Deep learning for agronomy

3rd year engineer

Ammouri Bilel

: ESAM
in: ammouri-bilel
: bilelammouri
: Ammouri-Bilel
: 0000-0002-5491-5172

Plan

Binary Classification

Notation

- A single training example is represented by (x, y), where $x \in \mathbb{R}^{n_x}$: feature vector and $y \in \{0, 1\}$
- m training examples: $\{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), ..., (x^{(m)}, y^{(m)})\}$
- Training data matrix:

$$X = \begin{pmatrix} \vdots & \vdots & \ddots & \vdots \\ x^{(1)} & a^{(2)} & \cdots & a^{(m)} \\ \vdots & \vdots & \ddots & \vdots \end{pmatrix}$$

- Training labels: $Y = \left[y^{(1)}, y^{(2)}, ..., y^{(m)}\right] \in \mathbb{R}^{1\times m}$

Binary Classification

- Given an image represented by a feature vector x, the algorithm will evaluate the probability of a "Bacillus bacteria"being in that image, given x want:

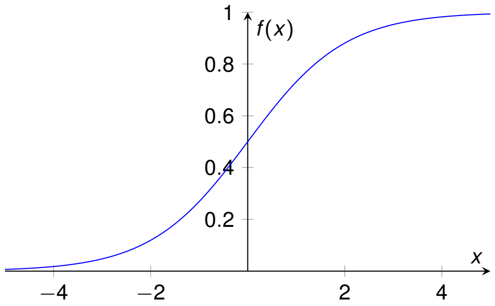
$$\hat{y} = P(y = 1|x) \tag{1}$$

- Mapping types:
  - Linear mapping

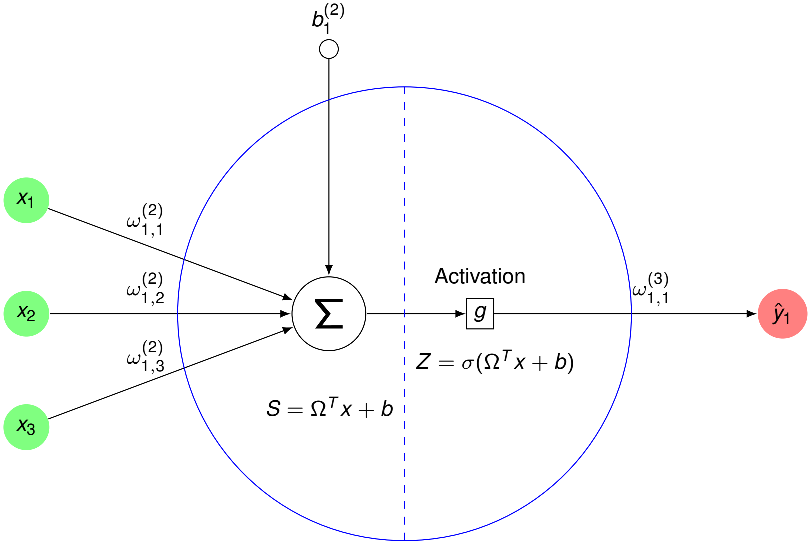
$$\hat{y} = \Omega^T x + b \tag{2}$$

  - Non-linear mapping

$$\hat{y} = \sigma(\Omega^T x + b) \tag{3}$$



Logistic Regression to Neuron



Cost Function

- To train the parameters Ω and b, we need to define a **cost function**.

$$\hat{y}^{(i)} = \sigma(\Omega^T x^{(i)} + b) \tag{4}$$

- Given $\{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), ..., (x^{(m)}, y^{(m)})\}$ we want $\hat{y}^{(i)} \approx y^{(i)}$

Loss (error) function

computes the error for a single training sample.

$$L(\hat{y}^{(i)}, y^{(i)}) = \frac{1}{2}(\hat{y}^{(i)} - y^{(i)})^2 \tag{5}$$

$$L(\hat{y}^{(i)}, y^{(i)}) = -(y^{(i)} \times \ln(\hat{y}^{(i)}) + (1 - y^{(i)}) \times \ln(1 - \hat{y}^{(i)})) \tag{6}$$

- If $y^{(i)} = 1$ then: $L(\hat{y}^{(i)}, y^{(i)}) = -\ln(\hat{y}^{(i)}) \Rightarrow$ want $\hat{y}^{(i)}$ to be large or close to 1.
- If $y^{(i)} = 0$ then: $L(\hat{y}^{(i)}, y^{(i)}) = -\ln(1 - \hat{y}^{(i)}) \Rightarrow$ want $\hat{y}^{(i)}$ to be small or close to 0.

