11. Learning from Examples: NNs and Reinforcement Learning

By: Alex S.

May, 2024

1. Artificial neural networks

Perceptron - is a single neuron neural network model (Figure 1).

It also represents a **feed-forward** network where input is fed towards the output.

The **recurrent** network can also communicate with the previous layers.

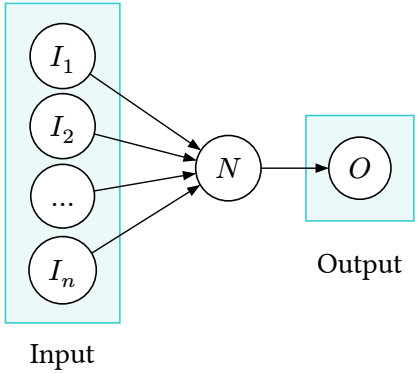


Figure 1: A perceptron

A neuron can be represented as a function in Figure 2. The model takes inputs $\{x_1, x_2, ..., x_n\} \in X$ and they are calculated in the summation function:

$$z_i = \sum_{j=1}^{n} x_j \cdot w_{ji}$$

where w_{ji} is the weight for the specific input and z_i is the summation result.

After that the result goes to the non-linear activation function φ that calculates the output($\hat{y}$) that is called a **prediction**:

$$\hat{y} = \varphi(z_i)$$

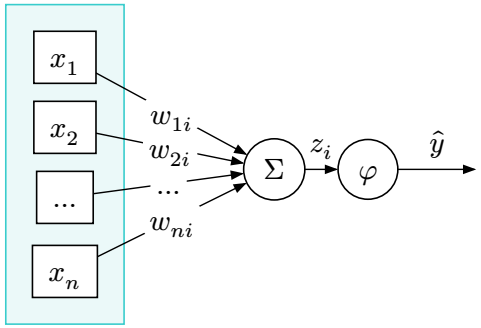


Figure 2: An i-th neuron mathematical representation

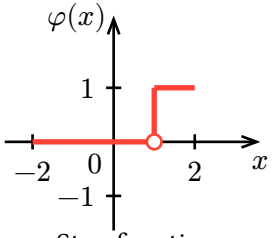
1.1. Activation functions

Activation functions are used to transform the summed weighted input from a node into an output value that is passed on to the next layer.

Step function:

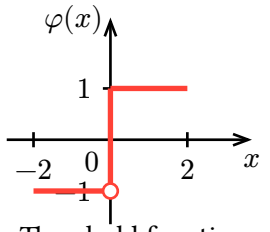
$$\varphi(x) = \begin{cases} 0 \text{ if } x < T \\ 1 \text{ if } x \geq T \end{cases}$$

In the image $T = 1$



Threshold function:

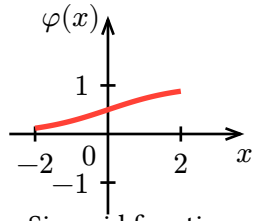
$$\varphi(x) = \begin{cases} -1 \text{ if } x < 0 \\ 1 \text{ if } x \geq 0 \end{cases}$$



Sigmoid function:

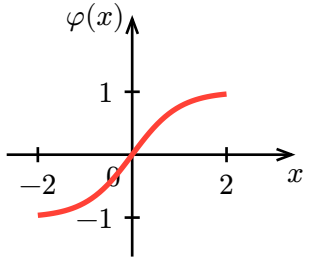
$$\varphi(x) = \frac{1}{1 + e^{-x}}$$

The derivative is $\varphi'(x) = \varphi \cdot (1 - \varphi)$



Tanh function:

$$\varphi(x) = \frac{e^{2x} - 1}{e^{2x} + 1}$$



1.2. Example

Let's say we want to classify black and white dots in Figure 3. The red dashed line represents a linear function that will do the classification job.

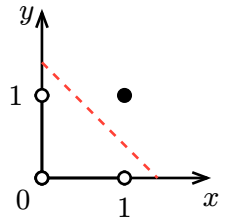


Figure 3: An **AND** classification problem

In order to solve this problem, we can use a simple perceptron with the step function(Figure 4).

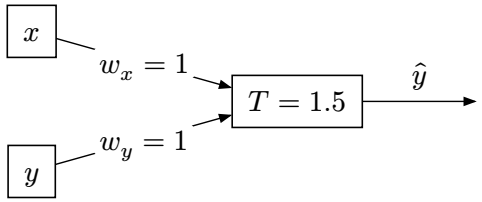


Figure 4: Logical **AND** representation with the perceptron

Step function's threshold is $T = 1.5$, in this case we can create a table to check the results:

x	y	output	$\hat{y}$	y
0	0	0	0	0
0	1	1	0	0
1	0	1	0	0
1	1	2	1	1

Figure 5: Check table for **AND** neuron

The similar perceptron model can be created for the logical "OR" function(Figure 6), the only change is in the step function's threshold.

