Advanced Machine Learning

Matteo Amabili
matteo.amabili@unibo.it

Course Outline

In every chapter we will have some example in python

Recommended reading

- **Goodfellow, Ian, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016**.
- **James, G., Witten, D., Hastie, T., & Tibshirani, R. (2013). *An introduction to statistical learning* (Vol. 112, p. 18). New York: springer.**
- Hastie, T., Tibshirani, R., Friedman, J. H., & Friedman, J. H. (2009). *The elements of statistical learning: data mining, inference, and prediction* (Vol. 2, pp. 1-758). New York: springer.
- Rogers, S., & Girolami, M. (2016). *A first course in machine learning*. Chapman and Hall/CRC.
- Many resources available online, stackoverflow, standford and MIT courses etc…

Chapter 1: Introduction

What is Machine Learning

$$\begin{tikzcd}
 & \textbf{Machine Learning} \arrow[dl] \arrow[dr] & \\
\text{stands for } \textbf{Algorithm} & & \text{stands for } \textbf{Optimization}
\end{tikzcd}$$

Machine Learning Foundamentals

Which Problems do ML solves ?

ML Aims to learn (approximate) a model f, that relate the features x_i to the response y

$$\boxed{y} = \boxed{f}(\boxed{x_1, \dots, x_n}; \boxed{\beta_1, \dots, \beta_n}; \boxed{\gamma_1, \dots, \gamma_m})$$

Hyper-parameter of the model

Parameter of the model

Response variable:
1. Income
2. Amount of sales
3. If a mail is spam or not

The model

Predictor:
1. Age, gender, education,…
2. Previous sale, day, holidays, …
3. Text in the mail, sender, …

Machine Learning Foundamentals

$$y = f(x_1, \dots, x_n; \beta_1, \dots, \beta_n; \gamma_1, \dots, \gamma_m)$$

How do we learn f?

The aim is to find **"the best"** model f given a **training dataset** $\boldsymbol{D} = \{ Y_i, \boldsymbol{X}_i \}$ $i = 1, \dots, n$ where n are the rows of D, $\boldsymbol{X}_i$ is the matrix of the predictor variables and Y_i is the response variable.

How to formalize the expression "the best"? → **Optimization**

We fix the shape for the model (e.g. linear model, tree based model, ecc..) and solve the following optimization problem:

$\beta, \gamma = argmin_{\beta,\gamma} L(Y, f(X; \beta; \gamma))$ usually simplified* as $\beta = argmin_{\beta} L(Y, f(X; \beta; \gamma^*))$

L is called the **Loss Function** and measures how well our model approximate the data

* Having fixed γ with other methods, e.g. cross-validation (see chapter 5)

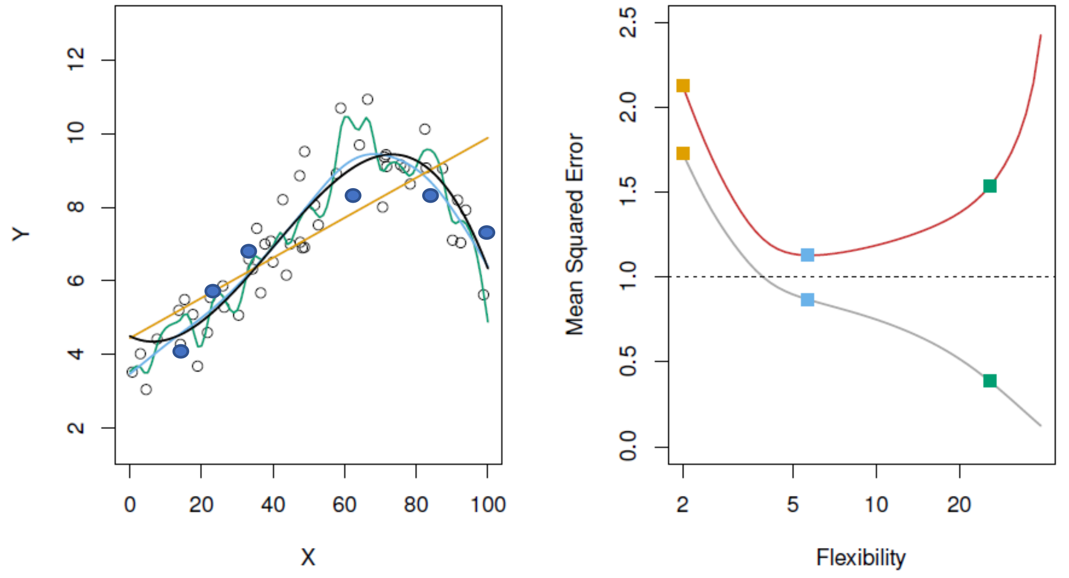
Model Generalization

- In all the real-world situation, we do not really care how the model works well on the **training data**. Rather, we are interested in the performance of the model to previously **unseen test data**:
    - Performance on unseen test data are refered as the **generalization performance** of the model.
    - A model that perform well on training, but has low perfomance on test suffers from **overfitting**
    - Trade-off between generalization & overfitting is often referred as **bias-variance trade-off.**
- Moreover, we need also a procedure to compute the **hyper-parameters of a given model**

Overfitting

Just consider two sets of data:

- Train data: used to build the model
- Test data: used to evaluate the model (must remain unseen)



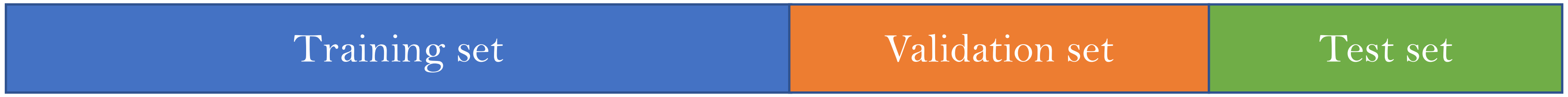
Right) From linear regression, to high order polynomial regression. Training data are the empty dots.

Left) Performance on train data (gray line) improves as the model complexity increase while, performance on test data (red line) becomes lower. This is a fundamental property of statistical learning holding regardless of the datasets and of the model being used. Model with order equal to 20 **overfits.**

Idea: Seems reasonable to choose the model with polynomial order equal to 5: it show good performance for both sets (but we are selecting the model looking at data that should be unseen: **data leak**). **We are discussing together the generalization performance and the choose of the hyper-parameter of the model.**

The simpler setup: train-test-validation splitting

If we are in a **data-rich situation**, the best approach is to randomly divide the dataset into three parts:



Used to fit the model
(70% of data)

Used to evaluate the
model performances
(15% of data)

Used to evaluate the
model generalization
(15% of data)

In practice: define a grid for all the hyper-parameter e.g., $\lambda \in \{\lambda_1, \ldots, \lambda_L\}$:

- For each value of λ, train the model on the train set
- Evaluate the performance of the model on the validation-set
- Choose the hyper-parameter λ^* with the **best** validation performance (why not optimization* ??)
- Retrain the model with λ^* and evaluate the generalizzation error on the test. **Finally compare performances between sets to adress possible overfitting!**

This setup work well in a data-rich situation

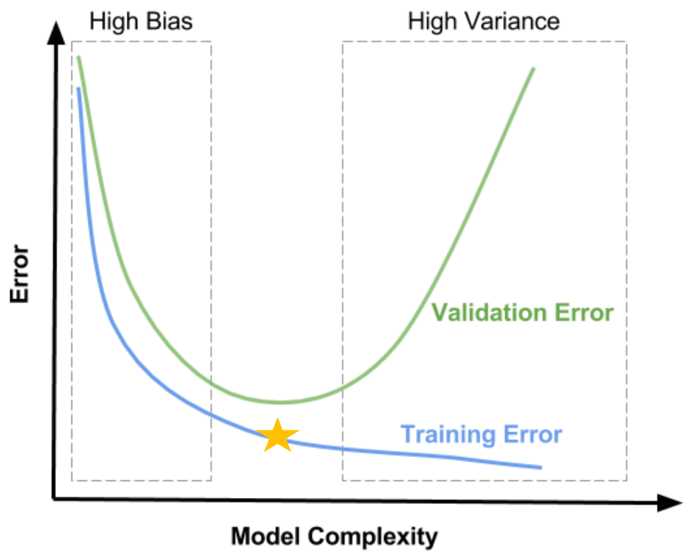
Preferred methods for training Neural Networks

* One can also use optimization methods to find the best λ^* (see e.g. Optuna python) but this method are beyond the scope of this course.

The simpler setup: train-test-validation splitting

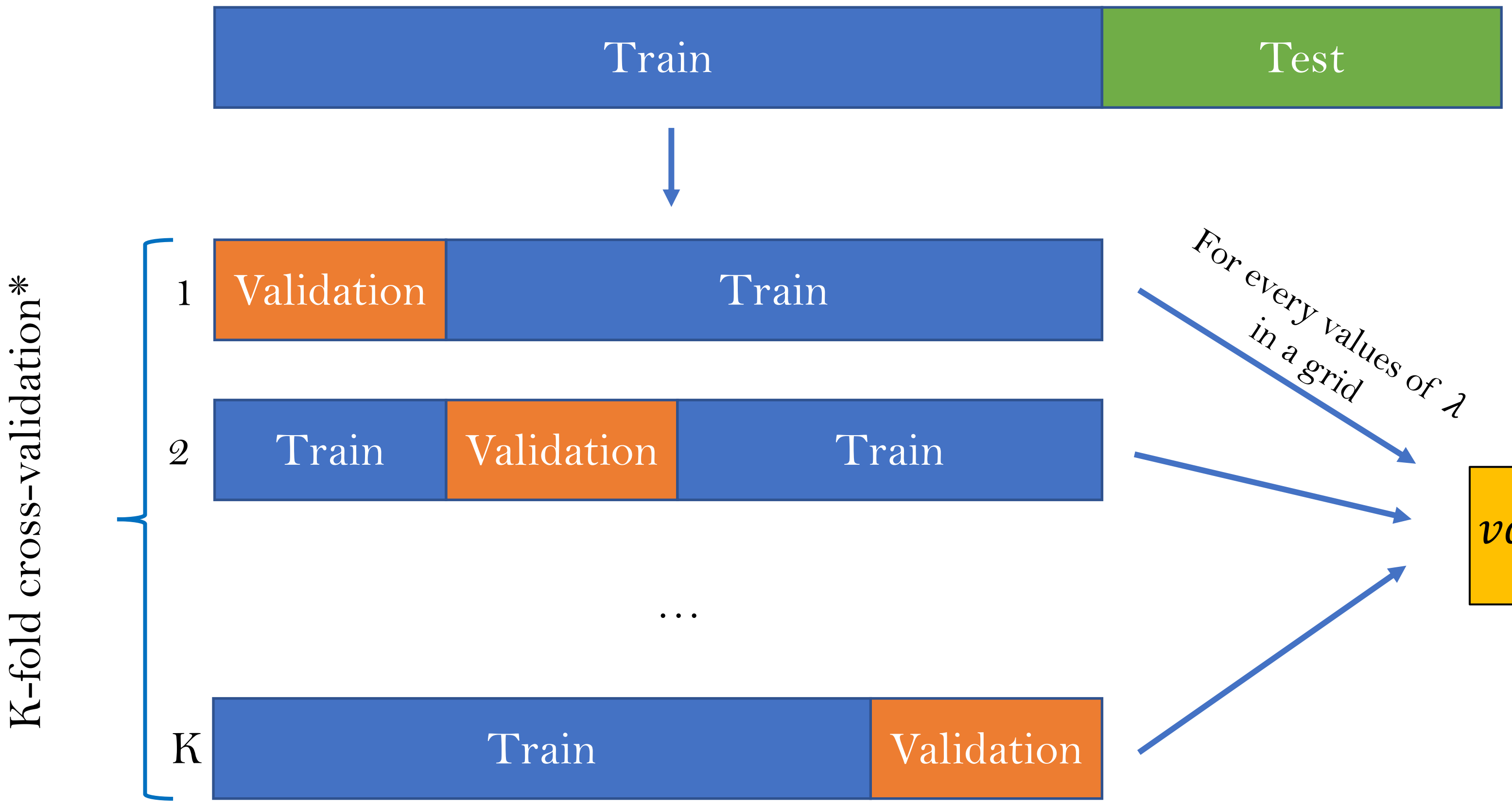
If we are in a **data-rich situation**, the best approach is to randomly divide the dataset into three parts:

Training set	Validation set	Test set
Used to fit the model (70% of data)	Used to evaluate the **model performances** (15% of data)	Used to evaluate the **model generalization** (15% of data)



★ Best model

Cross-validation: less data-rich situation



$$val_{error} = \frac{1}{K}\sum_k val_{error}^k$$

Compute avarege of performances on the validation sets

Choose the hyperparameter λ^* with the **best** cross-validation error!
This approach is computationally expensive, and is **typically not used with Neural Networks**

Machine Learning Foundamentals

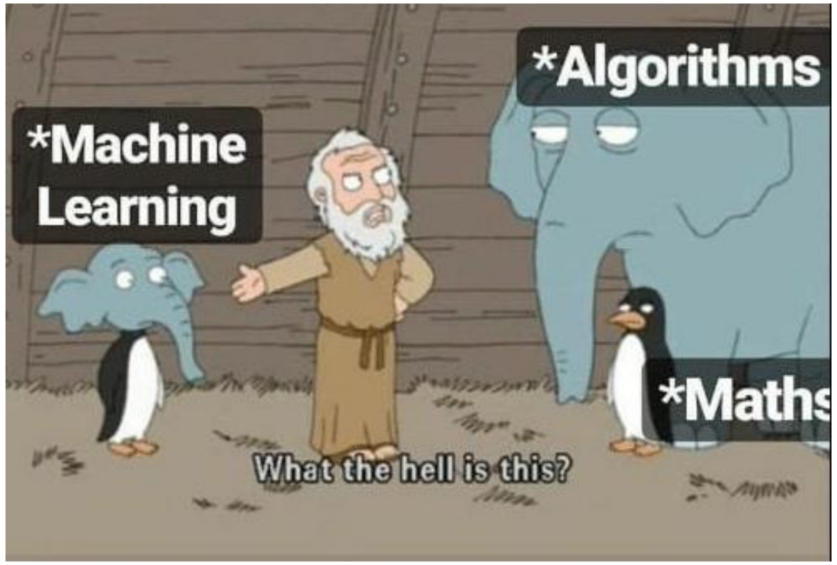
$$y = f(x_1, \dots, x_n; \beta_1, \dots, \beta_n; \gamma_1, \dots, \gamma_m)$$

How do we learn f?

$$\beta = argmin_\beta \; L(Y, f(X; \beta; \gamma^*)) \quad (1)$$

Recipe:

- Define a shape for f
- Define suitable loss function L for the problem at hand
- Define a robust statistical framework to evaluate the model
- Solve the minimization problem in (1)
- Understand the result in term of business insight



