



Artificial Intelligence II: Deep learning methods

Dragoș Burileanu, Ana Neacșu & Vlad Vasilescu, Georgian Nicolae
Lecture 3: Deep Neural Networks

National University of Science and Technology POLITEHNICA Bucharest, Romania
BIOSINF Master Program

March 2024

Overview

1 Intro

2 Transfer operators

3 Differential programming

4 Gradient descent algorithms

- First Order methods

Intro

Radial Basis Function Nets

RBF

Consider pairs $(x, y) \in \mathbb{R}^d$.

- 1-layer architecture, which learns means and variances
 $\forall i \in [0 \dots N - 1] \quad (\mu_i, \sigma_i)$ as “weights”.

dat \downarrow

param \uparrow

$$\underline{\phi(x)} = \begin{pmatrix} \exp \frac{1}{2\sigma_0^2} \\ \vdots \\ \exp \frac{-||x - \mu_{N-1}||^2}{2\sigma_{N-1}^2} \end{pmatrix}$$

$$x \in \mathbb{R} \rightarrow \mathbb{R}^n$$

- we model $y = g(w^T \phi(x))$, where g is called the transfer function
 - can be used to solve different problems like: *regression, binary classification* and *multi-classification*. Classification

11

multi-classification

7d

Learning mechanism

- Place the centers and variances randomly and uniformly, by vector quantization (K-means++, GNG, etc.)

SGS-W

2-steps learning

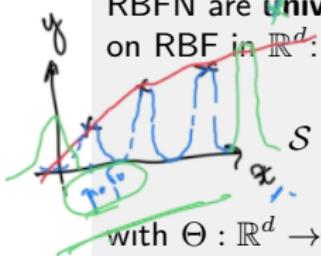
- ① fit the centers (μ_i) and variances (σ_i).
 - ② fit the weights w . — SGD

3-steps learning

- ① fit the centers (μ_i) and variances (σ_i). — k -means
 - ② fit the weights w . — SGD
 - ③ fit everything ($\nabla_{\mu}\mathcal{L}$, $\nabla_{\sigma}\mathcal{L}$, $\nabla_w\mathcal{L}$) where \mathcal{L} is the cost function.

Universal approximator theorem

RBFN are universal approximators. Denote \mathcal{S} the family of functions based on RBF in \mathbb{R}^d : $\mathcal{S} = \{f : \mathbb{R}^d \rightarrow \mathbb{R}\}$



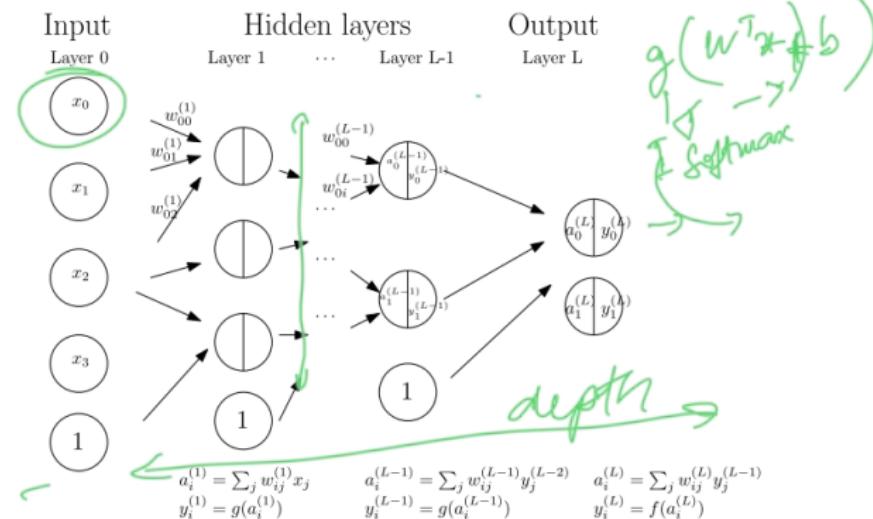
$$\mathcal{S} = \left\{ \zeta : \mathbb{R}^d \rightarrow \mathbb{R}, \zeta(x) = \sum_i \Theta\left(\frac{x - \mu_i}{\sigma_i}\right), w \in \mathbb{R}^N \right\},$$

with $\Theta : \mathbb{R}^d \rightarrow \mathbb{R}$ continuous (almost everywhere) and $\int_{\mathbb{R}^d} \Omega(x) dx \neq 0$.

Then \mathcal{S} is dense in $\ell_p(\mathbb{R})$ for every $p \in [1, \infty)$.

Feedforward Neural Network Architecture

—
Dense



Notations

- **Width** : number of neurons per layer
 - **Depth** : number of weight layers
 - **Parameters** : weight matrices and bias vector associated with every layer
 - **Activation** : a hidden (non-linear) transfer function, denoted with f for layers $l \in \{1 \dots L-1\}$ and g for the last layer L .

