Machine Learning and
Computer Vision Group

IST AUSTRIA
Institute of Science and Technology

Deep Learning with TensorFlow

http://cvml.ist.ac.at/courses/DLWT_W18

Lecture 10:
Deep Q-Learning

Q-Learning - Deep Learning with TensorFlow (DLWT) '18

Mathias Lechner

IST Austria

mathias.lechner@ist.ac.at

January 20, 2019

Overview

1. Reinforcement Learning
   - Definitions
   - Different approaches
2. Q-Learning
   - With tables
   - Deep-Q-Networks (DQN)
3. Advanced methods

Types of Machine Learning

Supervised Learning:

Given: Labeled samples $(x_1, y_1), (x_2, y_2), \dots (x_n, y_n)$

Task: Find $f : x \mapsto \hat{y}$, that has minimal loss $L(y, \hat{y})$

Types of Machine Learning

Supervised Learning:

Given: Labeled samples $(x_1, y_1), (x_2, y_2), \dots (x_n, y_n)$

Task: Find $f : x \mapsto \hat{y}$, that has minimal loss $L(y, \hat{y})$

Reinforcement Learning:

Given: Interactive environment

Task: Find interacting policy, that maximizes reward

Types of Machine Learning

Supervised Learning:

Given: Labeled samples $(x_1, y_1), (x_2, y_2), \ldots (x_n, y_n)$

Task: Find $f : x \mapsto \hat{y}$, that has minimal loss $L(y, \hat{y})$

Reinforcement Learning:

Given: Interactive environment

Task: Find interacting policy, that maximizes reward

What's an *"Interactive environment"*?

Markov-Decision-Process (MDP)

- $MDP = (S, A, P, R)$
  - Set of states S
  - Set of actions A
  - Initial state distribution $P_0 = \mathbb{P}[s_0]$
  - Transition probability $P(s, a, s') = \mathbb{P}[s'|s, a]$
  - Reward function $R : S \rightarrow \mathbb{R}$

Markov-Decision-Process (MDP)

- $MDP = (S, A, P, R)$
  - Set of states S
  - Set of actions A
  - Initial state distribution $P_0 = \mathbb{P}[s_0]$
  - Transition probability $P(s, a, s') = \mathbb{P}[s'|s, a]$
  - Reward function $R : S \to \mathbb{R}$
- Policy $\pi : S \to A$

Markov-Decision-Process (MDP)

- $MDP = (S, A, P, R)$
  - Set of states S
  - Set of actions A
  - Initial state distribution $P_0 = \mathbb{P}[s_0]$
  - Transition probability $P(s, a, s') = \mathbb{P}[s'|s, a]$
  - Reward function $R : S \to \mathbb{R}$
- Policy $\pi : S \to A$

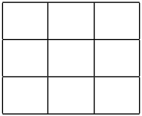
```python
state = env.reset()
for _ in range(1000):
  action = policy(state)
  state, reward, done, info \
      = env.step(action)
```

Markov-Decision-Process (MDP)

- $MDP = (S, A, P, R)$
  - Set of states S
  - Set of actions A
  - Initial state distribution $P_0 = \mathbb{P}[s_0]$
  - Transition probability $P(s, a, s') = \mathbb{P}[s'|s, a]$
  - Reward function $R : S \to \mathbb{R}$
- Policy $\pi : S \to A$

```
state = env.reset()
for _ in range(1000):
    action = policy(state)
    state, reward, done, info \
        = env.step(action)
```

s_0



Markov-Decision-Process (MDP)

- $MDP = (S, A, P, R)$
  - Set of states S
  - Set of actions A
  - Initial state distribution $P_0 = \mathbb{P}[s_0]$
  - Transition probability $P(s, a, s') = \mathbb{P}[s'|s, a]$
  - Reward function $R : S \to \mathbb{R}$
- Policy $\pi : S \to A$

```python
state = env.reset()
for _ in range(1000):
    action = policy(state)
    state, reward, done, info \
        = env.step(action)
```

$s_0 \xrightarrow{a_0}$

X		

Markov-Decision-Process (MDP)

- $MDP = (S, A, P, R)$
  - Set of states S
  - Set of actions A
  - Initial state distribution $P_0 = \mathbb{P}[s_0]$
  - Transition probability $P(s, a, s') = \mathbb{P}[s'|s, a]$
  - Reward function $R : S \to \mathbb{R}$
- Policy $\pi : S \to A$

```
state = env.reset()
for _ in range(1000):
    action = policy(state)
    state, reward, done, info \
        = env.step(action)
```

$s_0 \xrightarrow{a_0} r_0, s_1$

$$r_0 = 0$$

X		
	O	

Markov-Decision-Process (MDP)