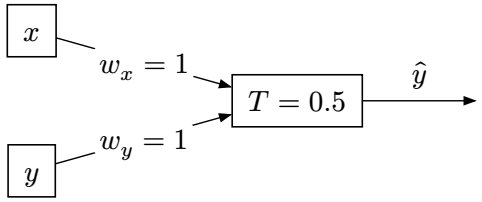


Figure 6: Logical **AND** representation with the perceptron

However, if we want to separate classes with two lines as in Figure 7, i.e. make an **XOR** operation, then it requires to add an additional neuron to the network.

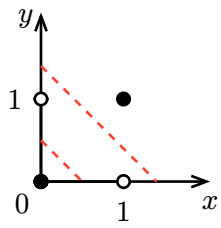


Figure 7: An **XOR** classification problem

2. Multilayer feed-forward neural networks

The generalized structure can be obtained the following way:

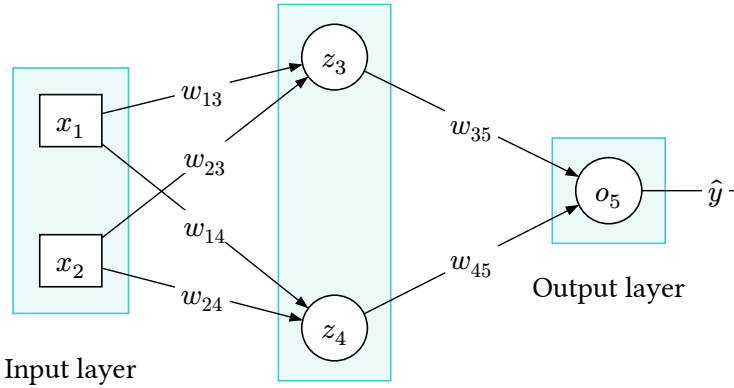


Figure 8: General multilayer neural network scheme

In Figure 8 in order to get the $\hat{y}$ value, the following calculations must be done:

$$o_5 = \varphi(w_{35} \cdot z_3 + w_{45} \cdot z_4)$$
$$z_4 = \varphi(w_{14} \cdot x_1 + w_{24} \cdot x_2)$$
$$z_3 = \varphi(w_{13} \cdot x_1 + w_{23} \cdot x_2)$$

where φ is the activation function

The **XOR** problem mentioned previously can be solved with the following network in Figure 9

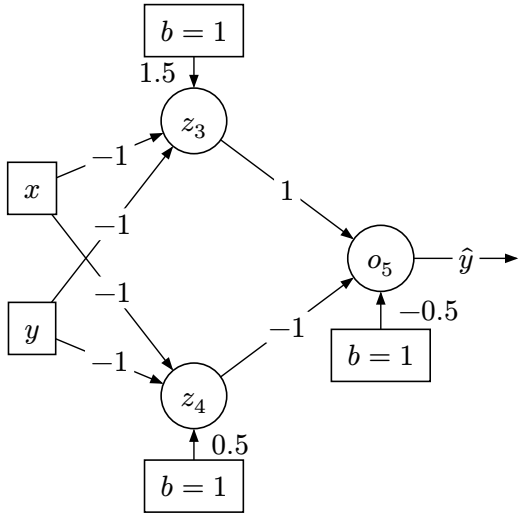


Figure 9: A multilayer network solution for **XOR** problem.

For the network all neurons have a *threshold activation function*:

$$\varphi(x) = \begin{cases} 1 \text{ if } x \geq 0 \\ 0 \text{ if } x < 0 \end{cases}$$

x	y	z_3	$\varphi(z_3)$	z_4	$\varphi(z_4)$	o_5	$\hat{y}$	y
0	0	1.5	1	0.5	1	−0.5	0	0
0	1	0.5	1	−0.5	0	0.5	1	1
1	0	0.5	1	−0.5	0	0.5	1	1
1	1	−0.5	0	−1.5	0	−0.5	0	0

Figure 10: Check table for **XOR** operation neural network. *Note*: $\varphi(o_5) = \hat{y}$

> **Note**
>
> We added the bias to the linear summation.
> E.g. $z_3 = -1 \cdot x + -1 \cdot y + 1.5 \cdot b$

3. Learning in neural networks

- For the perceptron to learn it needs to get *enough* examples to learn a linear function.
- *Enough* examples can be calculated by using Novikov's theorem:

$$N = \frac{D^2}{d^2}$$

where D is the class set diameter, i.e. the maximum distance between examples, and d is the distance between sets, i.e. the distance between the nearest neighbors(examples) of sets

- Learning happens by sequentially providing examples to the perceptron, then calculating the error and updating the weights.

