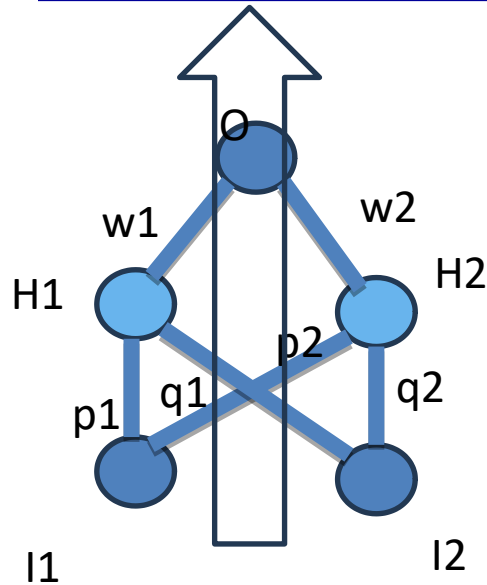

A.I. tutorial and step by step excel/Python samples

About learning rules with examples

Paolo Ferrari, Weibo Li, Giorgia Stendardo

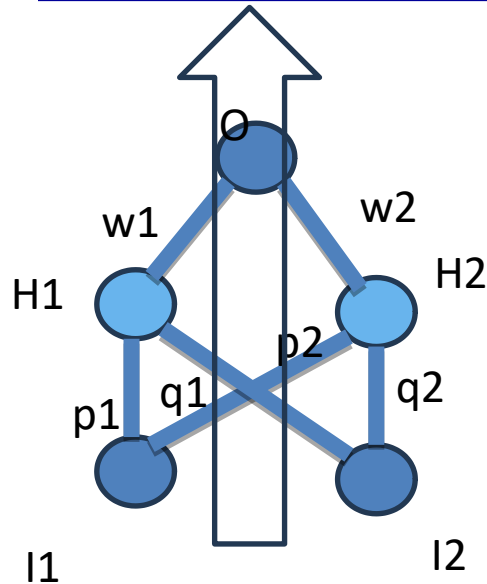
Feed forward network and activation functions



- The perceptron (with or without hidden layer) is a feed forward network in which the information flows in one direction from input nodes to output node.
- Activation function used to determine the state of the nodes can be linear:

$$\left(\quad \right) = \begin{cases} 1, & \geq 0 \\ 0, & < 0 \end{cases} \quad \left(\quad \right) =$$

Feed forward network and activation functions

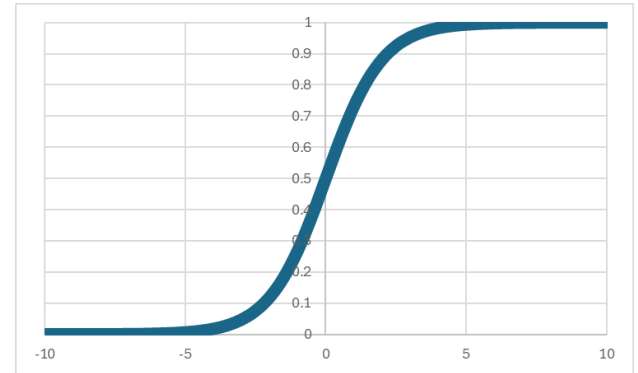


Or non-linear function (that are more effective) as sigmoid, tanh, or ReLU:

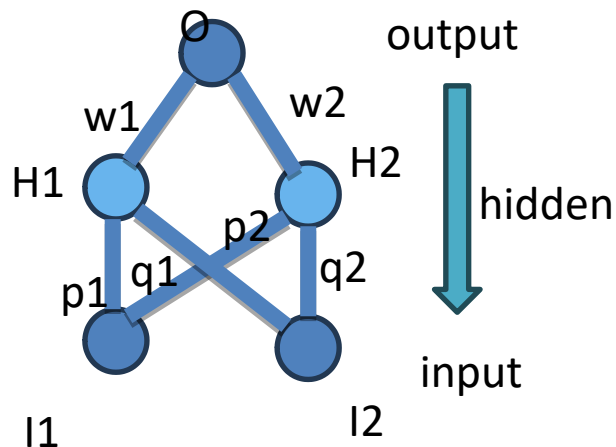
$$\left(\right) = \frac{-}{+}$$

$$\left(\right) = \frac{1}{1+}$$

$$\left(\right) = \max(0; x)$$

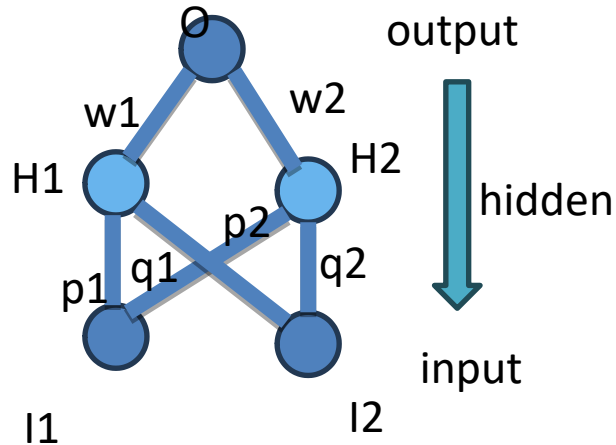


How can we improve the output?



- We consider a figure of merit we want to optimize, a loss function we want to reduce, D
- That can be done using the derivative, i.e. we can compute how D changes with respect a change in the parameter defining our neural network (the weights)

How can we improve the output?



