Deep Learning

Formalism, Data Visualization, from ML to DeepLearning

Alexandre Fournier Montgieux

M2 BDMA, CentraleSupélec, Université Paris Saclay

September 25, 2024

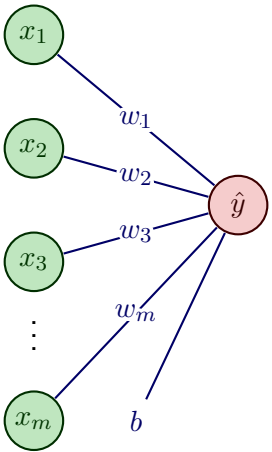
Plan

Perceptron

- A perceptron is an algorithm for supervised learning of binary classifiers (two classes).
- Suppose we have a dataset $\mathbf{X} \in \mathbb{R}^{n \times m}$ associated with a vector of labels $\mathbf{y} \in \{0, 1\}^n$
- It learns a function $\tilde{f}$ parametrized by a vector of weights $\mathbf{w} \in \mathbb{R}^m$ and a bias b such as:

$$\tilde{f}(\mathbf{x}) = \begin{cases} 1 & \text{if } \mathbf{w} \cdot \mathbf{x} + b > 0, \\ 0 & \text{otherwise} \end{cases}$$

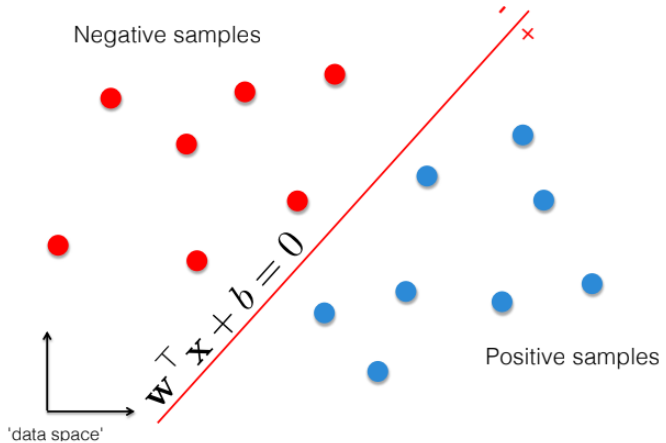
Perceptron visualization



$$\hat{y} = w_1 x_1 + w_2 x_2 + ... + w_m x_m + b$$

Perceptron Decision Boundary

A perceptron can linearly separate data



Gradient Descent for Perceptron

Algorithm 1: Gradient Descent for Perceptron

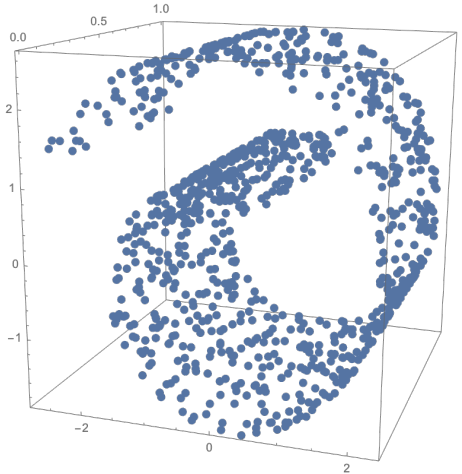
Input: Data points $X = \{x_1, x_2, \ldots, x_n\}$, labels $Y = \{y_1, y_2, \ldots, y_n\}$, learning rate η, number of epochs T

Output: Weights w, bias b

Initialize w and b randomly;
for $t = 1$ **to** T **do**
  foreach $(x_i, y_i) \in (X, Y)$ **do**
    Compute $\hat{y}_i = wx_i + b$;
    Compute loss $L = \frac{1}{2}(\hat{y}_i - y_i)^2$;
    Compute gradients $\frac{\partial L}{\partial w} = (\hat{y}_i - y_i)x_i$;
    Compute gradients $\frac{\partial L}{\partial b} = (\hat{y}_i - y_i)$;
    Update $w := w - \eta \frac{\partial L}{\partial w}$;
    Update $b := b - \eta \frac{\partial L}{\partial b}$;

Perceptron limitation

A perceptron cannot separate non linear data



Percepton limitations mitigation : Adaboost

Let's $\{h\}$ be a set of perceptions (weeak classifiers), $H = sign(\sum(h(x))$, we want to find the right h to get the best H possible.

Adaboost Algorithm

Input: $\mathbf{H}$: a chosen class of "weak" binary classifiers
Output: $F_t = \text{sign}(H_t)$

Initialize: $w_1(i) = \frac{1}{n}$, $H_0 = 0$;
for $t = 1$ **to** T **do**
  $h_t = \arg\min_{h \in \mathbf{H}} \epsilon_t(h)$;
  where $\epsilon_t(h) = \sum_{i \sim w_t} [h(x_i) \neq y_i]$;
  Choose α_t;
  Update w_{t+1};
  $H_t = H_{t-1} + \alpha_t h_t$;
end

Output: $F_T = \text{sign}(H_T)$;

Updating parameters

- $\alpha_t = \frac{1}{2} log(\frac{1-\epsilon_t}{\epsilon_t}$
- $w_{t+1} = \frac{w_t(I) e^{-\alpha_t y_i h_t(x_i)}}{Z_{t+1}}$, Z is chosen so that the sum of w is equal to 1

Resulting steps

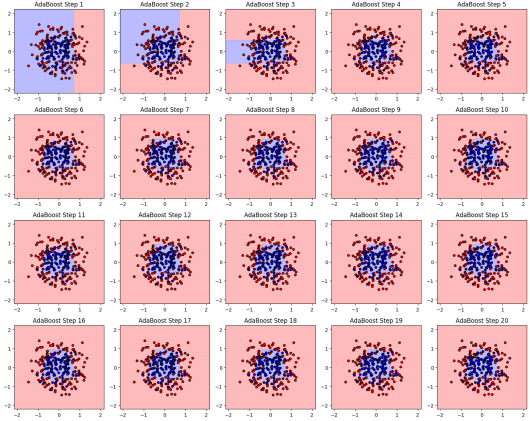


Figure: Resulting classifier for each step

Last session reminder

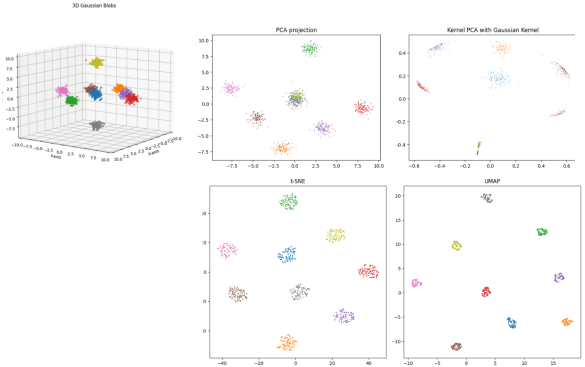


Figure: Vizualization tools comparison

Last session reminder

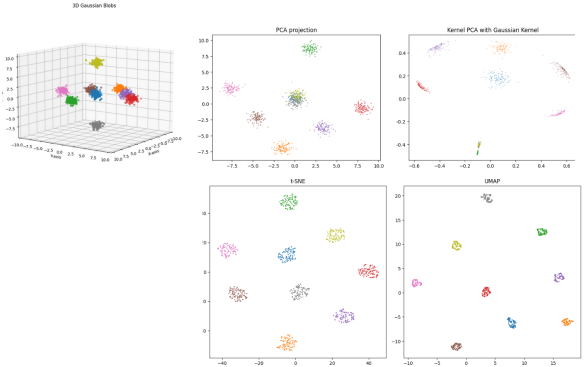


Figure: Vizualization tools comparison

PCA important details

- Some library implementation might use the kernel PCA formulation as a general formalism
- Remind that we then have : $\Sigma = \frac{1}{n-1} \sum_i^n \phi(x_i)\phi(x_i)^T$, with $\phi = id$ for PCA
- As a safe measure, do not forget to center the data !
- For kernel PCA centering is performed by updating

