Reinforcement Learning

Atari Games

Isabell Lederer
github.com/mathebell/RL_Atari_Games

AKNUM Maschinelles Lernen SE

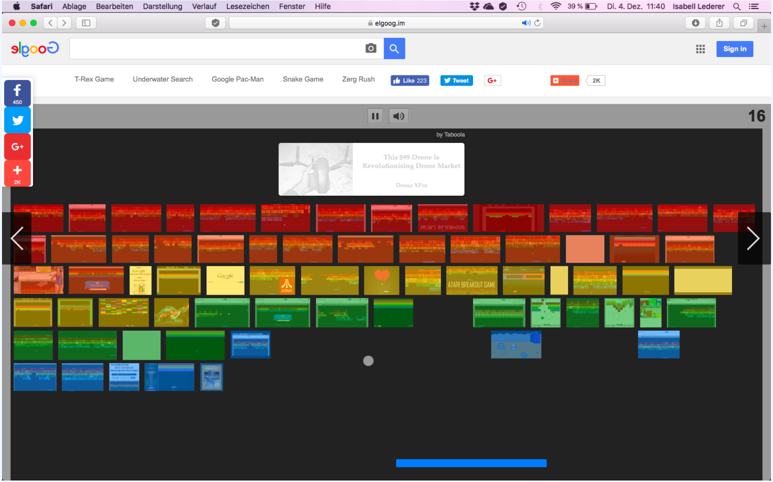
Wien, 18. Dezember 2018

Contents:

Atari Games

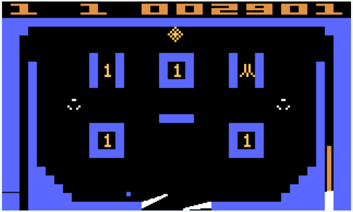

POWER
ON
OFF
TV TYPE
B•W
VIDEO COMPUTER SYSTEM™
GAME SELECT
GAME RESET
ATARI 2600

Atari Games

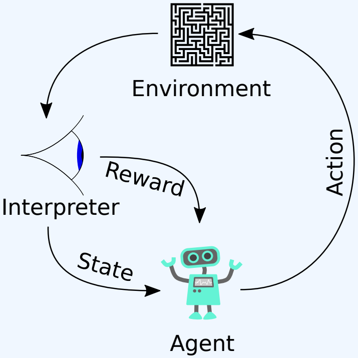


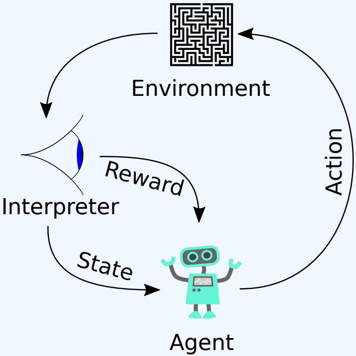
Breakout at https://elgoog.im/breakout/

Some other Atari games



Reinforcement Learning - Basics





- policy function:

$$\pi(a \mid s) = \Pr(A_t = a \mid S_t = s)$$

- new problem

action-value methods vs. policy gradient methods

methods that approximate an action-value function followed by a policy

Action-value methods - Dynamic Programming

- perfect model of environment as Markov Decision Process (MDP)

- perfect model of environment as Markov Decision Process (MDP)
- dynamics function p:

$$p(s', r \mid s, a) = \Pr(S_t = s', R_t = r \mid S_{t-1} = s, A_{t-1} = a)$$

- optimal state-value function v_* and action-value function q_* via Bellmann equations:

$$\begin{aligned} v_*(s) &= \max_a \mathbb{E}\left[R_{t+1} + \gamma v_*(S_{t+1}) \,|\, S_t = s, A_t = a\right] \\ &= \max_a \sum_{s',r} p(s', r \,|\, s, a)\left[r + \gamma v_*(s')\right], \\ q_*(s,a) &= \mathbb{E}\left[R_{t+1} + \gamma \max_{a'} q_*(S_{t+1}, a') \,|\, S_t = s, A_t = a\right] \\ &= \sum_{s',r} p(s', r \,|\, s, a)\left[r + \gamma \max_{a'} q_*(s', a')\right]. \end{aligned}$$

- characteristic for DP: perfect model as a MDP, approximate optimal value function, bootstrapping

- off-policy algorithm that can directly learn from raw experience without a model of the environment's dynamic

- off-policy algorithm that can directly learn from raw experience without a model of the environment's dynamic
- it has been shown that under some assumptions Q converges with probability 1 to q_*

