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
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 \(L(y, \hat{y})\)
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
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 \( P_0 = \Pr[s_0] \)
  - Transition probability
    \( P(s, a, s') = \Pr[s'|s, a] \)
  - Reward function \( R : S \rightarrow \mathbb{R} \)

- Policy \( \pi : S \rightarrow A \)
Markov-Decision-Process (MDP)

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

- Policy $\pi : S \rightarrow A$
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 \( R : S \rightarrow \mathbb{R} \)

- Policy \( \pi : S \rightarrow A \)

```
state = env.reset()
for _ in range(1000):
    action = policy(state)
    state, reward, done, info \n    = 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 \(R : S \rightarrow \mathbb{R}\)

- Policy \(\pi : S \rightarrow A\)

\[
\begin{array}{c}
S_0 \xrightarrow{a_0} \\
\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad X \\
\end{array}
\]
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 \( R : S \rightarrow \mathbb{R} \)

- Policy \( \pi : S \rightarrow A \)

\[
\begin{array}{c}
s_0 \xrightarrow{a_0} r_0, s_1 \\
r_0 = 0
\end{array}
\]
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 \( R : S \rightarrow \mathbb{R} \)

- Policy \( \pi : S \rightarrow A \)

\[
s_0 \xrightarrow{a_0} r_0, s_1 \xrightarrow{a_1} r_0 = 0
\]
Markov-Decision-Process (MDP)

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

- Policy \(\pi : S \rightarrow A\)

\[
\begin{array}{c}
  s_0 \xrightarrow{a_0} r_0, s_1 \xrightarrow{a_1} r_1, s_2 \\
r_0 = 0 \\
r_1 = 0
\end{array}
\]
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 $R : S \rightarrow \mathbb{R}$

- Policy $\pi : S \rightarrow A$

\[ s_0 \xrightarrow{a_0} r_0, s_1 \xrightarrow{a_1} r_1, s_2 \xrightarrow{a_2} r_2 \]

\[
\begin{array}{c|c|c|c}
S & X & \times & \times \\
\times & 0 & X & \times \\
\times & 0 & & \\
\end{array}
\]

You lost!
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 \(R : S \rightarrow \mathbb{R}\)

- Policy \(\pi : S \rightarrow A\)

\[s_0 \xrightarrow{a_0} r_0, \quad s_1 \xrightarrow{a_1} r_1, \quad s_2 \xrightarrow{a_2} r_2, \quad s_3\]

\[
\begin{array}{c|cc}
    & 0 & x \\
\hline
    x & 0 & x \\
    0 & & 0
\end{array}
\]

You lost!
Let’s say we are in an arbitrary state $s_t$

The optimal action would maximize sum of future rewards $r_t, r_t+1, \ldots r_{t+n}$
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:

$$\max_\pi \mathbb{E}\left[ \sum_{i=t}^{t+n} r_i \gamma^{(i-t)} \bigg| \pi \right]$$

$$R_t$$
Different approaches to RL

- Reinforcement Learning
  - Model based
  - Model free
    - Q-Learning
    - Policy gradient
    - Gradient-free approaches
Different approaches to RL

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

\[ Q^*(s, a) = \max_{\pi} \mathbb{E}[R_t \mid s_t = s, a_t = a, \pi] \]
We define

$$Q^*(s, a) = \max \mathbb{E}_{\pi} \left[ R_t \mid s_t = s, a_t = a, \pi \right]$$

$$Q^*(s, a) = \text{What’s the expected discounted return if we execute action } a \text{ in state } s \text{ and then follow the optimal policy}$$
We define

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

\( Q^*(s, a) = \text{What’s the expected discounted return if we execute action } a \text{ in state } s \text{ and then follow the optimal policy} \)

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

$$Q^*(s, a) = \max_{\pi} \mathbb{E} \left[ R_t \mid s_t = s, a_t = a, \pi \right]$$

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

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

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

$$Q^*(s, a) = \max_{\pi} \mathbb{E}\left[ R_t \mid s_t = s, a_t = a, \pi \right]$$

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

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

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

This identity is known as Bellman equation.
Idea: Learn State-Action function by performing iterative Bellman updates

$$Q_{i+1}(s,a) := E_{s'}[r + \gamma \max_{a'} Q_i(s',a')]$$

This is known as value iteration algorithm and has been shown to converge to $$Q^*$$ for $$i \to \infty$$. 
Idea: Learn State-Action function by performing iterative Bellman updates

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

\[
\pi(s) = \arg\max_a Q(s, a)
\]
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

\[ \pi(s) = \arg\max_a Q(s, a) \]

or \(\varepsilon\)-greedy:

\[ \pi(s) = \begin{cases} 
\arg\max_a Q(s, a) & \text{with probability } 1 - \varepsilon \\
 a \sim U(A) & \text{with probability } \varepsilon 
\end{cases} \]
Q-Learning with tables

### Q-Table

<table>
<thead>
<tr>
<th>$s$</th>
<th>$a$</th>
<th>$Q(s, a)$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Q-Learning with tables

<table>
<thead>
<tr>
<th>$s$</th>
<th>$a$</th>
<th>$Q(s, a)$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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 $Q^*(s, a)$

---

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:

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

and gradient descent:

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

\[
L(Q_\theta, \hat{q}) := \left( Q_\theta(s, a) - \hat{q} \right)^2
\]

and gradient descent:

\[
\theta_{i+1} := \theta_i - \alpha \frac{dL}{d\theta}
\]
How to encode Q-Network?

$$Q_\theta : S \times A \rightarrow \mathbb{R}$$
How to encode Q-Network?

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

DQN: First attempt:

