It is the same situation in RL settings than in MDP settings?

Not. In RL settings we do not know the **transitions** neither the **rewards**.

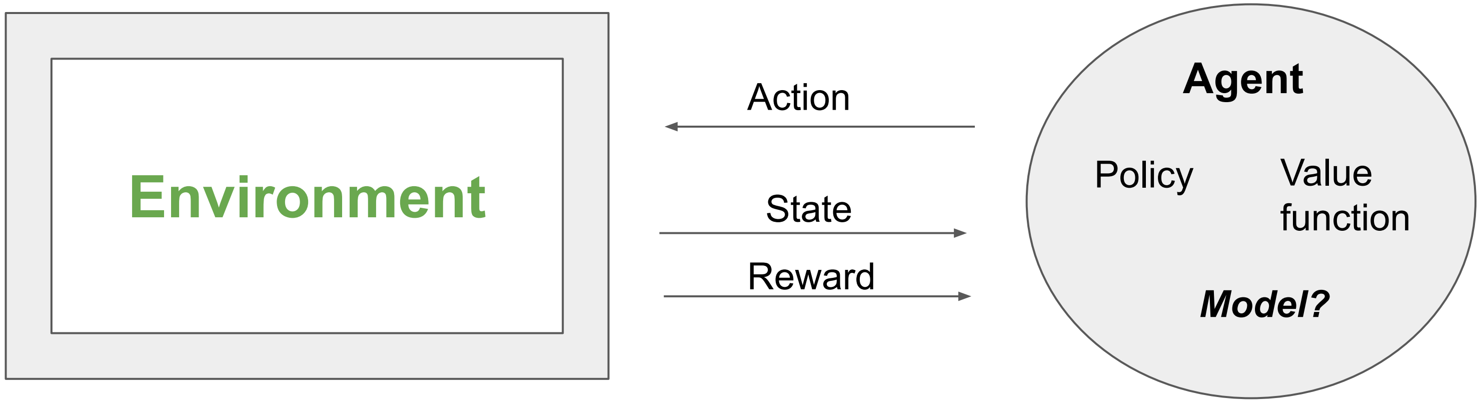
Therefore, in RL we have to experiment with the environment to figure out something about T and R. How? Maybe:

First:

$$\hat{T} = \frac{count(s,a,s')}{\sum_{s''} count(s,a,s'')} \text{ and } \hat{R} = \frac{\sum_{t=1}^{count(s,a,s')} R_t(s,a,s')}{count(s,a,s')}$$

Drawbacks? Any? … YES!

Remember the overall view of RL



Policy: how to choose an action.
Action: performed by the agent on the environment.
Value function: to valuate how well it is the current situation for the agent.
Model:* *an idealized representation of the real environment (might be useful for the goals of the agent).
State: the set of values that characterize the current situation in the environment.
Reward: the value returned by the environment to the agent in response to the consequences of the action performed.

Model Free vs Model Based approaches

Suppose we want to estimate the expectation of an unknown function f(X)

$$E[f(X)] = \sum_x p(x) f(x)$$

given K samples.

Model Based approaches tries to estimates the probability distribution of this function (**the model)** before estimating the expectation. So, in this case, works by sampling K points from a distribution P(x), then, estimates

$$\hat{p}(X) = \frac{count(x)}{K}$$

, and finally, the expectation of *f(X)* is calculated by $E[f(X)] \approx \sum \hat{p}(x) f(x)$

Model Free vs Model Based approaches

Model Free approaches estimates the expectation of f(X) as:

$$E[f(X)] \approx \frac{\sum_{i=1}^{K} f(x_i)}{K}$$

So, it is valued how each x_i contributed to the *f(X)* averaging.

How the Q-value iteration algorithm, previously exposed, can be rewritten now in this RL context using the sample-based approach?