Cost function

computes the average error over all training samples.

$$J(\omega, b) = \frac{1}{m}\sum_{i=1}^{m} L(\hat{y}^{(i)}, y^{(i)}) = -\frac{1}{m}\sum_{i=1}^{m}[y^{(i)} \times \ln(\hat{y}^{(i)}) + (1 - y^{(i)}) \times \ln(1 - \hat{y}^{(i)})] \tag{7}$$

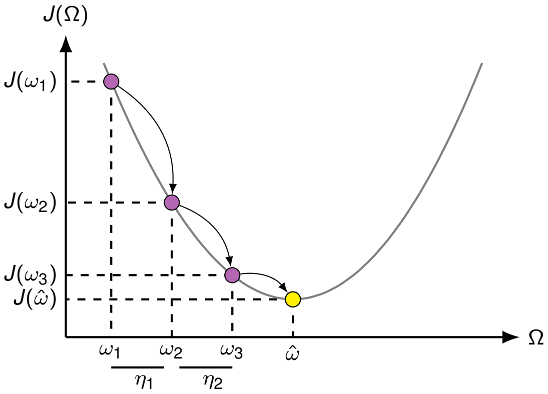
Gradient Descent

Definition

- Gradient descent serves as a widely employed optimization technique in machine learning;
- aiming to train a model by concentrating on the minimization of the cost function $J(\Omega)$ concerning the parameters Ω;
- This cost function essentially quantifies the disparity between the hypothesis $\hat{y}$ and the actual observed data y;
- Throughout the learning process, we systematically compute the gradient of the cost function and adjust the model's parameters;
- These parameter updates occur in the opposite direction of the gradient of $J(\Omega)$.

$\Rightarrow$ If the gradient of $J(\Omega)$ is positive, the parameters are reduced, and conversely, they are increased. The model continues to fine-tune these parameters until it attains the minimal possible error $J(\Omega)$.

Gradient Descent



Gradient Descent

Formulation

- The primary objective is to iteratively adjust the parameters to minimize the cost function $J(\Omega)$;
- We continue this updating process until we achieve the minimum possible error;
- Define the cost function, $J(\Omega)$ as a $\frac{1}{2}MSE$ betwenn predictive and observed data:

$$J(\omega, b) = \frac{1}{2m}\sum_{i=1}^{m}(\hat{y}^{(i)} - y^{(i)})^2 \tag{8}$$

  Where, m number of training set and $\Omega \in \mathbb{R}^{n+1}$
- If we just want to consider for only one training example $m = 1$:

$$J(\omega, b) = \frac{1}{2}(\hat{y} - y)^2 \tag{9}$$

Gradient Descent

Formulation

The Gradient

- The gradient of a function J, represented as ∇J, is simply a vector comprised of its partial derivatives.
- For a multivariable function $J\,(\omega_0, \omega_1, \omega_2, ..., \omega_n)$, the gradient takes the following form:

$$\nabla J(\Omega)^{tr} = \left[\frac{\partial J}{\partial \omega_1}, \frac{\partial J}{\partial \omega_2}, ..., \frac{\partial J}{\partial \omega_{d+1}}\right] \tag{10}$$

Parameters Update Rule

- The parameters are updated iteratively in the reverse direction of $\nabla J(\Omega)$ throughout the learning process
- We can formulate the parameter update rule as follows:

$$\Omega := \Omega - \eta \nabla J(\Omega) \tag{11}$$

Where η is the learning rate ($\eta \in]0, 1[$)