- We start from the output and consider the variation of D with respect to the result of the output layer
- Then we move backward toward considering its variation with respect to the input layer, passing through the hidden layer(s).

Error back propagation

- The status of the output nodes is:
- The status of the hidden nodes is:
- We want a rule that changes the weights in order to minimize the distance between the output of the network, given by the states of the output nodes, O_i , and the wanted configuration, Z_i

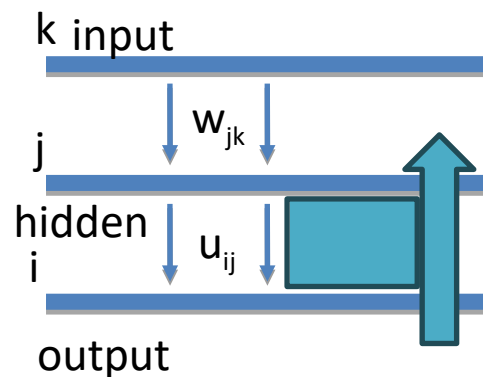
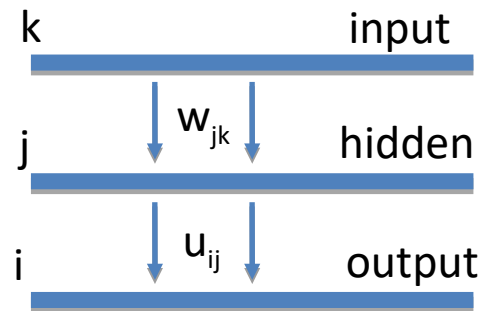
Error back propagation

- We want to minimize: - for one configuration
- If we have p configurations it is:
- This can be done considering the derivative of D with respect the weights starting from the output and going back to the input



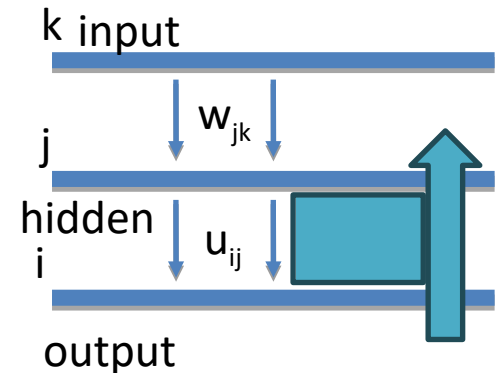
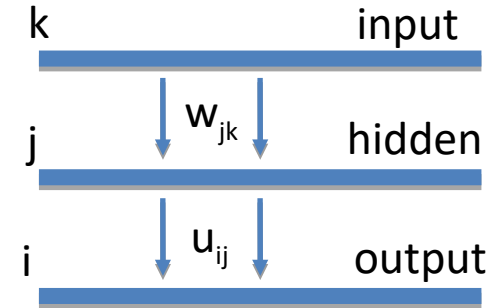
Changing the u_{ji} in case of 1 configuration (2 inp 1 out)

- We start from the output with the aim of obtaining the expected values
- And:
- First, we have to calculate the variation of the weights between the output and the hidden layers through the derivative of D



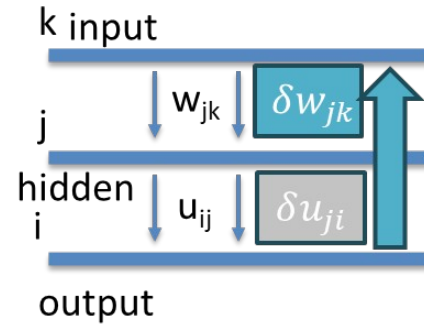
Changing the u_{ji} in case of 1 configuration (2 inp 1 out)

- =
- Where
- And Where is a constant that rules the speed of the variation (learning rate)



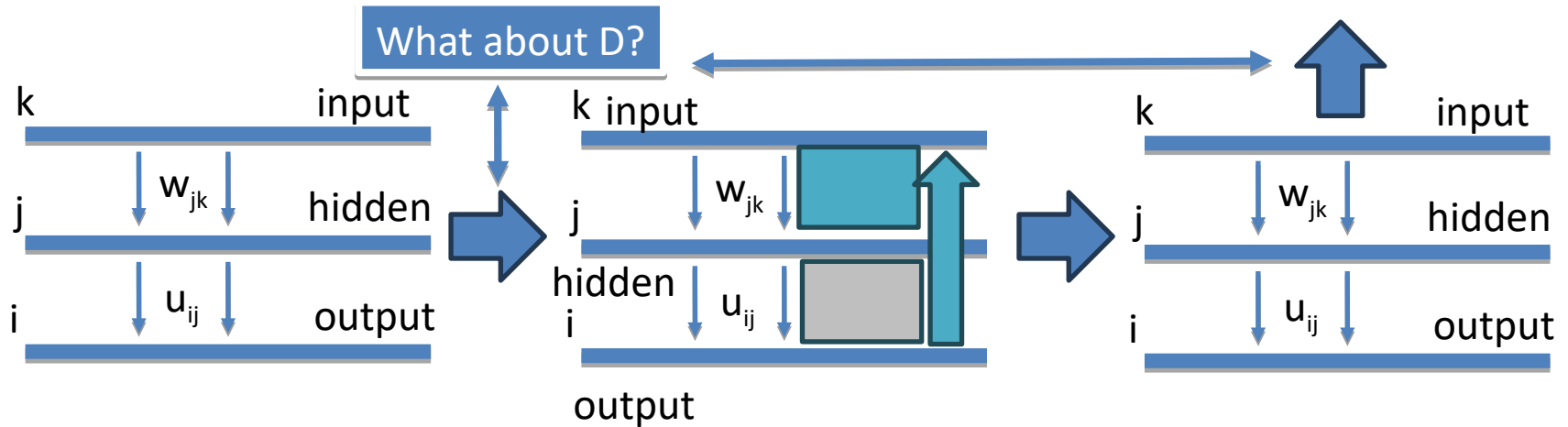
Changing the w_{jk} with 1 configuration(2 inp 1 out)