- $MDP = (S, A, P, R)$
  - Set of states S
  - Set of actions A
  - Initial state distribution $P_0 = \mathbb{P}[s_0]$
  - Transition probability
    $P(s, a, s') = \mathbb{P}[s' | s, a]$
  - Reward function $R : S \to \mathbb{R}$
- Policy $\pi : S \to A$

```
state = env.reset()
for _ in range(1000):
    action = policy(state)
    state, reward, done, info \
        = env.step(action)
```

$s_0 \xrightarrow{a_0} r_0, s_1 \xrightarrow{a_1}$

$r_0 = 0$

X		X
	O	

Markov-Decision-Process (MDP)

- $MDP = (S, A, P, R)$
  - Set of states S
  - Set of actions A
  - Initial state distribution $P_0 = \mathbb{P}[s_0]$
  - Transition probability $P(s, a, s') = \mathbb{P}[s' | s, a]$
  - Reward function $R : S \to \mathbb{R}$
- Policy $\pi : S \to A$

```
state = env.reset()
for _ in range(1000):
    action = policy(state)
    state, reward, done, info \
        = env.step(action)
```

$$s_0 \xrightarrow{a_0} r_0, s_1 \xrightarrow{a_1} r_1, s_2$$

$$r_0 = 0$$

$$r_1 = 0$$

X	O	X
	O	

Markov-Decision-Process (MDP)

- $MDP = (S, A, P, R)$
  - Set of states S
  - Set of actions A
  - Initial state distribution $P_0 = \mathbb{P}[s_0]$
  - Transition probability $P(s, a, s') = \mathbb{P}[s'|s, a]$
  - Reward function $R : S \to \mathbb{R}$
- Policy $\pi : S \to A$

```
state = env.reset()
for _ in range(1000):
    action = policy(state)
    state, reward, done, info \
            = env.step(action)
```

$$s_0 \xrightarrow{a_0} r_0, s_1 \xrightarrow{a_1} r_1, s_2 \xrightarrow{a_2}$$

$$r_0 = 0$$

$$r_1 = 0$$

		X
X	O	X
	O	

Markov-Decision-Process (MDP)

- $MDP = (S, A, P, R)$
  - Set of states S
  - Set of actions A
  - Initial state distribution $P_0 = \mathbb{P}[s_0]$
  - Transition probability $P(s, a, s') = \mathbb{P}[s'|s, a]$
  - Reward function $R : S \to \mathbb{R}$
- Policy $\pi : S \to A$

```python
state = env.reset()
for _ in range(1000):
    action = policy(state)
    state, reward, done, info \
        = env.step(action)
```

$$s_0 \xrightarrow{a_0} r_0, s_1 \xrightarrow{a_1} r_1, s_2 \xrightarrow{a_2} r_2, s_3$$

$$r_0 = 0$$
$$r_1 = 0$$
$$r_2 = -1$$

	O	X
X	O	X
	O	

You lost!

Objective

- Let's say we are in an arbitrary state s_t
- The optimal action would maximize sum of future rewards $r_t, r_{t+1}, \dots r_{t+n}$

Objective

- Let's say we are in an arbitrary state s_t
- The optimal action would maximize sum of future rewards $r_t, r_{t+1}, \ldots r_{t+n}$
  - But $n \to \infty$
  - and r_i are random variables
  - that depend on future actions

Objective

- Let's say we are in an arbitrary state s_t
- The optimal action would maximize sum of future rewards $r_t, r_{t+1}, \ldots r_{t+n}$
  - But $n \to \infty$
  - and r_i are random variables
  - that depend on future actions

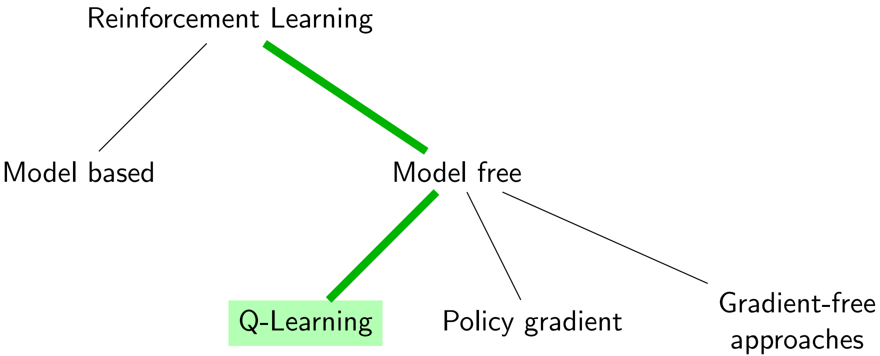
Optimal policy:

$$\underset{\pi}{\text{maximize}}\ \mathbb{E}\Big[\underbrace{\sum_{i=t}^{t+n} r_i\ \gamma^{(i-t)}}_{R_t}\ \Big|\ \pi\Big]$$

