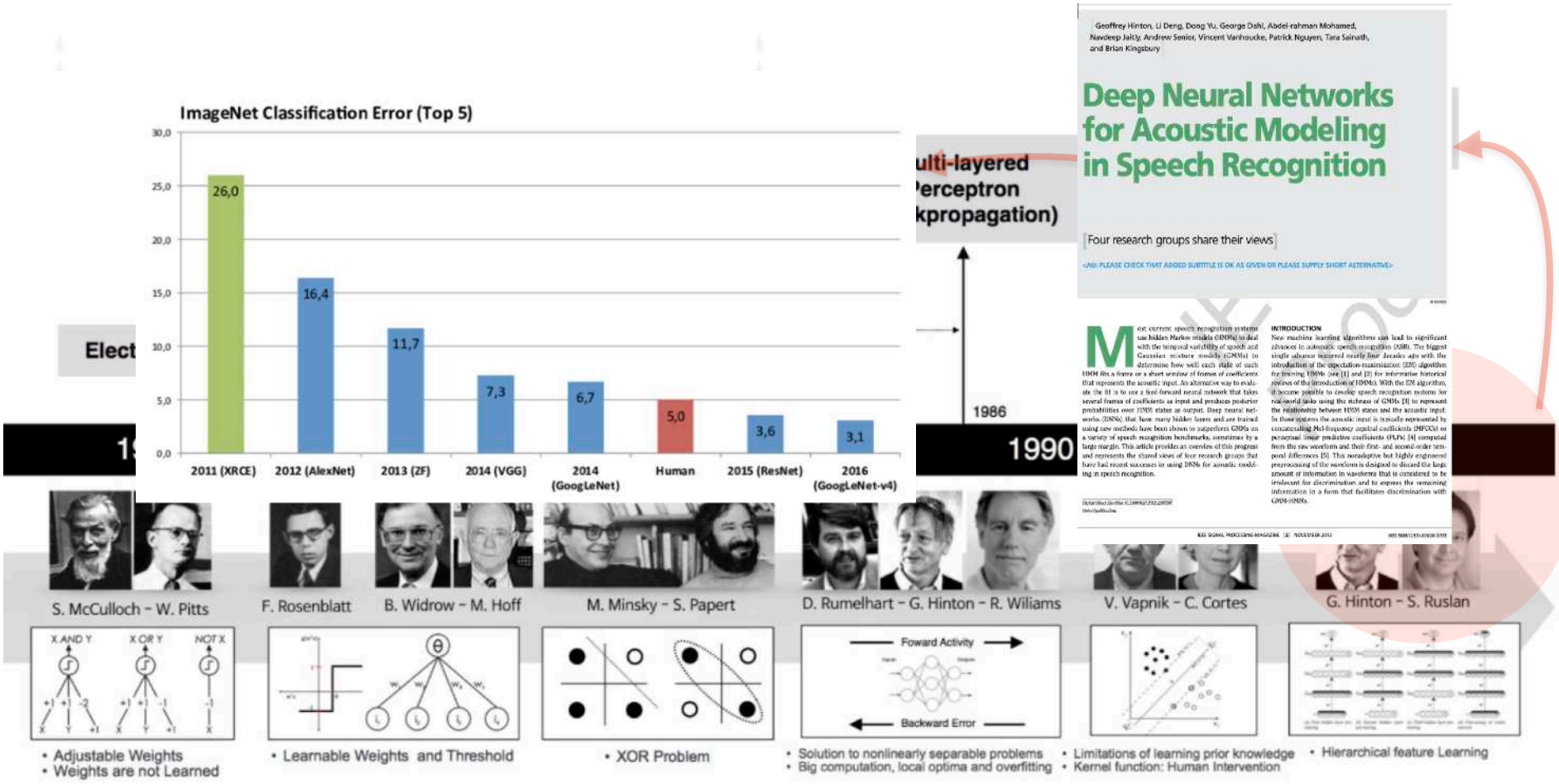
Feature learning examples



From Lee, Grosse, Ranganathn and Ng, "Convolutional Deep Belief Networks", ICML, 2009

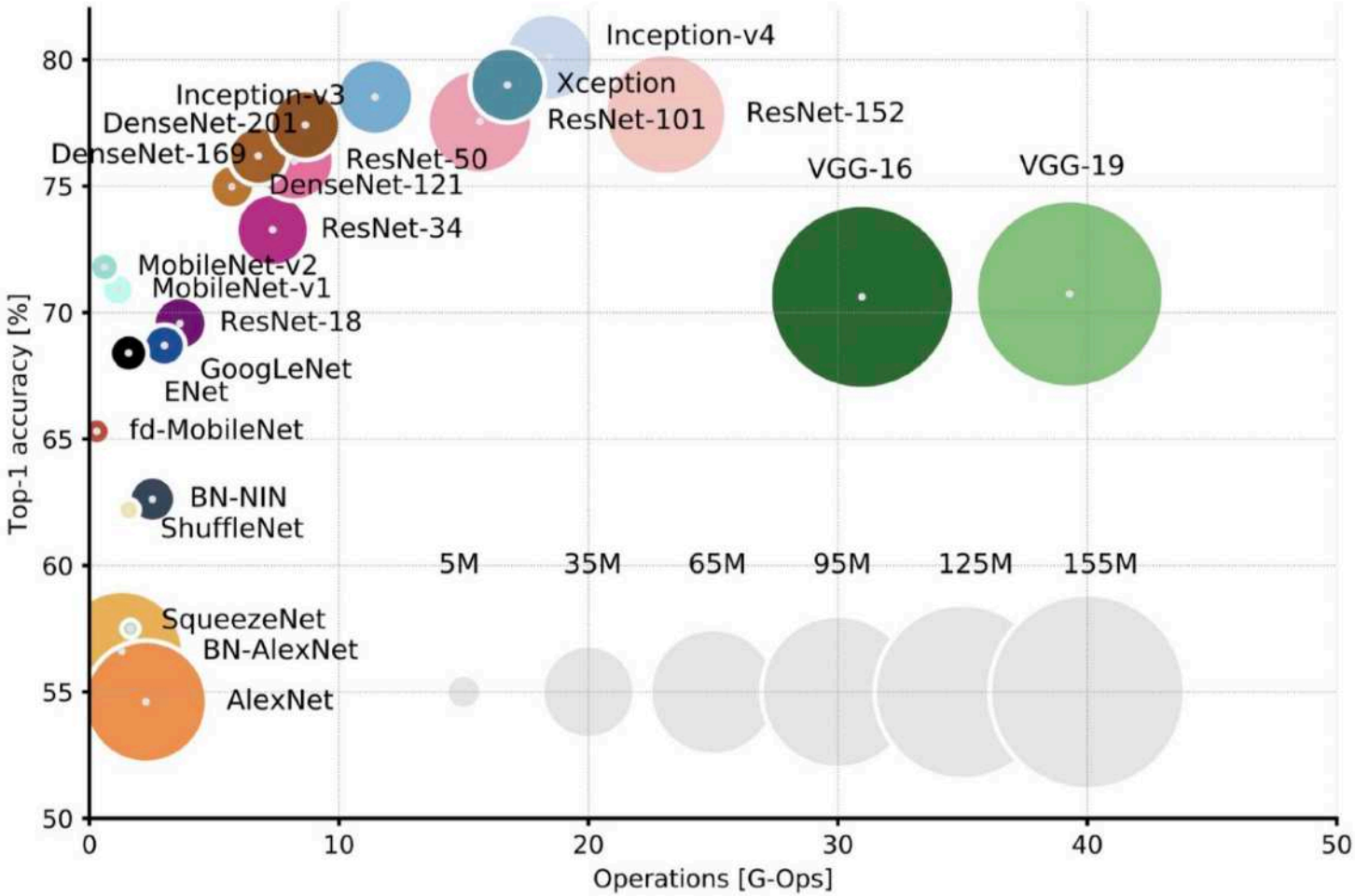
https://github.com/arthurmeyer/Convolutional_Deep_Belief_Network

Deep Learning and Neural Networks : History



https://stateofther.github.io/post/deep_learning/deep-learning-history.html#2

Deep Learning and Neural Networks : History



https://towardsdatascience.com/neural-network-architectures-156e5bad51ba

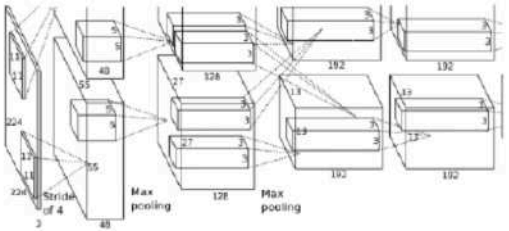
Deep Learning and Neural Networks : History

***Deep Learning* =**
Lots of training data* + *Parallel Computation* + *Scalable, smart algorithms

The Deep Learning "Computer Vision Recipe"

Big Data: ImageNet

+

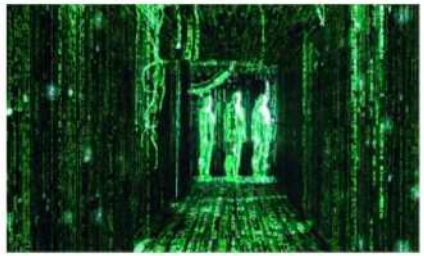


Deep Convolutional Neural Network

+

Backprop on GPU

=

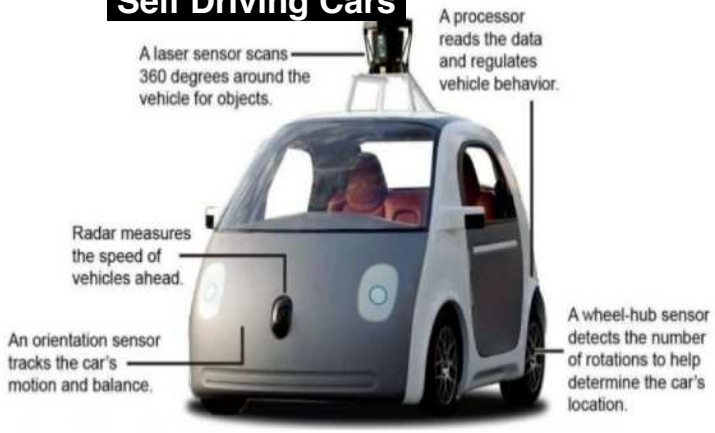


Learned Weights

Applications

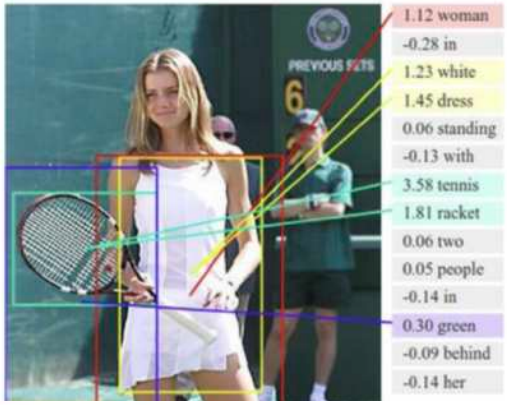
Automatic Speech Recognition

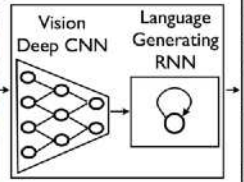
Self Driving Cars



Google Alpha Go

Automatic picture captioning

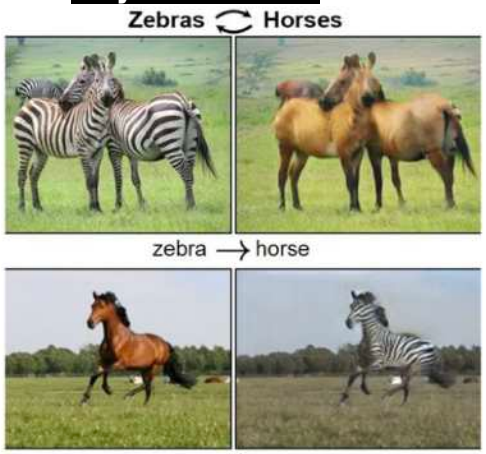


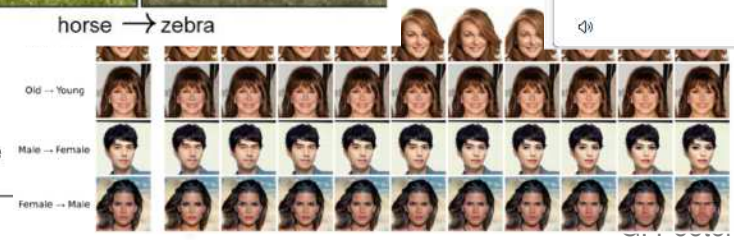


A group shopping at an outdoor market.

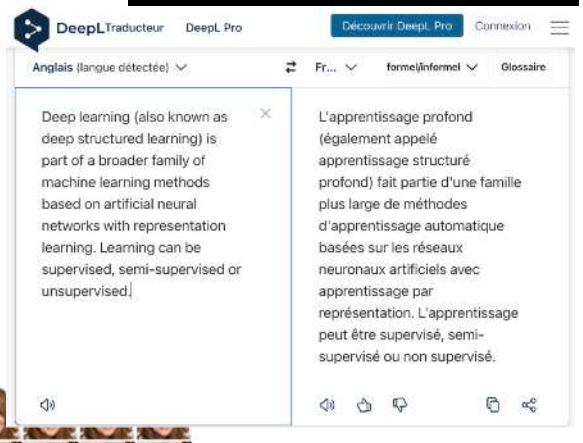
There are many vegetables at the fruit stand.

Style Transfer





Neural Machine Translation



Applications
Generative AI (GenAI)

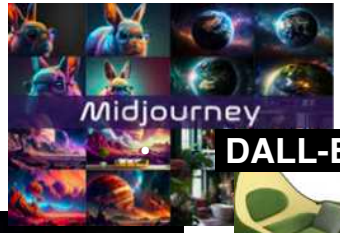
NLP

Video

https://www.youtube.com/watch?v=8LjfxFQ5iNk

Music

Computer Vision



DALL-E

Speech (deep fake)

Voice dubbing

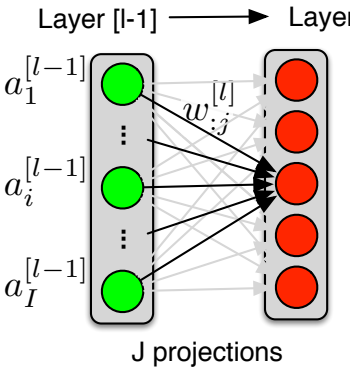
Roadmap

Roadmap

Architectures: *distinguish the way neurons are inter-connected*

Lecture 1

Fully Connected Neural Network

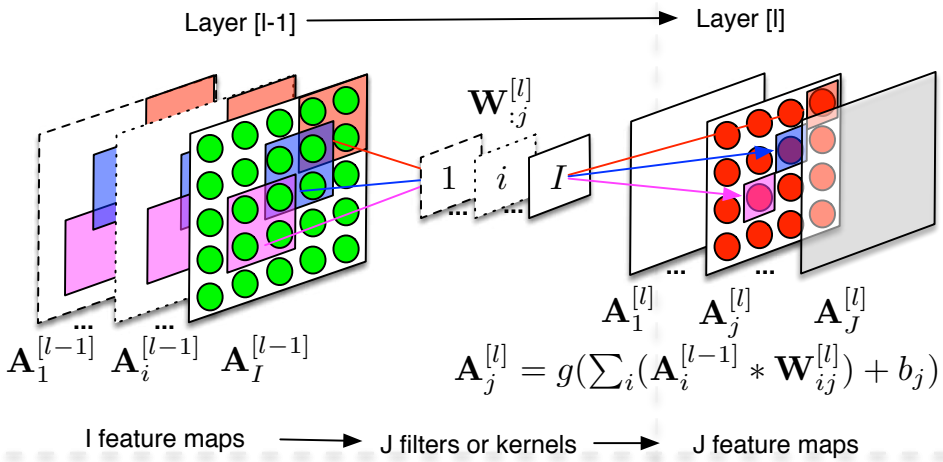


$$a_j^{[l]} = g(\sum_i a_i^{[l-1]} w_{ij}^{[l]} + b_j^{[l]})$$

$$\mathbf{a}^{[l]} = g(\mathbf{a}^{[l-1]}\mathbf{W}^{[l]} + \mathbf{b}^{[l]})$$

Lecture 2

Convolutional Neural Network



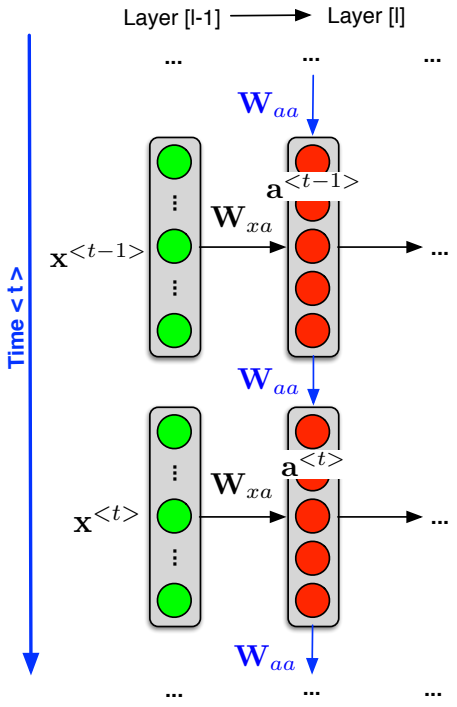
$$\mathbf{A}_j^{[l]} = g(\sum_i (\mathbf{A}_i^{[l-1]} * \mathbf{W}_{ij}^{[l]}) + b_j)$$

I feature maps → J filters or kernels → J feature maps

Lecture 4

Recurrent Neural Network

$$\mathbf{a}^{<t>} = g(\mathbf{x}^{<t>}\mathbf{W}_{xa} + \mathbf{a}^{<t-1>}\mathbf{W}_{aa} + \mathbf{b}_a)$$

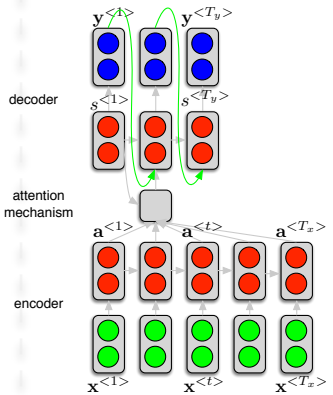


Roadmap

Meta-architectures: *the way architectures are plug to form a system*

Lecture 4

Encoder-Decoder with Attention



Transformer

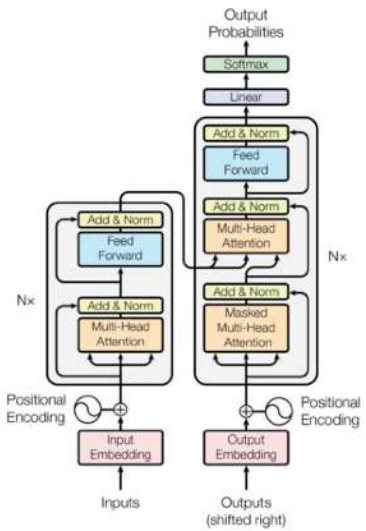
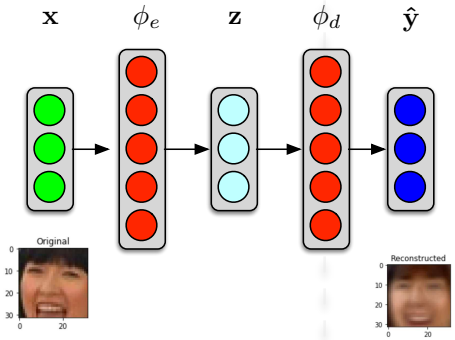


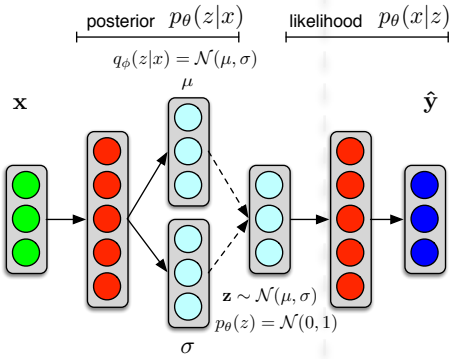
Figure 1: The Transformer - model architecture.

Lecture 3

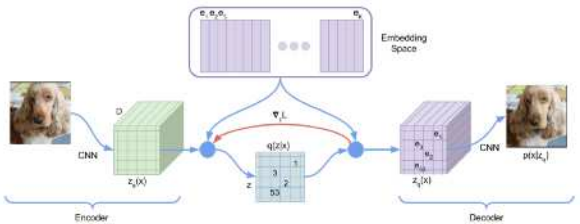
Auto-Encoder



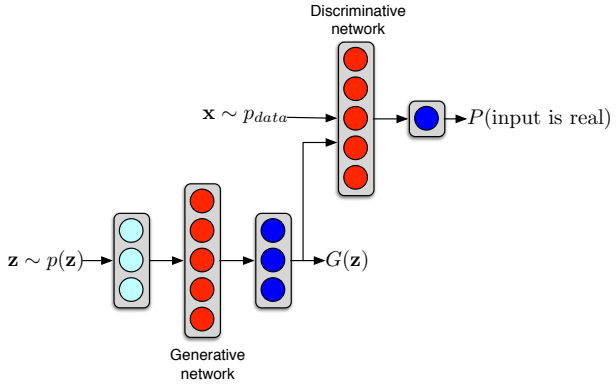
Variational-Auto-Encoder



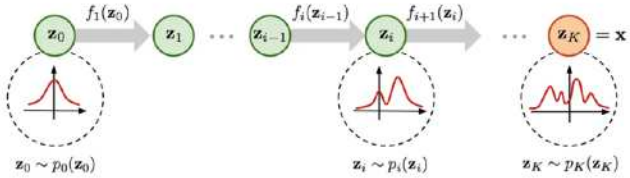
VQ-VAE



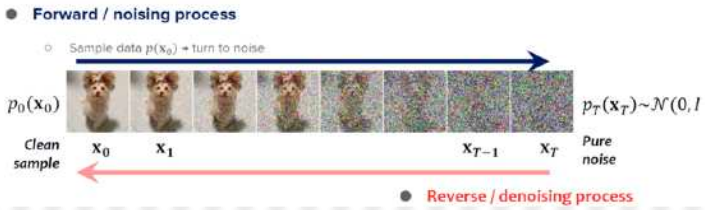
Generative Adversarial Network



Normalizing FLows

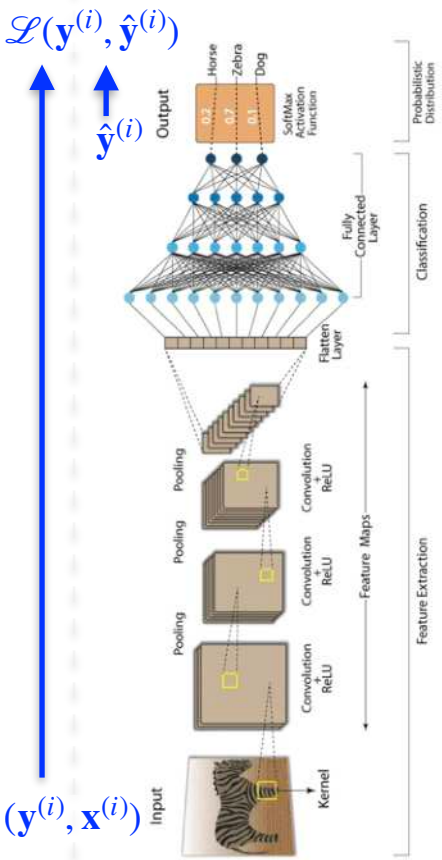


Denoising Diffusion Models



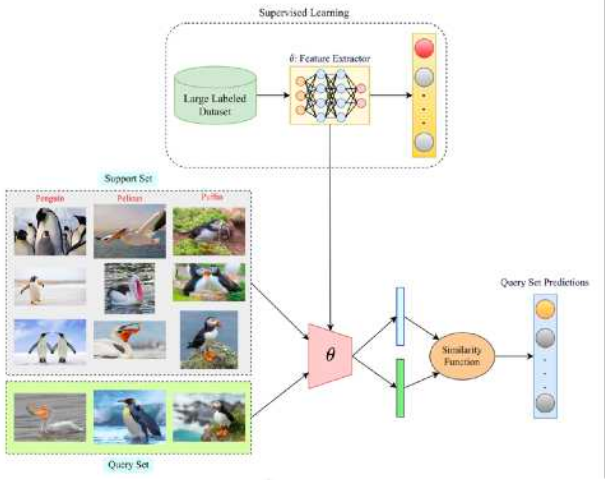
Roadmap
Training-paradigms: *the paradigm we use to train a system*

Supervised learning

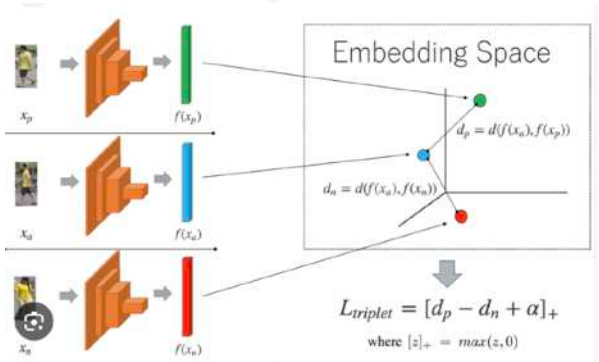


from https://larbifarihi.medium.com/classifying-cats-and-dogs-using-convolutional-neural-networks-cnns-ce6bc7bad64d

Zero/few Shot learning

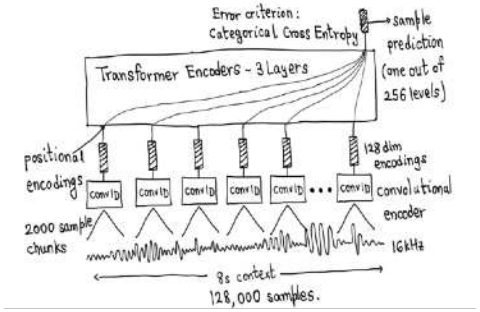


Metric learning

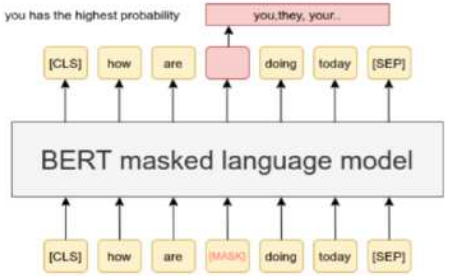


Self Supervised learning

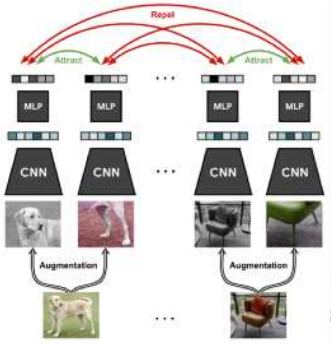
Auto-regressive (WaveNet, ChatGPT-



Masked LLM (BERT)



SimCLR



CSC_4AI07_TP
"Introduction to Deep Learning"

Lecture 1: Multi-Layer-Perceptron

Geoffroy Peeters

contact: geoffroy.peeters@telecom-paris.fr
Télécom-Paris, IP-Paris, France

Overview

Data	Supervised learning	$(x^{(i)}, y^{(i)})$	
Model	Architecture	Linear regression Non-linear regression Logistic regression Softmax regression Neural Network Deep Learning	Output activation depends on the task Avoid vanishing gradient Parameter initialization Batch Normalization
Cost/Loss	Maximize Likelihood	Gaussian (MSE) Bernoulli (BCE)	Cost depends on the task
Regularization	Avoid over-fitting: L1/L2,DropOut		
Optimizer	For convex optimization For non-convex optimization (SGD, Momentum, ...) First order (SGD), Second order (Hessian)		
Training			

Dataset/ Dataloader	Get mini-batch
Forward	Compute prediction $\hat{y}^{(i)}$
Backward (back-propagation)	Compute gradients
Update	
Early Stopping	Avoid over-fitting

Overview

Data	Supervised learning	$(x^{(i)}, y^{(i)})$	
Model	Architecture	Linear regression Non-linear regression Logistic regression Softmax regression Neural Network Deep Learning	Output activation depends on the task Avoid vanishing gradient Parameter initialization Batch Normalization
Cost/Loss	Maximize Likelihood	Gaussian (MSE) Bernoulli (BCE)	Cost depends on the task
Regularization	Avoid over-fitting: L1/L2,DropOut		
Optimizer	For convex optimization For non-convex optimization (SGD, Momentum, ...) First order (SGD), Second order (Hessian)		
Training			
Dataset/ Dataloader	Get mini-batch		
Forward	Compute prediction $\hat{y}^{(i)}$		
Backward (back-propagation)	Compute gradients		
Update			
Early Stopping	Avoid over-fitting		

Supervised classification

- **Training/Learning**
  - **Given** a set of m input/output examples $(\mathbf{x}^{(i)}, \mathbf{y}^{(i)})_{i\in\{1,\ldots,m\}}$
    - $\mathbf{x}^{(i)} \in \mathbb{R}^d$
    - $\mathbf{y}^{(i)}$
      - $y^{(i)} \in \mathbb{R}$ for regression
      - $y^{(i)} \in \{0,1\}$ for binary classification
      - $\mathbf{y}^{(i)} \in \{0,1\}^C$ for multi-class classification (C classes)
  - **Estimate** the parameters θ of a model f_θ
    - $\{\mathbf{x}^{(1:m)}, \mathbf{y}^{(1:m)}\} \rightarrow$ Learner $\rightarrow \theta$
- **Testing/ Prediction**
  - **Given** a new input $\mathbf{x}^{(m+1)}$ and the parameters θ of the model f_θ
  - **Predict** the output $\mathbf{y}^{(m+1)}$
    - $\{\mathbf{x}^{(m+1)}, \theta\} \rightarrow$ Prediction $\rightarrow \hat{\mathbf{y}}^{(m+1)}$

Overview

Data	Supervised learning	$(x^{(i)}, y^{(i)})$	
Model	Architecture	Linear regression Non-linear regression Logistic regression Softmax regression Neural Network Deep Learning	Output activation depends on the task Avoid vanishing gradient Parameter initialization Batch Normalization
Cost/Loss	Maximize Likelihood	Gaussian (MSE) Bernoulli (BCE)	Cost depends on the task
Regularization	Avoid over-fitting: L1/L2,DropOut		
Optimizer	For convex optimization For non-convex optimization (SGD, Momentum, ...) First order (SGD), Second order (Hessian)		
Training			
Dataset/ Dataloader		Get mini-batch	
Forward		Compute prediction $\hat{y}^{(i)}$	
Backward (back-propagation)		Compute gradients	
Update			
Early Stopping		Avoid over-fitting	