Model Free approaches (**Q-learning**)

Recall from the Q-value-state iteration rule:

$$Q_*(s,a) = \sum_{s'} T(s,a,s')(R(s,a,s') + \gamma \max_{a'} Q_*(s',a'))$$

In RL we don’t know transitions neither rewards, so we have to try an action from a state and observe what happens. The above formula tells us how we can iteratively compute the Q values for the ***optimum policy*** by ***bootstrapping***. But, the summatory is unknown: samples ***(s, a, s’, r)*** are obtained one by one, Then, for each sample the above formula for compute Q*(s,a) is just :

$$R(s,a,s') + \gamma \max_{a'} Q_*(s',a')$$

Model Free approaches (**Q-learning**)

First option: taking k samples from the same pair (state, action):

sample$_1$ $R(s, a, s'_1) + \gamma \max_{a'} Q(s'_1, a')$

sample$_2$ $R(s, a, s'_2) + \gamma \max_{a'} Q(s'_2, a')$

sample$_k$ $R(s, a, s'_k) + \gamma \max_{a'} Q(s'_k, a')$

And then:

$$Q(s, a) = \frac{1}{k} \sum_{i=1}^{k} sample_i = \frac{1}{k} \sum_{i=1}^{k} (R(s, a, s'_i) + \gamma \max_{a'} Q(s'_i, a'))$$

Drawbacks? Any? … YES!

Model Free approaches (**Q-learning**)

Second option (EMA):

Using the **E**xponential **M**oving **A**verage: $\overline{x}_n = \frac{x_n + (1-\alpha)x_{n-1} + (1-\alpha)^2 x_{n-2} + \dots}{1 + (1-\alpha) + (1-\alpha)^2 + ..}$

But, given that 0 < α <= 1, then $1 + (1-\alpha) + (1-\alpha)^2 + .. = \frac{1}{\alpha}$. We get:

$$\overline{x}_n = \alpha x_n + (1-\alpha)\overline{x}_{n-1}$$

Then, using **EMA** for estimating the Q values, we get:

$$sample = R(s,a,s') + \gamma \max_{a'} Q_i(s',a')$$

$$\begin{aligned} Q_{i+1}(s,a) &= \alpha(R(s,a,s') + \gamma \max_{a'} Q_i(s',a')) + (1-\alpha)Q_i(s,a) \\ &= Q_i(s,a) + \alpha(R(s,a,s') + \gamma \max_{a'} Q_i(s',a') - Q_i(s,a)) \end{aligned}$$

Model Free approaches (**Q-learning**)

Example: Given a MDP as the one pictured in the classroom. Let suppose that we don't know transitions neither rewards. Let's suppose that we had collected the following samples in that order: *(s1, a1, s1, 1)*, *(s1, a1, s2, -1)*, *(s2, a2, s1, 1)* and using an alpha value of 0.75 and a gamma value of 0.5. Of course, Q(s,a) = 0 for step 0 for all s and all a. Calculate the values of Q for those samples. Recall that:

$$Q_{i+1}(s,a) = Q_i(s,a) + \alpha(R(s,a,s') + \gamma \max_{a'} Q_i(s',a') - Q_i(s,a))$$

or, equivalently:

$$Q_{i+1}(s,a) = \alpha(R(s,a,s') + \gamma \max_{a'} Q_i(s',a')) + (1-\alpha)Q_i(s,a)$$

Model Free approaches (**Q-learning**)

Algorithm to compute the Q values

```
Set $Q_0$ (s,a) arbitrarily (e.g. as 0 for all pairs state,action)
For i in range(number_steps):
     *Choose an action a (****)*
     Collect a sample: $sample = R(s,a,s') + \gamma \max_{a'} Q_i(s',a')$
     $Q_{i+1}(s,a) = \alpha(sample) + (1-\alpha)Q_i(s,a)$
                  $= Q_i(s,a) + \alpha(sample - Q_i(s,a))$
```

How to choose the value of α?

Bigger values makes the process unstable.
Smaller values slows down the convergence.

How to figure out in each step which action to execute?

The ε-greedy approach to select an action

The ε-greedy approach tries to balance between **exploration** and **exploitation**.

- **Exploration** tries to figure out other ways to act from the most performed.
- **Exploitation** tries to follow up the most successful actions found up to now.

This approach does the following:

  - With probability ε, sample a random action.
  - With probability 1 - ε, choose the best currently available action.

The value of ε should decay with time after certain step during training.