Learning process:

1. Get the output for an example, i.e. predicted value $\hat{y}$, from the network
2. Calculate the error: $E = y - \hat{y}$, where y is the actual value
3. Update the weights with $w_j{}' = w_j + \alpha \cdot x_j \cdot E$ where α is the learning rate, x_j is the input value, E an error
4. Continue the training until all examples are not seen
5. Continue until weights do not change or iteration count is exceeded

3.1. Learning in the perceptron model

Consider the example of getting predictions on the student's marks:

- x_1 - does the student do the assignments(no - 0, yes - 1)
- x_2 - does the student go to lections(no - 0, yes - 1)
- x_3 - does the student prepare for the exam(no - 0, yes - 1)
- $x_4 = 1$ - is a bias

Overall: there are $2^3 = 8$ data points available

Output: +1 or −1 for good and results

Activation function: Threshold function

x_1	x_2	x_3	x_4	y
0	0	0	1	1
1	0	0	1	1
0	1	1	1	−1
1	1	0	1	1

Figure 11: A training dataset for perceptron

Let's do some training. *1st iteration*:

#	Input				Weights				z	$\hat{y}$	y	New weights			
	x_1	x_2	x_3	x_4	w_1	w_2	w_3	w_4				$w_1{}'$	$w_2{}'$	$w_3{}'$	$w_4{}'$
1	0	0	0	1	0	0	0	0	0	1	1	0	0	0	0
2	1	0	0	1	0	0	0	0	0	1	1	0	0	0	0
3	0	1	1	1	0	0	0	0	0	1	−1	0	−1	−1	−1
4	1	1	0	1	0	−1	−1	−1	−2	−1	1	1	0	−1	0

Figure 12: 1st iteration of table with perceptron training. *Note*: $z = \sum_j^n w_j \cdot x_j$

Until the 3rd example, no change in weights was necessary.

3rd example weights updates:

Note: $\alpha = 0.5$

$$E = y - \hat{y} = -1 - 1 = -2$$
$$w_1{}' = w_1 + \alpha \cdot x_1 \cdot E = 0 + 0.5 \cdot 0 \cdot (-2) = 0$$
$$w_2{}' = w_2 + \alpha \cdot x_2 \cdot E = 0 + 0.5 \cdot 1 \cdot (-2) = -1$$
$$w_3{}' = w_3 + \alpha \cdot x_3 \cdot E = 0 + 0.5 \cdot 1 \cdot (-2) = -1$$
$$w_4{}' = w_4 + \alpha \cdot x_4 \cdot E = 0 + 0.5 \cdot 1 \cdot (-2) = -1$$

4th example weights updates:

$$E = y - \hat{y} = 1 - (-1) = 2$$
$$w_1{}' = w_1 + \alpha \cdot x_1 \cdot E = 0 + 0.5 \cdot 1 \cdot 2 = 1$$
$$w_2{}' = w_2 + \alpha \cdot x_2 \cdot E = -1 + 0.5 \cdot 1 \cdot 2 = 0$$
$$w_3{}' = w_3 + \alpha \cdot x_3 \cdot E = -1 + 0.5 \cdot 0 \cdot 2 = -1$$
$$w_4{}' = w_4 + \alpha \cdot x_4 \cdot E = -1 + 0.5 \cdot 1 \cdot 2 = 0$$

2nd iteration:

#	Input				Weights				z	$\hat{y}$	y	New weights			
	x_1	x_2	x_3	x_4	w_1	w_2	w_3	w_4				$w_1{}'$	$w_2{}'$	$w_3{}'$	$w_4{}'$
1	0	0	0	1	1	0	−1	0	0	1	1	1	0	−1	0
2	1	0	0	1	1	0	−1	0	1	1	1	1	0	−1	0
3	0	1	1	1	1	0	−1	0	−1	−1	−1	1	0	−1	0
4	1	1	0	1	1	0	−1	0	1	1	1	1	0	−1	0

Figure 13: 2nd iteration of table with perceptron training. *Note*: $z = \sum_j^n w_j \cdot x_j$

Weights have converged and no change was done.

