



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 Dense Neural networks
- 3 Differential programming
- 4 Gradient descent algorithms
- 5 Initialization mechanisms

Intro

Radial Basis Function Nets

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

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

$$\phi(x) = \begin{pmatrix} 1 \\ \exp \frac{-||x - \mu_0||^2}{2\sigma_0^2} \\ \vdots \\ \exp \frac{-||x - \mu_{N-1}||^2}{2\sigma_{N-1}^2} \end{pmatrix}$$

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

Learning mechanism

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

2-steps learning

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

3-steps learning

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

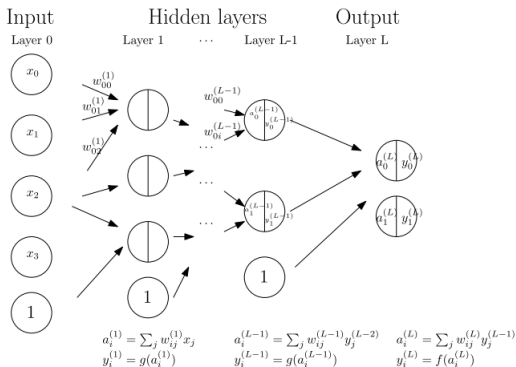
Universal approximator theorem

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

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

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 $p \in [1, \infty)$.

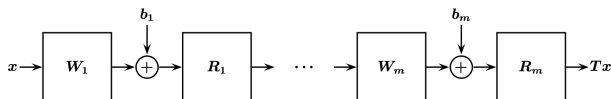
Feedforward Neural Network Architecture



Notations

- **Width** : number of neurons per layer
- **Depth** : number of weight layer
- **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 .

Feed-forwards neural networks



An m layers FCN network (T) can be viewed as a composition of elementary functions modelling layers:

$$T = T_m \circ \dots \circ T_1.$$

$$(\forall i \in \{1, \dots, m\}) \quad T_i(x_i) = R_i(W_i^\top x_i + b_i)$$

$x_i \in \mathbb{R}^{N_{i-1}}$ – input of the layer
matrix,

$b_i \in \mathbb{R}^{N_i}$ – bias parameter,
operator.

$W_i \in \mathbb{R}^{N_i \times N_{i-1}}$ – weight

$R_i: \mathbb{R}^{N_i} \rightarrow \mathbb{R}^{N_i}$ – activation

Dense Neural networks

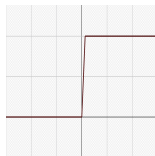
Classic Activation Operators

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

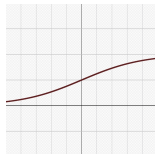


- Binary step:

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



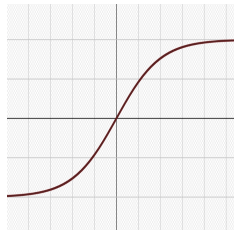
- Sigmoid: $\rho : \mathbb{R} \rightarrow (0, 1) \quad \rho(x) = \frac{1}{1+e^{-x}}$



Classic Activation Operators

- Hyperbolic tangent:

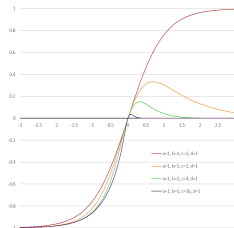
$$\rho : \mathbb{R} \rightarrow (-1, 1) \quad \rho(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$



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



More favorable for the gradient flow, but may suffer from [dying ReLU](#) problem.

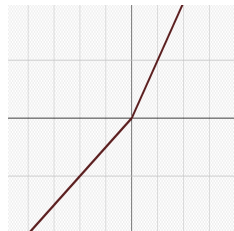
Other variants have been proposed to mitigate this problem, see next slide.

ReLU variants

- Leaky Rectified Linear Unit:

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

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

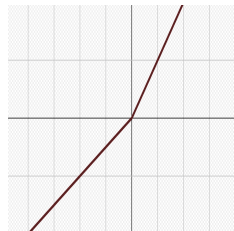


- Parametric Rectified Linear Unit:

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

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

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



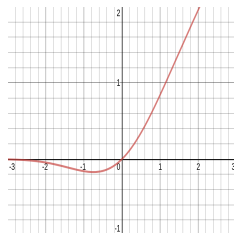
Other ReLU variants

- 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 normal distribution.