Type of Problem

$$y = f(x_1, \dots, x_n)$$

Supervised

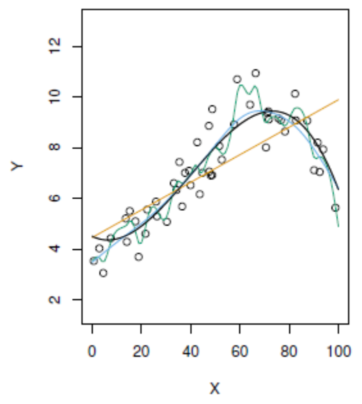
We observe both x_i, y → fit model that relate predictor to response

Unsupervised

We observe only x_i → understand the relationships between the variables

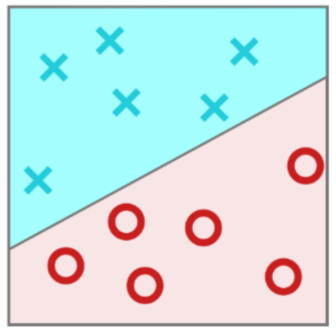
Regression

$y \in R$

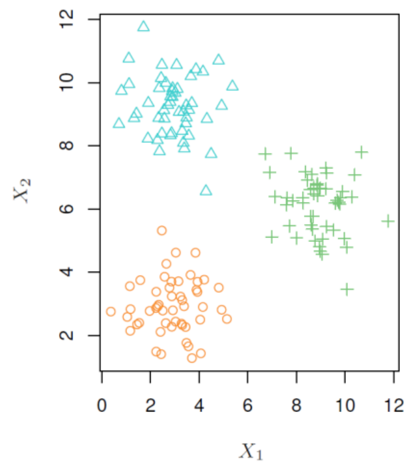


Classification

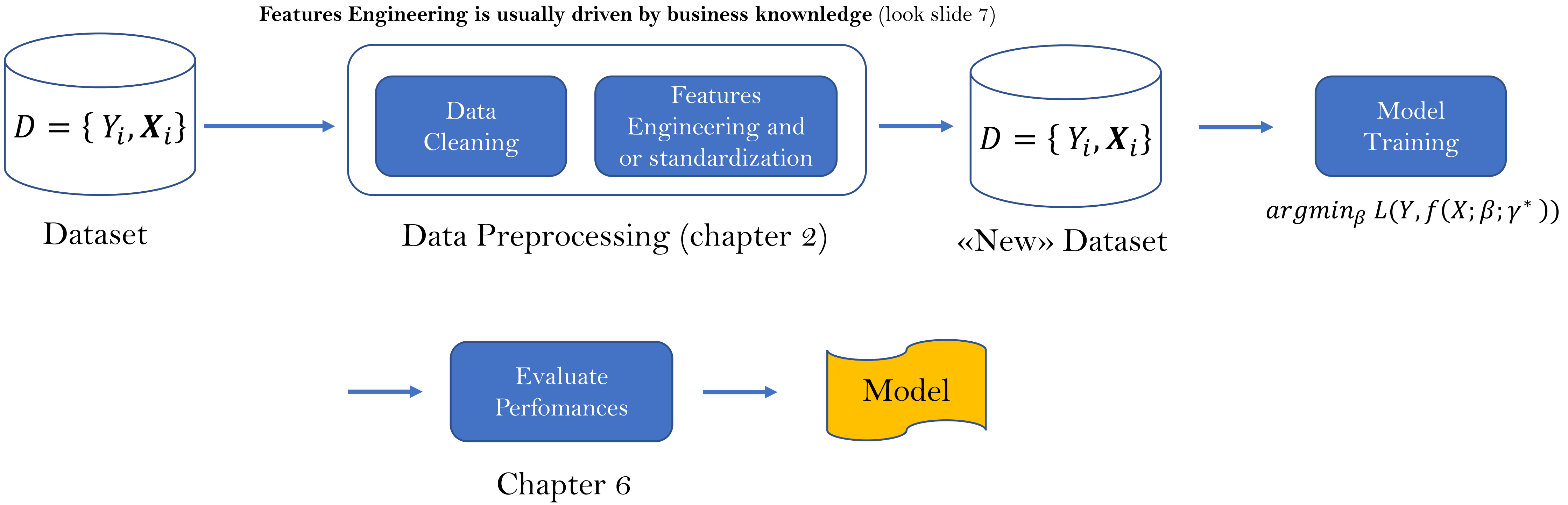
$y \in \{0,1,\dots,m\}$



Clustering Methods



Machine Learning Pipeline


Features Engineering is usually driven by business knowledge (look slide 7)
$D = \{ Y_i, \boldsymbol{X}_i \}$
Dataset
Data Cleaning
Features Engineering and or standardization
Data Preprocessing (chapter 2)
$D = \{ Y_i, \boldsymbol{X}_i \}$
«New» Dataset
Model Training
$argmin_\beta \, L(Y, f(X; \beta; \gamma^*))$
Evaluate Perfomances
Chapter 6
Model

Chapter 1: Neural Network

Machine Learning Foundamentals

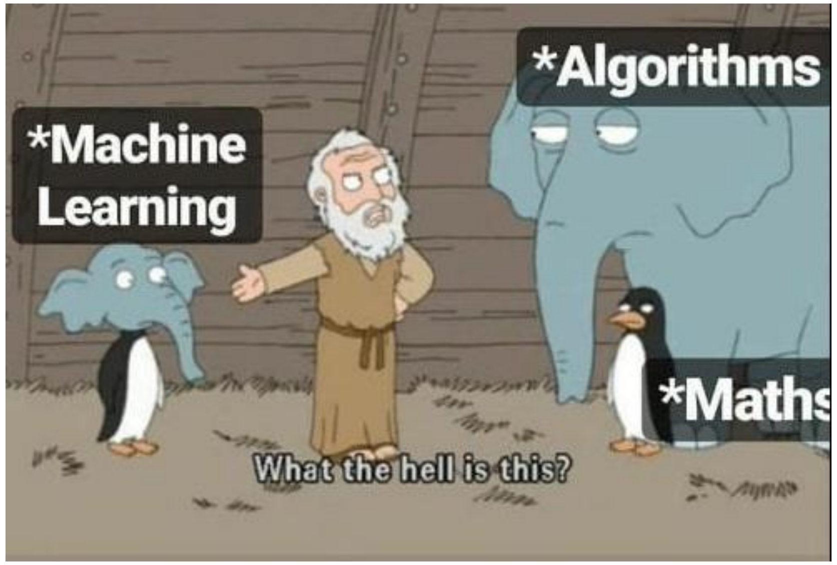
$$y = f(x_1, \dots, x_n; \beta_1, \dots, \beta_n; \gamma_1, \dots, \gamma_m)$$

How do we learn f?

$$\beta = \ argmin_\beta \ L(Y, f(X; \beta; \gamma^*)) \quad (1)$$

Recipe:

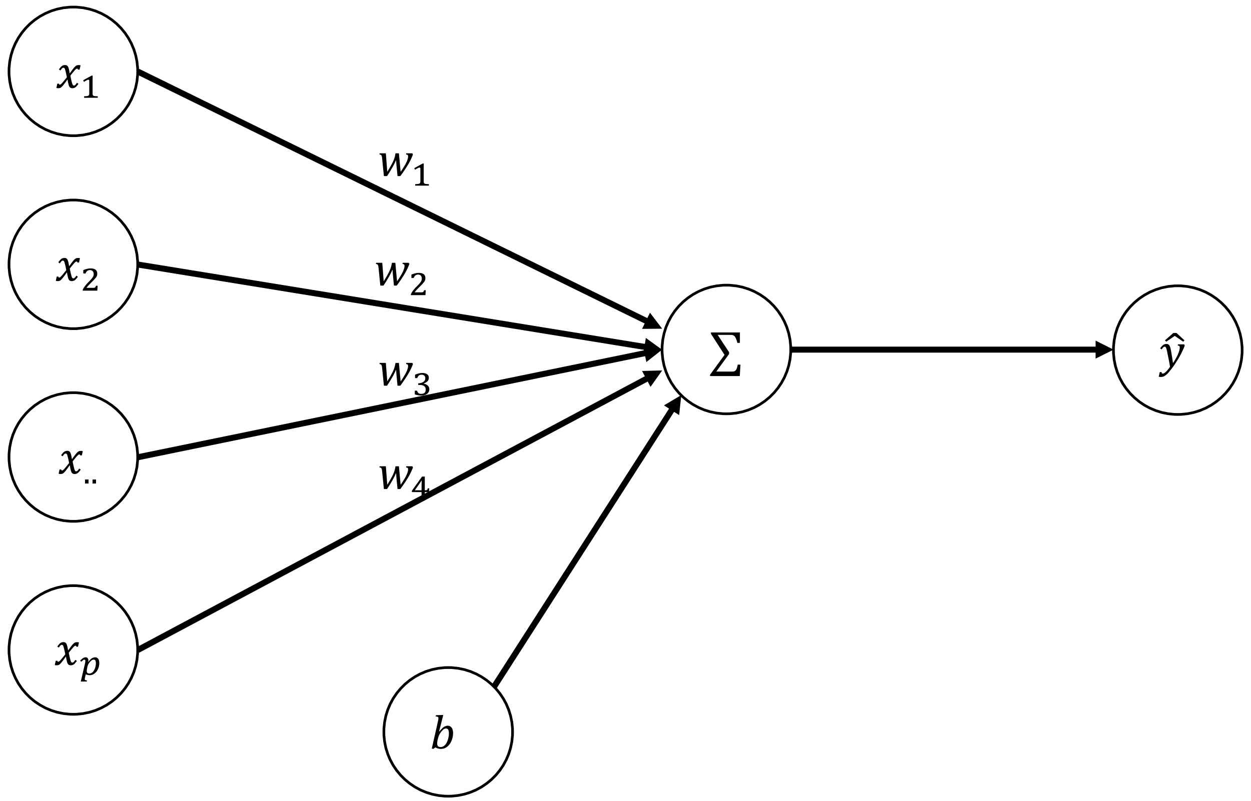
- **Define a shape for *f***
- Define suitable loss function L for the problem at hand
- Define a robust statistical framework to evaluate the model
- Solve the minimization problem in (1)
- Understand the result in term of business insight



Preliminary: Linear Model

Let us start with a linear model and try to represent it via a **computational graph**

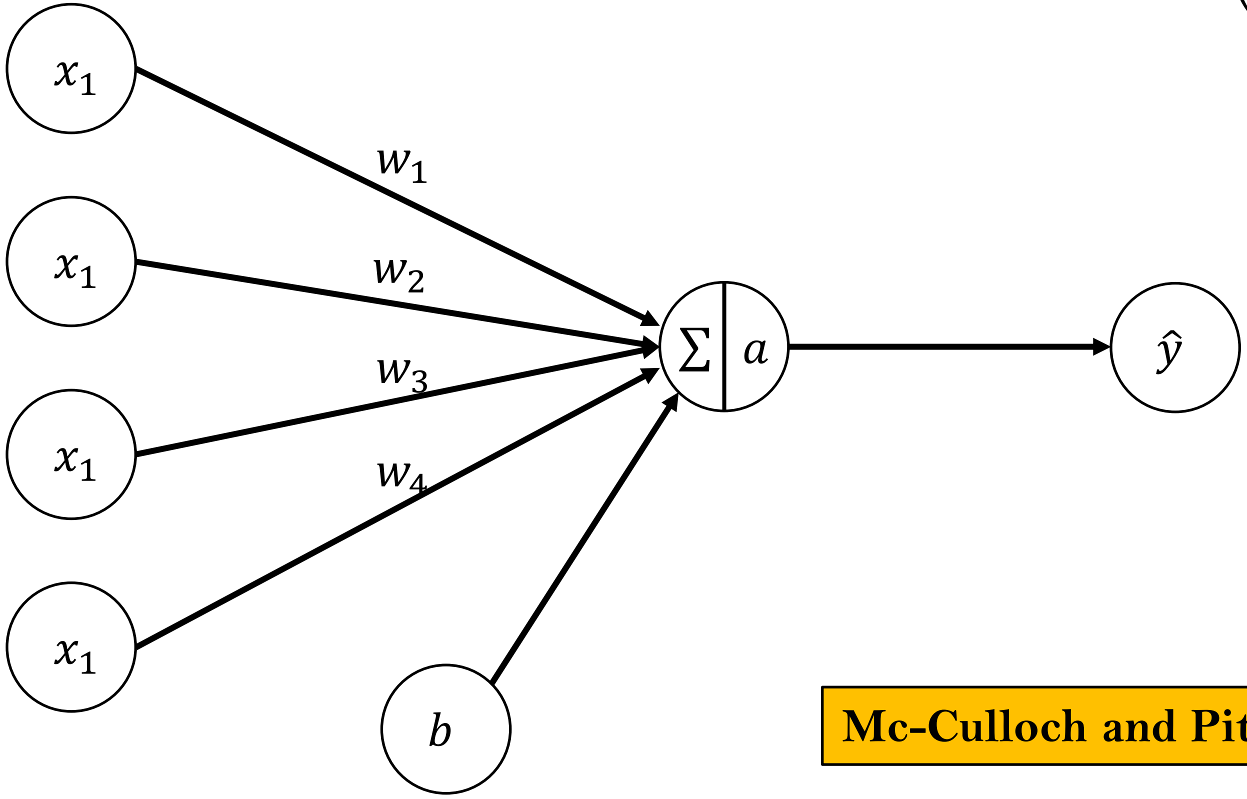
$$\hat{y} = f(x) = b + w_1 x_1 + \ldots + w_p x_p = b + \sum_{i=1}^{p} w_i x_i$$



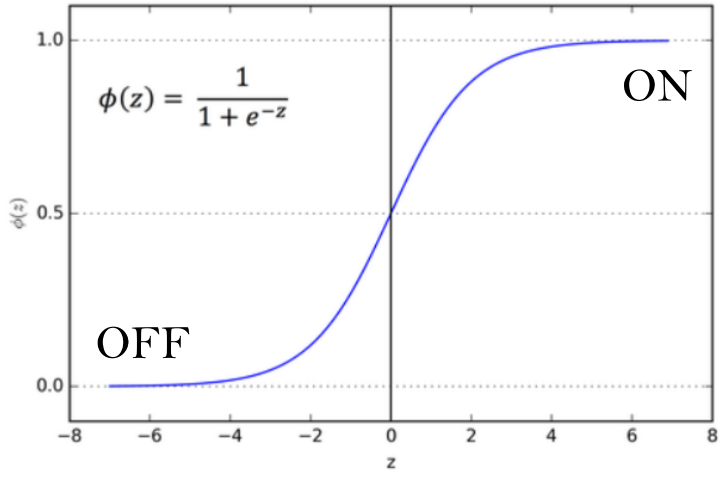
Preliminary: Adding nonlinearity

Activation function

$$\hat{y} = a(b + w_1 x_1 + \ldots + w_p x_p) = a\left(b + \sum_{i=1}^{p} w_i x_i \right) = a(z)$$



