Deep learning 1/2

Mathurin Massias

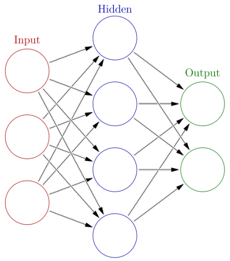
https://mathurinm.github.io

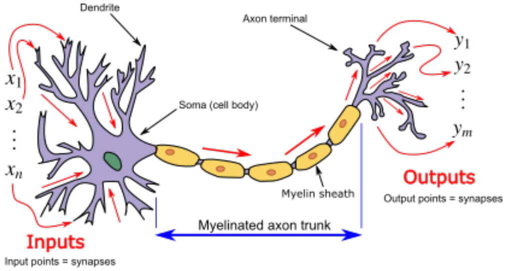
Inria, OCKHAM team

12/03/2025

Artificial neural networks (ANNs)

- Computing systems inspired by the biological neural networks that constitute biological brains.
- Collection of connected units or nodes called artificial neurons
- An artificial neuron receives signals then processes them and can signal neurons connected to it.
- Connection edge, like synapses, can transmit a signal (a real number) to other neurons.



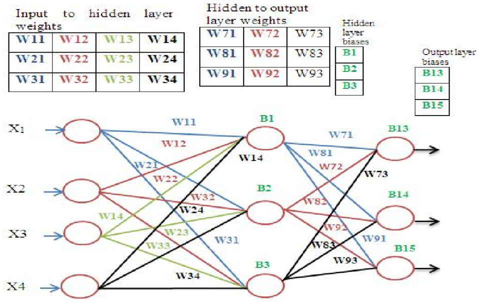


ANNs: weighted directed graphs

ANNs are parametric models

$$\boldsymbol{\theta} = (w_{1,1}, w_{1,2}, \ldots, b_1, \ldots, w_{9,1}, w_{9,2}, \ldots, b_{15})$$

- Neurons and edges typically have a weight that increases or decreases the strength of the signal at a connection.



Input to hidden layer weights

W11	W12	W13	W14
W21	W22	W23	W24
W31	W32	W33	W34

Hidden to output layer weights

W71	W72	W73
W81	W82	W83
W91	W92	W93

Hidden layer biases

B1
B2
B3

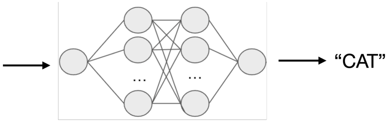
Output layer biases

B13
B14
B15

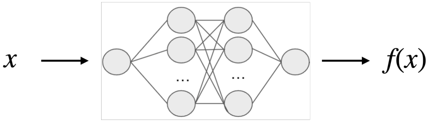
ANNs: a flexible model

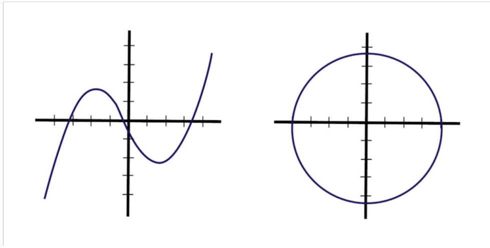
ANNs are parametric models used for regression or classification

Classification

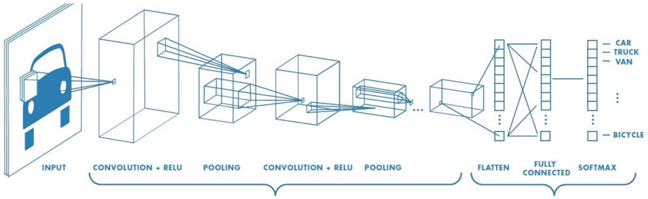


Regression

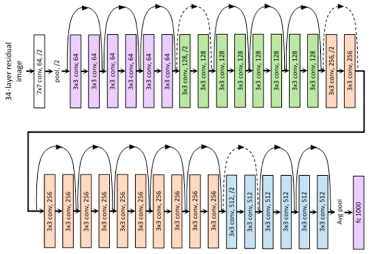




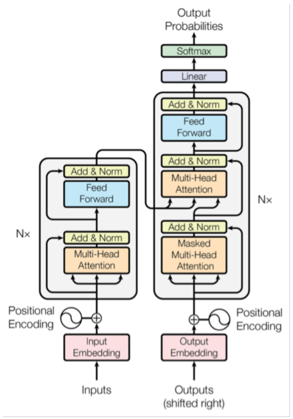
ANNs: many different architectures



(a)

