50.039 Theory and Practice of Deep Learning

W11S2 Introduction to Reinforcement Learning

Matthieu De Mari

SINGAPORE UNIVERSITY OF TECHNOLOGY AND DESIGN

About this week (Week 11)

1. What is **Reinforcement Learning**?
2. What are the key ideas behind **reinforcement learning** and its **framework**?
3. What is the **exploration vs. exploitation tradeoff**?
4. How do we **train** an **RL agent** by exploring, then progressively exploiting?
5. What are some **advanced strategies** in **multi-arm bandit problems**?
6. What are the **Q** and **V functions** for a RL problem?
7. What is **Q-learning** and how can it be implemented in RL problems?

About this week (Week 11)

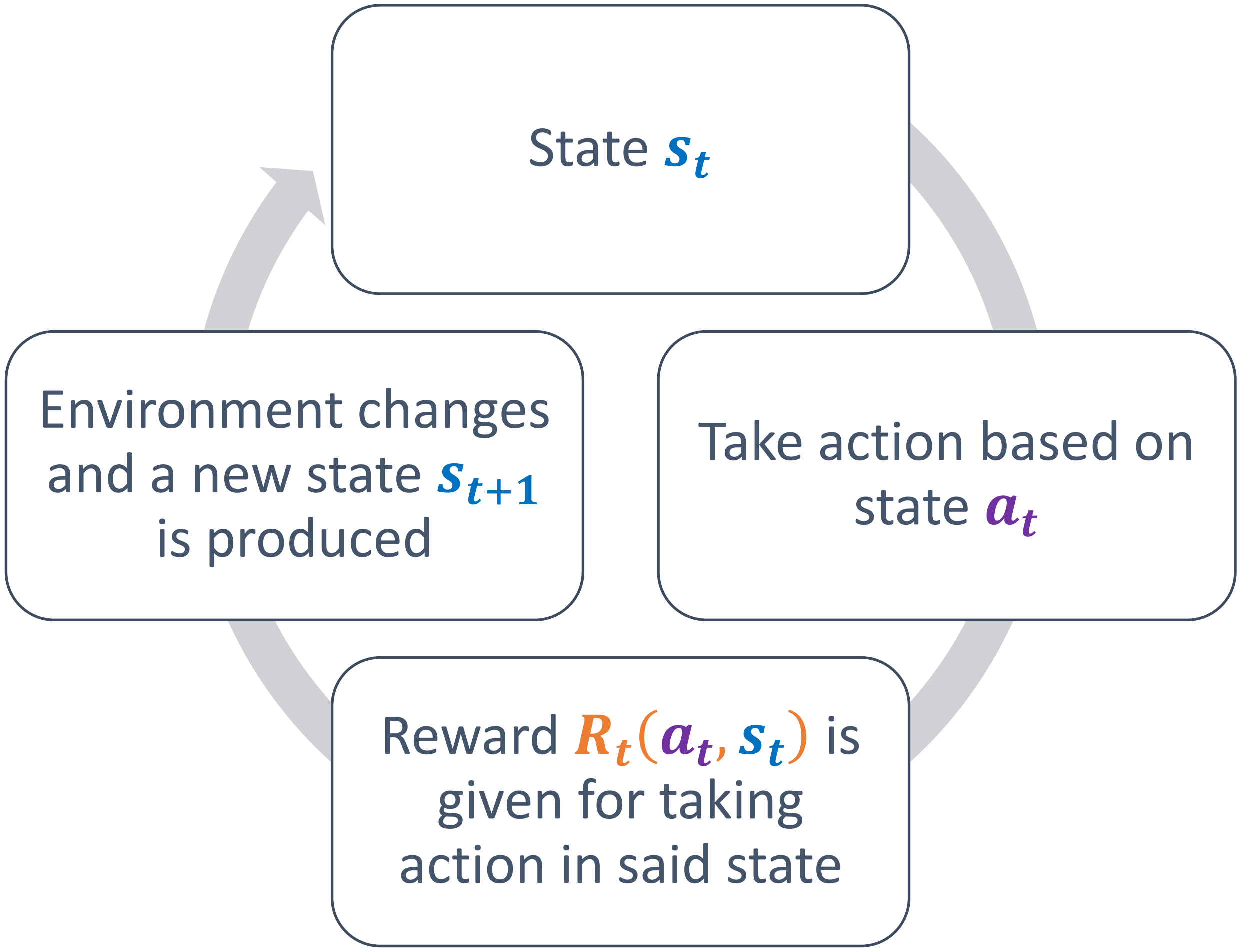
8. What is **Deep Q Learning**? And which problem does it address?
9. What is a **trainer** and a **main Q-network**? What is the interleaved training for Deep Q learning?
10. What are **actor-critic** learning methods? And which problems do these approaches address?
11. What are more advanced problems in RL?
    - Markov states
    - Partially observable environment
    - SARSA
    - Non-stationary problems

Reusing the RL formalism

Definition (agent):

In RL, we refer to the AI, as an agent. At each step, the agent:

- Looks at the current **state** s_t,
- Then takes an **action**, in this given state, a_t.
- The **action** has an effect, which is eventually measured in terms of reward, R_t.
- And a **new state** s_{t+1} is produced.



Restricted

Exploration and exploitation tradeoff

Definition (exploration and exploitation):

The phase during which, you try out things to acquire knowledge about the problem is called **exploration**.

The second phase, where you rely on your acquired knowledge, and play what you feel is the best move, is called **exploitation**.

Definition (exploration vs. exploitation tradeoff):

A good RL-based AI, needs to smartly combine **exploration** and **exploitation** phases.

- **Too much exploration?** You have wasted coins trying out bad machines.
- **No enough exploration?** You might end up choosing the wrong machine as the “best” one.

A second toy problem

The Maze problem

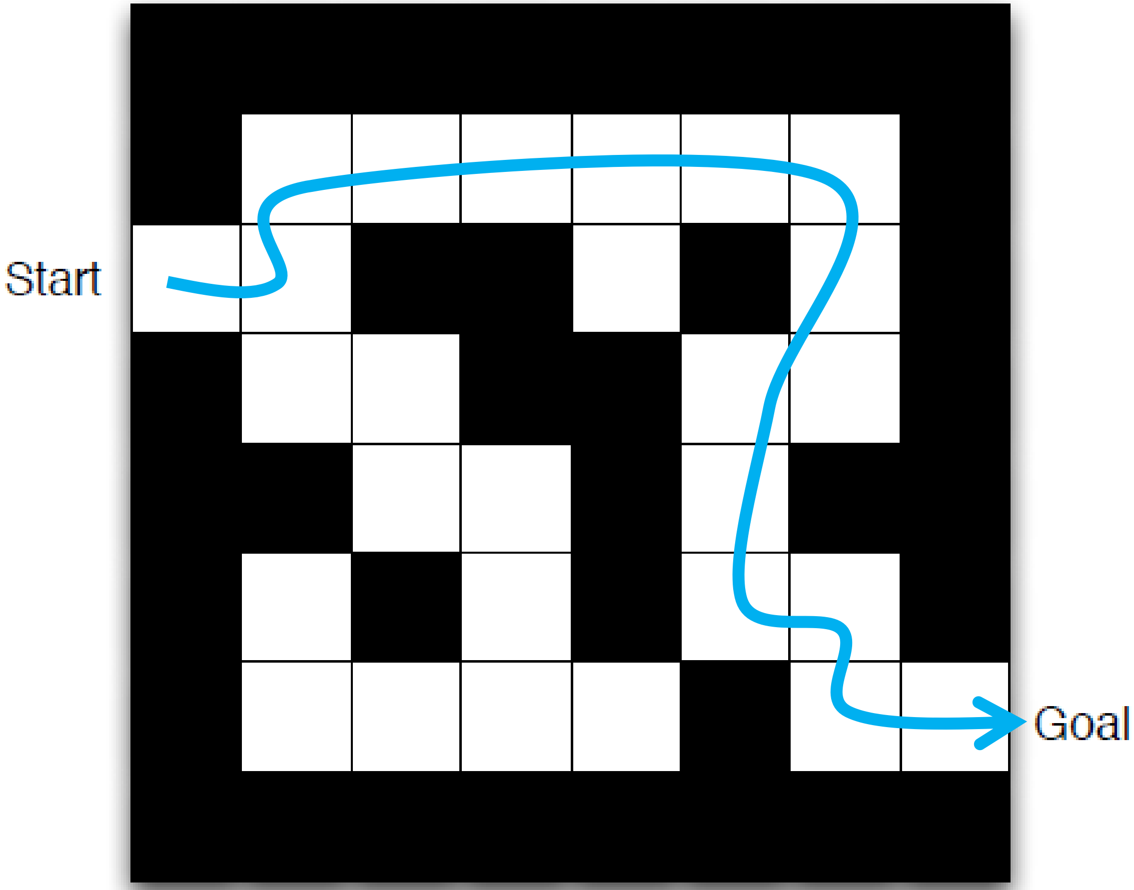
Restricted

The maze problem

Let us consider the maze problem, described on the right.