Intro
○○○○

Transfer operators
●○○○○○○○○○○

Differential programming
○○○○○○○○○○○○

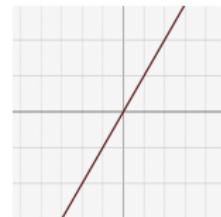
Gradient descent algorithms
○○○○○○○○○○○○○○○○

Transfer operators

Classic Activation Operators

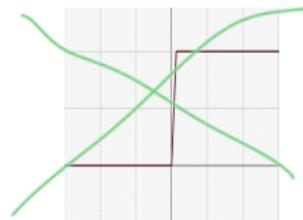
- Identity: $\rho : \mathbb{R} \rightarrow \mathbb{R}$ $\rho(x) = x$

281



- Binary step:

$$\rho : \mathbb{R} \rightarrow \{0, 1\} \quad \rho(x) = \begin{cases} 0, & \text{if } x < 0 \\ 1, & \text{if } x \geq 0 \end{cases}$$



- Sigmoid: $\rho : \mathbb{R} \rightarrow (0, 1)$ $\rho(x) = \frac{1}{1+e^{-x}}$



Classic Activation Operators

$$\tanh x = \frac{\sinh x}{\cosh x}$$


- Hyperbolic tangent:

$$\rho : \mathbb{R} \rightarrow (-1, 1) \quad \rho(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

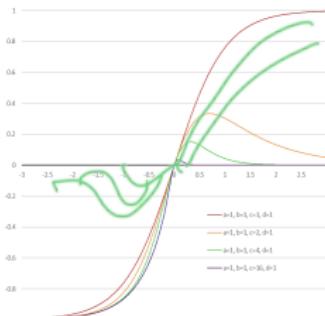
$$\frac{d}{dx} \tanh(x) = \frac{\text{sech}^2 x \cosh x - \text{sech} x \cosh^2 x}{\text{sech}^3 x - \text{sech}^2 x} = \frac{1 - \tanh^2 x}{1 + \tanh^2 x}$$

$$\frac{d}{dx} g(x) = 1 - g^2(x)$$

- Soboleva modified hyperbolic tangent:

$$\rho : \mathbb{R} \rightarrow (-1, 1) \quad \rho(x) = \frac{e^{ax} - e^{-bx}}{e^{cx} + e^{-dx}}$$

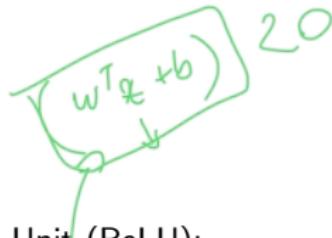
$a, b, c, d \in \mathbb{R}$



Saturating function usually suffer from **vanishing gradient** problem (more on that later).



Modern Activation Operators



- Rectified Linear Unit (ReLU):
 $\rho : \mathbb{R} \rightarrow [0, \infty) \quad \rho(x) = \max(0, x)$

$$\frac{\partial \rho(x)}{\partial x} = \begin{cases} 0 & ; x \leq 0 \\ 1 & ; x > 0 \end{cases}$$



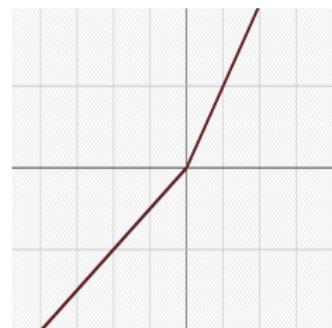
More favorable for the gradient flow, but may suffer from **dying ReLU** problem.

Other variants have been proposed to mitigate this problem, see next slide.

ReLU variants

- Leaky Rectified Linear Unit:

$$\rho(x) = \begin{cases} 0.01x & \text{if } x \leq 0 \\ x & \text{if } x > 0 \end{cases}$$



- Parametric Rectified Linear Unit:

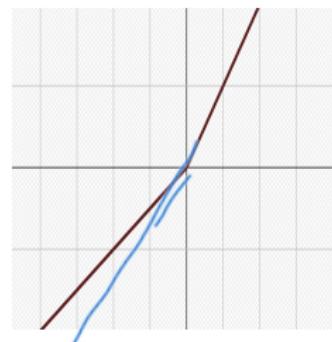
$$\rho : \mathbb{R} \rightarrow \mathbb{R}$$

$$\rho(x) = \begin{cases} \alpha x & \text{if } x \leq 0 \\ x & \text{if } x > 0, \end{cases}$$

hyperparameter

with $\alpha \in \mathbb{R}^+$.

↳ bearable



Other ReLU variants

$$\text{Enf} = \frac{2}{\sqrt{\pi}} \int_{-\infty}^{\infty} e^{-x^2} dt \quad (\text{erf})'(x) = \frac{2}{\sqrt{\pi}} e^{-x^2} = q$$

- Gaussian Error Linear Unit (GELU):

$$\rho : \mathbb{R} \rightarrow [-0.17, \infty)$$

$$\rho(x) = \frac{1}{2}x \left(1 + \operatorname{erf}\left(\frac{x}{\sqrt{2}}\right)\right) = x\Phi(x),$$

where $\Phi(x) = P(X \leq x)$, $X \approx \mathcal{N}(0, 1)$ is the cumulative distribution function of the standard normal distribution.

$$\frac{\partial}{\partial x} \varphi(x) = \Phi(x) + x \psi(x)$$

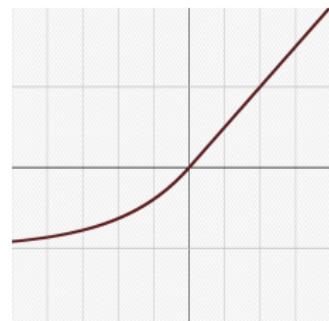
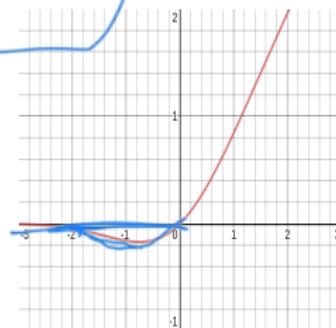
- Exponential Linear Unit (ELU):

$$\rho : \mathbb{R} \rightarrow [-\alpha, \infty)$$

$$\rho(x) = \begin{cases} \alpha(e^x - 1) & \text{if } x \leq 0 \\ x & \text{if } x > 0 \end{cases}$$

where $\alpha \in \mathbb{R}^+$.