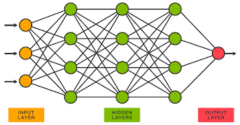


(b)

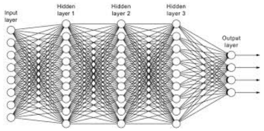


Feedforward neural networks

For today we study just the simple feedforward, fully connected neural network model



which however can be already quite complex (million of variables)



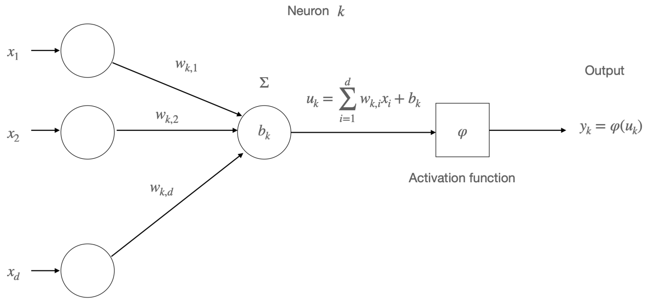
ANNs: two main tasks

- (supervised) Training: how the optimal values of the weights are found based on the training set
- Forward pass: after training, how the input is propagated through the network to produce the output

Differently from other models we have seen, the forward pass may be expensive ! (long series of large matrix-vector multiplications)

Forward pass: information propagation through a neuron

- Neurons and edges typically have a weight that increases or decreases the strength of the signal at a connection.

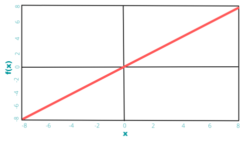


Output of neuron k:

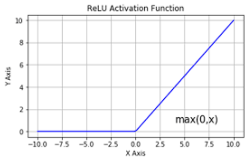
$$y_k = \underbrace{\varphi(\underbrace{\mathbf{w}_k^T \mathbf{x} + b_k}_{\text{linear}})}_{\text{non linear}}$$

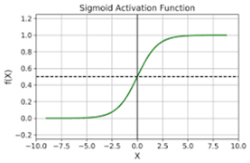
Activation functions: introduce nonlinearity

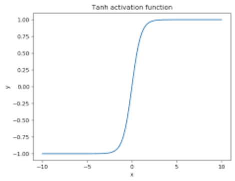
- Linear $\varphi(x) = x$
- Rectified linear (ReLU) $\varphi(x) = \max(x, 0)$
- Sigmoid $\varphi(x) = \frac{1}{1+e^{-x}}$
- Hyperbolic tangent $\varphi(x) = \tanh(x)$
- ELU, GELU, SiLU, CELU, PReLU, LeakyReLU, Swish, ...