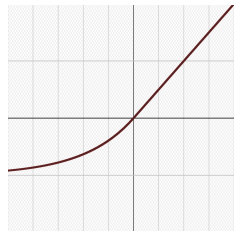


- Exponential Linear Unit (ELU):

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

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

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



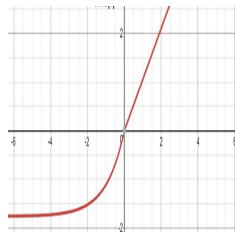
More ReLU activation functions

- Scaled Exponential Linear Unit (SELU):

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

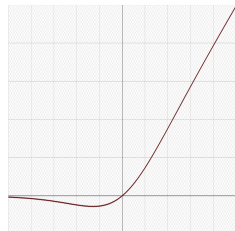
$$\rho(x) = \lambda \begin{cases} \alpha(e^x - 1) & \text{if } x \leq 0 \\ x & \text{if } x > 0 \end{cases}$$

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



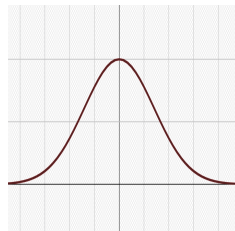
- Sigmoid Linear Unit (SiLU): $\rho : \mathbb{R} \rightarrow [-0.278, \infty)$

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

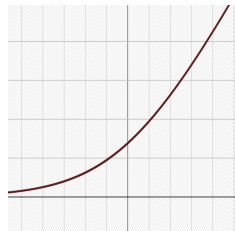


Other activation functions

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



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



Output activation functions

- **Softmax:**

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

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

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

- **Maxout:**

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

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

The output activation is task-dependent.

Regression

- $\rho \rightarrow \text{identity}$
- ℓ_2 loss

Binary classification

- $\rho \rightarrow \text{sigmoid}$
- BCE loss

Multi classification

- $\rho \rightarrow \text{softmax}$
- CE loss

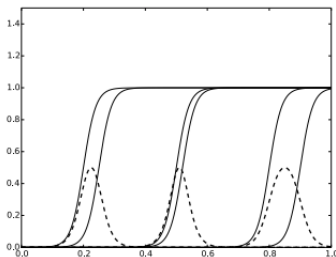
Universal Approximators

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



- 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.

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 $\frac{\partial \mathcal{L}}{\partial w}$?

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

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. $\frac{\partial \mathcal{L}}{\partial \eta}$).

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.

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:

$$x \in \mathbb{R}^{n_x}, \quad y(x) \in \mathbb{R}^{n_y}, \quad \frac{dy}{dx}(x) = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \frac{\partial y_1}{\partial x_2} & \cdots & \frac{\partial y_1}{\partial x_{n_x}} \\ \frac{\partial y_2}{\partial x_1} & \frac{\partial y_2}{\partial x_2} & \cdots & \frac{\partial y_2}{\partial x_{n_x}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial y_{n_y}}{\partial x_1} & \frac{\partial y_{n_y}}{\partial x_2} & \cdots & \frac{\partial y_{n_y}}{\partial x_{n_x}} \end{bmatrix} (x)$$

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 :

$$\frac{\partial y}{\partial x} = \sum_{i=3,4} \frac{\partial y}{\partial y_i} \frac{\partial y_i}{\partial x} = y_4 \frac{\partial y_3}{\partial x} + y_3 \frac{\partial y_4}{\partial x} = \dots$$

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

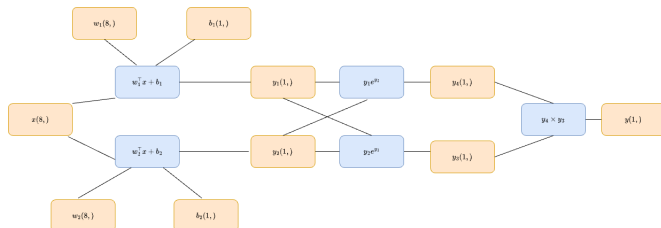
- The total number of paths : m layers and N neurons $\rightarrow N^m$ possible paths.
- This process must be repeated for each variable w.r.t. we have to differentiate
- some computations can be factored

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 $\frac{\partial y}{\partial x}$

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

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

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

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:

$$\delta_2 = \frac{\partial \mathcal{L}}{\partial z_2} = \frac{\partial \phi(y_i, z_2)}{\partial x_i}$$

$$\delta_1 = \frac{\partial \mathcal{L}}{\partial z_1} = \frac{\partial \mathcal{L}}{\partial z_2} \frac{\partial z_2}{\partial z_1}$$

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

Practical example using PyTorch

- Most deep learning frameworks compute all the derivatives automatically with autograd.
- The computational graphs can be build static or dynamically (eager mode)
- The learning process has 2 phases: forward pass (eval. of outputs) and reverse-mode differentiation (backward pass)
- The reverse-mode differentiation uses the variables computed in the forward pass → learning through SGD!