Activation function

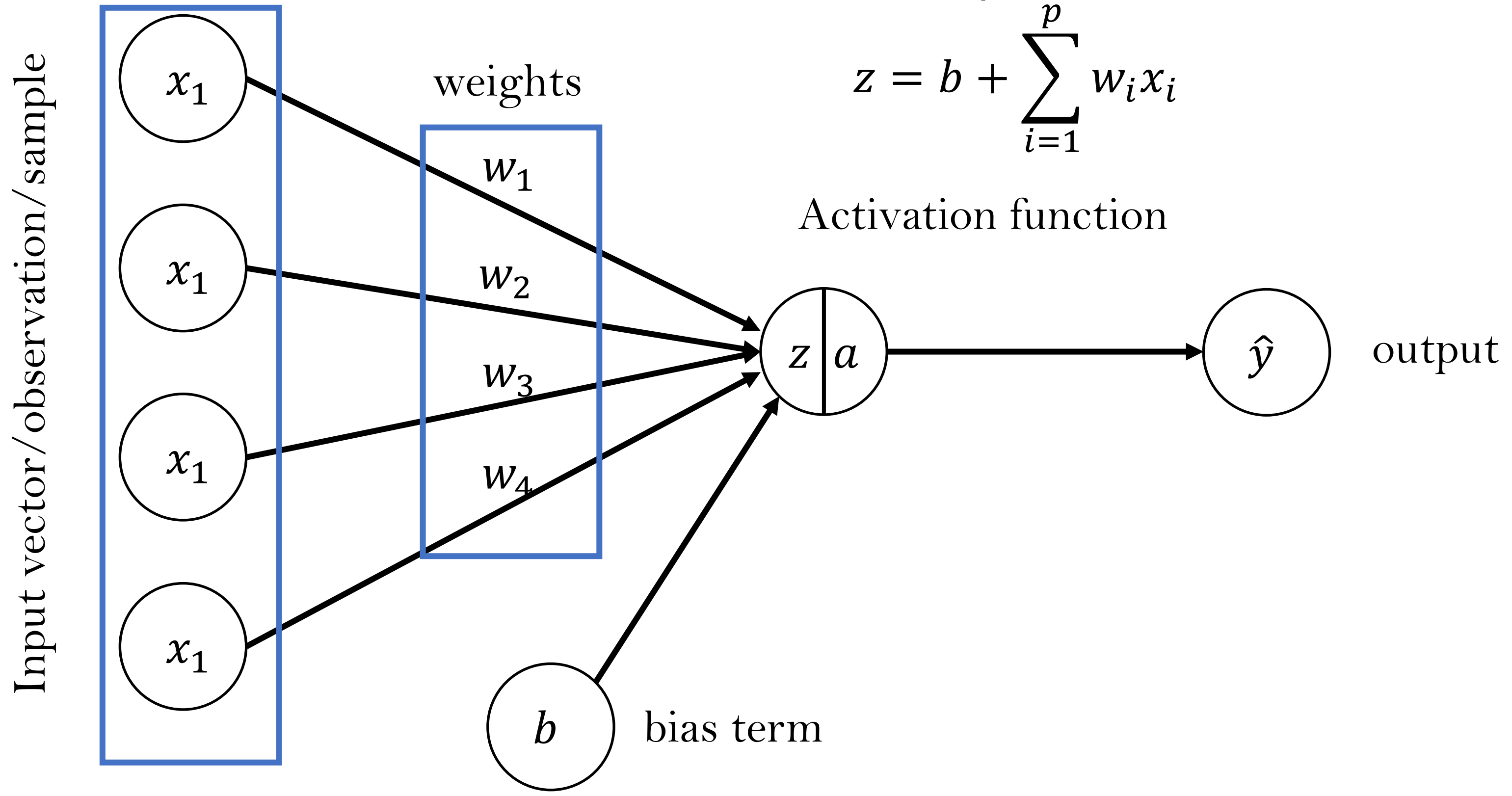


Sigmoid function → Logistic Regression

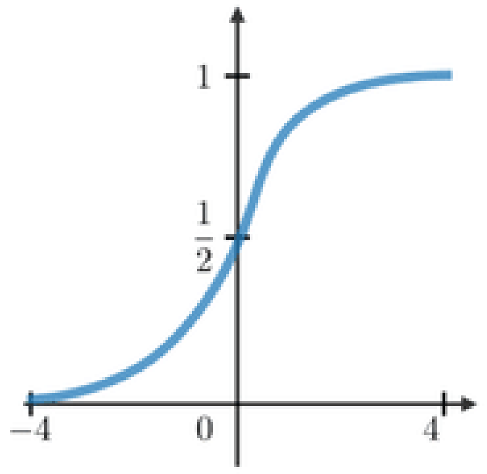
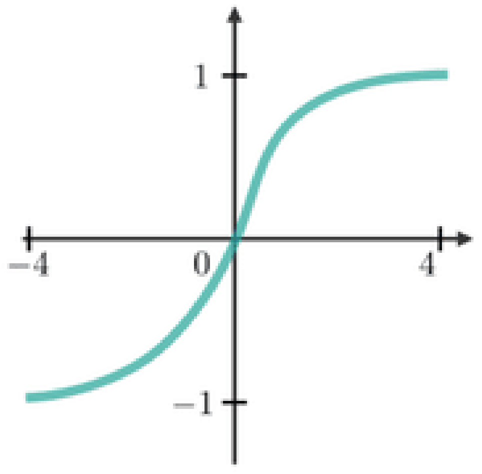
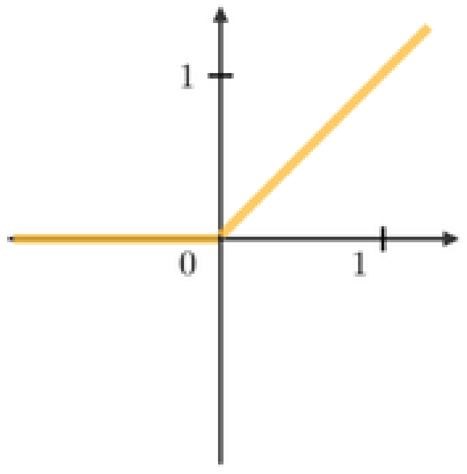
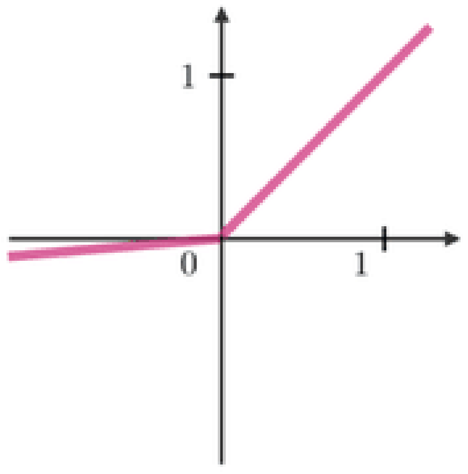
Mc-Culloch and Pitts Neuron

Preliminary: naming convention

$$\hat{y} = a(b + w_1 x_1 + \ldots + w_p x_p) = a\left(\underbrace{b + \sum_{i=1}^{p} w_i x_i}_{z = b + \sum_{i=1}^{p} w_i x_i} \right) = a(z)$$



Activation function

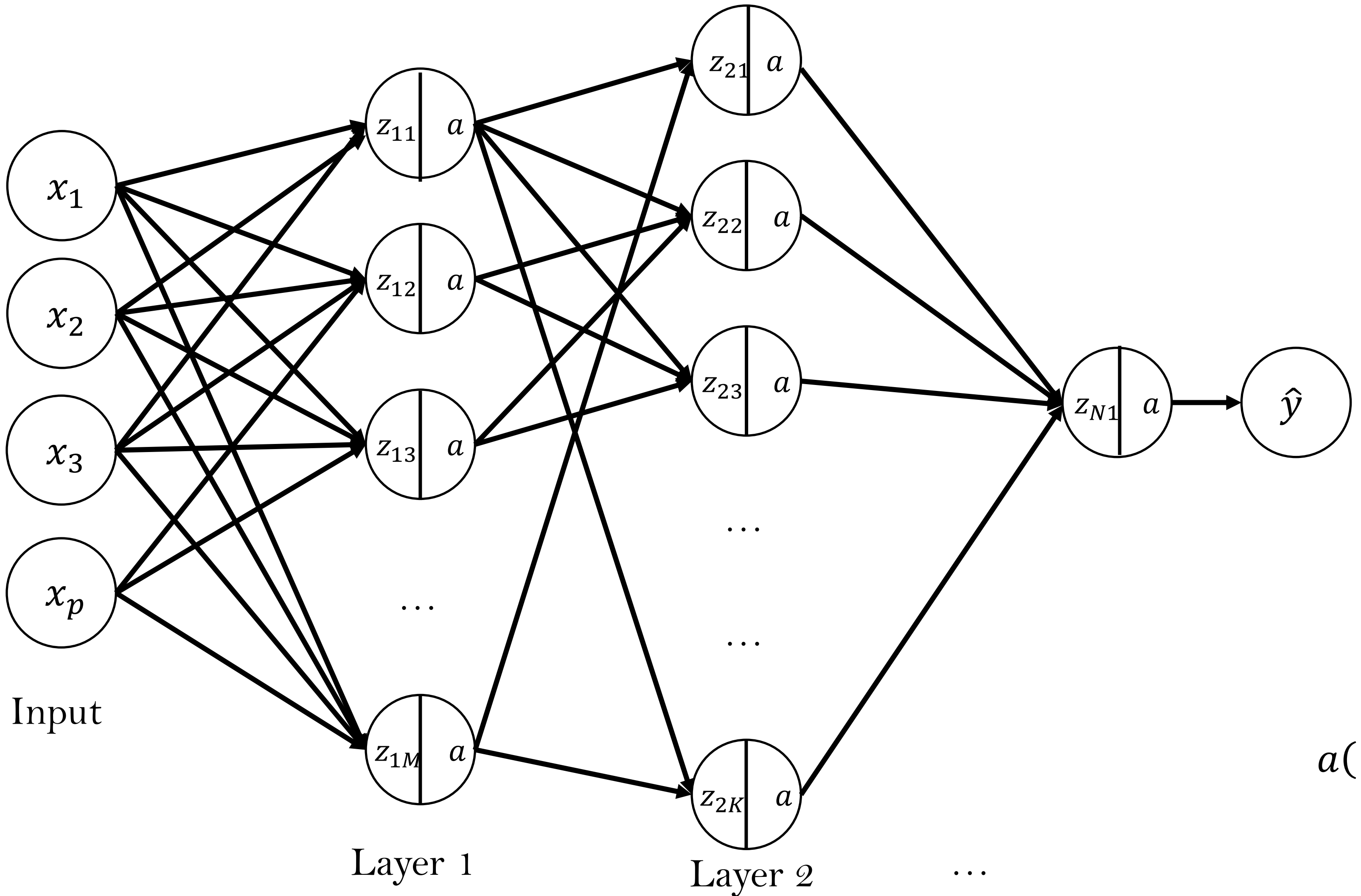
Sigmoid	Tanh	ReLU	Leaky ReLU
$g(z) = \dfrac{1}{1+e^{-z}}$	$g(z) = \dfrac{e^z - e^{-z}}{e^z + e^{-z}}$	$g(z) = \max(0, z)$	$g(z) = \max(\epsilon z, z)$ with $\epsilon \ll 1$
			

Most used

Similar to ReLU but introduced to avoid *vanishing gradient*

Neural Network

A neural network is a stack of nodes connected by links and activation function



Math shape of a NN

$$X \in (p, 1)$$

$$Z_1 = W_1 X \quad W_1 \in (M, p)$$

$$Z_2 = W_2\, a(Z_1) \quad W_2 \in (K, M)$$

$$Z_3 = W_3\, a(Z_2) \quad W_3 \in (1, K)$$

$$\hat{y} = a(Z_3)$$

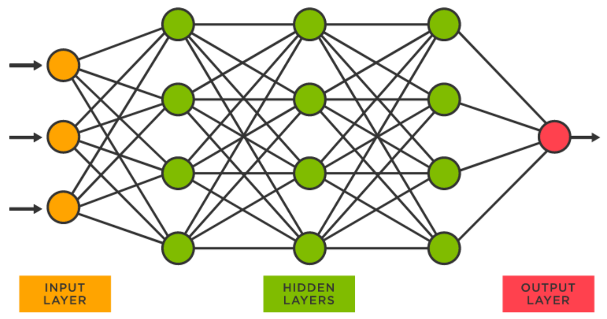
$$\hat{y} = a(W_3 a(W_2 a(W_1 X)))$$

Note that:

$$a([z_1, z_2, \dots, z_M]) = [a(z_1), a(z_2), \dots, a(z_M)]$$

Neural Network: training

Consider a Dataset $D = \{x_i, y\}$ with p features and n rows **(training dataset)**



$$\hat{y} = NN(\boldsymbol{x}; W) \qquad where\ W = [W_1, W_2, \ldots]$$

$$\boxed{W = argmin_W\ L(Y, NN(x, W))}$$

Regression

$$L = \frac{1}{n}\sum_{i=0}^{n}(y_i - NN(x_i, W))^2$$

The activation of the last layer is the identity (or ReLU if y>0, eg price)

Classification

$$L = \sum_{i=0}^{n}(y_i \ln NN(x_i, W) + (1 - y_i)\ln(1 - NN(x_i, W))$$

the activation of the last layer is the sigmoid function

MultiClass

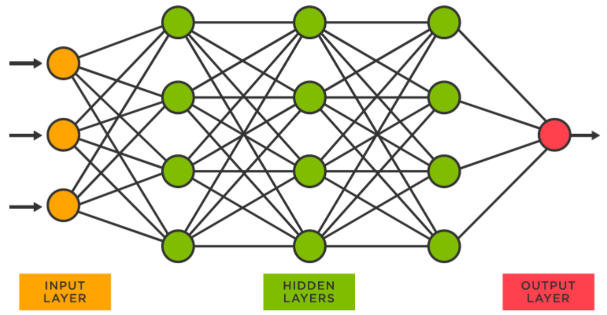
$$L = \sum_{k=0}^{K}\sum_{i=0}^{n} 1_{y_i \in k} \ln(NN(y_i \in k))$$

the activation of the last layer is the softmax function and has K neurons

The architecture of the neural networks (i.e. how many hidden layer and how many neuron in each layer) is an hyper-parameter of the model and is typically chosen by trial & error (or more sophysticated methods)