Gradient Descent

Formulation

Matrix form

$$\begin{pmatrix} \omega_1 \\ \omega_2 \\ \vdots \\ \omega_{n+1} \end{pmatrix} := \begin{pmatrix} \omega_1 \\ \omega_2 \\ \vdots \\ \omega_{n+1} \end{pmatrix} - \eta \begin{pmatrix} \frac{\partial J}{\partial \omega_1} \\ \frac{\partial J}{\partial \omega_2} \\ \vdots \\ \frac{\partial J}{\partial \omega_{n+1}} \end{pmatrix} \tag{12}$$

$$\Omega := \Omega - \frac{\eta}{m}((\hat{y} - y)^{tr} x)^{tr} \tag{13}$$

To simplify, we rewrite matrix form in the following form with a notice that all the parameters must be updated simultaneously:

$$\omega_i := \omega_i - \eta \frac{\partial J}{\partial \omega_i}, \forall i \in [1, 2, ..., n+1] \tag{14}$$

Gradient Descent

Gradient Descent for a Single Training Example

- We'll illustrate the process of computing the partial derivative of $J(\Omega)$;
- We'll begin by examining a single training example;
- Let's restate the cost function $J(\theta)$ for a single training example (x, y), where $\mathrm{x} \in \mathbb{R}^{n+1}$, $y \in \mathbb{R}$, and $\Omega \in \mathbb{R}^{n+1}$:

$$J(\Omega, b) = \frac{1}{2}(\hat{y} - y)^2 \tag{15}$$

- Substitute (15) in (14) we have:

$$\omega_i := \omega_i - \eta \frac{\partial(\frac{1}{2}(\hat{y} - y)^2)}{\partial \omega_i}, \forall i \in [1, 2, ..., n+1] \tag{16}$$

Gradient Descent

Gradient Descent for a Single Training Example

- We can solve the partial derivative as :

$$\begin{aligned}\frac{\partial(\frac{1}{2}(\hat{y}-y)^2)}{\partial\omega_i} &= (\hat{y}-y)\frac{\partial((\hat{y}-y))}{\partial\omega_i}\\ &= (\hat{y}-y)\frac{\partial((\omega_0+\omega_1 X_1+\omega_2 x_2+...+\omega_{n+1}x_{n+1}-y))}{\partial\omega_i}\\ &= (\hat{y}-y)x_i\end{aligned} \tag{17}$$

- we finally have the gradient descent formula for a single training example as follows:

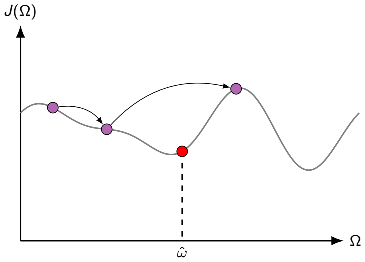
$$\omega_i := \omega_i - \eta(\hat{y}-y)x_i, \forall i \in [1, 2, ..., n+1] \tag{18}$$