$$\mathbf{K} = K - 1_n K - K 1_n + 1_n K 1_n$$

Remind on supervised approach

- Supervised learning through perception allows a **linear** separation of the space.
- Perceptron based solution exists (such as Adaboost) to perform nonlinear classification
- Fitting each weak classifier is done iteratively using a dynamic weighting of the samples

Theoritical reminder: Differentiability

A function $f : U \subset \mathbb{E} \to \mathbb{F}$, defined on an open set U is differentiable in $x \in U$ if there exists $L_x : \mathbb{E} \to \mathbb{F}$ linear and continuous s.t. for all $h \in U$ with $x + h \in U$:

$$f(x + h) =_{h \to 0} f(x) + L_x(h) + o(h)$$

Then L_x is called the differential of f at the point x and is denoted $df(x)$

Theoritical reminder: Jacobian function

Lets be a function $f : U \subset \mathbb{R}^n \to \mathbb{R}^p = (f_1, f_2, ..., f_p)$, with each f_i a function of $U \subset \mathbb{R}^n \to \mathbb{R}$.
Let $x \in U$ a point at which all f_i have defined partial derivatives, then we define the Jacobian matrix as follows:

$$Jac_f(x) = (\frac{\partial f_i}{\partial x_j})_{1 \leq i \leq p, 1 \leq j \leq n}, Jac_f : U \to \mathbb{R}^{p \times n}$$

Keep in mind the **Jacobian chain rule** :

$$Jac_{g \circ f}(x) = Jac_g(f(x)) \times Jac_f(x)$$

Gradient Definition

For a differentiable function $f : U \subset \mathbb{R}^n \to \mathbb{R}$, we define at each point $x \in U$ the differentiate $df(x) : \mathbb{R}^n \to \mathbb{R}$. Then, following the Rietz theorem, there exists a unique vector, noted $\nabla_x f$ and called gradient s.t. :

$$\forall h \in U : df(x)(h) =< \nabla_x f | h >$$

Gradient in Differentiable Calculus

For a differentiable function $f : \mathbb{R}^n \to \mathbb{R}$, the gradient can be computed with the partial derivative of f :

$$\nabla_x f = \left(\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \ldots, \frac{\partial f}{\partial x_n} \right)^T = Jac_f(x)^T$$

Example: Perceptron

Consider a perceptron with weights $\mathbf{w}$ and input $\mathbf{x}$. The output is given by:

$$y = \mathbf{w} \cdot \mathbf{x} + b$$

where b is the bias term.

Perceptron Loss Function

The perceptron loss function for a single training example $(\mathbf{x}, t)$ is:

$$L(\mathbf{w}) = -t(\mathbf{w} \cdot \mathbf{x} + b)$$

where t is the target label.

Gradient of the Perceptron Loss

The gradient of the perceptron loss function with respect to the weights $\mathbf{w}$ is:

$$\nabla L(\mathbf{w}) = -t\mathbf{x}$$

Gradient Descent Update Rule

Using gradient descent, the weights are updated as follows:

$$\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla L(\mathbf{w})$$

where η is the learning rate.

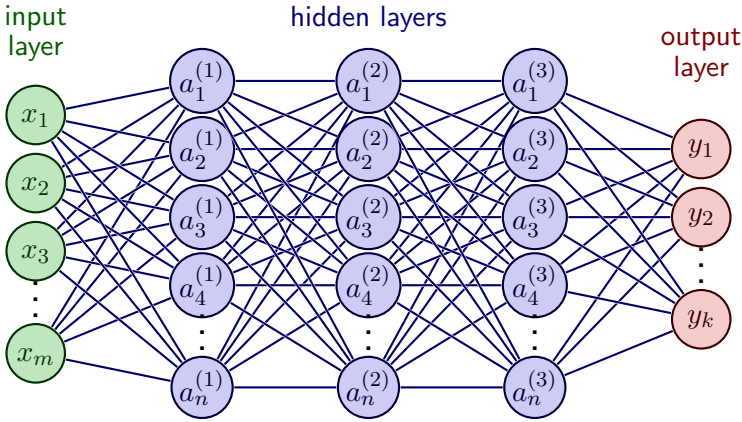
Applying the Gradient to the Perceptron

For a misclassified example $(\mathbf{x}, t)$, the update rule becomes:

$$\mathbf{w} \leftarrow \mathbf{w} + \eta t \mathbf{x}$$

This adjusts the weights to reduce the classification error.

Multi-layer perceptron



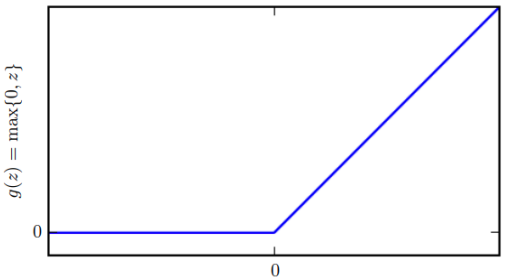
- "Deep" neural networks are perceptrons with multiple hidden layers
- **Problem:** still a *linear* model.

Piecewise-linear perceptrons

ReLU: (Rectified Linear Unit)

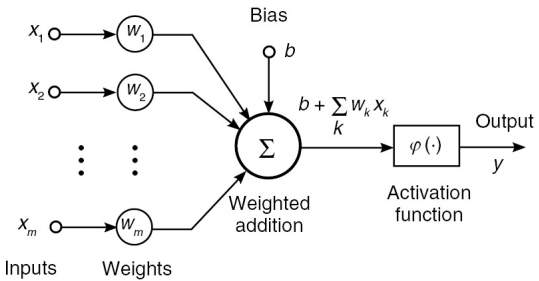
Non-linear real function defined as:

$$\forall z \in \mathbb{R} \quad g(z) = \begin{cases} z & \text{if } \ z \geq 0, \\ 0 & \text{otherwise} \end{cases} = \max(z; 0)$$



Piecewise-linear perceptrons

- ReLU is actually a **piecewise-linear** function and preserves much of the good optimization properties of a linear function (differentiable)
- We can use **activation functions** such as ReLU between hidden layers to introduce non-linearity in the model.