- The differentiation is now applied to the w_{jk} is a double differentiation



- and δu_{ji} was calculated in the previous step

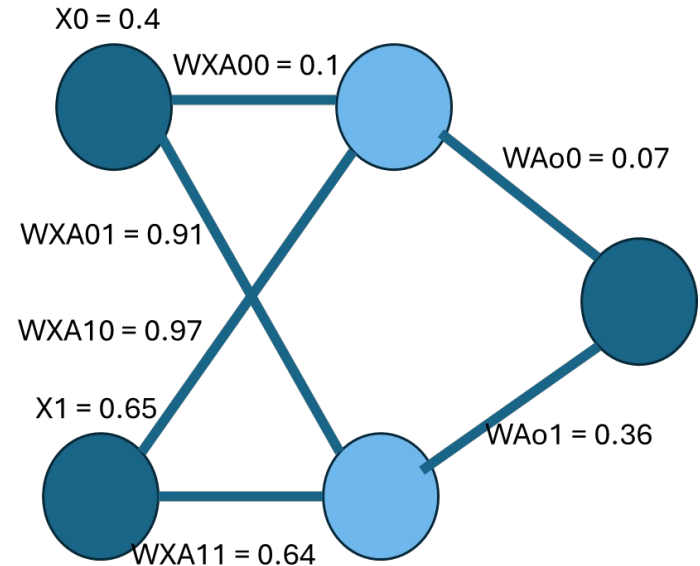
Error back propagation: iterating



Same example but with numbers

The network and the initial states

- X_0 and X_1 are the input nodes
- A_0 and A_1 are the hidden nodes
- The output node is o
- W_{XA} are the weights between input and hidden
- W_{Ao} are the weights between hidden and output

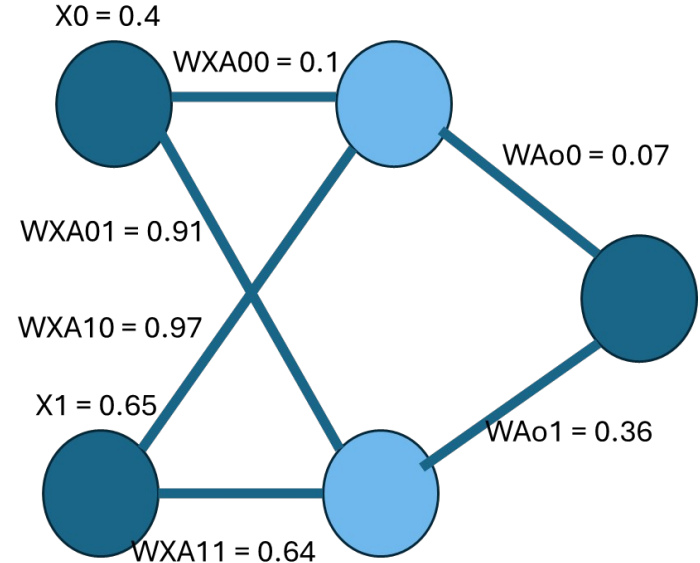


The network and the initial states

- The activation function is the Sigmoid:

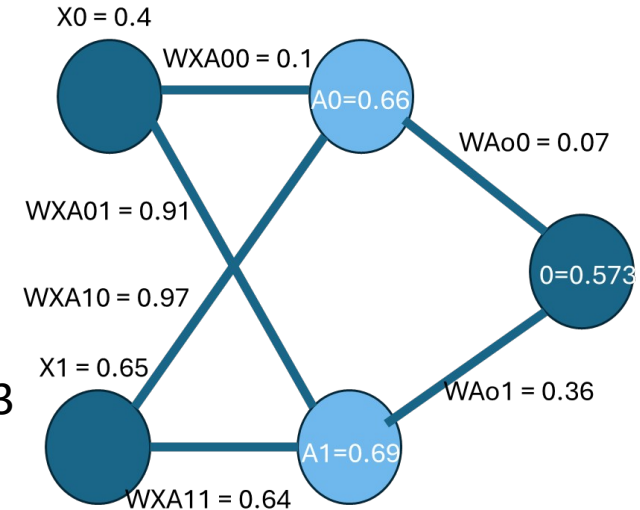
$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

- The expected output Z is 0.51
- The learning rate ε is 0.8



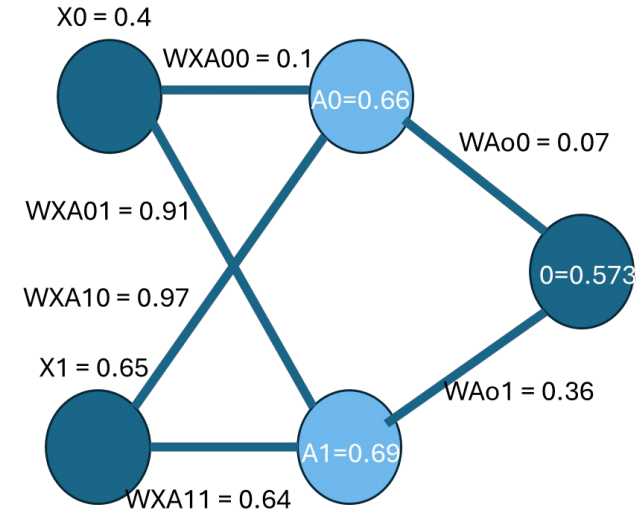
First step: we calculate the initial output and D