Gradient Descent

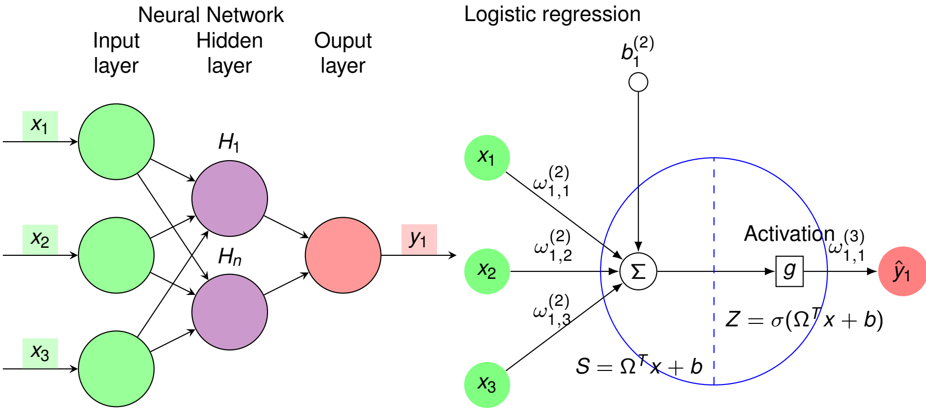
The problem of local minima

Numerical Optimization Issues

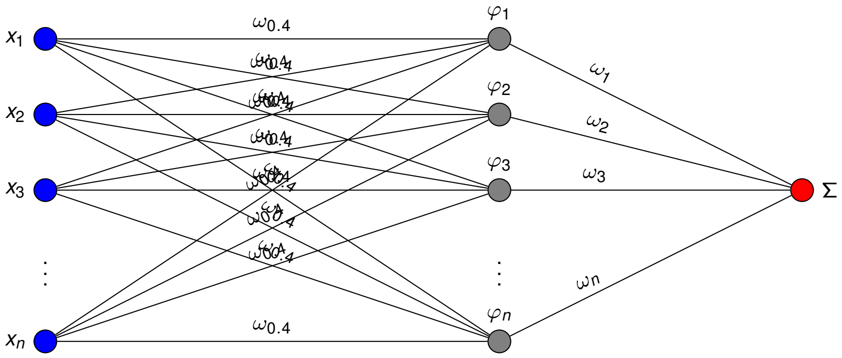
1. How to choose the learning rate η?
   - Too small $\eta \longrightarrow$ slow convergence
   - Too large $\eta \longrightarrow$ overshoot, no convergence (!)
2. Local/Global minimum? Or a saddle point?
3. When to terminate?
4. There are many forms of the gradient descent: Newton's descent, the momentum term, conjugate gradient algorithm, etc.,
5. Exploration versus exploitation



What is a Neural Network?



Neural Network Representation



Neural Network Representation

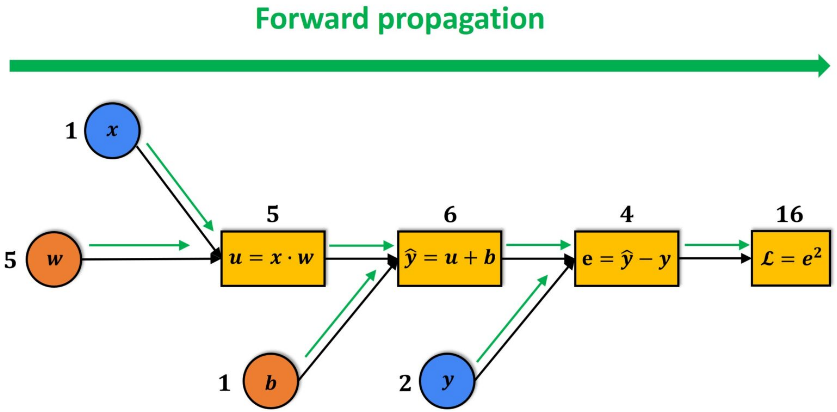
Forward Pass

Forward propagation, also known as the forward pass, is the process of computing and storing intermediate variables, including outputs, within a neural network. This calculation unfolds sequentially from the input layer to the output layer, establishing the foundation for subsequent stages in the neural network's operation.

- X = $\begin{bmatrix} \vdots & \vdots & \vdots \\ x^1 & x^2 & x^m \\ \vdots & \vdots & \vdots \end{bmatrix}$; Z = $\begin{bmatrix} \vdots & \vdots & \vdots \\ z^{[1]()} & z^{[1](2)} & z^{[1](m)} \\ \vdots & \vdots & \vdots \end{bmatrix}$; A = $\begin{bmatrix} \vdots & \vdots & \vdots \\ a^{[1]()} & a^{[1](2)} & a^{[1](m)} \\ \vdots & \vdots & \vdots \end{bmatrix}$
- $\varphi_1 = \begin{cases} Z^1 & = W^1 X + b^1 \quad \textit{hidden layer} \\ A^1 & = \phi(Z^1) \quad \textit{hidden activation vector} \end{cases}$
- Output layer variable : $\omega = W^2 A^1$;
- The loss term for a single data example : $\Sigma = l(\omega, y)$
- $J = \Sigma + \theta$; avec $\theta = \frac{\lambda}{2}(\|W^1\|^2 + \|W^2\|^2)$ regularization term

Neural Network Representation

Forward Pass



Source: `https://datahacker.rs/004-computational-graph-and-autograd-with-pytorch/`

Neural Network Representation

Backpropagation

Backpropagation is a technique employed for computing the gradients of neural network parameters. In essence, this method involves traversing the network in reverse, moving from the output layer to the input layer, utilizing the chain rule from calculus. Throughout this process, the algorithm systematically retains intermediate variables, namely partial derivatives, necessary for the comprehensive calculation of gradients concerning specific parameters.