The objective is

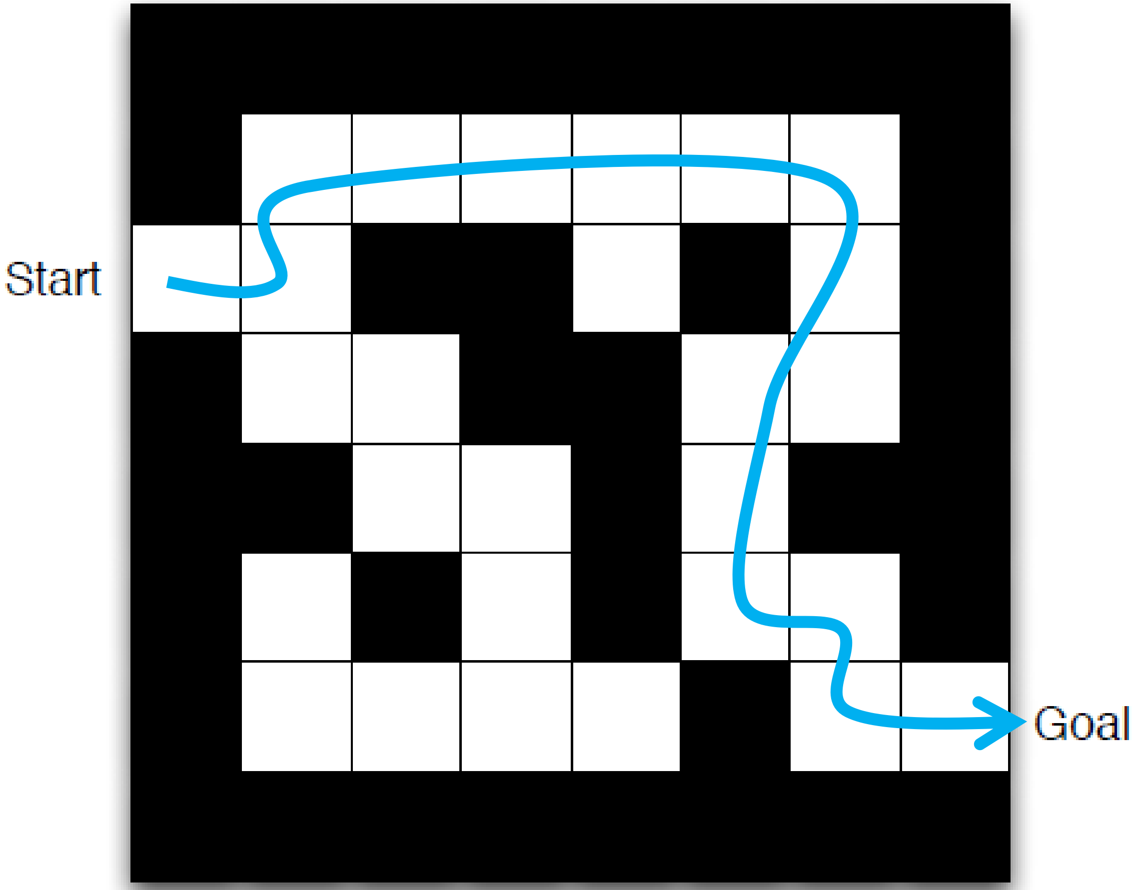
- to find the shortest path between the start and goal squares,
- without any knowledge of what the maze is beforehand.



The maze problem

Agent state:

- Our agent will start at the starting position, with coordinates **(3,1)**. This is the initial **state s_1**.



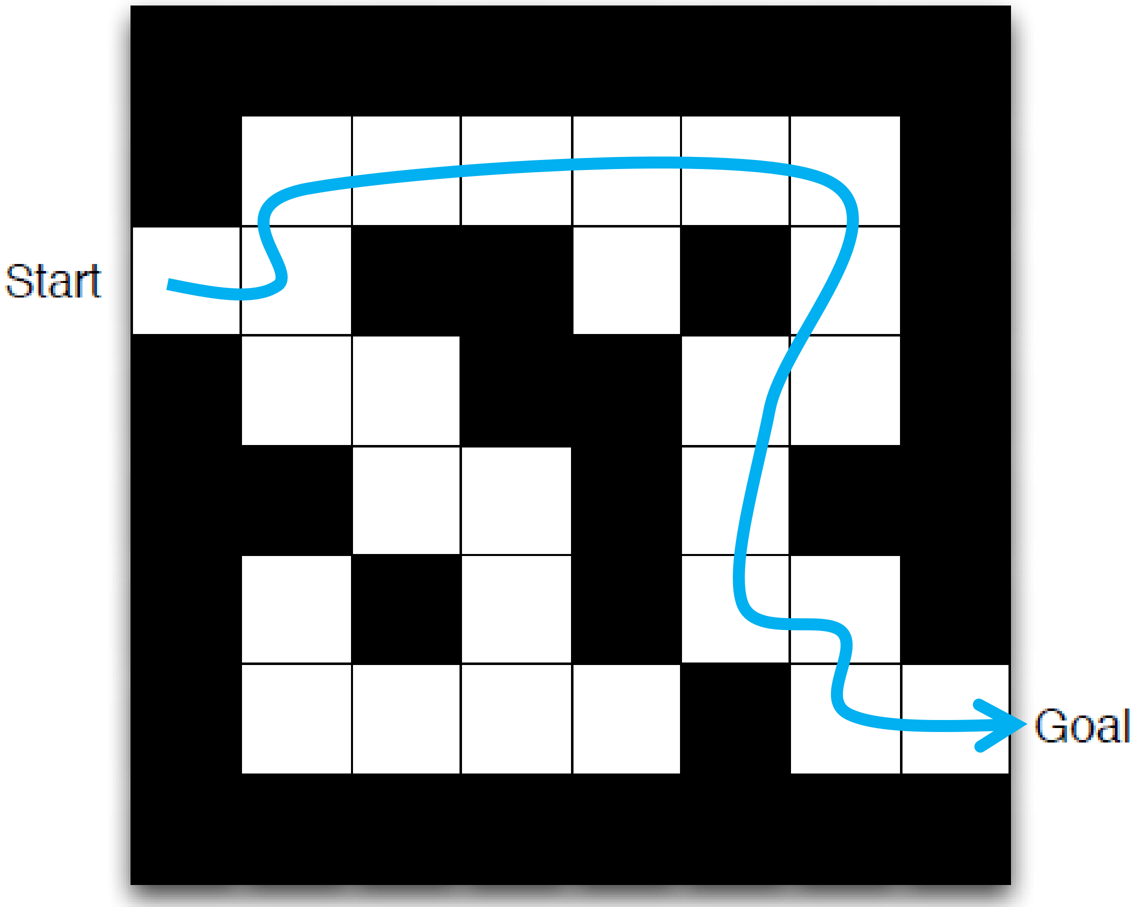
The maze problem

Agent state:

- Our agent will start at the starting position, with coordinates **(3,1)**. This is the initial **state s_1**.

Action space:

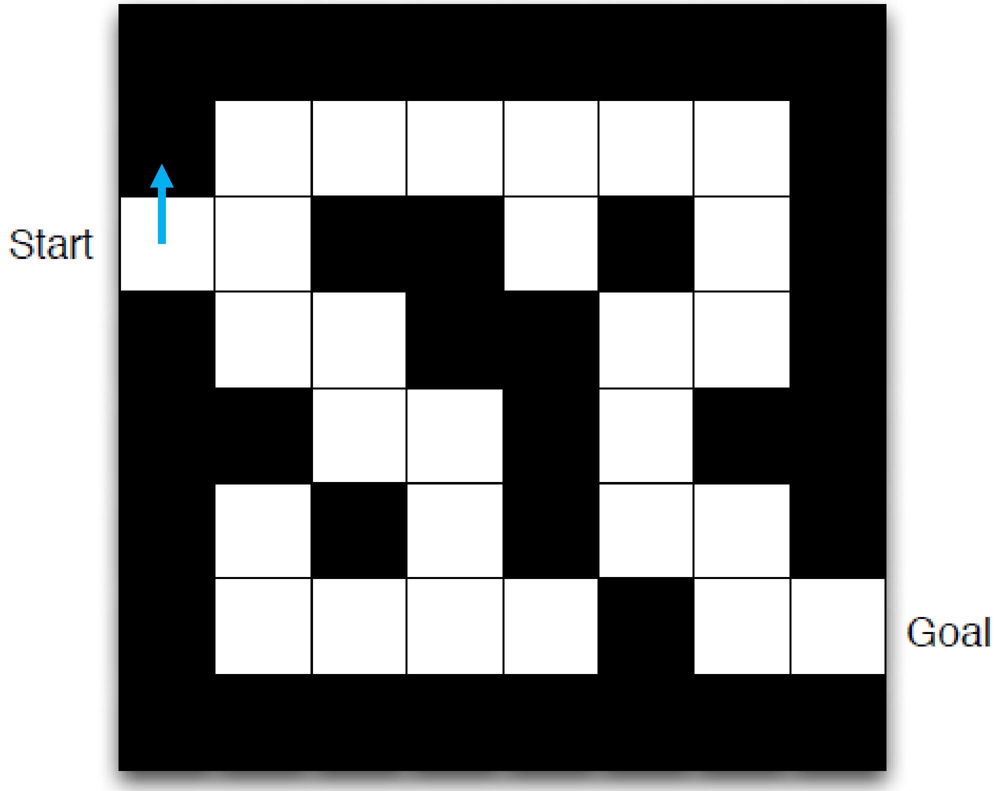
- In every position of the maze, four **actions** are possible $A = \{Go\ North,$ $Go\ South, Go\ East, Go\ West\}$