$$\text{where } \alpha \in \mathbb{R}^+. \quad f(x) = \begin{cases} \alpha e^x; & x \leq 0 \\ 1; & x > 0 \end{cases}$$



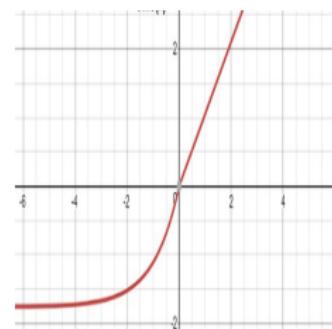
More ReLU activation functions

- Scaled Exponential Linear Unit (SELU):

$$\rho : \mathbb{R} \rightarrow [-\alpha\lambda, \infty)$$

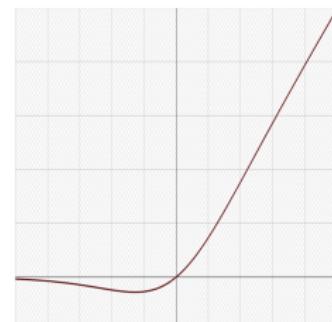
$$\rho(x) = \lambda \begin{cases} \alpha(e^x - 1) & \text{if } x \leq 0 \\ x & \text{if } x > 0 \end{cases}$$

where $\alpha = 1.67326$ and $\lambda = 1.0507$.



- Sigmoid Linear Unit (SiLU): $\rho : \mathbb{R} \rightarrow [-0.278, \infty)$

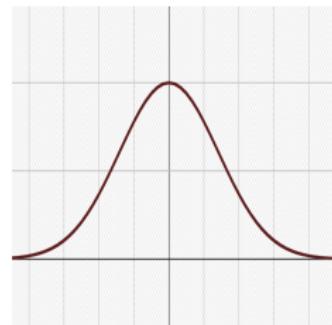
$$\rho(x) = \frac{x}{1+e^{-x}},$$



Other activation functions

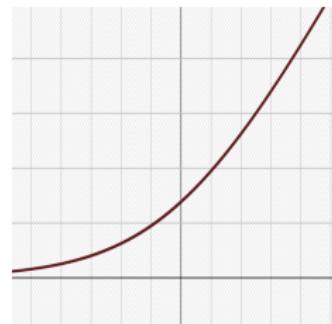
- Gaussian: $\rho : \mathbb{R} \rightarrow [0, 1] \quad \rho(x) = e^{-x^2}$

$$\underline{\hspace{1cm}} \quad \underline{\hspace{1cm}}$$



- Softplus: $\rho : \mathbb{R} \rightarrow [0, \infty) \quad \rho(x) = \ln(1 + e^x)$

$$\frac{\partial}{\partial x} \rho(x) = \frac{e^x}{1 + e^x} = \frac{1}{1 + e^{-x}} \Rightarrow$$
$$\frac{\partial}{\partial x} \rho(x) = \sigma(x)$$



Output activation functions

- Softmax:

$$\forall x = (x_i)_{1 \leq i \leq J} \quad \rho : \mathbb{R} \rightarrow (0, 1) \quad \rho_i(x) = \frac{e^{x_i}}{\sum_{j=1}^J e^{x_j}}$$

Derivative: $\frac{\partial \rho_i(x)}{\partial x_j} = \rho_i(x)(\delta_{i,j} - \rho_j(x)),$

where $\delta_{i,j}$ is the Kronecker delta.

Matlab

?
(.)

- Maxout:

$$\forall x = (x_i)_{1 \leq i \leq J} \quad \rho : \mathbb{R} \rightarrow \mathbb{R} \quad \rho_i(x) = \max_i x_i$$

Derivative: $\frac{\partial \rho_i(x)}{\partial x_j} = \begin{cases} 1 & \text{if } j = \operatorname{argmax}_i (x_i) \\ 0 & \text{if } j \neq \operatorname{argmax}_i (x_i) \end{cases}$

The output activation is task-dependent.

Regression

- $\rho \rightarrow$ identity
- ℓ_2 loss

ℓ_1

Binary classification

- $\rho \rightarrow$ sigmoid
- BCE loss

softmax

Multi classification

- $\rho \rightarrow$ softmax
- CE loss

Universal Approximators

Theorem : Any well behaved function can be arbitrarily approximated with a single layer FNN.

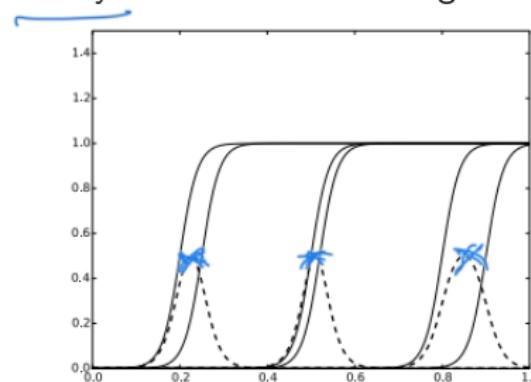
Proof: A visual proof can be found in [here](#). The original results were presented in (Cybenko, 1989).

Some intuition about this:

- Each layer uses a linear transform : $y = w^T x + b$
- We can consider the sigmoid transfer function $\sigma = \rho(y) = \frac{1}{1+e^{-y}}$
- Combine multiple σ -layer activations to build gaussian kernels.

$$\underbrace{\begin{bmatrix} x \end{bmatrix}}_{\text{input}} \cdot \underbrace{\begin{bmatrix} w^T \\ \text{opt} \end{bmatrix}}_{\text{weights}} + \underbrace{\begin{bmatrix} b \\ \text{opt} \end{bmatrix}}_{\text{bias}} = \underbrace{\begin{bmatrix} y \end{bmatrix}}_{\text{output}}$$

RELU



- subtract and weight consecutive σ - layers to obtain RBF kernels, which hold the universal approximation theorem.