Model

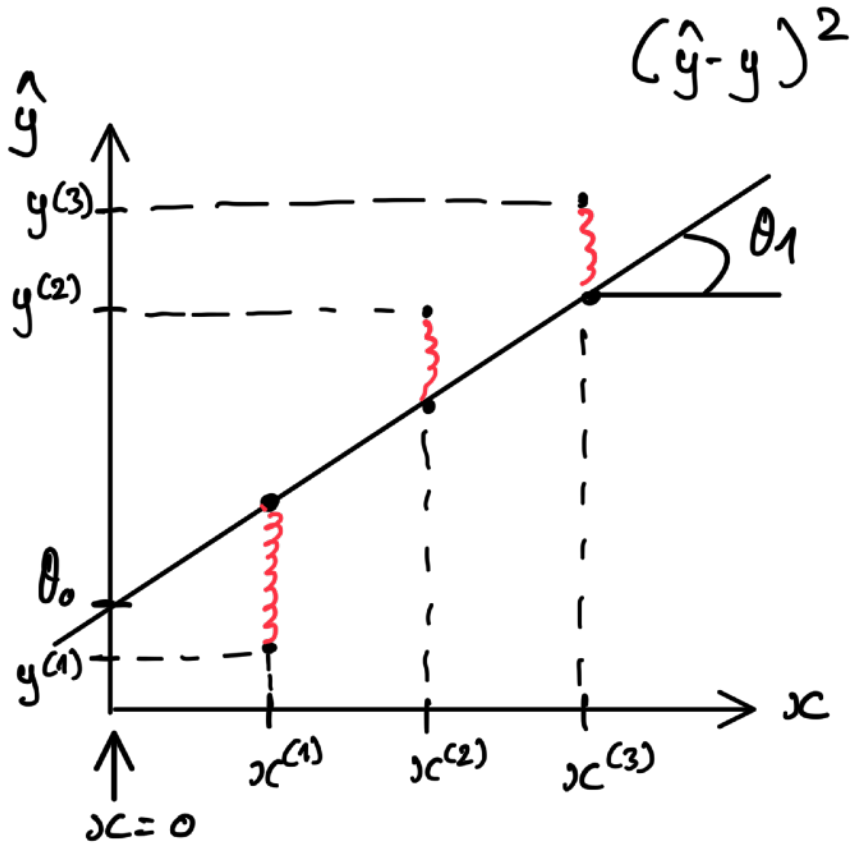
$$\hat{\mathbf{y}}^{(i)} = \theta_0 + x^{(i)}\theta_1$$

Cost function (objective function, energy, loss)

- Mean Square Error (MSE)

$$J(\theta) = \sum_{i=1}^{m} (y^{(i)} - \hat{y}^{(i)})^2$$

$$= \sum_{i=1}^{m} (y^{(i)} - (\theta_0 + x^{(i)}\theta_1))^2$$



Task: Regression
Model: Linear ... d dimension

Model

$$\mathbf{y}^{(i)} = \theta_0 + \sum_{j=1}^{d} x_j^{(i)} \theta_j$$

$$= \theta_0 + x_1^{(i)}\theta_1 + x_2^{(i)}\theta_2 + \ldots + x_d^{(i)}\theta_d$$

- If we note $x_0^{(i)} = 1$ then $\mathbf{y}^{(i)} = \sum_{j=0}^{d} x_j^{(i)} \theta_j$
- In matrix form

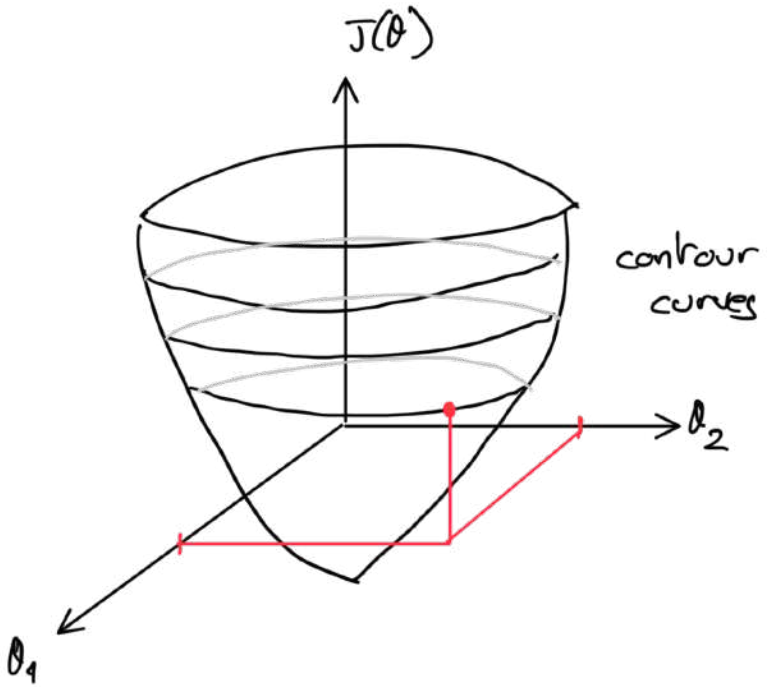
$$\underset{(\mathbb{R}^{(m,1)})}{\hat{\mathbf{y}}} = \underset{(\mathbb{R}^{(m,d+1)})}{\mathbf{X}} \quad \underset{(\mathbb{R}^{(d+1,1)})}{\theta}$$

$$\begin{bmatrix} \hat{y}^{(1)} \\ \vdots \\ \hat{y}^{(m)} \end{bmatrix} = \begin{bmatrix} x_0^{(1)} \cdots x_d^{(1)} \\ \vdots \ \ddots \ \vdots \\ x_0^{(m)} \cdots x_d^{(m)} \end{bmatrix} \begin{bmatrix} \theta_0 \\ \vdots \\ \theta_d \end{bmatrix}$$

Cost function

$$J(\theta) = \sum_{i=1}^{m} (y^{(i)} - \mathbf{x}^{(i)^T}\theta)^2$$

$$= (\mathbf{y} - \mathbf{X}\,\theta)^T(\mathbf{y} - \mathbf{X}\,\theta)$$



Overview

Data	Supervised learning	$(x^{(i)}, y^{(i)})$	
Model	Architecture	Linear regression Non-linear regression Logistic regression Softmax regression Neural Network Deep Learning	Output activation depends on the task Avoid vanishing gradient Parameter initialization Batch Normalization
Cost/Loss	Maximize Likelihood	Gaussian (MSE) Bernoulli (BCE)	Cost depends on the task
Regularization	Avoid over-fitting: L1/L2,DropOut		
Optimizer	For convex optimization For non-convex optimization (SGD, Momentum, ...) First order (SGD), Second order (Hessian)		
Training			

Dataset/ Dataloader	Get mini-batch
Forward	Compute prediction $\hat{y}^{(i)}$
Backward (back-propagation)	Compute gradients
Update	
Early Stopping	Avoid over-fitting

Optimisation

– Find θ that minimises $J(\theta)$

$$\frac{\partial J(\theta)}{\partial \theta} = 0 = \frac{\partial}{\partial \theta}(\mathbf{y} - \mathbf{X}\,\theta)^T(\mathbf{y} - \mathbf{X}\,\theta)$$

$$= \frac{\partial}{\partial \theta}(\mathbf{y}^T\mathbf{y} - 2\mathbf{y}^T\mathbf{X}\,\theta + \theta^T\mathbf{X}^T\mathbf{X}\,\theta)$$

$$= 0 - 2\mathbf{X}^T\mathbf{y} + 2\mathbf{X}^T\mathbf{X}\theta$$

$$\theta = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}$$

$$\underbrace{\underbrace{[(m,d)^T(m,d)]}_{(d,d)}(m,d)^T(m)}_{d,m}$$

Results from matrix differentiation

$$\frac{\partial A\theta}{\partial \theta} = A^T$$

$$\frac{\partial \theta^T A\theta}{\partial \theta} = 2A^T\theta$$

Overview

Data	Supervised learning	$(x^{(i)}, y^{(i)})$	
Model	Architecture	Linear regression Non-linear regression Logistic regression Softmax regression Neural Network Deep Learning	Output activation depends on the task Avoid vanishing gradient Parameter initialization Batch Normalization
Cost/Loss	Maximize Likelhood	Gaussian (MSE) Bernoulli (BCE)	Cost depends on the task
Regularization	Avoid over-fitting: L1/L2,DropOut		
Optimizer	For convex optimization For non-convex optimization (SGD, Momentum, ...) First order (SGD), Second order (Hessian)		
Training			
Dataset/ Dataloader		Get mini-batch	
Forward		Compute prediction $\hat{y}^{(i)}$	
Backward (back-propagation)		Compute gradients	
Update			
Early Stopping		Avoid over-fitting	

Maximum Likelihood Estimation (MLE)
Likelihood for linear regression

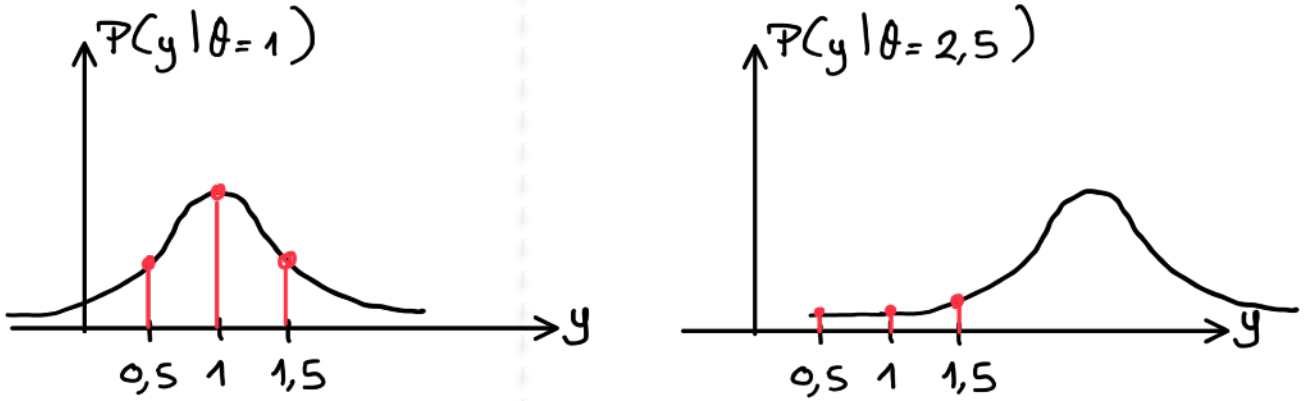
- Suppose we have 3 observations
  - $y^{(1)} = 1, y^{(2)} = 0.5, y^{(3)} = 1.5$
  - which are independent samples from a Gaussian with unknown mean θ and variance 1

$$y^{(i)} \sim \mathcal{N}(\theta,1) = \theta + \mathcal{N}(0,1)$$

- The likelihood of the observations $y^{(i)}$ is

$$p(y^{(1)}, y^{(2)}, y^{(3)}) | \theta) = p(y^{(1)} | \theta) \cdot p(y^{(2)} | \theta) \cdot p(y^{(3)} | \theta)$$

- Which guess of θ is more likely ?
  - $\theta = 1$
  - $\theta = 2.5$

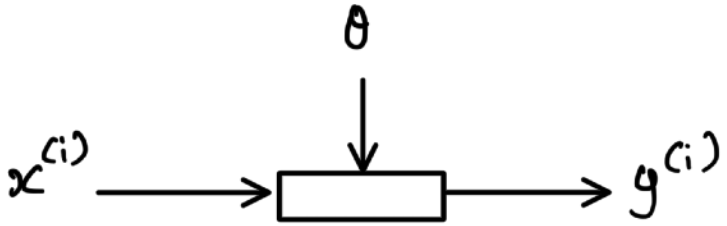


- We take the θ that maximises the likelihood

Maximum Likelihood Estimation (MLE)
Likelihood for linear regression

- Suppose $y^{(i)}$ is Gaussian distributed with mean $\mu = \mathbf{x}^{(i)^T}\theta$ and variance σ^2

$$y^{(i)} \sim \mathcal{N}(\mu = \mathbf{x}^{(i)^T}\theta, \sigma^2)$$
$$= \mathbf{x}^{(i)^T}\theta + \mathcal{N}(0,\sigma^2)$$



- The likelihood is

$$p(\mathbf{y}\,|\,\mathbf{X},\theta,\sigma) = \prod_{i=1}^{m} p(y^{(i)}\,|\,\mathbf{x}^{(i)},\theta,\sigma)$$
$$= \prod_{i=1}^{m} (2\pi\sigma)^{-1/2} e^{-\frac{1}{2\sigma^2}(y^{(i)}-\mathbf{x}^{(i)^T}\theta)^2}$$
$$= (2\pi\sigma)^{-m/2} e^{-\frac{1}{2\sigma^2}\sum_{i=1}^{m}(y^{(i)}-\mathbf{x}^{(i)^T}\theta)^2}$$
$$= (2\pi\sigma)^{-m/2} e^{-\frac{1}{2\sigma^2}(\mathbf{y}-\mathbf{X}\,\theta)^T(\mathbf{y}-\mathbf{X}\,\theta)}$$

- This is equivalent to $p(\text{data}\,|\,\text{parameters}) = \frac{1}{Z} e^{-Loss(\text{data},\text{parameters})}$
  - with $p(\text{data}\,|\,\text{parameters}) = J(\theta) = \sum_{i=1}^{m} (y^{(i)} - \mathbf{x}^{(i)^T}\theta)^2$
  - is the MSE (Mean Square Error) or Linear Regression
  - **Maximising the likelihood is equivalent to minimising the MSE Loss !**

Maximum Likelihood Estimation (MLE)
Likelihood for linear regression

- Maximising the **Likelihood** ⇒ minimising the **Negative Log-Likelihood** (NLL)

$$p(\mathbf{y}\,|\,\mathbf{X},\theta,\sigma) = (2\pi\sigma^2)^{-m/2} e^{-\frac{1}{2\sigma^2}(\mathbf{y}-\mathbf{X}\,\theta)^T(\mathbf{y}-\mathbf{X}\,\theta)}$$

$$-\log p(\mathbf{y}\,|\,\mathbf{X},\theta,\sigma) = \frac{m}{2}\log(2\pi\sigma^2) + \frac{1}{2\sigma^2}(\mathbf{y}-\mathbf{X}\,\theta)^T(\mathbf{y}-\mathbf{X}\,\theta)$$

Optimisation

– MLE of θ (same solution as MSE)

$$\theta_{ML} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}$$

Proof

$$0 = \frac{\partial}{\partial\theta}\left(\frac{1}{2\sigma^2}(\mathbf{y}-\mathbf{X}\,\theta)^T(\mathbf{y}-\mathbf{X}\,\theta)\right)$$

– MLE of σ ⇒ *we also get a confidence in the prediction !*

$$\sigma^2_{ML} = \frac{1}{m}(\mathbf{y}-\mathbf{X}\,\theta)^T(\mathbf{y}-\mathbf{X}\,\theta)$$
$$= \frac{1}{m}\sum_{i=1}^{n}(\mathbf{y}^{(i)} - \mathbf{x}^{(i)T}\theta)^2$$

Proof

$$0 = \frac{\partial}{\partial\sigma}\left(\frac{m}{2}\log(2\pi\sigma^2) + \frac{1}{2\sigma^2}(\mathbf{y}-\mathbf{X}\,\theta)^T(\mathbf{y}-\mathbf{X}\,\theta)\right)$$
$$= m\frac{1}{\sigma} - \frac{1}{\sigma^3}(\mathbf{y}-\mathbf{X}\,\theta)^T(\mathbf{y}-\mathbf{X}\,\theta)$$

Overview

Data	Supervised learning	$(x^{(i)}, y^{(i)})$	
Model	Architecture	Linear regression Non-linear regression Logistic regression Softmax regression Neural Network Deep Learning	Output activation depends on the task Avoid vanishing gradient Parameter initialization Batch Normalization
Cost/Loss	Maximize Likelhood	Gaussian (MSE) Bernoulli (BCE)	Cost depends on the task
Regularization	Avoid over-fitting: L1/L2,DropOut		
Optimizer	For convex optimization For non-convex optimization (SGD, Momentum, ...) First order (SGD), Second order (Hessian)		
Training			
Dataset/ Dataloader		Get mini-batch	
Forward		Compute prediction $\hat{y}^{(i)}$	
Backward (back-propagation)		Compute gradients	
Update			
Early Stopping		Avoid over-fitting	

Regularisation : Ridge, Lasso

Least-square regression

$$J(\theta) = (\mathbf{y} - \mathbf{X}\theta)^T(\mathbf{y} - \mathbf{X}\theta)$$

$$\hat{\theta} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}$$

- inversion of $(\mathbf{X}^T\mathbf{X})$ can lead to problems (if system poorly conditioned)
- add a small constant δ to the diagonal elements

Ridge regression

$$\hat{\theta} = (\mathbf{X}^T\mathbf{X} + \delta^2\mathbb{I})^{-1}\mathbf{X}^T\mathbf{y} \text{ with } \mathbb{I} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

- Corresponds to the regularised quadratic cost function

$$J(\theta) = (\mathbf{y} - \mathbf{X}\,\theta)^T(\mathbf{y} - \mathbf{X}\,\theta) + \delta^2\theta^T\theta$$

- equivalent to $\min_{\theta:\theta^T\theta \leq t(\delta)} (\mathbf{y} - \mathbf{X}\theta)^T(\mathbf{y} - \mathbf{X}\theta)$
  - where t is an arbitrary function
  - also named weight decay

Proof

$$\begin{aligned} 0 &= \frac{\partial}{\partial\theta}(\theta^T\mathbf{X}^T\mathbf{X}\theta - 2\mathbf{y}^T\mathbf{X}\theta + \mathbf{y}^T\mathbf{y} + \delta^2\theta^T\theta) \\ &= 2\mathbf{X}^T\mathbf{X}\theta - 2\mathbf{X}^T\mathbf{y} + 2\delta^2\mathbb{I}\theta \\ &= 2(\mathbf{X}^T\mathbf{X} + \delta^2\mathbb{I})\theta - 2\mathbf{X}^T\mathbf{y} \\ \hat{\theta}_{ridge} &= (\mathbf{X}^T\mathbf{X} + \delta^2\mathbb{I})^{-1}\mathbf{X}^T\mathbf{y} \end{aligned}$$

Regularisation : Ridge, Lasso

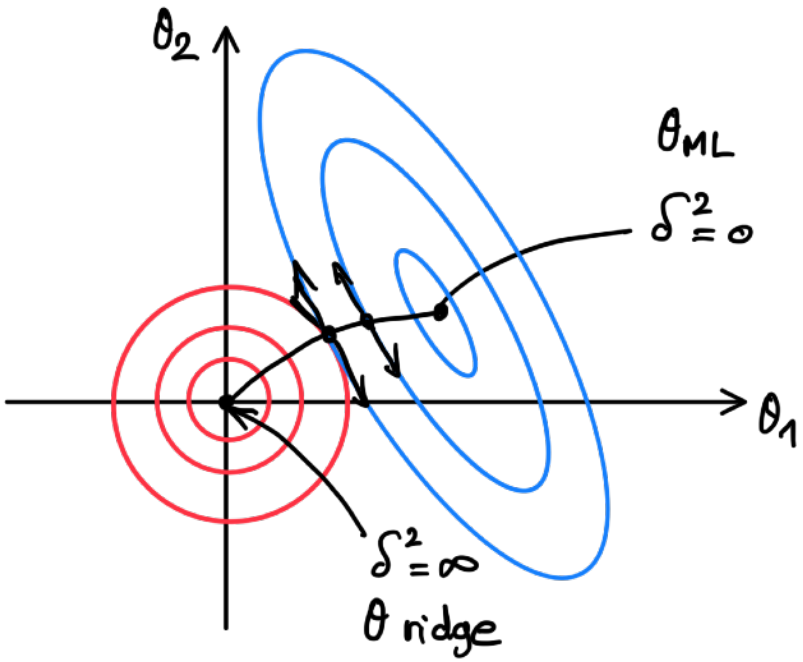
Example

$$\theta^T = [\theta_1, \theta_2]$$

$$\rightarrow \theta^T\theta = [\theta_1, \theta_2]\begin{bmatrix}\theta_1 \\ \theta_2\end{bmatrix}$$

$$= \theta_1^2 + \theta_2^2 = \textbf{const}$$

- equation of a circle
- automatically remove irrelevant axes



Regularisation : Ridge, Lasso

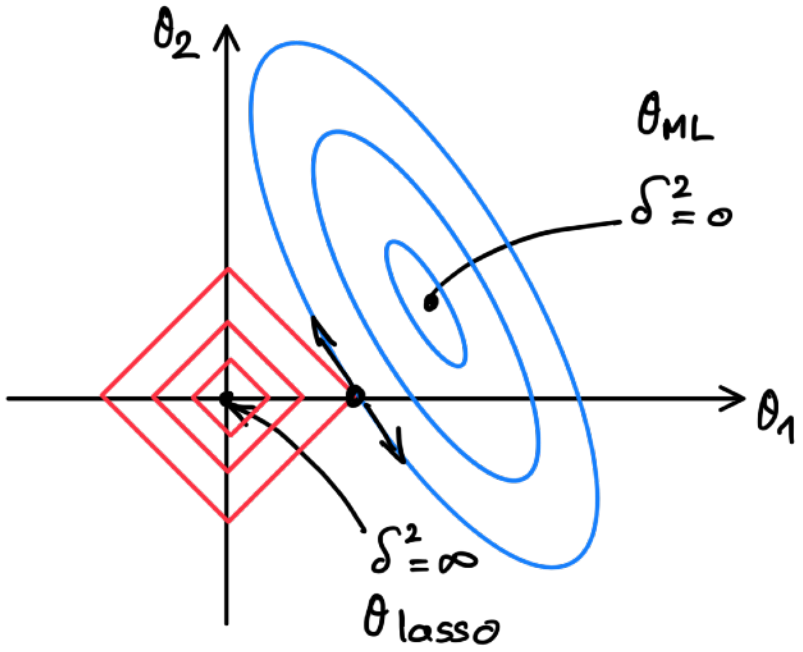
Ridge

$$\|\theta\|_2^2 = \theta^T\theta$$

Lasso

$$\|\theta\|_1 = \sum_d |\theta_d| = \theta_1 + \theta_2$$

- intersections happen at corners ($\theta_2 = 0$)
- sparse, compress-sensing



Overview

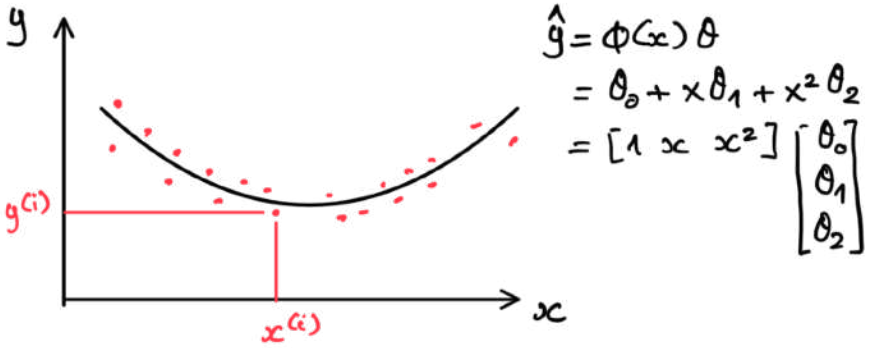
Data	Supervised learning	$(x^{(i)}, y^{(i)})$	
Model	Architecture	Linear regression Non-linear regression Logistic regression Softmax regression Neural Network Deep Learning	Output activation depends on the task Avoid vanishing gradient Parameter initialization Batch Normalization
Cost/Loss	Maximize Likelhood	Gaussian (MSE) Bernoulli (BCE)	Cost depends on the task
Regularization	Avoid over-fitting: L1/L2,DropOut		
Optimizer	For convex optimization For non-convex optimization (SGD, Momentum, ...) First order (SGD), Second order (Hessian)		
Training			
Dataset/ Dataloader		Get mini-batch	
Forward		Compute prediction $\hat{y}^{(i)}$	
Backward (back-propagation)		Compute gradients	
Update			
Early Stopping		Avoid over-fitting	

Task: Regression
Model: Non-linear (basis, kernel)

Model

$\hat{y}(\mathbf{x}) = \phi(\mathbf{x})\,\theta + \epsilon$

with $\phi(\,.\,)$ are basis functions



– **Examples:**

- in 1 dimensions: $\phi(x) = [1, x, x^2]$

$$\hat{y} = \phi(x)\,\theta = \theta_0 + x\theta_1 + x^2\theta_2 = [1 \;\; x \;\; x^2] \begin{bmatrix} \theta_0 \\ \theta_1 \\ \theta_2 \end{bmatrix}$$

- in 2 dimensions: $\phi(\mathbf{x}) = [1, x_1, x_2, x_1^2, x_2^2, x_1 x_2]$

Optimisation

$$\hat{\theta} = (\mathbf{X}^T\mathbf{X} + \delta^2\mathbb{I})^{-1}\mathbf{X}^T\mathbf{y} \quad \rightarrow \quad \hat{\theta} = (\phi^T\phi + \delta^2\mathbb{I})^{-1}\phi^T\mathbf{y}$$

Example of basis functions: RBF (Radial Basis Function) kernel

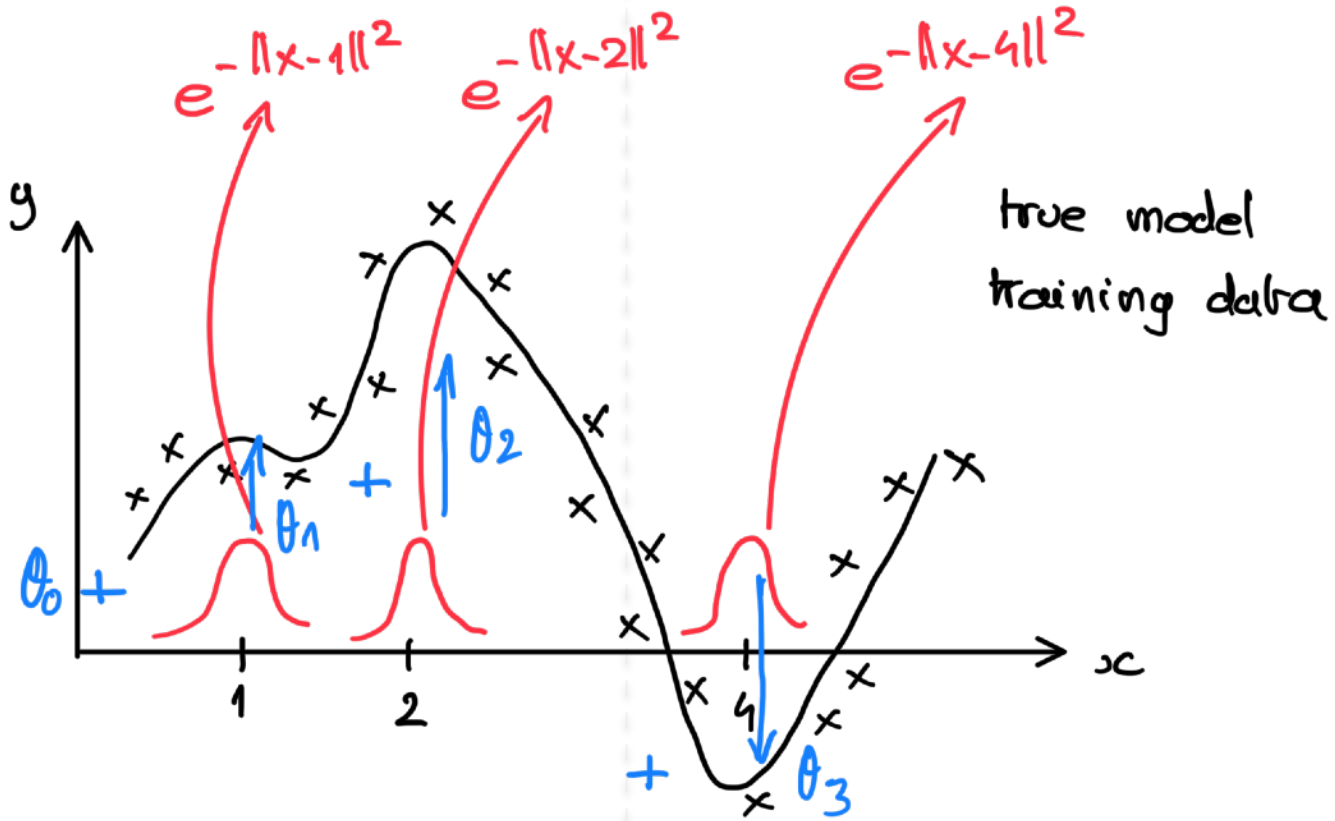
$$\phi(\mathbf{x}) = [k(\mathbf{x}, \mu_1, \lambda), \ldots, k(\mathbf{x}, \mu_d, \lambda)]$$

- with $k(\mathbf{x}, \mu_d, \lambda) = e^{-\frac{1}{\lambda}||\mathbf{x}-\mu_d||^2}$

$$\hat{y}(\mathbf{x}^{(i)}) = \phi(\mathbf{x}^{(i)}) \ \ \theta$$
$$= 1 \cdot \theta_0 + k(\mathbf{x}^{(i)}, \mu_1, \lambda) \cdot \theta_1 + \ldots + k(\mathbf{x}^{(i)}, \mu_d, \lambda) \cdot \theta_d$$

Example of basis functions: RBF (Radial Basis Function) kernel

- if $\lambda = 1$
- then $\hat{y}(x) = \theta_0 + e^{-\|x-1\|^2}\theta_1 + e^{-\|x-2\|^2}\theta_2 + e^{-\|x-4\|^2}\theta_3$



Optimisation

- We note $\phi(x^{(i)})$ the 4-dimensional (3 bases):
- Basis function:

$$\phi(\mathbf{x}^{(i)}) = [1, k(\mathbf{x}^{(i)}, \mu_1, \lambda), k(\mathbf{x}^{(i)}, \mu_2, \lambda), k(\mathbf{x}, \mu_3, \lambda)]$$

- For m data

$$\hat{\mathbf{y}} = \mathbf{\Phi}\,\theta$$

$$\hat{\theta}_{LS} = (\mathbf{\Phi}^T\mathbf{\Phi})^{-1}\mathbf{\Phi}^T\mathbf{y}$$

$$\hat{\theta}_{Ridge} = (\mathbf{\Phi}^T\mathbf{\Phi} + \delta^2\mathbb{I})^{-1}\mathbf{\Phi}^T\mathbf{y}$$

with $\mathbf{y} = \begin{bmatrix} y^{(1)} \\ \vdots \\ y^{(m)} \end{bmatrix}, \mathbf{\Phi} = \begin{bmatrix} \phi(x^{(1)}) \\ \vdots \\ \phi(x^{(m)}) \end{bmatrix}, \theta = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \theta_2 \\ \theta_3 \end{bmatrix}$

- ⇒It is still a regression !