The maze problem

State update:

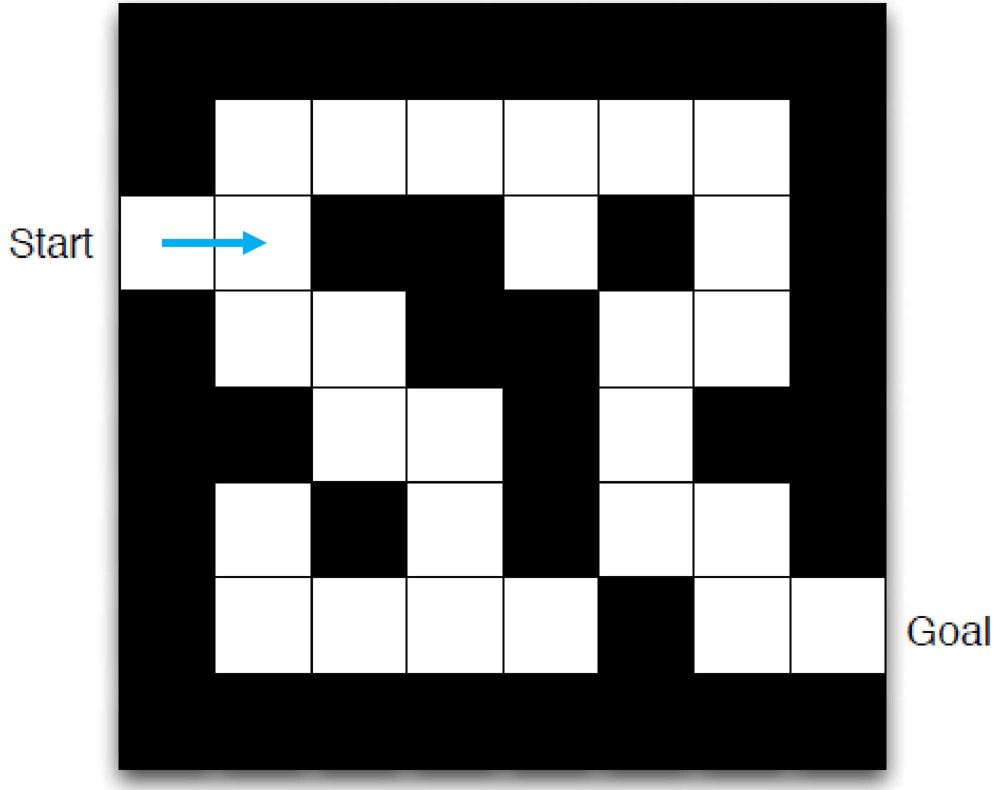
- For a given **state** s_t, and an **action** a_t, the **state** will be updated following these rules.
- If the **action** a_t moves the user in a wall or left of the start point, the **new state** s_{t+1} is the same as the **previous state** s_t.



The maze problem

State update:

- For a given **state** s_t, and an **action** a_t, the **state** will be updated following these rules.
- If the **action** a_t moves the user in a wall or left of the start point, the **new state** s_{t+1} is the same as the **previous state** s_t.
- Otherwise, move the use to next square and this becomes s_{t+1}.

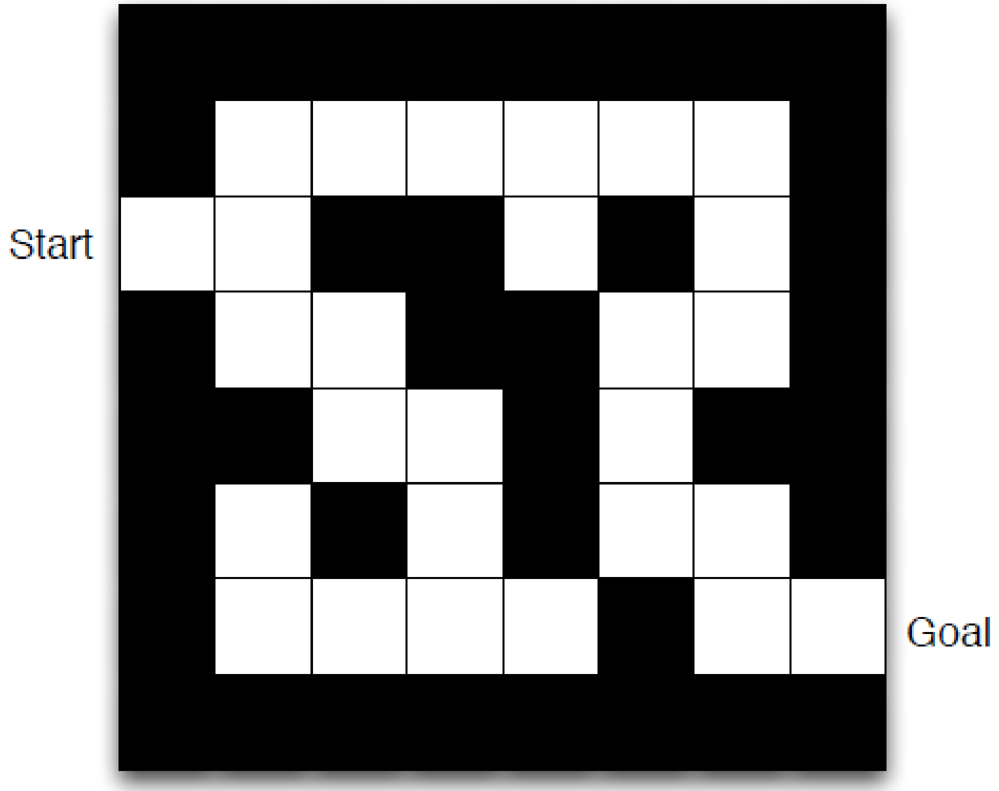


$$s_t = (3, 1) + a_t = East \rightarrow s_{t+1} = (3, 2)$$

The maze problem

Reward value:

- For every **state** and every **action** of the agent, the **reward** $\boldsymbol{R_t}$ is set to -1.
- The game stops when the agent reaches the Goal square, i.e. **state becoming (8, 7)**.



The maze problem

Reward value:

- For every **state** and every **action** of the agent, the **reward** R_t is set to -1.
- The game stops when the agent reaches the Goal square, i.e. **state becoming (8, 7)**.
- The **cumulated reward/gain** $G_t = \sum_t R_t$ will be equal to minus 1 times the number of steps taken to get from the start square to the goal square.
- Maximizing $G_t = \sum_t R_t$ is then strictly equivalent to **finding the shortest path out of the maze**.
- This will be our **RL framework** for this task.

Value function V

Definition (value function V):

The **value function V** is a prediction/estimation of the future cumulated reward/gain, if in state s_t at time t, for the policy π.

$$\forall t, \forall s_t, \qquad V_t^\pi(s_t) = E_\pi[R_t + R_{t+1} + R_{t+2} + \cdots |s_t]$$

In general, we add a parameter $\gamma \in [0,1]$, which gives more or less importance to the future or present rewards.

$$\forall t, \forall s_t, \qquad V_t^\pi(s_t) = E_\pi[R_t + \gamma R_{t+1} + \gamma^2 R_{t+2} \ldots |s_t]$$

Value function V

Definition (value function V):

The **value function V** is a prediction/estimation of the future cumulated reward/gain, if in state s_t at time t, for the policy π.

In general, we add a parameter $\gamma \in [0,1]$, which gives more or less importance to the future or present rewards, in the same way as regularization.

$$\forall t, \forall s_t, \qquad V_t^{\pi}(s_t) = E_{\pi}[R_t + \gamma R_{t+1} + \gamma^2 R_{t+2} \dots | s_t]$$

This function is used to **evaluate the goodness/badness of a state s_t**, and **can help to identify the best action to use in a given state**.

Optimization with value function V

The **value function V** can be used to rewrite our optimization problem, one step at a time.

Let us assume we are current at time t, in **state** $\boldsymbol{s_t}$.

The best **action** $\boldsymbol{a_t}$ to use in this current state will simply maximize

- The **immediate reward** we will get after playing this action, that is $\boldsymbol{R_t}$
- Plus the **expected reward** we will get in the future, if we end up in a new **state** $\boldsymbol{s_{t+1}}$ after playing **action** $\boldsymbol{a_t}$.

$$a_t = \arg\max_{a \in A} \left[R_t(a, s_t) + V_{t+1}^{\pi}(s_{t+1}) \right]$$

The state-action function Q

Definition (state-action function Q):

The **state-action function Q** is a function, which is used to quantify the goodness/badness of taking **action $\boldsymbol{a_t}$** in **state $\boldsymbol{s_t}$** at time t, according to our policy π.