Why do we need deep networks?

- It is true that FFNs are universal approximators, but the hidden layer can be arbitrarily large.
- Having a large number of layers → explainable networks & helps us to build high level features. Example for an image:
 - * first layer: compute contours, texture, rough features ...
 - * second layer: learn corners, curves, crosses
 - * next layers: build up more complex features.
- Using a shallow network → we are not able to compose low features into higher ones, we need to learn all the possibilities at once.

204

Intro
○○○○

Transfer operators
○○○○○○○○○○○○

Differential programming
●○○○○○○○○○○○○

Gradient descent algorithms
○○○○○○○○○○○○○○○○

Differential programming

Error backpropagation

- Training the network is an iterative algorithm, performed by gradient descent (or its variants).
 - The algorithm was first introduced by ([Werbos, 1981](#)) and then popularized by ([Rumelhart and co., 1986](#)).

Training mechanism

- ① initialize network parameters (weights and biases - η) at $t = 0$
 - ② at every iteration t compute: $w_t \leftarrow w_{t-1} - \gamma \nabla_\eta \mathcal{L}$

• Usually a mini-batch version of gradient descent is used (see previous Lecture for details).

- The main question is how do we compute $\frac{\partial \mathcal{L}}{\partial w}$? 

$$T_m^{(x)} = g(\omega_m^T x + b)$$

Computational graph

Computational graph

→ directed acyclic graph where the nodes may be:

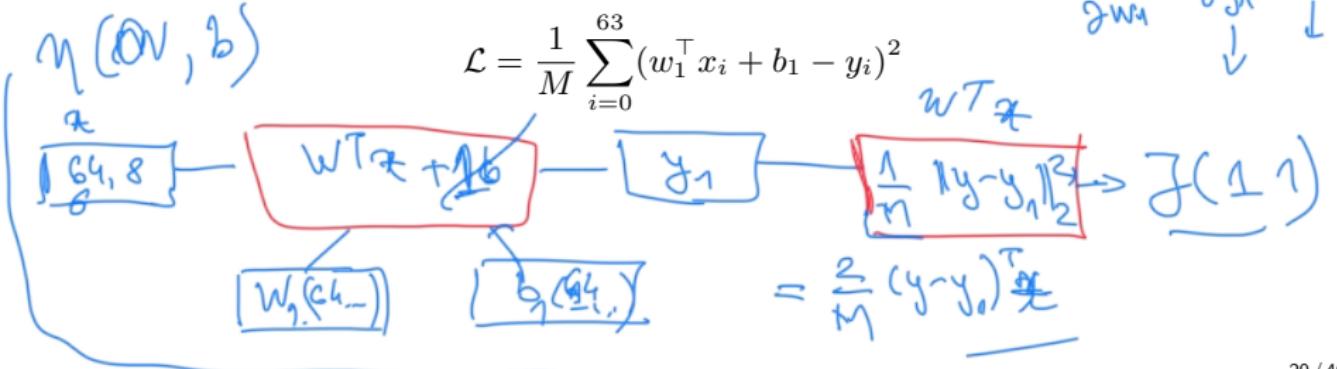
- **variables** : weights, targets, inputs, outputs,

- **operators** : Softmax, $w^\top x + b$, ReLU, ℓ_2 normalization, ...

Let us take an example for a graph associated to linear regression in $\mathbb{R}^8 \mapsto \mathbb{R}$, considering a minibatch with $M = 64$ samples (X, y) .

In this case, the loss function is defined as follows:

$$\mathcal{L} = \frac{1}{M} \sum_{i=0}^{63} (w_1^\top x_i + b_1 - y_i)^2$$



Computational graph

A

We denote with η all the variables to be updated in the neural network during training (e.g. weights and biases).

Problem : compute the partial derivatives w.r.t. to η (i.e. $\frac{\partial \mathcal{L}}{\partial \eta}$).

$$\frac{\partial \mathcal{L}}{\partial x}$$

Solution : compute the local derivatives w.r.t. inputs and apply chain rule.

We assume Jacobian (numerator) layout, e.g. $\frac{\partial \mathcal{L}}{\partial w_1} \in \mathcal{M}_{(1,8)}(\mathbb{R})$.

Otherwise, we transpose and reverse the jacobian product order. This is called the denominator (hessian) layout.

$$[f(g(x))]' = f'(g(x)) \cdot g'(x)$$

$$\frac{\partial}{\partial x} \sin(3x) = 3 \cos 3x,$$

The chain rule

General rule : The derivative of a scalar w.r.t. a vector is a **row vector**:

$$y \in \mathbb{R}, \quad x \in \mathbb{R}^n, \quad \frac{dy}{dx} \in \mathcal{M}_{1,n}(\mathbb{R})$$

The derivative of a vector function $y : \mathbb{R}^{n_x} \mapsto \mathbb{R}^{n_y}$ w.r.t. input (**Jacobian**) is an $n_y \times n_x$ matrix:

J ↘ $x \in \mathbb{R}^{n_x}, \quad y(x) \in \mathbb{R}^{n_y}, \quad \frac{dy}{dx}(x) =$

$$\begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \frac{\partial y_1}{\partial x_2} & \cdots & \frac{\partial y_1}{\partial x_{n_x}} \\ \frac{\partial y_2}{\partial x_1} & \frac{\partial y_2}{\partial x_2} & \cdots & \frac{\partial y_2}{\partial x_{n_x}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial y_{n_y}}{\partial x_1} & \frac{\partial y_{n_y}}{\partial x_2} & \cdots & \frac{\partial y_{n_y}}{\partial x_{n_x}} \end{bmatrix} (x)$$