- **Specific case (SVM-like):**
  - choose the μ_i equal to the training data point $\mathbf{x}^{(i)}$
  - ⇒ one kernel on each training data point

Overview

Data	Supervised learning	$(x^{(i)}, y^{(i)})$	
Model	Architecture	Linear regression Non-linear regression Logistic regression Softmax regression Neural Network Deep Learning	Output activation depends on the task Avoid vanishing gradient Parameter initialization Batch Normalization
Cost/Loss	Maximize Likelihood	Gaussian (MSE) Bernoulli (BCE)	Cost depends on the task
Regularization	Avoid over-fitting: L1/L2,DropOut		
Optimizer	For convex optimization For non-convex optimization (SGD, Momentum, ...) First order (SGD), Second order (Hessian)		
Training			
Dataset/ Dataloader		Get mini-batch	
Forward		Compute prediction $\hat{y}^{(i)}$	
Backward (back-propagation)		Compute gradients	
Update			
Early Stopping		Avoid over-fitting	

Optimisation

- **Gradient**

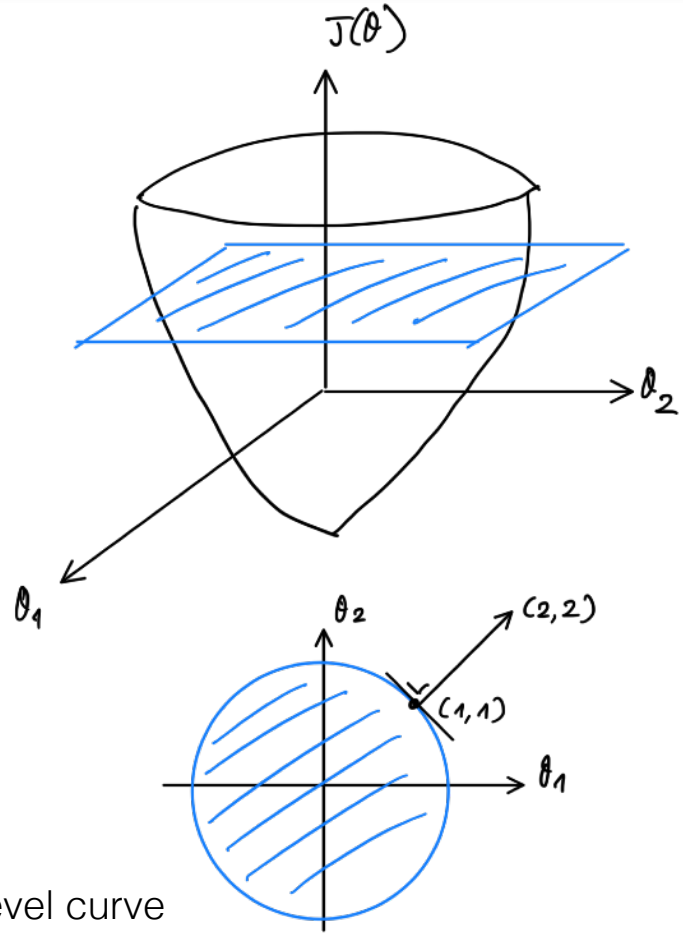
$$\frac{\partial f(\theta_1,\theta_2)}{\partial \theta_1} = \lim_{\Delta\theta_1 \to 0} \frac{f(\theta_1+\Delta\theta_1,\theta_2) - f(\theta_1,\theta_2)}{\Delta\theta_1}$$

- **Example**: consider the function $f(\theta_1,\theta_2) = \theta_1^2 + \theta_2^2$

$$\frac{\partial f}{\partial \theta_1} = 2\theta_1$$

$$\frac{\partial f}{\partial \theta_2} = 2\theta_2$$

$$\nabla f(\theta) = \begin{bmatrix} 2\theta_1 \\ 2\theta_2 \end{bmatrix}$$



- **Properties**:
  - the gradient is always orthogonal to the tangent of the level curve
  - the gradient indicates the direction of
    - the maximum change,
    - the steepest ascent

Optimisation

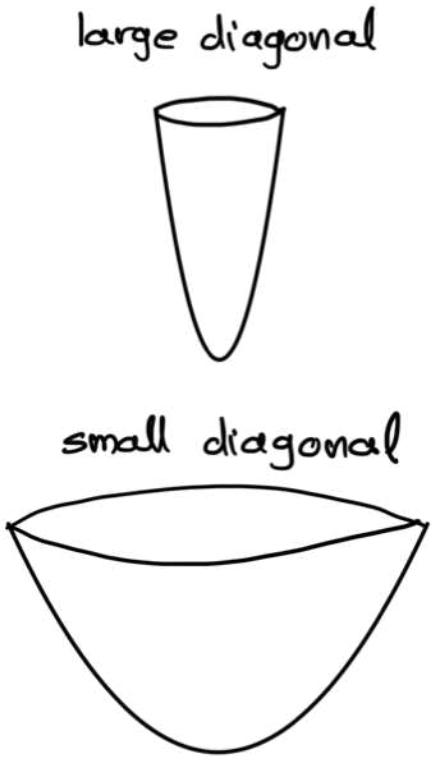
- **Hessian**

$$\frac{\partial}{\partial\theta_1}\left(\frac{\partial f(\theta)}{\partial\theta_1}\right) = \frac{\partial^2 f}{\partial\theta_1^2} = 2$$

$$\frac{\partial^2 f}{\partial\theta_2^2} = 2$$

$$\frac{\partial^2 f}{\partial\theta_1\theta_2} = \frac{\partial^2 f}{\partial\theta_2\theta_1} = 0$$

$$H = \begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix}$$



- **Properties**:
  - the Hessian represents the curvature,
  - the Hessian allows to check (local) convexity (saddle point)
    - Convexity
      - $f(\lambda x + (1-\lambda)y) \leq \lambda f(x) + (1-\lambda)f(y) \quad \forall x, y \in \mathbb{R}^n, \lambda \in [0,1]$
    - Convex functions have a (local) minimum
  - if a minimum (local or global) exists, eigen values are all positive

Optimisation

- **Gradient:** $\mathbb{R}^d \rightarrow \mathbb{R}$

$$\nabla_\theta f(\theta) = \begin{bmatrix} \frac{\partial f(\theta)}{\partial \theta_1} \\ \vdots \\ \frac{\partial f(\theta)}{\partial \theta_d} \end{bmatrix}$$

- **Hessian:** $\mathbb{R}^d \rightarrow \mathbb{R}$

$$\nabla^2_\theta f(\theta) = \begin{bmatrix} \frac{\partial^2 f(\theta)}{\partial \theta_1^2} & \cdots & \frac{\partial f(\theta)}{\partial \theta_1 \theta_d} \\ \vdots & \ddots & \vdots \\ \frac{\partial^2 f(\theta)}{\partial \theta_d \theta_1} & \cdots & \frac{\partial f(\theta)}{\partial \theta_d^2} \end{bmatrix}$$

How to estimate then ?

$$\text{function } f(\theta) = f(\theta, x^{(1:m)}) = \frac{1}{m}\sum_{i=1}^{m} f(\theta, x^{(i)})$$

$$\text{gradient } \mathbf{g}(\theta) = \nabla_\theta f(\theta) = \frac{1}{m}\sum_{i=1}^{m} \nabla_\theta f(\theta, x^{(i)})$$

$$\text{hessian } \mathbf{H}(\theta) = \nabla^2_\theta f(\theta) = \frac{1}{m}\sum_{i=1}^{m} \nabla^2_\theta f(\theta, x^{(i)})$$

- **Jacobian:** $\mathbb{R}^d \rightarrow \mathbb{R}^{d'}$

$$J(\theta) = \begin{bmatrix} \frac{\partial f_1(\theta)}{\partial \theta_1} & \cdots & \frac{\partial f_1(\theta)}{\partial \theta_d} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_{d'}(\theta)}{\partial \theta_1} & \cdots & \frac{\partial f_{d'}(\theta)}{\partial \theta_d} \end{bmatrix}$$

Optimisation
Gradient and Hessian for linear regression

- Linear regression: $y^{(i)} = \theta_0 + x^{(i)}\theta_1$

- **Scalar form** (applying the chain rule):

$$f(\theta) = \sum_i \left(y^{(i)} - (\theta_0 + x^{(i)}\theta_1)\right)^2 + \delta^2\theta_1^2$$

$$\nabla_\theta f = \begin{bmatrix} \frac{\partial}{\partial\theta_0} \\ \frac{\partial}{\partial\theta_1} \end{bmatrix}$$

$$= \begin{bmatrix} \sum_i 2(y^{(i)} - (\theta_0 + x^{(i)}\theta_1))(-1) \\ \sum_i 2(y^{(i)} - (\theta_0 + x^{(i)}\theta_1))(-x^{(i)}) + 2\delta^2\theta_1 \end{bmatrix}$$

$$= \begin{bmatrix} 2\sum_i 1 \cdot \left((\theta_0 + x^{(i)}\theta_1) - y^{(i)}\right) \\ 2\sum_i x^{(i)} \cdot \left((\theta_0 + x^{(i)}\theta_1) - y^{(i)}\right) + 2\delta^2\theta_1 \end{bmatrix}$$

- **Matrix form**

$$f(\theta) = (\mathbf{y} - \mathbf{X}\,\theta)^T(\mathbf{y} - \mathbf{X}\,\theta)$$

$$\nabla_\theta f(\theta) = \frac{\partial}{\partial\theta}(\mathbf{y}^T\mathbf{y} - 2\mathbf{y}^T\mathbf{X}\,\theta + \theta^T\mathbf{X}^T\mathbf{X}\,\theta)$$

$$= -2\mathbf{X}^T\mathbf{y} + 2\mathbf{X}^T\mathbf{X}\,\theta$$

$$= 2\mathbf{X}^T(\mathbf{X}\,\theta - \mathbf{y})$$

$$\nabla^2_\theta f(\theta) = 0 + 2\mathbf{X}^T\mathbf{X}$$

$$= 2\underset{(d,m)}{\mathbf{X}^T}\underset{(m,d)}{\mathbf{X}}$$

Optimisation
1) Steepest/Stochastic Gradient Descent (SGD) algorithm

- Notation:
  - θ_k value of θ at iteration k

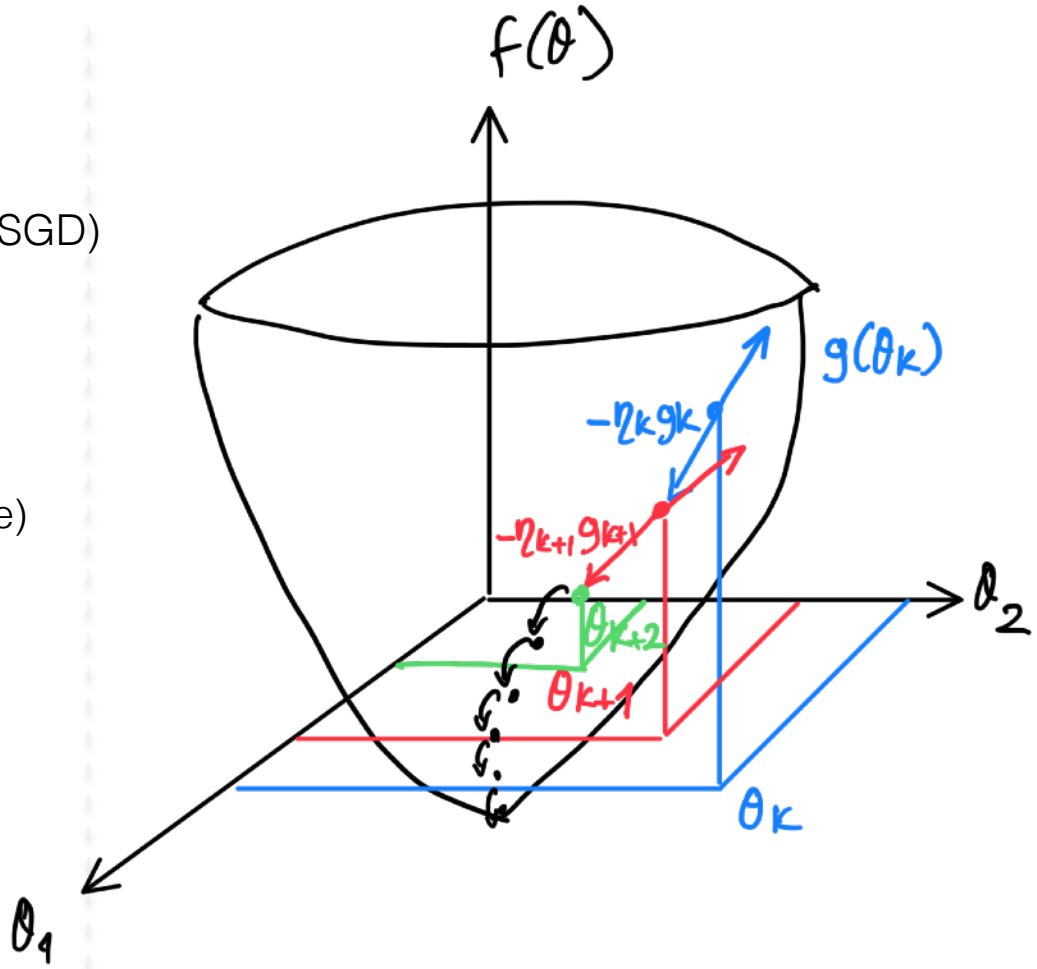
- Steepest/Stochastic Gradient Descent (SGD) algorithm

$$\theta_{k+1} = \theta_k - \eta_k \mathbf{g}_k$$
$$= \theta_k - \eta_k \nabla_\theta f(\theta_k)$$

  - where η_k is the learning rate (step size)

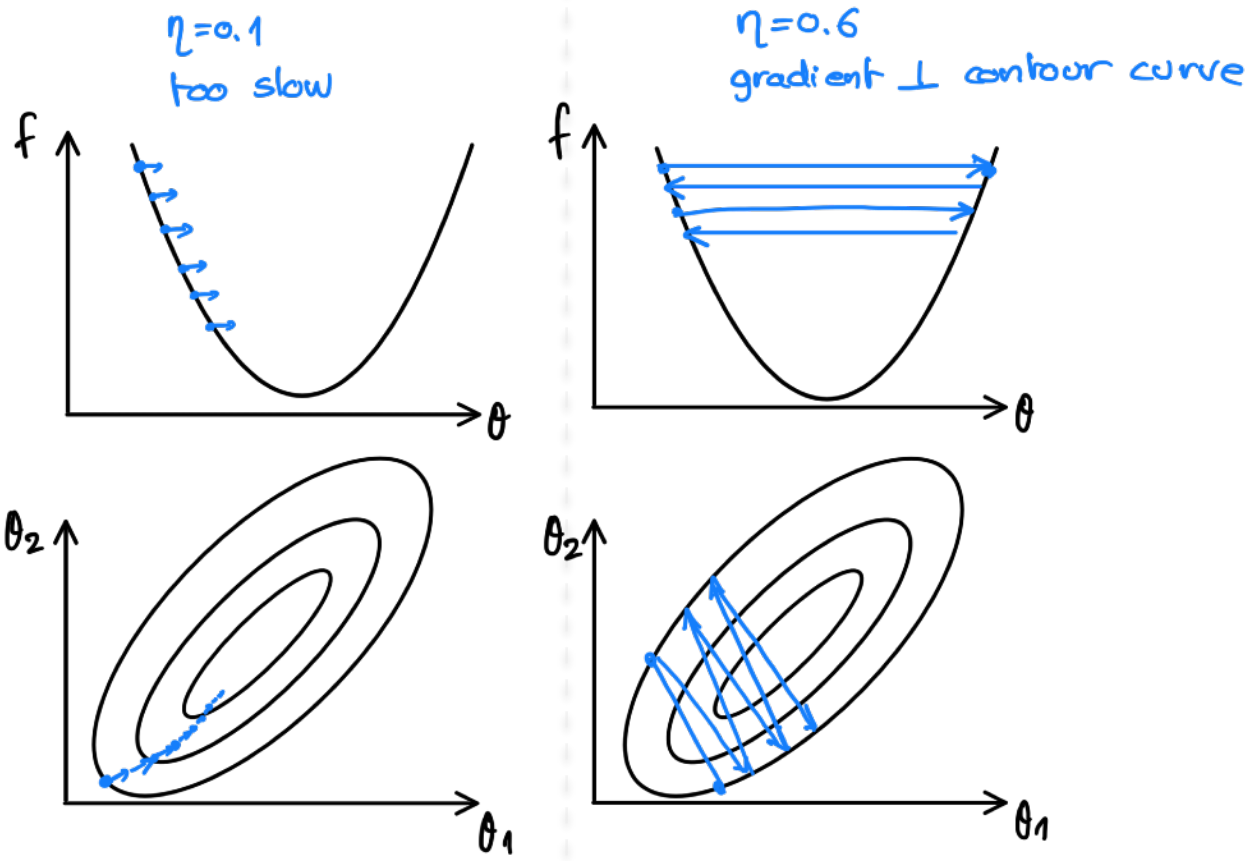
- SGD for linear regression

$$\theta_{k+1} = \theta_k - \eta_k \left[2\mathbf{X}^T(\mathbf{X}\,\theta_k - \mathbf{y})\right]$$



Optimisation
1) Steepest/Stochastic Gradient Descent (SGD) algorithm

- How to choose the learning rate η_k ?
  - $\eta = 0.1$ too slow
  - $\eta = 0.6$ too large



Optimisation
2) Newton/Hessian algorithm

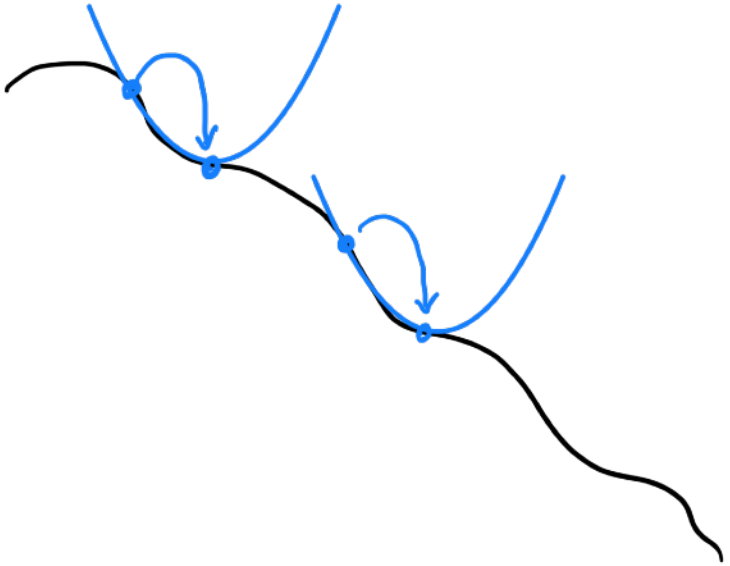
- If we knew the curvature, we could adapt η_k !
- **Taylor series expansion** (approximate $f(\theta)$ around θ_k with a quadratic ball):

$$f_T(\theta) = f(\theta_k) + \mathbf{g}_k^T(\theta - \theta_k) + \frac{1}{2}(\theta - \theta_k)^T \mathbf{H}_k(\theta - \theta_k)$$

- Goal: find the minimum of the quadratic ball

$$\begin{aligned} 0 &= \nabla_\theta f_T(\theta) \\ &= 0 + \mathbf{g}_k + \mathbf{H}_k(\theta - \theta_k) \\ -\mathbf{g}_k &= \mathbf{H}_k(\theta - \theta_k) \\ \theta_{k+1} &= \theta_k - \mathbf{H}_k^{-1}\mathbf{g}_k \end{aligned}$$

  - $\Rightarrow$ this is Newton's method



Optimisation
2) Newton/Hessian algorithm

- Newton's for Linear Regression:

$$\begin{aligned}\theta_{k+1} &= \theta_k - \mathbf{H}_k^{-1}\mathbf{g}_k \\ &= \theta_k - (2\mathbf{X}^T\mathbf{X})^{-1}[2\mathbf{X}^T(\mathbf{X}\theta_k - \mathbf{y})] \\ &= \theta_k - (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{X}\theta_k + (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y} \\ &= (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}\end{aligned}$$

using (as seen before)

Matrix form

$$\begin{aligned}f(\theta) &= (\mathbf{y} - \mathbf{X}\,\theta)^T(\mathbf{y}\mathbf{X}\,\theta) \\ \nabla_\theta f(\theta) &= \frac{\partial}{\partial\theta}(\mathbf{y}^T\mathbf{y}2\mathbf{y}^T\mathbf{X}\,\theta + \theta^T\mathbf{X}^T\mathbf{X}\,\theta) \\ &= -2\mathbf{X}^T\mathbf{y} + 2\mathbf{X}^T\mathbf{X}\,\theta \\ &= 2\mathbf{X}^T(\mathbf{X}\,\theta - \mathbf{y}) \\ \nabla^2_\theta f(\theta) &= 0 + 2\mathbf{X}^T\mathbf{X} \\ &= 2\underset{(d,m)}{\mathbf{X}^T}\underset{(m,d)}{\mathbf{X}}\end{aligned}$$

- This is the theoretical solution we found before
  - we get it in just one step !

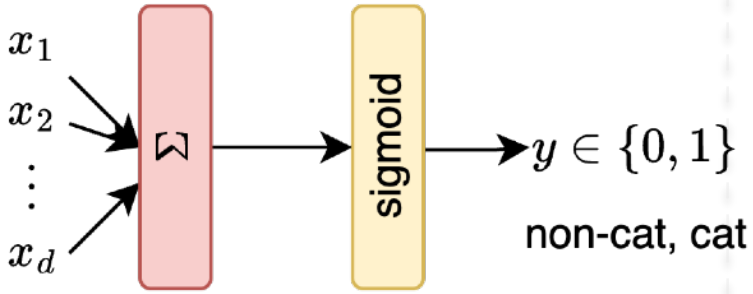
- However, in practice, Newton/Hessian algorithm is very costly
  - ⇒ we need to store $\mathbf{H}$ (if we have 1.000.000 neurons this is a 1.000.000x1.000.000 matrix !)
  - ⇒ solution: BFGS (Broyden-Fletcher-Goldfarb-Shanno) algorithm, L-BFGS (Limited memory) algorithm

Overview

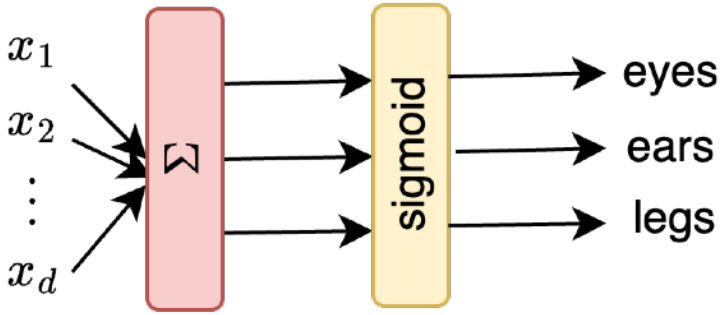
Data	Supervised learning	$(x^{(i)}, y^{(i)})$	
Model	Architecture	Linear regression Non-linear regression Logistic regression Softmax regression Neural Network Deep Learning	Output activation depends on the task Avoid vanishing gradient Parameter initialization Batch Normalization
Cost/Loss	Maximize Likelihood	Gaussian (MSE) Bernoulli (BCE)	Cost depends on the task
Regularization	Avoid over-fitting: L1/L2,DropOut		
Optimizer	For convex optimization For non-convex optimization (SGD, Momentum, ...) First order (SGD), Second order (Hessian)		
Training			
Dataset/ Dataloader	Get mini-batch		
Forward	Compute prediction $\hat{y}^{(i)}$		
Backward (back-propagation)	Compute gradients		
Update			
Early Stopping	Avoid over-fitting		

Various tasks

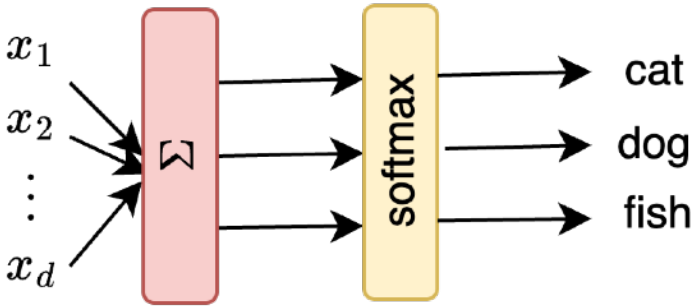
BINARY CLASSIFICATION



MULTI-LABEL

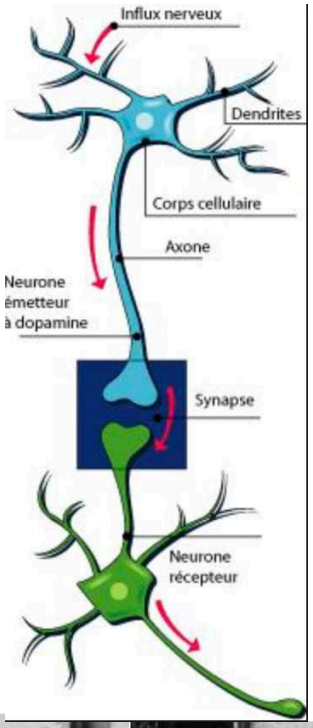


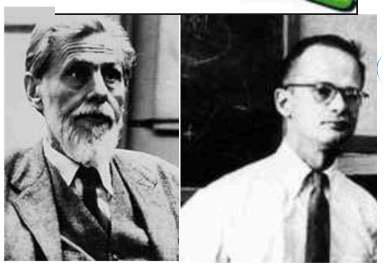
MULTI-CLASS

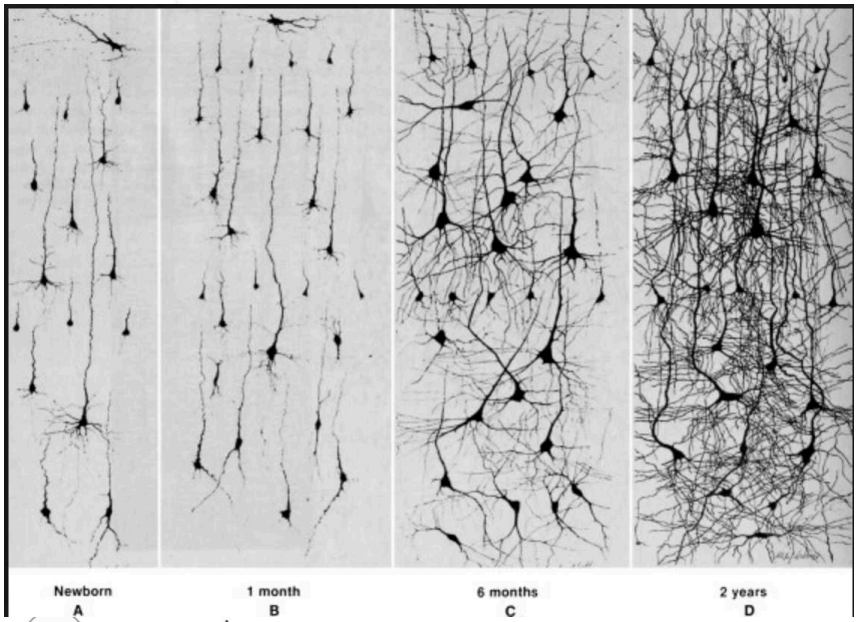


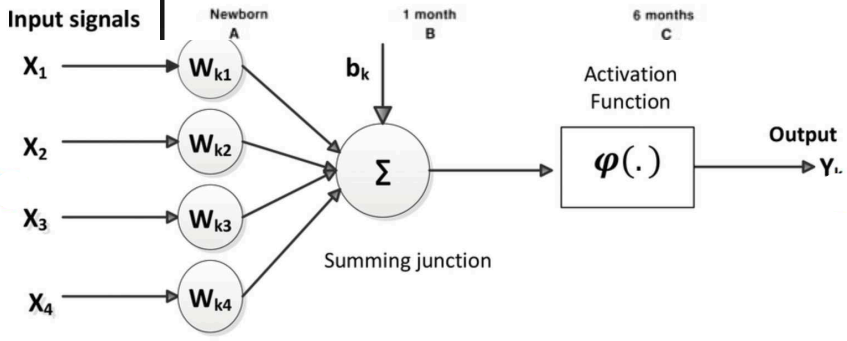
Task: Binary classification
Model: Logistic Regression

[McCulloch - Pitts, 1943] model of a neuron









McCulloch-Pitts computational model of a neuron.

