Machine Learning for biology

V. Monbet

UFR de Mathématiques
Université de Rennes

Outline

Outline

Context

Data:

- "Responses":
  Regression case: $y_i \in \mathbb{R}$ for $i = 1, \cdots, n$
  Classification case: $y_i \in C$ for $i = 1, \cdots, n$ and $C = \{1, \cdots, k\}$
- "Explanatory variables" or "predictors," $x_i \in \mathbb{R}^p$ for $i = 1, \cdots, n$.
- Each pair (y_i, x_i) represents an experiment (individual).

Problem:

- Explain Y in terms of X

Principle:

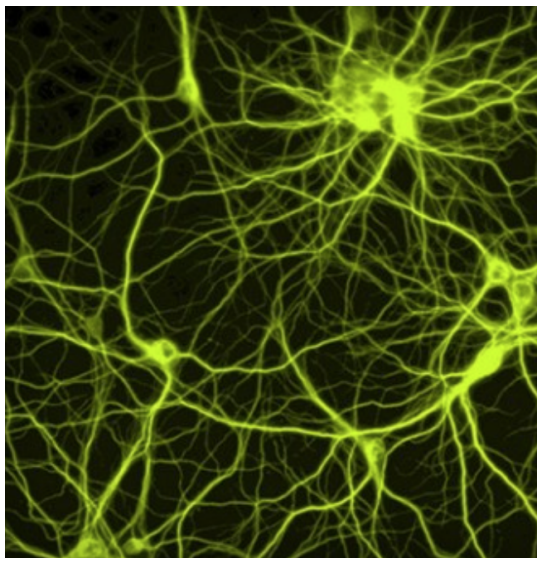
- Sample assumed to be independent and identically distributed observations :
  $(Y_1, X_1), (Y_2, X_2), \cdots, (Y_n, X_n)$
  $\rightarrow$ estimated model $Y \simeq \hat{f}(X)$.

Examples:

- Y: spam/non-spam, X: type of sender's address, number of addresses, title word frequency, message word frequency, number of words in the message, etc.
- Y: normal connection/attack connection, X: connection time, connection hour, click history, etc.

Introduction

- Artificial neural networks (ANN) are non linear regression models with a particular structure.
  They can be interpreted as a functional/mathematical imitation of a simplified model of biological neurons.
  They are used to predict numerical or categorical variables.
- As for the linear model, the most important steps to calibrate an ANN are
  - choosing the structure,
  - estimating the parameters.
- First studies of ANNs started in the 1940s. In the 1980s, computers were able to fit reasonable ANNs. Recently, there has been a renewed interest in ANNs because of deep learning.



Review of Linear Regression

Univariate Linear Regression: We explain the variable y based on a variable x

$$y = \beta_0 + \beta_1 x + \epsilon$$

Multivariate Linear Regression: We explain the variable y based on d variables $x_1, x_2, \cdots, x_d$

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_d x_d + \epsilon$$

which can also be written as

$$y = W\mathbf{x} + b + \epsilon$$

with the predictor $\mathbf{x} = (x_1, \cdots, x_d)^T$, the intercept $b = \beta_0$, and the weight vector $W = (\beta_1, \cdots, \beta_d)$.

Given the observations $(y_i, \mathbf{x}_i)_{i=1,\cdots,n}$, we estimate b and W by minimizing the least squares loss function.

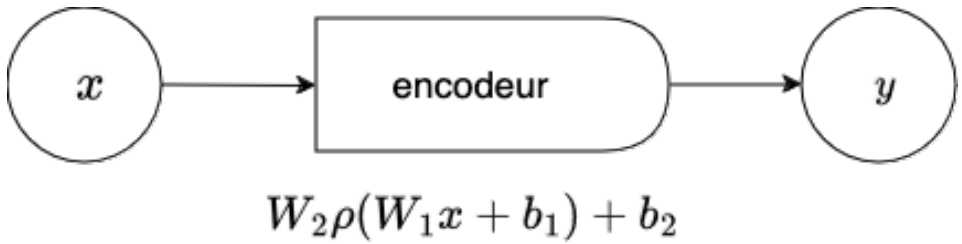
Example (multivariate):

- y: car braking time
- $\mathbf{x}$: (car speed, car weight, tire condition, road humidity)

Outline

Neural Network

A **"feedforward" neural network** is an algorithm that processes an input $x \in \mathbb{R}^d$ and returns an output $y \in \mathbb{R}$ (or $\mathbb{R}^k$).



The algorithm involves the repeated application of two simple operations:

1. An affine linear transformation,

$$A(x) = Wx + b$$

2. A nonlinear activation function ρ
applied coordinate-wise.

In the simplest case, these two operations are repeated multiple times through composition.

A Neuron = Perceptron

A single neuron is defined as follows:

$$\Phi(x) = \rho(Wx + b)$$

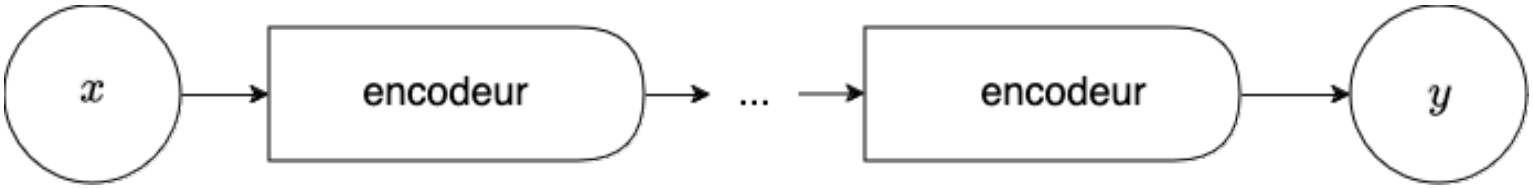
where ρ is the **activation function**, W represents the **weights**, and b is the **bias**.

This is called a **perceptron**.

The unknown **trainable** parameters are the weights W and the bias b.

Neural Network

A **"feedforward" neural network** processes information by composition.



1. Let the **input** be $\hat{x}^0 = x$.
2. For $1 \leq \ell \leq L$,
$$x^\ell = A^\ell(\rho(x^{\ell-1})) = W^\ell \rho(\hat{x}^{\ell-1}) + b^\ell$$
3. The **output** is $y = \Phi(x) = x^L$ (in the case of regression).

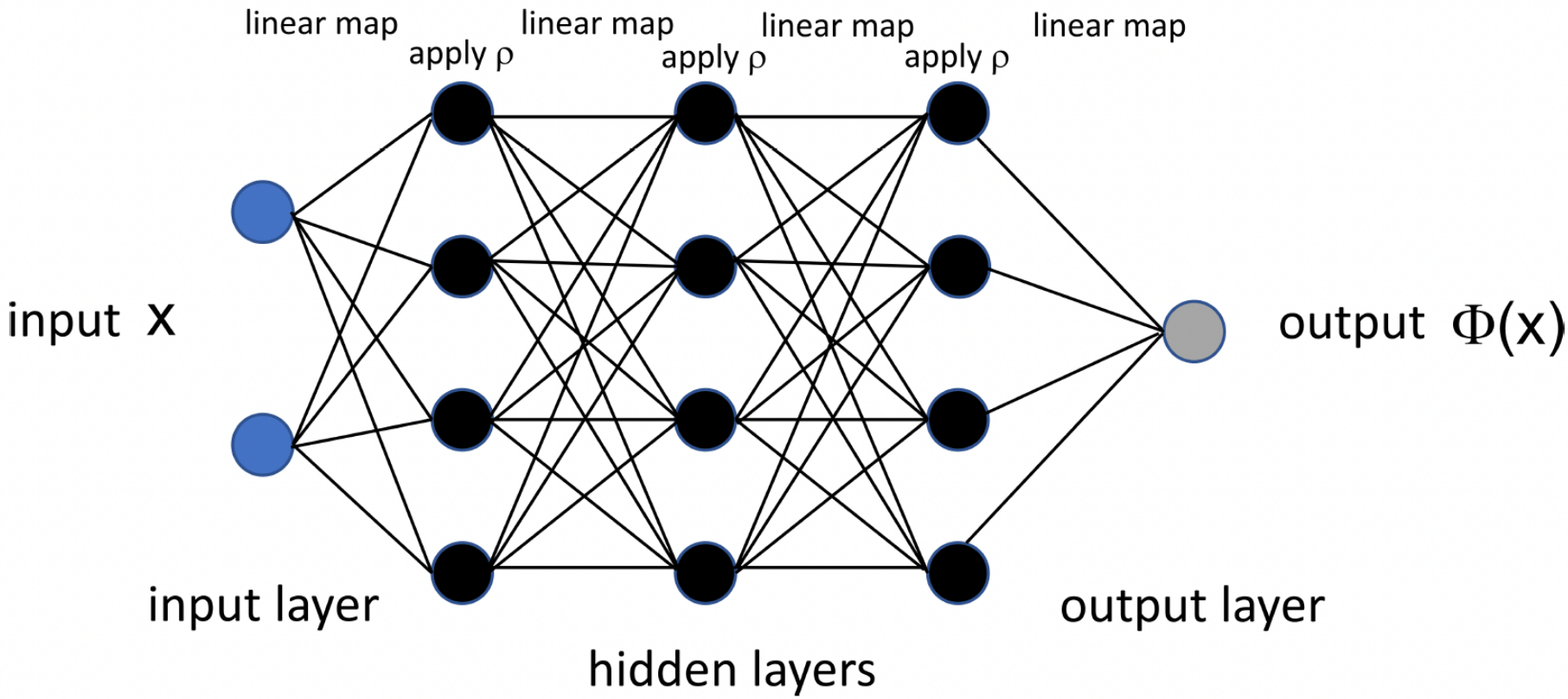
The values x^ℓ, $1 \leq \ell \leq L-1$, which are not directly observed, correspond to the **latent layers**[1] or **latent variables** of the neural network.

Note: We will see later that the values of the weight vectors W^ℓ and biases b^ℓ, $1 \leq \ell \leq L$, are estimated by minimizing a loss function.

[1] Also known as **hidden layers**

Neural Network

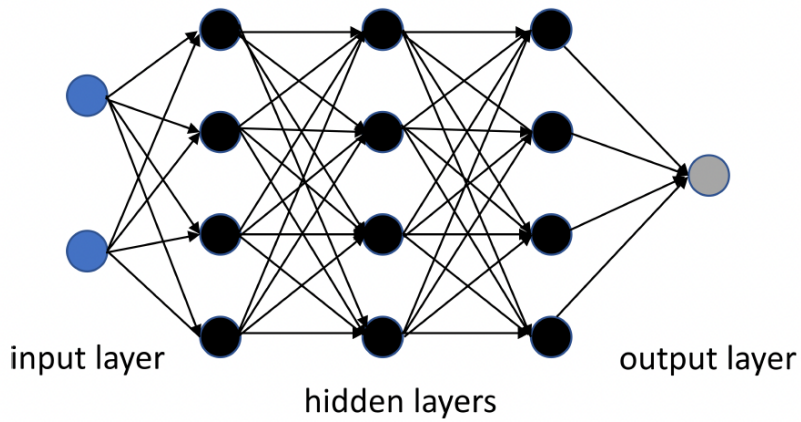
Graphical representation of a neural network with input $x \in \mathbb{R}^2$, output $\Phi(x) \in \mathbb{R}$, and 4 layers (L=4) including 3 hidden layers.