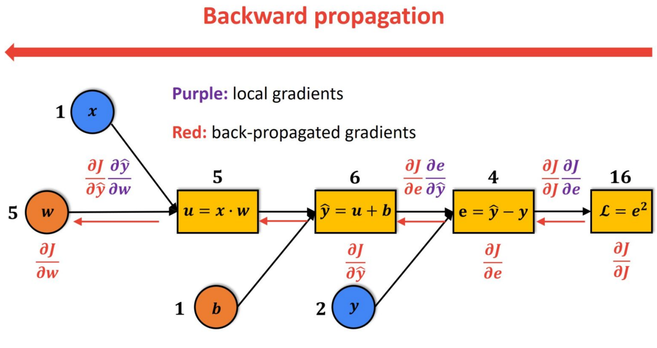
Neural Network Representation

Backpropagation

1. calculate the gradients of the objective function (loss and regularization term) : $\frac{\partial J}{\partial \Sigma} = 1$; $\frac{\partial J}{\partial s} = 1$
2. compute the gradient of the objective function (output layer) : $\frac{\partial J}{\partial \omega} = \frac{\partial J}{\partial \Sigma}\frac{\partial \Sigma}{\partial \omega} = \frac{\partial \Sigma}{\partial \omega}$
3. calculate the gradients of the regularization term : $\frac{\partial s}{\partial W^1} = \lambda W^1$ and $\frac{\partial \Sigma}{\partial W^2} = \lambda W^2$
4. calculate the gradient: $\frac{\partial J}{\partial W^2} = \frac{\partial J}{\partial \omega}\frac{\partial \omega}{\partial W^2} + \frac{\partial J}{\partial s}\frac{\partial s}{\partial W^2} = \frac{\partial J}{\partial \omega}{A^1}^T + \lambda W^2$
5. The gradient with respect to the hidden layer output: $\frac{\partial J}{\partial A^1} = \frac{\partial J}{\partial \omega}\frac{\partial \omega}{\partial A^1} = {W^2}^T\frac{\partial J}{\partial \omega}$
6. $\frac{\partial J}{\partial Z} = \frac{\partial J}{\partial A^1}\frac{\partial A^1}{\partial Z} = \frac{\partial J}{\partial A^1} \odot \phi^{'}(Z)$
7. Finally the gradient: $\frac{\partial J}{\partial W^1} = \frac{\partial J}{\partial Z}\frac{\partial Z}{\partial W^1} + \frac{\partial J}{\partial s}\frac{\partial s}{\partial W^1} = \frac{\partial J}{\partial Z}X^T + \lambda W^1$

Neural Network Representation

Backpropagation


Backward propagation
Purple: local gradients
Red: back-propagated gradients
1 x
5 w
1 b
2 y
5 $u = x \cdot w$
6 $\hat{y} = u + b$
4 $e = \hat{y} - y$
16 $\mathcal{L} = e^2$
$\frac{\partial J}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial w}$
$\frac{\partial J}{\partial e} \frac{\partial e}{\partial \hat{y}}$
$\frac{\partial J}{\partial J} \frac{\partial J}{\partial e}$
$\frac{\partial J}{\partial w}$
$\frac{\partial J}{\partial \hat{y}}$
$\frac{\partial J}{\partial e}$
$\frac{\partial J}{\partial J}$

$$\frac{\partial J}{\partial e} = \frac{\partial J}{\partial J}\frac{\partial J}{\partial e} = \frac{\partial J}{\partial J}\frac{\partial e^2}{\partial e} = 1 \cdot 2e = 8$$

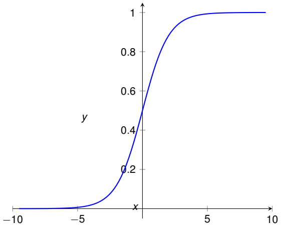
$$\frac{\partial J}{\partial \hat{y}} = \frac{\partial J}{\partial e}\frac{\partial e}{\partial \hat{y}} = \frac{\partial J}{\partial e}\frac{\partial\,(\hat{y} - y)}{\partial \hat{y}} 2e \cdot 1 = 8$$

$$\frac{\partial J}{\partial w} = \frac{\partial J}{\partial \hat{y}}\frac{\partial \hat{y}}{\partial w} = \frac{\partial J}{\partial \hat{y}}\frac{\partial xw}{\partial w} = 2e \cdot x = 8$$