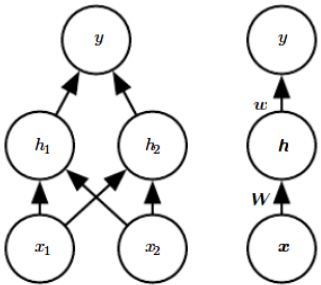
Example: Learning XOR

Objective

Learn the XOR function by **finding the correct weight values** of a 2-layer perceptron.

x_1	x_2	y
0	0	0
0	1	0
1	0	0
1	1	1

Table: XOR function



Example: Learning XOR

2-layer perceptron with ReLU activation

- **Hidden layer**: $f^{(1)}(\boldsymbol{x}) = \boldsymbol{W}^\top \boldsymbol{x} + \boldsymbol{c}$ where $\boldsymbol{W} \in \mathbb{R}^{2\times 2}$ and $\boldsymbol{c} \in \mathbb{R}^{2\times 1}$
- **Activation function (ReLU)**: $\boldsymbol{h} = g(f^{(1)}(\boldsymbol{x})) = \max(0; f^{(1)}(\boldsymbol{x}))$
- **Output layer**: $y = f^{(2)}(\boldsymbol{h}) = \boldsymbol{w}^\top \boldsymbol{x} + b$ where $\boldsymbol{w} \in \mathbb{R}^{2\times 1}$ and $b \in \mathbb{R}$

$$\begin{aligned} y &= f(\boldsymbol{x}; \boldsymbol{W}, \boldsymbol{w}, \boldsymbol{c}, b) \\ &= f^{(2)}(g(f^{(1)}(\boldsymbol{x}))) \\ &= \boldsymbol{w}^\top \max(0, \boldsymbol{W}^\top \boldsymbol{x} + \boldsymbol{c}) + b \end{aligned} \tag{1}$$

Example: Learning XOR

One solution (**not unique !**)

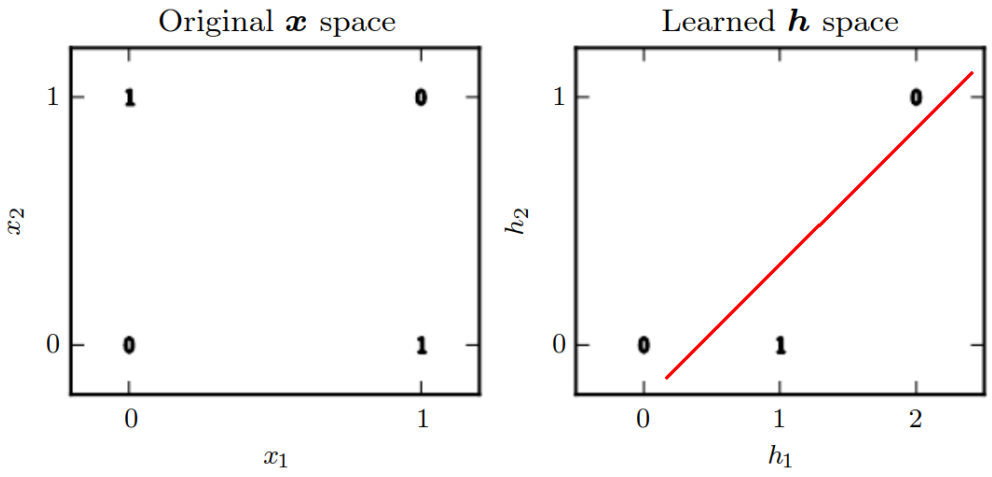
$$\boldsymbol{W} = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}, \boldsymbol{c} = \begin{bmatrix} 0 \\ -1 \end{bmatrix}, \boldsymbol{w} = \begin{bmatrix} 1 \\ -2 \end{bmatrix} \text{ and } b = 0$$

Proof:

Left to the reader (use $\boldsymbol{X} = \begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix}$). This is just one solution among an infinity, the system of equations being underconstrained.

In general, deep neural networks are **overparametrized** (more parameters than training samples). They have thus many local optimum on a given optimization task with a given dataset.

Example: Learning XOR



Deep Feedforward Networks

Building blocks of a **deep feedforward network** (or MLP):

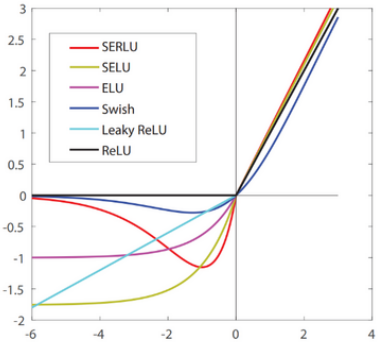
- multiple consecutive **linear layers** (≥ 2)
- **activation functions** (hidden units) in between layers to introduce non-linearities
- optional **output unit** (activation)

Activation functions

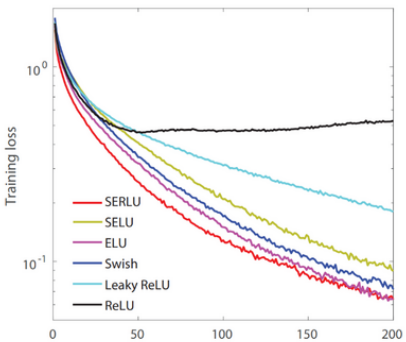
Activation	Definition	Derivative	Range	Smoothness
 Identity (Id)	$g(x) = x$	$g'(x) = 1$	$\mathbb{R}$	C^∞
 Sigmoid (Logistic, σ)	$g(x) = \frac{1}{1+e^{-x}}$	$g'(x) = g(x)(1-g(x))$	$(0,1)$	C^∞
 Heaviside (H)	$g(x) = \begin{cases} 0, & x<0 \\ 1, & x \geq 0 \end{cases}$	$g'(x) = 0,\ x \neq 0$	$\{0,1\}$	none
 Hyperbolic tangent (tanh)	$g(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$	$g'(x) = 1 - g(x)^2$	$(-1,1)$	C^∞
 Rectified linear unit (ReLU)	$g(x) = \max\{x, 0\}$	$g'(x) = \begin{cases} 0, & x<0 \\ 1, & x>0 \end{cases}$	$[0,\infty)$	C
 Parametric rectified linear unit (PReLU)	$g(x) = \begin{cases} \alpha x, & x<0 \\ x, & x \geq 0 \end{cases}$ $\alpha > 0$	$g'(x) = \begin{cases} \alpha, & x<0 \\ 1, & x>0 \end{cases}$	$\mathbb{R}$	C
 Exponential linear unit (ELU)	$g(x) = \begin{cases} \alpha(e^x - 1), & x<0 \\ x, & x \geq 0 \end{cases}$ $\alpha > 0$	$g'(x) = \begin{cases} g(x) + \alpha, & x<0 \\ 1, & x>0 \end{cases}$	$(-\alpha,\infty)$	C^1, $\alpha = 1$ C, $\alpha \neq 1$
 Softplus	$g(x) = \log(1 + e^x)$	$g'(x) = \frac{1}{1+e^{-x}}$	$(0,\infty)$	C^∞
 Gaussian	$g(x) = e^{-x^2}$	$g'(x) = -2xg(x)$	$(0,1]$	C^∞

Activation functions

ReLU is now slowly being replaced by better alternatives...



(a): Different activation functions



(b): Performance on CIFAR10 without dropout

Learning principle of deep neural networks

- Architecture design
- Choice of layers (MLP, CNN, RNN, attention (Transformer), GNN...)
- Non-linearities
- Cost function $\mathcal{L}$
- Minimize $\mathcal{L}$ with stochastic gradient-descent
  - Train on a **training** dataset
  - Estimate error on an **evaluation** dataset
  - Gradient computation by backpropagation
- Aiming for local minimum (or at least reducing training error), instead of global minimum.
  - Deep neural networks have surprisingly good local & non-global optimum !