The previous algorithm for training using this ε-greedy approach, as the code line marked as (****) in the previous shown algorithm is known as:

Q-learning algorithm

Model Free approaches (**Q-learning**)

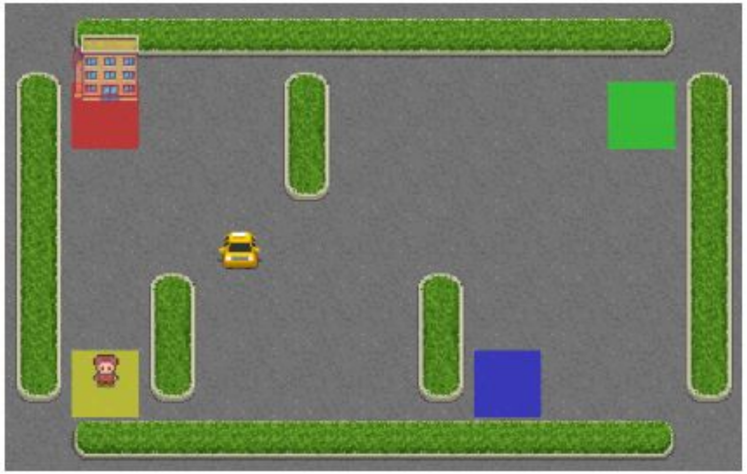
An easy example, but powerful: How to train an taxi from scratch to behave optimally without programming how to resolve its goals? **Q.learning!**

In the aula virtual you can read the Q-learning code example for this problem.
This scenario is drawn by the Gymnasium initiative driven by Open-AI to facilitate several examples for algorithms of RL to be applied to.

You can take a look at https://gymnasium.farama.org/index.html

Model Free approaches (**Q-learning**)

The taxi environment is shown in the following animation image (its features are deeply explained in the code link at the aula virtual)



Q-learning example: Learning to play Blackjack

Blackjack is a game whose goal is to obtain cards from a deck, the sum of whose numerical values be as much as possible without exceeding 21. An ace can count as 1 or 11 points, and all face cards count as 10. There is a dealer and a player. The game starts with two cards dealt to both the dealer and the player. The dealer has one card face up and one face down, while the player has both cards face up. A *natural* is a starting hand in which a player has exactly 21 (an ace and a face card), and in this case, the player wins unless the dealer also has 21. In the latter case, it's a draw (nobody wins or loses). If the player does not have a natural, it can request more cards, one by one (*hit*) until she decides to stop (*stick*) or it exceeds 21 (*go bust* and lose). If the player sticks, it is the dealer turn. Suppose that the dealer strategy is to stick on any sum of 17 or greater, and to hit otherwise. If the dealer goes bust, player wins; otherwise,the outcome of the game is determined by whose final sum is closer to 21 (description closely drawn from the book *Reinforcement Learning* by *Sutton and Barto*)

Q-learning example: Learning to play Blackjack

How to approach this game as an RL problem for the player?

States? Three values characterize each possible state: the sum of the cards from the player (**X**), the sum for the cards face up by the dealer (**Y**) and if the player has an usable ace or not (**Z**): **(X, Y, Z)** with **X** from 12 to 21, **Y** from 1 to 10 and **Z** binary (whether or not she holds and usable ace). How many possible states? *200*.

Actions? Two actions: hit (**0**), asking for one more card, or stick (**1**)

Rewards? **+1** if player wins, **-1** if player loses, **0** if player draws

— Read, discuss and understand the Colab Notebook for Q-learning Blackjack available in the aula virtual —

Model Free approaches (**limitations of Q-learning**)

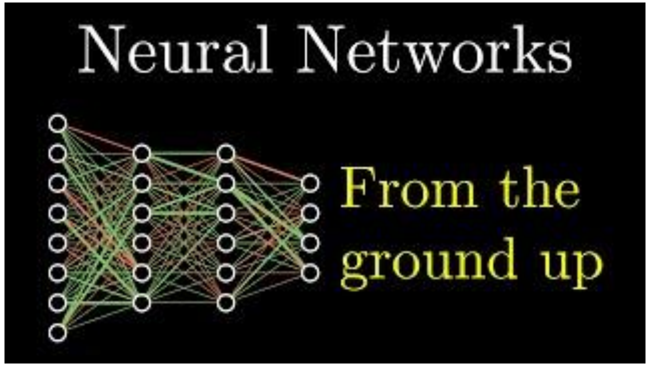
Q-learning is a great method for RL, but what could happen if the number of states and/or actions is really big? We might need a very big table for representing all the values for the Q values, and maybe this would be inefficient, or even impossible!