- off-policy algorithm that can directly learn from raw experience without a model of the environment's dynamic
- it has been shown that under some assumptions Q converges with probability 1 to q_*
- Q-learning iteration:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)].$$

Q-learning Pseudocode

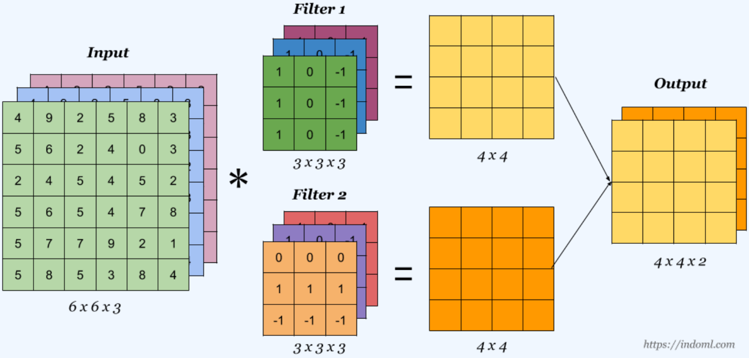
Parameters: step size $\alpha \in (0, 1]$, small $\epsilon > 0$
Initialize $Q(s, a)$, for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$

Loop for each episode:
    Initialize S
    Loop for each step of episode:
        Choose A by ϵ-greedy selection
        Take action A, observe R, S'
        $Q(S, A) \leftarrow Q(S, A) + \alpha \left[R + \gamma \max_a Q(S', a) - Q(S, A)\right]$
    until S is terminal

DQN - Deep Q-Network

DQN - Deep Q-Network

- combines the idea of Q-learning with a deep convolutional network



- learn parametrized policy

$$\pi(a \mid s, \theta) = \Pr(A_t = a \mid S_t = s, \theta_t = \theta)$$

Policy Gradient Methods

- learn parametrized policy

$$\pi(a \mid s, \theta) = \Pr(A_t = a \mid S_t = s, \theta_t = \theta)$$

- learn the parameter θ based on the gradient of some scalar performance measure $J(\theta)$

- learn parametrized policy

$$\pi(a \mid s, \boldsymbol{\theta}) = \Pr(A_t = a \mid S_t = s, \boldsymbol{\theta}_t = \boldsymbol{\theta})$$

- learn the parameter $\boldsymbol{\theta}$ based on the gradient of some scalar performance measure $J(\boldsymbol{\theta})$
- updating via gradient ascent

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha \widehat{\nabla J(\boldsymbol{\theta}_t)}$$

- can be parametrized in any way as long as the policy is differentiable
- to ensure exploration we require that $\pi(a \mid s, \theta)$ never becomes deterministic
- advantage: can approach a deterministic policy

Policy Gradient Methods

- can be parametrized in any way as long as the policy is differentiable
- to ensure exploration we require that $\pi(a \mid s, \theta)$ never becomes deterministic
- advantage: can approach a deterministic policy
- one can proof better performance for policy gradient methods than for action value methods

- define performance as

$$J(\boldsymbol{\theta}) := v_{\pi_{\boldsymbol{\theta}}}(s_0).$$

- define performance as

$$J(\theta) := v_{\pi_\theta}(s_0).$$

- $\widehat{\nabla J(\theta_t)}$ = ???

Policy Gradient Theorem

Theorem

Let $\pi(a \mid s, \theta)$ be a parameterized policy, $q_\pi(s, a)$ the action-value function under π and $\mu(s)$ the on-policy distribution under π, thus

$$\nabla J(\theta) \propto \sum_s \mu(s) \sum_a q_\pi(s, a) \nabla \pi(a \mid s, \theta).$$

Proof.

$$
\begin{aligned}
\nabla v_\pi(s) &= \nabla\Big[\sum_a \pi(a|s) q_\pi(s,a)\Big] && (1)\\
&= \sum_a \Big[\nabla\pi(a|s) q_\pi(s,a) + \pi(a|s)\nabla q_\pi(s,a)\Big] && (2)\\
&= \sum_a \Big[\nabla\pi(a|s) q_\pi(s,a) + \pi(a|s) \\
&\qquad \nabla\Big[\sum_{s',r} p(s',r|s,a)\left(r + v_\pi(s')\right)\Big]\Big] && (3)\\
&= \sum_a \Big[\nabla\pi(a|s) q_\pi(s,a) + \pi(a|s)\sum_{s'} p(s'|s,a)\nabla v_\pi(s')\Big] && (4)
\end{aligned}
$$