Linear Activation Function

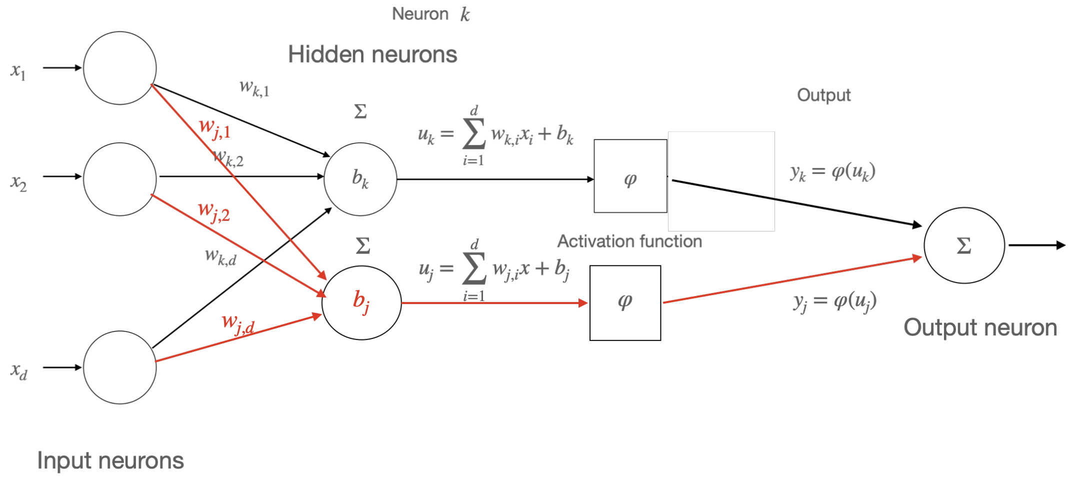




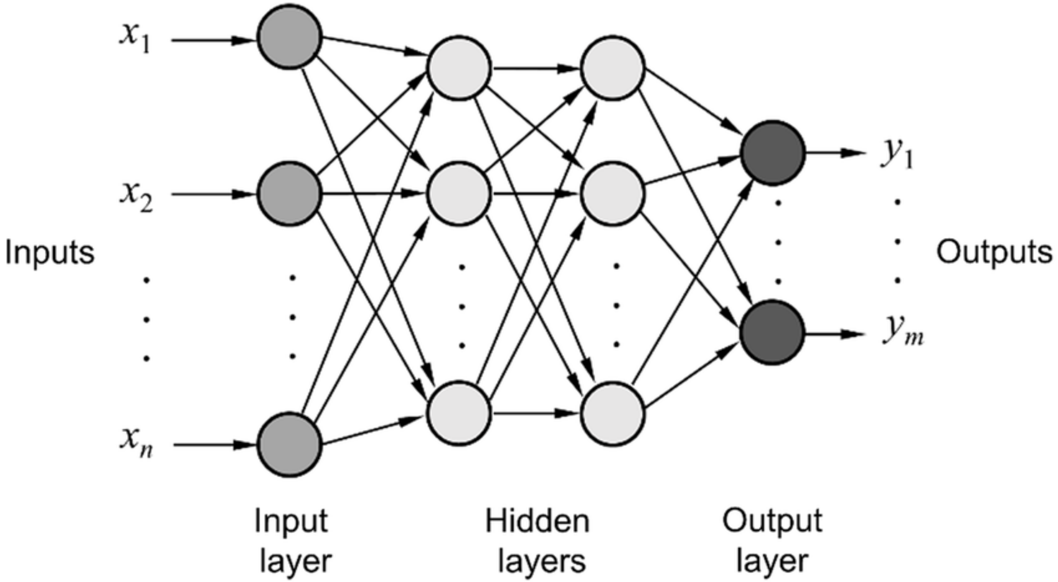


φ is applied componentwise: For $\mathbf{x} \in \mathbb{R}^n$, $\varphi(\mathbf{x}) = \begin{bmatrix} \varphi(x_1) \\ \vdots \\ \varphi(x_n) \end{bmatrix}$

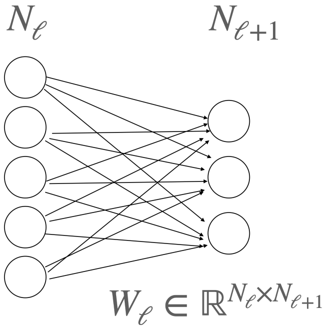
Multiple neurons



Multiple layer networks



Matrix notation



TODO W_ℓ has wrong shape The output of layer ℓ is

$$y_\ell = \varphi(W_\ell y_{\ell-1} + \mathbf{b}_\ell)$$

The output of the network is

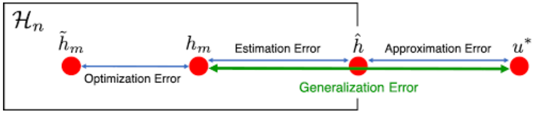
$$F(x) = \varphi(W_L(\varphi(W_{L-1}(\dots\varphi(W_1\varphi(W_0 x + \mathbf{b}_0) + \mathbf{b}_1)\dots + \mathbf{b}_{L-1}) + \mathbf{b}_L)$$

$$F(x) = y_L \circ \dots \circ y_0(x)$$

How good are neural networks?

Three error components:

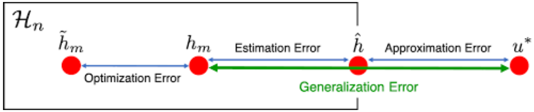
- approximation error
- statistical error
- optimization error



How good are neural networks?

Three error components:

- approximation error
- statistical error
- optimization error



- $\tilde{h}_m$ our network,
- h_m a perfectly trained network on the dataset,
- $\hat{h}$ function minimizing the problem with infinitely many data,
- u^* the function we are trying to model

Outline

Approximation property: two universal approximation theorems

Theorem (1 layer on compact sets)

Let ϕ be non polynomial. Let $f : \mathbb{R}^n \to \mathbb{R}^d$ be continuous. Then for every compact $K \subset \mathbb{R}^n$, every $\varepsilon > 0$, there exists $k \in \mathbb{N}$, $A \in \mathbb{R}^{k \times n}, b \in \mathbb{R}^k, C \in \mathbb{R}^{m \times k}$ such that:

$$\sup_{x \in K} \|f(x) - g(x)\| \leq \varepsilon$$

where $g(x) = C\phi(Ax + b)$.