- *Hidden layer:*
- Summation on A0: $h_0 = 0.4 * 0.1 + 0.65 * 0.97 = 0.67$
- Calculating the activation function $\text{sigmoid}(0.67) = 0.66$
- Summation on A1: $h_1 = 0.4 * 0.91 + 0.65 * 0.64 = 0.78$
- Calculating the activation function $\text{sigmoid}(0.78) = 0.69$
- *Output layer :*
- Summation on o: $g = 0.07 * 0.66 + 0.36 * 0.69 = 0.29$
- Calculating the activation function $\text{sigmoid}(0.29) = 0.573$
- Calculating D: $0.5 * (0.51 - 0.573)^2 = 0.001985$



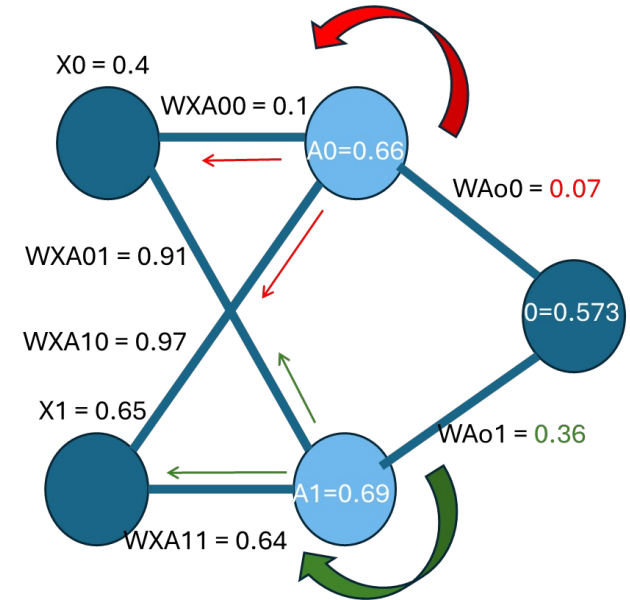
Second step: variation of the weights output - hidden

- The derivative of the sigmoid function is: (sigmoid) (1-sigmoid)
- $Z = 0.51$; ;
- =
- Thus delta is: $-0.06 * 0.573 * (1 - 0.573) = -0.015$
- Variation of the weight output – hidden (where we have 2 nodes):
- Delta_WAo0: $0.8 * (-0.015) * 0.66 = -0.0081$
- Delta_WAo1: $0.8 * (-0.015) * 0.69 = -0.0084$



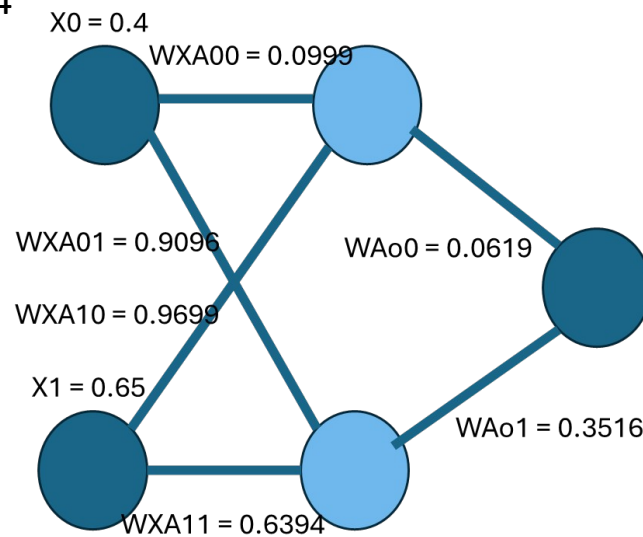
Third step: variation of the weights hidden - input

- -0.015
- $= -0.015 * 0.07 * (1 - 0.66) * 0.66 = -0.0002$
- $= -0.015 * 0.36 * (1 - 0.69) * 0.69 = -0.0012$
-
- Variation of the weight hidden – input :
- $\Delta_{WXA00} = * * X_0 = 0.8 * (-0.0003) * 0.4 = -0.0001$
- $\Delta_{WXA01} = * * X_0 = 0.8 * (-0.0012) * 0.4 = -0.0004$
- $\Delta_{WXA10} = * * X_1 = 0.8 * (-0.0003) * 0.65 = -0.0001$
- $\Delta_{WXA11} = * * X_1 = 0.8 * (-0.0012) * 0.65 = -0.0006$



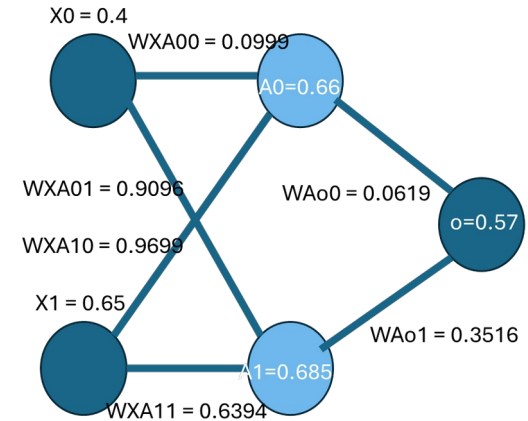
Fourth step: apply the variation to the weights