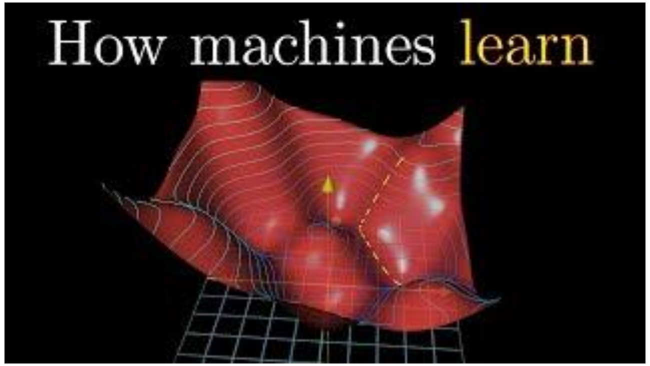
Even more, how Q-learning can deal with states representing continuous measures? Well, there is always the option to discretize the continuous values in sequential and no overlapping intervals, but some features of the environment could be lost.

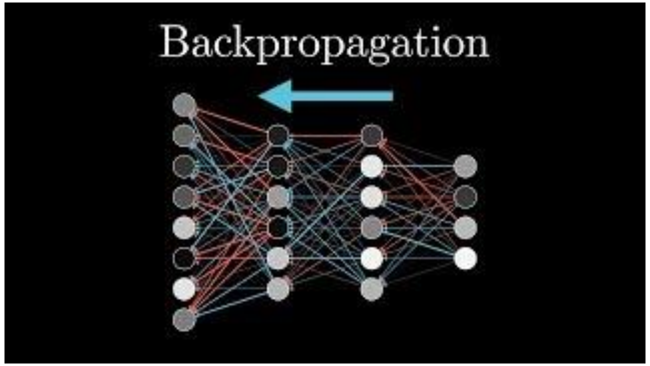
More drawbacks? Yes. If the number of states is big (being this number continuous or discrete) how you can guarantee that the experiences run by the RL agent visit every one of these states for each possible action? The most usual setting is that you can not guarantee that!

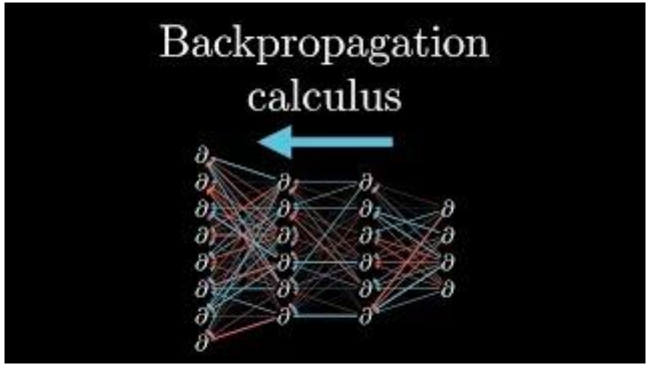
But, we could use another approach to deal with these issues: represent the states, and the whole process of Q-learning, as continuous functions that we want to approximate with computational techniques. **Advantages of this approach? A lot!**

Model Free approaches (**Neural network**)









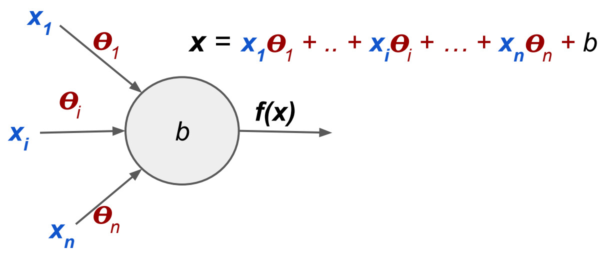
Model Free approaches (**Deep Q-learning**)

DQN (Deep Q-Network) is Q-learning + Artificial Neural Network

What is an Artificial Neural Network (ANN)?

An ANN is a computational structure composed of several layers of neurons.

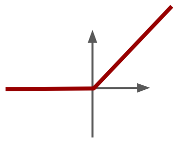
Each neuron receives some weighted (θ) linear combination of input data and produces a new datum according to its function of activation. All neurons in a layer usually apply the same activation function.