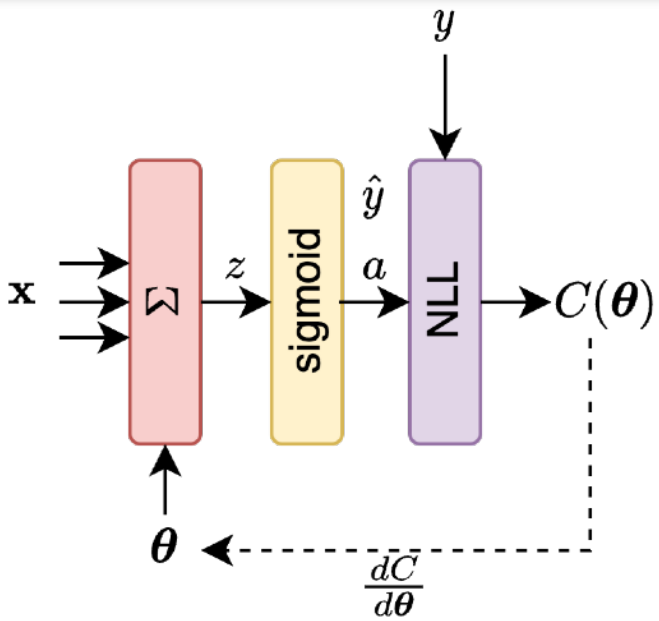
Task: Binary classification
Model: Logistic Regression

Model

- The logistic regression
  - is a **binary classification** algorithm
  - it outputs the probability of the class 1

$$\hat{y}^{(i)} = a^{(i)} \in [0,1] = P(y^{(i)} = 1 \,|\, \mathbf{x}^{(i)}, \theta)$$

  - It is a linear regression
    $z^{(i)} = \mathbf{x}^{(i)}\, \boldsymbol{\theta}$
  - followed by a sigmoid function
    $a^{(i)} = \sigma(\mathbf{z}^{(i)})$



Activation function

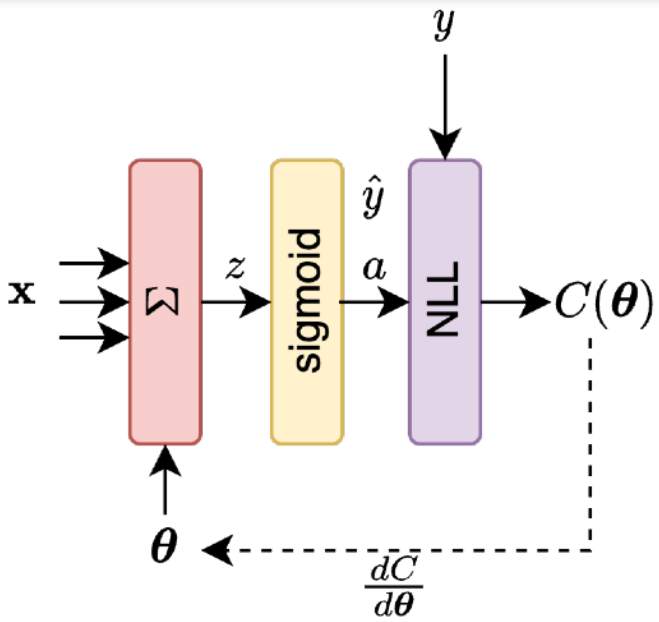
– **Logistic** (inverse of the logit*)
  - a specific case of sigmoid function (S-shaped)

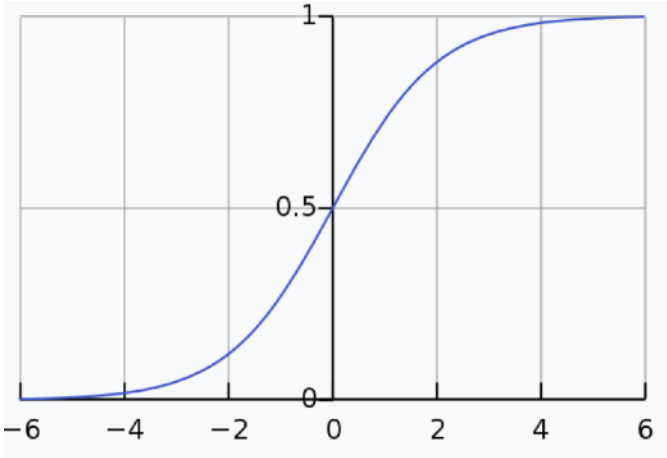
$$a^{(i)} = \sigma(z^{(i)}) = \frac{1}{1+e^{-z^{(i)}}}$$

– **Derivative** of the logistic function w.r.t. its input

$$\frac{\partial \sigma(z)}{\partial z} = \sigma(z)(1-\sigma(z)) = a(1-a)$$

* logit function: $z = \log\left(\frac{p}{1-p}\right)$





Linear separating hyper-plane

– The logistic regression defines a linear separating hyper-plane

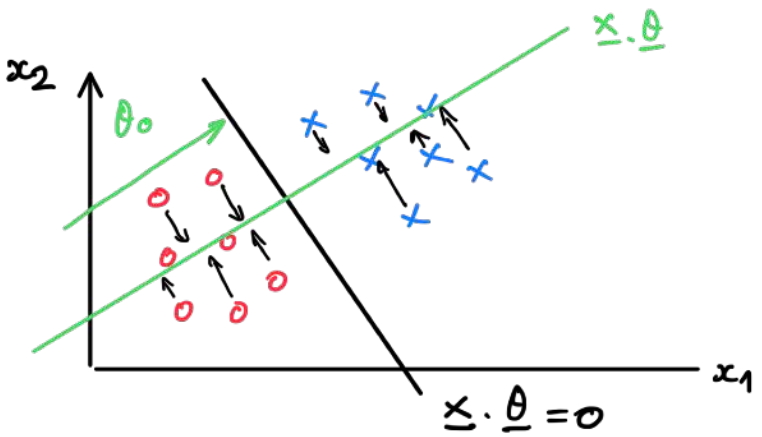
$$P(y^{(i)} = 1 \,|\, \mathbf{x}^{(i)}, \theta) = \frac{1}{2}$$

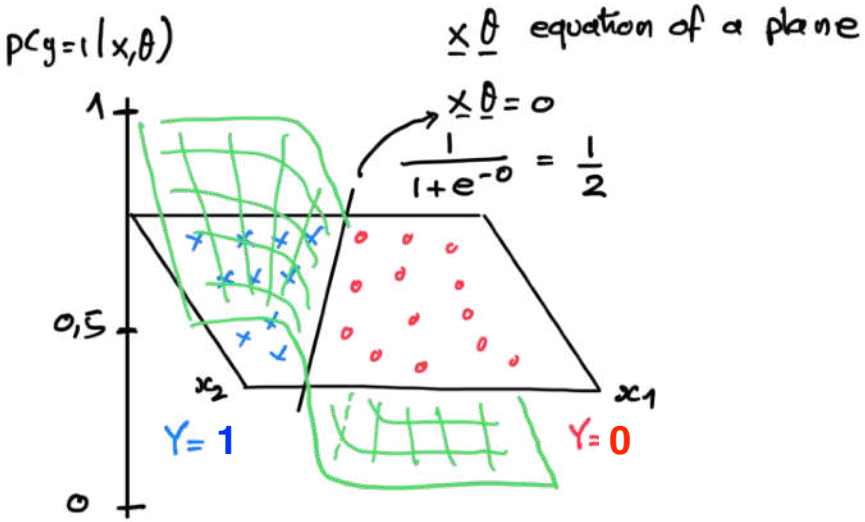
$$\sigma(z) = \frac{1}{2}$$

$$\frac{1}{1 + e^{-z}} = \frac{1}{2}$$

$$z = \mathbf{x}^{(i)}\theta = 0$$

- $\mathbf{x}^{(i)}\theta = 0$ is the equation of a plane





Task: Binary classification
Model: Logistic Regression

Cost function (1)

- For Linear Regression
  - $\Rightarrow$ uncertainty (random variable) is model by a Gaussian distribution
- For Logistic Regression
  - $\Rightarrow$ uncertainty (random variable) is model by a Bernoulli distribution

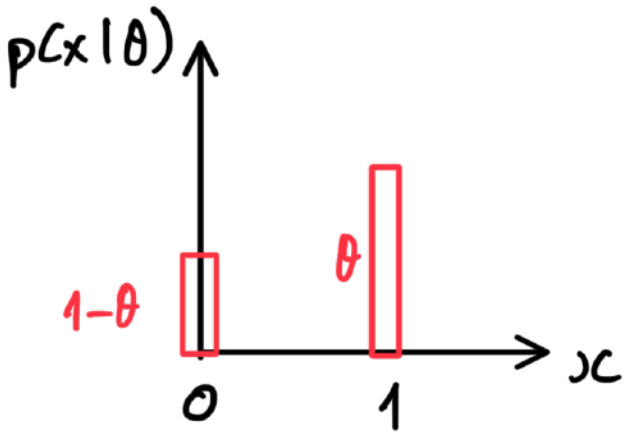
- Learning is about **minimising uncertainty → entropy**

Maximum Likelihood Estimation (MLE)
Bernoulli distribution

- Bernoulli distribution: X takes value in $\{0,1\}$
  - model of a coin (probability of head or tail)

$$P(X=x|\theta) = \begin{cases} p & \text{if } x=1 \\ 1-p & \text{if } x=0 \end{cases}$$
$$= p^x(1-p)^{(1-x)}$$

  - with $p \in [0,1]$



Maximum Likelihood Estimation (MLE)
Bernoulli distribution

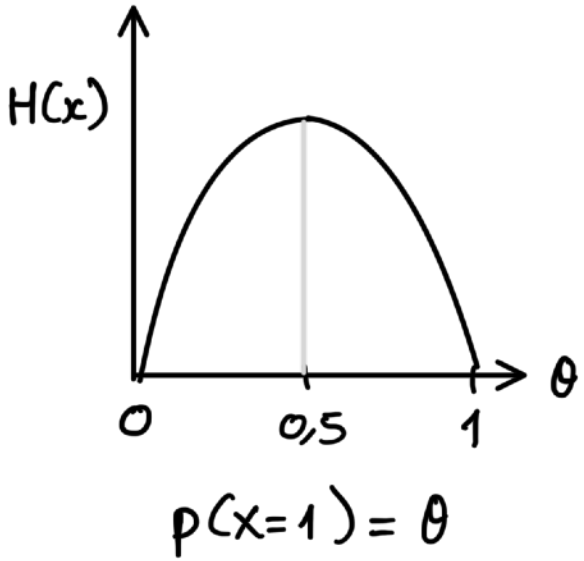
- **Entropy**: is a measure of the uncertainty associated with a random variable:

$$H(X) = -\sum_{x} p(x|\theta) \quad \log(p(x|\theta))$$

- **Entropy of a Bernoulli distribution:**

$$H(X) = -\sum_{x=0}^{1} p^x(1-p)^{(1-x)} \quad \log\left(p^x(1-p)^{(1-x)}\right)$$

$$= -\underbrace{[(1-p)\log(1-p)}_{if\ x=0} + \underbrace{p\log(p)}_{if\ x=1}]$$



- **Entropy of a Gaussian distribution** at D dimensions

$$H(\mathcal{N}(\mu, \Sigma)) = \frac{1}{2}\log\left((2\pi e)^D |\Sigma|\right)$$

  - proportional to the determinant of the covariance matrix

Cost function (2)

- Logistic regression specifies the probability of a binary output $y^{(i)} \in \{0,1\}$ given input $\mathbf{x}^{(i)}$

$$p(y \mid \mathbf{x}, \theta) = \prod_{i=1}^{n} p(y^{(i)} \mid \mathbf{x}^{(i)}, \theta)$$
$$= \prod_{i=1}^{n} \text{Ber}(y^{(i)} \mid \mathbf{x}^{(i)}, \theta)$$
$$= \prod_{i=1}^{n} [\hat{\mathbf{y}}^{(i)}]^{y^{(i)}} [1 - \hat{\mathbf{y}}^{(i)}]^{1-y^{(i)}}$$

BERNOULLI DISTRIBUTION

$y=1 \rightarrow \pi$
$y=0 \rightarrow (1-\pi)$
$\Big\}\ P(Y=y) = \pi^{y}(1-\pi)^{(1-y)}$

  - with $\hat{\mathbf{y}}^{(i)} = a^{(i)} = \dfrac{1}{1+e^{-z^{(i)}}}$ and $z^{(i)} = \mathbf{x}^{(i)}\theta$

- **Cost to minimise** = Negative Log-Likelihood (NLL), **Binary Cross-Entropy** $H(y^{(i)}, \hat{\mathbf{y}}^{(i)})$

$$C(\theta) = -\log p(y \mid \mathbf{x}, \theta)$$
$$= -\sum_{i=1}^{n} y^{(i)} \log \hat{\mathbf{y}}^{(i)} + (1 - y^{(i)}) \log(1 - \hat{\mathbf{y}}^{(i)})$$

- **Derivative** of the cost w.r.t. to the output

$$\frac{\partial C(\theta)}{\partial \hat{\mathbf{y}}} = -\sum_{i=1}^{n} \left(\frac{y^{(i)}}{\hat{\mathbf{y}}^{(i)}} - \frac{(1 - y^{(i)})}{1 - \hat{\mathbf{y}}^{(i)}} \right)$$
$$= \sum_{i=1}^{n} \frac{\hat{\mathbf{y}}^{(i)} - y^{(i)}}{\hat{\mathbf{y}}^{(i)}(1 - \hat{\mathbf{y}}^{(i)})}$$

Optimisation using Gradient

– To minimise the cost $C(\theta)$, we compute the gradient of the $C(\theta)$ w.r.t. to the parameters θ

- we use the **chain-rule** (* next slide):

$$\mathbf{g} = \frac{\partial C(\theta)}{\partial \theta}$$

$$= \frac{\partial C(\theta)}{\partial \hat{\mathbf{y}}} \frac{\partial \hat{\mathbf{y}}}{\partial z} \frac{\partial z}{\partial \theta}$$

$$= \sum_{i=1}^{n} \frac{\hat{\mathbf{y}}^{(i)} - y^{(i)}}{\hat{\mathbf{y}}^{(i)}(1-\hat{\mathbf{y}}^{(i)})} \cdot \hat{\mathbf{y}}^{(i)}(1-\hat{\mathbf{y}}^{(i)}) \cdot \mathbf{x}^{(i)}$$

$$= \sum_{i=1}^{n} (\mathbf{x}^{(i)})^T (\hat{\mathbf{y}}^{(i)} - y^{(i)})$$

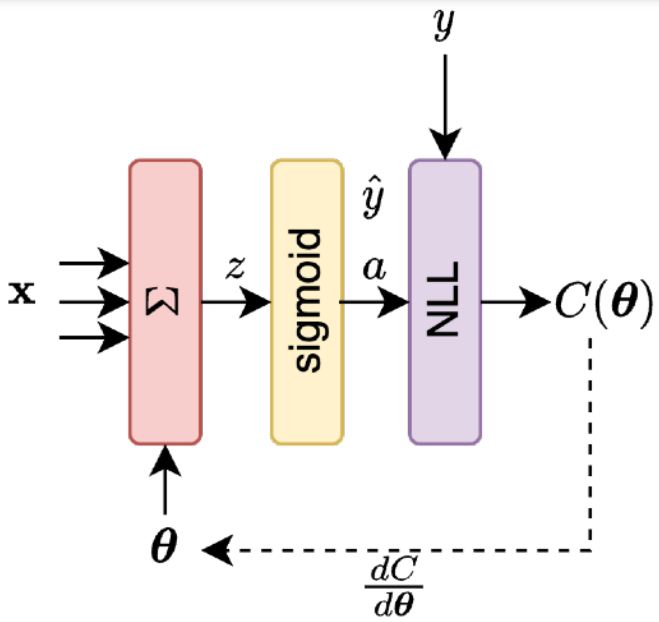
$$= \mathbf{X}^T(\hat{\mathbf{y}} - \mathbf{y})$$

– SGD update at iteration $k+1$

$$\theta_{k+1} = \theta_k - \eta_k \mathbf{g}_k$$

$$= \theta_k - \eta_k \mathbf{X}^T(\hat{\mathbf{y}}_k - \mathbf{y})$$

- where $\hat{\mathbf{y}}_k^{(i)} = \sigma(\mathbf{x}^{(i)}\theta_k)$ at iteration k



Note: it looks a bit like the least-square solution :

$$\theta_{k+1} = \theta_k - \eta_k \left[2\mathbf{X}^T(\mathbf{X}\,\theta_k - \mathbf{y})\right]$$

Optimisation using Hessian (Iteratively Reweighted Least Square - IRLS)

– Newton update:

$$\theta_{k+1} = \theta_k - \mathbf{H}_k^{-1}\mathbf{g}_k$$

- with Gradient and Hessian

$$\mathbf{g}_k = \mathbf{X}^T(\hat{y}_k - \mathbf{y})$$
$$\mathbf{H}_k = \mathbf{X}^T\mathbf{S}_k\mathbf{X}$$

– Newton update: at iteration $k+1$

$$\begin{aligned}\theta_{k+1} &= \theta_k - H_k^{-1}g_k \\ &= \theta_k + \left(\mathbf{X}^T\mathbf{S}_k\mathbf{X}\right)^{-1}\mathbf{X}^T(\mathbf{y} - \hat{y}_k) \\ &= \left(\mathbf{X}^T\mathbf{S}_k\mathbf{X}\right)^{-1}\left[(\mathbf{X}^T\mathbf{S}_k\mathbf{X}\theta_k + \mathbf{X}^T(\mathbf{y} - \hat{y}_k)\right] \\ &= \left(\mathbf{X}^T\mathbf{S}_k\mathbf{X}\right)^{-1}\mathbf{X}^T\left[(\mathbf{S}_k\mathbf{X}\theta_k + (\mathbf{y} - \hat{y}_k)\right]\end{aligned}$$

Proof

$$\begin{aligned}H &= \frac{\partial}{\partial\theta}g(\theta)^T \\ &= \sum_{i=1}^{n}\hat{\mathbf{y}}^{(i)}(1-\hat{\mathbf{y}}^{(i)})\mathbf{x}^{(i)}(\mathbf{x}^{(i)})^T \\ &= \mathbf{X}^T\text{diag}\left(\hat{\mathbf{y}}^{(i)}(1-\hat{\mathbf{y}}^{(i)})\right)\mathbf{X}\end{aligned}$$

We note $\mathbf{S}_k = \text{diag}\left(\hat{\mathbf{y}}_k^{(1)}(1-\hat{\mathbf{y}}_k^{(1)})\cdots\hat{\mathbf{y}}_k^{(n)}(1-\hat{\mathbf{y}}_k^{(n)})\right)$

H is positive definite, hence the NLL is convex and has a unique minimum

Note : it looks a bit like the least-square solution :

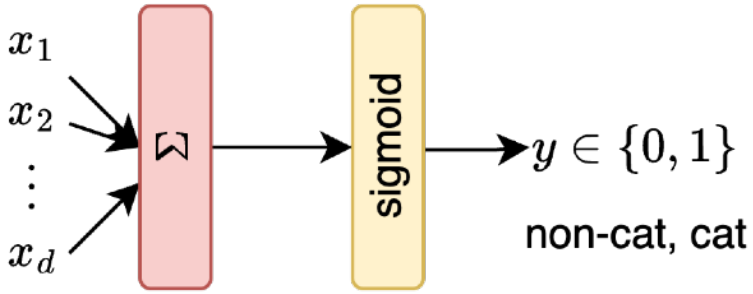
$$\begin{aligned}\theta_{k+1} &= \theta_k - \mathbf{H}_k^{-1}\mathbf{g}_k \\ &= \theta_k - (2\mathbf{X}^T\mathbf{X})^{-1}[2\mathbf{X}^T(\mathbf{X}\theta_k - \mathbf{y})] \\ &= \theta_k - (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{X}\theta_k + (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y} \\ &= (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}\end{aligned}$$

Overview

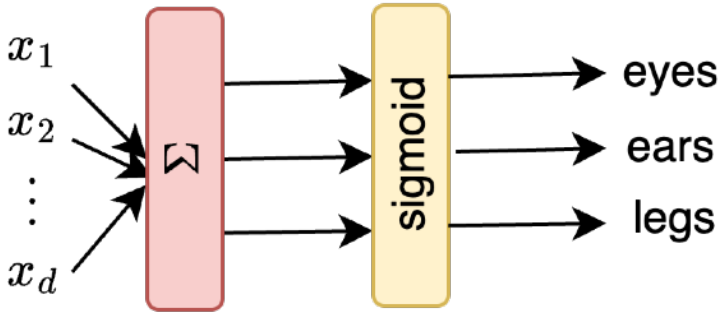
Data	Supervised learning	$(x^{(i)}, y^{(i)})$	
Model	Architecture	Linear regression Non-linear regression Logistic regression Softmax regression Neural Network Deep Learning	Output activation depends on the task Avoid vanishing gradient Parameter initialization Batch Normalization
Cost/Loss	Maximize Likelihood	Gaussian (MSE) Bernoulli (BCE)	Cost depends on the task
Regularization	Avoid over-fitting: L1/L2,DropOut		
Optimizer	For convex optimization For non-convex optimization (SGD, Momentum, ...) First order (SGD), Second order (Hessian)		
Training			
Dataset/ Dataloader	Get mini-batch		
Forward	Compute prediction $\hat{y}^{(i)}$		
Backward (back-propagation)	Compute gradients		
Update			
Early Stopping	Avoid over-fitting		

Various tasks

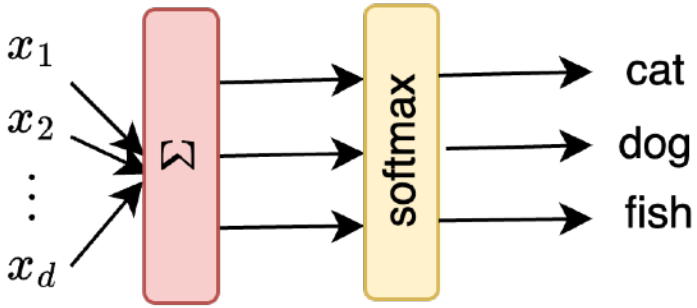
BINARY CLASSIFICATION



MULTI-LABEL



MULTI-CLASS

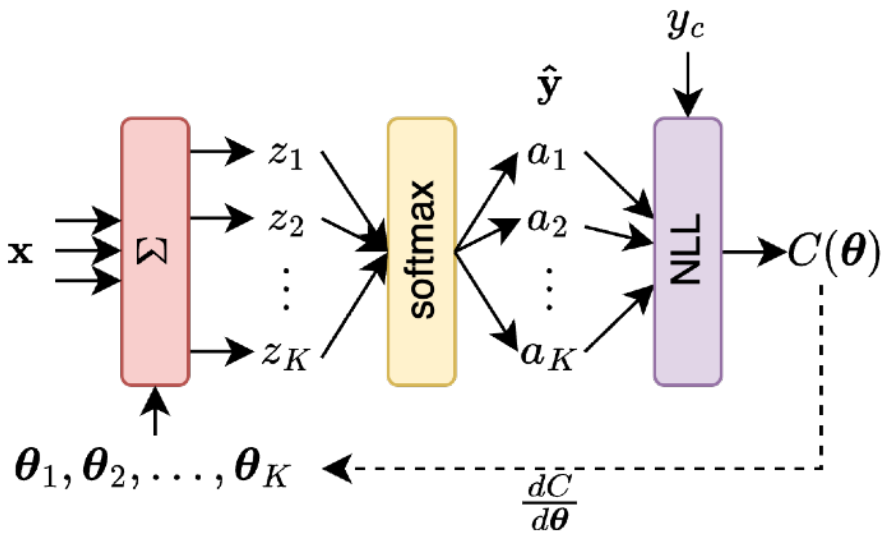


Model

- The softmax regression
  - is a **multi-class classification** algorithm
  - it outputs the vector of probability for all classes

$$\hat{y}_c^{(i)} = a_c^{(i)} = P(y_c^{(i)} = 1 \,|\, \mathbf{x}^{(i)}, \theta)$$
$$= P(y^{(i)} = c \,|\, \mathbf{x}^{(i)}, \theta)$$

$$\hat{y}_c^{(i)} \in [0,1], \quad \sum_c \hat{y}_c^{(i)} = 1$$



  - It is a linear regression

$$z_c^{(i)} = \mathbf{x}^{(i)} \boldsymbol{\theta}_c$$

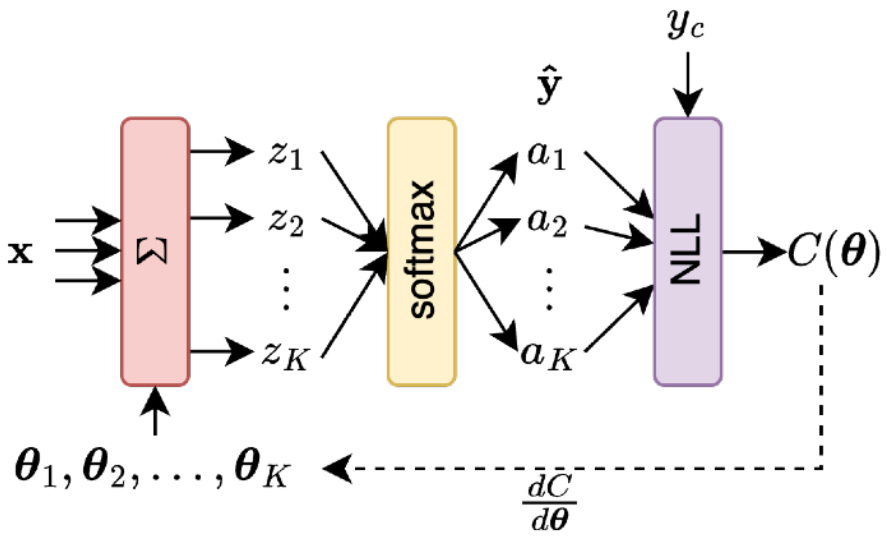
  - followed by a softmax function

Activation functions

– **Softmax function**

$$a_c^{(i)} = \text{softmax}(z_c^{(i)}) = \frac{e^{\mathbf{z}_c^{(i)}}}{\sum_{c'=1}^{K} e^{z_{c'}^{(i)}}}$$

_ It is easy to show that $a_1^{(i)} + \cdots + a_K^{(i)} = 1$



Task: Multi-class classification
Model: Softmax Regression

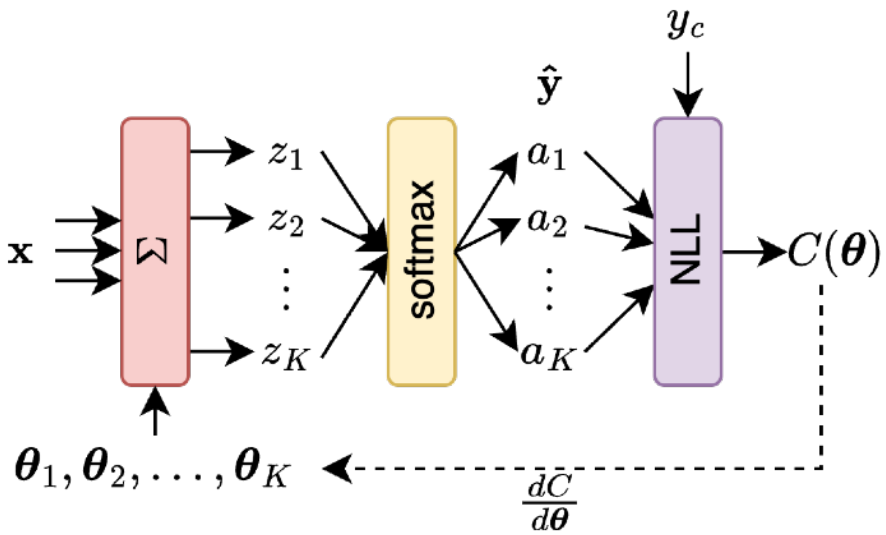
Cost function

- Likelihood function (to maximise)

- Notation: indicator function
  - $\mathbb{I}_c(y^{(i)}) \triangleq \begin{cases} 1 & \text{if } y^{(i)} = c \\ 0 & \text{otherwise} \end{cases}$

$$P(\mathbf{y} \mid \mathbf{x}, \theta) = \prod_{i=1}^{m} p(y^{(i)} \mid \mathbf{x}^{(i)}, \theta)$$

$$= \prod_{i=1}^{m} \prod_{c=1}^{K} (a_c^{(i)})^{\mathbb{I}_c(y^{(i)})}$$



- **Cost** (to minimise): Negative Log-Likelihood (NLL), **Categorical Cross Entropy**

$$C(\theta) = -\log p(\mathbf{y} \mid \mathbf{x}, \theta)$$

$$= -\sum_{i=1}^{m} \sum_{c=1}^{K} \mathbb{I}_c(y^{(i)}) \log(a_c^{(i)})$$

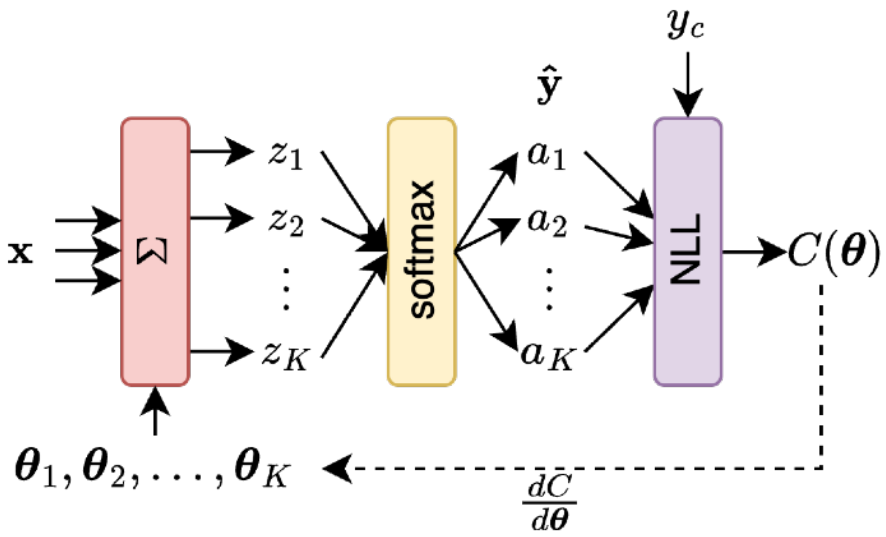
Task: Multi-class classification
Model: Softmax Regression

For the specific case of two classes (1)

– Likelihood

$$P(y^{(i)}=1\,|\,\mathbf{x}^{(i)},\theta) = (a_1^{(i)})^1(a_2^{(i)})^0 = a_1^{(i)} = \frac{e^{\mathbf{x}^{(i)}\theta_1}}{e^{\mathbf{x}^{(i)}\theta_1}+e^{\mathbf{x}^{(i)}\theta_2}}$$

$$P(y^{(i)}=2\,|\,\mathbf{x}^{(i)},\theta) = (a_1^{(i)})^0(a_2^{(i)})^1 = a_2^{(i)} = \frac{e^{\mathbf{x}^{(i)}\theta_2}}{e^{\mathbf{x}^{(i)}\theta_1}+e^{\mathbf{x}^{(i)}\theta_2}}$$



– Negative Log-Likelihood (NLL)

$$C(\theta) = -\log p(\mathbf{y}\,|\,\mathbf{x},\theta)$$

$$= -\sum_{i=1}^{m} \mathbb{I}_1(y^{(i)})\log(a_1^{(i)}) + \mathbb{I}_2(y^{(i)})\log(a_2^{(i)})$$

$$= -\sum_{i=1}^{m} y_1^{(i)}\log(a_1^{(i)}) + y_2^{(i)}\log(a_2^{(i)})$$

For the specific case of two classes (2)

– Derivative of the cost w.r.t. the parameters

$$\frac{\partial}{\partial \theta_2} C(\theta) = \frac{\partial}{\partial \theta_2} \left(- \sum_i \mathbb{I}_1(y^{(i)}) \log(a_1^{(i)}) + \mathbb{I}_2(y^{(i)}) \log(a_2^{(i)}) \right)$$

$$= - \sum_i \mathbb{I}_1(y^{(i)}) \frac{\partial}{\partial \theta_2} \log(a_1^{(i)}) + \mathbb{I}_2(y^{(i)}) \frac{\partial}{\partial \theta_2} \log(a_2^{(i)})$$

$$\frac{\partial}{\partial \theta_2} \log(a_1^{(i)}) = \frac{\partial}{\partial \theta_2} \log \frac{e^{\mathbf{x}^{(i)}\theta_1}}{e^{\mathbf{x}^{(i)}\theta_1} + e^{\mathbf{x}^{(i)}\theta_2}}$$

$$= \frac{\partial}{\partial \theta_2} \left(\mathbf{x}^{(i)}\theta_1 - \log(e^{\mathbf{x}^{(i)}\theta_1} + e^{\mathbf{x}^{(i)}\theta_2}) \right)$$

$$= 0 - \frac{\mathbf{x}^{(i)} e^{\mathbf{x}^{(i)}\theta_2}}{e^{\mathbf{x}^{(i)}\theta_1} + e^{\mathbf{x}^{(i)}\theta_2}}$$

$$= - \mathbf{x}^{(i)} a_2^{(i)}$$

$$\frac{\partial}{\partial \theta_2} \log(a_2^{(i)}) = \cdots = \mathbf{x}^{(i)}(1 - a_2^{(i)})$$

$$\frac{\partial C(\theta)}{\partial \theta_2} = - \sum_i \mathbb{I}_1(y^{(i)})(-\mathbf{x}^{(i)} a_2^{(i)}) + \mathbb{I}_2(y^{(i)}) \mathbf{x}^{(i)}(1 - a_2^{(i)})$$

For the specific case of two classes (3)

- Now we use the notation of the Logistic regression $y^{(i)} \in \{0,1\}$:
  - $\mathbb{I}_1(y^{(i)}) = (1 - y^{(i)})$
  - $\mathbb{I}_2(y^{(i)}) = y^{(i)}$

$$\begin{aligned}
\frac{\partial C(\theta)}{\partial \theta_2} &= -\sum_i \mathbb{I}_1(y^{(i)})(-\mathbf{x}^{(i)} a_2^{(i)}) + \mathbb{I}_2(y^{(i)})\mathbf{x}^{(i)}(1 - a_2^{(i)}) \\
&= -\sum_i (1 - y^{(i)})(-\mathbf{x}^{(i)} a_2^{(i)}) + y^{(i)}\mathbf{x}^{(i)}(1 - a_2^{(i)}) \\
&= -\sum_i -\mathbf{x}^{(i)} a_2^{(i)} + y^{(i)}\mathbf{x}^{(i)} a_2^{(i)} + y^{(i)}\mathbf{x}^{(i)} - y^{(i)}\mathbf{x}^{(i)} a_2^{(i)} \\
&= \sum_i \mathbf{x}^{(i)}(a_2^{(i)} - y^{(i)})
\end{aligned}$$

- **Conclusion:** multi-class with two classes is equivalent to the logistic regression !!!