Neural Network: training

Consider a Dataset $D = \{x_i, y\}$ with p features and n rows **(training dataset)**



$$\hat{y} = NN(\boldsymbol{x}; W) \qquad where\ W = [W_1, W_2, \dots]$$

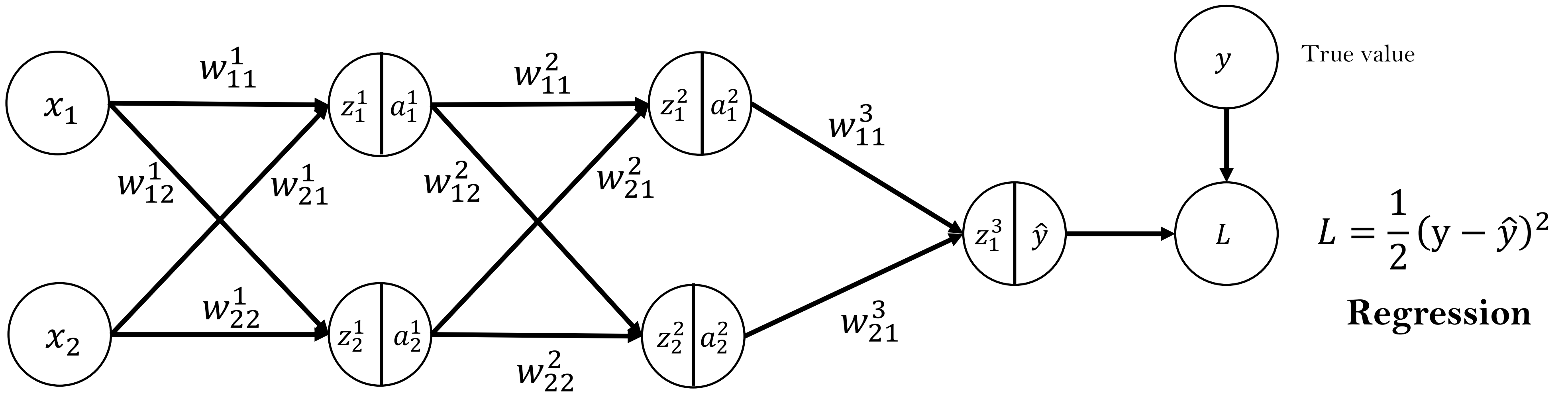
$$\boxed{W = argmin_W\ L(Y, NN(x, W))}$$

The widely used algorithm for training a neural network is **backpropagation,** that is just a funny name for:

- **Computing derivatives** of L with respect to the weights W using the **chain rules of derivatives**
- Apply a **gradient descent** method (or its variations) to update W

Compute derivatives

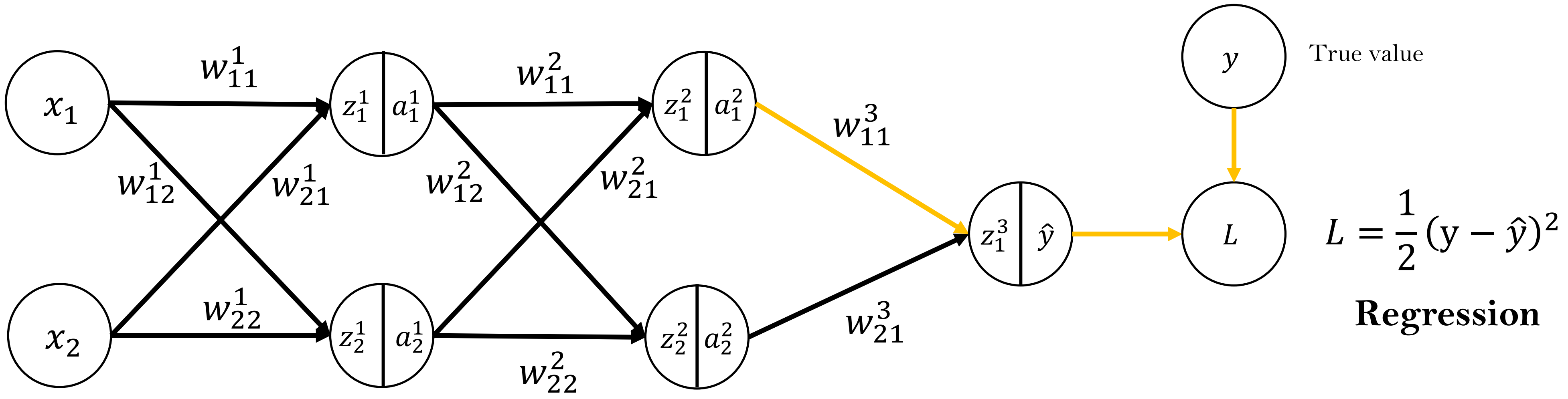
Let us consider a very simple computational graph (then we will try to generalize the approach):



- w_{ij}^k : weight for node *i* in layer k for incoming node j
- z_i^k: product sum for node I in layer k
- a_i^k: activation for node i in layer k

Compute derivatives

Let us consider a very simple computational graph (then we will try to generalize the approach):



$$L = \frac{1}{2}(y - \hat{y})^2$$

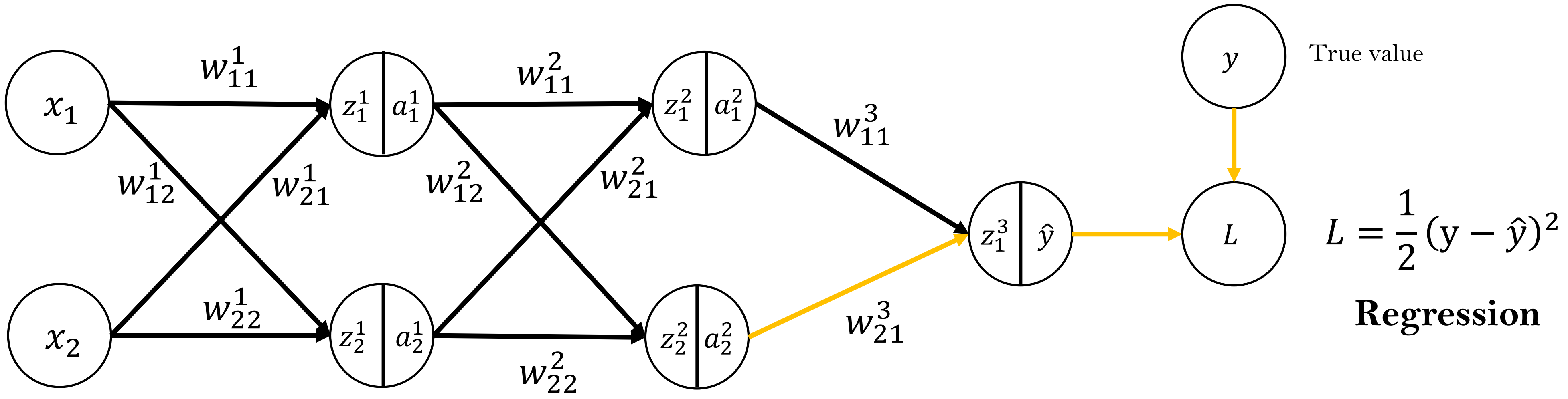
Regression

$$\hat{y} = a(z_1^3) \qquad z_1^3 = w_{11}^3\, a_1^2 + w_{21}^3\, a_2^2$$

$$\frac{dL}{dw_{11}^3} = \frac{dL}{d\hat{y}}\frac{d\hat{y}}{dz_1^3}\frac{dz_1^3}{dw_{11}^3} = (y - \hat{y})a'(z_1^3)a_1^2 = \delta^3 a_1^2$$

Compute derivatives

Let us consider a very simple computational graph (then we will try to generalize the approach):

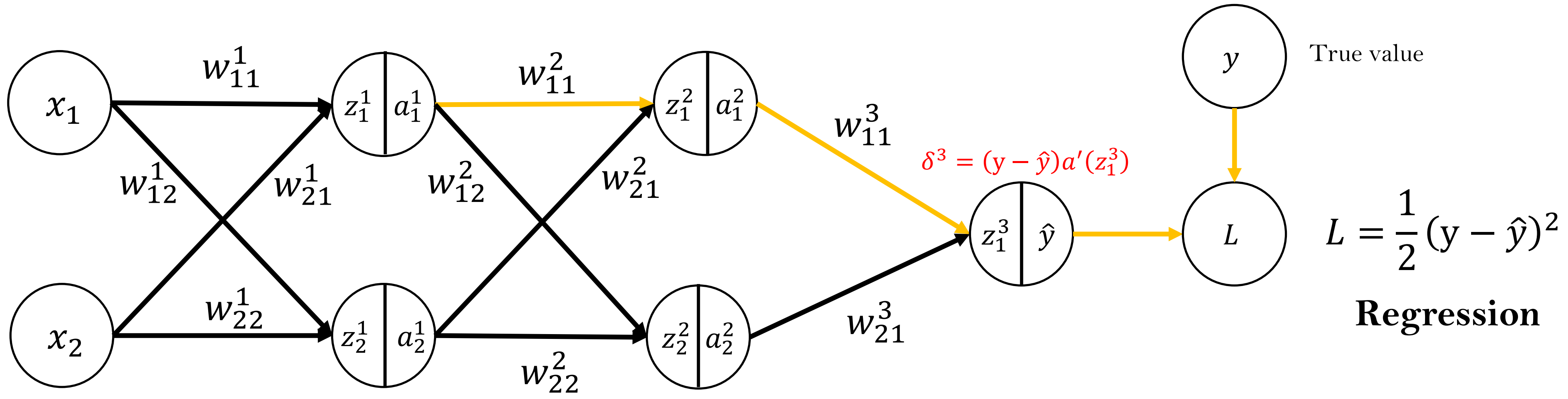


$$\hat{y} = a(z_1^3) \qquad z_1^3 = w_{11}^3\, a_1^2 + w_{21}^3\, a_2^2$$

$$\frac{dL}{dw_{21}^3} = \frac{dL}{d\hat{y}}\frac{d\hat{y}}{dz_1^3}\frac{dz_1^3}{dw_{21}^3} = (y - \hat{y})a'(z_1^3)a_2^2 = \delta^3 a_2^2$$

Compute derivatives

Let us consider a very simple computational graph (then we will try to generalize the approach):



$$L = \frac{1}{2}(y - \hat{y})^2$$

Regression

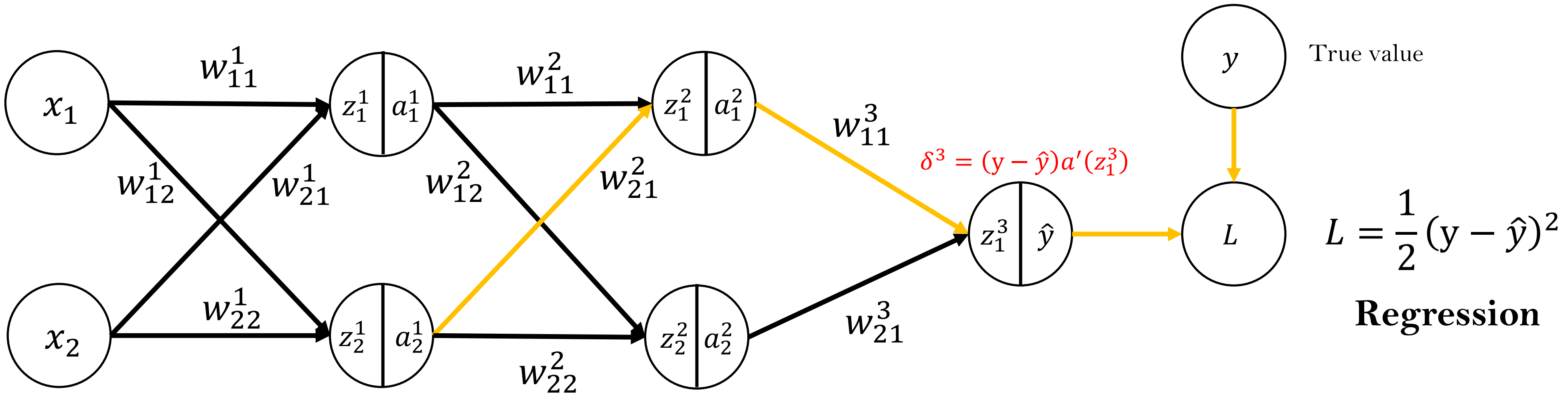
$$\hat{y} = a(z^3_1) \qquad z^3_1 = w^3_{11}\, a^2_1 + w^3_{21}\, a^2_2 \qquad a^2_1 = a(z^2_1)$$

$$\frac{dL}{dw^2_{11}} = \frac{dL}{d\hat{y}}\frac{d\hat{y}}{dz^3_1}\frac{dz^3_1}{da^2_1}\frac{da^2_1}{dz^2_1}\frac{dz^2_1}{dw^2_{11}} = (y - \hat{y})a'(z^3_1)\, w^3_{11}a'(z^2_1)a^1_1 = \delta^2_1 a^1_1$$

$$\delta^2_1 = \delta^3 w^3_{11} a'(z^2_1)$$

Compute derivatives

Let us consider a very simple computational graph (then we will try to generalize the approach):



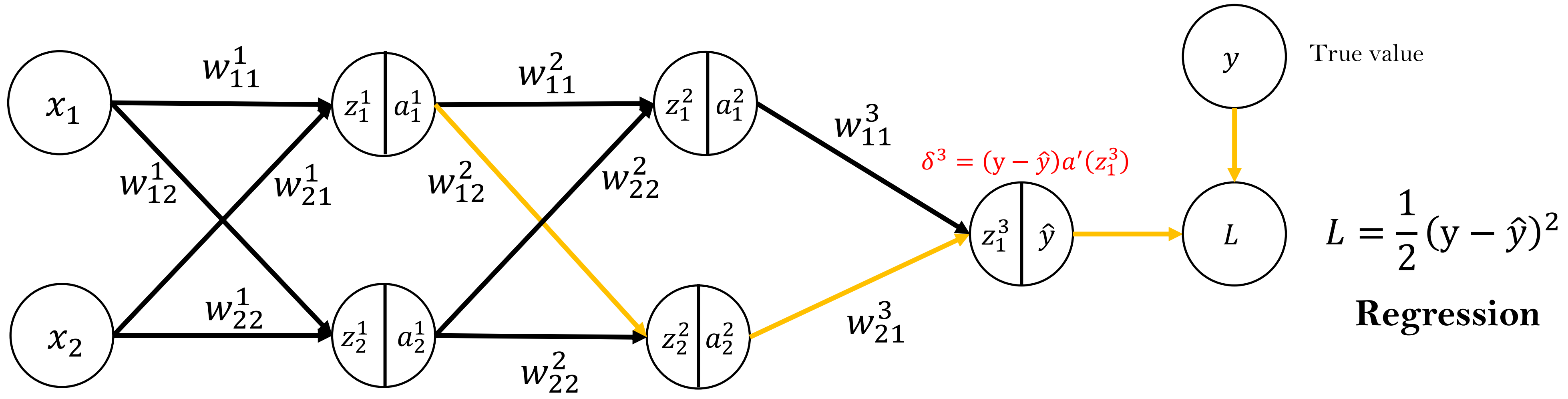
$$\hat{y} = a(z^3_1) \qquad z^3_1 = w^3_{11}\, a^2_1 + w^3_{21}\, a^2_2 \qquad a^2_1 = a(z^2_1)$$

$$\frac{dL}{dw^2_{21}} = \frac{dL}{d\hat{y}}\frac{d\hat{y}}{dz^3_1}\frac{dz^3_1}{da^2_1}\frac{da^2_1}{dz^2_1}\frac{dz^2_1}{dw^2_{21}} = (y - \hat{y})a'(z^3_1)\, w^3_{11}a'(z^2_1)a^1_1 = \delta^2_1 a^1_2$$

$$\delta^2_1 = \delta^3 w^3_{11} a'(z^2_1)$$

Compute derivatives

Let us consider a very simple computational graph (then we will try to generalize the approach):