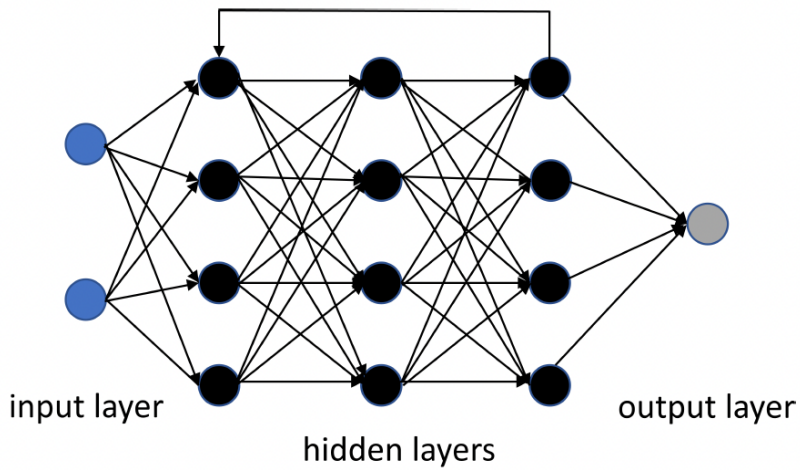
The information is propagated from the input to the output
$\rightarrow$ **feedforward architecture**

Neural Network

When information is propagated from input to output, the neural network is called a **feedforward neural network** or a **multilayer perceptron**.
Neural networks that include backward connections are called **recurrent neural networks** (not covered in this course).



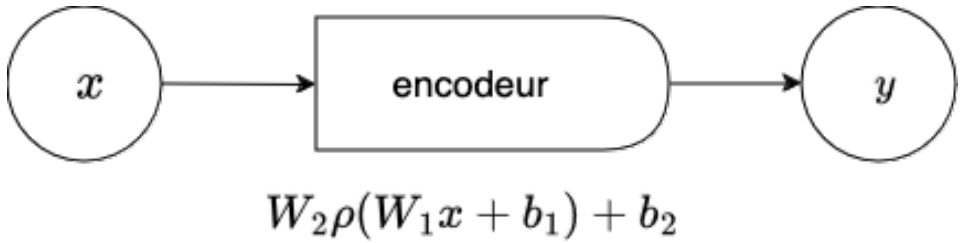
Feedforward neural network



Recurrent neural network

Feedforward Neural Network

A feedforward neural network with one hidden layer is called a shallow network.



In this case, we apply a linear transformation to the input $x \in \mathbb{R}^d$, followed by the activation function ρ, and then another linear transformation:

$$\Phi(x) = W^2\rho(W^1x + b^1) + b^2$$

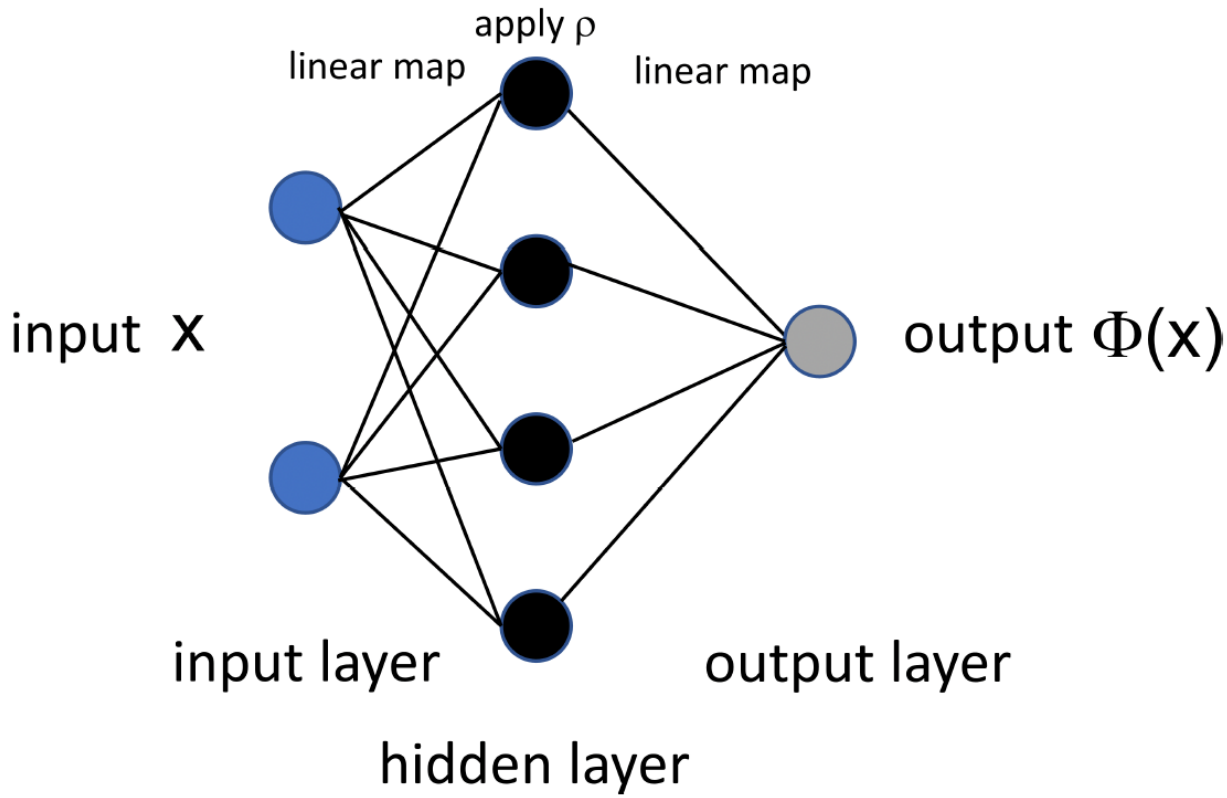
where $W^1 \in \mathbb{R}^{d,N}$, $b^1 \in \mathbb{R}^N$, $W^2 \in \mathbb{R}^{m,1}$, $b^2 \in \mathbb{R}$, and N is the number of hidden neurons.

If the input is univariate ($d = 1$), we can write:

$$\Phi(x) = W^2\rho(\sum_{j=1}^{N} W_j^1 x + b_j^1) + b^2$$

Exercise:
1. Represent a shallow neural network with one hidden layer containing 4 neurons (L=2), with an input $x \in \mathbb{R}^2$ (i.e., two variables) and an output $\Phi(x) \in \mathbb{R}$.
2. Write the equation for computing $\Phi(x)$ and specify the dimensions of the weight matrices and bias vectors.

Graphical representation of a shallow neural network with one hidden layer containing 4 neurons (L=2), with an input $x \in \mathbb{R}^2$ (i.e., two variables) and an output $\Phi(x) \in \mathbb{R}$.



$$\Phi(x) = W^2 \rho(W^1 x + b^1) + b^2$$

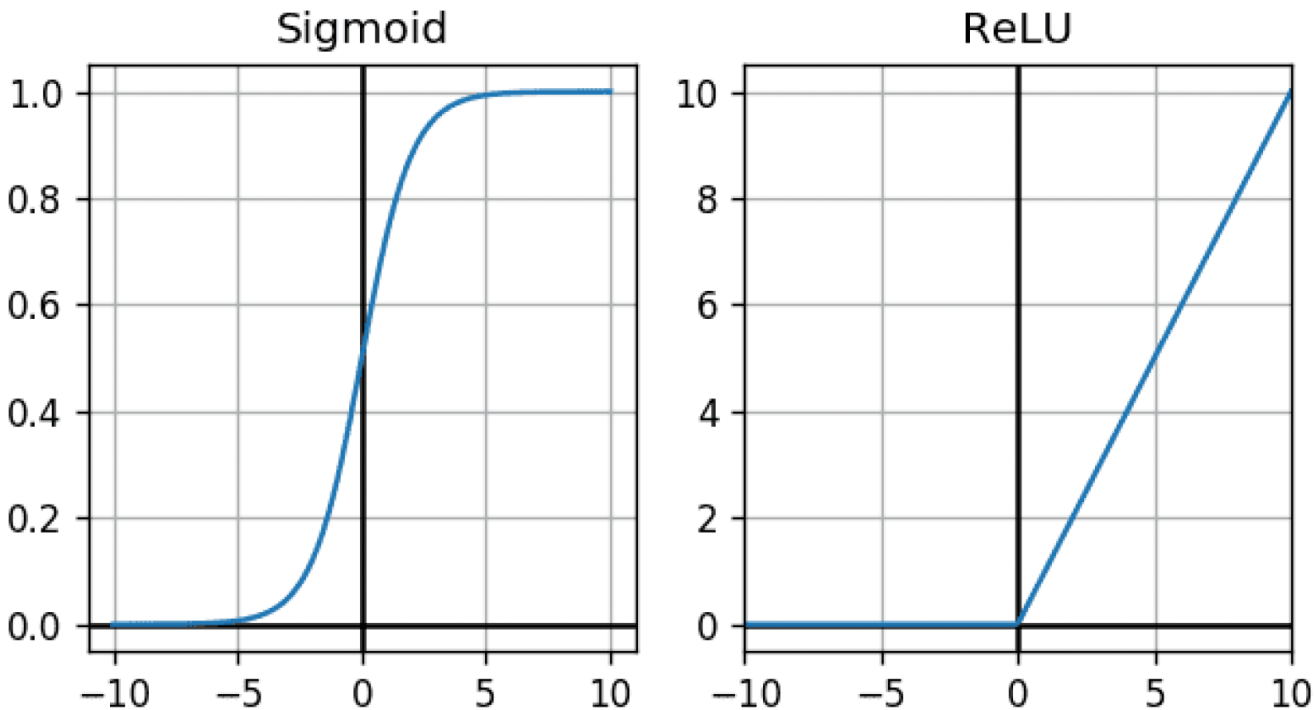
The weights are stored in matrices $W^1 \in \mathbb{R}^{4,2}$ and $W^2 \in \mathbb{R}^{1,4}$.

Activation Function

The role of the activation function is to introduce non-linearities into the neural network.

The two most common **activation functions** ρ are:

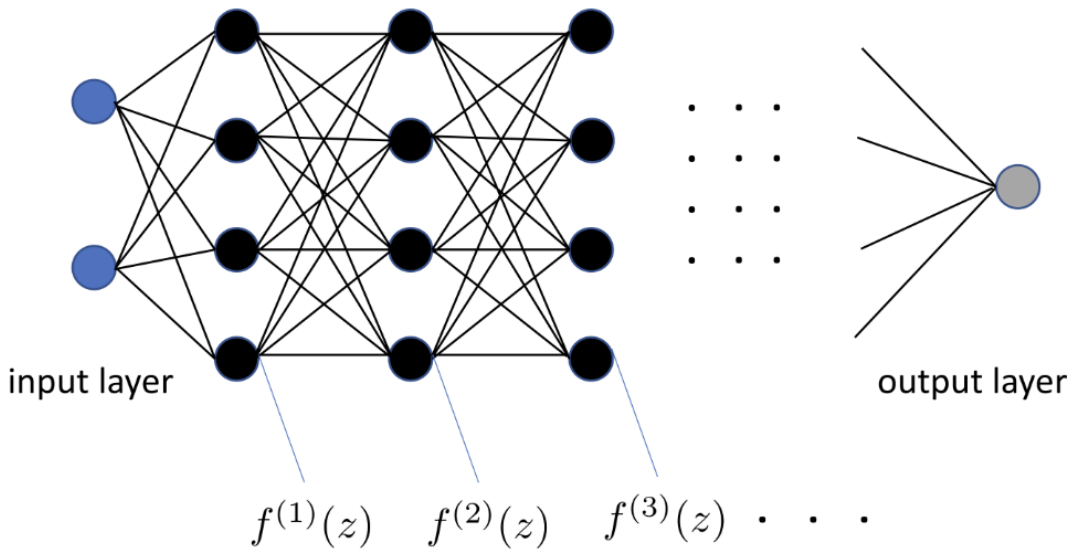
- The **sigmoid function**: $\rho(x) = \frac{1}{1+e^{-x}}$
- The **Rectified Linear Unit (ReLU)**: $\rho(x) = \max(x, 0)$



The ReLU function is the most commonly used, but the sigmoid function has the advantage of being differentiable. An alternative to the sigmoid function is the tanh function.