```python
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:

```python
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?

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

DQN: First attempt:

```python
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?

\[
\max_{a'} Q_\theta(s', a') \text{ requires } |A| \text{ evaluations of the network}
\]
How to encode a Q-Network?

\[ Q_\theta : S \rightarrow \mathbb{R}^{|A|} \]
How to encode a Q-Network?

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

DQN:
Second attempt:

```python
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:

```python
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?

Mathias Lechner (IST Austria)

Q-Learning

January 20, 2019
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) \n      * 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

<table>
<thead>
<tr>
<th>vs random</th>
<th>vs smart</th>
<th>vs smarter</th>
</tr>
</thead>
</table>

Questions so far?
Experience Replay Buffer

Store \((s, a, r, s')\) in a buffer and re-use multiple times
- Increases efficiency

Stabilizes learning

- Batch multiple \((s, a, r, s')\) updates together
Experience Replay Buffer

- Batch multiple \((s, a, r, s')\) updates together

```python
# 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

  ```python
  # 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 and re-use multiple times
  - Increases efficiency
### Advanced Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Included in</th>
</tr>
</thead>
<tbody>
<tr>
<td>Experience Replay Buffer</td>
<td>DQN (2013/2015)</td>
</tr>
<tr>
<td>Double Q-Learning</td>
<td>Rainbow (2017)</td>
</tr>
<tr>
<td>Prioritized Experience Replay</td>
<td></td>
</tr>
<tr>
<td>Duelling Q networks</td>
<td></td>
</tr>
<tr>
<td>Multistep-Learning</td>
<td></td>
</tr>
<tr>
<td>Distributional DQN</td>
<td></td>
</tr>
<tr>
<td>Noisy Nets</td>
<td></td>
</tr>
<tr>
<td>Distributed Prioritized Experience Replay</td>
<td>Ape-X (2018)</td>
</tr>
</tbody>
</table>
Performance

Median human-normalized score vs. Millions of frames for different algorithms:
- DQN
- DDQN
- Prioritized DDQN
- Dueling DDQN
- A3C
- Distributional DQN
- Noisy DQN
- Rainbow

Human-normalized score (Median over 57 Games) vs. Training Time (Hours):
- Ape-X DQN (120hrs)
- Ape-X DQN (70hrs)
- Ape-X DQN (20hrs)
- Rainbow
- C51
- Prioritized DQN
- Gorilla
- DQN
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