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

for e in range(epochs):
    for X,y in train_dataloader:
        optimizer.zero_grad()
        ...
        loss.backward()
        optimizer.step()
```

Gradient descent algorithms

Why do we use it in deep learning

- Neural networks → **non-convex optimization**
- Using Gradient Descent → guarantee to get a **local minima**
- *Can we do better?*

Maybe yes, but for large/deep nets it has been proven (empirically though) that most local minima are close (in performance) with the global one.
- Saddle points are critical!!

Mini-batch gradient descent

Algorithm 1: Minibatch 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  $\theta_0$ ;
3   for iteration  $t = 0 \dots$  do
4      $\theta_{t+1} = \theta_t - \alpha \nabla_{\theta} \mathcal{L}(\theta_t)$  ;
5      $\mathcal{L}(\theta) = \frac{1}{M} \sum_i \phi(\theta, x_i, y_i)$ 

```

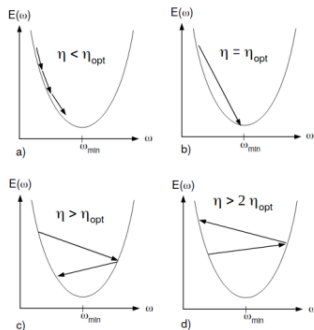
Taylor series expansion : $\mathcal{L}(\theta_{t+1}) \approx \mathcal{L}(\theta_t) + (\theta_{t+1} - \theta_t)^{\top} \nabla_{\theta} \mathcal{L}(\theta_t)$

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?



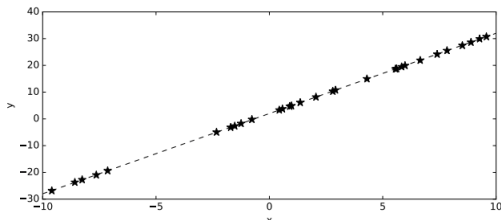
Efficient Backprop, LeCun et al., 1998

- 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 : $\mathcal{L}(y_i, \rho(x_i)) = (y_i - \rho(x_i))^2$

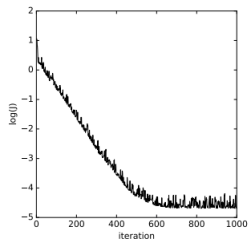
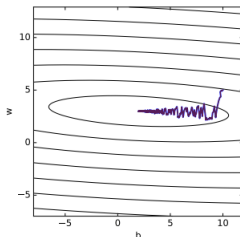


SGD minimization

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      $\theta_{t+1} = \theta_t + v_{t+1}$ ;

```

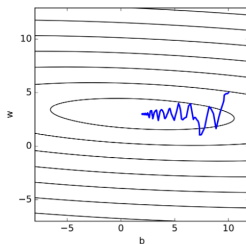
Here-above, $\beta \in \mathbb{R}$ is a hyper-parameter, usually ≈ 0.9 .

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

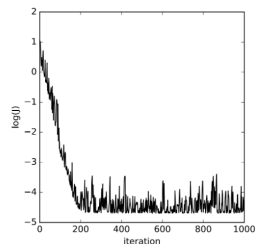
Back to our example

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