Overview

Data	Supervised learning	$(x^{(i)}, y^{(i)})$	
Model	Architecture	Linear regression Non-linear regression Logistic regression Softmax regression Neural Network Deep Learning	Output activation depends on the task Avoid vanishing gradient Parameter initialization Batch Normalization
Cost/Loss	Maximize Likelhood	Gaussian (MSE) Bernoulli (BCE)	Cost depends on the task
Regularization	Avoid over-fitting: L1/L2,DropOut		
Optimizer	For convex optimization For non-convex optimization (SGD, Momentum, ...) First order (SGD), Second order (Hessian)		
Training			
Dataset/ Dataloader		Get mini-batch	
Forward		Compute prediction $\hat{y}^{(i)}$	
Backward (back-propagation)		Compute gradients	
Update			
Early Stopping		Avoid over-fitting	

Back-propagation

Optimisation

- To perform SGD (gradient descent), we need gradients !
- We need an efficient way to compute the gradients ⇒ **back-propagation**

- To exemplify the back-propagation, we consider a softmax regression with 2 classes

$$C(\theta) = -\sum_{i=1}^{n} \mathbb{I}_1(y^{(i)}) \log\left(\frac{e^{\boldsymbol{\theta}_1 \mathbf{x}^{(i)}}}{e^{\boldsymbol{\theta}_1 \mathbf{x}^{(i)}} + e^{\boldsymbol{\theta}_2 \mathbf{x}^{(i)}}}\right) + \mathbb{I}_2(y^{(i)}) \log\left(\frac{e^{\boldsymbol{\theta}_2 \mathbf{x}^{(i)}}}{e^{\boldsymbol{\theta}_1 \mathbf{x}^{(i)}} + e^{\boldsymbol{\theta}_2 \mathbf{x}^{(i)}}}\right)$$

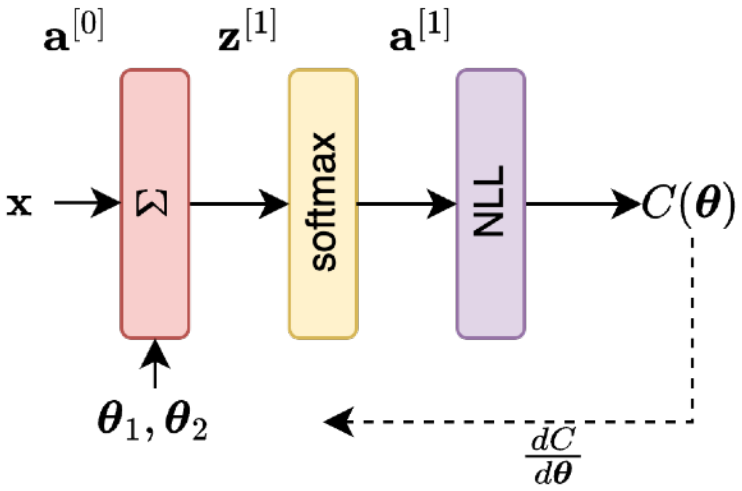
- We consider it as a succession of 4 layers
  - **Forward:** compute the layers successively

$$\mathbf{a}^{[0]} = \mathbf{x}$$

$$z_1^{[1]} = \boldsymbol{\theta}_1 \mathbf{a}^{[0]} \qquad z_2^{[1]} = \boldsymbol{\theta}_2 \mathbf{a}^{[0]}$$

$$a_1^{[1]} = \log\left(\frac{e^{z_1^{[1]}}}{e^{z_1^{[1]}} + e^{z_2^{[1]}}}\right) \qquad a_2^{[1]} = \log\left(\frac{e^{z_2^{[1]}}}{e^{z_1^{[1]}} + e^{z_2^{[1]}}}\right)$$

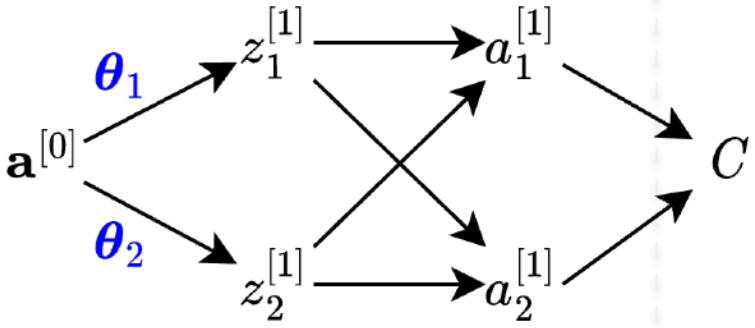
$$C = -\sum_i \mathbb{I}_1(y) a_1^{[1]} + \mathbb{I}_2(y) a_2^{[1]}$$

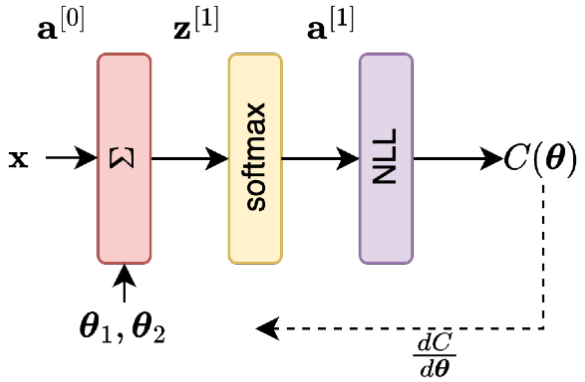


Back-propagation

- We consider it as a succession of layers
  - Composite functions

$$C(\boldsymbol{\theta}) = C\left\{ a_1^{[1]}\underline{\left[z_1^{[1]}(\mathbf{a}^{[0]},\boldsymbol{\theta}_1), z_2^{[1]}(a^{[0]},\boldsymbol{\theta}_2)\right]}, a_2^{[1]}\underline{\left[z_1^{[1]}(\mathbf{a}^{[0]},\boldsymbol{\theta}_1), z_2^{[1]}(\mathbf{a}^{[0]},\boldsymbol{\theta}_2)\right]} \right\}$$





  - Computing derivatives using the **chain rule**

$$\frac{\partial C(\theta)}{\partial \theta_1} = \frac{\partial C}{\partial a_1^{[1]}}\frac{\partial a_1^{[1]}}{\partial z_1^{[1]}}\frac{\partial z_1^{[1]}}{\partial \theta_1} + \frac{\partial C}{\partial a_1^{[1]}}\frac{\partial a_1^{[1]}}{\partial z_2^{[1]}}\frac{\partial z_2^{[1]}}{\partial \theta_1} + \frac{\partial C}{\partial a_2^{[1]}}\frac{\partial a_2^{[1]}}{\partial z_1^{[1]}}\frac{\partial z_1^{[1]}}{\partial \theta_1} + \frac{\partial C}{\partial a_2^{[1]}}\frac{\partial a_2^{[1]}}{\partial z_2^{[1]}}\frac{\partial z_2^{[1]}}{\partial \theta_1}$$

    - This is **Inefficient**: we compute several times the same things !!!

Back-propagation

Two ways to implement gradient computation

– Suppose we have the following composition of functions

$$f^{[L]}(f^{[L-1]}(\cdots f^{[2]}(f^{[1]}(\mathbf{x})))$$

– Solution 1

$$\frac{\partial f^{[L]}(f^{[L-1]}(\cdots f^{[2]}(f^{[1]}(\mathbf{x}))))}{\partial \mathbf{x}} = \frac{\partial f^{[L]}}{\partial f^{[L-1]}} \cdot \frac{\partial f^{[L-1]}}{\partial f^{[L-2]}} \cdots \frac{\partial f^{[2]}}{\partial f^{[1]}} \cdot \frac{\partial f^{[1]}}{\partial \mathbf{x}}$$

- chain rule
- need a lot of memory (need to store all matrices)

– Solution 2

- start from the end and move back to the start

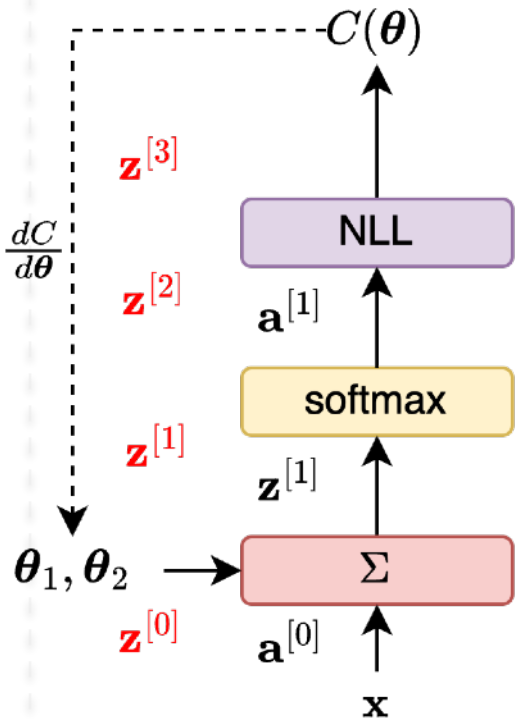
Jacobian

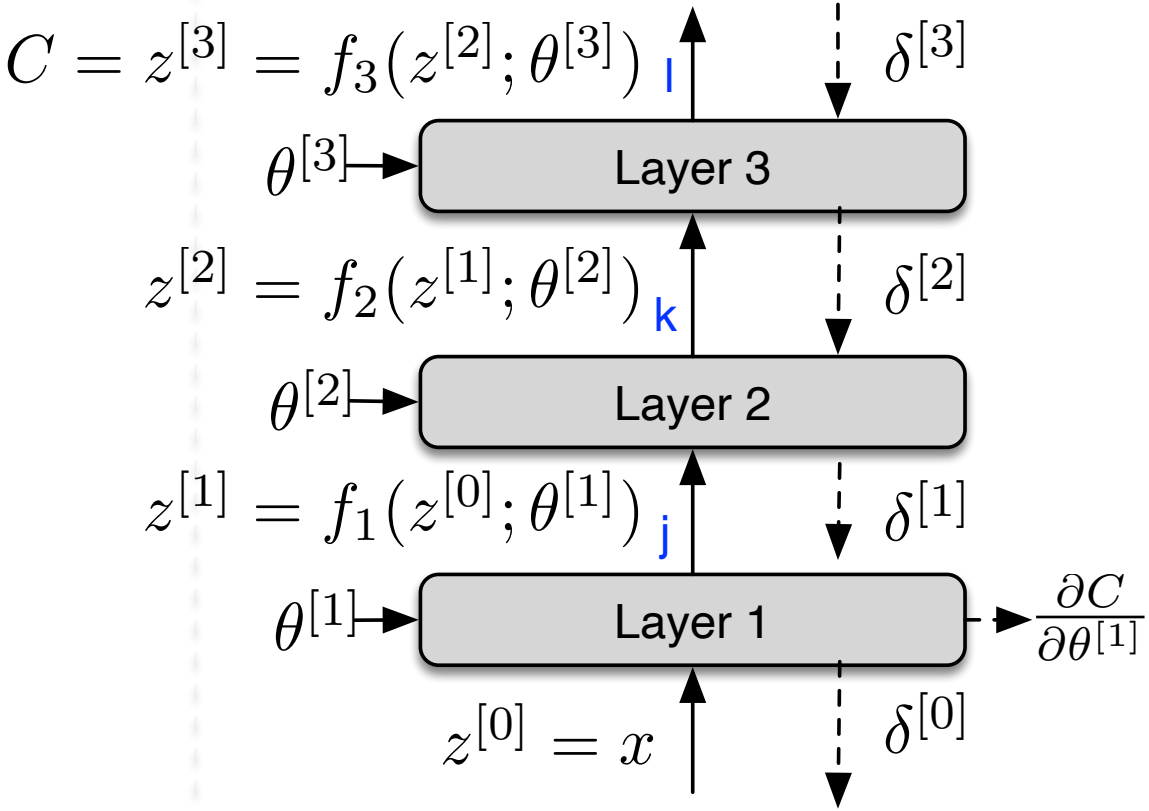
$$\frac{\partial f^{[L]}(f^{[L-1]}(\cdots f^{[2]}(f^{[1]}(\mathbf{x}))))}{\partial \mathbf{x}} = \left(\left(\left(\frac{\partial f^{[L]}}{\partial f^{[L-1]}} \cdot \frac{\partial f^{[L-1]}}{\partial f^{[L-2]}}\right)\cdots\right)\frac{\partial f^{[2]}}{\partial f^{[1]}}\right) \cdot \frac{\partial f^{[1]}}{\partial \mathbf{x}}$$

- Compute first (-) then second (-) then third (-)
- It uses less memory (the intermediate results are always vectors: vector x matrix = vector)
- ⇒ this is the **back-propagation algorithm**

Back-propagation

Generalisation: Layer specification





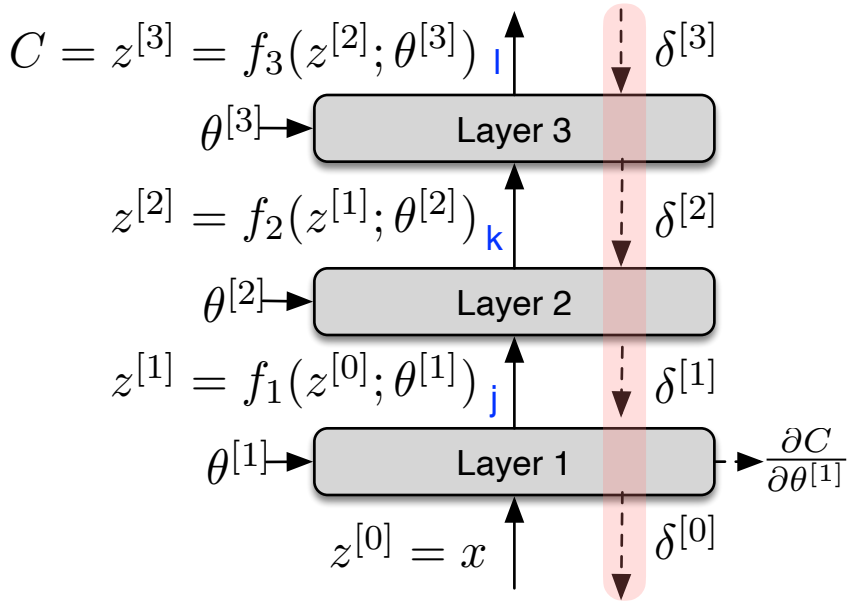
Back-propagation
Derivative via layer-specification

$$\frac{\partial C}{\partial \theta^{[1]}} = \sum_j \frac{\partial C}{\partial z_j^{[1]}} \frac{\partial z_j^{[1]}}{\partial \theta^{[1]}}$$

$$= \sum_j \left(\sum_k \frac{\partial C}{\partial z_k^{[2]}} \frac{\partial z_k^{[2]}}{\partial z_j^{[1]}} \right) \frac{\partial z_j^{[1]}}{\partial \theta^{[1]}}$$

$$= \sum_j \left(\sum_k \left(\sum_l \frac{\partial C}{\partial z_l^{[3]}} \frac{\partial z_l^{[3]}}{\partial z_k^{[2]}} \right) \frac{\partial z_k^{[2]}}{\partial z_j^{[1]}} \right) \frac{\partial z_j^{[1]}}{\partial \theta^{[1]}}$$

$$= \sum_j \left(\sum_k \left(\sum_{l=1} 1 \frac{\partial z_l^{[3]}}{\partial z_k^{[2]}} \right) \frac{\partial z_k^{[2]}}{\partial z_j^{[1]}} \right) \frac{\partial z_j^{[1]}}{\partial \theta^{[1]}}$$



- Note:
  - this is equivalent to what we have seen above but without redundant computation

$$\frac{\partial C(\theta}{\partial \theta_1} = \frac{\partial z^{[3]}}{\partial z_1^{[2]}} \frac{\partial z_1^{[2]}}{\partial z_1^{[1]}} \frac{\partial z_1^{[1]}}{\partial \theta_1} + \frac{\partial z^{[3]}}{\partial z_1^{[2]}} \frac{\partial z_1^{[2]}}{\partial z_2^{[1]}} \frac{\partial z_2^{[1]}}{\partial \theta_1} + \frac{\partial z^{[3]}}{\partial z_2^{[2]}} \frac{\partial z_2^{[2]}}{\partial z_1^{[1]}} \frac{\partial z_1^{[1]}}{\partial \theta_1} + \frac{\partial z^{[3]}}{\partial z_2^{[2]}} \frac{\partial z_2^{[2]}}{\partial z_2^{[1]}} \frac{\partial z_2^{[1]}}{\partial \theta_1}$$

- Forward $z^{[0]} = \mathbf{x}^{(i)} \rightarrow z^{[1]}(\mathbf{x}^{(i)}) \rightarrow z^{[2]}(\mathbf{x}^{(i)}) \rightarrow z^{[3]}(\mathbf{x}^{(i)}) = C$
- Backward $\delta^{[0]} \leftarrow \delta^{[1]} \leftarrow \delta^{[2]} \leftarrow \delta^{[3]} = 1$

Back-propagation
Layer specification

- We only have to design a **module** with
  - a **Forward** method:

$$z^{[l]} = f_l(z^{[l-1]}, \theta^{[l]})$$

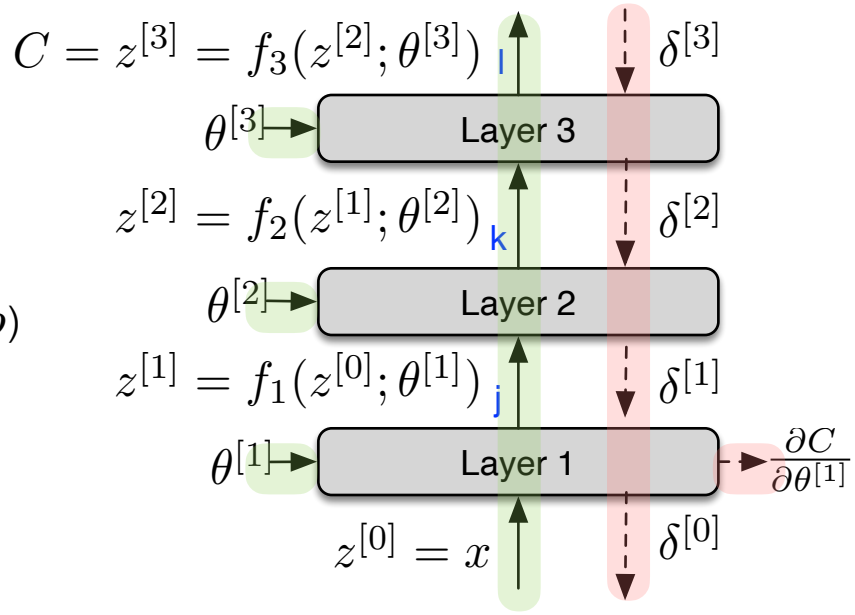
  - a **Backward** method (for inputs i and outputs o)
    - Input

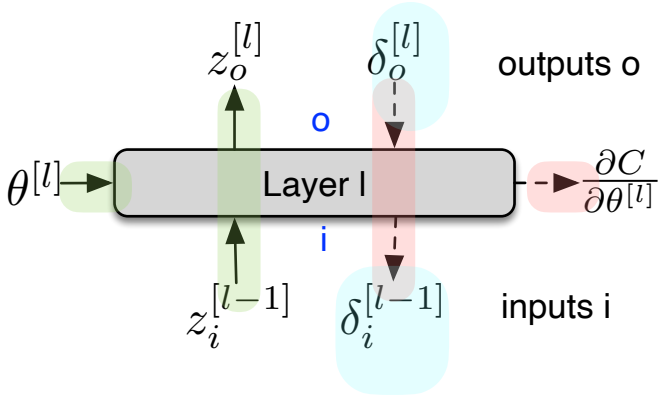
$$\delta_o^{[l]} \triangleq \frac{\partial C}{\partial z_o^{[l]}}$$

    - Computation

$$\delta_i^{[l-1]} = \sum_o \frac{\partial C}{\partial z_o^{[l]}} \frac{\partial z_o^{[l]}}{\partial z_i^{[l-1]}} = \sum_o \delta_o^{[l]} \frac{\partial z_o^{[l]}}{\partial z_i^{[l-1]}}$$

$$\frac{\partial C}{\partial \theta^{[l]}} = \sum_o \frac{\partial C}{\partial z_o^{[l]}} \frac{\partial z_o^{[l]}}{\partial \theta^{[l]}} = \sum_o \delta_o^{[l]} \frac{\partial z_o^{[l]}}{\partial \theta^{[l]}}$$





Overview

Data — Supervised learning — $(x^{(i)}, y^{(i)})$

Model — Architecture

- Linear regression
- Non-linear regression
- Logistic regression
- Softmax regression
- Neural Network
- Deep Learning

Output activation depends on the task
Avoid vanishing gradient
Parameter initialization
Batch Normalization

Cost/Loss — Maximize Likelihood

- Gaussian (MSE)
- Bernoulli (BCE)

Cost depends on the task

Regularization — Avoid over-fitting: L1/L2,DropOut

Optimizer

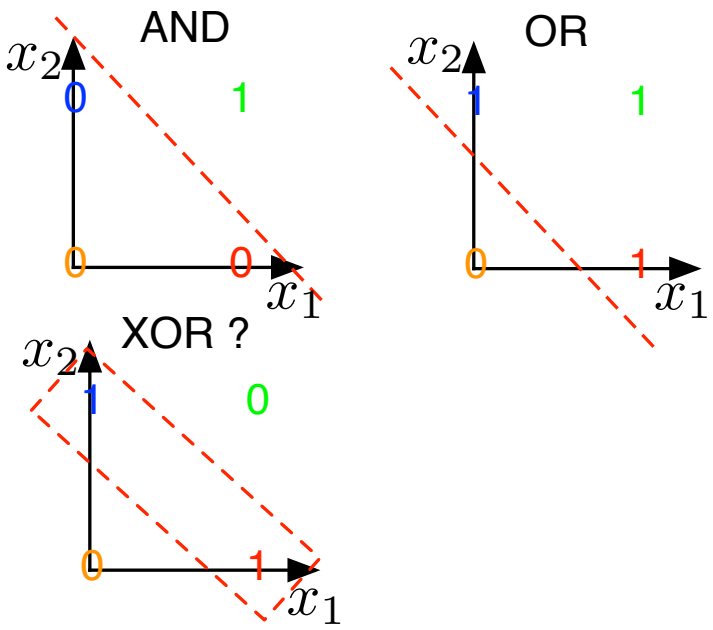
For convex optimization
For non-convex optimization (SGD, Momentum, ...)
First order (SGD), Second order (Hessian)

Training

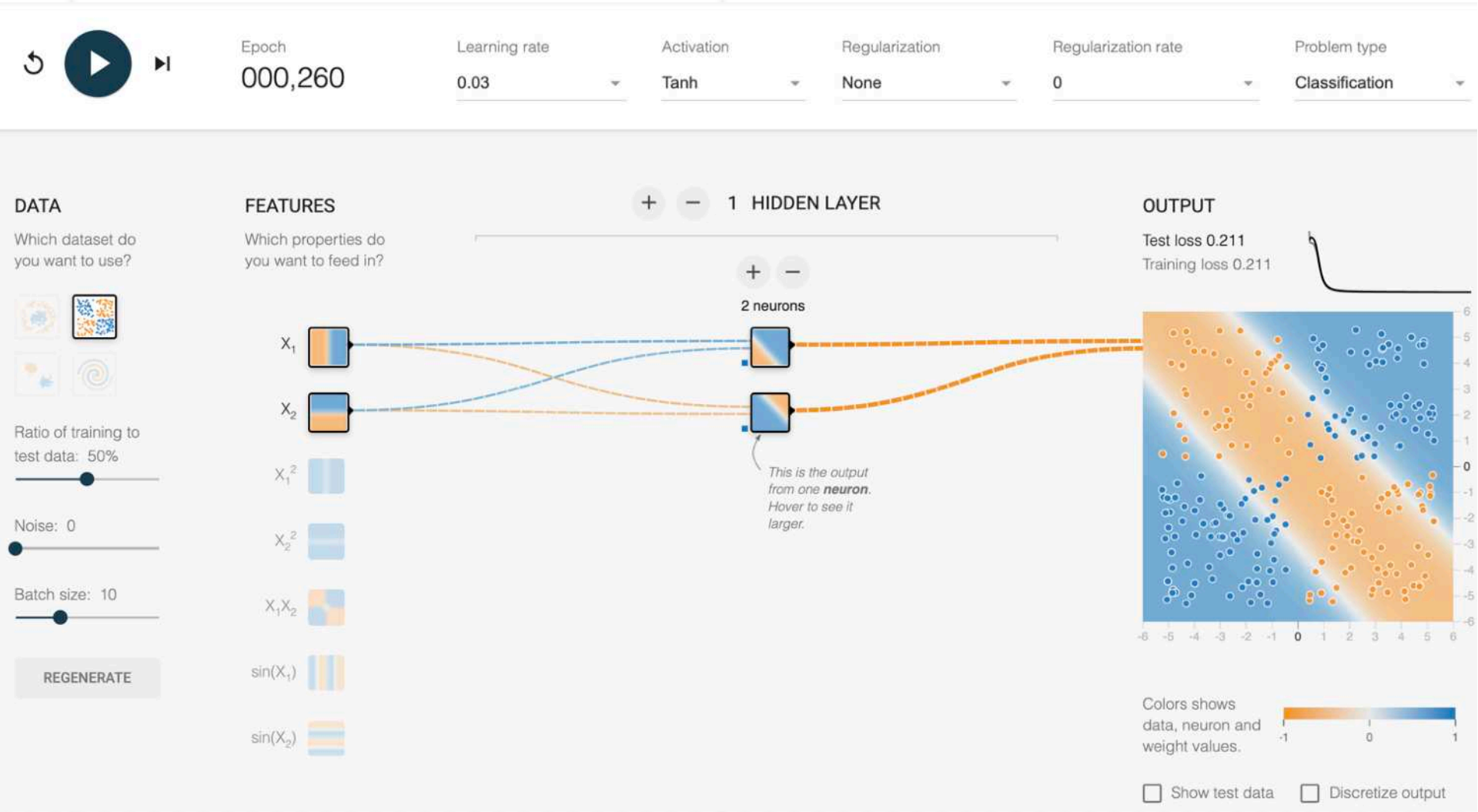
- Dataset/ Dataloader — Get mini-batch
- Forward — Compute prediction $\hat{y}^{(i)}$
- Backward (back-propagation) — Compute gradients
- Update
- Early Stopping — Avoid over-fitting

