Playing Atari with Deep Q-learning

Kulaga Roman

02.11.2017

Atari 2600

Figure: Atari 2600 with joystick

- CPU: 1.19MHz
- RAM: 128 bytes
- Controller: stick with 8 positions, 1 button (18 actions)

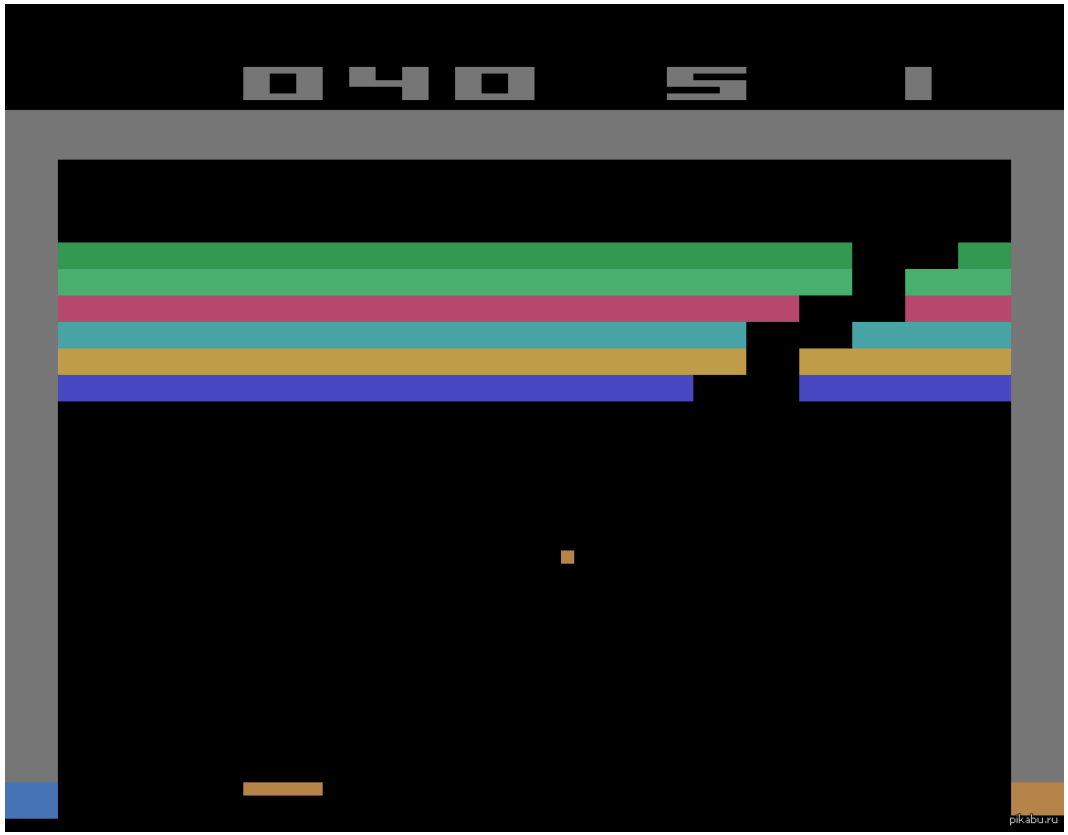
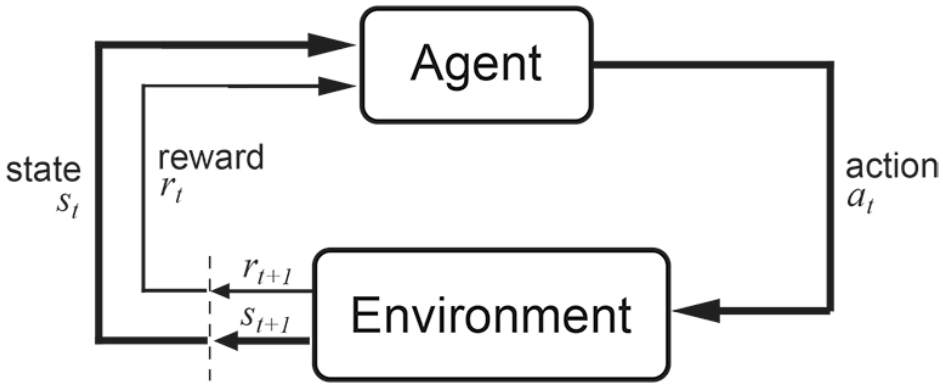