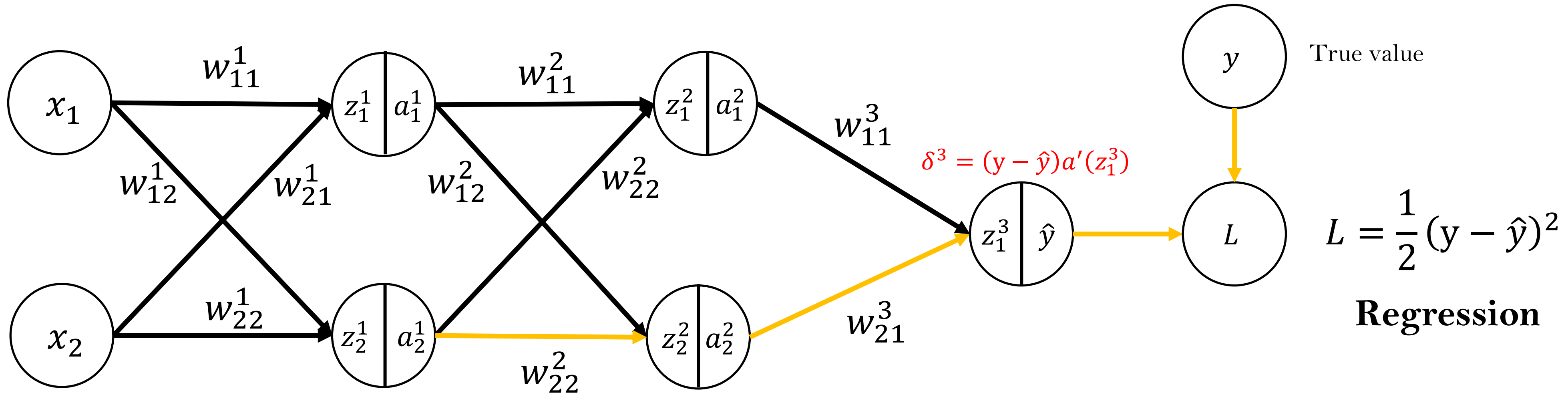
$$\hat{y} = a(z^3_1) \qquad z^3_1 = w^3_{11}\, a^2_1 + w^3_{21}\, a^2_2 \qquad a^2_1 = a(z^2_1)$$

$$\frac{dL}{dw^2_{12}} = \frac{dL}{d\hat{y}}\frac{d\hat{y}}{dz^3_1}\frac{dz^3_1}{da^2_2}\frac{da^2_2}{dz^2_2}\frac{dz^2_2}{dw^2_{12}} = (y-\hat{y})a'(z^3_1)\, w^3_{21}a'(z^2_2)a^1_1 = \delta^2_2 a^1_1$$

$$\delta^2_2 = \delta^3 w^3_{21} a'(z^2_2)$$

Compute derivatives

Let us consider a very simple computational graph (then we will try to generalize the approach):



$$L = \frac{1}{2}(y - \hat{y})^2$$

Regression

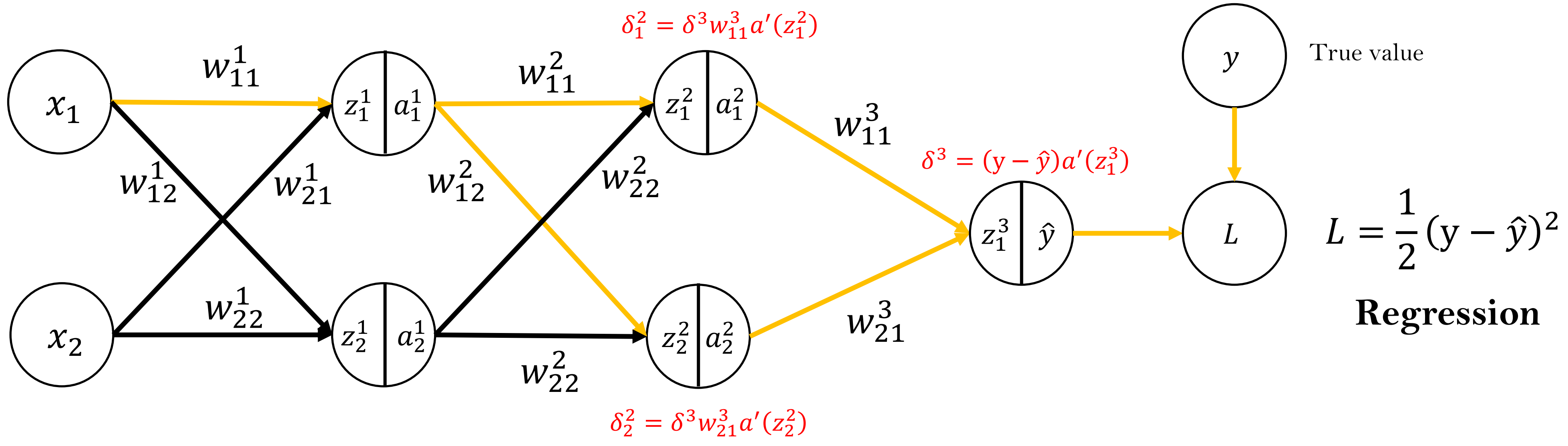
$$\hat{y} = a(z^3_1) \qquad z^3_1 = w^3_{11}\, a^2_1 + w^3_{21}\, a^2_2 \qquad a^2_1 = a(z^2_1)$$

$$\frac{dL}{dw^2_{22}} = \frac{dL}{d\hat{y}}\frac{d\hat{y}}{dz^3_1}\frac{dz^3_1}{da^2_2}\frac{da^2_2}{dz^2_2}\frac{dz^2_2}{dw^2_{22}} = (y - \hat{y})a'(z^3_1)\, w^3_{21}a'(z^2_2)a^1_2 = \delta^2_2 a^1_2$$

$$\delta^2_2 = \delta^3 w^3_{21} a'(z^2_2)$$

Compute derivatives

Let us consider a very simple computational graph (then we will try to generalize the approach):



$$\frac{dL}{dw_{11}^1} = \frac{dL}{d\hat{y}}\frac{d\hat{y}}{dz_1^3}\frac{dz_1^3}{da_1^2}\frac{da_1^2}{dz_1^2}\frac{dz_1^2}{da_1^1}\frac{da_1^1}{dz_1^1}\frac{dz_1^1}{dw_{11}^1} + \frac{dL}{d\hat{y}}\frac{d\hat{y}}{dz_1^3}\frac{dz_1^3}{da_2^2}\frac{da_2^2}{dz_2^2}\frac{dz_2^2}{da_1^1}\frac{da_1^1}{dz_1^1}\frac{dz_1^1}{dw_{11}^1}$$

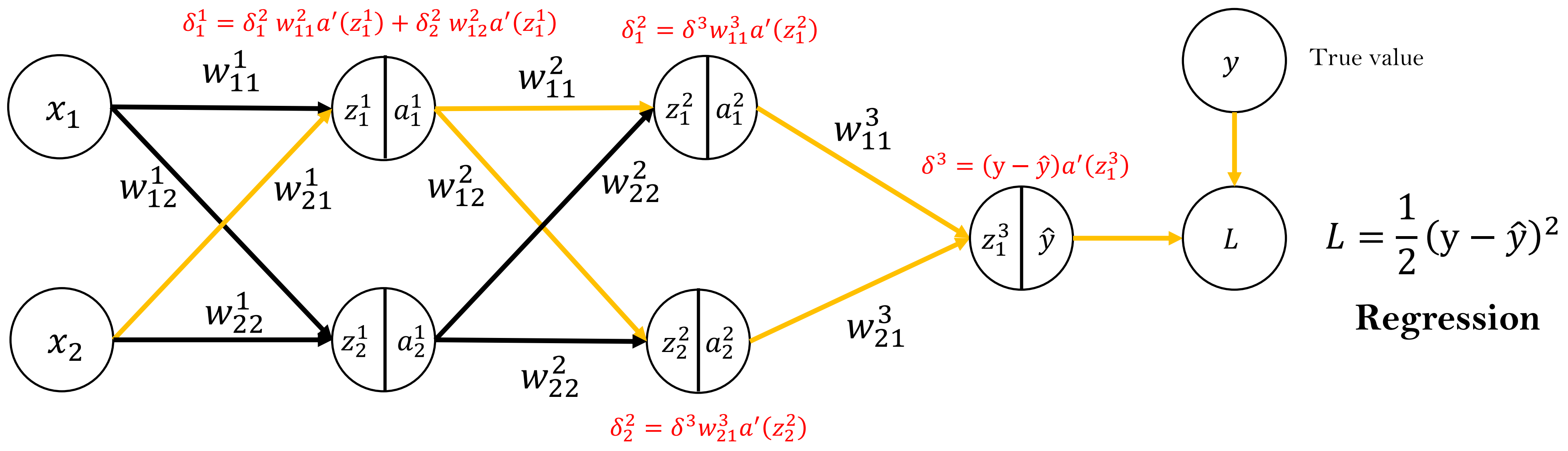
$$= (y - \hat{y})a'(z_1^3)\, w_{11}^3 a'(z_1^2) w_{11}^2 a'(z_1^1) x_1 + (y - \hat{y})a'(z_1^3) w_{21}^3 a'(z_2^2) w_{12}^2 a'(z_1^1) x_1$$

$$= (\delta_1^2\, w_{11}^2 a'(z_1^1) + \delta_2^2\, w_{12}^2 a'(z_1^1))x_1 = \delta_1^1 x_1$$

$$\delta_1^1 = \delta_1^2\, w_{11}^2 a'(z_1^1) + \delta_2^2\, w_{12}^2 a'(z_1^1)$$

Compute derivatives

Let us consider a very simple computational graph (then we will try to generalize the approach):


$\delta_1^1 = \delta_1^2 w_{11}^2 a'(z_1^1) + \delta_2^2 w_{12}^2 a'(z_1^1)$
$\delta_1^2 = \delta^3 w_{11}^3 a'(z_1^2)$
$\delta^3 = (y - \hat{y}) a'(z_1^3)$
$\delta_2^2 = \delta^3 w_{21}^3 a'(z_2^2)$
x_1
x_2
w_{11}^1
w_{12}^1
w_{21}^1
w_{22}^1
z_1^1 a_1^1
z_2^1 a_2^1
w_{11}^2
w_{12}^2
w_{22}^2
w_{22}^2
z_1^2 a_1^2
z_2^2 a_2^2
w_{11}^3
w_{21}^3
z_1^3 $\hat{y}$
y
True value
L
$L = \frac{1}{2}(y - \hat{y})^2$
Regression

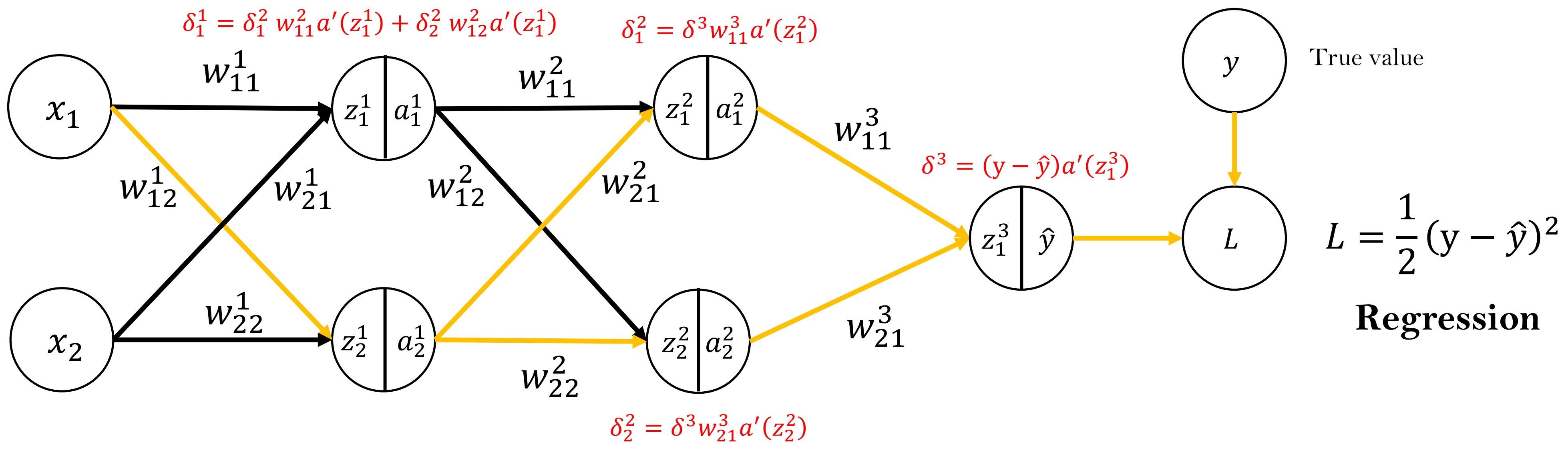
$$\frac{dL}{dw_{21}^1} = \frac{dL}{d\hat{y}}\frac{d\hat{y}}{dz_1^3}\frac{dz_1^3}{da_1^2}\frac{da_1^2}{dz_1^2}\frac{dz_1^2}{da_1^1}\frac{da_1^1}{dz_1^1}\frac{dz_1^1}{dw_{21}^1} + \frac{dL}{d\hat{y}}\frac{d\hat{y}}{dz_1^3}\frac{dz_1^3}{da_2^2}\frac{da_2^2}{dz_2^2}\frac{dz_2^2}{da_1^1}\frac{da_1^1}{dz_1^1}\frac{dz_1^1}{dw_{21}^1}$$

$$= (y - \hat{y})a'(z_1^3)\, w_{11}^3 a'(z_1^2) w_{11}^2 a'(z_1^1) x_2 + (y - \hat{y})a'(z_1^3) w_{21}^3 a'(z_2^2) w_{12}^2 a'(z_1^1) x_2$$

$$= (\delta_1^2\, w_{11}^2 a'(z_1^1) + \delta_2^2\, w_{12}^2 a'(z_1^1)) x_1 = \delta_1^1 x_2$$

$$\delta_1^1 = \delta_1^2\, w_{11}^2 a'(z_1^1) + \delta_2^2\, w_{12}^2 a'(z_1^1)$$