- New WXA00 = WXA00+Delta_WXA00=0.1 −0.0001 = 0.0999
- New WXA01 = WXA01+Delta_WXA01=0.91 −0.0004 = 0.9096
- New WXA10 = WXA10+Delta_WXA10=0.97 −0.0001 = 0.9699
- New WXA11 = WXA11+Delta_WXA11=0.64 −0.0006 = 0.6394
- New WAO0 = WAO0 + Delta_WAO0 = 0.07 −0.0081 = 0.0619
- New WAO1 = WAO1 + Delta_WAO1= 0.36 −0.0084 = 0.3516

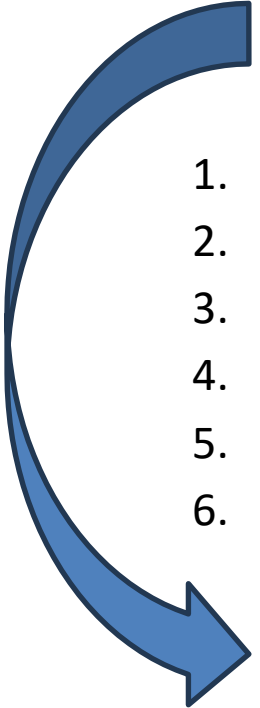
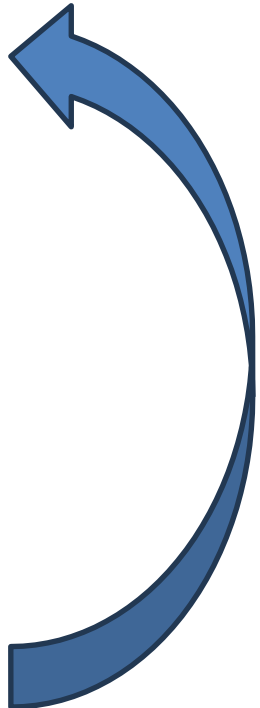


Fifth step: calculate the new output and D

- Summation on A0: $h_0 = 0.4 * 0.0999 + 0.65 * 0.9699 = 0.67$
- Calculating the activation function $\text{sigmoid}(0.67) = 0.66$
- Summation on A1: $h_1 = 0.4 * 0.9096 + 0.65 * 0.6394 = 0.779$
- Calculating the activation function $\text{sigmoid}(0.779) = 0.685$
- Summation on o: $g = 0.0619 * 0.66 + 0.3516 * 0.685 = 0.281$
- Calculating the activation function $\text{sigmoid}(0.281) = 0.57$
- Calculating the new D: $0.5 * (0.51 - 0.57)^2 = \mathbf{0.0018}$
- Initial D value = 0.001985 **(-9%)**



Final step: repeating the scheme

- 
- 
1. Calculate the derivative for the weights output-hidden
 2. Calculate the derivative for the weight hidden-input
 3. Apply the variation
 4. Calculate the output
 5. Calculate the loss function value
 6. If it is not satisfactory go to 1 proceeding with the gradient.

The same but with a code

The same problem using Python and PyTorch (I)

```
[1]: import numpy as np

[2]: import torch

[78]: import torch.nn as nn

# Define the Perceptron input and output
X = torch.tensor([0.4, 0.65], dtype=torch.float32)
y = torch.tensor([0.51], dtype=torch.float32)

# Build the neural network and the feed forward
class Perceptrone(nn.Module):
    def __init__(self):
        super().__init__()
        self.model = nn.Sequential(nn.Linear(2, 2), nn.Sigmoid(), nn.Linear(2, 1), nn.Sigmoid())
    # the activation function in 2x2x1 is a sigmoid
    def forward(self, x):
        return self.model(x)
```

The same problem using Python and PyTorch (II)

```
[79]: model = Perceptrone()

      loss_function = nn.MSELoss() # mean square error

      import torch.optim as optim
      optimizer = optim.SGD(model.parameters(), lr=0.01) # Stochastic Gradient Descent (SGD)

[80]: # Training Loop
      epochs = 3000
      for epoch in range(epochs):
          # Forward
          output = model(X)
          loss = loss_function(output, y)

          # Backward
          optimizer.zero_grad() # Zero the gradients
          loss.backward() # Compute the gradients for the loss function
          optimizer.step() # Update the parameters of the network

          if (epoch + 1) % 100 == 0:
              print(f"Epoch [{epoch+1}/{epochs}], Loss: {loss.item():.6f}")
```

The same problem using Python and PyTorch (III)

```
Epoch [100/3000], Loss: 0.041448
Epoch [200/3000], Loss: 0.030889
Epoch [300/3000], Loss: 0.022499
Epoch [400/3000], Loss: 0.016094
Epoch [500/3000], Loss: 0.011356
Epoch [600/3000], Loss: 0.007934
Epoch [700/3000], Loss: 0.005504
Epoch [800/3000], Loss: 0.003801
Epoch [900/3000], Loss: 0.002616
Epoch [1000/3000], Loss: 0.001797
Epoch [1100/3000], Loss: 0.001233
Epoch [1200/3000], Loss: 0.000845
Epoch [1300/3000], Loss: 0.000579
Epoch [1400/3000], Loss: 0.000396
Epoch [1500/3000], Loss: 0.000271
Epoch [1600/3000], Loss: 0.000186
Epoch [1700/3000], Loss: 0.000127
Epoch [1800/3000], Loss: 0.000087
Epoch [1900/3000], Loss: 0.000060
Epoch [2000/3000], Loss: 0.000041
Epoch [2100/3000], Loss: 0.000028
Epoch [2200/3000], Loss: 0.000019
Epoch [2300/3000], Loss: 0.000013
Epoch [2400/3000], Loss: 0.000009
Epoch [2500/3000], Loss: 0.000006
Epoch [2600/3000], Loss: 0.000004
Epoch [2700/3000], Loss: 0.000003
Epoch [2800/3000], Loss: 0.000002
Epoch [2900/3000], Loss: 0.000001
Epoch [3000/3000], Loss: 0.000001
```

```
# Test the model
with torch.no_grad():
    test_output = model(X)
    calculated = (test_output).float()
    print("\nFinal Predictions:")
    print(
        f"Input: {X}, expected: {y}, got: {calculated} "
```