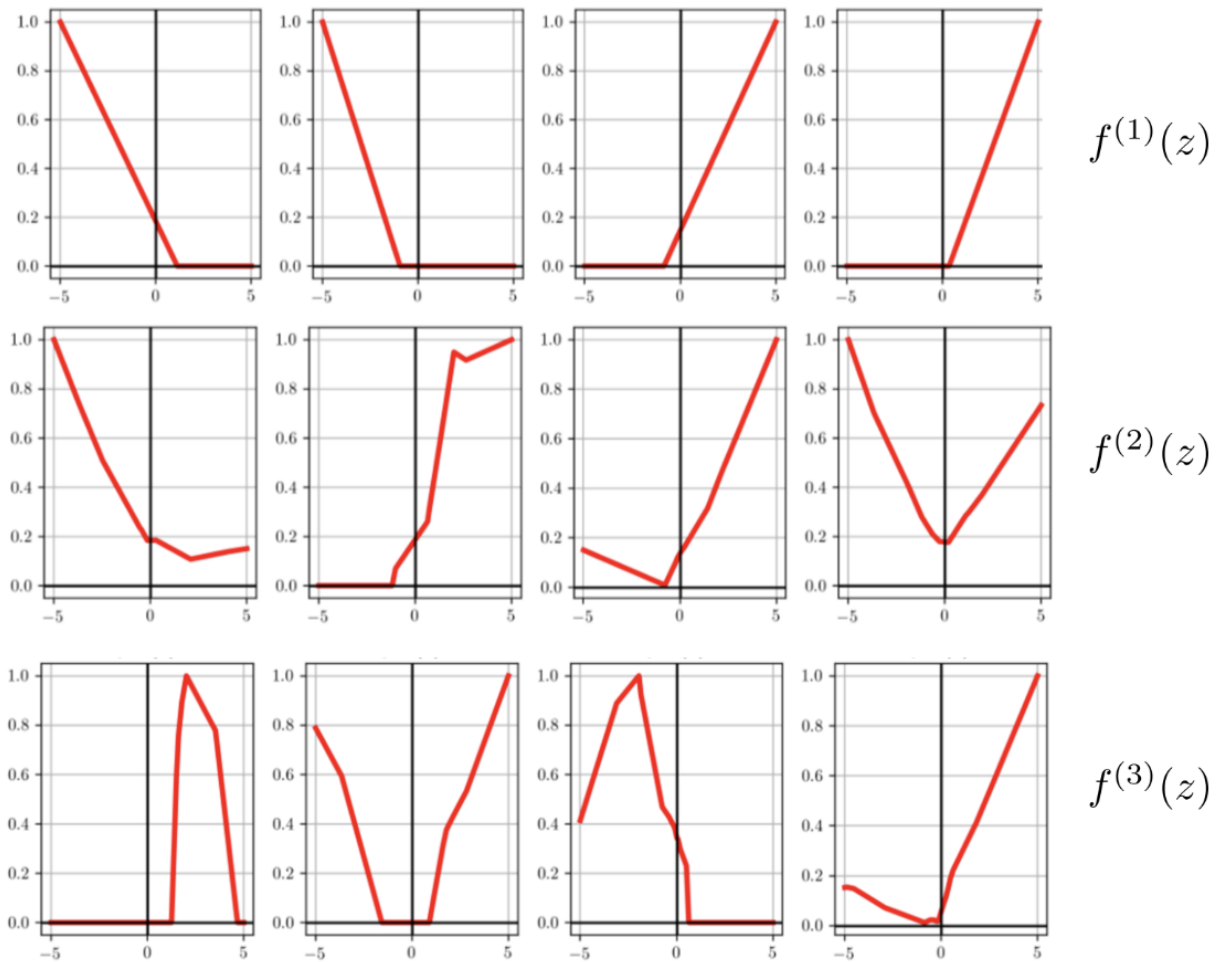
The Power of Neural Networks

By combining simple functions, a neural network can represent complex shapes.
Consider a neural network with at least 3 hidden layers as shown below.
We will show the neuron outputs for two different activation functions.



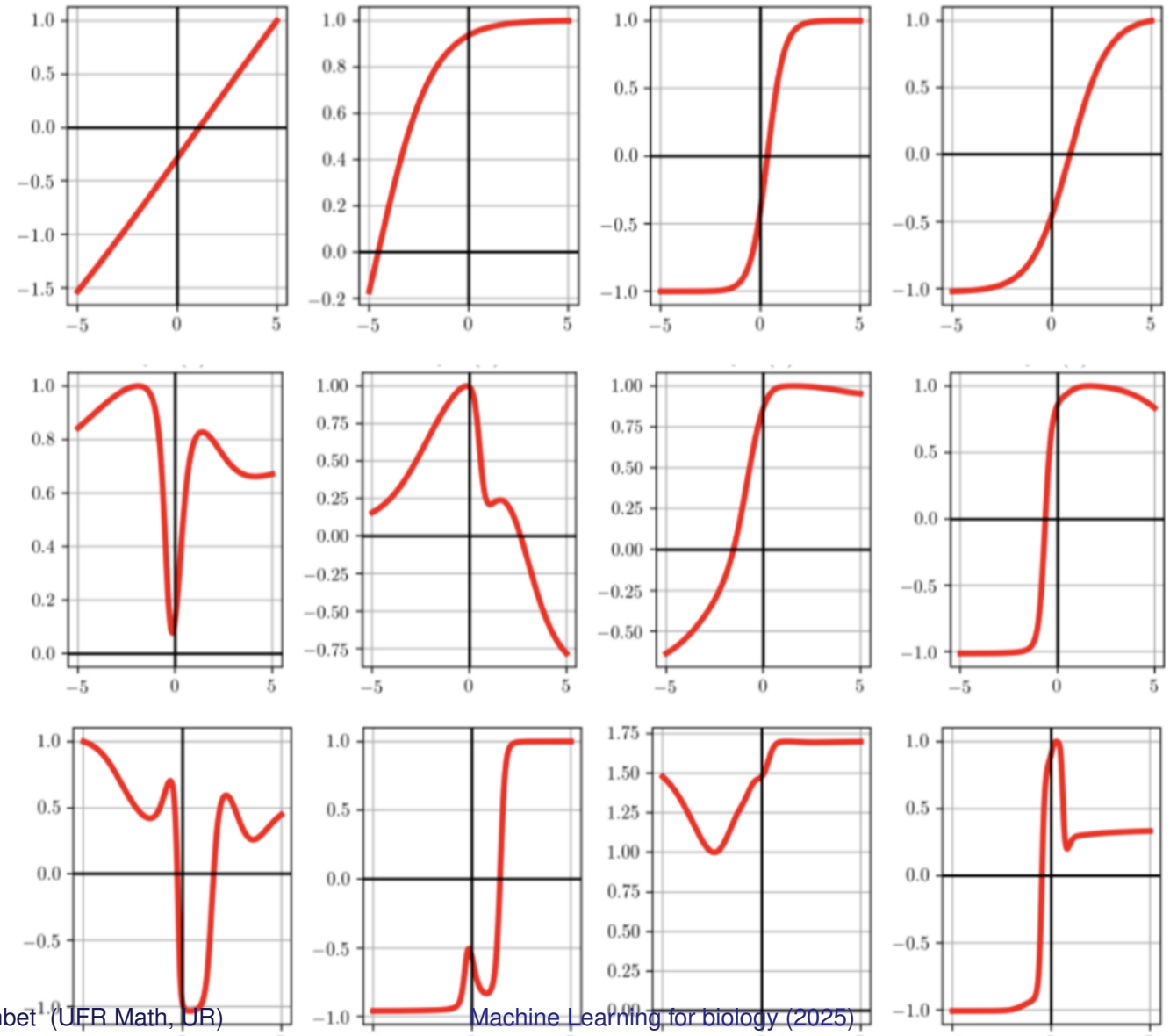
Activation Function

Here, we show the neuron outputs for the first 3 layers with random inputs using the ReLU activation function.



Activation Function

Here, we show the neuron outputs for the first 2 layers with random inputs using the tanh activation function.

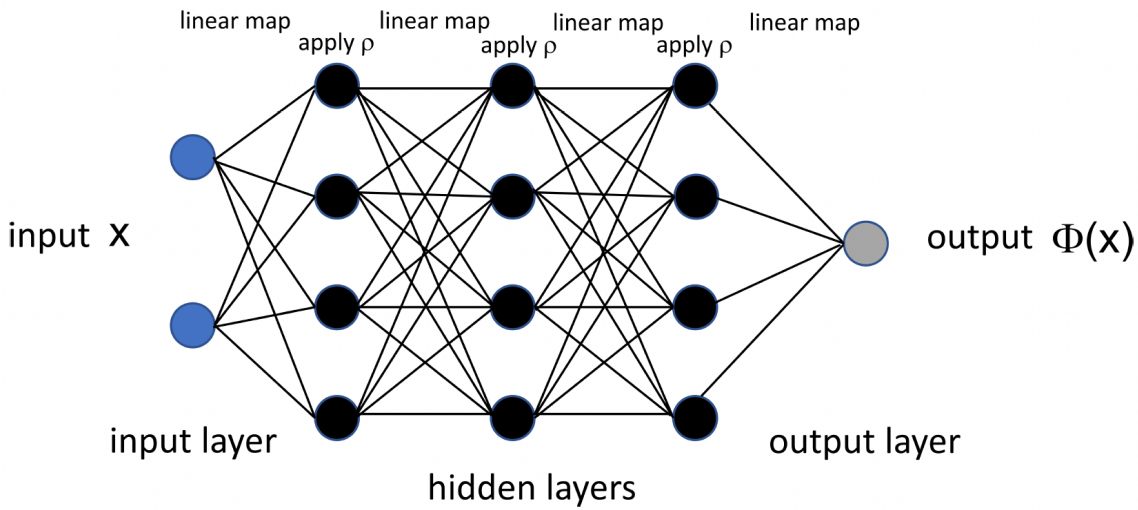


Deep Neural Network

A deep neural network has multiple hidden layers (sometimes a very large number).