Compute derivatives



$\delta_1^1 = \delta_1^2\, w_{11}^2 a'(z_1^1) + \delta_2^2\, w_{12}^2 a'(z_1^1)$
$\delta_1^2 = \delta^3 w_{11}^3 a'(z_1^2)$
$\delta^3 = (y - \hat{y}) a'(z_1^3)$
$\delta_2^2 = \delta^3 w_{21}^3 a'(z_2^2)$
x_1
x_2
w_{11}^1
w_{12}^1
w_{21}^1
w_{22}^1
z_1^1 a_1^1
z_2^1 a_2^1
w_{11}^2
w_{12}^2
w_{21}^2
w_{22}^2
z_1^2 a_1^2
z_2^2 a_2^2
w_{11}^3
w_{21}^3
z_1^3 $\hat{y}$
y
True value
L
$L = \frac{1}{2}(y - \hat{y})^2$
Regression

$$\frac{dL}{dw_{ij}^k} = \delta_j^k a_i^{k-1}$$

For the final layer (M)

$$\delta_1^M = a'(z_1^M)(y - \hat{y})$$

$$\delta_j^k = a'(z_j^k) \sum_{l=1}^{r^{k+1}} w_{jl}^{k+1} \delta_l^{k+1}$$

To compute derivatives we need the values of activation/output in each layer, i.e $a'(z_j^k)$. The backpropagation algorithm is a two step procedure:

- **Forward pass** (from first to last layer) to compute activation/output
  - **Backward pass** (from last to first layer) to compute δ_j^k and $\frac{dL}{dw_{ij}^k}$
    - *The final gradient is the* ***average*** *all over the traning data*

Apply the Gradient

The gradient are applied using a gradient descente approach:

$$w_{ij} = random\ init$$
$$w_{ij}(t) = w_{ij}(t-1) - \gamma \frac{dL}{dw_{ij}}$$

t is the iteration

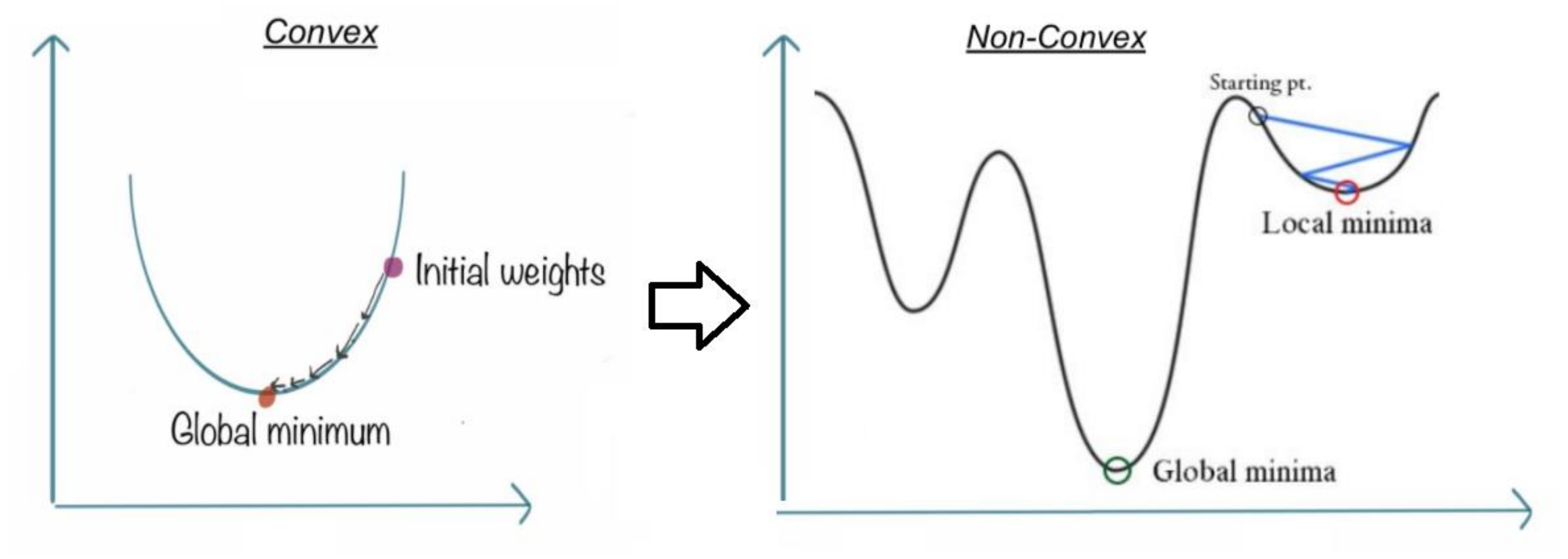
γ is **the learning rate (lr)** (delta-time in physics):

- The correct value depend on the problem at hand, i.e. it depends on the magnitude of $\frac{dL}{dw_{ij}}$
- The convergence of the algorithm critically depends on γ:
  - too small cause the process to get stuck
  - too large may cause instability

There exist more sophisticated algorithm variation, some example are:

- ADAM : calculates the exponential moving average of gradients to compute γ , also use momentum
- RMSPROP root mean square propagation, use momentum

Non convex optimization

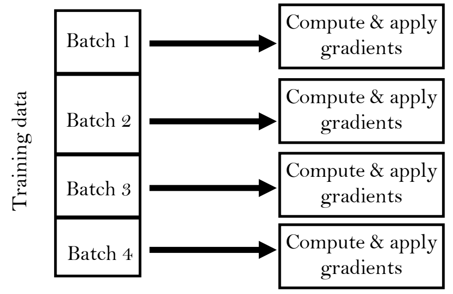


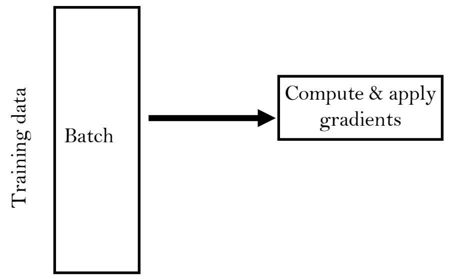
The cost function of a neural network is in general neither convex nor concave, this mean that gradient based optimization often does not converge to the global minima. This is not seen as huge a problem because the community believe that all the local minima are equivalent and are close to the global one (no demonstration just belief)

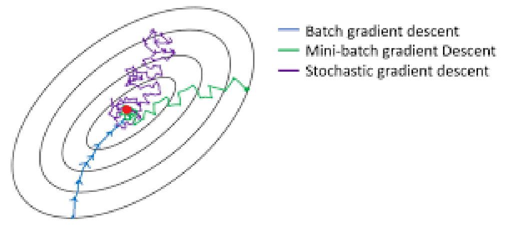
Some tips

- ReLU (or Leaky-ReLU) is the typical choice for the activation of the hidden layer, while the activation of the last layer depends on the task (regression/classification/etc…). The benefit is the reduced likelihood of the gradient to vanish. The gradient has a constant value (for x>0). In contrast, the gradient of sigmoids becomes increasingly small as the absolute value of x increases. The constant gradient of ReLUs results in faster learning.

- The gradient update is done in (mini)batch (called stochastich gradient descent), i.e you only take a subset of all your data during one iteration. **The iteration on all (mini)batch define an epoch.**


Training data
Batch 1
Batch 2
Batch 3
Batch 4
Compute & apply gradients
Compute & apply gradients
Compute & apply gradients
Compute & apply gradients

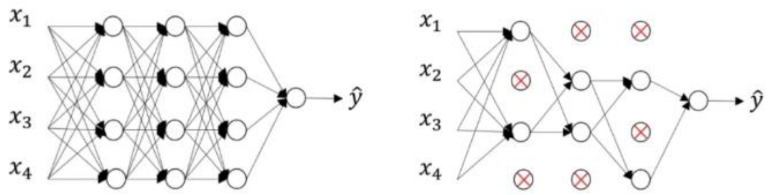




- Normalize/Standardize input features: activation and its derivatives act in limited region

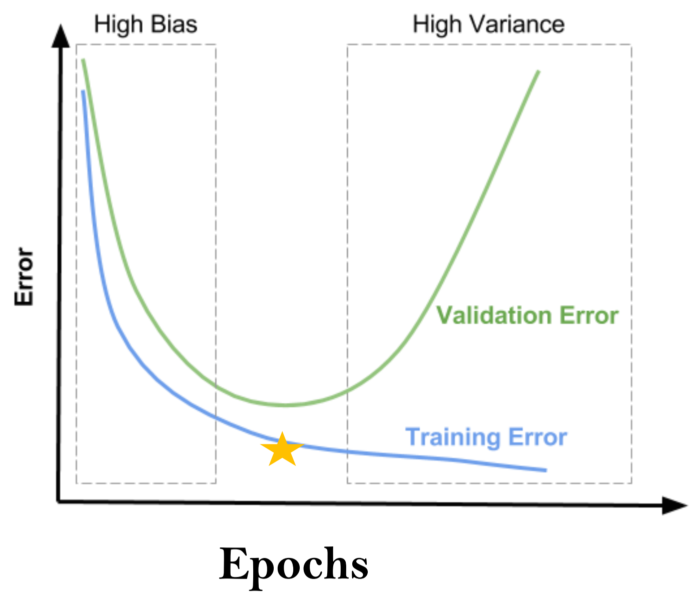
- Use regolarizations method:
  - Dropout: i.e. randomly remove some neurons of the network
  - Apply L_1 or L_2 weight regularization, i.e. add to the loss the terms $\lambda \sum w_{ij}^2$ or $\lambda \sum |w_{ij}|$

- **Early stopping**



Early stopping

Training set	Validation set	Test set
Used to fit the model (70% of data)	Used to evaluate the **model performances** (15% of data)	Used to evaluate the **model generalization** (15% of data)



★ **Early Stopping**

Stop the training process when the Validation Error increase

Categorical variables

Ordinal Encoding and one hot encoding can be a valid approch, but a valid alternatives is to use **Embedding Layers**

Cat features
Dog
Cat
…
Monkey

One hot indices of word $\longrightarrow$

Cat features
[0]
[1]
…
[27]

$\longrightarrow$

Emb 1	Emb 2	Emb 3
0.1	0.15	0.22
0.05	0.21	0.13
…	…	…
…	….	…

Lookup Table

Index	Emb 1	Emb 2	Emb 3
0	0.1	0.15	0.22
1	0.05	0.21	0.13
2	…	…	…
…	…	…	…
27	…	….	…

This weights are learning via backpropagation just any other parameters

Python hands on

Notebook **Python_basics**

Example of Datasets

Application to Finance

Motivation: the calibration problem

Calibration is an **inverse problems**: compute input parameters from observed output

Direct Problem:

Black-Scholes Model

$dS_t = rSdt + \sigma SdW_t$

r $\rightarrow$ Drift rate
σ $\rightarrow$ Volatility
S_t $\rightarrow$ underlying asset price

$$\xrightarrow{\text{Given } r, \sigma}$$

Compute price for
call & put options

$c(T, K) = E[\max\{S_T - K, 0\}]^*$

T $\rightarrow$ time to maturity
K $\rightarrow$ Strike price

Inverse Problem:

Compute parameters
$\boldsymbol{r, \sigma}$

$$\xleftarrow[\text{Assuming market obey BS Model}]{\text{Given call price}}$$

$call = E[\max\{S_T - K, 0\}]$

T $\rightarrow$ time to maturity
K $\rightarrow$ Strike price

* Can be computed by directly solve BS Eq., via montecarlo integration, or using Fast Fourier Trasform method etc…

Motivation: the calibration problem

Calibration is an **inverse problems**: compute input parameters from observed output

Direct Problem:

Black-Scholes Model

$$dS_t = rSdt + \sigma SdW_t$$

r $\rightarrow$ Drift rate
σ $\rightarrow$ Volatility
S_t $\rightarrow$ underlying asset price

$$\xrightarrow{\text{Given } r, \sigma}$$

Compute price for
call & put options

$$c(T, K) = E[\max\{S_T - K, 0\}]^*$$

T $\rightarrow$ time to maturity
K $\rightarrow$ Strike price

Inverse Problem:

Compute parameters
$\boldsymbol{r, \sigma}$

$$\xleftarrow[\text{Assuming market obey BS Model}]{\text{Given call price}}$$

$$call = E[\max\{S_T - K, 0\}]$$

T $\rightarrow$ time to maturity
K $\rightarrow$ Strike price

* Can be computed by directly solve BS Eq., via montecarlo integration, or using Fast Fourier Trasform method etc…

Motivation: the calibration problem

Calibration can be formalized as an **optimization problem**

$\Theta = \{r, \sigma\}$ Model parameters (depend on the model)

$$\Theta_{opt} = argmin_{\Theta}\, d(\boxed{c_{BS}^{Model}(\Theta, T, K)}, \boxed{c^{market}(T, K)})$$

This is a quantity computed from the model, it can be the price of a call option in the next chapter we will also make use of **implied volatility**

This is our data coming from **market**, is the imput of our calibration procedure! **In the next slide / chapter we will see how to deal with (T, K).**

d is the distance metric tipically we will use $d(x, y) = (x - y)^2$

Motivation: the calibration problem

Calibration can be formalized as an **optimization problem**

$\Theta = \{ r, \sigma \}$ Model parameters (depend on the model)

$$\Theta_{opt} = \ argmin_{\Theta}\ d(\boxed{c^{Model}(\Theta, T, K),} \boxed{c^{market}\,(T, K)})$$

T	K	c^{market}
T_1	K_1	c_1^{market}
T_2	K_2	c_2^{market}
…	…	…

This is a quantity is computed from the model, it can be the price of a call option or (see next chapters) we will also make use of **implied volatility**