Final Predictions:

Input: tensor([0.4000, 0.6500]), expected: tensor([0.5100]), got: tensor([0.5090])

Genetic algorithms

Evolving exploiting natural selections rules

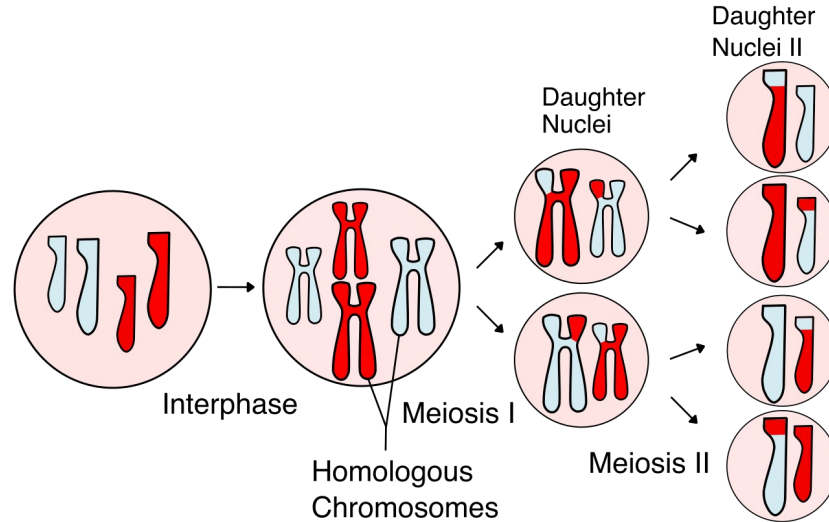
Basic concepts

Genetic algorithms mimic a natural selection process in biology: there is a population of states (individuals) in which the highest value (the fittest) has the highest probability to produce a successor state (offspring) that populate the next generation through a process called recombination.

Elements

- Mixing number: number of individuals needed to generate an offspring
- Selection: choosing a parent is related to its fitness
- Crossover point: that rules the recombination of the genetic material from the parents
- Mutation rate: rate that determines the random mutation of the material of the offspring
- Elitism: which means that the fitness of the subsequent generation does not decrease

Biological analogy

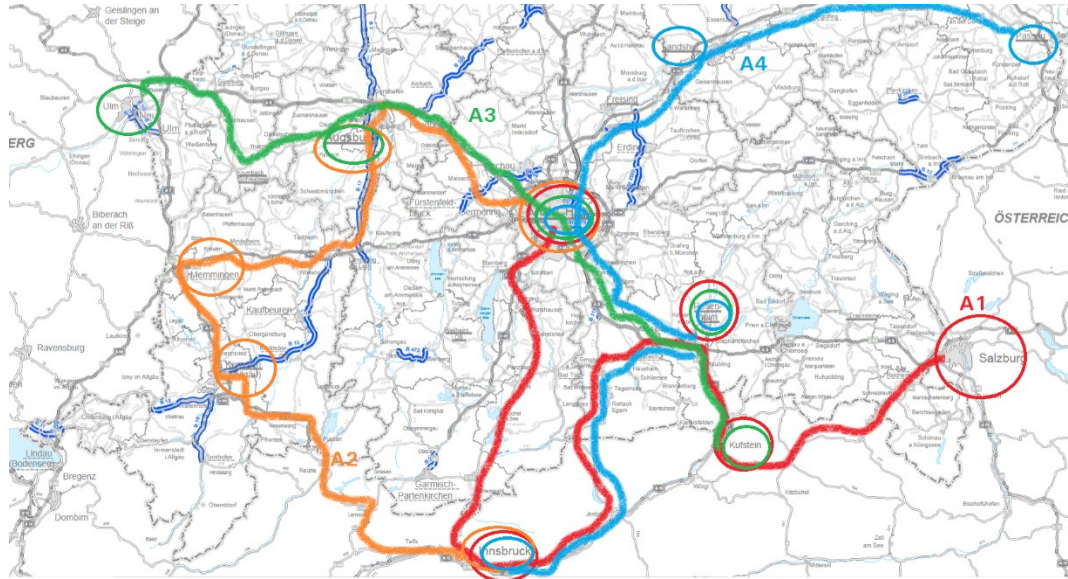


Meiosis image taken from wikipedia

An example of usage of genetic algorithm

Something similar to find a heuristic solutions for the Travel Salesman Problem

We consider 4 possible routes and different places



We are here for tourism. We assess to each place a score derived from the presence of palace/caste/walls (taken from mean Tripadvisor ratings).

Applying the elements of the genetic algorithm

Elements	Our history
Mixing number	We can combine 2 routes together
Selection	Fitness is computed summing for each place the mean value of the Tripadvisor rating
Crossover point	We can decide where to break the route
Mutation rate:	In our case the mutation rate is 100% if the place is already visited previously (we already visited that place, we do not want to come back!)