The final perceptron model:

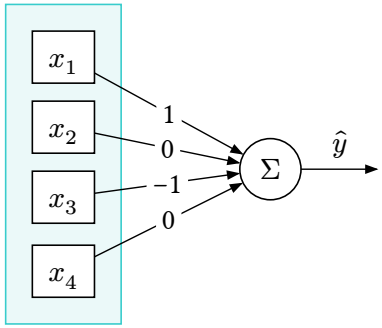


Figure 14: Final trained perceptron model

Let's see how it performs. The test data is in Figure 15.

x_1	x_2	x_3	x_4	y
1	0	1	1	1
0	0	1	1	−1
1	1	1	1	−1
0	1	0	1	−1

Figure 15: A testing dataset for perceptron

#	Input				Weights				z	$\hat{y}$	y	Ok?
	x_1	x_2	x_3	x_4	w_1	w_2	w_3	w_4				
1	1	0	1	1	1	0	−1	0	0	1	1	+
2	0	0	1	1	1	0	−1	0	−1	−1	−1	+
3	1	1	1	1	1	0	−1	0	0	1	−1	-
4	0	1	0	1	1	0	−1	0	0	1	−1	-

Figure 16: Results for perceptron testing. *Note*: $z = \sum_j^n w_j \cdot x_j$

The accuracy of the perceptron is only 50%...

3.2. Learning in multilayer model

For the learning of multilayer neural network model, it is needed to use a **backpropagation** algorithm. More on the algorithm see Russell & Norvig's book page 734.

The generalized multilayer network can be represented as in Figure 8.

The weight update process for the one hidden layer NN is the following:

1. Calculate the error:

$$E = y - \hat{y}$$

where y is the actual value, $\hat{y}$ - predicted

2. Update the weights for the hidden layer:
   a. Get the partial error:

$$\Delta_i = E \cdot \varphi'(z_j)$$

   where $\varphi'(z_j)$ is the derivative for the activation function, e.g. sigmoid will be calculated as $\varphi' = \varphi(1-\varphi)$, z_j is the weighted sum with of the neuron
   b. Update the weight:

$$w_{ji}{}' = w_{ji} + \alpha \cdot a_j \cdot \Delta_i$$

   where w_{ji} is the weight of the j-th neuron of the previous layer to the i-th neuron of the next layer. α is the learning rate.

3. Update the weights for the input layer:
   a. Get the partial error:

$$\Delta_j = \varphi'(z_j) \cdot \sum_i w_{ji} \cdot \Delta_i$$

   b. Update the weight:

$$w_{kj}{}' = w_{kj} + \alpha \cdot x_k \cdot \Delta_j$$

   where x_k is the input value

4. Repeat the process for the number of iterations

3.3. Multilayer learning example

Consider the problem of **XOR** operation, where one weight was modified to create an error:

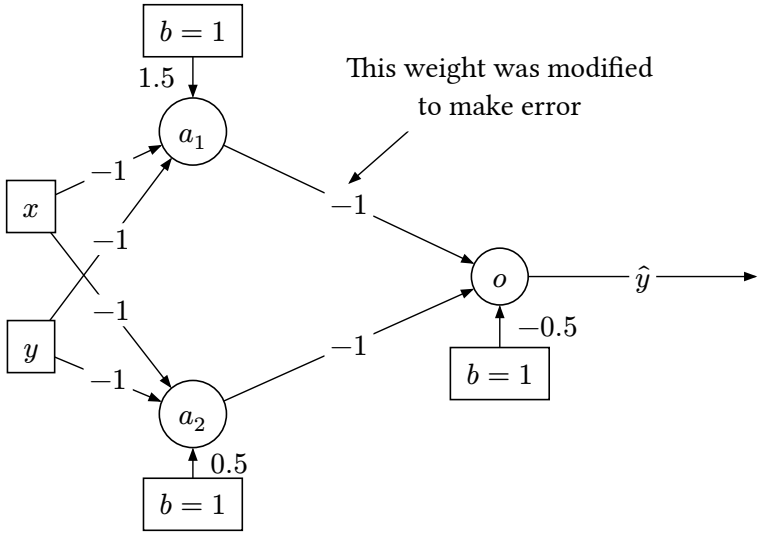


Figure 17: A multilayer network solution for **XOR** problem with one modified weight.