Approximation property: two universal approximation theorems

Theorem (ReLU activation, arbitrary depth, minimal width)

For any Lebesgue p-integrable function $f : \mathbb{R}^n \to \mathbb{R}^m$ and any $\varepsilon > 0$, there exists a fully connected ReLU network F of width exactly $d_m = \max\{n+1, m\}$, satisfying

$$\int_{\mathbb{R}^n} \|f(x) - F(x)\|^p \, \mathrm{d}x < \varepsilon.$$

Moreover, there exists a function $f \in L^p(\mathbb{R}^n, \mathbb{R}^m)$ and some $\varepsilon > 0$ for which there is no fully connected ReLU network of width less than $d_m = \max\{n+1, m\}$ satisfying the above approximation bound.

Many other variants exists with different assumptions

Outline

NNs are universal approximators

Training: SGD

Training: momentum

How to compute the gradients: backpropagation

How to train a neural network?

Given the training set $\{(\mathbf{x}_1, y_1), \dots, (\mathbf{x}_n, y_n)\}$, find the best parameters $\boldsymbol{\theta} \in \mathbb{R}^m$, $\boldsymbol{\theta} = \{W_1, \mathbf{b}_1, \dots, W_L, \mathbf{b}_L\}$ to fit the data.

ERM principle

$$\min_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}) = \mathbb{E}_{(X,Y)} \ell(Y, F_{\boldsymbol{\theta}}(X))) \sim \min_{\boldsymbol{\theta}} \mathcal{L}_n(\boldsymbol{\theta}) := \frac{1}{n} \sum_{i=1}^{n} \ell(y_i, F_{\boldsymbol{\theta}}(\mathbf{x}_i))$$

Minimization problem

$$\min_{\boldsymbol{\theta}} f(\boldsymbol{\theta}) \quad f : \mathbb{R}^m \to \mathbb{R}, \quad f(\boldsymbol{\theta}) = \sum_{i=1}^{n} f_i(\boldsymbol{\theta})$$

with m, n very large

Gradient descent (GD)

- Input: $\boldsymbol{\theta}(0)$ starting weights
- Update rule: for $k = 1, \dots,$ set $\boldsymbol{\theta}(k+1) = \boldsymbol{\theta}(k) - \alpha(k)\nabla f(\boldsymbol{\theta}(k))$
- Stopping criterion: stop if $\|\nabla f(\boldsymbol{\theta}(k))\| < \epsilon$

Limitations of GD in the training framework

- Compute $\nabla f(\boldsymbol{\theta}(k)) = \sum_{i=1}^{n} \nabla f_i(\boldsymbol{\theta}(k))$ with n very large: expensive
- Compute $\alpha(k)$: best way line-search $\rightarrow m$ very large: expensive

How to improve gradient descent?

$$\boldsymbol{\theta}(k+1) = \boldsymbol{\theta}(k) - \alpha(k)\frac{1}{n}\sum_{i=1}^{n}\nabla_{\boldsymbol{\theta}}\ell(y_i, F_\theta(\mathbf{x}_i))$$

$\hookrightarrow$ Replace the mean by a cheaper estimate?

Reduce the cost of the training

Stochastic gradient descent (SGD) for NN training

For all $k \geq 1$, choose randomly $i_k \in \{1, \dots, n\}$ and set

$$\boldsymbol{\theta}(k+1) = \boldsymbol{\theta}(k) - \alpha(k) \nabla_{\boldsymbol{\theta}} \mathcal{L}(y_{i_k}, F_\theta(\mathbf{x}_{i_k}))$$

Advantages: much cheaper

- Uses just one sample for the gradient: $\nabla f_{i_k}(\boldsymbol{\theta}(k))$ rather than $\sum_{i=1}^n \nabla f_i(\boldsymbol{\theta}(k))$
- Heuristic strategies to update $\alpha(k)$ or fixed α

Disadvantages: slow convergence

- Uses just partial information
- In general, needs $\lim_{k\to\infty} \alpha(k) = 0$ to converge

Convergence results

Assume f strongly convex, $\boldsymbol{\theta} \in \mathbb{R}^m$

- GD converges linearly:

$$f(\boldsymbol{\theta}(k)) - f^* \leq O(\rho^k), \quad \rho \in (0,1),$$

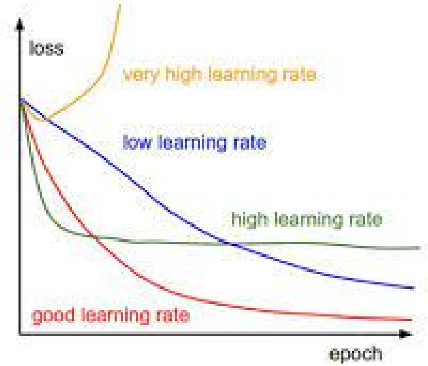
$\rightarrow$ number of iterations is proportional to $\log(1/\epsilon)$.