Figure: Atari 2600 game: Breakout

Reinforcement Learning

- agent (which sees states and rewards and decides on actions)
- environment (which sees actions, changes states and gives rewards)



The agent's goal is to maximize the discounted sum of rewards during the game

---

blog.deepsense.ai

- Environment $\mathcal{E}$ (can be stochastic)
- Action $a_t \in \mathcal{A} = \{1, \ldots, K\}$
- Image of emulator's internal state $x_t \in \mathbb{R}^d$
- Observation $s_t = x_1, a_1, x_2, a_2, \ldots, x_{t-1}, a_{t-1}, x_t$
- Future discounted return: $R_t = \sum_{t'=t}^{T} \gamma^{t'-t} r_{t'}$

Q-values

- Let's assume that the agent's strategy (the choice of the action in a given state) is fixed.
- Q-value of a state-action pair (s, a) is the cumulative discounted reward the agent will get if it is in a state s, executes the action a and follows his strategy from there on:

$$Q(s, a) = \max_{\pi} \mathbb{E}_{s' \sim \mathcal{E}} \left[R_t \middle| s_t = s, a_t = a, \pi \right]$$

If a strategy is optimal, the following holds:

$$Q(s_t, a_t) = \mathbb{E}_{s' \sim \mathcal{E}} \left[r_t + \gamma \max_{a'} Q(s_{t+1}, a') \middle| s_t, a \right]$$

Q-learning

If the environment $\mathcal{E}$ is not stochastic:

$$Q(s_t, a_t) = r_t + \gamma \max_{a'} Q(s_{t+1}, a')$$

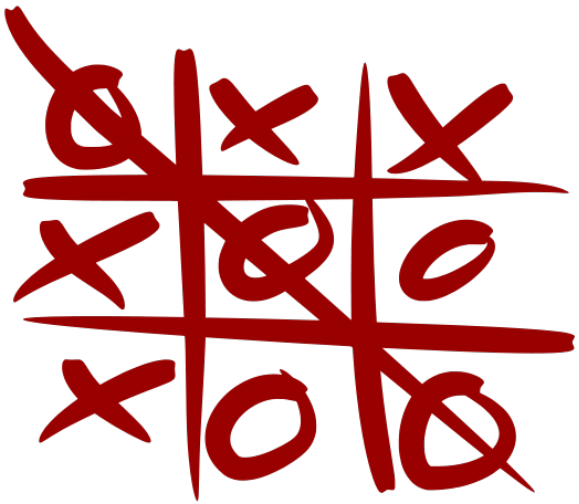
Such equation leads to a simple algorithm (Q-learning):

```
for all (s,a):
    Q[s, a] = 0 #initialize q-values of all state-action pairs
for all s:
    P[s] = random_action() #initialize strategy

# assume we know expected rewards for state-action pairs R[s, a] and
# after making action a in state s the environment moves to the state next_state(s, a)
# alpha : the learning rate - determines how quickly the algorithm learns;
#                             low values mean more conservative learning behaviors,
#                             typically close to 0, in our experiments 0.0002
# gamma : the discount factor - determines how we value immediate reward;
#                               higher gamma means more attention given to far-away goals
#                               between 0 and 1, typically close to 1, in our experiments 0.95

repeat until convergence:
  1. for all s:
        P[s] = argmax_a (R[s, a] + gamma * max_b(Q[next_state(s, a), b]))
  2. for all (s, a):
        Q[s, a] = alpha*(R[s, a] + gamma * max_b Q[next_state(s, a), b]) + (1 - alpha)Q[s, a]
```

Q-learning

Q-learning:

- solves the problem for simple games (like Tic-Tac-Toe)



- is a correct algorithm
- but not an efficient one

Q-learning optimization

We need some form of generalization: when we learn about the value of one state-action pair, we can also improve our knowledge about other similar state-actions.