Note: Using a sigmoid function

1. Calculate the predicted value:

$$z_1 = 1 \cdot 1.5 + 1 \cdot (-1) + 1 \cdot (-1) = -0.5$$

$$a_1 = \frac{1}{1 + e^{0.5}} = 0.38$$

$$z_2 = 1 \cdot 0.5 + 1 \cdot (-1) + 1 \cdot (-1) = -1.5$$

$$a_2 = \frac{1}{1 + e^{1.5}} = 0.18$$

$$z_3 = (-0.5) \cdot 1 + (-1) \cdot 0.38 + (-1) \cdot 0.18 = -1.06$$

$$o = \frac{1}{1 + e^{1.06}} = 0.26$$

2. There is an error, since $y = 0$, but we predicted $\hat{y} = 0.26$, so we need to update the weights. Calculate the error:

$$E = y - \hat{y} = 0 - 0.26 = -0.26$$

3. Get the partial error for the output layer:

$$\Delta_o = E \cdot g'(o) = -0.26 \cdot 0.26(1 - 0.26) = -0.05$$

4. Update weights for the hidden layer:
   a. ${w_{a_1,o}}' = w_{a_1,o} + \alpha \cdot a_1 \cdot \Delta_o = -1 + 1 \cdot 0.38 \cdot (-0.05) = -1.02$
   b. ${w_{a_2,o}}' = w_{a_2,o} + \alpha \cdot a_2 \cdot \Delta_o = -1 + 1 \cdot 0.18 \cdot (-0.05) = -1.01$
   c. ${w_{b,o}}' = w_{b,o} + \alpha \cdot b \cdot \Delta_o = -0.5 + 1 \cdot 1 \cdot (-0.05) = -0.55$

5. Get partial errors for the hidden layer:
   a. $\Delta_{a_1} = g'(a_1) \cdot w_{a_1,o} \cdot \Delta_o = a_1(1 - a_1) \cdot w_{a_1,o} \cdot \Delta_o = 0.38(1 - 0.38) \cdot -1 \cdot (-0.05) = 0.012$
      - We have only one edge to the output, therefore no summation!
   b. $\Delta_{a_2} = g'(a_2) \cdot w_{a_2,o} \cdot \Delta_o = a_2(1 - a_2) \cdot w_{a_2,o} \cdot \Delta_o = 0.18(1 - 0.18) \cdot -1 \cdot (-0.05) = 0.007$

6. Update the input to hidden layer weights:
   a. ${w_{x,a_1}}' = w_{x,a_1} + \alpha \cdot x \cdot \Delta_{a_1} = -1 + 1 \cdot 1 \cdot 0.012 = -0.988$
   b. ${w_{x,a_2}}' = w_{x,a_2} + \alpha \cdot x \cdot \Delta_{a_2} = -1 + 1 \cdot 1 \cdot 0.007 = -0.993$
   c. ${w_{y,a_1}}' = w_{y,a_1} + \alpha \cdot y \cdot \Delta_{a_1} = -1 + 1 \cdot 1 \cdot 0.012 = -0.988$
   d. ${w_{y,a_2}}' = w_{y,a_2} + \alpha \cdot y \cdot \Delta_{a_2} = -1 + 1 \cdot 1 \cdot 0.007 = -0.993$
   e. ${w_{b,a_1}}' = w_{b,a_1} + \alpha \cdot b \cdot \Delta_{a_1} = 1.5 + 1 \cdot 1 \cdot 0.012 = 1.512$
   f. ${w_{b,a_2}}' = w_{b,a_2} + \alpha \cdot b \cdot \Delta_{a_2} = 0.5 + 1 \cdot 1 \cdot 0.007 = 0.507$

The network after weight updates from the 1st example:

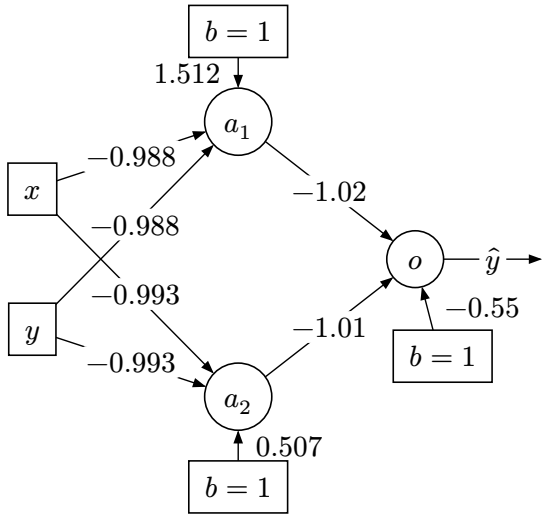


Figure 18: A multilayer network solution for **XOR** problem after weight update with 1 example

- The speed of learning is very slow, since weight corrections are minimal.
- The next step is to take the second example and repeat the weight update process once again until all 4 examples are met, after that the first iteration will be done.