It is closely related to the value function V, as it consists of our previous optimization function term, which was combining

- The **immediate reward** we will get after playing this action, that is $\boldsymbol{R_t}$
- Plus the **expected reward** we will get in the future, if we end up in a **new state $\boldsymbol{s_{t+1}}$** after playing **action $\boldsymbol{a_t}$**.

$$Q_t^\pi(a_t, s_t) = R_t(a_t, s_t) + V_{t+1}^\pi(s_{t+1})$$

On the policy, V and Q functions relationship

Theorem (Bellman principle of optimality):

The policy π, V and Q functions are closely related. For instance, we have a clear relationship between Q and V. The value of Q can be used to compute the value of V immediately, and vice-versa.

$$Q_t^{\pi}(a_t, s_t) = R_t(a, s_t) + V_{t+1}^{\pi}(s_{t+1})$$

The policy π, can then be derived from Q, as:

$$a_t = \pi_t(s_t) = \arg\max_{a \in A} Q_t^{\pi}(a, s_t)$$

Defining a Q-table

- In our problem, we have a finite number of **actions** and **states**.
- It is then simpler to write the Q function as a table, which is then referred to as the ***Q*-table**.
- Later on, we will let our agent explore the maze and update the values of the Q-table.
- Finding the converged Q-table, later gives us the best policy π to use in any state.

Initialized

Q-Table		Actions			
		South (0)	North (1)	East (2)	West (3)
States	0	0	0	0	0
	⋮	⋮	⋮	⋮	⋮
	327	0	0	0	0
	⋮	⋮	⋮	⋮	⋮
	499	0	0	0	0

Training

Q-Table		Actions			
		South (0)	North (1)	East (2)	West (3)
States	0	0	0	0	0
	⋮	⋮	⋮	⋮	⋮
	328	-2.30108105	-1.97092096	-2.30357004	-2.20591839
	⋮	⋮	⋮	⋮	⋮
	499	9.96984239	4.02706992	12.96022777	29

Updating the Q-table via exploration

- To update the Q-table, we will let our agent play a few rounds of the game and “explore” the maze.
- At the end of each round, we obtain a finite **history of actions, states and rewards**, i.e. a sequence of values for run k

$$h_k = \{s_1, a_1, R_1, s_2, a_2, R_2, \dots\}$$

Updating the Q-table via exploration

- This sequence can then be used, to update the Q table, according to

$$Q^{new}(s_t, a_t) \leftarrow \underbrace{Q(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \overbrace{\bigg(\underbrace{\underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1}, a)}_{\text{estimate of optimal future value}}}_{\text{new value (temporal difference target)}} - \underbrace{Q(s_t, a_t)}_{\text{old value}} \bigg)}^{\text{temporal difference}}$$

- In a sense, this formula is the “**equivalent**” of our **gradient descent update**, but in the case of reinforcement learning, and is commonly referred to as **Q-learning**.

Updating the Q table via exploration

- A quick note on its parameters…

$$Q^{new}(s_t, a_t) \leftarrow \underbrace{Q(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \overbrace{\bigg(\underbrace{\underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1}, a)}_{\text{estimate of optimal future value}}}_{\text{new value (temporal difference target)}} - \underbrace{Q(s_t, a_t)}_{\text{old value}} \bigg)}^{\text{temporal difference}}$$

- The learning rate α determines to what extent newly acquired information overrides old information in the Q-table.
- A value 0 makes the agent learn nothing (exclusively exploiting prior knowledge), while a factor of 1 makes the agent consider only the most recent information (ignoring prior knowledge to explore possibilities).

Restricted

Updating the Q table via exploration

- A quick note on its parameters...

$$Q^{new}(s_t, a_t) \leftarrow \underbrace{Q(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \overbrace{\Bigg(\underbrace{\underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1}, a)}_{\text{estimate of optimal future value}}}_{\text{new value (temporal difference target)}} - \underbrace{Q(s_t, a_t)}_{\text{old value}} \Bigg)}^{\text{temporal difference}}$$

- The discount factor γ determines the importance of future rewards.
- A value 0 will make the agent "myopic" (or short-sighted) by only considering current rewards, i.e. R_t (in the update rule above).
- A value approaching 1 will make it strive for a long-term high reward.

Updating the Q table via exploration

- A quick note on its parameters…

$$Q^{new}(s_t, a_t) \leftarrow \underbrace{Q(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \overbrace{\Bigg(\underbrace{\underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1}, a)}_{\text{estimate of optimal future value}}}_{\text{new value (temporal difference target)}} - \underbrace{Q(s_t, a_t)}_{\text{old value}} \Bigg)}^{\text{temporal difference}}$$

- Q-learning implicitly assumes some initial values in the Q-table before the first update occurs.
- High initial values, also known as "optimistic initial conditions“ tend to encourage exploration.

Training an agent

- Training an agent then requires to define a policy π, which will smoothly transition from exploration into exploitation, as in the random candy machines before.
- Upon seeing "convergence" on the values of the Q-table, we can then claim that the agent has been "properly" trained.
- From there, exploiting should then give the best strategy.

Initialized

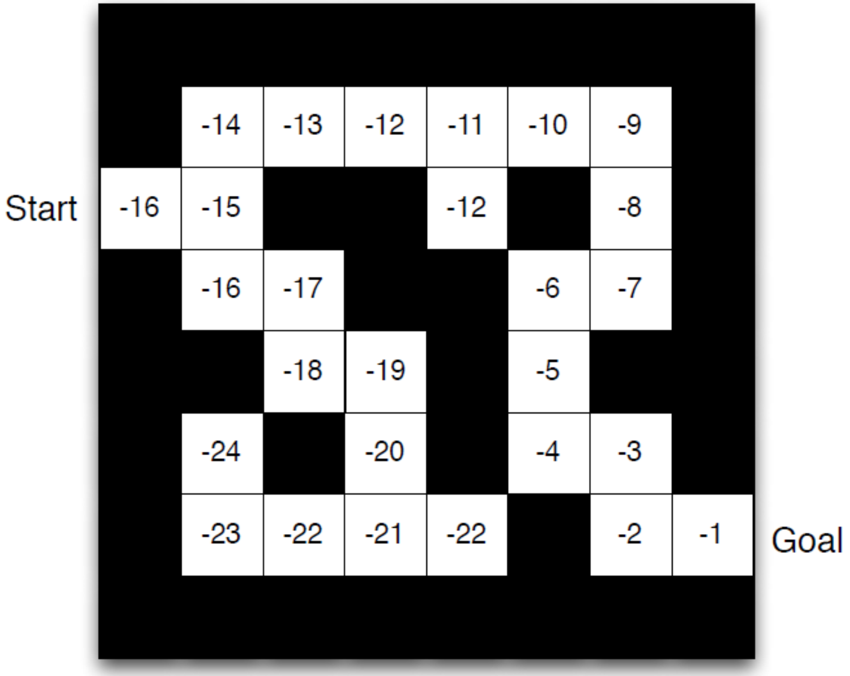
Q-Table		Actions			
		South (0)	North (1)	East (2)	West (3)
States	0	0	0	0	0
	⋮	⋮	⋮	⋮	⋮
	327	0	0	0	0
	⋮	⋮	⋮	⋮	⋮
	499	0	0	0	0

Training

Q-Table		Actions			
		South (0)	North (1)	East (2)	West (3)
States	0	0	0	0	0
	⋮	⋮	⋮	⋮	⋮
	328	-2.30108105	-1.97092096	-2.30357004	-2.20591839
	⋮	⋮	⋮	⋮	⋮
	499	9.96984239	4.02706992	12.96022777	29

Training an agent

- By successfully updating V and Q at the end of each round, we will eventually obtain a good estimate of the "true" V (or alternatively Q) values for our problem.
- That matches the minimal number of steps which should be taken from any given square of the maze to reach the end.