Initial population members

The first route starts in Salzburg A1: (Sa, Ku, Ro, In, Mu)

The second route starts in Innsbruck A2: (In, Ke, Me, Au, Mu)

The third route starts in Kufstein A3: (Ku, Ro, Mu, Au, U)

The forth route starts in Innsbruck A4: (In, Ro, Mu, La, Pa)

Au: Augsburg

La: Landshut

Ku: Kufstein

In: Innsbruck

Me: Memmingen

Ro: Rosenheim

Ke: Kempten

Mu: Munich

Sa: Salzburg

Pa: Passau

U: Ulm

Initial population members fitness

Based on Tripadvisor city gate/walls/castle
evaluation (no castle=0): A1(17.7)

A2(13.3)

A3(8.9)

A4(17.6)

total=57.5

We have then:

A1($17.7/57.5=0.31$), A2(0.23), A3(0.16), A4(0.30).

Selecting the two parents through the fitness

Based on Tripadvisor city gate/walls/castle evaluation (no castle=0)

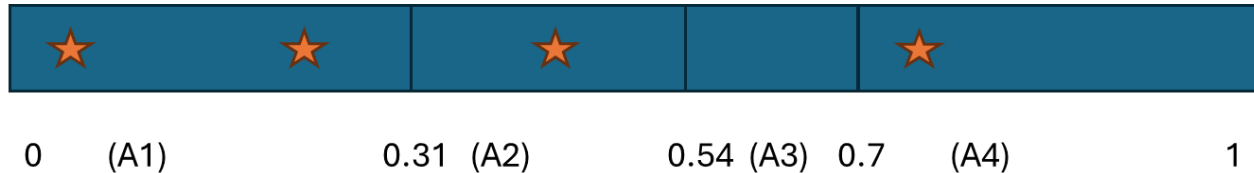
A1(0.31)

A2(0.23)

A3(0.16)

A4(0.30)

We can consider a cumulative distribution and extract a random number uniformly distributed between 0 and 1

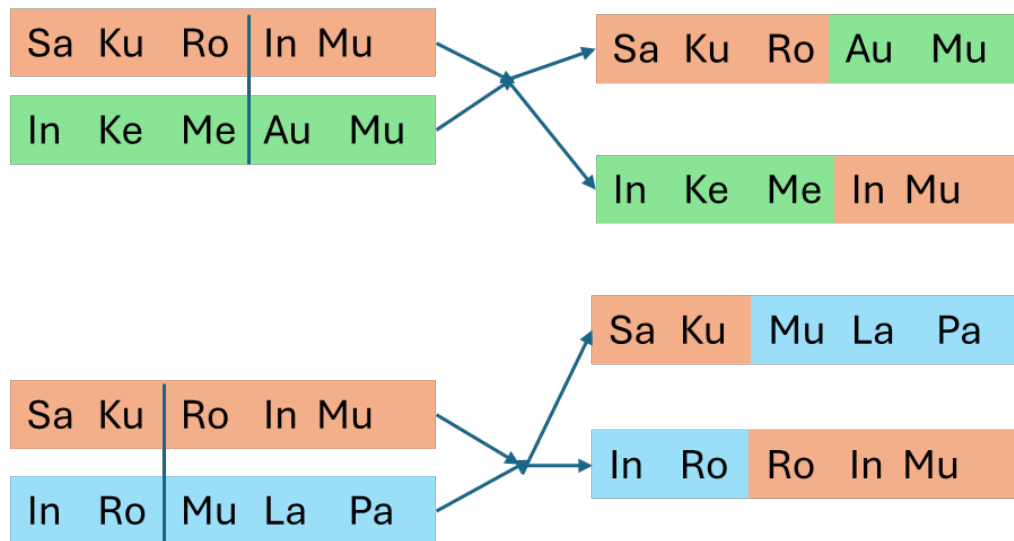


That means that A1 will be used TWICE to generate routes and A3 will NOT be used

The cross-over

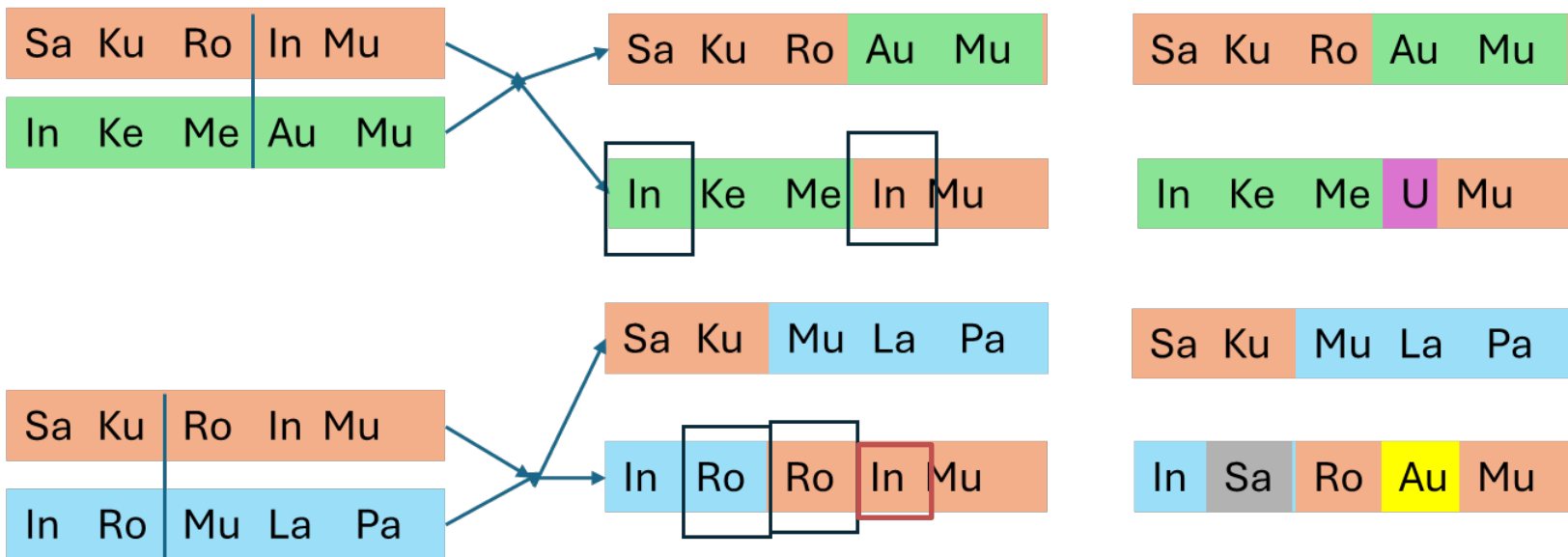
The size of population is (twice A1): A1(Sa, Ku, Ro, In, Mu), A2(In, Ke, Me, Au, Mu), A1(Sa, Ku, Ro, In, Mu), A4(In, Ro, Mu, La, Pa)