Policy Gradient Theorem

$$
\begin{aligned}
(1): &\quad v_\pi(s) = \sum_a \pi(s|a) q_\pi(s,a) \\
(1) \to (2): &\quad \text{product rule} \\
(2) \to (3): &\quad q_\pi(s,a) = \sum_{s',r} p(s',r|s,a)\left(r + v_\pi(s')\right) \\
(3) \to (4): &\quad p(s'|s,a) = \sum_r p(s',r|s,a)
\end{aligned}
$$

Proof.

$$= \sum_a \Big[\nabla \pi(a|s) q_\pi(s,a) + \pi(a|s) \sum_{s'} p(s'|s,a)$$
$$\sum_{a'} \Big[\nabla \pi(a', s') q_\pi(s', a') + \pi(a'|s') \sum_{s''} p(s''|s', a') \nabla V_\pi(s'') \Big] \Big] \tag{5}$$

$$= \sum_{x \in \mathcal{S}} \sum_{k=0}^{\infty} \Pr(s \to x, k, \pi) \sum_a \nabla \pi(a|x) q_\pi(x, a), \tag{6}$$

where $\Pr(s \to x, k, \pi)$ is the probability of transitioning from state s to state x in k steps under policy π.
$\eta(s)$ denote the number of time steps spent, on average, in s.
On-policy distribution is then $\mu(s) = \frac{\eta(s)}{\sum_{s'} \eta(s')}$.

Policy Gradient Theorem

Proof.

$$
\begin{aligned}
\nabla J(\theta) &= \nabla v_\pi(s_0) && (7)\\
&= \sum_s \left(\sum_{k=0}^{\infty} \Pr(s_0 \to s, k, \pi) \right) \sum_a \nabla \pi(a|s) q_\pi(s,a) && (8)\\
&= \sum_s \eta(s) \sum_a \nabla \pi(a|s) q_\pi(s,a) && (9)\\
&= \sum_{s'} \eta(s') \sum_s \frac{\eta(s)}{\sum_{s'} \eta(s')} \sum_a \nabla \pi(a|s) q_\pi(s,a) && (10)\\
&= \sum_{s'} \eta(s') \sum_s \mu(s) \sum_a \nabla \pi(a|s) q_\pi(s,a) && (11)\\
&\propto \sum_s \mu(s) \sum_a \nabla \pi(a|s) q_\pi(s,a) && (12)
\end{aligned}
$$

$\square$

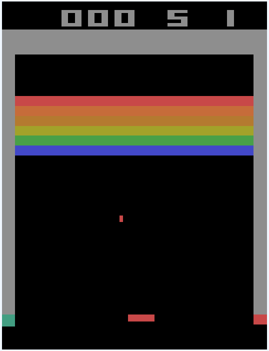
Implementation with Python

- OpenAI's Gym is a toolkit for developing and comparing reinforcement learning algorithms. It supports teaching agents everything from walking to playing games like Pong or Pinball.

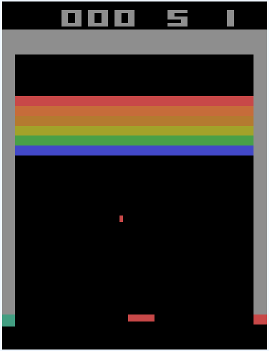
Emulator

- OpenAI's Gym is a toolkit for developing and comparing reinforcement learning algorithms. It supports teaching agents everything from walking to playing games like Pong or Pinball.
- Arcade Learning Environment (ALE)

- Input: 3 Atari frames in 210×160 pixels in RGB color



- Input: 3 Atari frames in 210×160 pixels in RGB color
- a lot of data for memory: memorize recent $1,000,000$ of $(S_t, A_t, R_t, S_{t+1}, \texttt{isTerminal})$



- Rescaling, e.g. from 210×160 pixels to 84×84

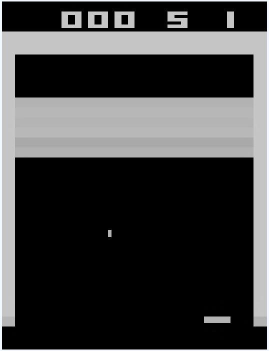
- Rescaling, e.g. from 210×160 pixels to 84×84
- Greyscaling

Preprocessing Input/images

- Rescaling, e.g. from 210 $\times$ 160 pixels to 84 $\times$ 84
- Greyscaling

scikit-image
image processing in python

- Rescaling, e.g. from 210×160 pixels to 84×84
- Greyscaling



Storing memory

- adequate collection for memory I can sample on
- deque perfect for needs, but bad runtime
- implement own RingBuffer?