4. Reinforcement Learning

Learning tasks:

1. Environment can be accessible or inaccessible(the agent needs to store the inner state)
2. The agent can start with the knowledge about the environment and actions, or this model should learned the same way as the utilities
3. Rewards can be given in the end or any other states
4. Rewards can be a part of the utility component(e.g. money, points in the game) that is tried to be maximized, or it could be a direction to the real utility, e.g. a “good move”.
5. The agent can be a passive learner, which follows the world and learns the utilities. Or it can be an active learner, which uses learned information and can create new problems to explore the environment

Reinforcement learning directions:

1. The agent learns the **utility-based function** which allows to choose the maximum result action. That is in utility-based agent.
2. The agent learns **action-utility-based function** which gives an expected utility, if the action chosen in the current state. That is **Q-learning**.
3. The reflex agent learns **policy** that maps states to actions.

5. Passive learning in accessible environment

5.1. Conditions

- We use a **naive** approach to update the utilities
- The game is the same 4x3 stochastic environment as used before(Figure 19).
- $R(s) = -0.04$

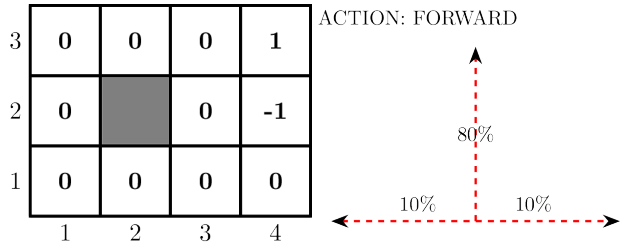


Figure 19: The environment starting state and the transition model

5.2. Starting policy

Let's imagine that the agent uses the following policy π from the start to learn utility function $U^{\pi}(s)$:

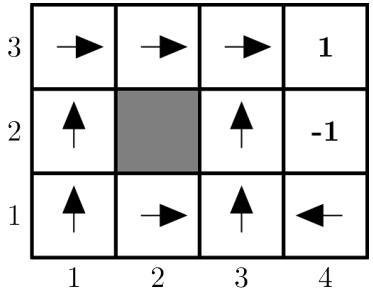


Figure 20: Starting policy

5.3. Training examples

To learn the new policy, the agent needs some examples. We will use the following 5 sequences of actions in the table as the learning examples. These sequences were generated by using the generated policy π before.

#	Sequence
1	$(1,1) \rightarrow (1,2) \rightarrow (1,3) \rightarrow (1,2) \rightarrow (1,3) \rightarrow (2,3) \rightarrow (3,3) \rightarrow (4,3)$
2	$(1,1) \rightarrow (1,2) \rightarrow (1,3) \rightarrow (2,3) \rightarrow (3,3) \rightarrow (3,2) \rightarrow (3,3) \rightarrow (4,3)$
3	$(1,1) \rightarrow (2,1) \rightarrow (3,1) \rightarrow (3,2) \rightarrow (4,2)$
4	$(1,1) \rightarrow (2,1) \rightarrow (3,1) \rightarrow (4,1) \rightarrow (3,1) \rightarrow (3,2) \rightarrow (3,3) \rightarrow (4,3)$
5	$(1,1) \rightarrow (2,1) \rightarrow (3,1) \rightarrow (2,1) \rightarrow (3,1) \rightarrow (4,1) \rightarrow (4,2)$

5.4. Naive approach

- We will go backwards through the sequence, each time adding the $R(s)$ to the final score.
- If the action was met several times, we take an average of it
- We go through each training example one by one

1. $(1,1)_{U=0.72} \rightarrow (1,2)_{U=0.76} \rightarrow (1,3)_{U=0.80} \rightarrow (1,2)_{U=0.84} \rightarrow (1,3)_{U=0.88} \rightarrow (2,3)_{U=0.92} \rightarrow (3,3)_{U=0.96} \rightarrow (4,3)_{U=1}$
   - Two states are repeated, therefore we use an average value:
     - $U(1,2) = \frac{0.76+0.84}{2} = 0.80$
     - $U(1,3) = \frac{0.88+0.80}{2} = 0.84$

3	**0.84**	**0.92**	**0.96**	**1**
2	**0.80**		**0**	**-1**
1	**0.72**	**0**	**0**	**0**
	1	2	3	4

Figure 21: First example results

2. $(1,1)_{U=0.72} \rightarrow (1,2)_{U=0.76} \rightarrow (1,3)_{U=0.80} \rightarrow (2,3)_{U=0.84} \rightarrow (3,3)_{U=0.88} \rightarrow (3,2)_{U=0.92} \rightarrow (3,3)_{U=0.96} \rightarrow (4,3)_{U=1}$
   - Utilities from previous calculation are also used to get averaged:
     - $U(1,1) = 0.72$
     - $U(1,2) = \frac{0.76+0.80}{2} = 0.78$
     - $U(1,3) = \frac{0.84+0.80}{2} = 0.82$
     - $U(2,3) = \frac{0.84+0.92}{2} = 0.88$
     - $U(3,3) = \frac{(\frac{0.88+0.96}{2})+0.96}{2} = 0.94$