Limitation of linear classifiers

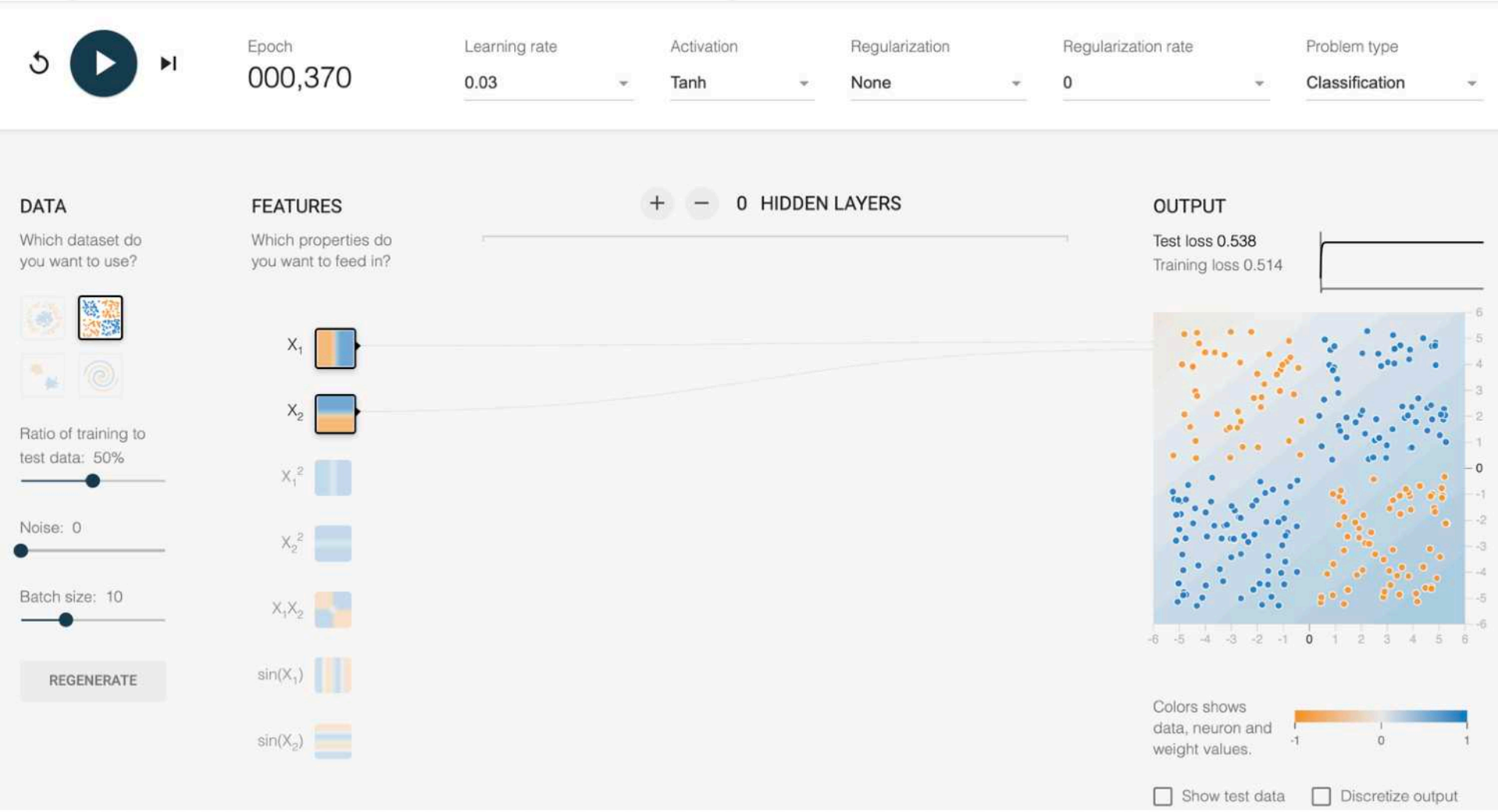
- Perceptron and Logistic Regression are linear classifiers
  - can model AND, OR operators
- What if classes are not linearly separable ?
  - cannot model XOR operator



Logistic Regression (0 hidden layers) illustration in https://playground.tensorflow.org/



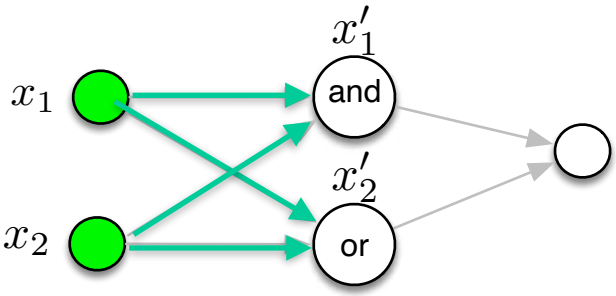
Logistic Regression (0 hidden layers) illustration in https://playground.tensorflow.org/

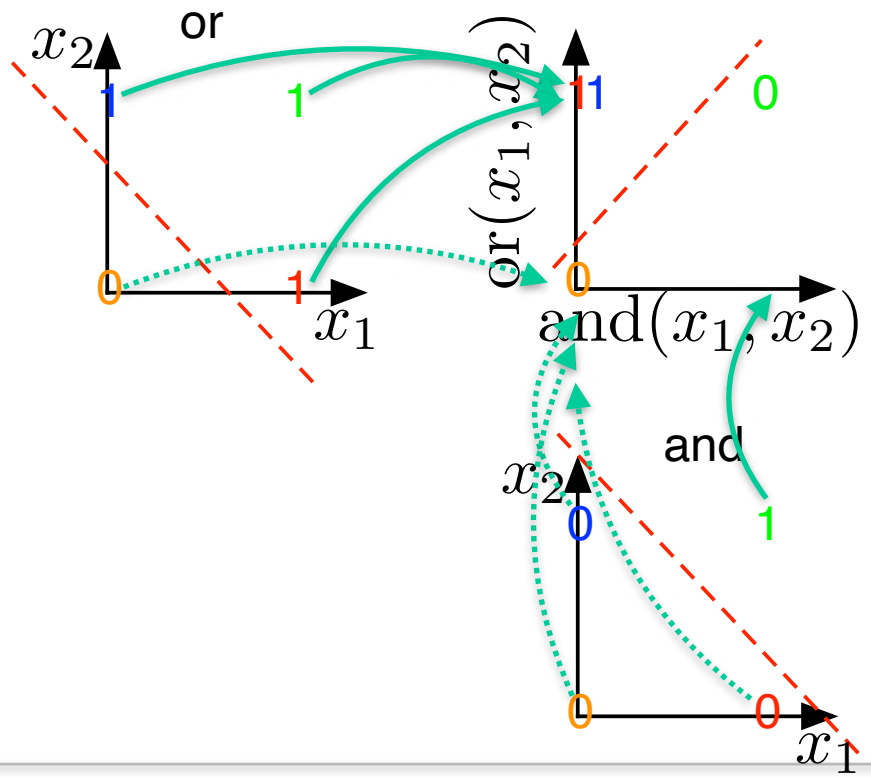


Logistic Regression (0 hidden layers)

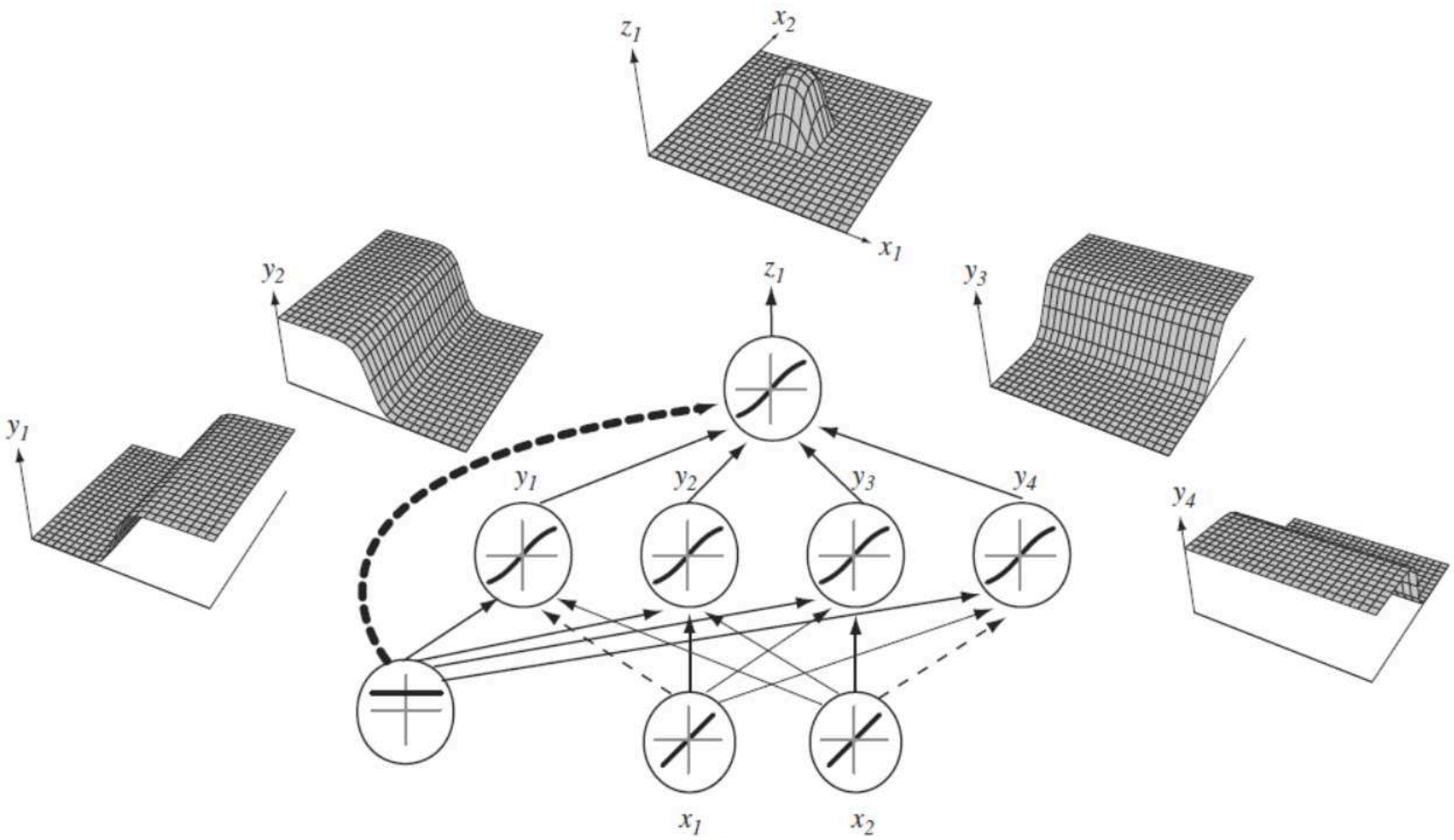
Limitation of linear classifiers

- **Solution to the XOR problem**:
  - project the input data $\mathbf{x}$ in a new space $\mathbf{x}'$
    - $x_1^{[1]} = \textbf{AND}(x_1^{[0]}, x_2^{[0]})$
    - $x_2^{[1]} = \textbf{OR}(x_1^{[0]}, x_2^{[0]})$
- We therefore need one hidden layer of projection
  - $\Rightarrow$ this is a 2 layers **Neural Network**
    $\Rightarrow$1 hidden projection
- In a Neural Network, f_1 and f_2 and trainable projections; they can be many more of such projections



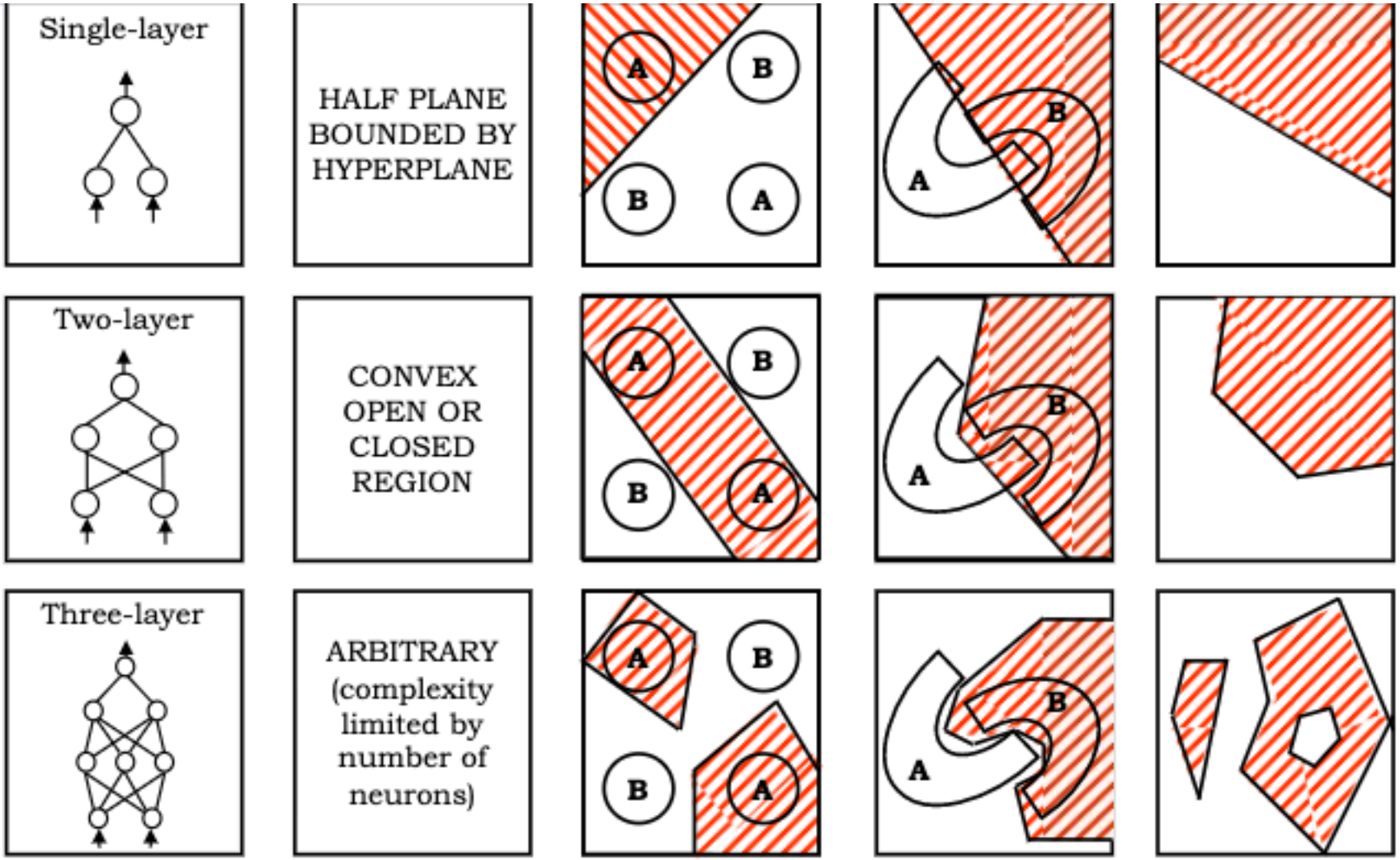


More than one layer



https://ukmie.tistory.com/203

More than one layer



https://ukmie.tistory.com/203

Overview

Data — Supervised learning — $(x^{(i)}, y^{(i)})$

Model — Architecture

- Linear regression
- Non-linear regression
- Logistic regression
- Softmax regression
- Neural Network
- Deep Learning

Output activation depends on the task
Avoid vanishing gradient
Parameter initialization
Batch Normalization

Cost/Loss — Maximize Likelhood

- Gaussian (MSE)
- Bernoulli (BCE)

Cost depends on the task

Regularization — Avoid over-fitting: L1/L2,DropOut

Optimizer

For convex optimization
For non-convex optimization (SGD, Momentum, ...)
First order (SGD), Second order (Hessian)

Training

- **Dataset/ Dataloader** — Get mini-batch
- **Forward** — Compute prediction $\hat{y}^{(i)}$
- **Backward (back-propagation)** — Compute gradients
- **Update**

Early Stopping — Avoid over-fitting

Task: Binary classification
Model: Logistic Regression

1 layer (0 hidden layer) / Binary classification

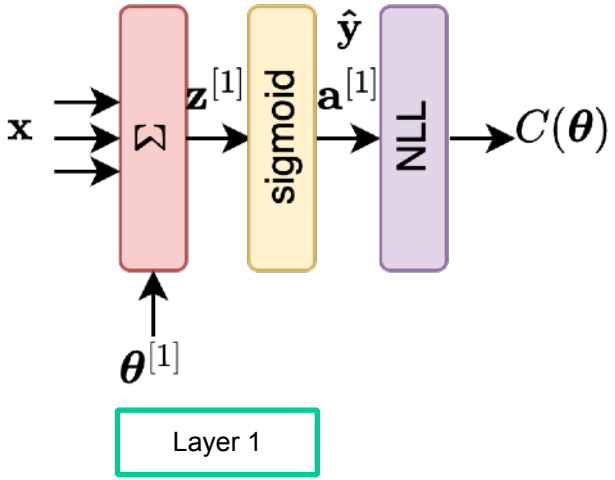
– **Model**

$$z^{[1](i)} = \theta_0^{[1]} + \theta_1^{[1]} x_1^{(i)} + \theta_2^{[1]} x_2^{(i)}$$

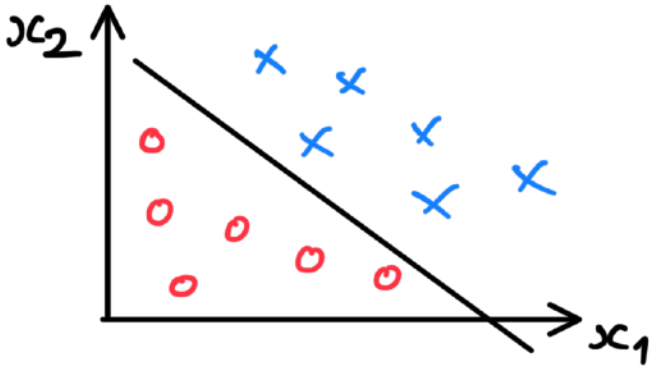
$$a^{[1](i)} = \frac{1}{1 + e^{-z^{[1](i)}}}$$

– **Output**

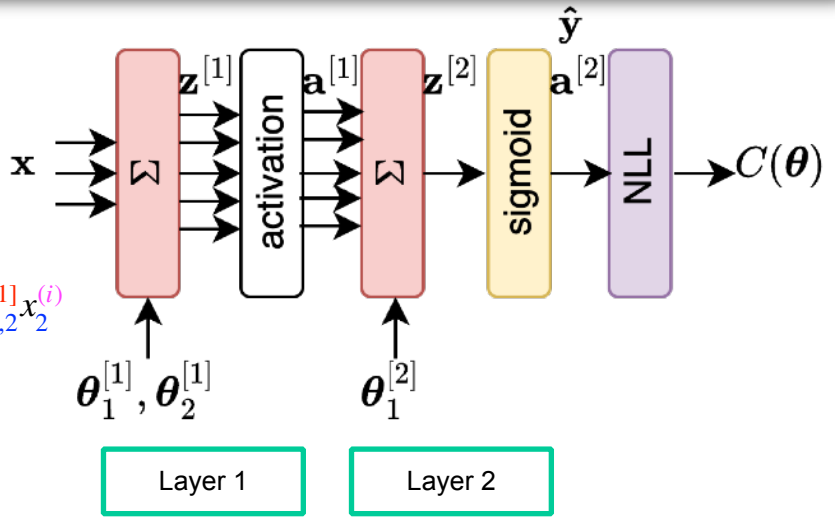
$$\hat{y}^{(i)} = a^{[1]}$$



Linear separating hyper-plane



2 layer (1 hidden layer) / Binary classification



– **Model**

- Layer 1 (hidden)

$$z_1^{[1](i)} = \theta_{0,1}^{[1]} + \theta_{1,1}^{[1]} x_1^{(i)} + \theta_{2,1}^{[1]} x_2^{(i)} \qquad z_2^{[1](i)} = \theta_{0,2}^{[1]} + \theta_{1,2}^{[1]} x_1^{(i)} + \theta_{2,2}^{[1]} x_2^{(i)}$$

$$a_1^{[1](i)} = \frac{1}{1+e^{-z_1^{[1](i)}}} \qquad a_2^{[1](i)} = \frac{1}{1+e^{-z_2^{[1](i)}}}$$

- Layer 2

$$z_1^{[2](i)} = \theta_{0,1}^{[2]} + \theta_{1,1}^{[2]} a_1^{[1](i)} + \theta_{2,1}^{[1]} a_2^{[1](i)}$$

$$a^{[2](i)} = \frac{1}{1+e^{-z_1^{[2](i)}}}$$

– **Output**

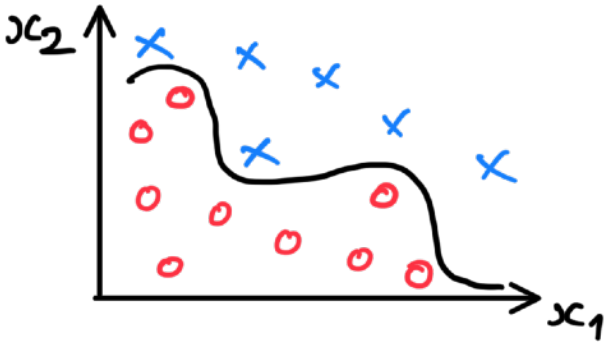
$$\hat{y}^{(i)} = a^{[2]}$$

– **Cost** (to minimise): binary cross-entropy

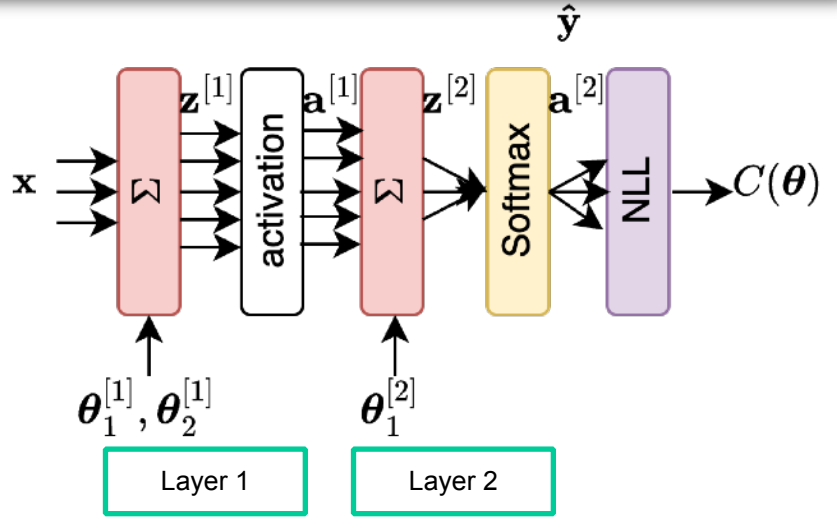
$$C(\theta) = -\log p(y \mid x, \theta)$$

$$= -\sum_{i=1}^{n} y^{(i)} \log \hat{y}^{(i)} + (1-y^{(i)}) \log(1-\hat{y}^{(i)})$$

Non-linear separating hyper-plane



2 layer (1 hidden layer) / Multi-Class classification



– **Output**

$$p(\mathbf{y}^{(i)} = [010] \,|\, \mathbf{x}^{(i)}, \theta) = p(y^{(i)} = 2 \,|\, \mathbf{x}^{(i)}, \theta)$$

$$= \textbf{softmax}(\mathbf{z}_2^{(i)}) = \frac{e^{\mathbf{z}_2^{(i)}}}{e^{\mathbf{z}_1^{(i)}} + e^{\mathbf{z}_2^{(i)}} + e^{\mathbf{z}_3^{(i)}}}$$

– **Likelihood**

$$p(y^{(i)} \,|\, x^{(i)}, \theta) = \left[\frac{e^{\mathbf{z}_1^{(i)}}}{\Sigma}\right]^{\mathbb{I}_1(y^{(i)})} + \left[\frac{e^{\mathbf{z}_2^{(i)}}}{\Sigma}\right]^{\mathbb{I}_2(y^{(i)})} + \left[\frac{e^{\mathbf{z}_3^{(i)}}}{\Sigma}\right]^{\mathbb{I}_3(y^{(i)})}$$

– **Cost** (to minimise) categorical cross-entropy

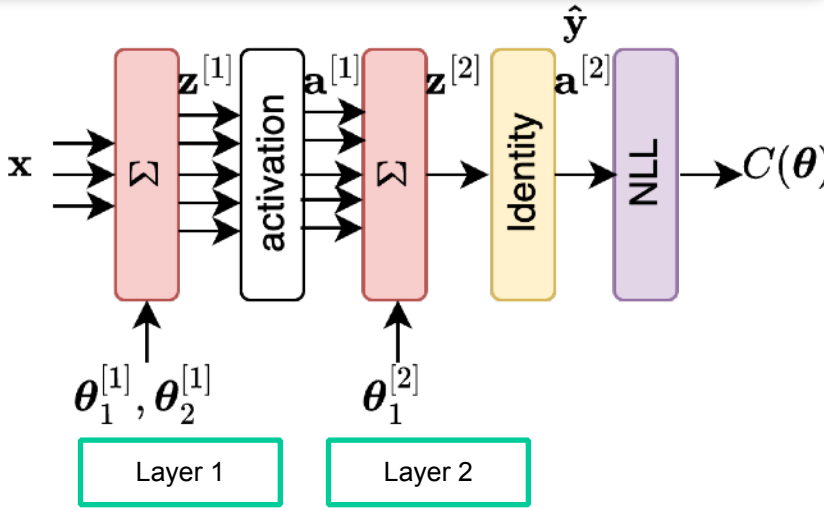
$$C(\theta) = -\log p(y \,|\, x, \theta)$$

$$= -\sum_{i=1}^{n}\sum_{c=1}^{3} \mathbb{I}_c(y^{(i)}) \log \frac{e^{\mathbf{z}c^{(i)}}}{\Sigma}$$

Separating hyper-planes

2 layer (1 hidden layer) / Regression

- Neural networks are function approximation tools



- **Model**

$$\hat{y}^{(i)} = \theta^{[2]}_{0,1} + \theta^{[2]}_{1,1}\frac{1}{1-e^{-(\theta^{[1]}_{0,1}+\theta^{[1]}_{1,1}x^{(i)})}} + \theta^{[2]}_{2,1}\frac{1}{1-e^{-(\theta^{[1]}_{0,2}+\theta^{[1]}_{1,2}x^{(i)})}}$$

- **Output**

$$\hat{y}^{(i)} = \mathbb{E}(y^{(i)}|x^{(i)},\theta)$$
$$= f(x^{(i)},\theta)$$

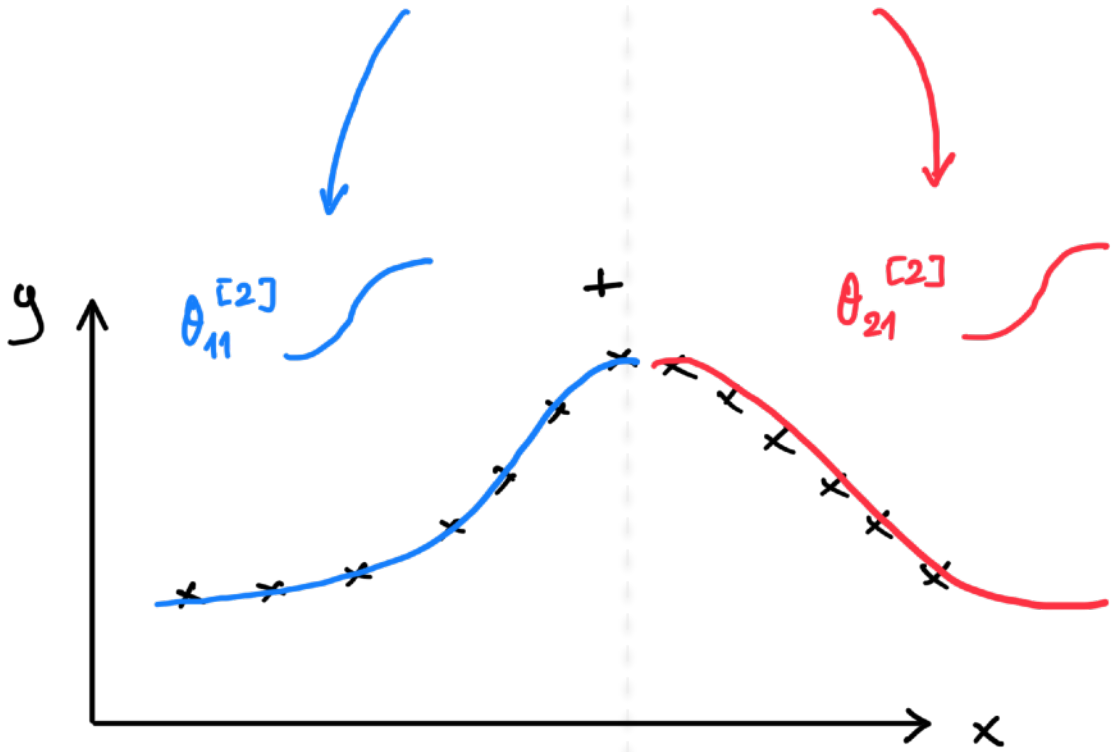
- **Likelihood** (to maximise)

$$p(y^{(i)}|x^{(i)},\theta) = (2\pi\sigma^2)^{-1/2}e^{-\frac{1}{2\sigma^2}(\hat{y}^{(i)}-y^{(i)})^2}$$

- **Cost** (to minimise): Mean Square Error

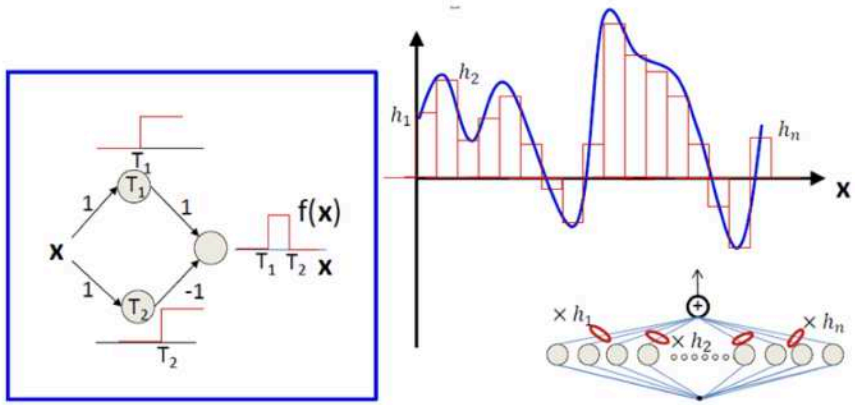
$$C(\theta) = \sum_{i=1}^{n}(y^{(i)}-\hat{y}^{(i)})^2$$

$$\hat{y}^{(i)} = \theta_{0,1}^{[2]} + \theta_{1,1}^{[2]} \frac{1}{1 - e^{-(\theta_{0,1}^{[1]} + \theta_{1,1}^{[1]} x^{(i)})}} + \theta_{2,1}^{[2]} \frac{1}{1 - e^{-(\theta_{0,2}^{[1]} + \theta_{1,2}^{[1]} x^{(i)})}}$$