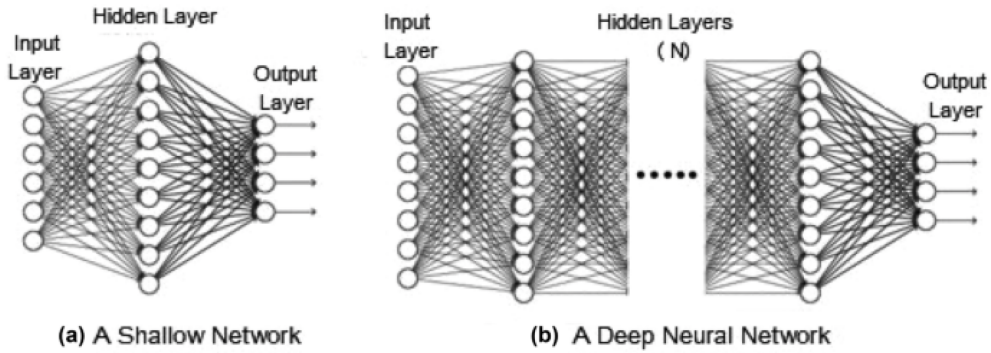
$$\Phi(x) = W^4\rho(W^3\rho(W^2\rho(W^1x + b^1) + b^2) + b^3) + b^4$$

In this example, we have a neural network with 4 layers, including 3 hidden layers.



Note: For the network Φ to be well-defined, the dimensions of the matrices W^ℓ and the vectors b^ℓ must be consistent.

Deep Neural Network



- Modern architectures are often very deep.
- A deep architecture involves a large number of parameters (weights).
- Designing a deep network with many parameters can be useful for learning complex models for non-trivial tasks, but it can also lead to significant overfittingif the training set is small or if the task is simple.

Outline

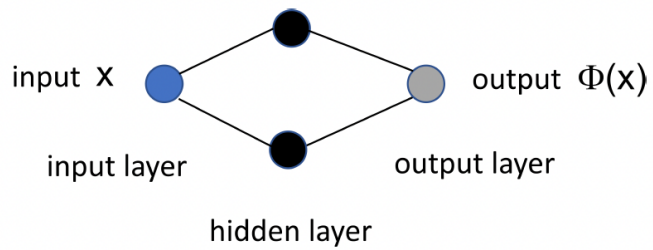
Approximation Theorems

- Universal approximation theorems indicate that neural networks have the capacity to approximate any continuous multivariate function with a certain level of accuracy.
- The representational power of a neural network increases with the number of neurons and layers.
- Caution: Even if a neural network is capable of representing a function, the training algorithm may fail to learn that specific function.
- Learning can fail
  (i) because the optimization algorithm used for training may not be able to find the parameter values corresponding to the desired function, or
  (ii) because the training algorithm might select the wrong function, resulting in overfitting.

Example 1: The Triangle Function

$$T(x) = \begin{cases} 2x & \text{if} \quad 0 \leq x \leq 1/2 \\ 2(1-x) & \text{if} \quad 1/2 \leq x \leq 1 \end{cases}$$

The function T can be approximated by a neural network with one hidden layer containing 2 hidden units and the ReLU activation function.



We have a neural network with 7 parameters:

$$\Phi(x) = [w_{21} \quad w_{22}]\rho\left(\begin{bmatrix} w_{11} \\ w_{12} \end{bmatrix} x + \begin{bmatrix} b_{11} \\ b_{12} \end{bmatrix}\right) + b_2$$

with

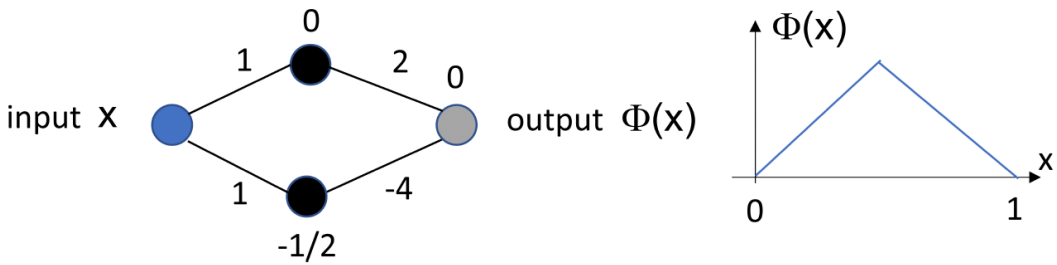
$$\rho(wx+b) = (wx+b)_+$$

We can rewrite:

$$\Phi(x) = (w_{21}(w_{11}x + b_{11})_+ + w_{22}(w_{12}x + b_{12})_+ + b_2)_+$$

Moreover, we can verify that:

$$T(x) = (2(x-0)_+ - 4(x-1/2)_+)_+$$



We can observe that each term of Φ is associated with a portion of a piecewise linear function.

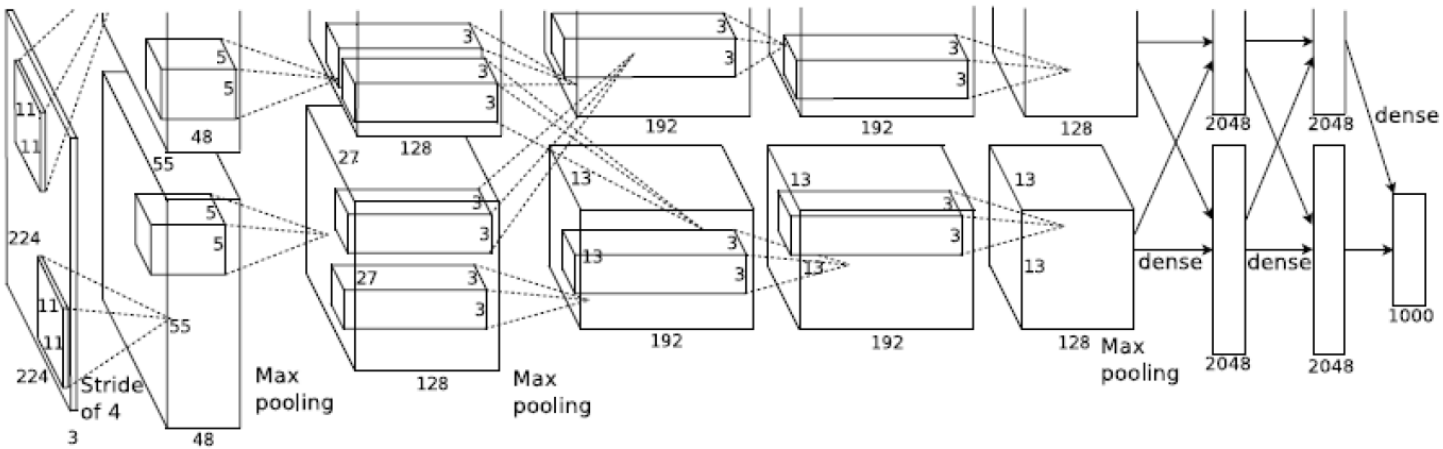
- Any triangle function can be represented with a neural network having 2 hidden neurons and ReLU activation functions.
- More generally, any piecewise linear function can be represented by a neural network with ReLU activation functions.

Example 2: Deep Neural Network for Image Classification

- The MNIST database (digits) contains 70,000 binary images of size 28×28, each representing a digit.
- To connect with what was mentioned earlier, $x \in [0,1]^{28\times 28}$, $y \in \{0,1\}^{10}$, and each y_j has exactly one non-zero coordinate.

This dataset is widely used for experimentation and comparison purposes. In a typical setup, the first layer is linear, and the layer dimensions are as follows: 784-1000-500-250-10.

- Even deeper architectures are used for classifying more complex images.



Outline

Artificial Intelligence, Machine Learning

Neural networks are used to automate a wide variety of tasks:

- Non-linear regression
  e.g.: predicting housing prices, predicting a clinical variable, predicting an atmospheric variable
- Classification
  e.g.: recognizing the content of an image
- Image generation
  e.g.: creating logos
- Text generation
  e.g.: ChatGPT, automatic translation

From one problem to another, we change the neural network architecture, the activation function on the output layer, and the loss function.

Let us firstly ook at en exmaple of prediciton via a neural network in a (very) smmal regresion problem.

Regression

Let $\{(x_1, y_1), \cdots, (x_n, y_n)\}$ be a training set where $x_i \in \mathbb{R}^d$ and $y_i \in \mathbb{R}$.

The neural network's prediction for a given input x_i is the result of the forward pass

$$\Phi(x_i; \theta)$$

where θ is the vector of all weights w and biases b.

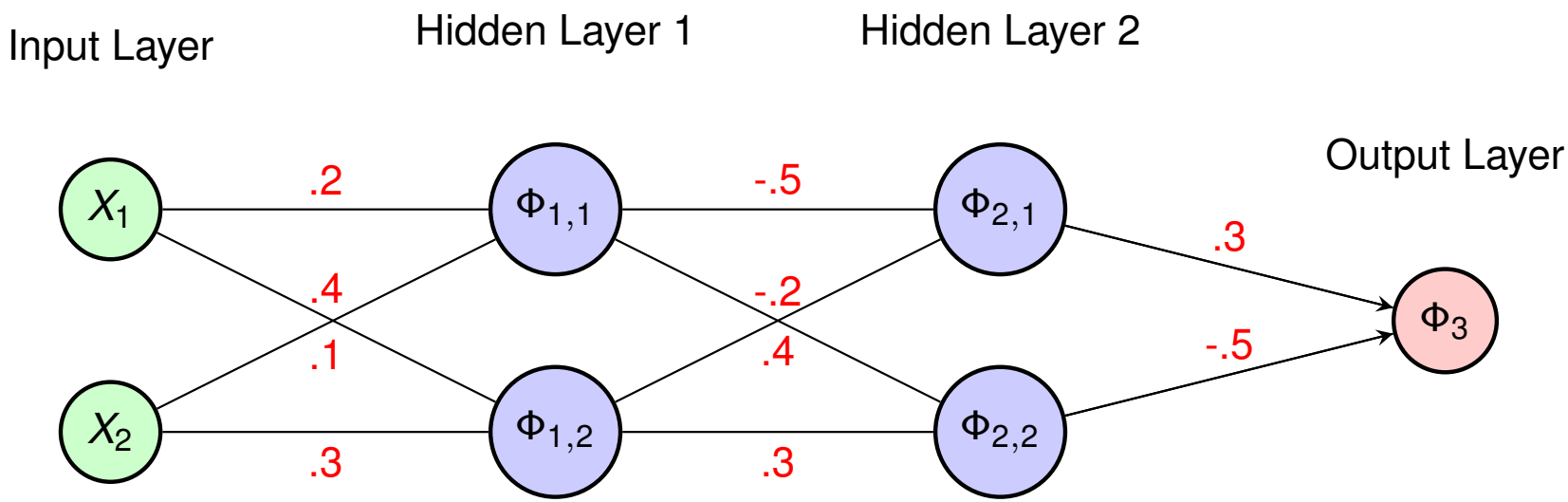
As in the case of linear regression, we use the least squares error to measure the quality of the prediction

$$L(x, y; \theta) = \frac{1}{2} \sum_{i=1}^{n} \left(y_i - \Phi(x_i; \theta)\right)^2$$

To estimate the parameters θ, we need to minimize the loss function and therefore calculate $\Phi(x_i; \theta)$ for all i. However, the function Φ does not have a simple analytical form: it is evaluated numerically through the forward pass.