Choice of cost function

- General case: maximum likelihood principle ()
- Highly connected to type of outputs of the NN

Max Likelihood reminder

- a dataset $\{x_1, x_2, \ldots, x_n\}$ i.i.d distributed from unknown true distribution $p_{\text{data}}(x)$
- parametric model family $p_{\text{model}}(x; \theta)$ (neural network). It estimates an approximation of the true distribution $p_{\text{data}}(x)$

MLE is:

$$\theta_{ML} = \arg\max_{\theta} \prod_{i=1}^{n} p_{\text{model}}(x_i; \theta)$$

Choice of cost function

- Maximmum likelihood principle is very common.
- We often choose log-likelihood in practice.
  $\theta_{ML} = \arg\max_{\theta} \sum_{i=1}^{n} \log p_{\text{model}} \left(x^{(i)}; \theta\right)$
- $\log$ is increasing and continuous function. $\Rightarrow$ maximum is the same for MLE and log-MLE.
-

MLE and cross-entropy

- **Maximum likelihood estimation:**

$$\begin{aligned}\theta_{ML} &= \arg\max_{\theta} \sum_{i=1}^{n} \log p_{\text{model}}\left(x_i; \theta\right) \\ &= \arg\max_{\theta} \mathbb{E}_{x \sim \hat{p}_{\text{data}}} \left[\log p_{\text{model}}\left(x; \theta\right)\right]\end{aligned}$$

- **Interpretation**: minimizing KL-divergence between predicted distribution true distribution

$$\begin{aligned}&\min KL\left(\hat{p}_{\text{data}}\ , p_{\text{model}}\ \right) \\ &\Leftrightarrow \min \mathbb{E}_{\text{x}\ \hat{p}_{\text{data}}} \left[\log \hat{p}_{\text{data}}\left(x\right) - \log p_{\text{model}}\left(x\right)\right] \\ &\Leftrightarrow \min -\mathbb{E}_{X \sim \hat{p}_{\text{data}}} \left[\log p_{\text{model}}\left(x\right)\right]\end{aligned}$$

This is called the **cross-entropy** between $\hat{p}_{\text{data}}$ and p_{model} .

MLE and cross-entropy

Summary of cost function for MLE

- A neural network is parametric model defining a distribution $p_{\text{model}}(\mathbf{y} \mid \mathbf{x}; \theta)$ ou $p_{\text{model}}(\mathbf{x}; \theta)$
- Supervised learning with MLE leads to optimization of the following cross-entropy (or negative log likelihood): $J(\theta) = -\mathbb{E}_{x,y \sim \hat{p}_{\text{data}}} [\log p_{\text{model}}(y \mid x; \theta)]$
- Exact form of $J(\theta)$ only depends on $p_{model} \Leftrightarrow$ only depends on the task at hand

Cross-entropy in practice

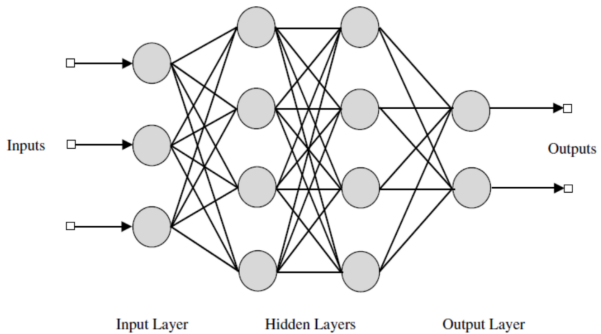
- Practical implementation of the cross-entropy for multi-class classification $J(\theta) = -\mathbb{E}_{x,y\sim\hat{p}_{\text{data}}} \left[\log p_{\text{model}} \left(y \mid x; \theta\right)\right]$ is the following:

$$J = -\sum_{i=1}^{N}\sum_{j=1}^{M} y_{i,j} \log\left(p_{i,j}\right)$$

- Where:
  - M number of classes
  - N number of samples (data points in dataset)
  - $y_{i,j}$ is 1 if class j is correct for sample i
  - $p_{ij} = ¶(x_i \in j)$
- for $M = 2$, we have $J = -(y\log(p) + (1-y)\log(1-p))$
- Probability always depends on the structure of the output layer –¿ activation functions !
- (there are other type of losses than cross-entropy)

Output units

MLP transforms inputs into features $h = f(x; \theta)$ with the hidden layers.



- How do we output the output values $\hat{y}$?
- What kind of decision? (classification, regression...?)
- We use the representation h (or "features") right before the last layer.

Output units

In the maximum likelihood estimation framework, we might apply activation functions to the output layer to get a desired structure for our distribution. This choice will also influence the mathematical form of the cost function. Examples:

- **Linear** units for regression
- **Linear** units for Gaussian distributions
- **Sigmoid** units for binary classification
- **Softmax** units for multi-class classification

Linear units for regression

Linear output layer

Given features h, we output $\hat{\mathbf{y}} = \mathbf{W}^T\mathbf{h} + b$.

- **Regression** problem: we predict a real-valued variable
- Examples:
  - price (stocks, oil, housing...)
  - biometrics (size, age, weight...)
  - size & position of detection boxes
  - physical system metric/sensor

Linear unit for Gaussian distributions

Gaussian output unit

Given features $\boldsymbol{h}$, a layer of linear output units produces a vector $\hat{\boldsymbol{y}} = \boldsymbol{W}^\top \boldsymbol{h} + \boldsymbol{b}$. Linear output layers (ie no activation) are often used to produce the mean and/or covariance matrix of a conditional Gaussian distribution: $p(\boldsymbol{y} \mid \boldsymbol{x}) = \mathcal{N}(\boldsymbol{y}; \hat{\boldsymbol{y}}, \boldsymbol{I})$. Covariance is usually:

- not modelled
- simplified to be diagonal (don't forget to make the output **non-negative** !)

Maximizing the log-likelihood is then equivalent to minimizing the mean squared error (in the case of identity covariance matrix).

Binary classification

Objective: predict binary y variable.

- Should we schedule for early reparation (predictive maintenance)?
- Is a mail fraudulent/spam?
- Should we buy/sell?
- Does an image contain something?
- Is there something unusual going on in the image/video/text/database...?

Neural network must predict $P(y = 1 \mid x)$. In order to be a valid probability (in $[0, 1]$) with linear units, we can take:

$$P(y = 1 \mid x) = \max\left\{0, \min\left\{1, \mathbf{w}^T \mathbf{h} + b\right\}\right\}$$

But if $\mathbf{w}^T \mathbf{h} + b \notin \left[\begin{array}{cc} 0 & 1 \end{array}\right]$ then $\nabla_w J(\theta)$ is zero –¿ gradient descent stops.

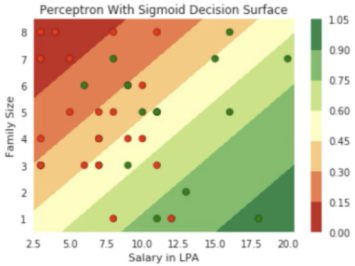
Sigmoid unit for binary classification

Binary classification task $\Leftrightarrow$ modelling a Bernoulli distribution conditioned on the input. NN needs to predict only $P(y = 1 \mid \boldsymbol{x})$. Problem: y is a valid probability if $y \in [0, 1]$. $\Rightarrow$ new constraint to apply to the output.