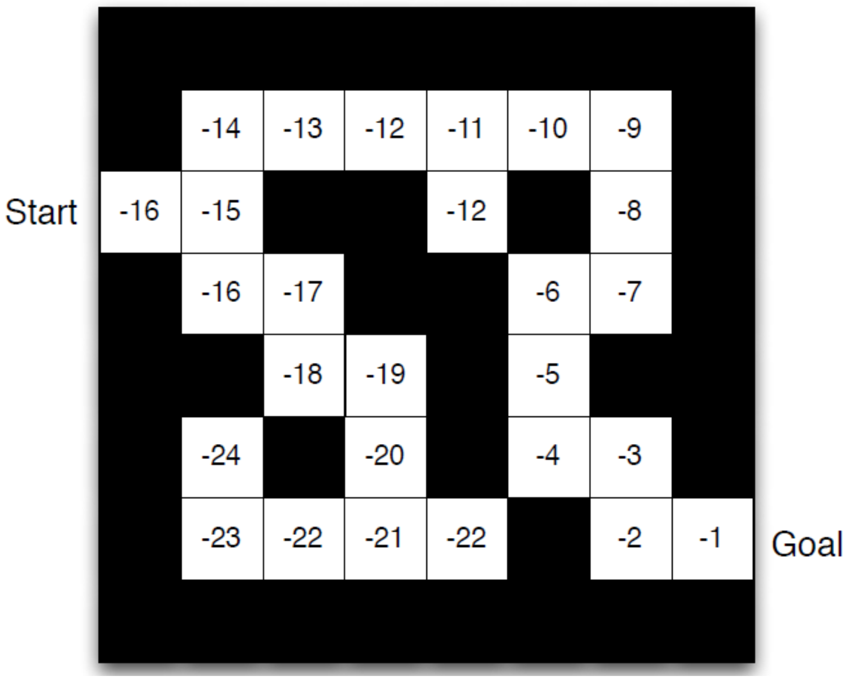
Note: the values above are the V ones, literally telling us the number of steps needed to reach goal from any position.

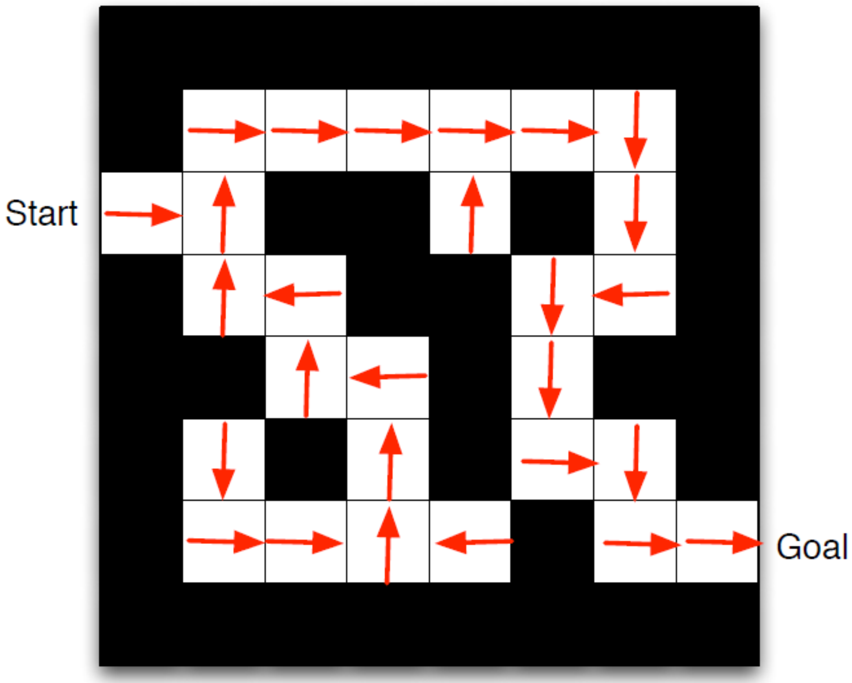
Training an agent

- Reusing the V (or Q) value allows to define the optimal policy π to use in any given state.
- That is, the optimal direction in which we should go for any given square.

$$a_t = \pi_t(s_t) = \arg\max_{a \in A} Q_t^{\pi}(a, s_t)$$

$$Q_t^{\pi}(a_t, s_t) = R_t(a, s_t) + V_{t+1}^{\pi}(s_{t+1})$$





Restricted

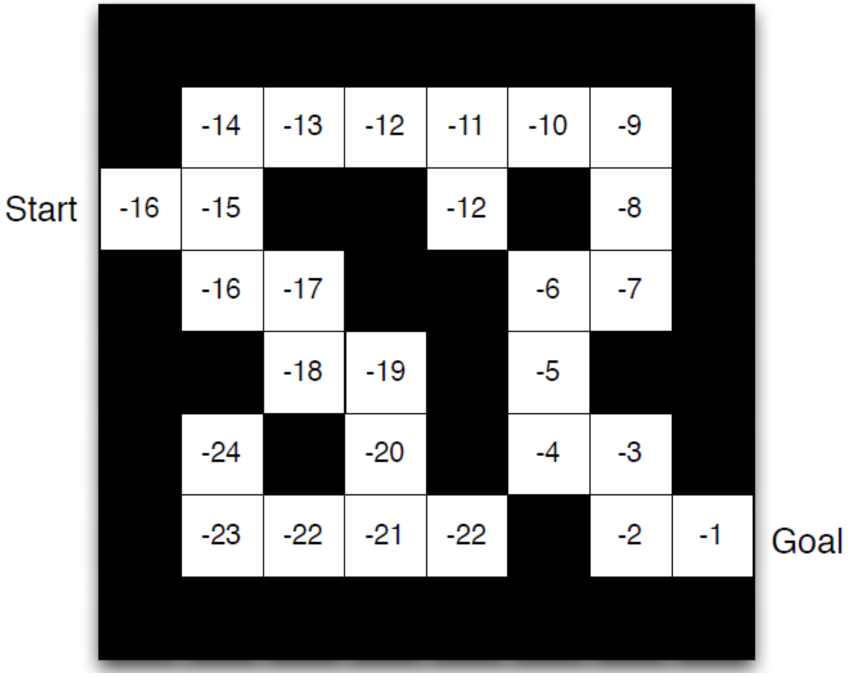
Training an agent

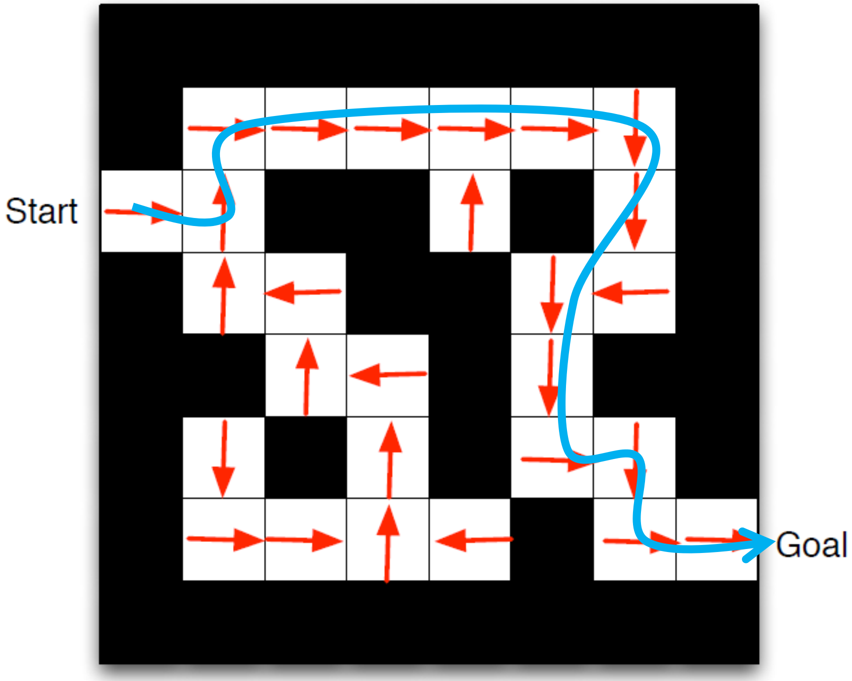
Later on, we can then replay the game,

- in full exploitation mode, i.e. by playing the best action according to our policy π every time,
- without any exploration moves.

This gives us the shortest path.

Our agent has then learnt to recognize the maze and figured the shortest path through trial and error!





Time for something a bit more advanced!

Definitely out-of-scope, but interesting nonetheless!

A representation problem

Problem: in many RL problems, the states and/or actions sets are not necessarily finite.