Forward pass example

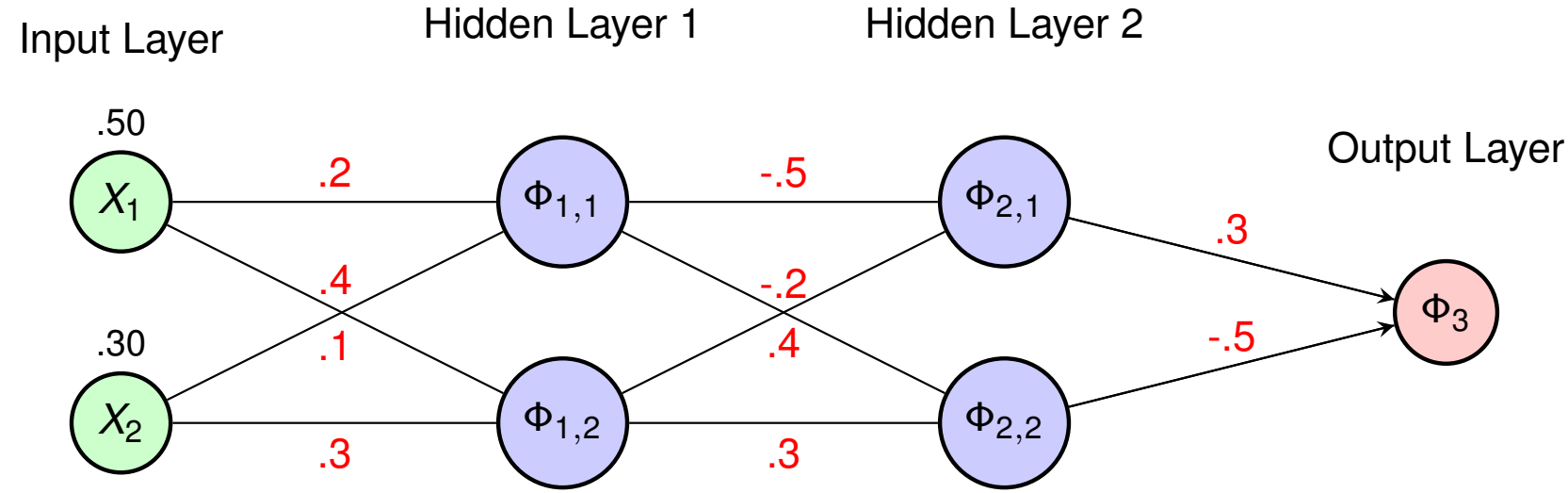
Let us consider a neural network with 2 inputs, 1 outpu et 2 hidden layers, with 2 units each. The bias are assumed to be 0. The activation function is the RELU one for all the hidden layers.



Exercice : run the forward pass to compute the output for the input $X = (.5, .3)$.

Forward pass example

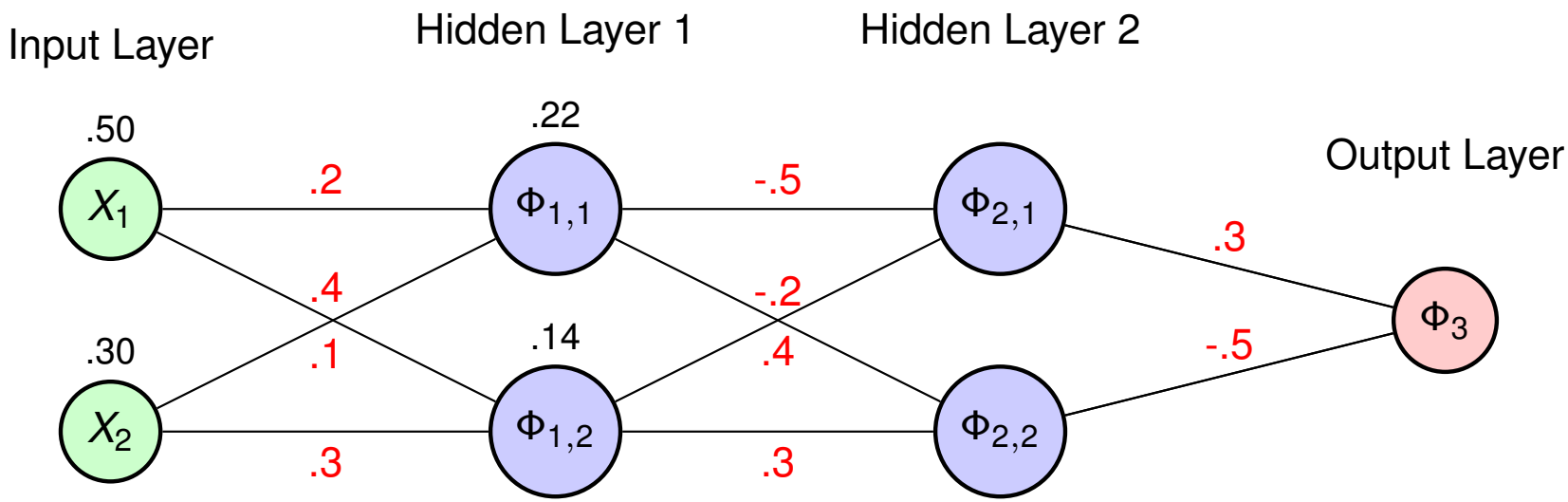
Entrée $X = (.5, .3)$



Forward pass example

First hidden layer (RELU activation function)

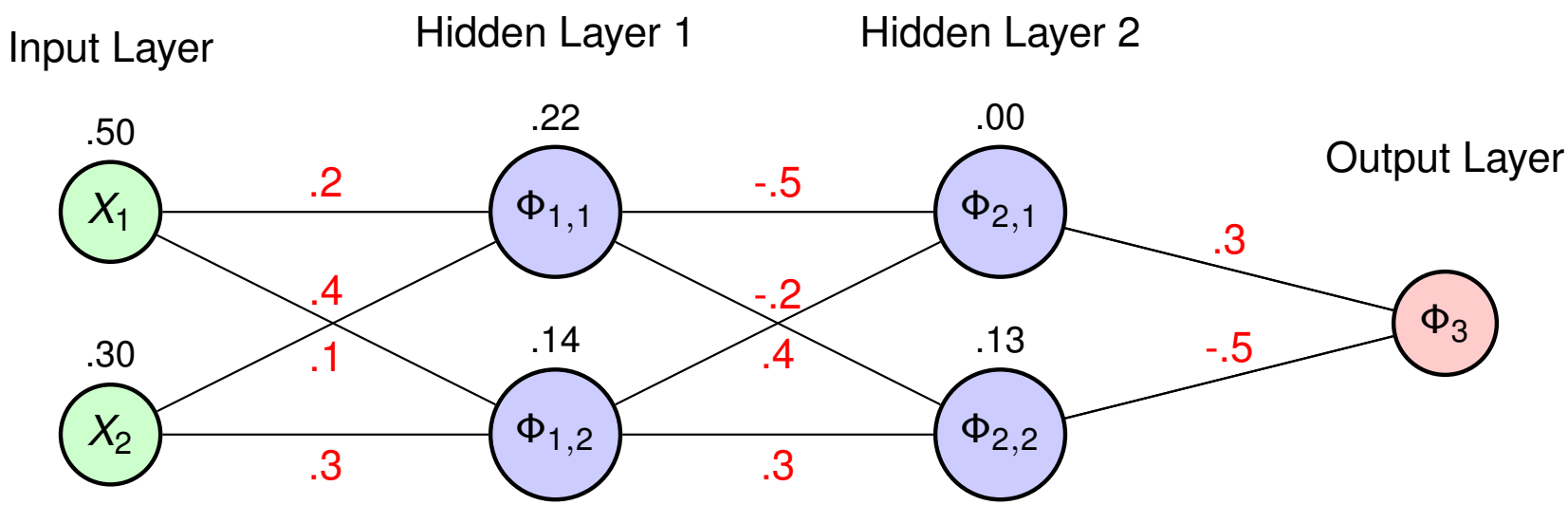
$$\Phi_{11}(X) = (.5 * .2 + .3 * .4)_+, \quad \Phi_{12}(X) = (.5 * .1 + .3 * .3)_+$$



Forward pass example

Second hidden layer (RELU activation function)

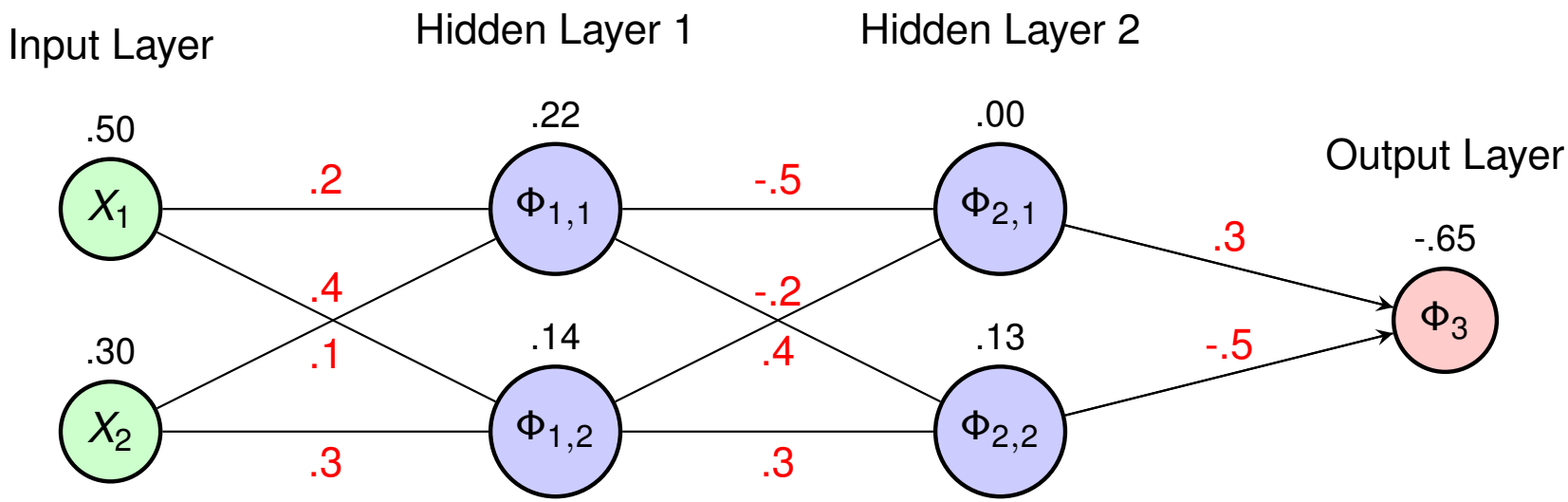
$$\Phi_{21}(X) = (-.22 * .5 - .14 * ..2)_+ = 0, \quad \Phi_{22}(X) = (.22 * .4 + .14 * .3)_+$$