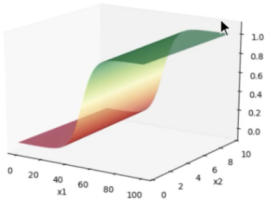
Sigmoid unit

$$\hat{y} = \sigma\left(\boldsymbol{w}^\top \boldsymbol{h} + b\right) = \frac{1}{1 + e^{-\boldsymbol{w}^\top \boldsymbol{h} + b}}$$

Sigmoid unit for binary classification: example



$$y = \frac{1}{1+e^{-(w^T x+b)}}$$

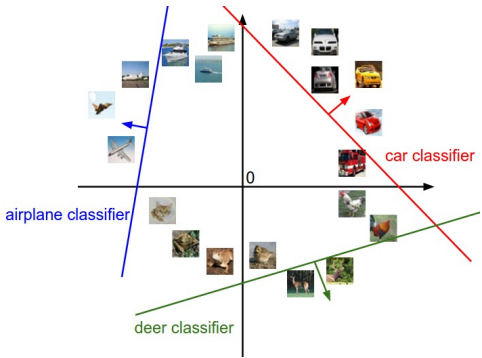


	Salary in LPA	Family Size	Buys Car?
0	11	8	1
1	20	7	1
2	4	8	0
3	8	7	0
4	11	5	1

Softmax unit for multi-class classification

- **Objective**: *One vs All* classification among $N > 2$ classes (otherwise we fall back to binary classification). Examples:

- Person recognition (image, voice...)
- Categories of objects in an image/video
- Choosing an action (RL)
- Sentiment analysis



- Generalization of binary case. We want to predict $\hat{y}$ with $\hat{y}_i = P(y = i \mid \mathbf{x})$
- Subject to: $\hat{y}_i \in [0, 1]$ and $\sum_i \hat{y}_i = 1$

Softmax unit for multi-class classification

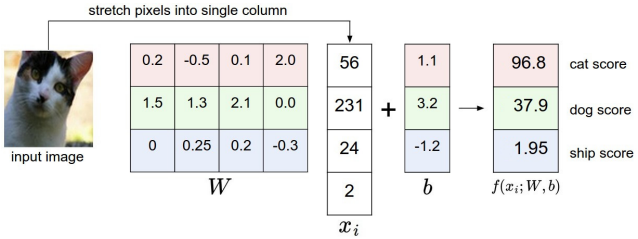
As before, we have at disposal the **logits** (unnormalized log probabilities):

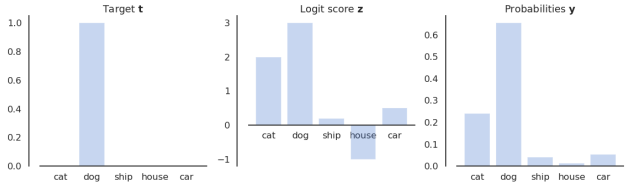
$$\mathbf{z} = \mathbf{W}^T\mathbf{h} + \mathbf{b} \text{ with } z_i = \log \tilde{P}(y = i \mid \mathbf{x})$$

Softmax output unit

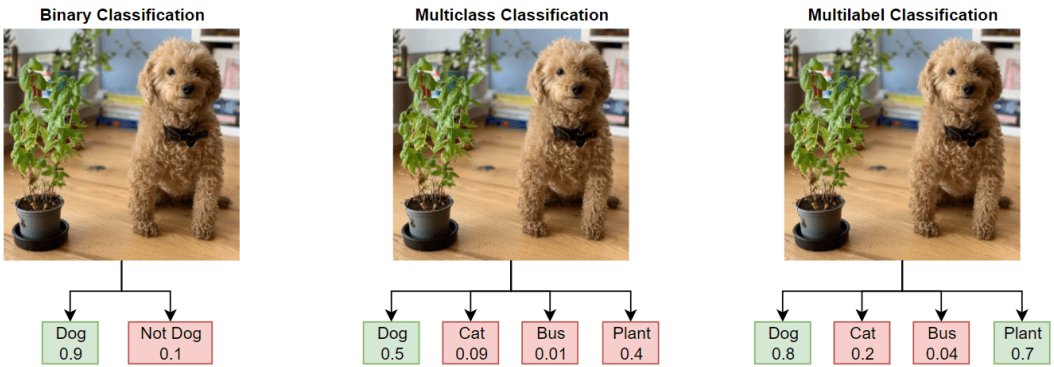
$$\text{softmax}(\mathbf{z})_i = \frac{e^{z_i}}{\sum_j e^{z_j}}$$

Softmax unit for multi-class classification

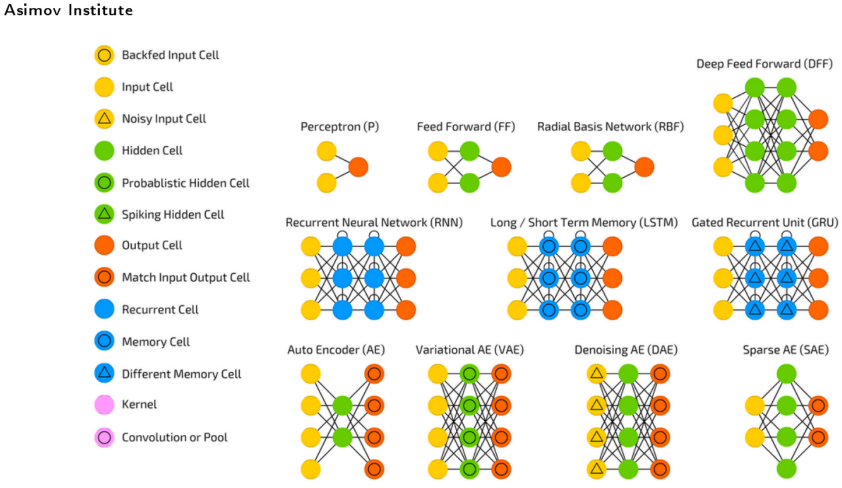




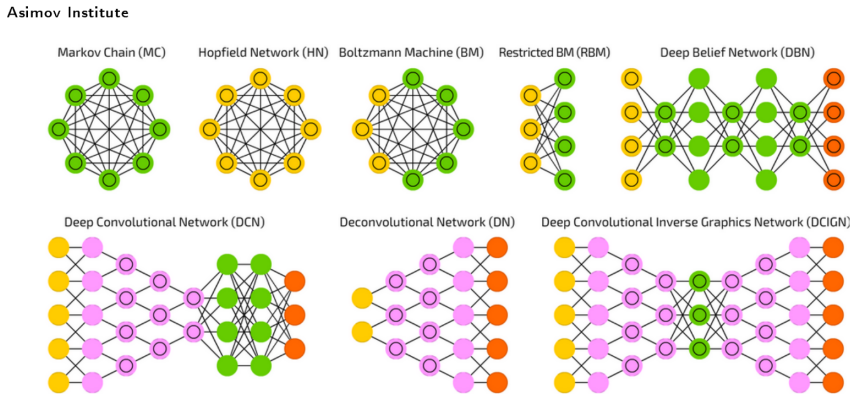
Different types of classification



Different types of neural networks

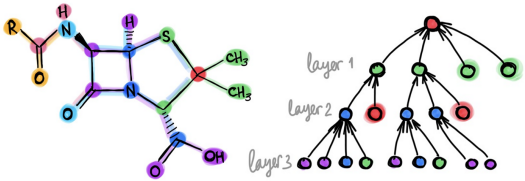


Different types of neural networks



More common and recent types

Graph Neural Networks (any graph data: social networks, molecules, interaction meshes, etc...)