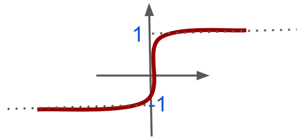
$$x = x_1\theta_1 + .. + x_i\theta_i + \dots + x_n\theta_n + b$$

Examples of neuron ***activation functions***:

ReLU: $f(x) = \max(0,x) = \begin{cases} 0 \text{ if } x < 0 \\ x \text{ otherwise} \end{cases}$



Tanh: $f(x) = \tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$

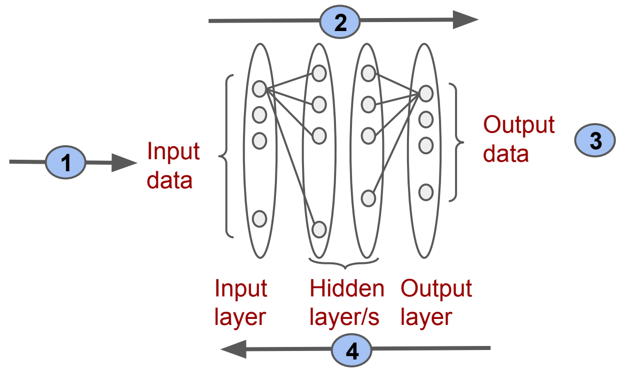


Model Free approaches (**Deep Q-learning**)

The whole ANN **goal** is to approximate an (usually) **unknown function** by iterating:

1. Provide input data to the first layer of neurons,
2. Move data from the neurons of the input layer to the neurons of the output layer. Each neuron uses its activation function given the weighted linear combination of its inputs to calculate its output value,
3. From the output data in the last layer of neurons compute the ***loss*** (i.e. comparison of this output data with the correct result),
4. And, finally, minimise the loss (e.g. using a ***gradient descent*** method) by changing the weights of the neurons from the output to the input layer ***(backpropagation)***

Neural network (NN)



What is a **Deep** ANN (DNN)? An ANN with 3 or more layers (e.g. the one above).

Model Free approaches (**Deep Q-learning**)

The **loss**, $\mathcal{L}$, enable to measure how far the network output is from the correct output.

So, we want to minimize this loss by adjusting the weight parameters of all neurons in the NN:

$$\boldsymbol{\theta}^* = \arg\min_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta})$$

The loss has to be differentiable, because the modifications on the weights will be carried out by using gradients of this loss.

A very usual definition of loss is the *Mean Square Error* (MSE) over a batch of M inputs:

$$\mathcal{L}(\theta) = \frac{1}{M} \sum_{i=1}^{M} (\underbrace{f(x_i)}_{\text{The true value of } f \text{ for the input } x_i} - \underbrace{\hat{f}(x_i;\theta)}_{\text{The output value of } f \text{ for the input } x_i \text{ provided by the NN}})^2$$

The **true** value of *f* for the input x_i

The **output value** of *f* for the input x_i provided by the NN

Model Free approaches (**Deep Q-learning**)

How to **minimize the loss**? A common *optimization technique* is the **gradient descent**.

The gradient of the loss function is: $\nabla_\theta \mathcal{L}(\theta) = \left(\frac{\partial(\theta)}{\partial\theta_1}, \dots \frac{\partial(\theta)}{\partial\theta_d} \right)$

(the vector in the space of all weights which points in the direction where the loss function is bigger)

As our aim is to minimize the loss function, we have to follow the ***negative gradient***!

There are three ways to sequentially update the weights using the negative gradients of the loss function (α, hyperparameter, is the **learning rate**, the "size" of the step to give in the direction of the negative gradient):

Batch gradient descent: *Big cost, low variance*

$$\theta \leftarrow \theta - \alpha \nabla_\theta \mathcal{L}(\theta|M), \text{M is the set of all samples}$$

Stochastic gradient descent: *Low cost, high variance from sample to sample*

$$\theta \leftarrow \theta - \alpha \nabla_\theta \mathcal{L}(\theta|d), \text{d is a sample}$$

Mini-batch gradient descent: *a balance between the above two*

$$\theta \leftarrow \theta - \alpha \nabla_\theta \mathcal{L}(\theta|B_i), B_1 \cup B_2 \cup \dots \cup B_n = M, \forall_{i,j} |B_i| = |B_j|$$