Different approaches to RL

- Reinforcement Learning
  - Model based
  - Model free
    - Q-Learning
    - Policy gradient
    - Gradient-free approaches

Different approaches to RL



State-Action function

We define

$$Q^*(s, a) = \max_{\pi} \mathbb{E}\Big[R_t \;\Big|\; s_t = s, a_t = a, \pi\Big]$$

State-Action function

We define

$$Q^*(s,a) = \max_{\pi} \mathbb{E}\Big[R_t \;\Big|\; s_t = s, a_t = a, \pi\Big]$$

$Q^*(s,a) =$ *What's the expected discounted return if we execute action a in state s and then follow the optimal policy*

State-Action function

We define

$$Q^*(s,a) = \max_\pi \mathbb{E}\Big[R_t \;\Big|\; s_t = s, a_t = a, \pi\Big]$$

$Q^*(s,a) =$ *What's the expected discounted return if we execute action a in state s and then follow the optimal policy*

By this definition $\max_a Q^*(s,a)$ is the optimal policy

State-Action function

We define

$$Q^*(s,a) = \max_{\pi} \mathbb{E}\Big[R_t \;\Big|\; s_t = s, a_t = a, \pi\Big]$$

$Q^*(s,a) =$ *What's the expected discounted return if we execute action a in state s and then follow the optimal policy*

By this definition $\max_a Q^*(s,a)$ is the optimal policy

$$Q^*(s,a) = \mathbb{E}_{s'}\Big[r + \gamma \max_{a'} Q^*(s',a')\Big]$$

State-Action function

We define

$$Q^*(s,a) = \max_\pi \mathbb{E}\Big[R_t \;\Big|\; s_t = s, a_t = a, \pi\Big]$$

$Q^*(s,a) =$ *What's the expected discounted return if we execute action a in state s and then follow the optimal policy*

By this definition $\max_a Q^*(s,a)$ is the optimal policy

$$Q^*(s,a) = \mathbb{E}_{s'}\Big[r + \gamma \max_{a'} Q^*(s',a')\Big]$$

This identity is know as *Bellman equation*

Q-Learning

Idea: Learn State-Action function by performing iterative Bellman updates

Q-Learning

Idea: Learn State-Action function by performing iterative Bellman updates

$$Q_{i+1}(s,a) := \mathbb{E}_{s'}\left[r + \gamma \max_{a'} Q_i(s',a')\right]$$

Q-Learning

Idea: Learn State-Action function by performing iterative Bellman updates

$$Q_{i+1}(s, a) := \mathbb{E}_{s'}\left[r + \gamma \max_{a'} Q_i(s', a')\right]$$

This is known as *value iteration* algorithm and has been shown to converge to Q^* for $i \to \infty$

Q-Learning sampling

Learn State-Action function from samples (s, a, r, s') :

$$Q^*(s,a) \approx r + \gamma \max_{a'} Q^*(s', a')$$

Q-Learning sampling

Learn State-Action function from samples (s, a, r, s') :

$$Q^*(s, a) \approx r + \gamma \max_{a'} Q^*(s', a')$$

with

$$\pi(s) = \operatorname*{argmax}_a Q(s, a)$$

Q-Learning sampling

Learn State-Action function from samples (s, a, r, s') :

$$Q^*(s,a) \approx r + \gamma \max_{a'} Q^*(s', a')$$

with

$$\pi(s) = \operatorname*{argmax}_a Q(s,a)$$

or ε-greedy:

$$\pi(s) = \begin{cases} \operatorname*{argmax}_a Q(s,a) & \text{with probability } 1-\varepsilon \\ a \sim U(A) & \text{with probability } \varepsilon \end{cases}$$

Q-Learning with tables

Q-Table

s	a	$Q(s,a)$
$\vdots$	$\vdots$	$\vdots$
$\vdots$	$\vdots$	$\vdots$

Q-Learning with tables

Q-Table

s	a	$Q(s,a)$
$\vdots$	$\vdots$	$\vdots$
$\vdots$	$\vdots$	$\vdots$

Questions?
(you need to implement such a table as part of the homework)

Beyond tables

- Using a table to store the Q function is inefficient
  - Sparse entries
  - No generalization