Transformers (NLP, computer vision, etc...)

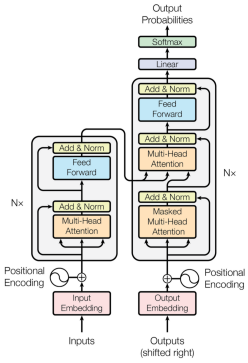
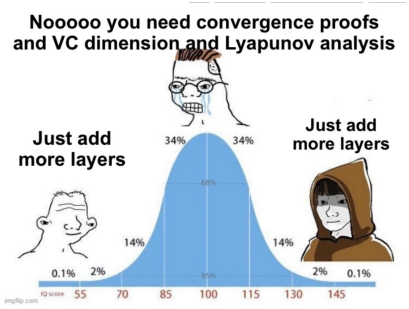


Figure 1: The Transformer - model architecture.

Why **deep** neural networks?

- **Depth** is the longest path data can take from input to output
- For deep feedforward networks, depth = number of hidden layer + 1 output layer
- State-of-the-art architectures used in practice (eg. computer vision, text...) have dozens to hundreds of layers $\Leftarrow$ **millions to billions of parameters**



Universal approximation theorem

Theorem

Let $\varphi()$ be a nonconstant, bounded, and monotonically-increasing continuous function. Let I_{m_0} denote the m_0-dimensional unit hypercube $[0,1]^{m_0}$. The space of continuous functions on I_{m_0} is denoted by $C\left(I_{m_0}\right)$. Then, given any function $f \in C\left(I_{m_0}\right)$ and $\epsilon > 0$, there exists an integer m_1 and sets of real constants α_i, b_i and $w_{ij} \in \Re$, where $i = 1, \ldots, m_1$ and $j = 1, \ldots, m_0$ such that we may define:

$$F(\mathbf{x}) = \sum_{i=1}^{m_1} \alpha_i \cdot \varphi \left(\sum_{j=1}^{m_0} w_{ij} \cdot x_j + b_i \right)$$

as an approximate realization of the function f; that is,

$$|F(\mathbf{x}) - f(\mathbf{x})| < \epsilon \text{ for all } x \in I_m$$

Universal approximation theorem

- The theorem says that for any input/output mapping function f in supervised learning, there exists a MLP with m_1 neurons in the hidden layer which is able to approximate it with a desired precision!
- It only proves the existence of a shallow MLP with m_1 neurons in the hidden layer that can approximate any function, but it does not tell how to find this number.
- Rule of thumb for generalization error: $\epsilon = \frac{\mathrm{VC_{dim}(MLP)}}{N}$
- More neurons in hidden layer = better training error but worse generalization error (overfitting)
- In practice, for most functions m_1 is very high (it grows *exponentially*) and becomes quickly computationally intractable. **We need to go deeper !**

No Free Lunch theorem

NFL theorem [Wolper, 1956]

Multiple informal formulations:

- For every learning algorithm A and B, there are as many problems where A has a better generalization error than problems where B has a better one.
- All learning algorithms have the same generalization error if we average over all learning problems.
- There is no universally better learning algorithm.

Why go deeper?

Depth property

The number of "polygonal" regions generated by a MLP with ReLU activation, d inputs, n neurons per hidden layer and l layers is:

$$O\left(\binom{n}{d}^{d(l-1)} n^d\right)$$

.

The number of regions grows **exponentially** with depth !!

Depth and space folding

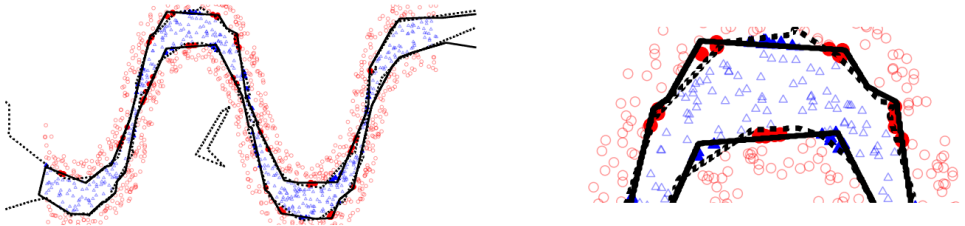
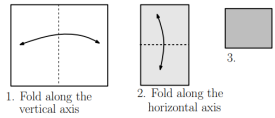
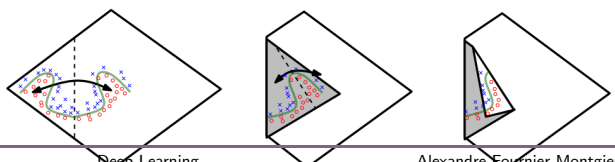


Figure 1: Binary classification using a shallow model with 20 hidden units (solid line) and a deep model with two layers of 10 units each (dashed line). The right panel shows a close-up of the left panel. Filled markers indicate errors made by the shallow model.

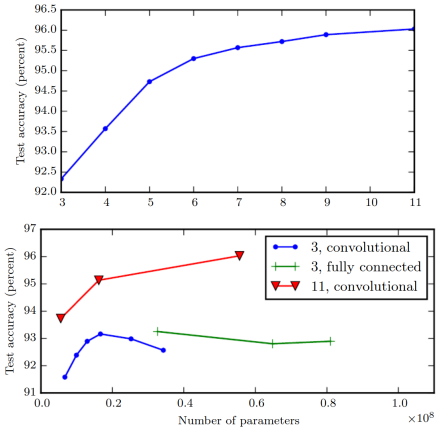
Depth and space folding





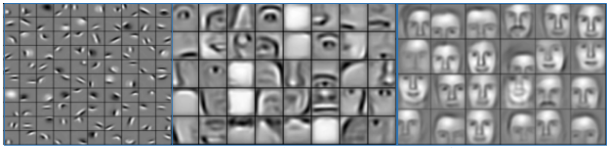
Deeper is better

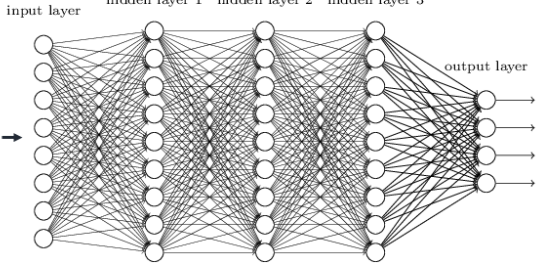
Depth improve generalization error, and depth is better than width!