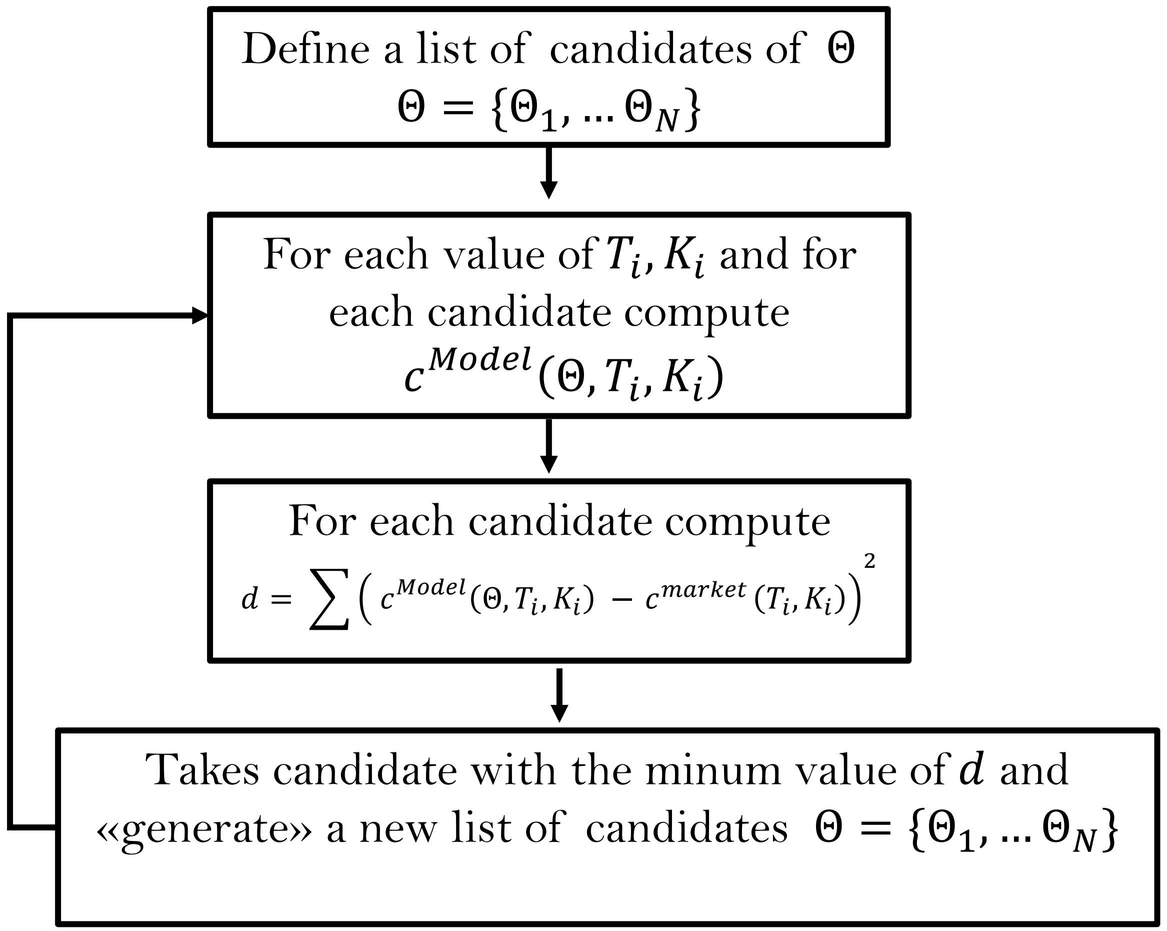
This is our data coming from **market**, is the imput of our calibration procedure!

d is the distance metric tipically we will use $d(x, y) = (x - y)^2$

$$\Theta_{opt} = \ argmin_{\Theta} \sum_i \left(c_{BS}^{Model}(\Theta, T_i, K_i) \ - c^{market}\,(T_i, K_i) \right)^2$$

Two approches : gradient free (addressed marginally in this course), **gradient based (aim of this course)**

Gradient Free



Define a list of candidates of Θ $\Theta = \{\Theta_1, \dots \Theta_N\}$

For each value of T_i, K_i and for each candidate compute $c^{Model}(\Theta, T_i, K_i)$

For each candidate compute $d = \sum \left(c^{Model}(\Theta, T_i, K_i) - c^{market}(T_i, K_i) \right)^2$

Takes candidate with the minum value of d and «generate» a new list of candidates $\Theta = \{\Theta_1, \dots \Theta_N\}$

- Stop the iteration when a given rules is achieved
- This class of algorithm are tipically referred as **«genetic» algortihm,** they differ on the way the «generate» and «stop» the iteration
- **The candidate with the minimum value of d at the end of the iterations is the solution of the calibration problem**

Gradient Based: gradient descent

$$\Theta_{opt} = argmin_{\Theta} \sum_i \left(c^{Model}(\Theta, T_i, K_i) - c^{market}(T_i, K_i) \right)^2 = argmin_{\Theta}\, d(\Theta)$$

We want to solve the problem: $\Theta_{opt} = argmin_{\Theta} d\,(\Theta)$ or equivalently, find Θ_{opt} such as $\frac{d}{d\Theta} d\,(\Theta)|_{\Theta_{opt}} = 0$

Define an iterative (pseudo-dynamics) algorithm as follow:

$$\Theta_0 = \Theta_0$$

$$\Theta_{t+1} = \Theta_t - \gamma \frac{d}{d\Theta} d(\Theta)\Big|_{\Theta_t}$$

$$\Big\downarrow \quad \overset{t\,\to\,\infty}{\Theta_t \to \Theta_{t+1} \to \Theta_{opt}}$$

$$\cancel{\Theta_{opt}} = \cancel{\Theta_{opt}} - \gamma \frac{d}{d\Theta} d(\Theta)\Big|_{\Theta_{opt}}$$

$$\frac{d}{d\Theta} d\,(\Theta)\Big|_{\Theta_{opt}} = 0$$

γ is **the learning rate (lr)** (delta-time in physics):

- The correct value depend on the problem at hand, i.e. it depends on the magnitude of $\frac{d}{d\beta} L(\beta)$
- The convergence of the algorithm critically depends on γ:
  - too small cause the process to get stuck
  - too large may cause instability

See "Introduction to ML" course for further details

Gradient Based

In gradient based method we want to compute the derivatives of $\frac{\mathrm{d}}{d\Theta} d(\Theta) \rightarrow \frac{d}{d\Theta} c_{BS}^{Model}(\Theta, T_i, K_i)$ but this is not always possible because we **do not have analytical functions** for $c^{Model}(\Theta, T_i, K_i)$.

The solutions is to approximate $c^{Model}(\Theta, T_i, K_i)$ using a Neural Network*!!!

* **Just because we are really good to compute gradient of neural networks**

Gradient Based

Generate a grid of $\Theta_{\mathrm{i}}, T_i, K_i$ and compute $c_{BS}^{Model}(\Theta_{\mathrm{i}}, T_i, K_i)$ via montacarlo, FFT, etc…

Train a neural network $NN(\Theta_{\mathrm{i}}, T_i, K_i \mid w)$ to approximate $c_{BS}^{Model}(\Theta_{\mathrm{i}}, T_i, K_i)$ i.e solve:

$$argmin_{\mathrm{w}} \sum_i (NN(\Theta_{\mathrm{i}}, T_i, K_i \mid w) - c_{BS}^{Model}(\Theta_{\mathrm{i}}, T_i, K_i))^2$$

we are computing *w* i.e. the network weights

given $NN(\Theta_{\mathrm{i}}, T_i, K_i \mid w)$
we aim to solve :

$$argmin_{\Theta} \sum_i \left(NN(\Theta_{\mathrm{i}}, T_i, K_i \mid w) - c^{market}(T_i, K_i)\right)^2$$

we are computing Θ i.e. the model parameters

At the end we should compute $\frac{d}{d\Theta} NN(\Theta, T_i, K_i)$ instead of $\frac{d}{d\Theta}\, c^{Model}(\Theta, T_i, K_i)$

At the end we should compute $\frac{d}{d\Theta} NN(\Theta, T_i, K_i)$ this will be similar to compute $\frac{d}{d\mathrm{w}} NN(\Theta, T_i, K_i \mid w)$ presented in the previous chapter

Lab 1: NN to price option

Presented in this lab session

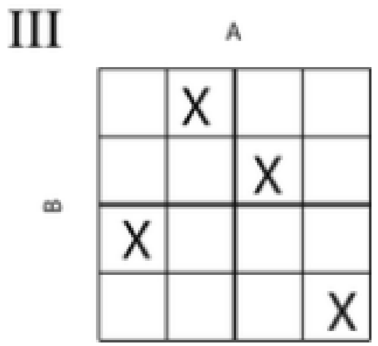
Generate a grid of $\Theta_{\mathrm{i}}, T_i, K_i$ and compute $c_{BS}^{Model}(\Theta_{\mathrm{i}}, T_i, K_i)$ via montacarlo, FFT, etc…

Train a neural network $NN(\Theta_{\mathrm{i}}, T_i, K_i \mid w)$ to approximate $c_{BS}^{Model}(\Theta_{\mathrm{i}}, T_i, K_i)$ i.e solve:

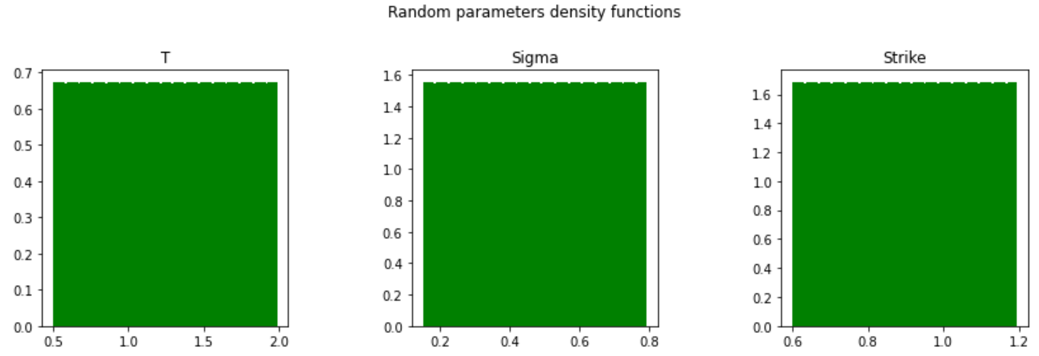
$$argmin_{\mathrm{w}} \sum_i (NN(\Theta_{\mathrm{i}}, T_i, K_i \mid w) - c_{BS}^{Model}(\Theta_{\mathrm{i}}, T_i, K_i))^2$$

Highlights

- Generate a mesh of point Θ_i, T_i, K_i using latin hypercube cube sampling https://en.wikipedia.org/wiki/Latin_hypercube_sampling



there is only one sample in each row and each column



- Use CFLib (by Prof.Rossi presented during the computational finance class) to generate $c_{BS}^{Model}(\Theta_i, T_i, K_i)$ (for BS model the solution is known and analytical formula are available)

Exercise for you: Try to understand what gen_BnS is doing

Tips:

- Put Call parity: $P - C = Ke^{-rT} - S_0$
- P = $N(-d_-)Ke^{-rT} - N(-d_+)S_0$ where $N(x)$ is the normal cumulative distribution function
- $d_+ = \frac{1}{\sigma\sqrt{T}}[\ln(\frac{S_0}{K}) + (r + \frac{\sigma^2}{2})T]$, $d_- = d_+ - \sigma\sqrt{T}$

- Train a NN to approximate $c_{BS}^{Model}(\Theta_i, T_i, K_i)$

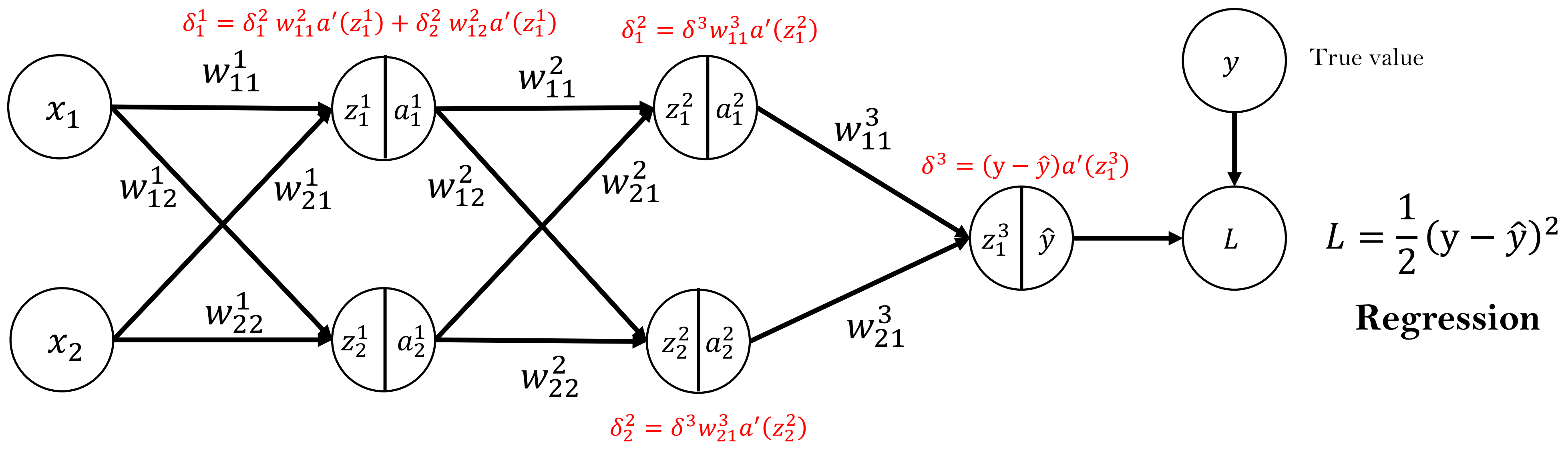
Lab 2: compute derivatives

Motivation

To solve $argmin_{\Theta} \sum_i \left(NN(\Theta_{\mathrm{i}}, T_i, K_i \mid w) - c^{market}(T_i, K_i)\right)^2$ using a gradient based approach we need to compute $\frac{d}{d\Theta} NN(\Theta, T_i, K_i)$ this can be done:

- Using backpropagation
- Using chain rules
- Using low level function of tensorflow

Backpropagation


$\delta_1^1 = \delta_1^2\, w_{11}^2 a'(z_1^1) + \delta_2^2\, w_{12}^2 a'(z_1^1)$
$\delta_1^2 = \delta^3 w_{11}^3 a'(z_1^2)$
$\delta^3 = (y-\hat{y})a'(z_1^3)$
$\delta_2^2 = \delta^3 w_{21}^3 a'(z_2^2)$
x_1
x_2
w_{11}^1
w_{12}^1
w_{21}^1
w_{22}^1
z_1^1 a_1^1
z_2^1 a_2^1
w_{11}^2
w_{12}^2
w_{21}^2
w_{22}^2
z_1^2 a_1^2
z_2^2 a_2^2
w_{11}^3
w_{21}^3
z_1^3 $\hat{y}$
y
True value
L
$L = \frac{1}{2}(y-\hat{y})^2$
Regression

$$\frac{dL}{dw_{ij}^k} = \delta_j^k a_i^{k-1}$$

For the final layer (M)

$$\delta_1^M = a'(z_1^M)(y-\hat{y})$$

$$\delta_j^k = a'(z_j^k)\sum_{l=1}^{r^{k+1}} w_{jl}^{k+1}\delta_l^{k+1}$$

$$\boxed{\frac{d\hat{y}}{d\mathbf{x}} = \delta_1^k W^1}$$

$$(1,K)\ x\ (K,N) = (1,N)$$

For the final layer (M)

$$\delta_1^M = a'(z_1^M)\cancel{(y-\hat{y})}$$

$$\delta_j^k = a'(z_j^k)\sum_{l=1}^{r^{k+1}} w_{jl}^{k+1}\delta_l^{k+1}$$

Chain Rules

$$NN(x) = f^n\left(f^{n-1}(\dots f^1(x))\right)$$

$$\frac{dNN(x)}{dx} = \frac{df^n}{df^{n-1}} \frac{df^{n-1}}{df^{n-2}} \cdots \frac{df^1}{dx}$$

$$x \rightarrow a\left(\underbrace{W^i x + b^i}_{a_1}\right) \rightarrow a\left(W^i a_1 + b^i\right) \rightarrow \ \dots \ \rightarrow W^n a_{n-1} + b^n$$