Model Free approaches (**Deep Q-learning**)

What is **backpropagation**? An ANN computes a compositional function: the weighted output of several activation functions as the input of another activation function, layer by layer up to the output layer). Then, the loss is calculated and the gradient descent to minimize the loss has to be applied to every weight in the network from the output to the input layer. The ***chain rule of derivatives*** can be applied in this backward pass for the compositional function of the ANN. Remember, the chain rule states that for *y = f(x)* and *z = g(y) =g(f(x))* then:

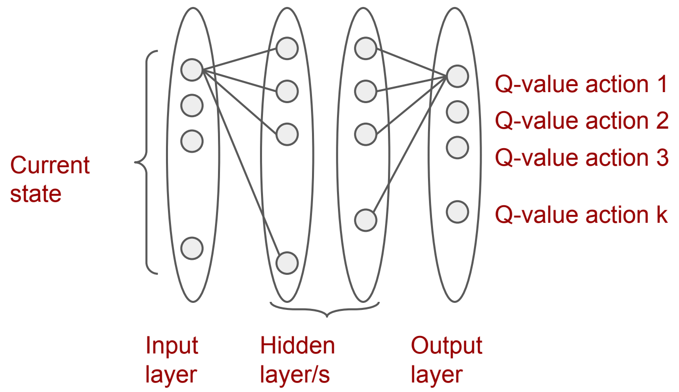
$$\nabla_x g = \left(\frac{\partial y}{\partial x}\right)^T \nabla_y z$$

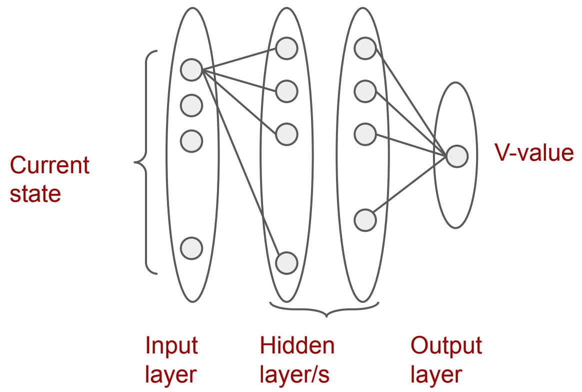
The gradients of a compositional function with respect to the inputs of a inner function is computed as a multiplication of the Jacobian of the inner function *g* and the gradient of the outer function with respect to its inputs.

The backpropagation algorithm uses automatic differentiation programming techniques to compute these gradients. It is usual that modern frameworks for computing with ANN (`pytorch`, `tensorflow`, etc) provide functions to automatically compute this backpropagation pass, hiding for the programmer its implementation details.

Model Free approaches (**Deep Q-learning**)

An ANN can be used to approximate a **policy function** or a **value function**: to map states (input data) to state-action pairs of Q values or to V values (output data)





Deep Q-network (DQN)

Model Free approaches (**Deep Q-learning**)

Remember, the **loss** function measures the differences between the true value and the output value provided by the ANN.

But in RL **we don't know the true values!** What can we do?

Let use as the approximate true value, the value known as the **target**: (as we just do in Q-learning)

$$r_t + \gamma \max_{a'} Q(s_{t+1}, a'; \theta)$$

Then, the loss function in a DQN is:

$$\mathcal{L}(\theta) = \left(r_t + \gamma \max_{a'} Q(s_{t+1}, a'; \theta) - Q(s_t, a'; \theta)\right)^2$$

Model Free approaches (**Deep Q-learning**)

However, RL with DNN is shown to be unstable. Why?

- There is a correlation between the Q-values and the target-values.
- In a DNN the target is fixed and does not change, in QDN the target value changes with every iteration during training.