- SGD with $\alpha_k = O(1/k)$ converges sublinearly in expectation [Theorem 4.7 `https://epubs.siam.org/doi/epdf/10.1137/16M1080173`]:

$$\mathbb{E}(f(\boldsymbol{\theta}(k)) - f^*) = O(1/k).$$

- GD cost : $m \log(1/\epsilon)$ (each iteration costs m)
- SGD cost: $1/\epsilon$, but does not depend on m!
- In the big data regime where m is large, $m \log(1/\epsilon) \gg 1/\epsilon$.
- exercise: compare 1 iteration cost for $f(\boldsymbol{\theta}) = \frac{1}{2}\|X\boldsymbol{\theta} - y\|^2$

Choosing the learning rate

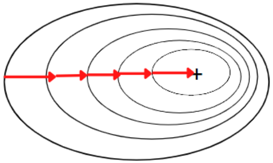


Iterations vs epochs

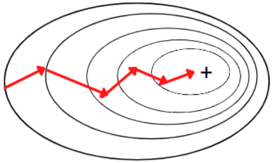
Iteration: one update of the parameters, epoch: a full pass over the data

SGD vs GD, minibatches

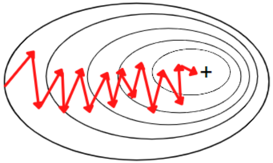
Batch Gradient Descent



Mini-Batch Gradient Descent



Stochastic Gradient Descent



Outline

NNs are universal approximators

Training: SGD

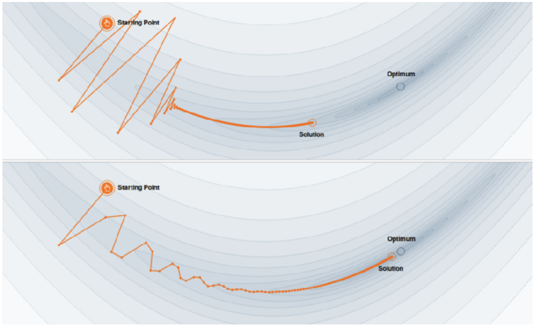
Training: momentum

How to compute the gradients: backpropagation

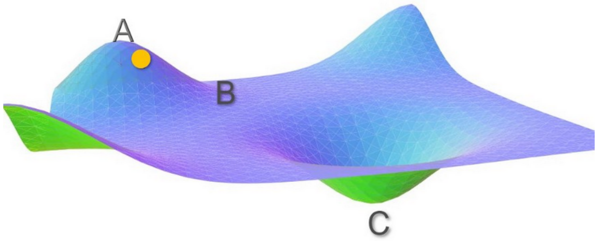
Improvement #2: momentum

- The gradient in a plateau is negligible or zero $\rightarrow$ very small steps.
- The path followed by gradient descent is very jittery

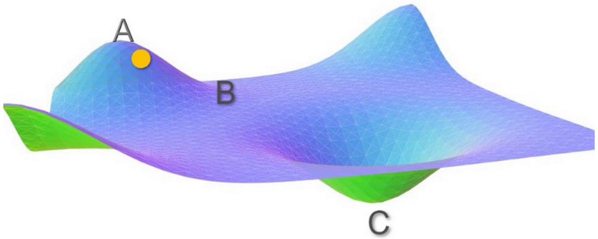
`https://distill.pub/2017/momentum/`



Momentum: keep memory of the past



Momentum: keep memory of the past



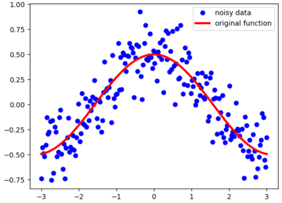
- Define the step as the average of past gradients instead of the gradient at the current iteration.
- Cannot consider all the gradients with equal weightage.
- Need to use some sort of weighted average

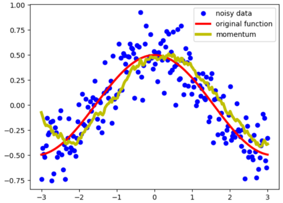
Exponential Moving Average (EMA)

Consider a noisy sequence $y(t)$. The EMA $s(t)$ for a series $y(t)$ may be calculated recursively as:

$$s(t) = \begin{cases} y(1) & \text{if } t = 1, \\ \beta s(t-1) + (1-\beta) y(t) & \text{if } t > 1 \end{cases}$$

where $\beta \in [0, 1]$ represents the degree of weighting increase. A lower β discounts older observations faster.