Forward pass example

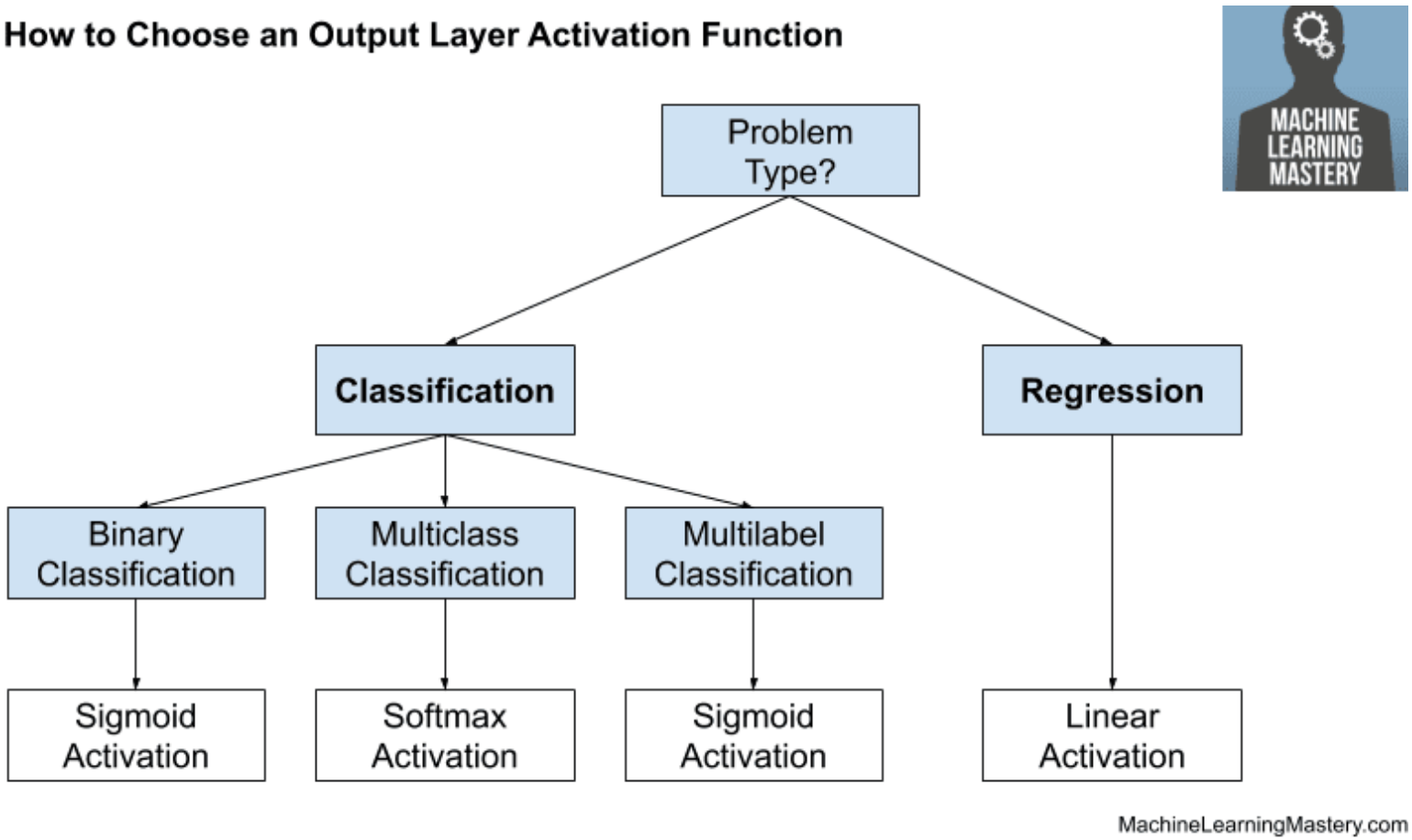
Output layer (identity activation function)

$$\Phi_3(X) = (.00 * .3 - .13 * ..5)_+ = -0.65$$



What change for classification problems?

Nothing but the output layer activation function!



sigmoid	softmax	linear
$\frac{exp(y)}{1+exp(y)}$	$\frac{exp(y_\ell)}{\sum_{k=1}^{K} exp(y_k)}$	y

Outline

Learning

Learning (or training) of a neural network involves estimating its parameters (i.e., weight matrices W_k for each layer k) given a training set $\{(x_1, y_1), \cdots, (x_n, y_n)\}$.

Regression

If the output y is a quantitative variable, the mean squared error loss function is commonly used:

$$\hat{\omega} = \arg\min_{\omega \in \Omega} \underbrace{\frac{1}{n} \sum_{i=1}^{n} (y_i - \Phi(x_i; \omega))^2}_{L(\omega)}$$

where $\mathcal{F}$ denotes the set of functions Φ obtained by varying the weight matrices (for a fixed architecture).

Classification

If the output y is a binary variable, $\Phi(x)$ models the probability that y is equal to 1, and the cross-entropy loss function[2] is used:

$$\hat{\omega} = \arg\min_{\omega \in \Omega} \underbrace{- \sum_{i=1}^{n} y_i \log \Phi(x_i) + (1 - y_i) \log(1 - \Phi(x_i))}_{L(\omega)}$$

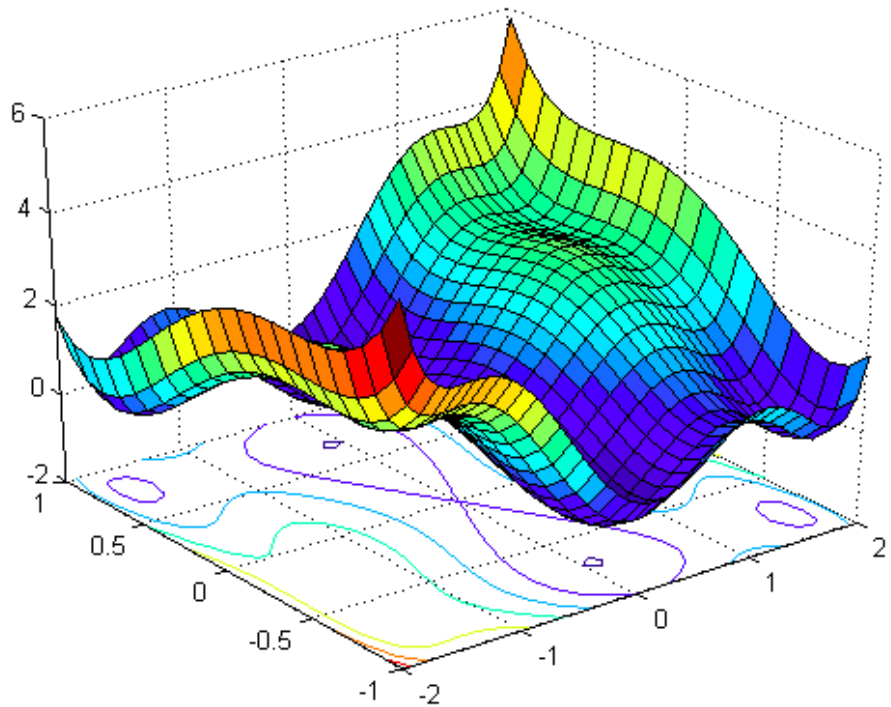
These problems do not have an explicit solution and require the use of a numerical optimization algorithm.

[2] In the mathematical community it is also called "likelihood"

Outline

Optimization problem

In Machine Learning, what ever the method we choose, we have to minimize a loss function L with respect to ω (the vector of all trainable parameters).
In other words, during the training process, we tweak and change the parameters (weights) of our model to try and minimize that loss function, and make our predictions as correct and optimized as possible. But how exactly do you do that? How do you change the parameters of your model, by how much, and when?



Source site:
`https://algorithmia.com/blog/introduction-to-loss-functions`

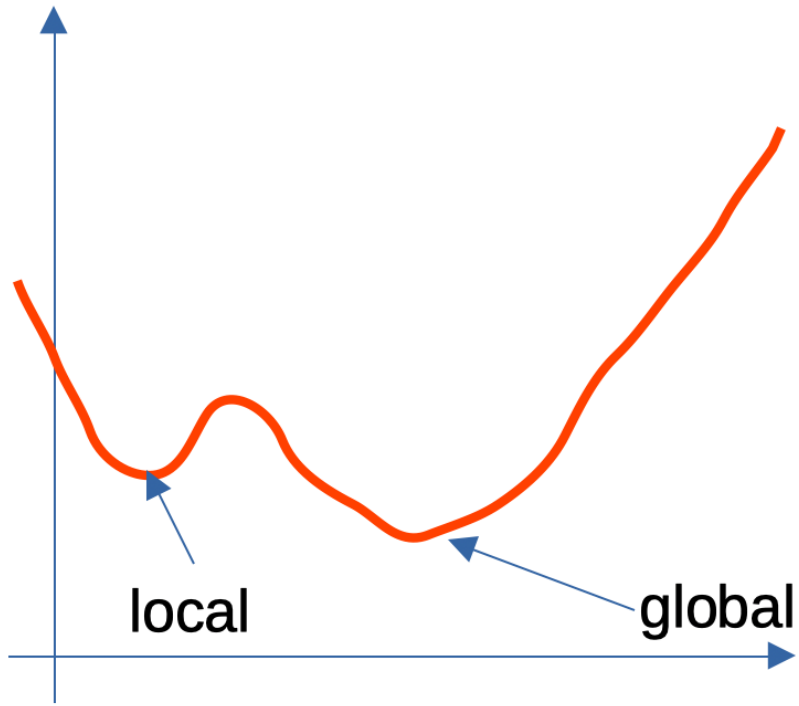
Local extrema necessary condition

Theorem

Local extrema necessary condition

Let $L : \mathbb{R} \to \mathbb{R}$ be a real valued function. If L admits a local mimum w and if it is differentiable then $\nabla L(w) = 0$.

Note that the **Gradient** ∇L is the generalization of the derivative to multivariate functions.



Gradient descent

Generally, if we want to find the minimum of a function, we set the derivative to zero and solve the obtained equations for the parameters.
It is done for instance solving a linear regression problem.

It turns out, however, it is impossible to get a closed form solution in many Machine Learning methods.

This is where optimizers come in. They tie together the loss function and model parameters by updating the model in response to the output of the loss function. In simpler terms, optimizers shape and mold your model into its most accurate possible form by futzing with the weights. The loss function is the guide to the terrain, telling the optimizer when it's moving in the right or wrong direction.

We iteratively search for a minimum using a method called gradient descent.

Gradient descent image

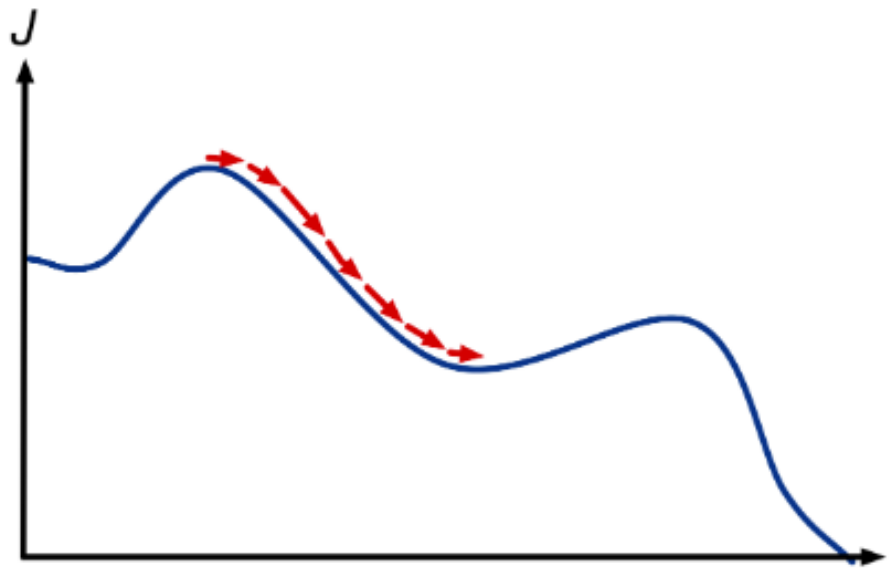
As a visual analogy, you can think of a hiker trying to get down a mountain with a blindfold on. It's impossible to know which direction to go in, but there's one thing she can know: if she's going down (making progress) or going up (losing progress). Eventually, if she keeps taking steps that lead her downwards, she'll reach the base.