3	**0.82**	**0.88**	**0.94**	**1**
2	**0.78**		**0.92**	**-1**
1	**0.72**	**0**	**0**	**0**
	1	2	3	4

Figure 22: Second example results

3. $(1,1)_{U=-1.16} \rightarrow (2_1)_{U=-1.12} \rightarrow (3,1)_{U=-1.08} \rightarrow (3,2)_{U=-1.04} \rightarrow (4,2)_{U=-1}$
   - We will omit average calculations from previous examples

3	**0.82**	**0.88**	**0.94**	**1**
2	**0.78**		**-0.06**	**-1**
1	**-0.22**	**-1.12**	**-1.08**	**0**
	1	2	3	4

Figure 23: Third example results

4. $(1,1)_{U=0.72} \rightarrow (2,1)_{U=0.76} \rightarrow (3,1)_{U=0.80} \rightarrow (4,1)_{U=0.84} \rightarrow (3,1)_{U=0.88} \rightarrow (3,2)_{U=0.92} \rightarrow (3,3)_{U=0.96} \rightarrow (4,3)_{U=1}$

3	0.82	0.88	0.95	1
2	0.78		0.43	-1
1	0.25	-0.18	-0.12	0.84
	1	2	3	4

Figure 24: Fourth example results

5. $(1,1)_{U=-1.24} \rightarrow (2,1)_{U=-1.20} \rightarrow (3,1)_{U=-1.16} \rightarrow (2,1)_{U=-1.12} \rightarrow (3,1)_{U=-1.08} \rightarrow (4,1)_{U=-1.04} \rightarrow (4,2)_{U=-1}$

3	0.82	0.88	0.95	1
2	0.78		0.43	-1
1	-0.495	-0.67	-0.62	-0.1
	1	2	3	4

Figure 25: Fifth example results

After 5 examples, we get the end policy that is very close to the optimal policy that was found in the complex decision works.

Take into account that (4, 1) is not completely solved and it is indifferent of choosing either of directions. More examples can make it more decisive.

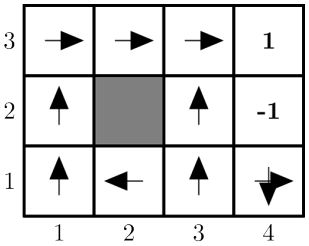


Figure 26: The policy after 5 examples

> **Note**
>
> The utility is defined to be the expected sum of (discounted) rewards obtained if policy π is followed as in equation:
>
> $$U^{\pi}(s) = \mathbb{E}\left[\sum_{t=0}^{\infty} \alpha^t R(S_t)\right]$$

5.5. Direct utility estimation

Direct utility estimation succeeds in reducing the reinforcement learning problem to an inductive learning problem, about which much is known.

The utility values obey the Bellman equations for a fixed policy:

$$U^{\pi}(s) = R(s) + \alpha \cdot \sum_{s'} P(s' \mid s, \pi(s)) \cdot U^{\pi}(s')$$

where $P(s' \mid s, \pi(s))$ can also be represented as the transition matrix $M_{s,s'}$

The transition model can also be represented as a graph:

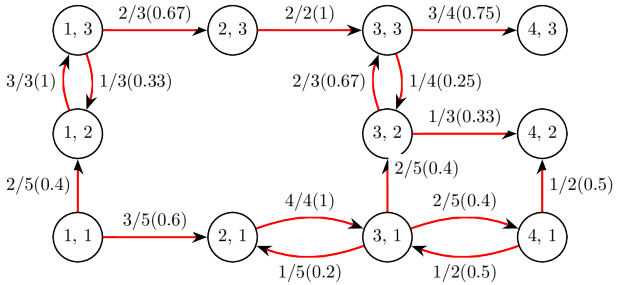


Figure 27: A transition model for the 4x3 world problem

It allows to get the following system of equations:

$$\begin{cases} U(1,1) = -0.04 + 0.4 \cdot U(1,2) + 0.6 \cdot U(2,1) \\ U(1,2) = -0.04 + U(1,3) \\ U(1,3) = -0.04 + 0.33 \cdot U(1,2) + 0.67 \cdot U(2,3) \\ U(2,1) = -0.04 + U(3,1) \\ U(2,3) = -0.04 + U(3,3) \\ U(3,1) = -0.04 + 0.2 \cdot U(2,1) + 0.4 \cdot U(3,2) \\ +0.4 \cdot U(4,1) \\ U(3,2) = -0.04 + 0.67 \cdot U(3,3) + 0.33 \cdot U(4,2) \\ U(3,3) = -0.04 + 0.25 \cdot U(3,2) + 0.75 \cdot U(4,3) \\ U(4,1) = -0.04 + 0.5 \cdot U(3,1) + 0.5 \cdot U(4,2) \\ U(4,2) = -1 \\ U(4,3) = 1 \end{cases}$$