EMA for gradients

- Instead of $\Delta\boldsymbol{\theta}(k) = \gamma\mathbf{g}(k)$, where $\mathbf{g}(k)$ is the gradient approximation (full gradient, or a mini-batch or stochastic gradient), use

$$\Delta\boldsymbol{\theta}(k) = \beta\Delta\boldsymbol{\theta}(k-1) + (1-\beta)\gamma\mathbf{g}(k)$$

- Often $\beta \to \beta(k)$ and $(1-\beta)\gamma \to \alpha(k)$

$$\boldsymbol{\theta}(k+1) = \underbrace{\boldsymbol{\theta}(k) - \alpha(k)g(k)}_{\text{standard gradient step}} + \underbrace{\beta(k)(\boldsymbol{\theta}(k) - \boldsymbol{\theta}(k-1))}_{\text{momentum term}},$$

- Special cases:
    - $\beta_k = 0$ for all $k \in \mathbb{N}$: classical GD/SGD
    - $\alpha_k = \alpha$ and $\beta_k = \beta$: *heavy ball method*.

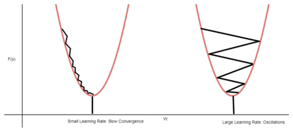
Heavy ball momentum

- By expanding the update:

$$\boldsymbol{\theta}(k+1) = \boldsymbol{\theta}(k) - \alpha \sum_{j=1}^{k} \beta^{k-j} g(k)$$

  each step is an exponentially decaying average of past gradients.
- Let’s analyze the contribution of β. Assume $\alpha = 1$.
  - $\beta = 0.1$: At $k = 3$; g_3 will contribute 100% of its value, g_2 10% and g_1 1%: contribution from earlier gradients decreases rapidly.
  - $\beta = 0.9$: g_3 will contribute 100% of its value, g_2 90% and g_1 81%.
  - Usually $\beta \sim 0.9$

How does momentum help?


F(x)
Small Learning Rate: Slow Convergence
W
Large Learning Rate: Oscillations

With gradient descent:

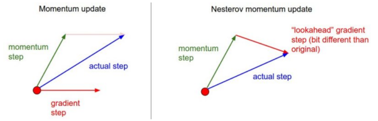
- LR too small: small steps, convergence takes a lot of time even when the gradient is high.
- LR too high: the sequence oscillates around the minima

How does momentum fix this?

- *All the past gradients have the same sign*: the summation term will become large and we will take large steps
- *Different signs*: the summation term will become small and the steps will be small, damping the oscillations.

Nesterov accelerated gradient vs heavy ball

- Treats the future approximate position $\tilde{\boldsymbol{\theta}}(k) = \boldsymbol{\theta}(k) + \beta(k)(\boldsymbol{\theta}(k) - \boldsymbol{\theta}(k-1))$ as a ”lookahead”
- It computes the gradient at $\tilde{\boldsymbol{\theta}}(k)$ instead of at the old position $\boldsymbol{\theta}(k)$



Nesterov momentum. Instead of evaluating gradient at the current position (red circle), we know that our momentum is about to carry us to the tip of the green arrow. With Nesterov momentum we therefore instead evaluate the gradient at this "looked-ahead" position.

- In momentum: first gradient descent step and then momentum term, in Nesterov: first momentum then gradient descent (with the gradient evaluated at $\tilde{\boldsymbol{\theta}}(k)$, not at $\boldsymbol{\theta}(k)$).
- Nesterov has better theoretical rates: see Chap 5 in M2 Lecture Notes: `https://mathurinm.github.io/assets/2022_ens/class.pdf`

Outline

Reminder: the Jacobian and the chain rule

On board; check Section 0.1 of M2 notes:
`https://mathurinm.github.io/assets/2022_ens/class.pdf`

How to actually compute the gradient?

Backpropagation algorithm

There exists a very efficient algorithm, called the backpropagation algorithm `http://neuralnetworksanddeeplearning.com/chap2.html`.

Backpropagation is about understanding how changing the weights and biases in a network affects the loss function.