- In that case, it is impossible to represent the Q and V functions as tables.
- **And, even worse, for these problems, coming up with a closed form expression of the V and Q functions might prove challenging.**

Initialized

Q-Table		Actions			
		South (0)	North (1)	East (2)	West (3)
States	0	0	0	0	0
	⋮	⋮	⋮	⋮	⋮
	327	0	0	0	0
	⋮	⋮	⋮	⋮	⋮
	499	0	0	0	0

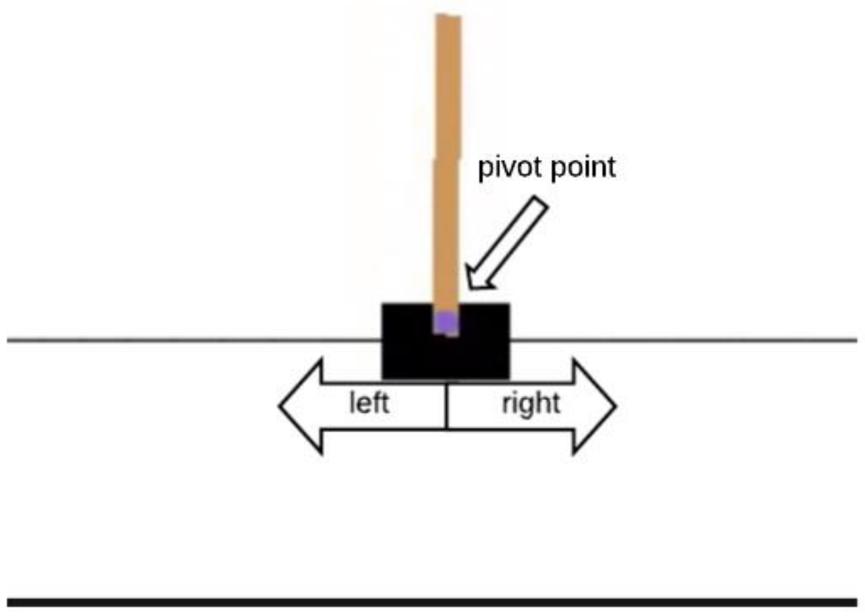
Training

Q-Table		Actions			
		South (0)	North (1)	East (2)	West (3)
States	0	0	0	0	0
	⋮	⋮	⋮	⋮	⋮
	328	-2.30108105	-1.97092096	-2.30357004	-2.20591839
	⋮	⋮	⋮	⋮	⋮
	499	9.96984239	4.02706992	12.96022777	29

Toy example #3 (shown, but implementation out of the scope of this class)

Example: the cart-pole problem.

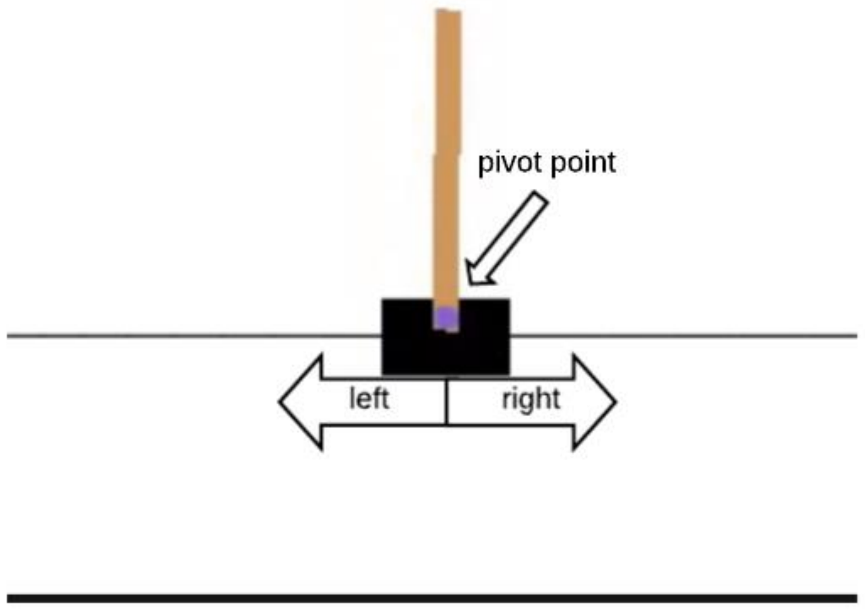
- **State:** our current visualization of the cart (i.e. an image).
- **Actions:** 2 of them, go left or go right at a fixed speed.
- **Reward:** +1 for each unit of time where the cart does not leave the screen and the pole does not fall below a certain angle.
- **Next state generation:** cart and pole both follow simple programmed rules of physics.



Toy example #3 (shown, but implementation out of the scope of this class)

Problem: in many RL problems, the states and/or actions sets are not necessarily finite.

- In that case, it is impossible to represent the Q and V functions as tables.



- **How do we address this issue?**
- **Give it to an AI! (as usual)**

Toy example #3 (shown, but implementation out of the scope of this class)

Solution: replace the Q-table with a Deep Neural Network, whose job is to estimate the value of each action (left/right) in the current state.

- The objective is then to train, just like before with our Q-table.
- **However, we are no longer changing the table values but the Neural Net parameters!**

```python
class DQN(nn.Module):

    def __init__(self, h, w, outputs):
        super(DQN, self).__init__()
        self.conv1 = nn.Conv2d(3, 16, kernel_size=5, stride=2)
        self.bn1 = nn.BatchNorm2d(16)
        self.conv2 = nn.Conv2d(16, 32, kernel_size=5, stride=2)
        self.bn2 = nn.BatchNorm2d(32)
        self.conv3 = nn.Conv2d(32, 32, kernel_size=5, stride=2)
        self.bn3 = nn.BatchNorm2d(32)

        # Number of Linear input connections depends on output of conv2d layers
        # and therefore the input image size, so compute it.
        def conv2d_size_out(size, kernel_size = 5, stride = 2):
            return (size - (kernel_size - 1) - 1) // stride  + 1
        convw = conv2d_size_out(conv2d_size_out(conv2d_size_out(w)))
        convh = conv2d_size_out(conv2d_size_out(conv2d_size_out(h)))
        linear_input_size = convw * convh * 32
        self.head = nn.Linear(linear_input_size, outputs)

    # Called with either one element to determine next action, or a batch
    # during optimization. Returns tensor([[left0exp,right0exp]...]).
    def forward(self, x):
        x = F.relu(self.bn1(self.conv1(x)))
        x = F.relu(self.bn2(self.conv2(x)))
        x = F.relu(self.bn3(self.conv3(x)))
        return self.head(x.view(x.size(0), -1))
```

Toy example #3 (shown, but implementation out of the scope of this class)

On each round of the game, use the Q network to compute the Q-value of both actions (left/right) in the current state.

- Use the one with the maximal value (**exploitation**) or a randomly chosen action (**exploration**).
- Use ***ϵ*-greedy policy** to decide how to explore/exploit.

```python
def select_action(state):
    global steps_done
    sample = random.random()
    eps_threshold = EPS_END + (EPS_START - EPS_END) * \
        math.exp(-1. * steps_done / EPS_DECAY)
    steps_done += 1
    if sample > eps_threshold:
        with torch.no_grad():
            # Here, t.max(1) will return largest column value of each row.
            # Second column on max result is index of where max element was
            # found, so we pick action with the larger expected reward.
            return policy_net(state).max(1)[1].view(1, 1)
    else:
        return torch.tensor([[random.randrange(n_actions)]], device=device, dtype=torch.long)
```

Toy example #3

To train this DNN, we need a dataset of some sort.

- Do so by playing the game multiple times and keeping a history of the (state, action, rewards, next_state, done) tuples.
- Here, done indicates that the game has ended (out of screen or low angle on pole).
- Structure is roughly similar to our dataloaders?

```python
# Define namedtuples for transitions and history
Transition = namedtuple('Transition', ('state', 'action', 'next_state', 'reward'))
```

```python
class ReplayMemory(object):

    def __init__(self, capacity):
        self.capacity = capacity
        self.memory = []
        self.position = 0

    def push(self, *args):
        """
        Saves a transition to memory.
        """
        if len(self.memory) < self.capacity:
            self.memory.append(None)
        self.memory[self.position] = Transition(*args)
        self.position = (self.position + 1) % self.capacity

    def sample(self, batch_size):
        """
        Get sample from history.
        """
        return random.sample(self.memory, batch_size)

    def __len__(self):
        """
        Get length of history (number of samples).
        """
        return len(self.memory)
```

Toy example #3 (shown, but implementation out of the scope of this class)

Core idea for memory replay: we are trying to approximate a complex, nonlinear function Q, with a Neural Network.

- To do this, we must calculate targets using the **Bellman equation** and then consider that we have a **supervised learning** problem at hand.
- **Important:** However, one of the fundamental requirements for SGD optimization is that the training data is independent and identically distributed and when the Agent interacts with the game, the sequence of experience tuples can be highly correlated.
- The naive Q-learning algorithm that learns from each of these experiences tuples in sequential order runs the risk of getting swayed by the effects of this correlation.

Toy example #3 (shown, but implementation out of the scope of this class)

Definition (experience buffer in RL):

- We can prevent action values from oscillating or diverging catastrophically by using a large buffer of our past experience and sample training data from it, instead of using our latest experience.
- This is called an **experience buffer**.
- The experience buffer contains a collection of experience tuples (state, action, rewards, next_state).
- The tuples are gradually added to the buffer as the agents keep on interacting with the game.

Toy example #3 (shown, but implementation out of the scope of this class)

Definition (experience replay):

- The simplest implementation is a buffer of fixed size, with new data added to the end of the experience buffer, so that it pushes the oldest experience out of it.
- The act of sampling a small batch of tuples from the experience buffer in order to learn is known as **experience replay**.
- In addition to breaking harmful correlations, experience replay allows us to learn more from individual tuples multiple times, recall rare occurrences, and in general make better use of our experience.

Toy example #3 (shown, but implementation out of the scope of this class)

To train this DNN, we need a dataset of some sort.