↑ *ny*
↓ *nx*

The Chain Rule

- If we only got a single path chain of vector functions

$y_1 \in \mathbb{R}^{n_1}, y_2 \in \mathbb{R}^{n_2}, \dots, y_n \in \mathbb{R}^{n_y}$, then

$$y_1 \rightarrow y_2 = \rho_1(y_1) \rightarrow y_3 = \rho_2(y_2) \cdots y_n = \rho_{n-1}(y_{n-1}),$$

and

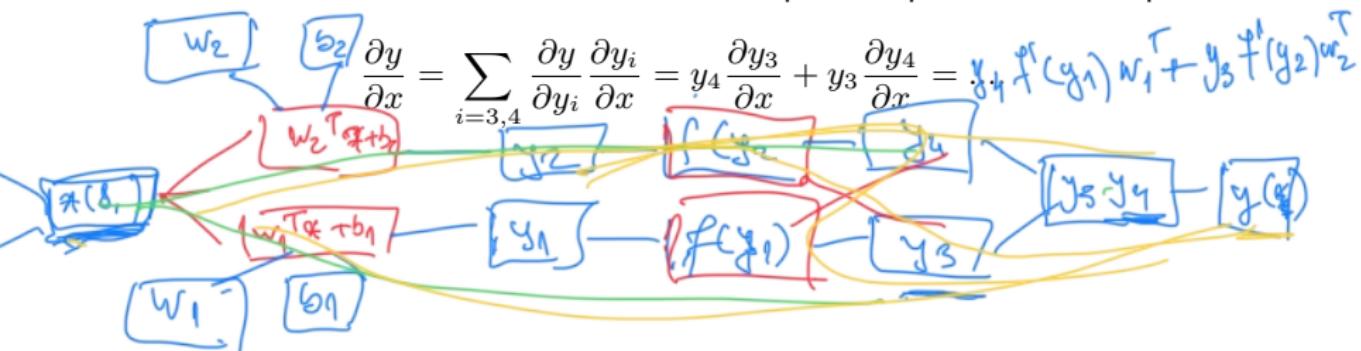
$$\frac{\partial y_n}{\partial y_1} = \frac{\partial y_n}{\partial y_{n-1}} \frac{\partial y_{n-1}}{\partial y_{n-2}} \cdots \frac{\partial y_2}{\partial y_1}.$$

Let us compute for our example: $\frac{\partial \mathcal{L}}{\partial w_1} = \dots$

In the case of matrix-like variables, things are a little more complex. We highly suggest you follow [Jeremy's course](#) for more details about multivariate calculus. Another useful insight about Tensor derivatives can be found [here](#).

Multiple paths

To obtain the final result, we sum over all the possible paths. For example :



For a reasonably large network, this is very computationally expensive:

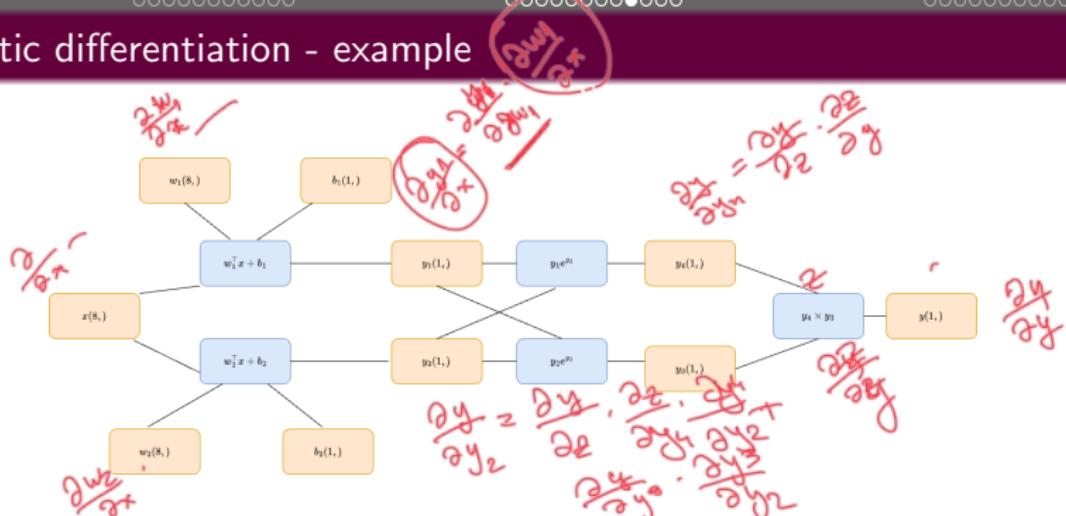
- The total number of paths : m layers and N neurons $\rightarrow N^m$ possible paths.
- This process must be repeated for each variable w.r.t. we have to differentiate
- some computations can be factored

We need to find a way to make it **more efficient!!**

Automatic differentiation

- Is a computational technique used to calculate the derivative of a function w.r.t its input variables.
- Unlike traditional methods such as symbolic differentiation or numerical differentiation, automatic differentiation leverages the chain rule of calculus to recursively break down a function into elementary operations and their derivatives.
- By automatically propagating these derivatives through the computation graph, automatic differentiation can provide accurate and efficient gradients for complex functions.
- It has two modes: forward and reverse differentiation.
- Using it is particularly useful for training neural networks and optimizing machine learning algorithms.
- Check out [Margossian's paper](#) for more details about this domain that has revolutionized the field of computational mathematics.