Neural Network as Universal Function Approximator

- A Neural Network with only one hidden layer can **approximate any function** (with enough hidden neurons)

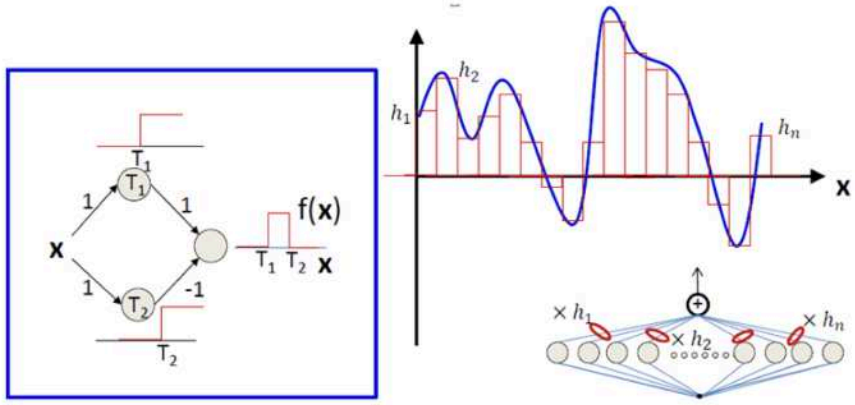


3 unit Multi Layer Perceptron using Step function to approximate a continuous function

https://medium.com/analytics-vidhya/neural-networks-and-the-universal-approximation-theorem-e5c387982eed

Neural Network as Universal Function Approximator

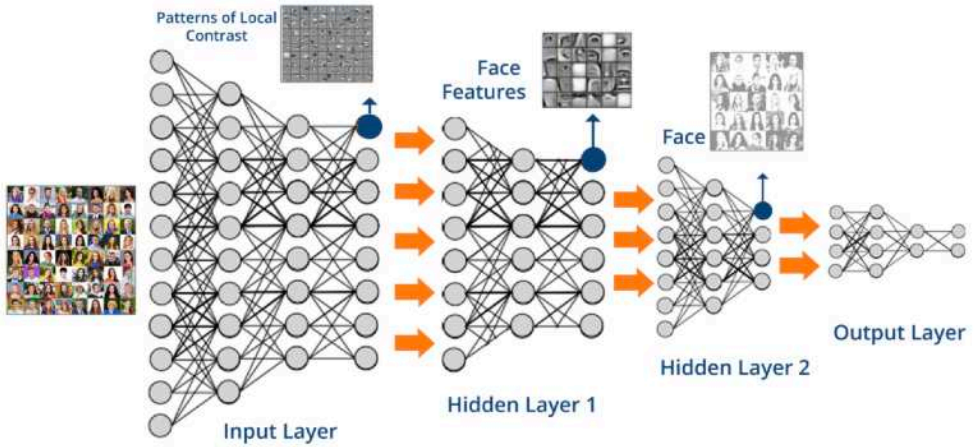
- A Neural Network with only one hidden layer can **approximate any function** (with enough hidden neurons)



3 unit Multi Layer Perceptron using Step function to approximate a continuous function

https://medium.com/analytics-vidhya/neural-networks-and-the-universal-approximation-theorem-e5c387982eed

- Deep Learning is about making the approximation of the function **progressively** (with several hidden layers of fewer neurons)



https://thedatascientist.com/what-deep-learning-is-and-isnt/

Overview

Data — Supervised learning — $(x^{(i)}, y^{(i)})$

Model — Architecture

- Linear regression
- Non-linear regression
- Logistic regression
- Softmax regression
- Neural Network
- Deep Learning

Output activation depends on the task
Avoid vanishing gradient
Parameter initialization
Batch Normalization

Cost/Loss — Maximize Likelihood

- Gaussian (MSE)
- Bernoulli (BCE)

Cost depends on the task

Regularization — Avoid over-fitting: L1/L2,DropOut

Optimizer

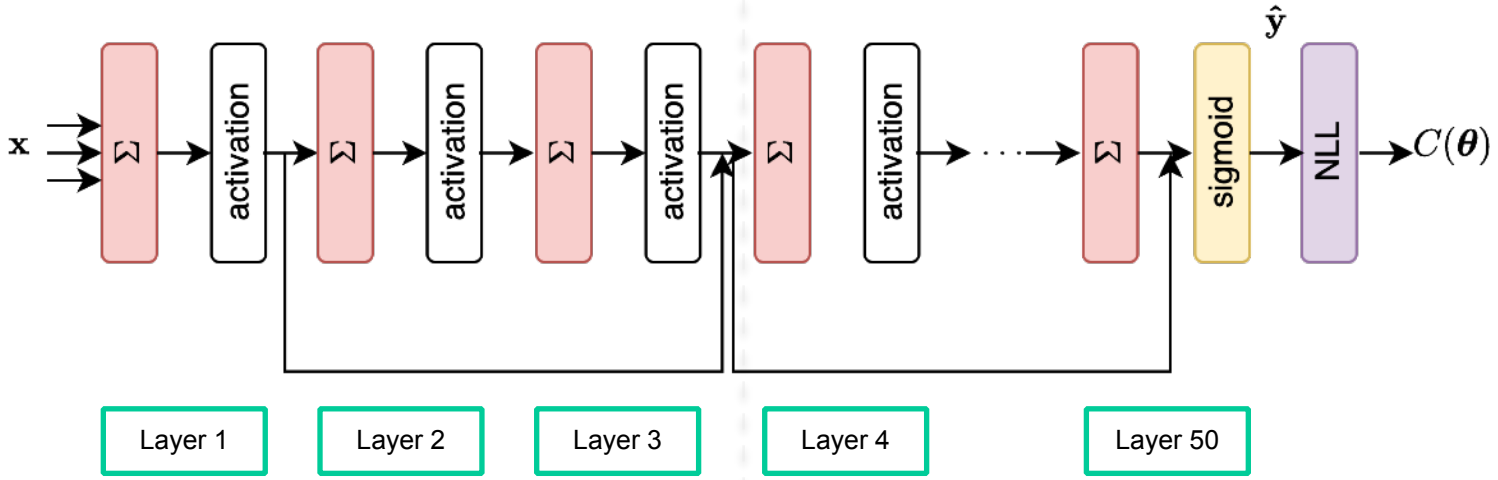
For convex optimization
For non-convex optimization (SGD, Momentum, ...)
First order (SGD), Second order (Hessian)

Training

- Dataset/ Dataloader — Get mini-batch
- Forward — Compute prediction $\hat{y}^{(i)}$
- Backward (back-propagation) — Compute gradients
- Update
- Early Stopping — Avoid over-fitting

Deep Learning

- Deep Learning ?
  - Neural networks with many many layers
  - Sequential, non-sequential, recursive
  - Tricks:
    - vanishing gradient problem (sigmoid, ReLu), faster training, better optimization
    - pre-training, data, GPU



Overview

Data — Supervised learning — $(x^{(i)}, y^{(i)})$

Model — Architecture

- Linear regression
- Non-linear regression
- Logistic regression
- Softmax regression
- Neural Network
- Deep Learning

- Output activation depends on the task
- Avoid vanishing gradient
- Parameter initialization
- Batch Normalization

Cost/Loss — Maximize Likelhood

- Gaussian (MSE)
- Bernoulli (BCE)

Cost depends on the task

Regularization — Avoid over-fitting: L1/L2,DropOut

Optimizer

- For convex optimization
- For non-convex optimization (SGD, Momentum, ...)
- First order (SGD), Second order (Hessian)

Training

- Dataset/ Dataloader — Get mini-batch
- Forward — Compute prediction $\hat{y}^{(i)}$
- Backward (back-propagation) — Compute gradients
- Update
- Early Stopping — Avoid over-fitting

Sigmoid σ

– **Sigmoid function**

$$a = g(z) = \sigma(z) = \frac{1}{1+e^{-z}}$$

– **Derivative:**

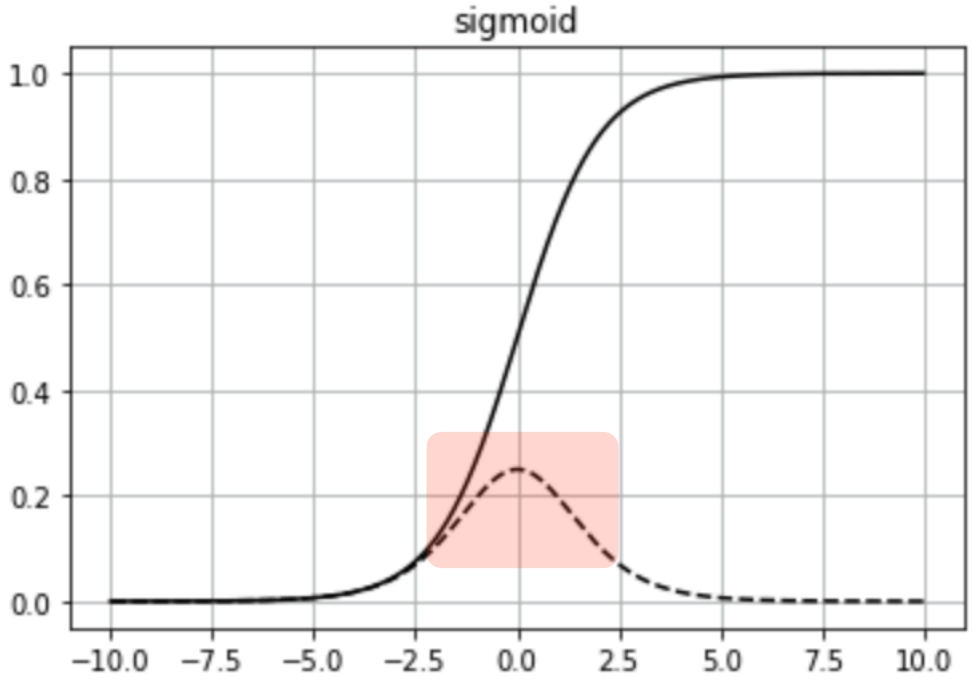
$$\sigma'(z) = \sigma(z)(1-\sigma(z))$$
$$g'(z) = a(1-a)$$

– Proof:

$$\sigma'(z) = --e^{-z}\frac{1}{(1+e^{-z})^2}$$
$$= \frac{1+e^{-z}-1}{(1+e^{-z})^2}$$
$$= \frac{1}{1+e^{-z}}\left(1-\frac{1}{1+e^{-z}}\right)$$
$$= \sigma(z)(1-\sigma(z))$$

– **Other properties:**

$$\sigma(-z) = 1-\sigma(z)$$



maximum value is 0.25 !!!

Activation functions

Hyperbolic tangent function

– **Sigmoid function**

$$a = g(z) = \tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

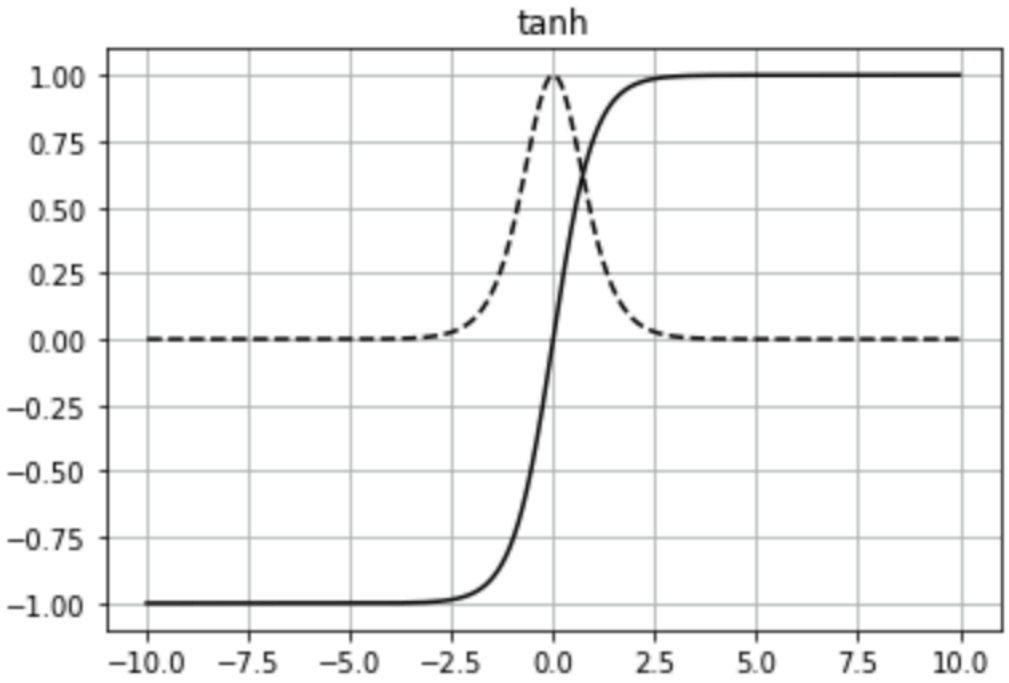
– **Derivative**

$$g'(x) = 1 - \big(\tanh(z)\big)^2$$

$$g'(z) = 1 - a^2$$

– **Other properties:**

$$\tanh(z) = 2\sigma(2z) - 1$$



– **Usage**

- $\tanh(z)$ better than $\sigma(z)$ in middle hidden layers because its mean = zero ($a \in [-1,1]$).

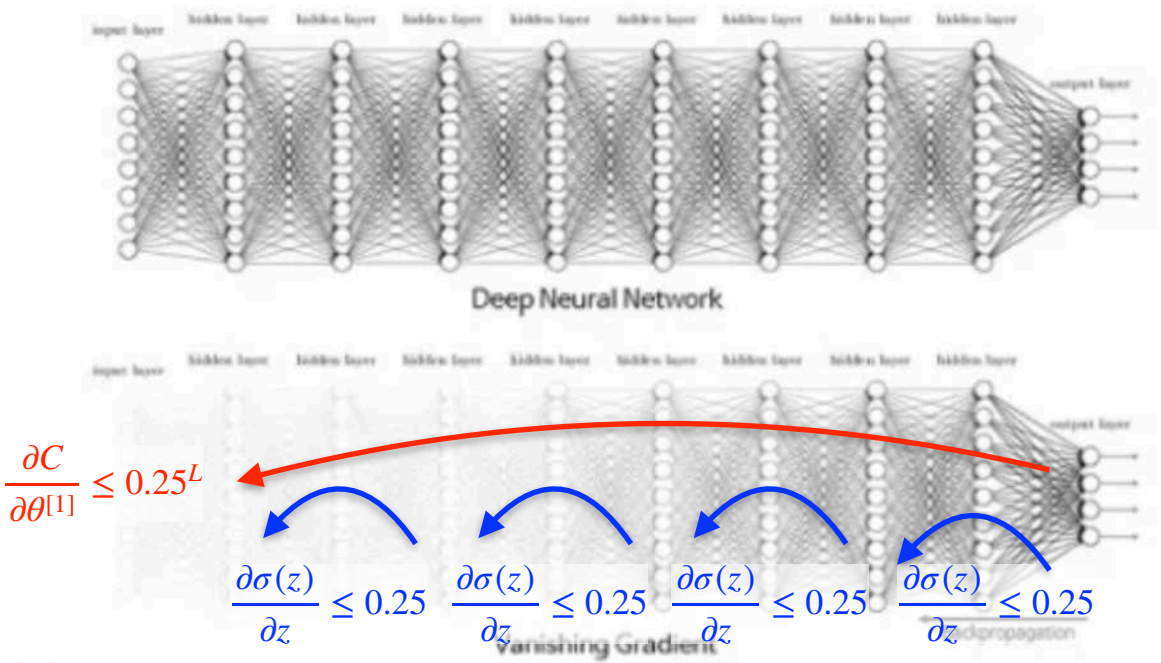
– **Problem** with σ and $\tanh$:

- if z is very small (negative) or very large (positive)
- ⇒ slope becomes zero
- ⇒ slow down Gradient Descent

Activation functions

Vanishing Gradient

- **Problem:** $\max_z \sigma'(z) = 0.25$!!!
- The deeper the network, the fastest the gradient diminishes/vanishes during back-propagation
- For L layers, $\frac{\partial C}{\partial \theta^{[1]}} \propto \frac{\partial \sigma(z^{[L]})}{\partial z^{[L]}} \ldots \frac{\partial \sigma(z^{[L-1]})}{\partial z^{[L-1]}} \ldots \frac{\partial \sigma(z^{[L-2]})}{\partial z^{[L-2]}} \ldots \frac{\partial \sigma(z^{[1]})}{\partial z^{[1]}} \frac{\partial z^{[1]}}{\partial \theta^{[1]}} \leq 0.25^L \simeq 0,$
- there is no gradient anymore $\Rightarrow$we cannot learn $\theta^{[1]}$ using SGD $\Rightarrow$ **the network stop learning !**



https://miro.medium.com/v2/resize:fit:1400/format:webp/1*-rPcuafUSo-SwZF121y_rQ.png

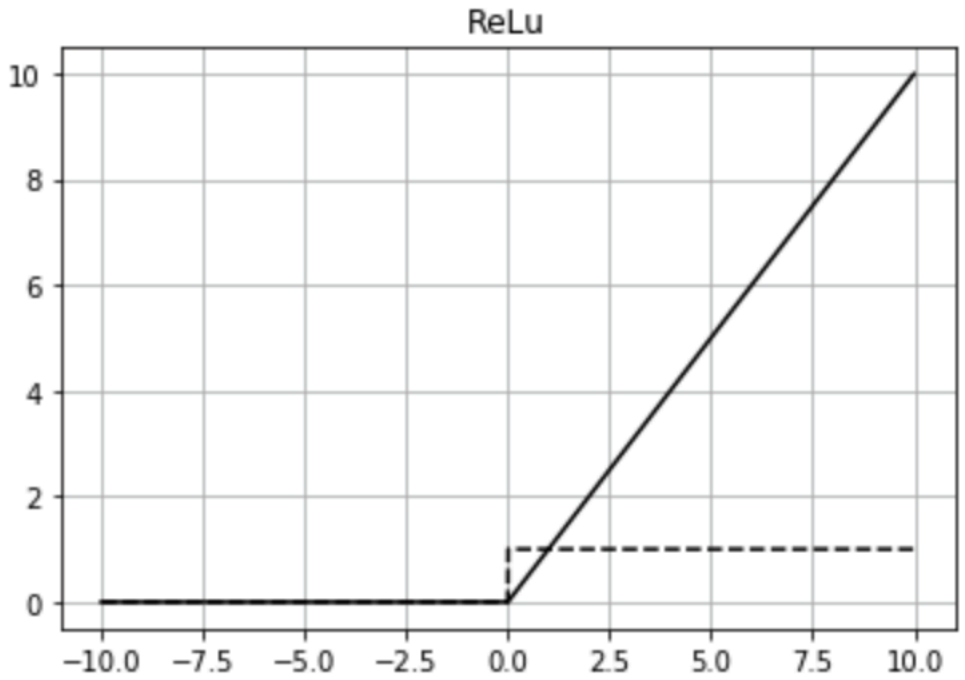
ReLU (Rectified Linear Units)

– **ReLU function**

$$a = g(z) = \max(0,z)$$

– **Derivative**

$$g'(x) = 1 \quad \text{if } z > 0$$
$$= 0 \quad \text{if } z \leq 0$$



_ For L layers, $\frac{\partial C}{\partial \theta^{[1]}} \propto \frac{\partial \text{ReLU}(z^{[L]})}{\partial z^{[L]}} \ldots \frac{\partial \text{ReLU}(z^{[L-1]})}{\partial z^{[L-1]}} \ldots \frac{\partial \text{ReLU}(z^{[L-2]})}{\partial z^{[L-2]}} \ldots \frac{\partial \text{ReLU}(z^{[1]})}{\partial z^{[1]}} \frac{\partial z^{[1]}}{\partial \theta^{[1]}} \leq 1^L$,

Overview

Data — Supervised learning — $(x^{(i)}, y^{(i)})$

Model — Architecture

- Linear regression
- Non-linear regression
- Logistic regression
- Softmax regression
- Neural Network
- Deep Learning

- Output activation depends on the task
- Avoid vanishing gradient
- Parameter initialization
- Batch Normalization

Cost/Loss — Maximize Likelihood

- Gaussian (MSE)
- Bernoulli (BCE)

Cost depends on the task

Regularization — Avoid over-fitting: L1/L2,DropOut

Optimizer

- For convex optimization
- For non-convex optimization (SGD, Momentum, ...)
- First order (SGD), Second order (Hessian)

Training

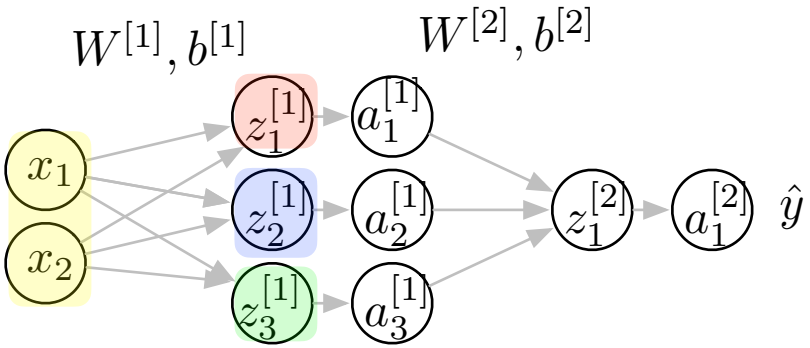
- Dataset/ Dataloader — Get mini-batch
- Forward — Compute prediction $\hat{y}^{(i)}$
- Backward (back-propagation) — Compute gradients
- Update
- Early Stopping — Avoid over-fitting

Weight initialisation

Initialise W with all zeros ?

$$\mathbf{W}^{[1]} = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

$$\mathbf{W}^{[2]} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$$



- then for any input $\Rightarrow a_1^{[1]} = a_2^{[1]} = a_3^{[1]}$
  - the three hidden units are computing exactly the same function
    - they are symmetric

$$\mathbf{z}^{[1]} = \mathbf{a}^{[0]}\mathbf{W}^{[1]}$$

$$\begin{pmatrix} z_1^{[1]} & z_2^{[1]} & z_3^{[1]} \end{pmatrix} = \begin{pmatrix} a_1^{[0]} & a_2^{[0]} \end{pmatrix} \begin{pmatrix} w_{11}^{[1]} & w_{12}^{[1]} & w_{13}^{[1]} \\ w_{21}^{[1]} & w_{22}^{[1]} & w_{23}^{[1]} \end{pmatrix}$$

$$\mathbf{z}^{[2]} = \mathbf{a}^{[1]}\mathbf{W}^{[2]}$$

$$z_1^{[2]} = \begin{pmatrix} a_1^{[1]} & a_2^{[1]} & a_3^{[1]} \end{pmatrix} \begin{pmatrix} w_{11}^{[2]} \\ w_{21}^{[2]} \\ w_{31}^{[2]} \end{pmatrix}$$

Weight initialisation with random values

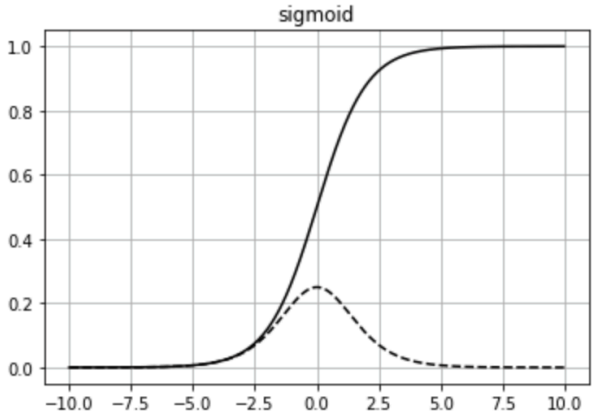
- **Random initialisation**
  - $\mathbf{W}^{[1]}$ = np.random.randn((2,3))*0.01
  - $\mathbf{b}^{[1]}$= 0
  - $\mathbf{W}^{[2]}$ = np.random.randn((3,1))*0.01
  - $\mathbf{b}^{[2]}$ = 0
- Remark: $\mathbf{b}$ doesn't have the symmetry problem

- **Why 0.01 ?**
  - If $\mathbf{W}$ is big $\Rightarrow$ Z is also big

$$\mathbf{Z}^{[1]} = \mathbf{X}\,\mathbf{W}^{[1]} + \mathbf{b}^{[1]}$$

$$\mathbf{A}^{[1]} = g^{[1]}(\mathbf{Z}^{[1]})$$

  - $\Rightarrow$we are in the flat part of the sigmoid/tanh
    - $\Rightarrow$ slope is small
    - $\Rightarrow$ gradient descent slow
    - $\Rightarrow$ learning slow
  - Better to initialise to a very small value (valid for $\sigma(z)$ and $\tanh(z)$)



Weight initialisation for Deep Neural Networks

- Suppose a single neuron network

  $z = w_1 x_1 + w_2 x_2 + \ldots + w_n x_n + b$

- In order to avoid vanishing/exploding gradient) \AR
  - The larger n is $\Rightarrow$the smallest w_d should be
- Solution ?
  - set Var$(w_i) = 1/n$
- In practice

  $\mathbf{W}^{[l]}$ = np.random.randn(shape) * np.sqrt $\left(1/n^{[l-1]}\right)$
- Other possibilities
  - For a ReLu: Var$(w_i) = \frac{2}{n}$works a bit better
  - For a $\tanh$
    - Var$(w_i) = \frac{1}{n^{[l-1]}}$: Xavier initialisation
    - Var$(w_i) = \frac{2}{n^{[l-1]} n^{[l]}}$: Bengio initialisation

Overview

Data — Supervised learning — $(x^{(i)}, y^{(i)})$

Model — Architecture

- Linear regression
- Non-linear regression
- Logistic regression
- Softmax regression
- Neural Network
- Deep Learning

- Output activation depends on the task
- Avoid vanishing gradient
- Parameter initialization
- Batch Normalization

Cost/Loss — Maximize Likelihood

- Gaussian (MSE)
- Bernoulli (BCE)

Cost depends on the task

Regularization — Avoid over-fitting: L1/L2,DropOut

Optimizer

- For convex optimization
- For non-convex optimization (SGD, Momentum, ...)
- First order (SGD), Second order (Hessian)

Training

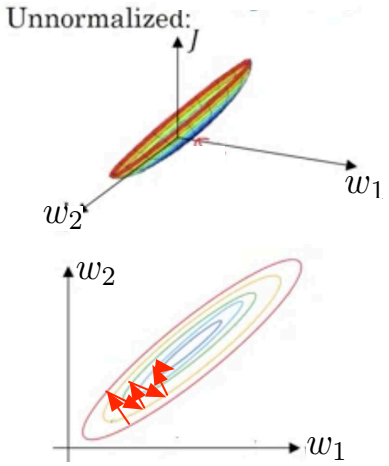
- Dataset/ Dataloader — Get mini-batch
- Forward — Compute prediction $\hat{y}^{(i)}$
- Backward (back-propagation) — Compute gradients
- Update
- Early Stopping — Avoid over-fitting

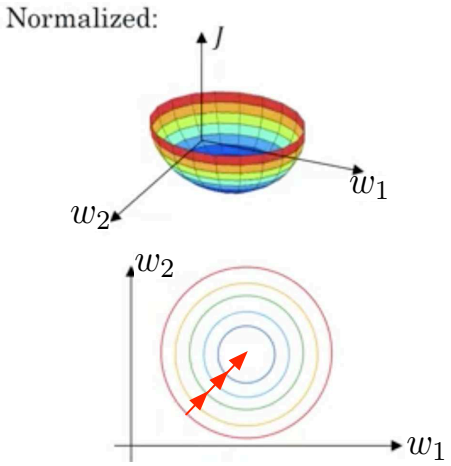
Normalising the inputs

- Suppose: $x_1 \in [1 \ldots 1000]$ and $x_2 \in [0 \ldots 1]$
  - Then w_1 and $w_2$will take very different value

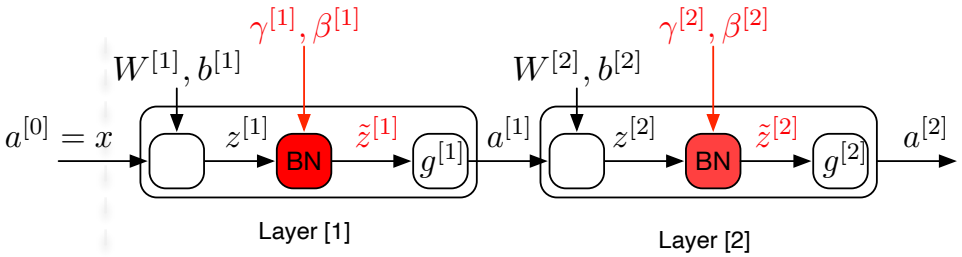
- if no normalisation
  - many oscillations
  - need to use a small learning rate

- **Why use input normalisation ?**
  - get similar values for w_1 and w_2
    - gradient descent can go straight to the minimum
    - can use large learning rate
  - avoid each activation to be large (see sigmoid activation)
  - numerical instability





Batch Normalisation (BN)



– **Objective ?**
  - Apply the same normalisation for the input of each layer $[l]$
  - allows to learn faster

– Try to **reduce the "covariate shift"**
  - the inputs of a given layer $[l]$ is the outputs of the previous layer $[l-1]$
  - these outputs $[l-1]$ depends on the parameters of the previous layer which change over training !
  - **normalise the output of the previous layer** $a^{[l-1]}$
  - in practice normalise the pre-activation $z^{[l-1]}$

– Don't want all units to always have mean 0 and standard-deviation 1
  - Learn an appropriate bias β and scale γ to apply to $z^{[l-1]}$ before the non-linear function $g^{[l]}$

[Sergey Ioffe and Christian Szegedy, "Batch Normalization : Accelerating Deep Network Training by Reducing Internal Covariate Shift", ICML, 2015] LINK

Overview

Data	Supervised learning	$(x^{(i)}, y^{(i)})$	
Model	Architecture	Linear regression Non-linear regression Logistic regression Softmax regression Neural Network Deep Learning	Output activation depends on the task Avoid vanishing gradient Parameter initialization Batch Normalization
Cost/Loss	Maximize Likelhood	Gaussian (MSE) Bernoulli (BCE)	Cost depends on the task
Regularization	Avoid over-fitting: L1/L2,DropOut		
Optimizer	For convex optimization For non-convex optimization (SGD, Momentum, ...) First order (SGD), Second order (Hessian)		
Training			
Dataset/ Dataloader		Get mini-batch	
Forward		Compute prediction $\hat{y}^{(i)}$	
Backward (back-propagation)		Compute gradients	
Update			
Early Stopping		Avoid over-fitting	

Optimisations
Variations

- Rewriting SGD as sum:

$$\theta_{k+1} = \theta_k - \eta_k \cdot \mathbb{E}[\nabla_\theta J(\theta_k)]$$

- Variations of SGD
  - **Batch** (all m data points)

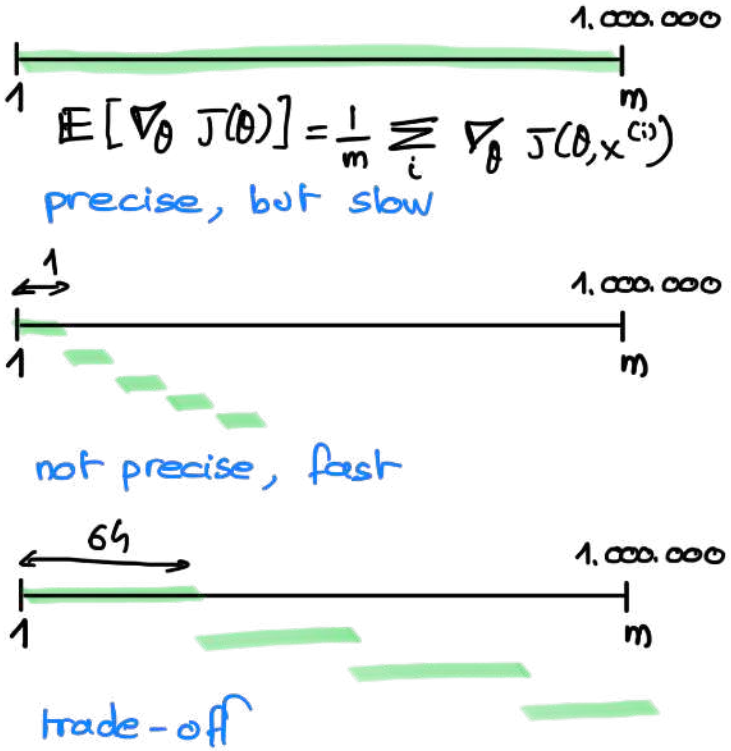
$$\theta_{k+1} = \theta_k - \eta_k \sum_{i=1}^{m} \nabla_\theta J(\theta_k, \mathbf{x}^{(i)})$$

  - **Stochastic** (a single data point)

$$\theta_{k+1} = \theta_k - \eta_k \nabla_\theta J(\theta_k, \mathbf{x}^{(i)})$$

  - **Mini-Batch** (a subset of data points)

$$\theta_{k+1} = \theta_k + \eta_k \sum_{j=1}^{64} \nabla_\theta J(\theta_k, \mathbf{x}^{(i)})$$



Optimisation
Why S stands for "Stochastic" in SGD ?