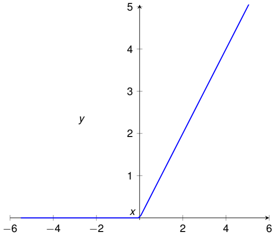
Source: `https://datahacker.rs/004-computational-graph-and-autograd-with-pytorch/`

Activation Functions

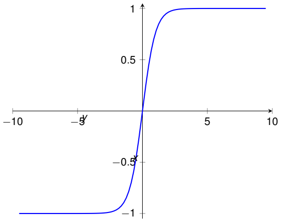
Sigmoid : $a = \frac{1}{1+e^{-z}}$



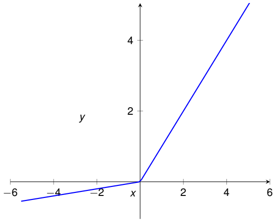
ReLU: $a = max(0, z)$



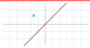
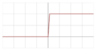
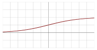
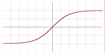
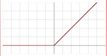
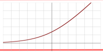
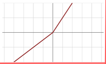
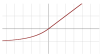
Tanh: $a = \frac{e^{z}-e^{-z}}{e^{z}+e^{-z}}$



Leaky ReLU: $a = max(0, z) + min(0, \alpha z)$



Activation Functions

ACTIVATION FUNCTION	PLOT	EQUATION	DERIVATIVE	RANGE
Linear		f(x) = x	f'(x) = 1	(-∞, ∞)
Binary Step		f(x) = $\begin{cases} 0 & \text{if } x < 0 \\ 1 & \text{if } x \geq 0 \end{cases}$	f'(x) = $\begin{cases} 0 & \text{if } x \neq 0 \\ \text{undefined} & \text{if } x = 0 \end{cases}$	{0, 1}
Sigmoid		f(x) = $\sigma(x) = \frac{1}{1+e^{-x}}$	f'(x) = $f(x)(1-f(x))$	(0, 1)
Hyperbolic Tangent(tanh)		f(x) = $\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$	f'(x) = $1 - f(x)^2$	(-1, 1)
Rectified Linear Unit(ReLU)		f(x) = $\begin{cases} 0 & \text{if x<0} \\ x & \text{if x≥0} \end{cases}$	f'(x) = $\begin{cases} 0 & \text{if } x < 0 \\ 1 & \text{if } x > 0 \\ \text{undefined} & \text{if } x = 0 \end{cases}$	[0, ∞)
Softplus		f(x) = $\ln(1+e^x)$	f'(x) = $\frac{1}{1+e^{-x}}$	(0, 1)
Leaky ReLU		f(x) = $\begin{cases} 0.01x & \text{if } x < 0 \\ x & \text{if } x \geq 0 \end{cases}$	f'(x) = $\begin{cases} 0.01 & \text{if } x < 0 \\ 1 & \text{if } x \geq 0 \end{cases}$	(-1, 1)
Exponential Linear Unit(ELU)		f(x) = $\begin{cases} \alpha(e^x - 1) & \text{if } x \leq 0 \\ x & \text{if } x > 0 \end{cases}$	f'(x)= $\begin{cases} \alpha e^x & \text{if } x < 0 \\ 1 & \text{if } x > 0 \\ 1 & \text{if } x = 0 \text{ and } \alpha = 1 \end{cases}$	[0, ∞)

Source: `https://iq.opengenus.org/linear-activation-function/`

Activation Functions

Pros and Cons of Activation Functions

- The sigmoid activation function is seldom employed, primarily reserved for the output layer in binary classification problems.
- Tanh surpasses sigmoid due to its ability to center the data, facilitating the learning process.
- Both sigmoid and tanh have a drawback: when the input z is extremely large or small, their derivatives approach zero, impeding gradient descent.
- The widely adopted alternatives are ReLU and Leaky ReLU activation functions.