Automatic differentiation - example



- Forward mode differentiation : compute $\frac{\partial y}{\partial x}$ forward propagate $\frac{\partial}{\partial x}$

$$\boxed{\frac{\partial y}{\partial x}} = y_3 e^{y_1} [w_2^\top + y_2 w_1^\top] + y_4 e^{y_2} [w_1^\top + y_1 w_2^\top]$$

- Reverse mode differentiation : compute $\frac{\partial y}{\partial x}$ by propagating backwards $\frac{\partial y}{\partial}$

$$\boxed{\frac{\partial y}{\partial x}} = (y_4 y_1 e^{y_2} + y_3 e^{y_1}) w_2^\top + (y_3 y_2 e^{y_1} + y_4 e^{y_2}) w_1^\top$$

This is more efficient when we have more inputs than outputs!

Error backpropagation

- Let us consider an $m = 2$ layer feedforward neural network, propagating one real sample x . Let $n_1, n_2, n_x \in \mathbb{N}^+$ be the number of neurons on each layer and the input dimension, respectively.
- In this case, the loss, assuming scalar, can be written as follows:

$$\mathcal{L} = \phi \left(y_i, \begin{bmatrix} W_2(n_2 \times n_1) \end{bmatrix} \rho \left(\begin{bmatrix} W_1(n_1 \times n_x) \end{bmatrix} \begin{bmatrix} x_i \end{bmatrix} \right) \right) \in \mathbb{R},$$

where ϕ is the loss function, which is task dependent.

- For example, for a regression problem, we have $n_2 = 1$ and ϕ a squared error. For a classification problem, we can use CE loss and $n_2 > 1$ is the number of classes.
- We neglect the *bias* operator for this example.

Modelling example

Let us denote some intermediate results, as follows:

$$z_1 = W_1 x_i, \quad z_2 = W_2 \rho(z_1)$$

Consider each error: $\delta_i = \frac{\partial \mathcal{L}}{\partial z_i} \in \mathbb{R}^{n_i}$. Then:

$$\left| \begin{array}{l} \delta_2 = \frac{\partial \mathcal{L}}{\partial z_2} = \frac{\partial \phi(y_i, z_2)}{\partial x_i} \\ \delta_1 = \frac{\partial \mathcal{L}}{\partial z_1} = \frac{\partial \mathcal{L}}{\partial z_2} \frac{\partial z_2}{\partial z_1} \end{array} \right.$$

The errors are integrated through the matrix used for forward pass.

Practical example using PyTorch


```
optimizer = optim.Adam(model.parameters())
```

PyTorch

```
for e in range(epochs):
    for x,y in train_dataloader:
        optimizer.zero_grad()
        ...
        loss.backward() → compute gradients
        optimizer.step() → update weights
```

Chain rule
update
fonction /

Intro
○○○○

Transfer operators
○○○○○○○○○○○○

Differential programming
○○○○○○○○○○○○

Gradient descent algorithms
●○○○○○○○○○○○○○○○○

Gradient descent algorithms

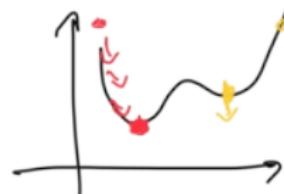
Why do we use it in deep learning

$$\begin{bmatrix} w_1 \\ x \end{bmatrix} = \begin{bmatrix} y \\ n \end{bmatrix}$$

$$\begin{bmatrix} w^2 \\ \vdots \\ y_1 \end{bmatrix} = \begin{bmatrix} y \\ \vdots \\ z \end{bmatrix}$$

- Neural networks → non-convex optimization
 - Using Gradient Descent → guarantee to get a local minimum
 - Can we do better?
Maybe yes, but for large/deep nets it has been proven (empirically) that most local minima are close (in performance) with the global minimum
 - Saddle points are critical!!

$\nabla J = 0$



Mini-batch stochastic gradient descent

Algorithm 1: Minibatch Stochastic Gradient Descent

Input: Learning rate α , minibatch size M , training dataset pairs (x_i, y_i)

Output: Updated model parameters θ

1 for every minibatch $B \subseteq D$ of size M do

2 | initialize (randomly) parameters θ_0 :

3 **for** iteration $t \equiv 0, \dots$ **do**

$$\theta_{t+1} \equiv \theta_t - \alpha \nabla_{\theta} \mathcal{L}(\theta_t) ;$$

$$\mathcal{L}(\theta) = \frac{1}{M} \sum_i \phi(\theta, x_i, y_i)$$

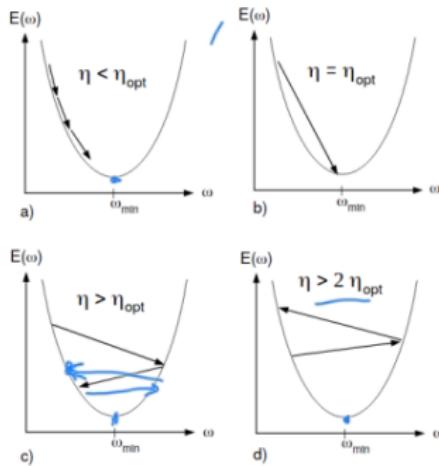
$$\nabla_{\theta} \mathcal{L}(\theta_t)^\top \nabla_{\theta} \mathcal{L}(\theta_t) \geq 0.$$

Is the size of M important?

- If $(M = 1)$ Stochastic Gradient Descent : noisy estimate, not GPU friendly
 - If $(M = N > 1)$ Batch Gradient Descent : generalization gap and local minima problem.

How about the learning rate?

The model may converge very slowly or even diverge if α is not appropriately chosen! So, what is the optimal learning rate?