- Stochastic ?
  - because we only have seen a fraction of the data, uncertainty

$$\nabla_\theta f(\theta) = \underbrace{\int \nabla_\theta f(x,\theta) p(x) dx}_{\mathbb{E}[\nabla_\theta f(x,\theta)]} \quad \text{we want =0}$$

- Can be re-written

$$\begin{aligned}
\mathbb{E}[\nabla_\theta f(x,\theta)] &= 0 \\
\eta \mathbb{E}[\nabla_\theta f(x,\theta)] &= 0 \cdot \eta = 0 \\
\theta + \eta \mathbb{E}[\nabla_\theta f(x,\theta)] &= \theta \\
\theta_{k+1} &= \theta_k - \eta \mathbb{E}[\nabla_\theta f(x,\theta)] \\
&\simeq \theta_k - \eta \frac{1}{m} \sum_{i=1}^{m} \nabla_\theta f(x^{(i)},\theta) \text{ with } x^{(i)} \sim p(x) \\
&\simeq \theta_k - \eta \nabla_\theta f(x^{(k)},\theta_k)
\end{aligned}$$

- Therefore

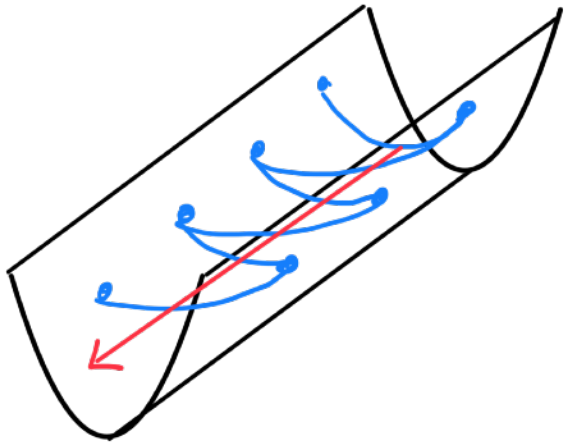
$$\underbrace{\theta_{k+1} = \theta_k - \eta \mathbb{E}[\nabla_\theta f(x,\theta)]}_{\text{what we want}} + \underbrace{\eta \left[\mathbb{E}[\nabla_\theta f(x,\theta)] - \nabla_\theta f(x^{(k)},\theta_k)\right]}_{\text{noise, changes over time, stochastic, bounded}}$$

- **Momentum** In this slide we denote by θ^k the value of θ at iteration k
  - Goal: accelerate descent in cumulated direction

Normal SGD:

$$\theta^{k+1} = \theta^k - \eta_k \mathbf{g}_k$$
$$= \theta^k - \eta_k \nabla_\theta f(\theta^k)$$

$$\theta^{k+1} = \theta^k + \alpha(\theta^k - \theta^{k-1}) + (1-\alpha)[-\eta \nabla J(\theta)]$$
$$\underbrace{\theta^{k+1} - \theta^k}_{\Delta\theta^{k+1}} = \alpha \underbrace{(\theta^k - \theta^{k-1})}_{\Delta\theta^k} + (1-\alpha)[-\eta \nabla J(\theta)]$$



- **AdaGrad**
  - Goal: put more weights on rare features
  - For feature d at step k

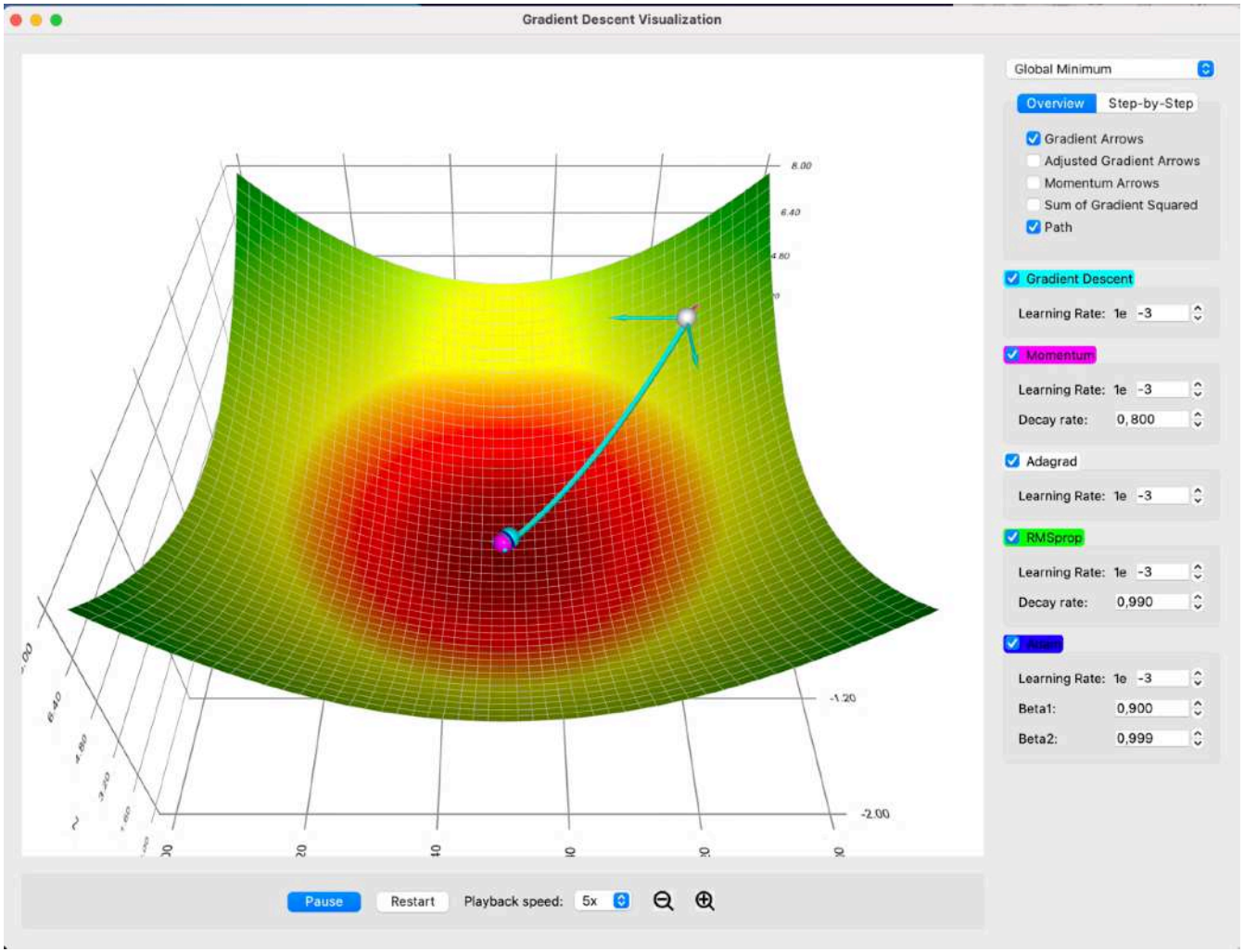
$$\theta_d^{k+1} = \theta_d^k - \frac{\eta}{\sum_{\tau=1}^{k} (g_d^\tau)^2} g_d^k$$

  - $\sum_{\tau=1}^{k} (g_d^\tau)^2$ is the history of how much the feature θ_d has changed in the past
  - if small (has been seen few times in the past) then put more weights

- **ADAM**
  - Momentum + AdaGrad

Alternatives to gradient descent

Example with shape = global minimum

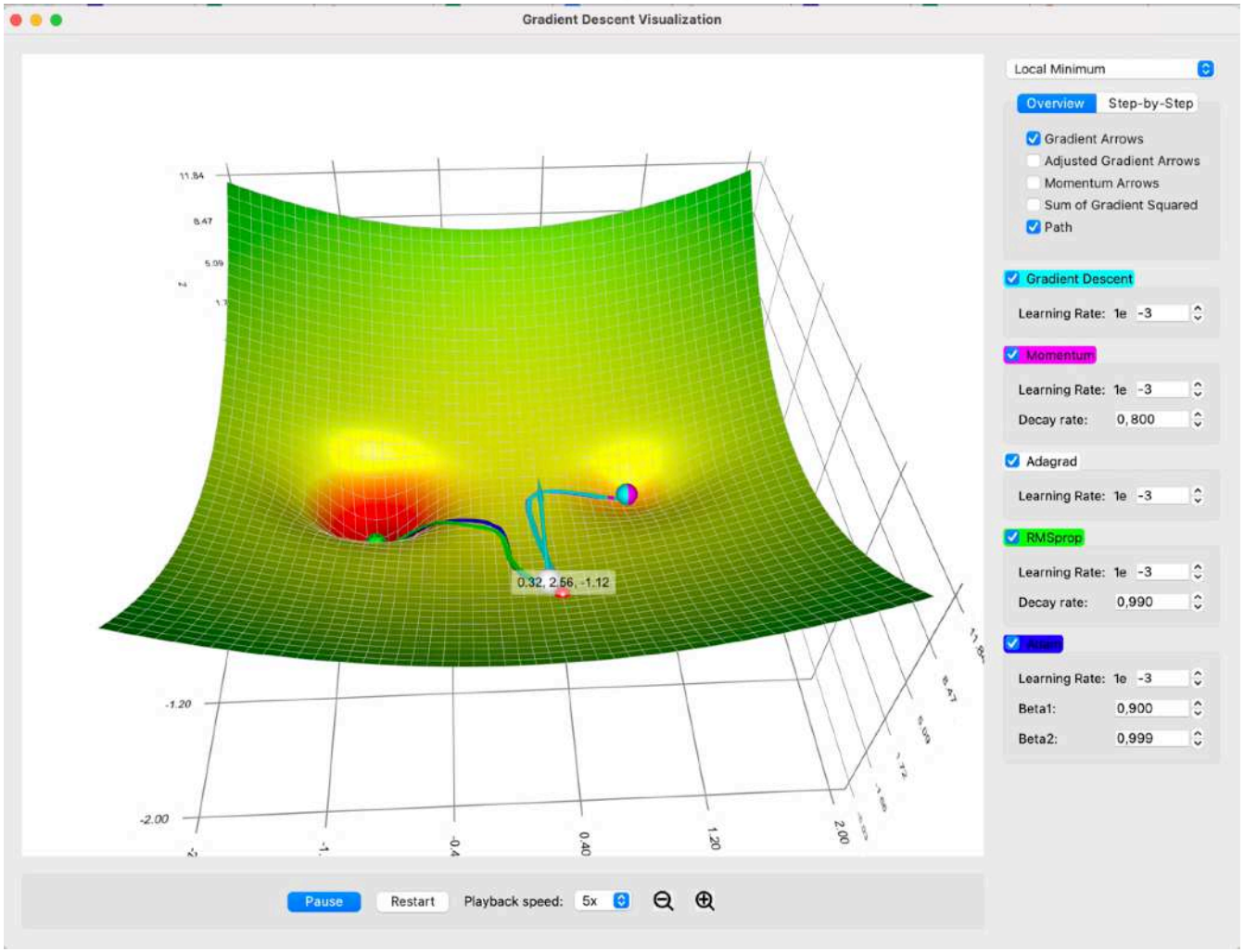


https://towardsdatascience.com/a-visual-explanation-of-gradient-descent-methods-momentum-adagrad-rmsprop-adam-f898b102325c

https://ruder.io/optimizing-gradient-descent/

Alternatives to gradient descent

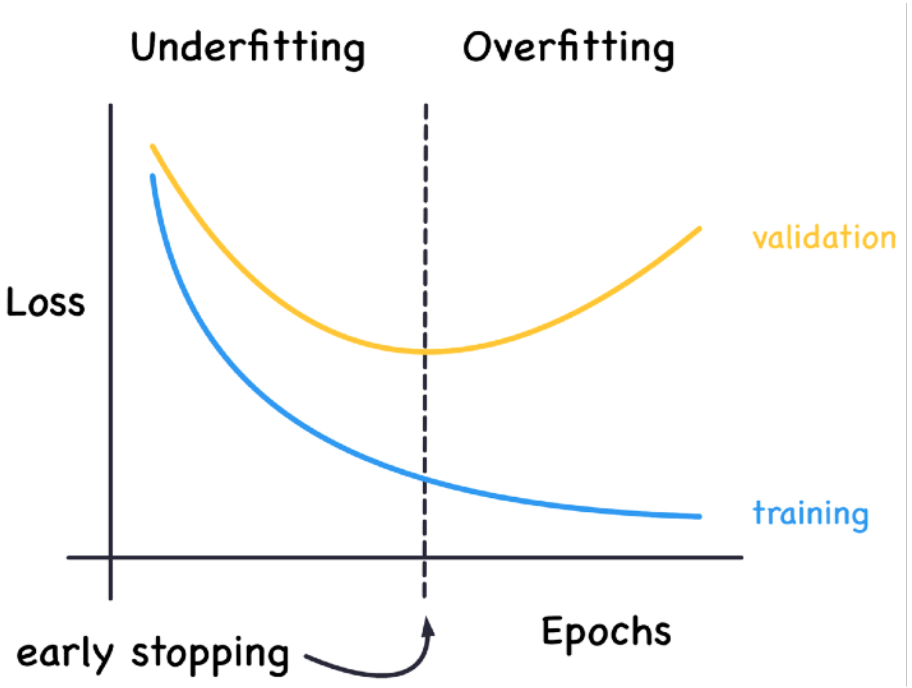
Example with shape = local minimum



https://towardsdatascience.com/a-visual-explanation-of-gradient-descent-methods-momentum-adagrad-rmsprop-adam-f898b102325c

Early stopping

- Split the data into train/ **validation**/ test
- **Monitor** the loss on the validation data
  - can also monitor the accuracy
- Stop the training when the loss stop decreasing on the validation data
  - "**patience**" parameter: defines how many epochs the model will continue training after the best validation performance has been seen.

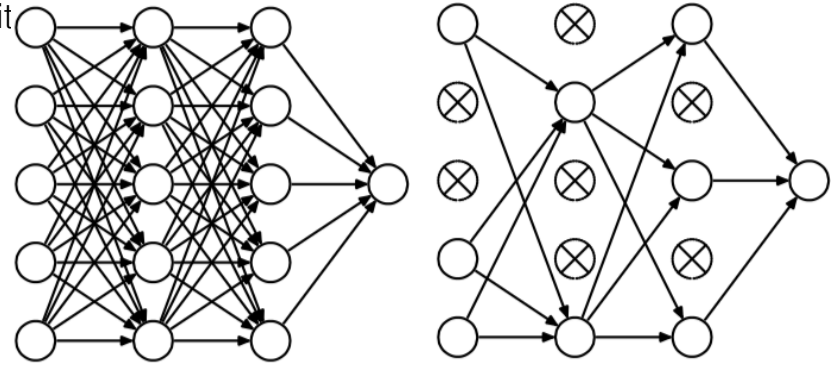


Overview

Data	Supervised learning	$(x^{(i)}, y^{(i)})$	
Model	Architecture	Linear regression Non-linear regression Logistic regression Softmax regression Neural Network Deep Learning	Output activation depends on the task Avoid vanishing gradient Parameter initialization Batch Normalization
Cost/Loss	Maximize Likelihood	Gaussian (MSE) Bernoulli (BCE)	Cost depends on the task
Regularization	Avoid over-fitting: L1/L2,DropOut		
Optimizer	For convex optimization For non-convex optimization (SGD, Momentum, ...) First order (SGD), Second order (Hessian)		
Training			
Dataset/ Dataloader		Get mini-batch	
Forward		Compute prediction $\hat{y}^{(i)}$	
Backward (back-propagation)		Compute gradients	
Update			
Early Stopping		Avoid over-fitting	

DropOut regularisation

- **During training**
  - for each training example **randomly turn-off** the neurons of hidden units (with $p = 0.5$)
    - this also removes the connections
  - for different training examples, turn-off different units
  - possible to vary the probability across layers
    - for large matrix $\mathbf{W} \Rightarrow p$ is higher
    - for small matrix $\mathbf{W} \Rightarrow p$ is lower
- **During testing**
  - no DropOut
- **DropOut effects:**
  - prevents co-adaptation between units
  - can be seen as averaging different models that share parameters
  - acts as a powerful regularisation scheme
  - since the network is smaller, it is easier to train (as regularisation)
  - The network cannot rely on any feature, it has to spread out weights
    - Effect: shrinking the squared norm of the weights (similar to L2 regularisation)
    - Can be shown to be an adaptive form of L2-regularisation



[Hinton, Geoffrey E.; Srivastava, Nitish; Krizhevsky, Alex; Sutskever, Ilya; Salakhutdinov, Ruslan R. (2012). "Improving neural networks by preventing co-adaptation of feature detectors"] LINK

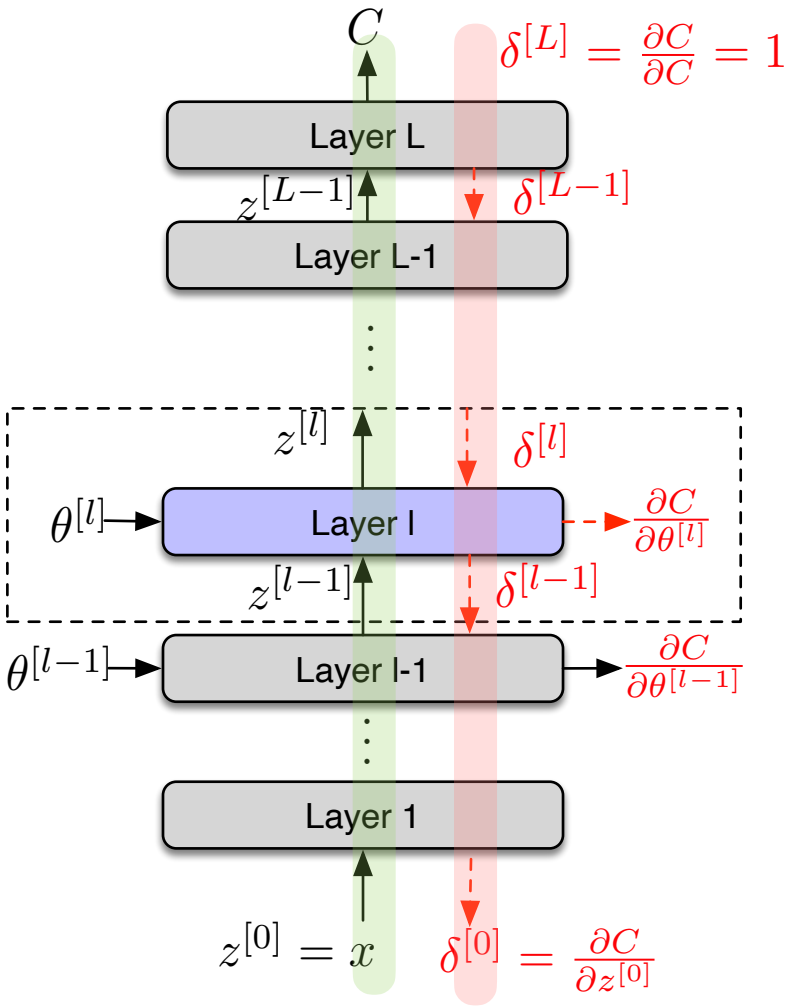
Overview

Data	Supervised learning	$(x^{(i)}, y^{(i)})$	
Model	Architecture	Linear regression Non-linear regression Logistic regression Softmax regression Neural Network Deep Learning	Output activation depends on the task Avoid vanishing gradient Parameter initialization Batch Normalization
Cost/Loss	Maximize Likelihood	Gaussian (MSE) Bernoulli (BCE)	Cost depends on the task
Regularization	Avoid over-fitting: L1/L2,DropOut		
Optimizer	For convex optimization For non-convex optimization (SGD, Momentum, ...) First order (SGD), Second order (Hessian)		
Training			

Dataset/ Dataloader	Get mini-batch
Forward	Compute prediction $\hat{y}^{(i)}$
Backward (back-propagation)	Compute gradients
Update	
Early Stopping	Avoid over-fitting

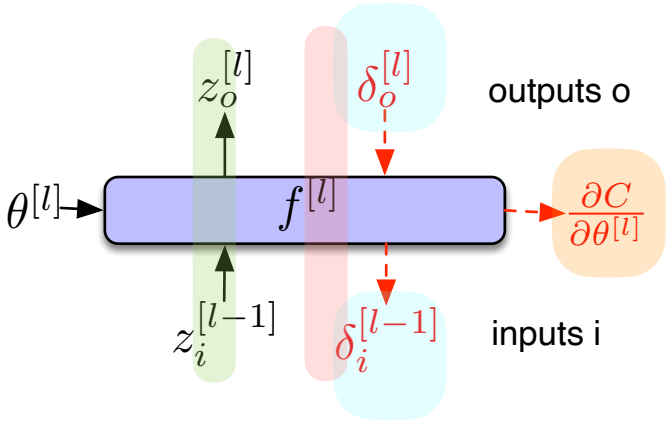
Deep Learning
Forward/ Backward propagation

- **Forward**
- **Backward**



Deep Learning
Forward/ Backward propagation

- **Layer** l
  - Input: $\mathbf{z}^{[l-1]}$
  - Output: $\mathbf{z}^{[l]}$



- **Forward**

$$\mathbf{z}^{[l]} = f^{[l]}(\mathbf{z}^{[l-1]}, \theta^{[l]})$$

- **Backward**
  - Gradients to propagate back
    - matrix form

$$\delta^{[l-1]} = \frac{\partial C}{\partial \mathbf{z}^{[l-1]}} = \frac{\partial C}{\partial \mathbf{z}^{[l]}} \frac{\partial \mathbf{z}^{[l]}}{\partial \mathbf{z}^{[l-1]}} = \delta^{[l]} \frac{\partial \mathbf{z}^{[l]}}{\partial \mathbf{z}^{[l-1]}}$$

      - where $\frac{\partial \mathbf{z}^{[l]}}{\partial \mathbf{z}^{[l-1]}}$ is a Jacobian matrix

    - scalar form

$$\delta_i^{[l-1]} = \sum_o \frac{\partial C}{\partial z_o^{[l]}} \frac{\partial z_o^{[l]}}{\partial z_i^{[l-1]}}$$

$$= \sum_o \delta_o^{[l]} \frac{\partial [f_o^{[l]}(\mathbf{z}l-1, \theta l)]}{\partial z_i^{[l-1]}}$$

  - Local gradients for parameters updating
    - matrix form

$$\frac{\partial C}{\partial \theta^{[l]}} = \frac{\partial C}{\partial \mathbf{z}^{[l]}} \frac{\partial \mathbf{z}^{[l]}}{\partial \theta^{[l]}} = \delta^{[l]} \frac{\partial f^{[l]}(\mathbf{z}^{[l-1]}, \theta^{[l]})}{\partial \theta^{[l]}}$$

    - scalar form

$$\frac{\partial C}{\partial \theta_i^{[l]}} = \sum_o \frac{\partial C}{\partial z_o^{[l]}} \frac{\partial z_o^{[l]}}{\partial \theta_i^{[l]}}$$

$$= \sum_o \delta_o^{[l]} \frac{\partial [f_o^{[l]}(\mathbf{z}l-1, \theta l)]}{\partial \theta_i^{[l]}}$$

Deep Learning
Linear layer

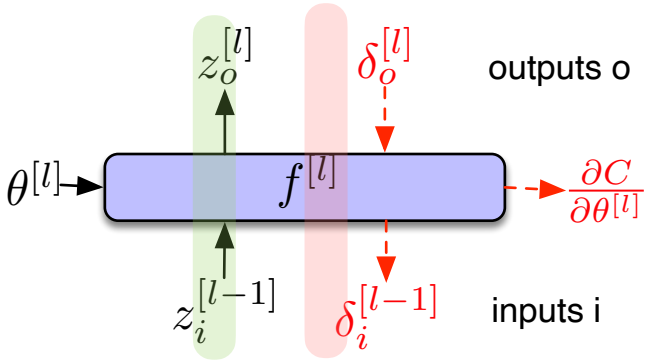
- **Forward**

$$z_o^{[l]} = f_o(\mathbf{z}^{[l-1]}, \theta) = \sum_i z_i^{[l-1]} \theta_{io}$$

- **Backward**

$$\delta_i^{[l-1]} = \sum_o \delta_o^{[l]} \frac{\partial [f_o(\mathbf{z}l-1,\theta]}{\partial z_i^{[l-1]}} = \sum_o \delta_o^{[l]} \theta_{io}$$

$$\frac{\partial C}{\partial \theta_{io}} = \sum_o \delta_o^{[l]} \frac{\partial [f_o(\mathbf{z}l-1,\theta]}{\partial \theta_{io}} = \delta_o^{[l]} z_i^{[l-1]}$$



Deep Learning
ReLu layer

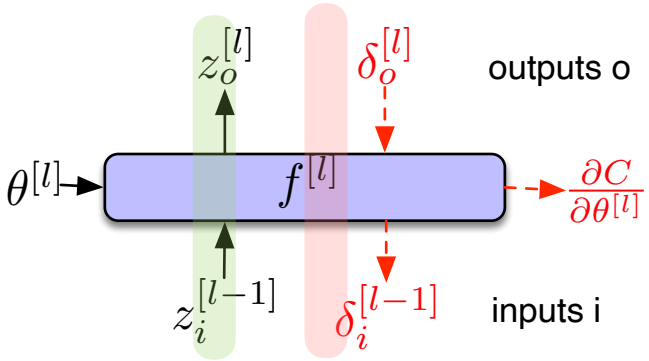
- **Forward**

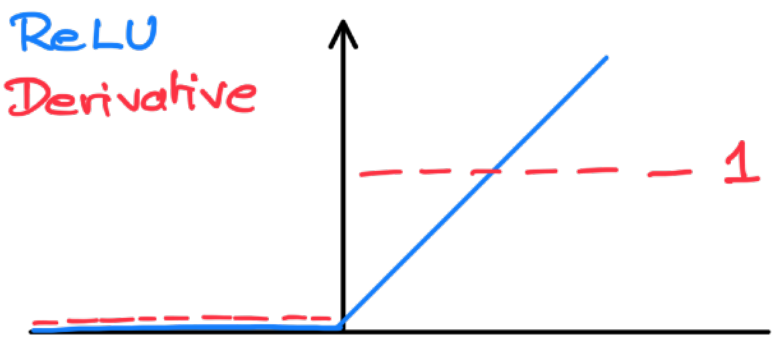
$$z_o = f_o(z_o^{[l-1]}) = \max(0, z_o^{[l-1]})$$

- **Backward**

$$\delta_i^{[l-1]} = \sum_o \delta_o^{[l]} \frac{\partial [f_o(z_o^{[l-1]})]}{\partial z_i^{[l-1]}}$$

$$= \delta_i^{[l]} \mathbb{I}_{[z_i^{[l-1]}>0]}$$





In pytorch

https://colab.research.google.com/drive/1TqjRB2NVB3XUQTBrYWgqdCddvuwReUAE#scrollTo=DeMtXw53lJHp

Data	Supervised learning	$(x^{(i)}, y^{(i)})$	
Model	Architecture	Linear regression Non-linear regression Logistic regression Softmax regression Neural Network Deep Learning	Output activation depends on the task Avoid vanishing gradient Parameter initialization Batch Normalization
Cost/Loss	Maximize Likelihood	Gaussian (MSE) Bernoulli (BCE)	Cost depends on the task
Regularization	Avoid over-fitting: L1/L2,DropOut		
Optimizer	For convex optimization For non-convex optimization (SGD, Momentum, ...) First order (SGD), Second order (Hessian)		
Training			
Dataset/ Dataloader		Get mini-batch	
Forward		Compute prediction $\hat{y}^{(i)}$	
Backward (back-propagation)		Compute gradients	
Update			
Early Stopping		Avoid over-fitting	