Hierarchical structure of depth

Deep neural networks learn hierarchical feature representations





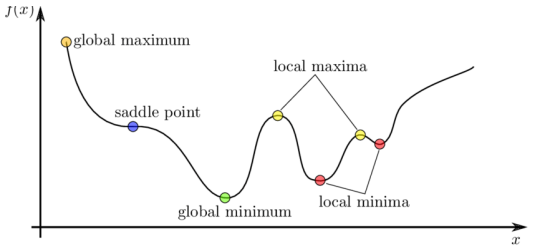
Optimization

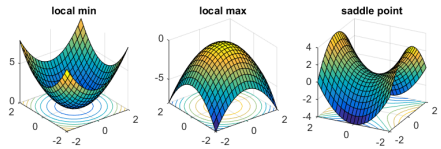
Optimization

Finding a global/local minima (or maxima) of a cost function $f(\theta) : \mathbb{R}^n \to \mathbb{R}$.

- $\max f(\theta) \Leftrightarrow \min -f(\theta)$
- $f(\theta)$ is called: *cost* function, *error* function or *loss function*
- Optimum: $\theta^* = \arg\min f(\theta)$
- Cost functions in deep learning have a lot of local minima and saddle points.
- BUT: a **good** local minimum is better than an untractable global optimimum

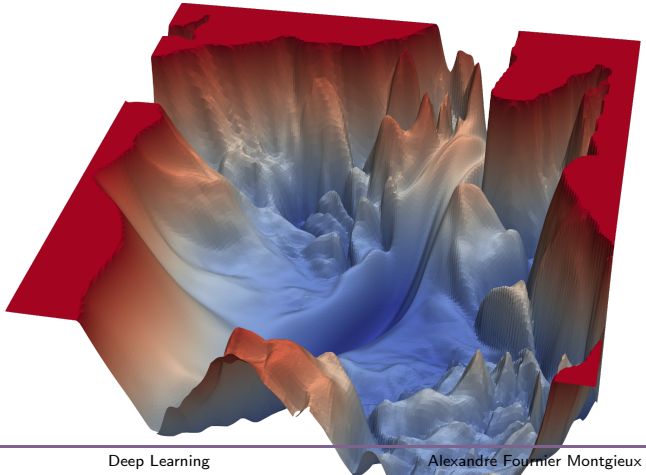
Minimum, maximum and saddle point





Loss landscape

This is a representation of the loss function values depending on the model weights θ. Here only 2 dimensions are observed. During optimization, θ will move on this surface.



Gradient

Gradient

For $f : \mathbb{R}^n \to \mathbb{R}$, its gradient $\nabla f : \mathbb{R}^n \to \mathbb{R}^n$ is defined at the point $p = (x_1, \ldots, x_n)$ in n-dimensional space as the vector:

$$\nabla f(p) = \begin{bmatrix} \frac{\partial f}{\partial x_1}(p) \\ \vdots \\ \frac{\partial f}{\partial x_n}(p) \end{bmatrix}$$

- It is the local derivative or "slope" of each dimension at a certain point
- Opposite direction of the gradient is a naïve but practical guess of direction of the local minimum

Gradient descent

The gradient $\nabla_\theta f(\theta)$ is the direction to follow in the space of θ to make $f(\theta)$ decrease the fastest.

Gradient-descent [Cauchy, 1847]

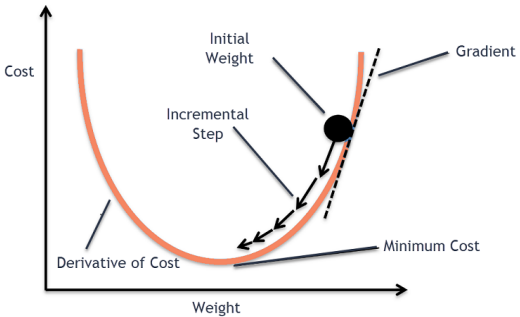
A parametric function $f(\theta)$ can be iteratively minimized by following the opposite direction of the gradient:

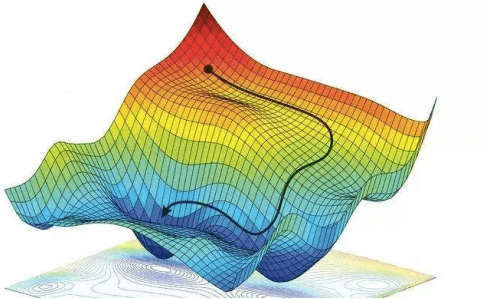
$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \epsilon \nabla_{\boldsymbol{\theta}} f(\boldsymbol{\theta})$$

Where $\epsilon > 0$ is the *learning rate*. We stop iterations when $\nabla_\theta f(\theta) \approx 0$.

- We don't need this iterative version if we have a closed-form for $\nabla_\theta f(\theta)$ (never the case or untractable for neural networks).
- We can vary ϵ during training: vary the **step size**.

Gradient descent





Stochastic gradient descent

Given a cost function $f(\theta)$, parameters of the network are updated with:

$$\theta \leftarrow \theta - \epsilon \nabla_\theta f(\theta)$$

For the negative log-likelihood (MLE):

$$f(\theta) = \frac{1}{m} \sum_{i=1}^{m} L\left(x^{(i)}, y^{(i)}, \theta\right)$$

Then the estimated gradient becomes:

$$\nabla_\theta f(\theta) = \frac{1}{m} \sum_{i=1}^{m} \nabla_\theta L\left(x^{(i)}, y^{(i)}, \theta\right)$$

$\rightarrow$complexity in $O(m)$: can go as high as 10^8 (images) - 10^{11} (text)

Stochastic gradient descent

Problem: with this approach to take a single step of gradient descent, we must compute the loss over the **whole** dataset everytime! $\rightarrow$not scalable. This is called *batch gradient descent*.
Solution: Compute the gradient with **1** sample only at each step!

$$\theta \leftarrow \theta - \epsilon \nabla_\theta f\left(\theta; x^{(i)}, y^{(i)}\right)$$

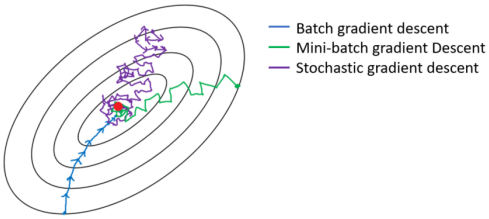
Very noisy and inefficient, but surprisingly still works !
(Intuition: software development of an app. You improve quicker by iterating over 1 review at a time than waiting for all the reviews.)

Mini-batch stochastic gradient descent

- Stochastic gradient descent: update with 1 sample
- Batch gradient descent: update will ALL samples
- Compromise: **mini-batch** gradient descent

$$\theta \leftarrow \theta - \epsilon \nabla_\theta f\left(\theta; x^{(i:i+B)}, y^{(i:i+B)}\right) = \theta - \frac{\epsilon}{B} \nabla_\theta \sum_{i=i_0}^{i_0+B} L\left(x^{(i)}, y^{(i)}, \theta\right)$$

- batch GD: slowest, perfect gradient
- SGD: fastest, noisy
- mini-batch SGD: **compromise**



Mini-batch stochastic gradient descent

- A "batch" is a collection of samples used at each iteration for performing (mini-batch) SGD in deep learning (batch of images, text, video fragments...)
- Bigger batch = better gradient estimation $\rightarrow$"faster" learning
- Bigger batch = more device memory used (GPU size and number is often the bottleneck in performance)
- Bigger batch = slower gradient descent (GPU is designed for parallel matrix multiplication, but it is not infinitely parallel)
- Thus there is often tradeoff between money and performance at companies
- In practice: batch is between 1 to 256 on one GPU $\rightarrow$8 GPU node = 2048 ! (Big AI companies AI/research lab can go even higher with superclusters: eg. Open AI Five (Dota 2 AI agent) 8,388,608 batch size!!)