```

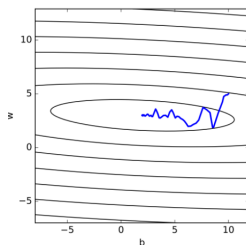
1 initialize parameters  $\theta_0$ ; initialize momentum vector  $v_0 = 0$ ;
2 for every minibatch  $B \subseteq D$  of size  $M$  do
3   for every iteration  $t = 0 \dots$  do
4      $v_{t+1} = \beta v_t - \alpha \nabla_{\theta} J(\theta_t + \beta v_t);$     (Nesterov lookahead update)
      $\theta_{t+1}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

```

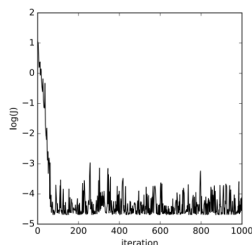
Back to our example

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

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

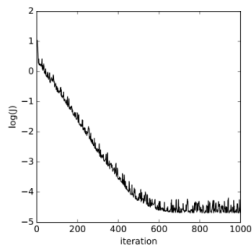


The optimization path

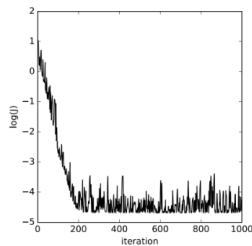


The value of the function

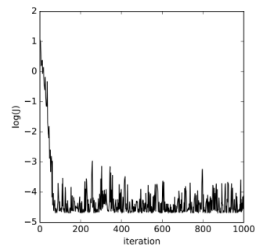
Comparison



SGD



Momentum



Nesterov

Adaptive Learning rate

Idea : Modify the learning rate during the training phase

Decreasing the learning rate – Robbins Monro conditions

- 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

Increasing learning rate

- 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 \rightarrow bigger learning rate for moving fast along flat directions
- High gradients \rightarrow 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, Adadelata 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

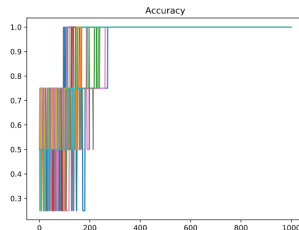
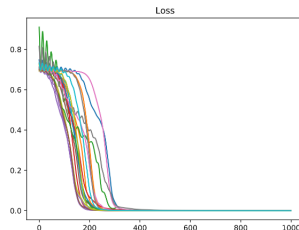
Initialization mechanisms

Does the starting point matter?

Problem : XOR classification

Model architecture :

- Layers: 3 Dense (2-4-1); Activation: Sigmoid
- Init: $\mathcal{U}(-10, 10)$, bias = 0
- Loss: BCE
- SGD ($\alpha = 0.1$, momentum = 0.99)
- $n = 1000$ epochs



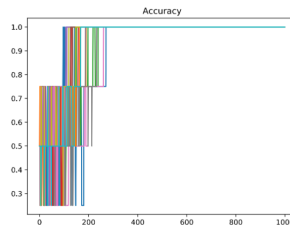
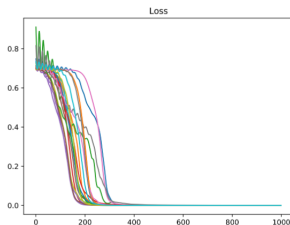
It fails 6/20 times.

Does the starting point matter?

Problem : XOR classification

Model architecture :

- Layers: 3 Dense (2-4-1); Activation: Sigmoid
- Init: $\mathcal{N}(0, \frac{1}{\sqrt{fan_{in}}})$, bias = 0
- Loss: BCE
- SGD ($\alpha = 0.1$, momentum = 0.99)
- $n = 1000$ epochs



It fails 0/20 times.

Standardization

- The starting point of a gradient descent has a dramatic impact.
- All gradient-based algorithms converge faster if the data are **normalized** and **decorrelated**.
- Several strategies are currently used, depending on the task.

Let x be your input data.

Min-Max scaling

$$x' = \frac{x - \min(x)}{\max(x) - \min(x)}$$

Z-score scaling

$$x' = \frac{x - \mu_x}{\sigma_x + \epsilon},$$

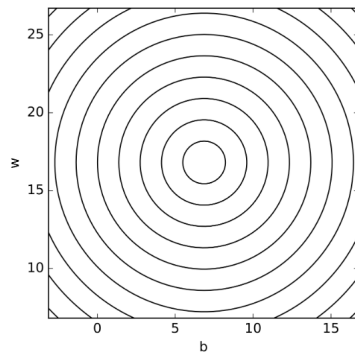
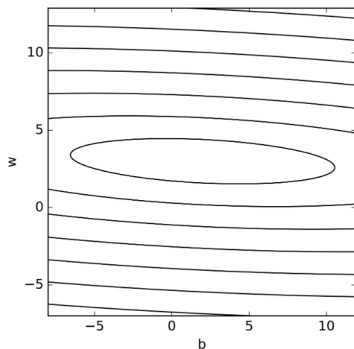
where μ_x, σ_x and ϵ are the mean, the standard deviation and a small constant, respectively.

ZCA whitening

$$X' = WX, W = \frac{1}{\sqrt{n-1}}(X'X^\top)^{(-1/2)}$$

The effect of standardization

Problem: $y = 3x + 2 + \mathcal{U}(-0.1, 0.1)$



How to chose a good initialization

- An effective initialization strategy should disrupt symmetry within the neural network; constant initialization methods tend to lead units to learn identical patterns.
- A favorable initialization should commence optimization from a region of limited capacity, e.g. linear neural networks.
- An ideal initialization scheme should maintain the distribution of both activations and gradients, guarding against issues such as exploding or vanishing gradients.

Exploding and vanishing gradients

- **The Fundamental Deep Learning Problem** – the gradient can either vanish or explode, especially in deep networks.