Source site: `https://www.deepideas.net/deep-learning-from-scratch-iv-gradient-descent-and-backpropagation/`

Gradient descent ideas

Gradient descent operates in a similar way when trying to find the minimum of a function: It starts at a random location in parameter space and then iteratively reduces the error J until it reaches a local minimum.

At each step of the iteration, it determines the direction of steepest descent and takes a step along that direction. This process is depicted for the 1-dimensional case in the following image.



Source site: `https://www.deepideas.net/deep-learning-from-scratch-iv-gradient-descent-and-backpropagation/`

Gradient descent algorithm

As you might remember, the direction of steepest ascent of a function at a certain point is given by the gradient at that point. Therefore, the direction of steepest descent is given by the negative of the gradient.

So now we have a rough idea how to minimize L:

1. Start with random values for the parameters (or weights) ω
2. Calculate what a small change in each individual weight would do to the loss function
$\xrightarrow{math}$ Compute the gradient of L with respect to ω.
3. Adjust each individual weight based on its gradient
$\xrightarrow{math}$ Take a small step along the direction of the negative gradient
4. Go back to 2 until convergence

Gradient represent what a small change in a weight or parameter would do to the function.

$$\nabla L(\omega_j) \simeq \frac{L(\omega_j + h) - L(\omega_j)}{h} \text{ for } h \text{ small}$$

Gradient, formal definition

Definition

The gradient of $f : (\omega_1, \ldots, \omega_p) \mapsto f(\omega_1, \ldots, \omega_p)$ is the vector of the partial derivatives.

$$\nabla f(\boldsymbol{\omega}) = \begin{pmatrix} \frac{\partial f(\boldsymbol{\omega})}{\partial \omega a_1} \\ \vdots \\ \frac{\partial f(\boldsymbol{\omega})}{\partial \omega_p} \end{pmatrix}$$

where $\boldsymbol{\omega} = (\omega_1, \ldots, \omega_p)$ and

$$\frac{\partial f(\omega_1, \ldots, \omega_p)}{\partial \omega_j} = \lim_{h \to 0} \frac{f(\omega_1, \cdots, \omega_{j-1}, \omega_j + h, \omega_{j+1}, \cdots, \omega_p) - f(\omega_1, \ldots, \omega_p)}{h}$$

We remark that the gradient as the same dimension as $\boldsymbol{\omega}$.

Gradient descent algorithm

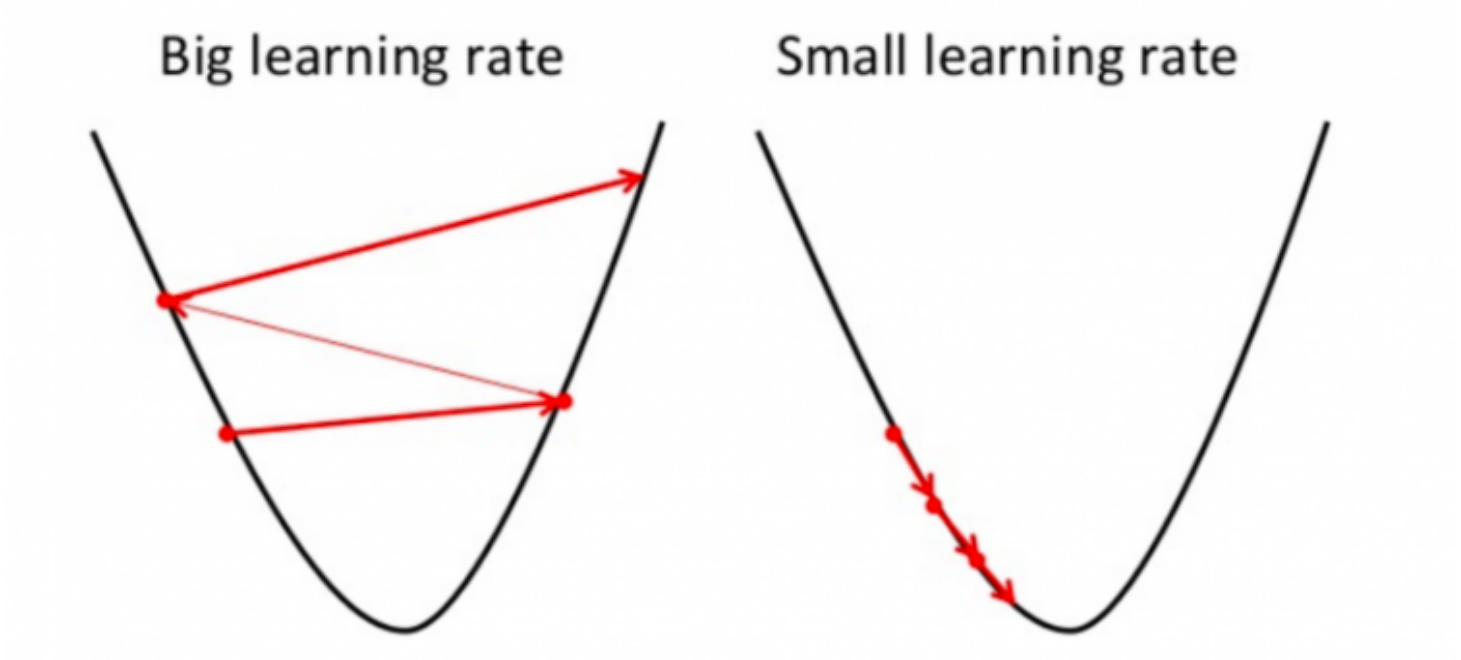
Gradient descent algorithm with **learning rate** α

1. Sample randomly an initial value $\omega_{[0]}$ for the weights
2. Compute the gradient $\nabla L(\omega_{[r-1]})$ of the loss function J at the current value of the weights $\omega_{[r-1]}$
3. Update the weights:

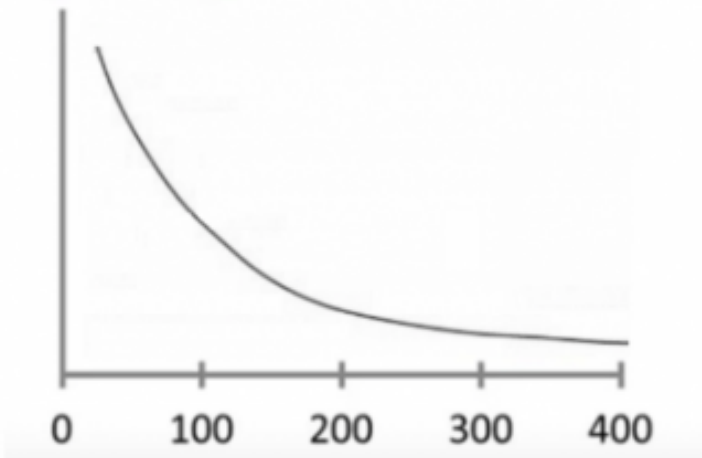
$$\omega_{[r]} = \omega_{[r-1]} - \alpha \nabla L(\omega_{[r-1]})$$

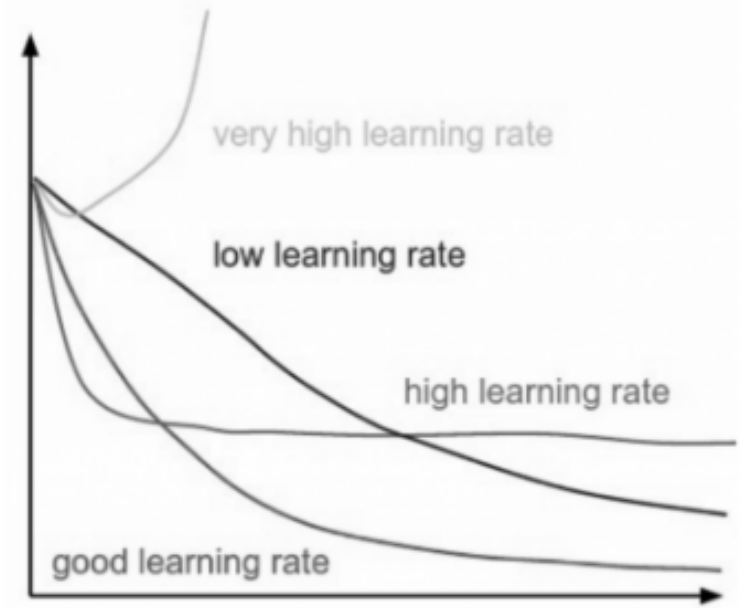
4. Go back to 2. until convergence.

Is it important to choose a good learning rate



A good way to make sure gradient descent runs properly is by plotting the cost function as the optimization runs. Put the number of iterations on the x axis and the value of the cost-function on the y axis.





When gradient descent can't decrease the cost-function anymore and remains more or less on the same level, it has converged. The number of iterations gradient descent needs to converge can sometimes vary a lot. It can take 50 iterations, 60,000 or maybe even 3 million, making the number of iterations to convergence hard to estimate in advance.

Let's practice!

Let's start with a simple 1d example. Just copy the url below, paste it on your favorite web navigator. The goal is to achieve the minimum of the function. Try different values from the learning rate and observe the results (achievement, total number of iterations).
`https://developers.google.com/machine-learning/crash-course/linear-regression/gradient-descent-exercise?hl=fr`

Now, you are ready to play with your 1st neural network training. Just copy the url below, paste it on your favorite web navigator. Go to Exercice 2.
`https://developers.google.com/machine-learning/crash-course/neural-networks/interactive-exercises?hl=fr`
Then click on the blue link "playground" at the top of the page (to get the right exercice) and follow the instructions.

Let's go back to learning a neural network

Learning (or training) of a neural network involves estimating its parameters (i.e., weight matrices W_k for each layer k) given a training set $\{(x_1, y_1), \cdots, (x_n, y_n)\}$.

Regression

If the output y is a quantitative variable, the mean squared error loss function is commonly used:

$$\hat{\omega} = \arg\min_{\omega \in \Omega} \underbrace{\sum_{i=1}^{n} (y_i - \Phi(x_i; \omega))^2}_{L(\omega)}$$

where $\mathcal{F}$ denotes the set of functions f obtained by varying the weight matrices (for a fixed architecture).

Classification

If the output y is a binary variable, $\Phi(x)$ models the probability that y is equal to 1, and the likelihood[3] is used:

$$\hat{\omega} = \arg\min_{\omega \in \Omega} \underbrace{-\sum_{i=1}^{n} y_i \log \Phi(x_i) + (1 - y_i) \log(1 - \Phi(x_i))}_{L(\omega)}$$

These problems do not have an explicit solution
and require the use of a numerical optimization algorithm.