Backpropagating gradients

- **Reminder**: learning of a NN is based on optimizing a cost function $J(\theta)$
- In practice, optimization performed by gradient descent $\rightarrow$we must be able to compute $\nabla_\theta J(\theta)$.
  - Problem: computationally very costly
- **Back-propagation** is an efficient gradient computation technique
  - **Not** a learning algorithm/training methodology
  - **Not** necessarily exclusive to neural networks

Chain rule

- The derivative of composition of functions is called the *chain rule* of calculus.
- If $y = g(x)$ and $z = f(y) = f(g(x)) = (f \circ g)(x)$:

$$\frac{df}{dx} = f'(g(x))g'(x) \quad \text{or} \quad \frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx}$$

- multivariate function (vector calculus):

$$\mathbf{x} \in \mathbb{R}^m, \mathbf{y} \in \mathbb{R}^n, g : \mathbb{R}^m \mapsto \mathbb{R}^n \text{ and } f : \mathbb{R}^n \mapsto \mathbb{R} :$$

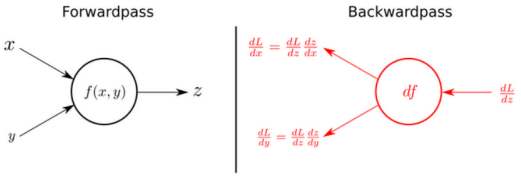
$$\frac{\partial z}{\partial x_i} = \sum_j \frac{\partial z}{\partial y_j}\frac{\partial y_j}{\partial x_i}$$

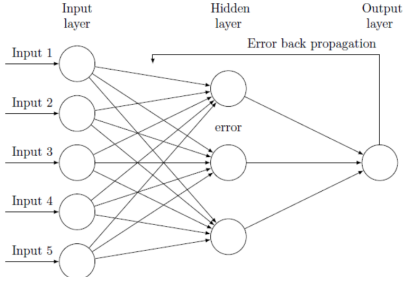
$$\nabla_{\mathbf{x}} z = \left(\frac{\partial \mathbf{y}}{\partial \mathbf{x}}\right)^T \nabla_{\mathbf{y}} z$$

where $\left(\frac{\partial y}{\partial x}\right)$ is the Jacobian (n x m) of $g()$

Back-propagation

- Backpropagation is a recursive application of the chain rule from the cost function
  - Forward pass compute the output of the network
  - Cost function computes the error between expected output and actual output of the network
  - Backpropagation evaluates individual gradients of each parameter and propagates them backward to update them

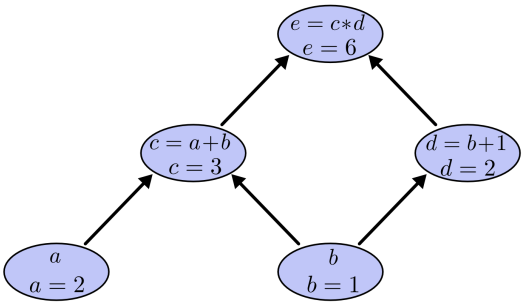




Forward-pass

- A feedforward network takes as input x and outputs $\hat{y}$
- Information flows from layer to layer ("forward propagation")

$$\mathbf{h}^{(1)} = g^{(1)}\left(\mathbf{W}^{(1)}\mathbf{x}^{(1)} + \mathbf{b}^{(1)}\right)$$
$$\mathbf{h}^{(2)} = g^{(2)}\left(\mathbf{W}^{(2)}\mathbf{h}^{(1)} + \mathbf{b}^{(2)}\right)$$
$$\dots$$
$$\hat{\mathbf{y}} = g^{(d)}\left(\mathbf{W}^{(d)}\mathbf{h}^{(d-1)} + \mathbf{b}^{(d)}\right)$$

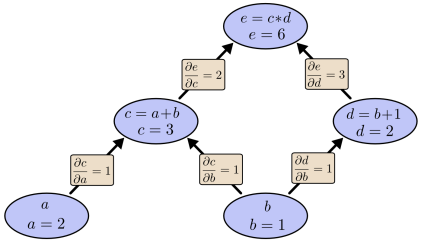


Derivatives in computational graph

- Local derivates of connected nodes are computed locally on the edges of the graph
- For non-connected nodes
  - product of edges connected between the nodes
  - sum over all paths

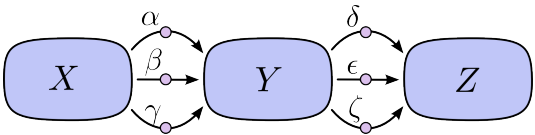
Just application of chain rule !

$$\frac{\partial e}{\partial b} = \frac{\partial e}{\partial c}\frac{\partial c}{\partial b} + \frac{\partial e}{\partial d}\frac{\partial d}{\partial b} = 2 \times 1 + 3 \times 1 = 5$$

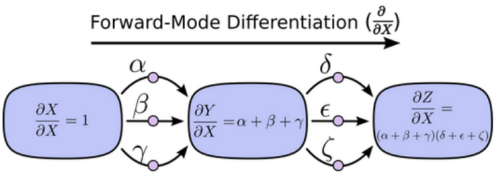


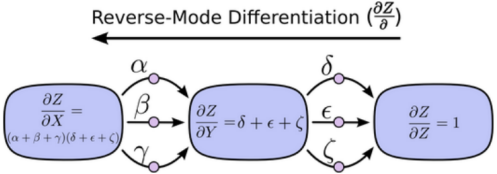
Derivatives in computational graph

- **Problem**: summing over all paths can quickly become computationally intractable
  - $\frac{\partial Z}{\partial X} = \alpha\delta + \alpha\epsilon + \alpha\zeta + \beta\delta + \beta\epsilon + \beta\zeta + \gamma\delta + \gamma\epsilon + \gamma\zeta$



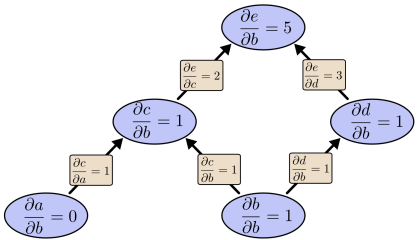
- Simplication by factorization: $\frac{\partial Z}{\partial X} = (\alpha + \beta + \gamma)(\delta + \epsilon + \zeta)$
  - 2 "modes", both doing only 1 path per node.

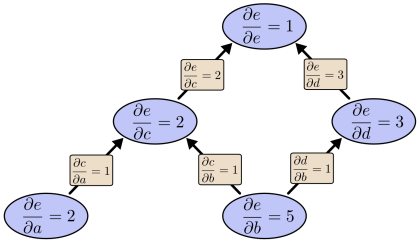




Why backpropagation?

- Which propagation mode to choose?





- reverse-mode differentiation (right) allows to obtain the derivative of the output with respect to **every** node directly in one pass $\rightarrow$**MASSIVE** parallelization (10 millions + in practice)
  - forward-mode differentiation only brings the derivative of one node with respect to one input during one pass

Next course

- Optimization
- Regularization
- TP: introduction to PyTorch, backpropagation, first NN training