- The deeper the network ($m \rightarrow \text{large}$), the more chances that through backprop the gradients either explode or vanish.
- Problems with activations such as *sigmoid*, *ReLU*.

Preventing vanishing/ exploding gradients

- ensure a good flow of gradient – use appropriate activation functions (e.g. ReLu, PreLu, etc..) or some architectural elements (e.g. skip-connections, constant error loops (LSTMs))
- use *clipping* methods → only for exploding gradients problem

Initialization of Y. LeCun

First introduced in [Gradient-Based Learning Applied to Document Recognition](#)

Goal : Initialize parameters such that the activation operator ρ stays in its linear part by using more layers:

- use symmetric transfer operations : R_i is symmetric, which means there exist $(c_i, d_i) \in (\mathbb{R}^{N_i})^2$ such that

$$(\forall x \in \mathbb{R}^{N_i}) \quad R_i(x) - d_i = -R_i(-x + c_i).$$

e.g. $R_i = 1.7159 \tanh(\frac{2}{3}x)$

- initialize the biases with 0 and the weights drawn randomly from $\mathcal{N}(\mu = 0, \sigma^2 = \frac{1}{fan_{N_i}})$

If $x \in \mathbb{R}^n$ is $\mathcal{N}(0, \Sigma = I_d)$ and $w \in \mathbb{R}^n$ is $\mathcal{N}(0, \Sigma = \frac{1}{n}I_d)$. Assume x_i, w_i pair wise independent, then:

$$E[w^\top x + b] = E[w^\top x] = \sum_i E[w_i x_i] \sum_i E[w_i] E[x_i] = 0$$

$$\begin{aligned} \text{var}[w^\top x + b] &= \text{var}[w^\top x] \\ &= \sum_i \sigma_{w_i}^2 \sigma_{x_i}^2 + \sigma_{w_i}^2 \mu_{x_i}^2 + \mu_{w_i}^2 \sigma_{x_i}^2 \\ &= \sum_i \sigma_{w_i}^2 \sigma_{x_i}^2 = \frac{1}{n} \sum_i \sigma_{x_i}^2 = 1 \end{aligned}$$

Glorot Initialization

Goal : preserve the same distribution along the forward and backward pass, which prevents saturating transfer functions.

$$T = R_m(W_m \dots R_1(W_1(W_0x + b_0) + b_1) \dots + b_m)$$

Glorot (Xavier) initialization

- Input data should be centered, normalized and decorrelated
- Symmetric activation functions should be used, with $R'_i(0) = 1$

Assuming the linear regime $R'_i() = 1$ for the network:

- Forward prop. var. constraint : $\forall i, fan_{in_i} \sigma_{W_i}^2 = 1$
- Backward prop. var. constraint : $\forall i, fan_{out_i} \sigma_{W_i}^2 = 1$

Compromise: $\forall i, \frac{1}{\sigma_{W_i}^2} = \frac{fan_{in} + fan_{out}}{2}$

- Glorot Uniform: $\mathcal{U}\left(-\frac{\sqrt{6}}{\sqrt{fan_{in} + fan_{out}}}, \frac{\sqrt{6}}{\sqrt{fan_{in} + fan_{out}}}\right)$ and $b = 0$
- Glorot Normal: $\mathcal{N}\left(0, \frac{\sqrt{2}}{\sqrt{fan_{in} + fan_{out}}}\right)$ and $b = 0$

He Initialization

Goal : preserve the same distribution along the forward and backward pass for rectifiers.

$$T = R_m(W_m \dots R_1(W_1(W_0x + b_0) + b_1) \dots + b_m)$$

- input dimensions should be centered, normalized, uncorrelated
- use ReLU operators: $\forall i \in \{1, \dots, m\} \quad R_i(x) = \max(0, x)$
- weights are initialized with symmetric distribution, zero-mean, independently
- biases are set to 0
- the components of x are assumed i.i.d.

Forward prop. var. constraint : $\forall i, \frac{1}{2} fan_{in_i} \sigma_{W_i}^2 = 1$

Backward prop. var. constraint : $\forall i, \frac{1}{2} fan_{out_i} \sigma_{W_i}^2 = 1$

Compromise: use one of them, e.g. $\sigma_{W_i}^2 = \frac{2}{fan_{out}}$

- He Uniform: $\mathcal{U} \left(-\frac{\sqrt{6}}{\sqrt{fan_{in}}}, \frac{\sqrt{6}}{\sqrt{fan_{in}}} \right)$ and $b = 0$
- He Normal: $\mathcal{N} \left(0, \frac{\sqrt{2}}{\sqrt{fan_{in}}} \right)$ and $b = 0$