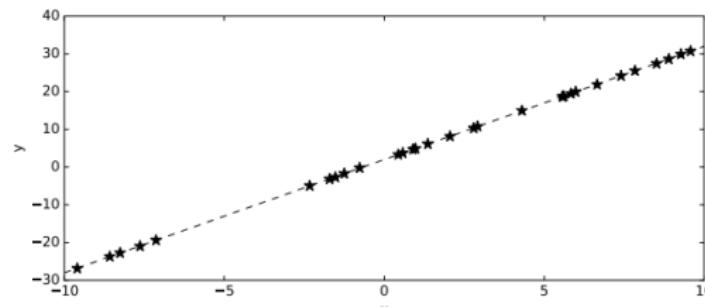
Efficient Backprop, LeCun et al., 1998

- Y.Bengio : “*The optimal learning rate is usually close to the largest learning rate that does not cause divergence of the training criterion.*”
 - Karpathy : “*0.0003 is the best learning rate for Adam, hands down.*”

Let us take a simple example

Let us consider a **regression problem** having the following setup

- $N = 30$ samples generated with : $y = 3x + 2 + \mathcal{U}(-0.1, 0.1)$
- Model: $\rho(x) = \theta^\top \begin{bmatrix} 1 \\ x \end{bmatrix}$,
- ℓ_2 loss : $\mathcal{L}(y_i, \rho(x_i)) = (y_i - \rho(x_i))^2$

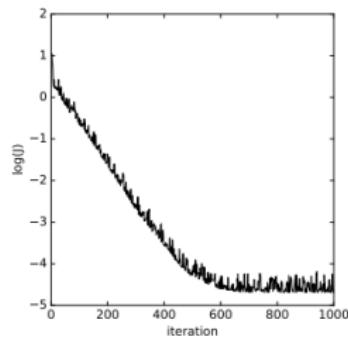
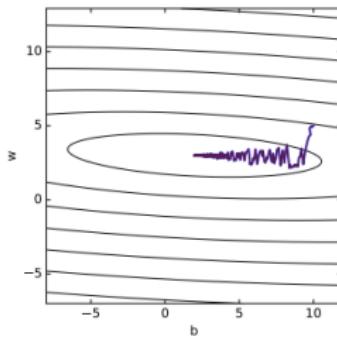


SGD minimization

Consider:

- learning rate : $\alpha = 0.005$,
 - Initial parameters : $\theta_0 = \begin{bmatrix} 10 \\ 5 \end{bmatrix}$
 - $t = 1000$ iterations.

Obtained solution : $\theta_{1000} = \begin{bmatrix} 1.9882 \\ 2.9975 \end{bmatrix}$



SGD with Momentum

Idea : Dampen the oscillations in the optimization path using a low pass filter on ∇_{θ} .

Algorithm 2: Minibatch Stochastic Gradient Descent with Momentum

Input: Learning rate α , momentum parameter β , minibatch size M , training dataset pairs (x_i, y_i)

Output: Updated model parameters θ

```
1 initialize parameters  $\theta_0$ ; initialize momentum vector  $v_0 = 0$ ;  
2 for every minibatch  $B \subseteq D$  of size  $M$  do  
3   for every iteration  $t = 0 \dots$  do  
4      $v_{t+1} = \beta v_t - \alpha \nabla_{\theta} J(\theta_t)$ ;  
5      $\theta_{t+1} = \theta_t + v_{t+1}$ ;
```

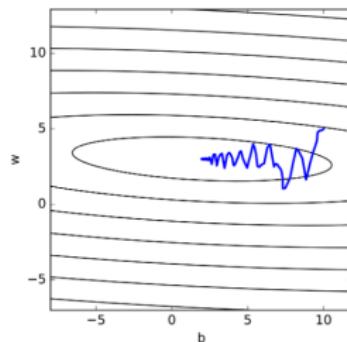
Here-above, $\beta \in \mathbb{R}$ is a hyper-parameter, usually ≈ 0.9 .

- The purpose is to accelerate the learning rate in constant directions and low curvature.
- Check [this link](#) for more details about this.

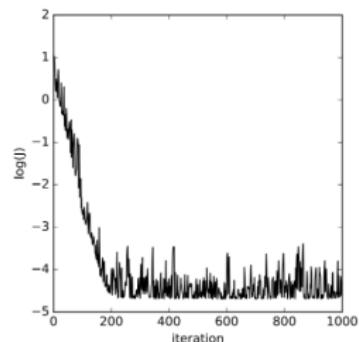
Back to our example

Parameters: $\alpha = 0.005$, $\beta = 0.6$, $\theta_0 = \begin{bmatrix} 10 \\ 5 \end{bmatrix}$

Solution : $\theta_{1000} = \begin{bmatrix} 1.9837 \\ 2.9933 \end{bmatrix}$



The optimization path



The value of the function

Gradient descent with Nesterov momentum

Idea : Based on Nesterov Accelerated Gradient. Look ahead to correct the update.

Algorithm 3: Minibatch Stochastic Gradient Descent with Nesterov Momentum

Input: Learning rate α , momentum parameter β , minibatch size M , training dataset pairs (x_i, y_i)

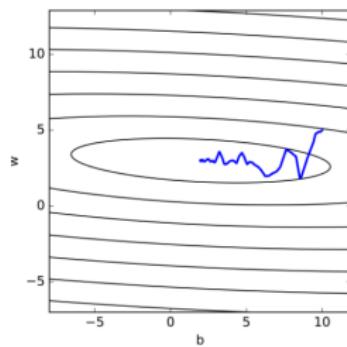
Output: Updated model parameters θ

- 1 initialize parameters θ_0 ; initialize momentum vector $v_0 = 0$;
 - 2 **for** every minibatch $B \subseteq D$ of size M **do**
 - 3 **for** every iteration $t = 0 \dots$ **do**
 - 4 $v_{t+1} = \beta v_t - \alpha \nabla_{\theta} J(\theta_t + \beta v_t)$; (Nesterov lookahead update)
 - $\theta_{t+1} = \theta_t + v_{t+1}$;
-