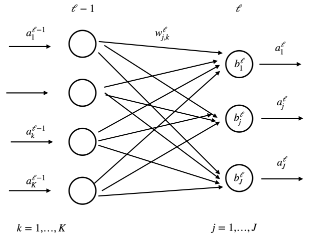
Given the loss function $\mathcal{L}(y(\mathbf{x}), a^L(\mathbf{x}))$ where $a^L(\mathbf{x})$ is the output of the network, $y(\mathbf{x})$ the true output, we want to compute $\frac{\partial \mathcal{L}}{\partial w}$ and $\frac{\partial \mathcal{L}}{\partial b}$ for every weight w and bias b

Mathematical formulation

Recall the propagation rule in a neural network:

$$a_j^\ell = \sigma\left(\sum_k w_{j,k}^\ell a_k^{\ell-1} + b_j^\ell\right)$$

- $\ell = 1, \ldots, L$ indexing the layers
- $k = 1, \ldots, K$ indexing the neurons in the $\ell - 1$-th layer
- $j = 1, \ldots, J$ indexing the neurons in the ℓ-th layer



Mathematical formulation

In vector form:

$$\begin{aligned} \mathbf{a}^\ell &= \sigma(W^\ell \mathbf{a}^{\ell-1} + \mathbf{b}^\ell) \\ \mathbf{z}^\ell &= W^\ell \mathbf{a}^{\ell-1} + \mathbf{b}^\ell \qquad \text{(preactivation)} \end{aligned}$$

with

$$W \in \mathbb{R}^{J\times K}, \mathbf{a}^{\ell-1} \in \mathbb{R}^K,\ \mathbf{b}^\ell \in \mathbb{R}^J$$

and σ applied componentwise.

Hadamard product

Given two vectors of the same size $\mathbf{x}, \mathbf{y} \in \mathbb{R}^m$, the Hadamard product is the componentwise product:

$$\mathbf{x} \odot \mathbf{y} = \begin{bmatrix} x_1 y_1 \\ x_2 y_2 \\ \vdots \\ x_m y_m \end{bmatrix}$$

Example

$$\begin{bmatrix} 1 \\ 3 \end{bmatrix} \odot \begin{bmatrix} 4 \\ 7 \end{bmatrix} = \begin{bmatrix} 4 \\ 21 \end{bmatrix}$$

Backpropagation: 4 fundamental equations

Backpropagation is based on 4 equations that allows to compute the gradient in an ”analytical” way.
We define an intermediary quantity that will be useful:

$$\delta_j^\ell := \frac{\partial \mathcal{L}}{\partial z_j^\ell}$$

Backpropagation: an equation for δ in the output layer

Scalar case $a^L \in \mathbb{R}$:

$$\delta^L = \frac{\partial \mathcal{L}}{\partial a^L} \sigma'(z^L) \tag{BP1}$$

- Everything in (BP1) is easily computed: we compute z^L in the forward pass, and it's only a small additional overhead to compute $\sigma'(z^L)$. For instance for the sigmoid $\sigma'(x) = \sigma(x)(1 - \sigma(x))$
- The exact form of $\frac{\partial \mathcal{L}}{\partial a^L}$ will depend on the choice of loss, but can be computed analytically
- Exercice: for the quadratic loss?

Backpropagation: an equation for δ in the output layer

Scalar case $a^L \in \mathbb{R}$:

$$\delta^L = \frac{\partial \mathcal{L}}{\partial a^L} \sigma'(z^L) \tag{BP1}$$

- Everything in (BP1) is easily computed: we compute z^L in the forward pass, and it's only a small additional overhead to compute $\sigma'(z^L)$. For instance for the sigmoid $\sigma'(x) = \sigma(x)(1-\sigma(x))$
- The exact form of $\frac{\partial \mathcal{L}}{\partial a^L}$ will depend on the choice of loss, but can be computed analytically
- Exercice: for the quadratic loss?

$$\mathcal{L} = \frac{1}{2n} \sum_{i=1}^{n} (y_i - a^L(\mathbf{x}_i))^2$$

$$\frac{\partial \mathcal{L}}{\partial a^L} = \frac{1}{n} \sum_{i=1}^{n} (y_i - a^L(\mathbf{x}_i))$$

Proof of BP1: the chain rule

By definition

$$\delta_1^L = \delta^L = \frac{\partial \mathcal{L}}{\partial z^L}$$

By the chain rule

$$\delta^L = \frac{\partial \mathcal{L}}{\partial a^L} \frac{\partial a^L}{\partial z^L}$$

and because $a^L = \sigma(z^L)$

$$\frac{\partial a^L}{\partial z^L} = \sigma'(z^L).$$

Backpropagation: a recursive equation for δ

$$\boldsymbol{\delta}^{\ell} = (W^{\ell+1})^{T}\boldsymbol{\delta}^{\ell+1} \odot \sigma'(\boldsymbol{z}^{\boldsymbol{\ell}}) \qquad \text{(BP2)}$$

Why backpropagation?