[3] In the machine learning community: "Cross-entropy"

Loss Function Minimization

The common method for optimizing the parameters of a neural network relies on a gradient descent algorithm.
Denoting ω as the parameter vector and

$$L(\omega) = \frac{1}{n}\sum_{i=1}^{n} L_i(\omega)$$

as the loss function, we iterate until convergence:

$$\omega_r = \omega_{r-1} + \alpha \frac{1}{n}\sum_{i=1}^{n} \nabla_\omega L_i(\omega_r) \tag{1}$$

In practice, we use the specific structure of the network to obtain the efficient algorithm of **gradient backpropagation** (see appendix).

Stochastic Gradient Descent (SGD)

The stochastic gradient descent algorithm is inspired by the gradient descent algorithm, but at each iteration, the step is computed from a single example.

$$J(\mathbf{w}) = \frac{1}{n}\sum_{i=1}^{n} J_i(\mathbf{w}) \text{ and } J_i(\mathbf{w}) = (y_i - \phi(x_i; \mathbf{w}))^2$$

Gradient Descent Algorithm

Initialization: choose a point $\mathbf{w}_0 \in \Omega$
While $\nabla J(\mathbf{w}) > \epsilon$ and $k <$ maxiter do
    $\text{step}_k = -\alpha_k \nabla_{\mathbf{w}} J(\mathbf{w}_k)$
    $\mathbf{w} \leftarrow \mathbf{w}_{k+1} = \mathbf{w}_k + \text{step}_k$

Strict Stochastic Gradient Descent Algorithm

Initialization: choose a point $\mathbf{w}_0 \in \Omega$
While $\nabla J(\mathbf{w}) > \epsilon$ and $k <$ maxiter do
    Randomly draw i from $\{1, \cdots, n\}$
    $\text{step}_k = -\alpha_k \nabla_{\mathbf{w}} J_i(\mathbf{w})$
    $\mathbf{w} \leftarrow \mathbf{w}_{k+1} = \mathbf{w}_k + \text{step}_k$

Spot the differences!

Stochastic Gradient Descent (SGD)

Advantages

- Less computation (to calculate the gradient)
- Better exploration of the solution space, thus reducing the chance of getting stuck in a local minimum

Disadvantages

- The "descent" is stochastic: the weight values fluctuate a lot from one iteration to another
- Convergence can be slow

The **mini-batch** compromise: a small subset of examples is used. This subset is called a **mini-batch**.

Stochastic Gradient Descent (SGD)

Spot the differences!

Strict Stochastic Gradient Descent Algorithm

Initialization: choose a point $\mathbf{w}_0 \in \Omega$
While $\nabla L(\mathbf{w}) > \epsilon$ and $k < \text{maxiter}$ do
    Randomly draw i from $\{1, \cdots, n\}$
    $\text{step}_k = -\alpha_k \nabla_{\mathbf{w}} e_i(\mathbf{w})$
    $\mathbf{w} \leftarrow \mathbf{w}_{k+1} = \mathbf{w}_k + \text{step}_k$

Stochastic Gradient Descent Algorithm (with mini-batch)

Initialization: choose a point $\mathbf{w}_0 \in \Omega$
While $\nabla L(\mathbf{w}) > \epsilon$ and $k < \text{maxiter}$ do
    Randomly draw a set MB of b indices from $\{1, \cdots, n\}$
    $\text{step}_k = -\alpha_k \nabla_{\mathbf{w}} \sum_{i \in MB} e_i(\mathbf{w})$
    $\mathbf{w} \leftarrow \mathbf{w}_{k+1} = \mathbf{w}_k + \text{step}_k$

Let's me show you an example

Connect to the web site https://developers.google.com/machine-learning/crash-course/introduction-to-neural-networks/playground-exercises
Key points

1. Generation of the data set (with noise or not, different sizes).
2. Let's observe the evolution of the boundary shape when the activation function is changed (compared to linear).
3. Let's observe the evolution of the boundary shape when neurons or layers are added (compared to a model with only one neuron).
4. What does mean "training loss" and "test loss"?
5. Remark that changing initialisation of the learning algorithm will lead to different results.

Let's practice

Connect to the web site https://developers.google.com/machine-learning/crash-course/introduction-to-neural-networks/playground-exercises and go to the Neural Net Spiral exercice at the bottom of the page.

Task 1: Run the model as given four or five times.
Before each trial, reset the network to get a new random initialization. Let each trial run for at least 500 steps to ensure convergence.
What shape does each model output converge to?
What does this say about the role of initialization in non-convex optimization?

Task 2: Try making the model slightly more complex by adding a layer and a couple of extra nodes.
Repeat the trials from Task 1.
Does this add any additional stability to the results?

Let's practice - some remarks

Task 1: *Run the model as given four or five times.*
Before each trial, reset the network to get a new random initialization. Let each trial run for at least 500 steps to ensure convergence.
What shape does each model output converge to?
What does this say about the role of initialization in non-convex optimization?
The learned model had different shapes on each run. The converged test loss varied almost 2X from lowest to highest.

Task 2: *Try making the model slightly more complex by adding a layer and a couple of extra nodes.*
Repeat the trials from Task 1.
Does this add any additional stability to the results?
Adding the layer and extra nodes produced more repeatable results. On each run, the resulting model looked roughly the same. Furthermore, the converged test loss showed less variance between runs.

Intermède - exercice 3: Neural Net Spiral

Visiter le site suivant (et descendre jusqu'au 2eme exercice.)
https://developers.google.com/machine-learning/crash-course/introduction-to-neural-networks/playground-exercises

The data set is a noisy spiral. Obviously, a linear model will fail here, but even manually defined feature crosses may be hard to construct.

Task 1: Train the best model you can, using just X1 and X2. Feel free to add or remove layers and neurons, change learning settings like learning rate, regularization rate, and batch size.
What is the best test loss you can get?
How smooth is the model output surface?
Task 2: Even with Neural Nets, some amount of feature engineering is often needed to achieve best performance. Try adding in additional cross product features or other transformations like sin(X1) and sin(X2).
Do you get a better model?
Is the model output surface any smoother?

Some issues in learning ANN

- The objective function usually exhibits a lot of local minima so that the initialization of the optimization task is very important. $\rightarrow$ It is typical to take random uniform weights over the range $[-0.7, +0.7]$
  One of the main features of deep learning is to find a tricky initialization of the network weights.

- Scale the inputs: it is best to standardize all inputs to have mean zero and standard deviation one.

- For highly non linear problems, it is often more efficient to build a network with several hidden layers and a low number of neurons instead of a network with only one hidden layer and a lot of neurons.

- Backpropagation: the learning of the weights of the layer which are close to the input is slow compare to the ones which are close to the output. Indeed, in the back propagation gradient are multiplied to each other ($\max \sigma' = 0.25$), it may lead to almost 0 gradients and slow update of the weights. $\rightarrow$ work with ReLU activation function.
- Overfitting

Preventing Overfitting

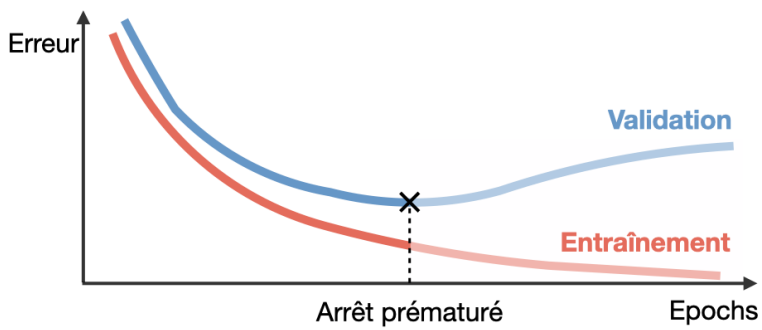
To prevent overfitting, there are several solutions that can be combined.

- A **regularization** term is added to the function we are optimizing:

$$\min_{\mathbf{w}} \underbrace{L(X, y; \mathbf{w})}_{\text{decreases with } \mathbf{w}} + \lambda \underbrace{||\mathbf{w}||_2}_{\text{increases with } \mathbf{w}}$$

to control the growth of $||\mathbf{w}||_2$. The optimization algorithm aims to assign zero weights to the variables/neurons that do not help decrease the loss function $L(X, y; \mathbf{w})$.
The constant λ is used to balance the two terms.
Example: https://playground.tensorflow.org

- Additionally, one can use the **early stopping** trick, which consists of stopping the optimization algorithm when the validation loss starts to increase.



ANN in practice

- Example with keras + google colab

`https://drive.google.com/open?id=1BItUAmMXXZYPCmoct-yu8SRFheVGjRZ_`

L'algorithme de **rétropropagation du gradient** est basé sur un **algorithme de gradient stochastique**.

A chaque itération, on tire une observation x_i au hasard et on fait une mise à jour des paramètres sachant le gradient de la fonction de perte évaluée à cette observation

$$\omega_r = \omega_{r-1} + \alpha_n \nabla_\omega J_i(\omega_{r-1})$$

avec α_n le taux d'apprentissage.

Chaque itération de l'algorithme consiste en deux étapes.
1. Une passe avant dans la quelle on calcule la valeur associée à chaque neurone pour une entrée donnée x_i et la valeur courante des paramètres.
2. Une passe arrière pendant laquelle, on commence par comparer la valeur de sortie $\hat{f}(x_i)$ à la valeur attendue y_i puis on propage l'erreur en arrière pour calculer le gradient et mettre à jour les poids.

d'où le nom de "rétropropagation du gradient".

Calcul du gradient

Exemple d'un réseau à 2 couches cachées

$$x \quad \Rightarrow \quad h_1 \quad \Rightarrow \quad h_2 \quad \Rightarrow \quad y$$

Le calcul du gradient se ramène au calcul de la dérivée de $f_\omega(x_i)$ par rapport aux poids. Or on a

$$h_1(x) = \sigma(x, W_1), \quad h_2(x) = \sigma(h_1, W_2), \quad f(x) = \sigma(h_2, W_3)$$

En notant σ_W et σ_x les gradients de la fonction (vectorielle) $\sigma(x, W) = \sigma(Wx)$ par rapport à W et x, on obtient

$$\frac{\partial f(x)}{\partial W_3} = \sigma_W(h_2, W_3)$$
$$\frac{\partial f(x)}{\partial W_2} = \sigma_h(h_2, W_3)\frac{\partial h_2(x)}{\partial W_2} = \sigma_h(h_2, W_3)\sigma_W(h_1, W_2)$$
$$\frac{\partial f(x)}{\partial W_1}\sigma_h(h_2, W_3)\sigma_h(h_1, W_2)\sigma_W(x, W_1)$$

On voit donc à chaque fois le produit des σ_h depuis la fin jusqu'au niveau postérieur au niveau concerné, multiplié ensuite par le σ_W correspondant.

En outre, on note que les fonctions de lien usuelles conduisent à des gradients simples et peu couteux à calculer.

Par exemple

$$\begin{array}{rr} \max(0,x) \to & \mathbb{I}_{x\geq 0} \\ \sigma(x) = \dfrac{1}{1-e^{-x}} \to & \sigma(x)(1-\sigma(x)) \end{array}$$

Voir aussi :
https://developers-dot-devsite-v2-prod.appspot.com/machine-learning/crash-course/backprop-scroll/

Outline