Beyond tables

- Using a table to store the Q function is inefficient
  - Sparse entries
  - No generalization
- Idea: Let's use a "deep" neural net $Q_\theta(s, a)$ to approximate $Q^*(s, a)$

Beyond tables

- Using a table to store the Q function is inefficient
  - Sparse entries
  - No generalization
- Idea: Let's use a "deep" neural net $Q_\theta(s, a)$ to approximate $Q^*(s, a)$

Mnih, Volodymyr, et al. "Human-level control through deep reinforcement learning." Nature 518.7540 (2015): 529

Training procedure of a Deep Q-Network (DQN)

Training procedure of a Deep Q-Network (DQN)

Target value: For every sample (s, a, r, s') compute

$$\hat{q} := r + \gamma \max_{a'} Q_\theta(s', a')$$

Training procedure of a Deep Q-Network (DQN)

Target value: For every sample (s, a, r, s') compute

$$\hat{q} := r + \gamma \max_{a'} Q_\theta(s', a')$$

With squared error loss:

$$L(Q_\theta, \hat{q}) := \Big(Q_\theta(s, a) - \hat{q}\Big)^2$$

and gradient descent:

$$\theta_{i+1} := \theta_i - \alpha \frac{dL}{d\theta}$$

How to encode Q-Network?

$$Q_\theta : S \times A \to \mathbb{R}$$

How to encode Q-Network?

$$Q_\theta : S \times A \to \mathbb{R}$$

DQN: First attempt:

```
s_input = tf.placeholder(tf.float32,shape=[state_dim])
a_input = tf.placeholder(tf.float32,shape=[action_dim])

x = tf.concat([s_input,a_input],axis=0)
h1 = tf.layers.dense(x,units=100,activation=tf.nn.tanh)
q_prediction = tf.layers.dense(h1,units=1)
```

How to encode Q-Network?

$$Q_\theta : S \times A \rightarrow \mathbb{R}$$

DQN: First attempt:

```
s_input = tf.placeholder(tf.float32,shape=[state_dim])
a_input = tf.placeholder(tf.float32,shape=[action_dim])

x = tf.concat([s_input,a_input],axis=0)
h1 = tf.layers.dense(x,units=100,activation=tf.nn.tanh)
q_prediction = tf.layers.dense(h1,units=1)
```

Question: Why is that a bad idea?

How to encode Q-Network?

$$Q_\theta : S \times A \to \mathbb{R}$$

DQN: First attempt:

```
s_input = tf.placeholder(tf.float32,shape=[state_dim])
a_input = tf.placeholder(tf.float32,shape=[action_dim])

x = tf.concat([s_input,a_input],axis=0)
h1 = tf.layers.dense(x,units=100,activation=tf.nn.tanh)
q_prediction = tf.layers.dense(h1,units=1)
```

Question: Why is that a bad idea?

$\max_{a'} Q_\theta(s', a')$ requires $|A|$ evaluations of the network

How to encode a Q-Network?

$$Q_\theta : S \to \mathbb{R}^{|A|}$$

How to encode a Q-Network?

$$Q_\theta : S \rightarrow \mathbb{R}^{|A|}$$

DQN:
Second attempt:

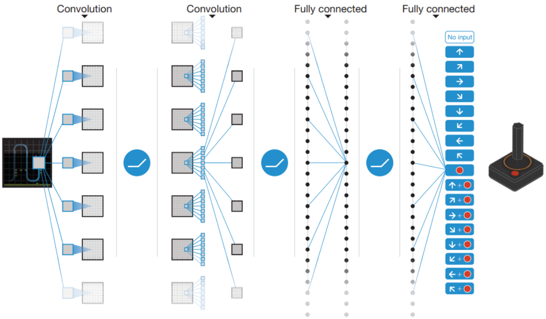
```
s_input = tf.placeholder(tf.float32,shape=[state_dim])
h1 = tf.layers.dense(s_input,units=100,activation=tf.nn.tanh)
q_prediction = tf.layers.dense(h1,units=num_of_possible_actions)
```

How to encode a Q-Network?

$$Q_\theta : S \to \mathbb{R}^{|A|}$$

DQN:
Second attempt:

```
s_input = tf.placeholder(tf.float32,shape=[state_dim])
h1 = tf.layers.dense(s_input,units=100,activation=tf.nn.tanh)
q_prediction = tf.layers.dense(h1,units=num_of_possible_actions)
```



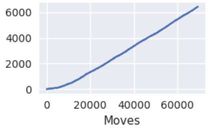
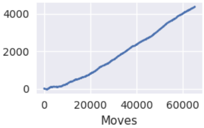
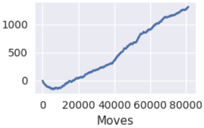
How to train a Q-Network?