The system can be solved, for example, with the *Gauss elimination method.*

5.6. Temporal difference learning

- The main idea is to limit local limits of each transition
- It is described with:

$$U^{\pi}(s) = U^{\pi}(s) + \alpha(R(s) + U^{\pi}(s') - U^{\pi}(s))$$

where α is the learning rate

Let's try this approach with 5 examples and $\alpha = 0.5$

1. $(1,1) \rightarrow (1,2) \rightarrow (1,3) \rightarrow (1,2) \rightarrow (1,3) \rightarrow (2,3) \rightarrow (3,3) \rightarrow (4,3)$

$$U(3,3) = 0 + 0.5 \cdot (-0.04 + 1 - 0) = 0.48$$

$$U(2,3) = 0 + 0.5 \cdot (-0.04 + 0.48 - 0) = 0.22$$

$$U(1,3) = 0 + 0.5 \cdot (-0.04 + 0.22 - 0) = 0.09$$

Previous value of $U(1,3)$

$$U(1,2) = 0 + 0.5 \cdot (-0.04 + 0.09 - 0) = 0.025$$

$$U(1,3) = 0.09 + 0.5 \cdot (-0.04 + 0.025 - 0.09) = 0.0375$$

$$U(1,2) = 0.025 + 0.5 \cdot (-0.04 + 0.0375 - 0.025) = 0.0112$$

$$U(1,1) = 0 + 0.5 \cdot (-0.04 + 0.0112 - 0) = -0.0144$$

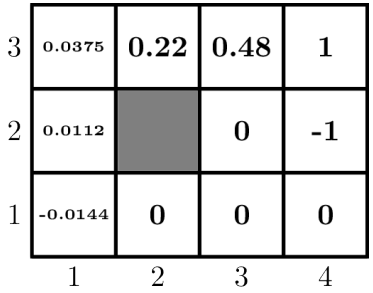


Figure 28: Results after 1st example

2. $(1,1) \to (1,2) \to (1,3) \to (2,3) \to (3,3) \to (3,2) \to (3,3) \to (4,3)$

$$U(3,3) = 0.48 + 0.5 \cdot (-0.04 + 1 - 0.48) = 0.72$$

$$U(3,2) = 0 + 0.5 \cdot (-0.04 + 0.72 - 0) = 0.34$$

$$U(3,3) = 0.72 + 0.5 \cdot (-0.04 + 0.34 - 0.72) = 0.51$$

$$U(2,3) = 0.22 + 0.5 \cdot (-0.04 + 0.51 - 0.22) = 0.345$$

$$U(1,3) = 0.0375 + \\ +0.5 \cdot (-0.04 + 0.345 - 0.0375) = 0.1713$$

$$U(1,2) = 0.0112 + \\ +0.5 \cdot (-0.04 + 0.1713 - 0.0112) = 0.0713$$

$$U(1,1) = -0.0144 + \\ +0.5 \cdot (-0.04 + 0.0713 - (-0.0144)) = 0.0084$$

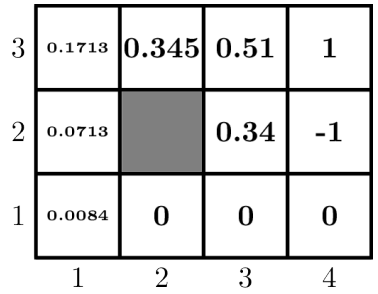


Figure 29: Results after 2nd example

Doing these operations for all five examples we receive the final utilities and the policy π:

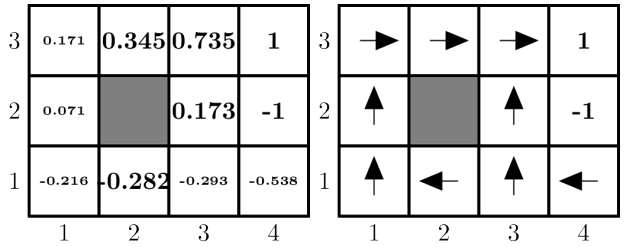


Figure 30: Utilities and policy after 5 examples

The training gave a good result in comparison to the optimal policy.

> **Info**
>
> This is the end of the course