We can combine A1
with A2 and A4,
choosing the breaking
point of the cross-over



The mutations

In our case the mutation rate is 100% if the place is already visited previously and an additional one



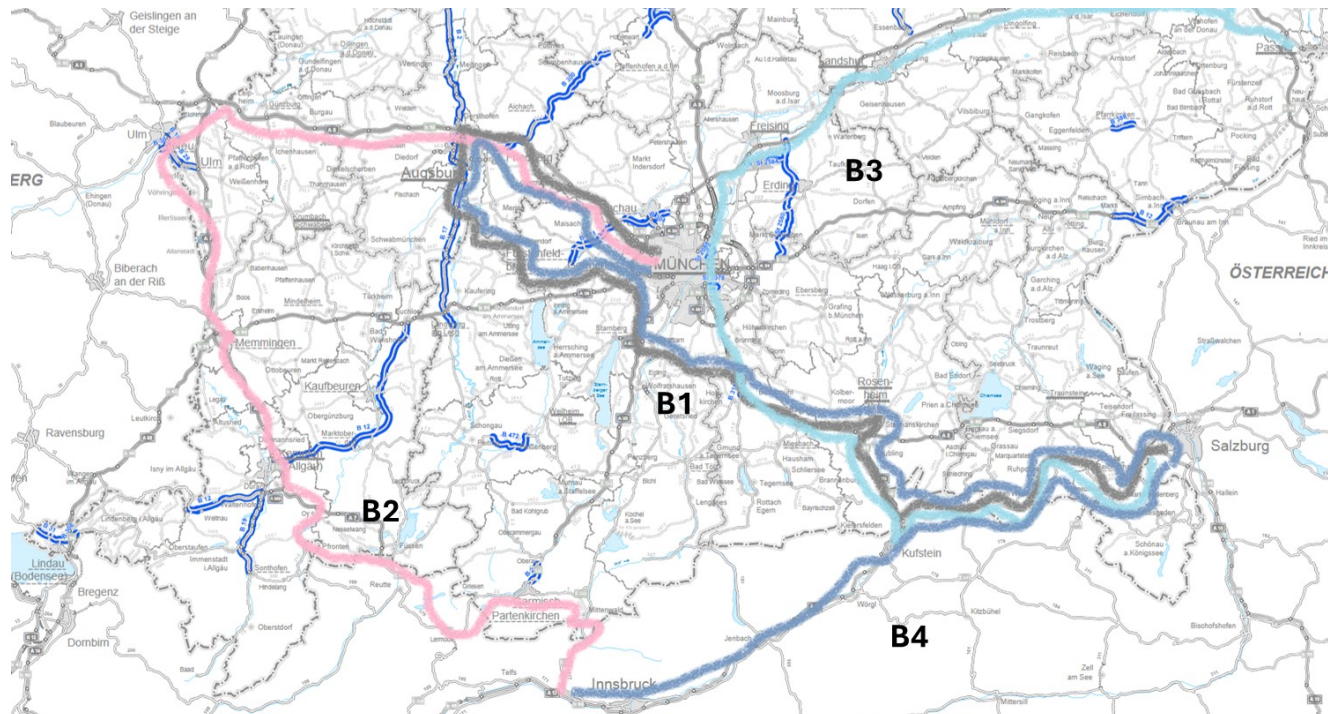
Check the elitism of the offspring

Elitism: which means that the fitness of the subsequent generation does not decrease

New generation	Fitness
B1(Sa, Ku, Ro, Au, Mu)	13.4
B2(In, Ke, Me, U, Mu)	13.3
B3 (Sa, Ku, Mu, La, Pa)	23.3
B4(In, Sa, Ro, Au, Mu)	13.3

Elitism: fitness total score from 57.5 (1st gen.) to 62.2 (2nd gen.)

The new routes



Literature

- Mueller, Reinhardt & Strickland. ***Neural Networks. An introduction.*** 1995
- Russell & Norvig. ***Artificial Intelligence A Modern Approach.*** 4th 2022
- Ashish Ranjan Jha ***Mastering PyTorch.*** 2nd 2024