How to train a Q-Network?

```python
# ... Build the computation graph
target_q = tf.placeholder(tf.float32)
target_index = tf.placeholder(tf.int32)
loss = tf.square(target_q - q_prediction) \
        * tf.one_hot(target_index,num_of_possible_actions)
update_step = tf.train.GradientDescentOptimizer(0.001).minimize(loss)

# ... Learning updates
def updateQ(s,a,r,s_prime):
    q_next = tf_session.run(q_prediction,{s_input:s_prime})
    q_max = np.max(q_next)
    # Warning: Make sure that max is actually a valid action

    q = r + 0.99 * q_max

    tf_session.run(update_step,{s_input:s,target_index:a,target_q:q})

# ... Training loop
state = env.reset()
for _ in range(1000):
    action = policy(state)
    next_state, reward, done, info = env.step(action)

    # Learn
    updateQ(state,action,reward,next_state)

    state = next_state
```

How does a Q-Network perform?

Cumulative reward over time on Tic-Tac-Toe

vs random	vs smart	vs smarter
		

Questions so far?

Experience Replay Buffer

Experience Replay Buffer

- Batch multiple (s, a, r, s') updates together
  -
    ```
    # Replace
    s_input = tf.placeholder(tf.float32,shape=[state_dim])
    # with
    s_input = tf.placeholder(tf.float32,shape=[None,state_dim])
    ```
  - *Stabilizes* learning

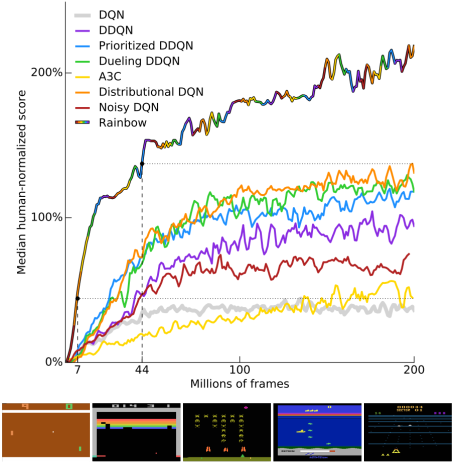
Experience Replay Buffer

- Batch multiple (s, a, r, s') updates together
  -
    ```
    # Replace
    s_input = tf.placeholder(tf.float32,shape=[state_dim])
    # with
    s_input = tf.placeholder(tf.float32,shape=[None,state_dim])
    ```
  - *Stabilizes* learning
- Store (s, a, r, s') in a buffer an re-use multiple times
  - Increases efficiency

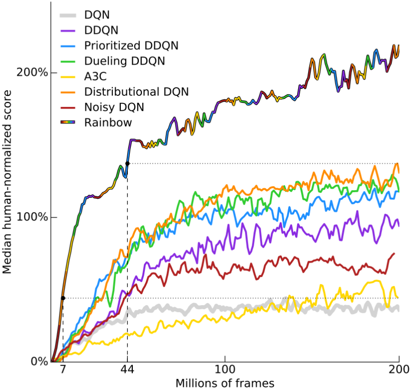
Advanced Methods

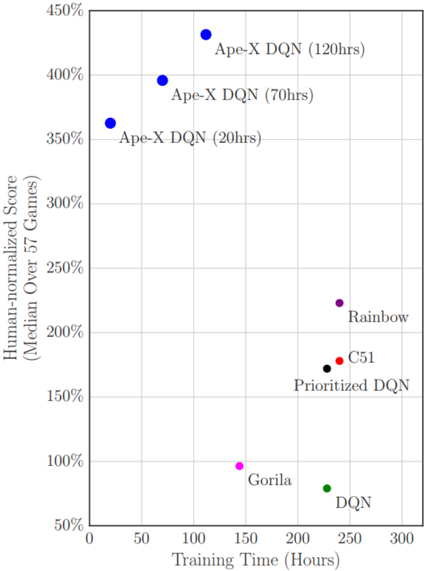
Method	Included in
Experience Replay Buffer	DQN (2013/2015)
Double Q-Learning Prioritized Experience Replay Duelling Q networks Multistep-Learning Distributional DQN Noisy Nets	Rainbow (2017)
Distributed Prioritized Experience Replay	Ape-X (2018)

Performance



Performance





Conclusion

- Q-Learning with tables: $Q_{i+1}(s,a) := r + \gamma \max_{a'} Q_i(s',a')$
  - Poor scaling to large action/state spaces
  - No generalization
- Solution: Approximation with neural net
  - No convergence guarantee to Q^*
- Active research on improving Q-Learning
  - e.g Experience Replay Buffer