- Do so by playing the game multiple times and keeping a history of the (state, action, rewards, next_state, done) tuples.
- Here, done indicates that the game has ended (out of screen or low angle on pole).
- Structure is roughly similar to our dataloaders?

```python
# Define namedtuples for transitions and history
Transition = namedtuple('Transition', ('state', 'action', 'next_state', 'reward'))
```

```python
class ReplayMemory(object):

    def __init__(self, capacity):
        self.capacity = capacity
        self.memory = []
        self.position = 0

    def push(self, *args):
        """
        Saves a transition to memory.
        """
        if len(self.memory) < self.capacity:
            self.memory.append(None)
        self.memory[self.position] = Transition(*args)
        self.position = (self.position + 1) % self.capacity

    def sample(self, batch_size):
        """
        Get sample from history.
        """
        return random.sample(self.memory, batch_size)

    def __len__(self):
        """
        Get length of history (number of samples).
        """
        return len(self.memory)
```

Toy example #3 (shown, but implementation out of the scope of this class)

To train this DNN, we need a loss function and weight update procedure of some sort, as well.

Our previous Q-learning was using this iterative update formula.

$$Q^{new}(s_t, a_t) \leftarrow \underbrace{Q(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \overbrace{\Bigg(\underbrace{\underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1}, a)}_{\text{estimate of optimal future value}}}_{\text{new value (temporal difference target)}} - \underbrace{Q(s_t, a_t)}_{\text{old value}} \Bigg)}^{\text{temporal difference}}$$

Toy example #3 (shown, but implementation out of the scope of this class)

To train this DNN, we need a loss function and weight update procedure of some sort, as well.

Our previous Q-learning was using this iterative update formula.

$$Q^{new}(s_t, a_t) \leftarrow \underbrace{Q(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \overbrace{\Bigg(\underbrace{\underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1}, a)}_{\text{estimate of optimal future value}}}_{\text{new value (temporal difference target)}} - \underbrace{Q(s_t, a_t)}_{\text{old value}} \Bigg)}^{\text{temporal difference}}$$

- **Problem:** update $Q(s_t, a)$ via $Q(s_{t+1}, a)$. However, both states have only one step between them. This makes them very similar, and it is very hard for a Neural Network to distinguish between them.

Toy example #3 (shown, but implementation out of the scope of this class)

Solution: use **two Neural Networks**, one for **training** $Q(s_t, a)$ and one for producing **targets** $Q(s_{t+1}, a)$.

- That is, the predicted Q values of this second Q-network, called the target network, are used to backpropagate and train the main Q-network.
- **Note:** the target network's parameters are not trained, but they are periodically synchronized with the parameters of the main Q-network, every K iterations (hyperparameter, value of K to be freely chosen).
- The idea is that using the target network's Q values to train the main Q-network will improve the stability of the training.

```python
def optimize_model():
    if len(memory) < BATCH_SIZE:
        return
    transitions = memory.sample(BATCH_SIZE)
    # Transpose the batch (see https://stackoverflow.com/a/19343/3343043 for
    # detailed explanation). This converts batch-array of Transitions
    # to Transition of batch-arrays.
    batch = Transition(*zip(*transitions))

    # Compute a mask of non-final states and concatenate the batch elements
    # (a final state would've been the one after which simulation ended)
    non_final_mask = torch.tensor(tuple(map(lambda s: s is not None,
                                          batch.next_state)), device=device, dtype=torch.bool)
    non_final_next_states = torch.cat([s for s in batch.next_state
                                                if s is not None])
    state_batch = torch.cat(batch.state)
    action_batch = torch.cat(batch.action)
    reward_batch = torch.cat(batch.reward)

    # Compute Q(s_t, a) - the model computes Q(s_t), then we select the
    # columns of actions taken. These are the actions which would've been taken
    # for each batch state according to policy_net
    state_action_values = policy_net(state_batch).gather(1, action_batch)

    # Compute V(s_{t+1}) for all next states.
    # Expected values of actions for non_final_next_states are computed based
    # on the "older" target_net; selecting their best reward with max(1)[0].
    # This is merged based on the mask, such that we'll have either the expected
    # state value or 0 in case the state was final.
    next_state_values = torch.zeros(BATCH_SIZE, device=device)
    next_state_values[non_final_mask] = target_net(non_final_next_states).max(1)[0].detach()
    # Compute the expected Q values
    expected_state_action_values = (next_state_values * GAMMA) + reward_batch

    # Compute Huber loss
    loss = F.smooth_l1_loss(state_action_values, expected_state_action_values.unsqueeze(1))

    # Optimize the model
    optimizer.zero_grad()
    loss.backward()
    for param in policy_net.parameters():
        param.grad.data.clamp_(-1, 1)
    optimizer.step()
```

Toy example #3 (shown, but implementation out of the scope of this class)

To train this DNN, we need a loss function and weight update procedure of some sort.

To create a loss function, let us first recall that

$$Q_t^{\pi}(s_t, a_t) = R_t(s_t, a_t) + \gamma Q_{t+1}^{\pi}(s_{t+1}, \pi(s_{t+1}))$$

Let us denote the error δ as

$$\delta = Q_t^{\pi}(s_t, a_t) - \left(R_t(s_t, a_t) + \gamma \max_a Q_{t+1}^{\pi}(s_{t+1}, a)\right)$$

Toy example #3 (shown, but implementation out of the scope of this class)

To train this DNN, we need a loss function and weight update procedure of some sort.

To train our DNN, we want to minimize this error δ.

We will use the L1 norm on delta to do so.

$$L(\delta) = |\delta|$$

Toy example #3 (shown, but implementation out of the scope of this class)

To train this DNN, we need a loss function and weight update procedure of some sort.

To train our DNN, we want to minimize this error δ.

We will use the L1 norm on delta to do so.

Note: we can also use a slightly different loss function known as the **Huber loss**, which is slightly more robust to outliers.

$$L_d(\delta) = \begin{cases} \frac{1}{2}\delta^2 & if\ |\delta| \leq d \\ d\left(|\delta| - \frac{1}{2}d\right) & else \end{cases}$$

```python
def optimize_model():
    if len(memory) < BATCH_SIZE:
        return
    transitions = memory.sample(BATCH_SIZE)
    # Transpose the batch (see https://stackoverflow.com/a/19343/3343043 for
    # detailed explanation). This converts batch-array of Transitions
    # to Transition of batch-arrays.
    batch = Transition(*zip(*transitions))

    # Compute a mask of non-final states and concatenate the batch elements
    # (a final state would've been the one after which simulation ended)
    non_final_mask = torch.tensor(tuple(map(lambda s: s is not None,
                                          batch.next_state)), device=device, dtype=torch.bool)
    non_final_next_states = torch.cat([s for s in batch.next_state
                                                if s is not None])
    state_batch = torch.cat(batch.state)
    action_batch = torch.cat(batch.action)
    reward_batch = torch.cat(batch.reward)

    # Compute Q(s_t, a) - the model computes Q(s_t), then we select the
    # columns of actions taken. These are the actions which would've been taken
    # for each batch state according to policy_net
    state_action_values = policy_net(state_batch).gather(1, action_batch)

    # Compute V(s_{t+1}) for all next states.
    # Expected values of actions for non_final_next_states are computed based
    # on the "older" target_net; selecting their best reward with max(1)[0].
    # This is merged based on the mask, such that we'll have either the expected
    # state value or 0 in case the state was final.
    next_state_values = torch.zeros(BATCH_SIZE, device=device)
    next_state_values[non_final_mask] = target_net(non_final_next_states).max(1)[0].detach()
    # Compute the expected Q values
    expected_state_action_values = (next_state_values * GAMMA) + reward_batch

    # Compute Huber loss
    loss = F.smooth_l1_loss(state_action_values, expected_state_action_values.unsqueeze(1))

    # Optimize the model
    optimizer.zero_grad()
    loss.backward()
    for param in policy_net.parameters():
        param.grad.data.clamp_(-1, 1)
    optimizer.step()
```

Trainer function

- Our trainer function will play the game 500 times.
- Keep track of different histories over the 500 games.
- Sample from history to train our main Q-Network.
- Backpropagate with mixed main and target Q-networks values.
- Occasionally update the target network.