Model architecture of DQN

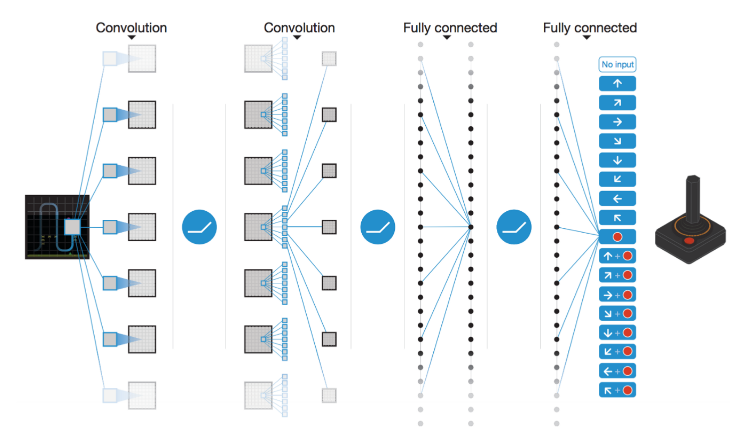
```
#Create Model
ATARI_SHAPE = (105, 80,3)

# Inputs
frames_input = keras.layers.Input(shape = ATARI_SHAPE, name = 'frames')
actions_input = keras.layers.Input(shape = (n_actions,), name = 'actions')

# "The first hidden layer convolves 16 8×8 filters with stride 4 with the input image
# and applies a rectifier nonlinearity."
conv_1 = keras.layers.convolutional.Conv2D(16, (8, 8), strides = 4, activation = 'relu')(frames_input)
# "The second hidden layer convolves 32 4×4 filters with stride 2, again followed by a
# rectifier nonlinearity."
conv_2 = keras.layers.convolutional.Conv2D(32,(4, 4),strides = 2, activation = 'relu')(conv_1)
# Flattening the second convolutional layer.
conv_flattened = keras.layers.core.Flatten()(conv_2)
# "The final hidden layer is fully-connected and consists of 256 rectifier units."
hidden = keras.layers.Dense(256, activation = 'relu')(conv_flattened)
# "The output layer is a fully-connected linear layer with a single output for each
# valid action."
output = keras.layers.Dense(n_actions)(hidden)
filtered_output = keras.layers.multiply([output, actions_input]) # get corresponding Q val

model = keras.models.Model(input = [frames_input, actions_input], output = filtered_output)
# parameters from paper
optimizer = keras.optimizers.RMSprop(lr = 0.00025, rho = 0.95, epsilon = 0.01)
model.compile(optimizer, loss='mse')
```

Model architecture of DQN



Implementation with keras/ tensorflow

Pseudocode DQN

Pseudocode

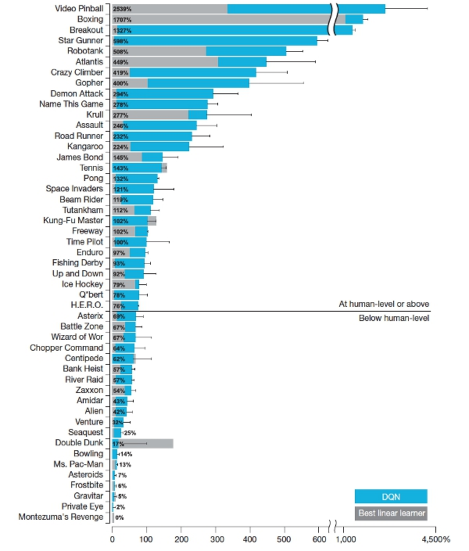
```
env = gym.make('BreakoutDeterministic-v4')
Create model, initialize memory D
Loop for each episode:
    s = env.reset() and preprocess s
    env.render()
    Loop for each step of episode:
        Choose ε, choose with probability ε a random action,
        otherwise choose best action predicted by model
        s', r, is_done, info = env.step(a)
        preprocess s' and append (s, a, s', is_done) to D
        sample batch from D, predict Q_{t+1}, update Q_t
        fit batch of (s,a) with Q_t to model
       env.render()
    until S is terminal
```

Youtube movie of training

Video of Playing Atari with Deep Reinforcement Learning:
`https://www.youtube.com/watch?v=V1eYniJ0Rnk`

Results

Performance of DQN agent on various Atari games



Questions?

Literatur I

R. S. Sutton, A. G. Barto.
***Reinforcement Learning - An Introduction*.**
The MIT Press, 2018.

Google DeepMind.
Human-level control through deep reinforcement learning.
Nature, doi:10.1038/nature14236, 2015.