How to turn the combination of RL with DNN stable? By using two techniques:

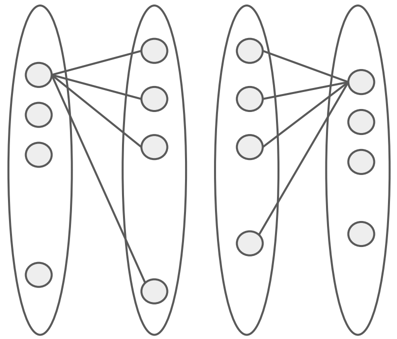
- *Experience play*. It is a technique to stabilize the training of the Q-function by allowing the agent to “replay” again past experiences (from s_i, a_j was obtained r_k, s_t). These past experiences are stored in a ***replay memory (RM).*** During the process of training, the agent randomly samples a batch of experiences from the RM to update the Q-function.

- *Target network*. It is a copy of the DQN, which is used to approximate the Q-function

Model Free approaches (**Deep Q-learning**)

Q-Network

Current state



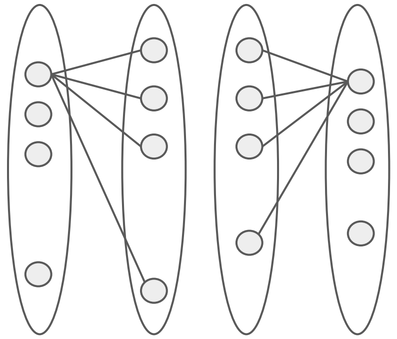
$Q(s,a;\boldsymbol{\theta})$

$$Loss = (r + \gamma \max_{a'} Q_T(s',a';\theta_i^-) - Q(s,a;\theta_i))$$

$\boldsymbol{\theta}$ updates $\boldsymbol{\theta}^-$ every C (hyperparameter) time steps

Target Network

Current state



$Q_T(s,a;\boldsymbol{\theta}^-)$

Model Free approaches (**Deep Q-learning**)

Therefore, in the Q-Network the internal weights of the neurons are modified by backpropagation the loss subject to the gradients of the activation function in each neuron:

$$\theta \leftarrow \theta + \alpha(r + \gamma \max_{a'} Q_T(s', a'; \theta^-) - Q(s, a; \theta))\nabla_\theta Q(s, a; \theta)$$

The Target-Network stabilizes the training process of the DQN because the most recent changes of the weights in the Q-Network are only applied to the Target-Network every C time steps.

In the aula virtual you can get a Google Colab tutorial on how to control a *cart-pole scenario* using a DQN. Do, first!, the also available brief tutorial on *pytorch*, one of the most popular packages to help in the programming of DNN.

Deep Q-learning: The Cart-pole example

Each state of the cart-pole is composed of 4 data:

(Cart-position [*-4.8,4.8*],
Cart-Velocity [*-inf, +inf*],
Pole-angle [*-0.418, 0.418*],
Pole-angular-velocity [*-inf, +inf*])

The agent receives a reward of **+1** every time step that the pole remains upright on the cart -> Goal is to maintain the pole upright as much time steps as possible.
There are two actions: **0**, push the cart to the left, or **1**, push the cart to the right

Deep Q-learning: Cart-pole *pytorch* code examples

```
class DQN(nn.Module):

    def __init__(self, n_observations, n_actions, hidden_size):
        super(DQN, self).__init__()
        self.layer1 = nn.Linear(n_observations, hidden_size)
        self.layer2 = nn.Linear(hidden_size, hidden_size)
        self.layer3 = nn.Linear(hidden_size, n_actions)

    def forward(self, x):
        x = F.relu(self.layer1(x))
        x = F.relu(self.layer2(x))
        return self.layer3(x)
```

Some lines of code using the above definition:

```
policy_network = DQN(4, 2, 128).to(device)
target_network = DQN(4 ,2, 128).to(device)
# Updates the parameters of the target_network with the
#    parameters of the policy_network
target_network.load_state_dict(policy_network.state_dict())
```