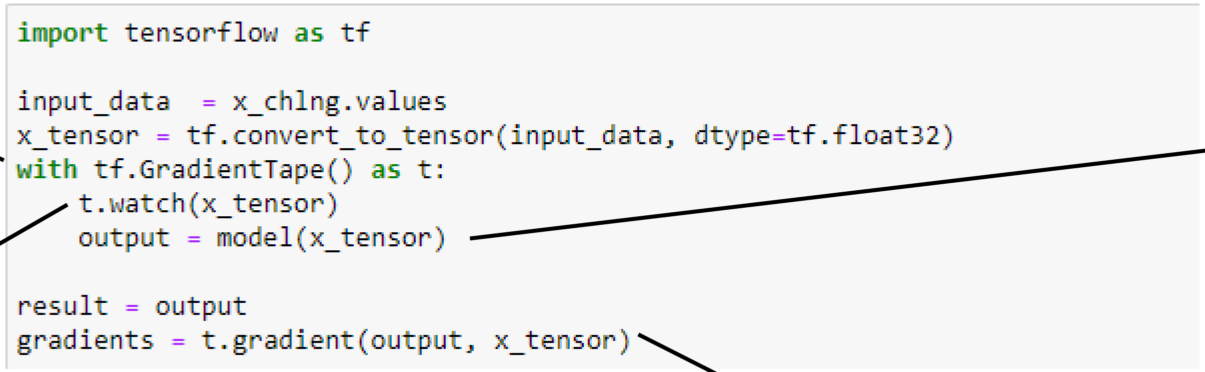
$$\frac{df^1}{dx} = W^1 \sigma'(W^1 x + b^1) \qquad a_1 = \sigma(W^1 x + b^1)$$

$$\frac{df^1}{df^2} = W^2 \sigma'(W^2 a_1 + b^2) \qquad a_2 = \sigma(W^2 x + b^2)$$

$$\frac{df^n}{df^{n-1}} = W^n$$

Tensorflow

Used to record operations for automatic differentiation.



```
import tensorflow as tf

input_data  = x_chlng.values
x_tensor = tf.convert_to_tensor(input_data, dtype=tf.float32)
with tf.GradientTape() as t:
    t.watch(x_tensor)
    output = model(x_tensor)

result = output
gradients = t.gradient(output, x_tensor)
```

Operation in the Tape

Tracing a tensor inside a Tape

The gradient

Internally it uses backpropagation

Lab 3: Calibration

Compute price using FFT

FFT : Fast **Fourier Transform** (see notes on Lab3: generate_price_with_fft_Heston.pdf)

$$\mathbb{E}[(K-S_T)^+] \simeq \frac{1}{2}(K-S_0) + \frac{2}{\pi}\sum_{n=1}^{N/4}\frac{1}{2n-1}\Big[\sin(2\pi k\omega_{2n-1})\Re[K\hat{f}(\omega_{2n-1}) - S_0\hat{f}(\omega_{2n-1}-\frac{i}{2\pi})] - \cos(2\pi k\omega_{2n-1})\Im[K\hat{f}(\omega_{2n-1}) - S_0\hat{f}(\omega_{2n-1}-\frac{i}{2\pi})]\Big].$$

$\omega_n = \frac{n}{2X_c}$

$f: PDF\ of\ the\ \log return\ s_t$

$\hat{f}$: characteristic function

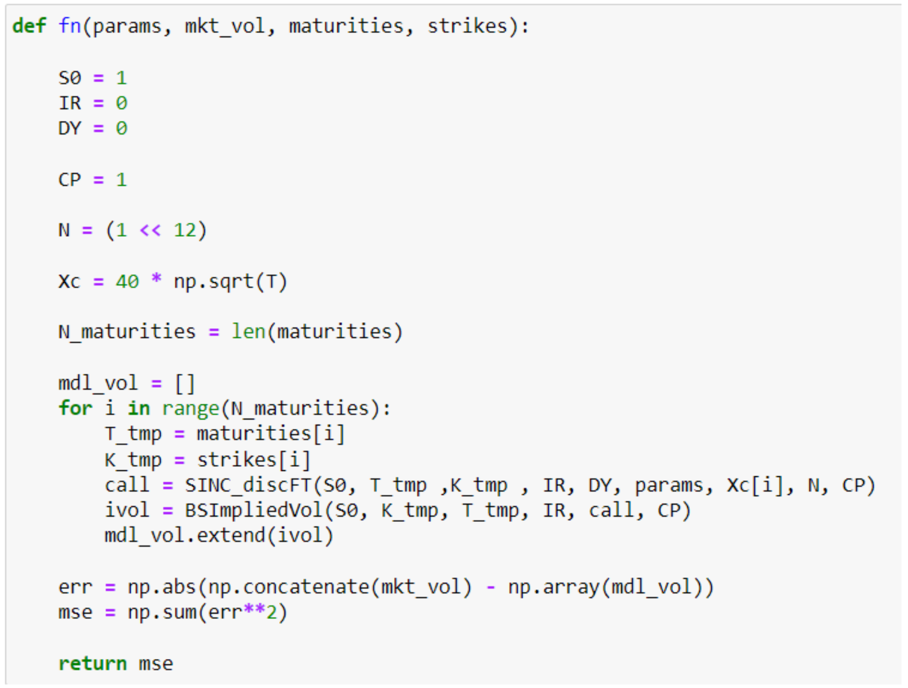
This method is used to generate price and volatilies for the Heston Model

References

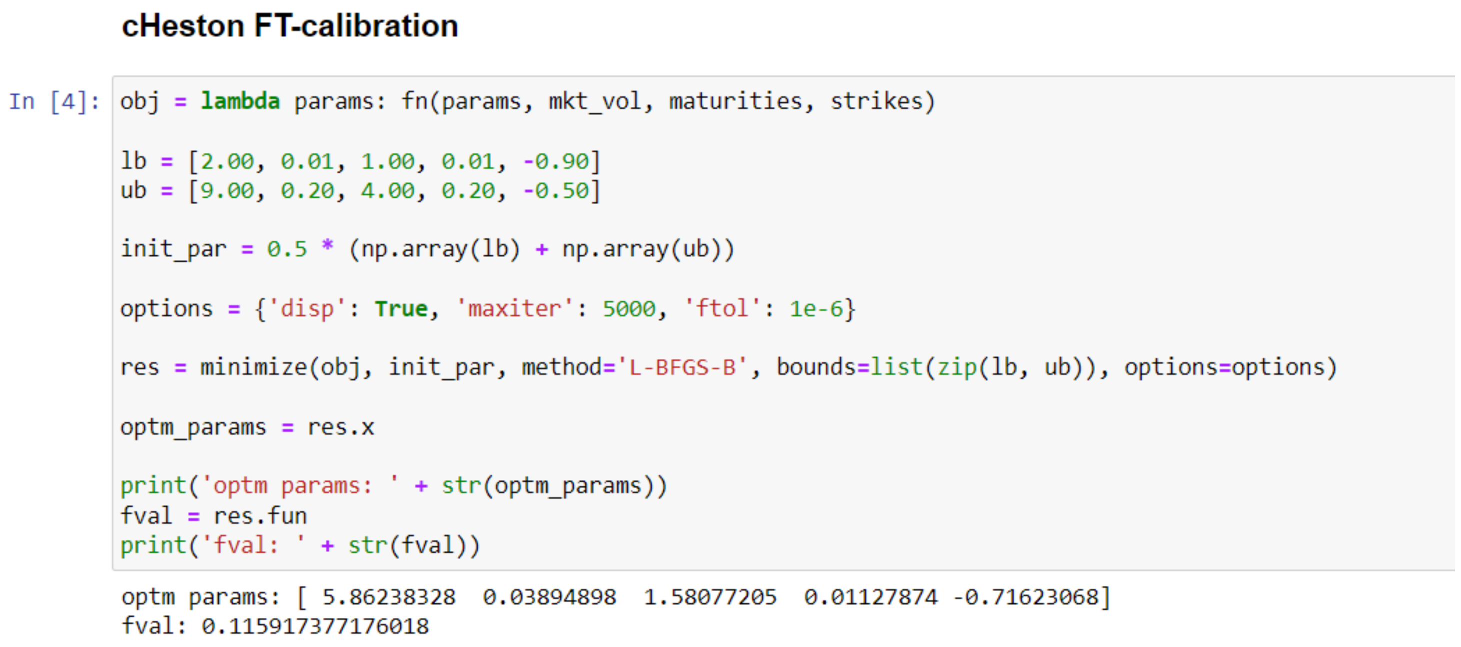
- Deep calibration with random grids, Fabio Baschetti et al
- Deep Learning Volatility: A deep neural network perspective on pricing and calibration in (rough) volatility models, Blanka Horvath et al
- A neural network-based framework for financial model calibration, Liu et al

Lab 4: To the Market

Calibration via FFT price method



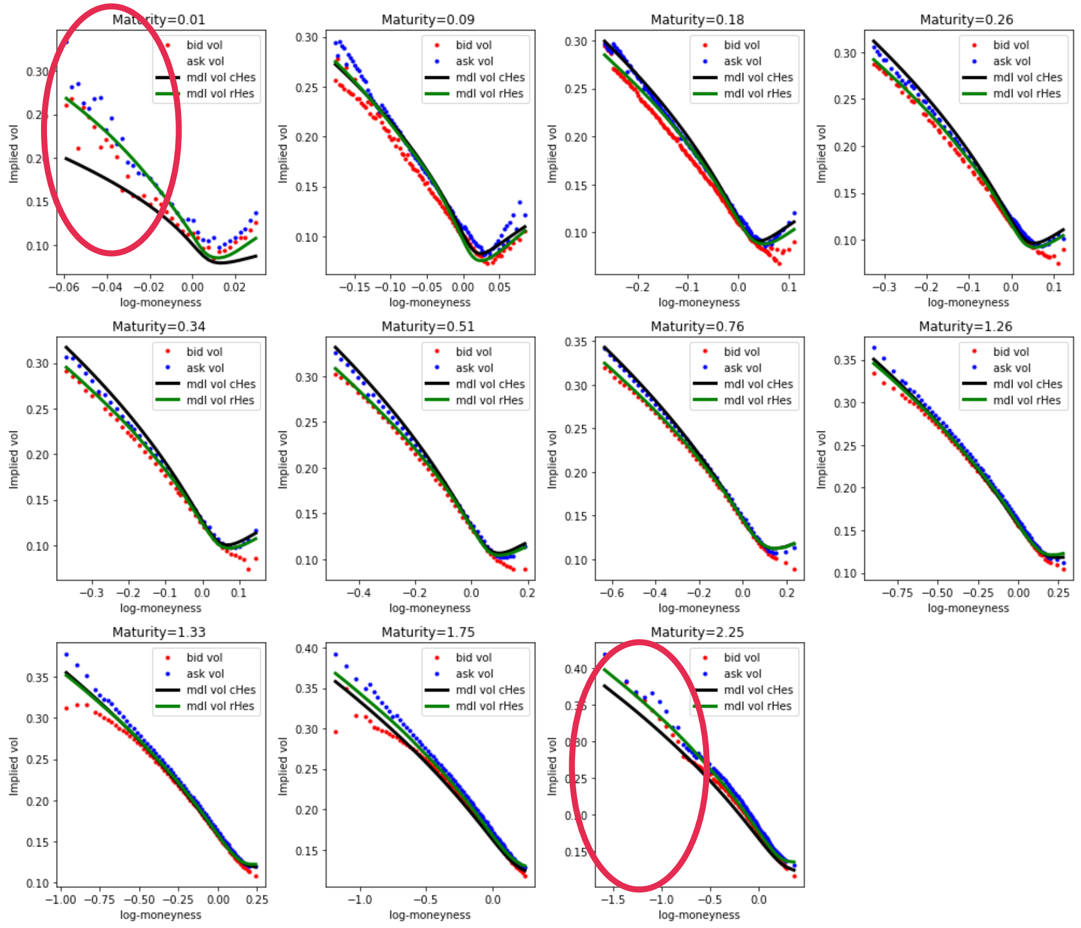
```python
def fn(params, mkt_vol, maturities, strikes):

    S0 = 1
    IR = 0
    DY = 0

    CP = 1

    N = (1 << 12)

    Xc = 40 * np.sqrt(T)

    N_maturities = len(maturities)

    mdl_vol = []
    for i in range(N_maturities):
        T_tmp = maturities[i]
        K_tmp = strikes[i]
        call = SINC_discFT(S0, T_tmp ,K_tmp , IR, DY, params, Xc[i], N, CP)
        ivol = BSImpliedVol(S0, K_tmp, T_tmp, IR, call, CP)
        mdl_vol.extend(ivol)

    err = np.abs(np.concatenate(mkt_vol) - np.array(mdl_vol))
    mse = np.sum(err**2)

    return mse
```



cHeston FT-calibration

```
In [4]: obj = lambda params: fn(params, mkt_vol, maturities, strikes)

        lb = [2.00, 0.01, 1.00, 0.01, -0.90]
        ub = [9.00, 0.20, 4.00, 0.20, -0.50]

        init_par = 0.5 * (np.array(lb) + np.array(ub))

        options = {'disp': True, 'maxiter': 5000, 'ftol': 1e-6}

        res = minimize(obj, init_par, method='L-BFGS-B', bounds=list(zip(lb, ub)), options=options)

        optm_params = res.x

        print('optm params: ' + str(optm_params))
        fval = res.fun
        print('fval: ' + str(fval))

        optm params: [ 5.86238328  0.03894898  1.58077205  0.01127874 -0.71623068]
        fval: 0.115917377176018
```

Rough Heston vs Classical Heston



Bias – Variance trade-off

It can be shown that the expected error on a new unseen points x_o can be written as a sum of 2 terms:

$$E\left(y_0 - \hat{f}(x_0)\right)^2 = \mathrm{Var}(\hat{f}(x_0)) + [\mathrm{Bias}(\hat{f}(x_0))]^2$$

Mean squared error case

The overall expected test MSE can be computed by averaging over all possible values of x_0 in the test set

Variance refers to the amount by which the estimated model f would change if we use a different training data set. In general, more **complex/flexible** statistical methods have higher variance.

Bias refers to the error that is introduced by approximating a real-world problem, by a much simpler model.

- in order to minimize the expected test error, we need to select a statistical learning method that achieves low variance and low bias but…
- For more **complex/flexible** methods, the variance will increase, and the bias will decrease. **Should find a trade-off**.