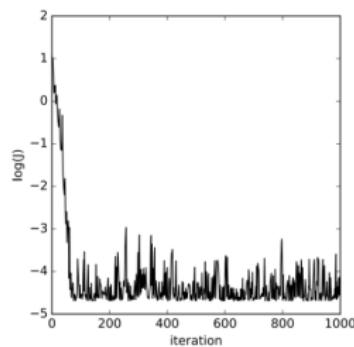
Back to our example

Parameters: $\alpha = 0.005$, $\beta = 0.8$, $\theta_0 = \begin{bmatrix} 10 \\ 5 \end{bmatrix}$

Solution : $\theta_{1000} = \begin{bmatrix} 1.9738 \\ 2.9914 \end{bmatrix}$

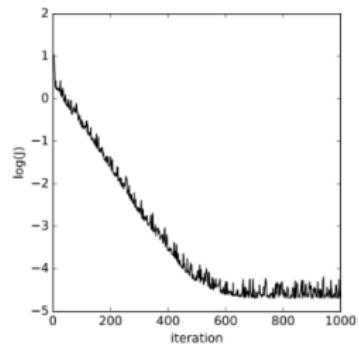


The optimization path

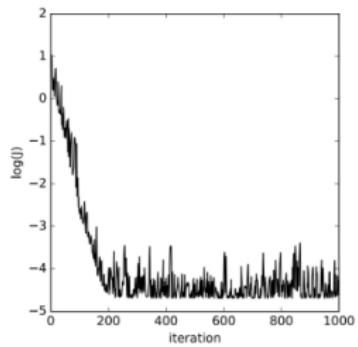


The value of the function

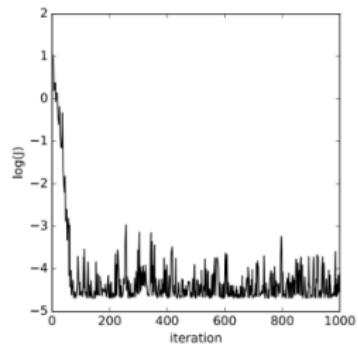
Comparison



SGD



Momentum



Nesterov

Adaptive Learning rate

Idea : Modify the learning rate during the training phase

Decreasing the learning rate

- Linear decrease from α_0 to α_f
- Halve the learning rate when the validation error stops improving
- Halve the learning rate on a fixed schedule

More recent approaches are not advocating for decreasing the learning rate

"Robbins Monro conditions"

- The 1-cycle policy
- Stochastic Gradient Descent with Warm Restart

Adaptive first order : ADAGRAD

Adagrad - Adaptive gradient proposed in 2011

- Accumulate the square of the gradient

$$r_{t+1} = r_t + \nabla_{\theta} J(\theta_t) \odot \nabla_{\theta} J(\theta_t)$$

- Scale individually the learning rates

$$\theta_{t+1} = \theta_t - \frac{\alpha}{\delta + \sqrt{r_{t+1}}} \odot \nabla_{\theta} J(\theta_t),$$

where $\delta \approx 1e^{-8}$

- Small gradients → bigger learning rate for moving fast along flat directions
- High gradients → smaller learning rate to calm down on high curvature

RMSProp

Idea : using an exponential moving average when accumulating the gradient.

- Accumulate the square of the gradient:

$$r_{t+1} = \rho r_t + (1 - \rho) \nabla_{\theta} J(\theta_t) \odot \nabla_{\theta} J(\theta_t)$$

- Scale each learning rate accordingly:

$$\theta_{t+1} = \theta_t - \frac{\alpha}{\delta + \sqrt{r_{t+1}}} \odot \nabla_{\theta} J(\theta_t),$$

where ρ is a training parameter, $\rho \approx 0.9$

ADAM algorithm

Idea : Use Adaptive Moments of past gradients

- Store running averages of past gradients:

$$m_{t+1} = \beta_1 m_t + (1 - \beta_1) \nabla_{\theta} J(\theta_t)$$

$$v_{t+1} = \beta_2 v_t + (1 - \beta_2) \nabla_{\theta} J(\theta_t) \odot \nabla_{\theta} J(\theta_t),$$

$m(t)$ and $v(t)$ are the first and second (uncentred) moments of $\nabla_{\theta} J$.
Correct the bias \hat{m}_t and \hat{v}_t :

$$\theta_{t+1} = \theta_t - \frac{\alpha}{\delta + \sqrt{\hat{v}_{t+1}}} \hat{m}_{t+1}$$

Some references

- Overview of gradient descent
- Recent gradient descent algorithms

First order methods summary

Goodfellow, Bengio, & Courville – “ There is currently no consensus [...] no single best algorithm has emerged [...] ”

- The most popular and actively in use include SGD, SGD with momentum, RMSprop, RMSprop with momentum, Adadelta and Adam
- It is task-specific mostly.

Second order methods

Idea : use the second derivative, and Taylor series expansion

$$J(\theta_t) \approx J(\theta_0) + (\theta - \theta_0)^\top \nabla_{\theta} J(\theta_0) + \frac{1}{2} (\theta - \theta_0)^\top \nabla_{\theta}^2 J(\theta_0) (\theta - \theta_0),$$

where $H = \nabla^2 J$ - Hessian matrix

Methods

- Conjugate gradient : using line search along $\nabla_{\theta} J(\theta_k)$
- Newton : just to find critical points
- Quasi Newton : BFGS (approximating H^{-1}), L-BFGS, and saddle-free versions