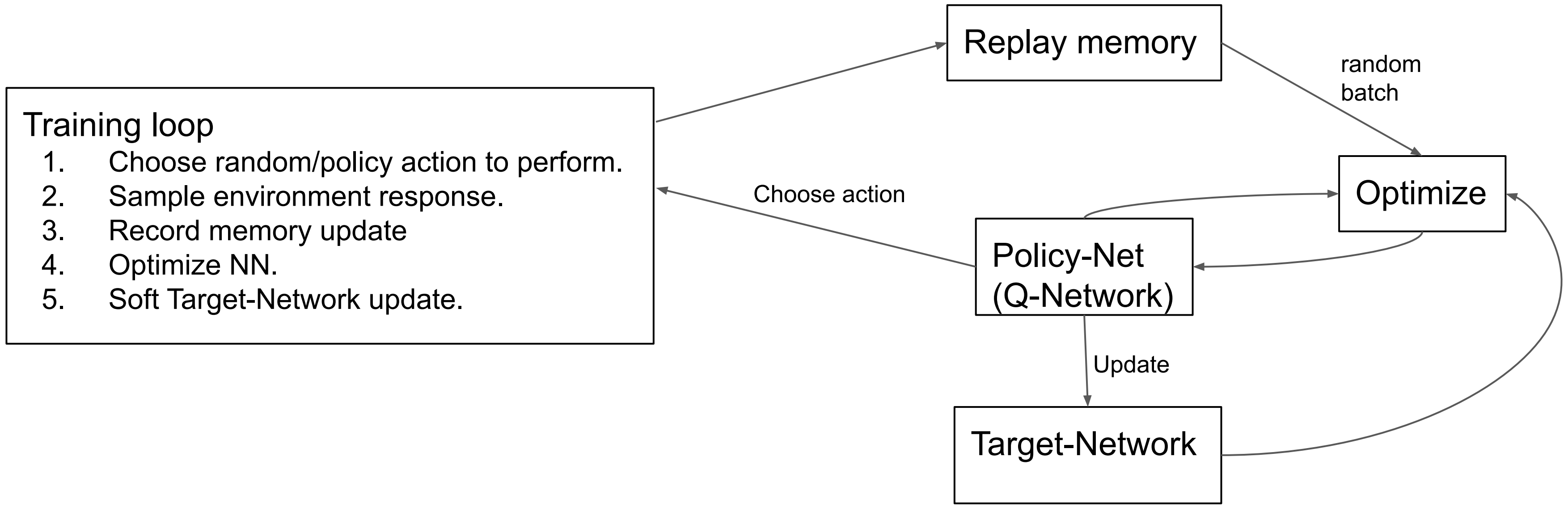
```
Transition = namedtuple('Transition', ('state', 'action', 'next_state', 'reward'))

class ReplayMemory(object):

    def __init__(self, capacity):
        self.memory = deque([], maxlen=capacity)

    def push(self, *args):
        self.memory.append(Transition(*args))

    def sample(self, batch_size):
        return random.sample(self.memory, batch_size)

    def __len__(self):
        return len(self.memory)
```

Some lines of code using the above definitions:

```
memory = ReplayMemory(10000)
memory.push(state, action, next_state, reward)
transition = memory.sample(128)
```

Deep Q-learning: The Cart-pole example

Abstract view of the software architecture of the model with DQN



Training loop

1. Choose random/policy action to perform.
2. Sample environment response.
3. Record memory update
4. Optimize NN.
5. Soft Target-Network update.

Deep Q-learning: The Cart-pole example

What is doing the optimizer?

Take a batch sample of the RM.
Compute $Q(s_t, a)$ with the Policy Network.
Compute $V(s_{t+1}) = \max_a Q(s_{t+1}, a)$ with the Target Network.
Calculate expected Q-value: reward + γ $V(S_{t+1})$
Compute ***Huber Loss*** between state-action-values and expected state-action-values-

The ***Huber Loss*** acts like a Mean Square Error (MSE) when the error is small and like the Mean Absolute Error when the error is large. This makes more robust to outliers (uncommon unexpected results) when the estimates of Q are very noisy.

Also, the technique of gradient clipping is performed to optimize the Policy Network: If the parameters of the Policy Network exceed 100 in magnitude, they will be set to 100 (or -100 for negative numbers).

Deep Q-learning: The Cart-pole example

The main program in the Cart-pole solution provided in the aula virtual does the following:

```
Set features of the environment and the values for the hyperparameters of the agent.
Create Policy-Network and Target-Network.
Set Optimizer.
Set Memory for RM.
for each episode:
    Start environment
    Set initial state.
    for not done: # max number of steps in a episode or pole falling down.
        Choose action
        Get experience (s, a, r, s')
        Set done True or False
        Put experience in RM
        Move to the next state (s')
        Optimize model.
        Replace $\theta'$ in Target-Network with those $\theta$ of Policy-Network.
```