By combining (BP2) with (BP1) we can compute the error for any layer in the network. We start by using (BP1) to compute δ^{L}, then apply (BP2) to compute $\boldsymbol{\delta}^{L-1}$, then (BP2) again to compute $\boldsymbol{\delta}^{L-2}$, and so on, all the way back through the network

Backpropagation: an equation for the rate of change of the loss with respect to any bias

$$\frac{\partial \mathcal{L}}{\partial b_j^\ell} = \delta_j^\ell \qquad \text{(BP3)}$$

Good news, δ_j^ℓ is exactly equal to the rate of change $\frac{\partial \mathcal{L}}{\partial b_j^\ell}$ and (BP1) and (BP2) have already told us how to compute it!

Backpropagation: An equation for the rate of change of the loss with respect to any weight

$$\frac{\partial \mathcal{L}}{\partial w_{j,k}^{\ell}} = a_k^{\ell-1} \delta_j^{\ell} \tag{BP4}$$

This tells us how to compute the partial derivatives $\frac{\partial \mathcal{L}}{\partial w_{j,k}^{\ell}}$ in terms of the quantities δ^{ℓ}, $\mathbf{a}^{\ell-1}$, which we already know how to compute.

The backpropagation algorithm

- Given input $\mathbf{x}$, set the corresponding activation $\mathbf{a}^1$ for the input layer: $\mathbf{a}^1 = \sigma(W^1\mathbf{x} + \mathbf{b}^1)$
- Feedforward pass: for each $\ell = 2, 3, \ldots, L$ compute $\mathbf{z}^\ell = W^\ell \mathbf{a}^{\ell-1} + \mathbf{b}^\ell$ and $\mathbf{a}^\ell = \sigma(\mathbf{z}^\ell)$.
- Compute $\delta^L = \frac{\partial \mathcal{L}}{\partial a^L}\sigma'(z^L)$.
- Backpropagate the error: For each $\ell = L-1, L-2, \ldots, 1$ compute $\boldsymbol{\delta}^\ell = ((W^{\ell+1})^T \boldsymbol{\delta}^{\ell+1}) \odot \sigma'(\mathbf{z}^\ell)$.
- Output: The gradient of the cost function is given by $\frac{\partial \mathcal{L}}{\partial w^\ell_{j,k}} = a_k^{\ell-1}\delta_j^\ell$ and $\frac{\partial \mathcal{L}}{\partial b_j^\ell} = \delta_j^\ell$

Full training of a neural network

For each epoch:

- At each iteration select a batch B of training samples.
- For each training example $\mathbf{x}$ in the batch perform the following steps:
    - Forward pass
    - Compute $\delta^L = \frac{\partial \mathcal{L}_B}{\partial a^L} \sigma'(z^L)$.
    - Backpropagate the error and obtain $\frac{\partial \mathcal{L}_B}{\partial w^\ell_{j,k}}$ and $\frac{\partial \mathcal{L}_B}{\partial b^\ell_j}$
- Gradient descent: For each $\ell = L, L-1, \ldots, 1$ update the parameters:

$$w^\ell_{j,k} \leftarrow w^\ell_{j,k} - \alpha \frac{\partial \mathcal{L}_B}{\partial w^\ell_{j,k}}$$

$$b^\ell_j \leftarrow b^\ell_j - \alpha \frac{\partial \mathcal{L}_B}{\partial b^\ell_j}$$

- When all the training examples have been used, start a new epoch

Backpropagation is a fast algorithm

- Alternative to compute the derivatives for some weight w: finite differences

$$\frac{\partial \mathcal{L}}{\partial w_j} \sim \frac{\mathcal{L}(w + he_j) - \mathcal{L}(w)}{h}$$

  where $h > 0$ small, e_j is the unit vector in the jth direction.
- It's simple conceptually, easy to implement, but extremely slow!
- For each w_j we need to compute $\mathcal{L}(w + he_j)$: millions of weights $\rightarrow$ compute the loss function a million times, thus a million forward passes through the network.
- With backpropagation we simultaneously compute all the partial derivatives with just one forward pass through the network, followed by one backward pass
- Computational cost of the backward pass is about the same as the forward pass (dominant cost in the forward pass is multiplying by the weight matrices, while in the backward pass it's multiplying by the transposes of the weight matrices)
- Total cost of backpropagation is roughly the same as making just two forward passes through the network.