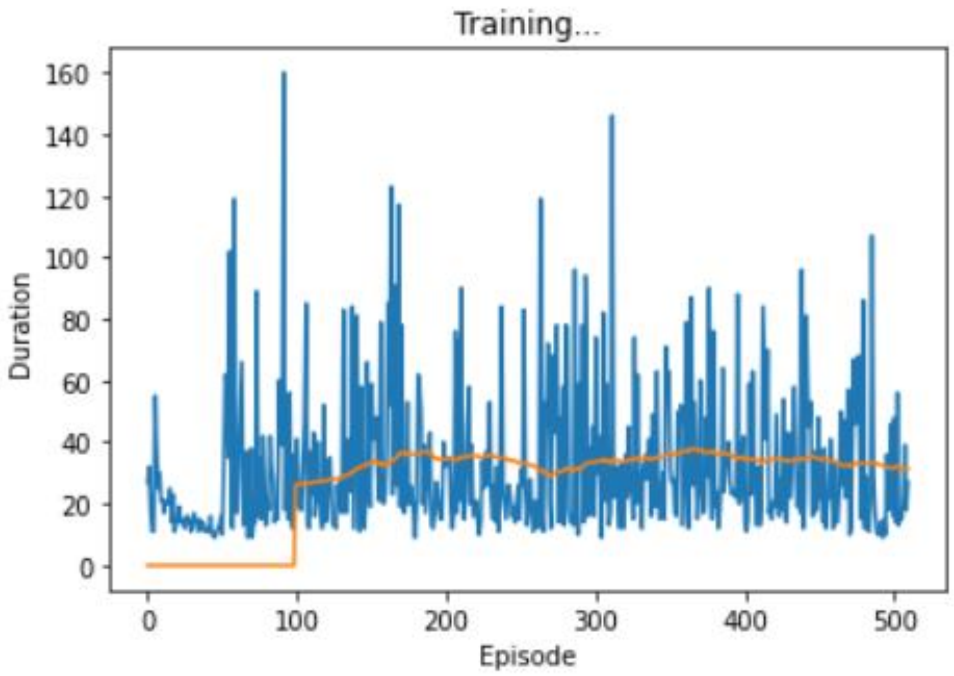
```python
"""
Full trainer on 500 iteration (for meaningful improvements)
"""
num_episodes = 500
for i_episode in range(num_episodes):
    print("Episode:", i_episode)
    # Initialize the environment and state
    env.reset()
    last_screen = get_screen()
    current_screen = get_screen()
    state = current_screen - last_screen
    for t in count():
        # Select and perform an action
        action = select_action(state)
        _, reward, done, _ = env.step(action.item())
        reward = torch.tensor([reward], device=device)

        # Observe new state
        last_screen = current_screen
        current_screen = get_screen()
        if not done:
            next_state = current_screen - last_screen
        else:
            next_state = None

        # Store the transition in memory
        memory.push(state, action, next_state, reward)

        # Move to the next state
        state = next_state

        # Perform one step of the optimization (on the policy network)
        optimize_model()
        if done:
            episode_durations.append(t + 1)
            plot_durations()
            break
    # Update the target network, copying all weights and biases in DQN
    if i_episode % TARGET_UPDATE == 0:
        target_net.load_state_dict(policy_net.state_dict())
```

Restricted

Training results

- Our RL agent will learn to balance the pole on the cart, by playing the game.
- Can display the length of each game/episode to see the progression!
- This RL approach of training some DNNs to replace the Q functions is commonly referred to as **Deep Q-learning**.

```python
def plot_durations():
    """
    Show episode durations for each episode.
    """
    plt.figure(2)
    plt.clf()
    durations_t = torch.tensor(episode_durations, dtype=torch.float)
    plt.title('Training...')
    plt.xlabel('Episode')
    plt.ylabel('Duration')
    plt.plot(durations_t.numpy())
    # Take 100 episode averages and plot them too
    if len(durations_t) >= 100:
        means = durations_t.unfold(0, 100, 1).mean(1).view(-1)
        means = torch.cat((torch.zeros(99), means))
        plt.plot(means.numpy())
```

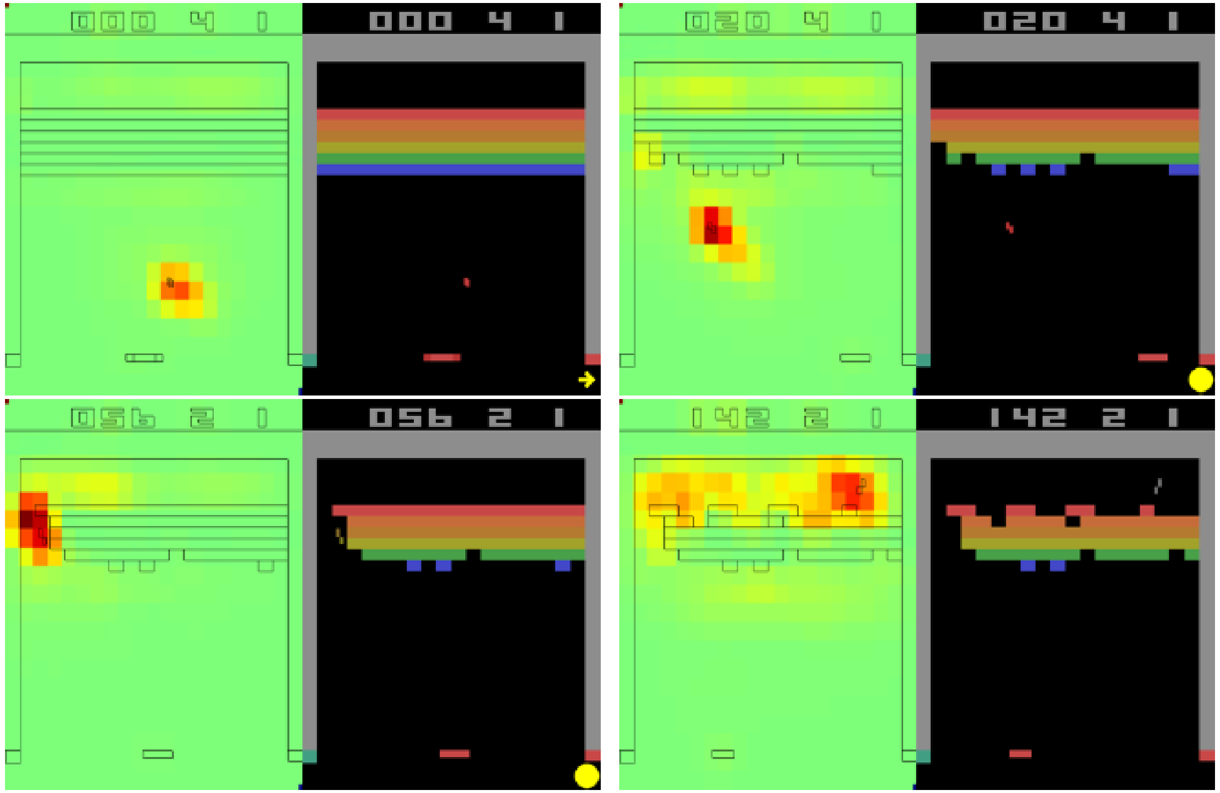


Following this cart-pole balance idea…

- Train an AI to keep a robot on its feet, despite some *"minor environment perturbations"* *(a polite way of saying you kick the hell out of the robot for fun).*
- Video: https://www.youtube.com/watch?v=NR32ULxbjYc
- **BostonDynamics** blog: https://blog.bostondynamics.com/

Following this idea of using computer vision to identify state and act...

- Train an AI to play video games with Deep Reinforcement Learning (Mnih, 2013)!
- Paper: https://www.cs.toronto.edu/~vmnih/docs/dqn.pdf
- Video: https://www.youtube.com/watch?v=TmPfTpjtdgg



Learn more about these topics

Out of class, for those of you who are curious

- [TheBibleOfRL] R. **Sutton** et al., “Reinforcement learning: An Introduction, 2nd edition”, 2018. http://www.incompleteideas.net/book/RLbook2020.pdf
- [Mnih2018] **Mnih** et al., “Playing Atari with Deep Reinforcement Learning”, 2018. https://arxiv.org/abs/1312.5602
- The **BostonDynamics** blog (some more robots updates!) https://blog.bostondynamics.com/

Learn more about these topics

Tracking important names (Track their works and follow them on Scholar, Twitter, or whatever works for you!)

- **Richard Sutton: Professor** at **University of Alberta,** also **DeepMind**. Co-author of the Bible of RL (possibly most influential professor in the field of RL).
  https://scholar.google.ca/citations?user=6m4wv6gAAAAJ&hl=en
  https://www.ualberta.ca/admissions-programs/online-courses/reinforcement-learning/index.html
  http://www.incompleteideas.net/book/RLbook2020.pdf
- **Andrew Barto: Professor** at **University of Massachussets**. Co-author of the Bible of RL.
  https://people.cs.umass.edu/~barto/
  https://scholar.google.com/citations?user=CMIgrCgAAAAJ&hl=en

Learn more about these topics

Tracking important names (Track their works and follow them on Scholar, Twitter, or whatever works for you!)

- **David Sliver: Reasearcher** at **DeepMind, Adjunct Professor** at **University College London** (?). Inventor of Alpha GO AIs, has a fantastic course on RL as well.
  https://www.davidsilver.uk/
  https://scholar.google.com/citations?user=-8DNE4UAAAAJ&hl=en
  https://www.davidsilver.uk/teaching/
- **Volodymir Mnih: Reasearcher** at **DeepMind.**
  http://www.cs.toronto.edu/~vmnih
  https://scholar.google.com/citations?user=rLdfJ1gAAAAJ&hl=en

Learn more about these topics

Out of class, for those of you who are curious

- [AlphaGo] How AlphaGo was created using DL and RL. https://jonathan-hui.medium.com/alphago-zero-a-game-changer-14ef6e45eba5
- Full Alpha Go movie on DeepMind's Youtube channel. Award-winning movie, pretty cool. https://www.youtube.com/watch?v=WXuK6gekU1Y