The deep Q-learning algorithm uses the convolutional neural network as a function approximating the Q-value function.

Deep Q-Network

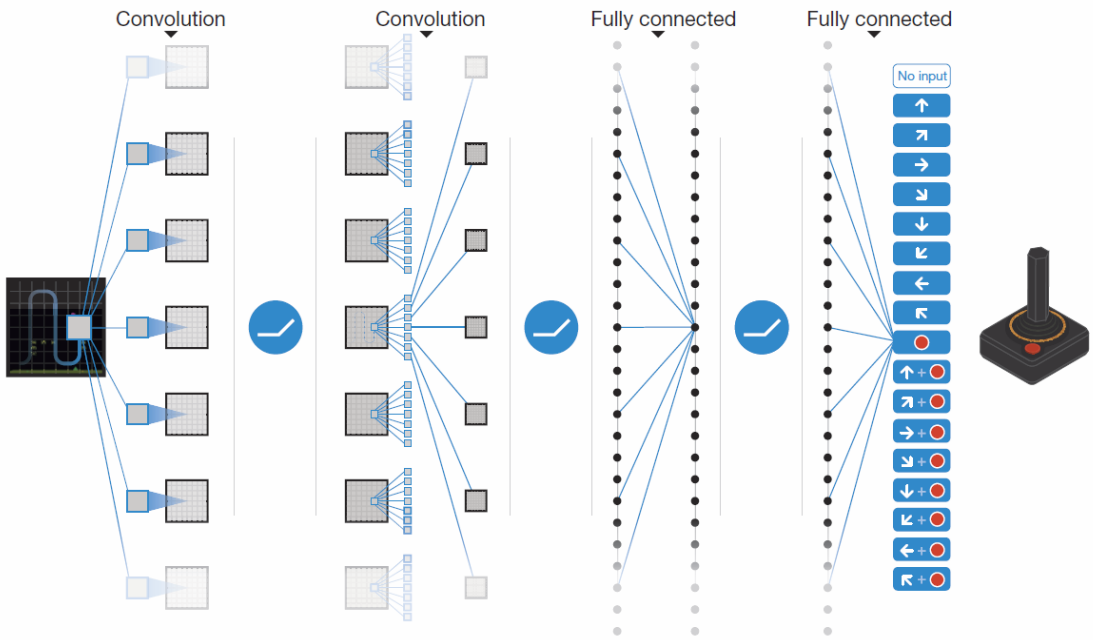


Figure: Schematic illustration of the convolutional neural network. Input - 210 $\times$ 160 images in colour. Output - up to 18 dimensional vector

doi:10.1038/nature14236

Loss function

Let us consider:

- θ is the vector of Deep Q-Network's weights
- $Q(s, a; \theta)$ is the approximate of Q-value.

A Q-network can be trained by minimising a sequence of loss functions $L_i(\theta_i)$ at iteration number i:

$$L_i(\theta_i) = \mathbb{E}_{(s,a)\sim\rho(\cdot)}\left[(y_i - Q(s, a; \theta_i))^2\right],$$

where $y_i = \mathbb{E}_{s'\sim\mathcal{E}}\left[r + \gamma \max_{a'} Q(s', a'; \theta_{i-1})\middle| s, a\right]$ and $\rho(s, a)$ is a behaviour distribution.

Loss function gradient

Differentiating the loss function with respect to θ_i we get:

$$\nabla_{\theta_i} L_i(\theta_i) = \mathbb{E}_{(s,a)\sim\rho(\cdot);s'\sim\mathcal{E}}[\{r + \gamma \max_{a'} Q(s', a'; \theta_{i-1}) - Q(s, a; \theta_i)\} \cdot \nabla_{\theta_i} Q(s, a; \theta_i)] \quad (1)$$

Rather than computing the full expectations in the above gradient, it is often computationally expedient to optimise the loss function by stochastic gradient descent.

Experience replay

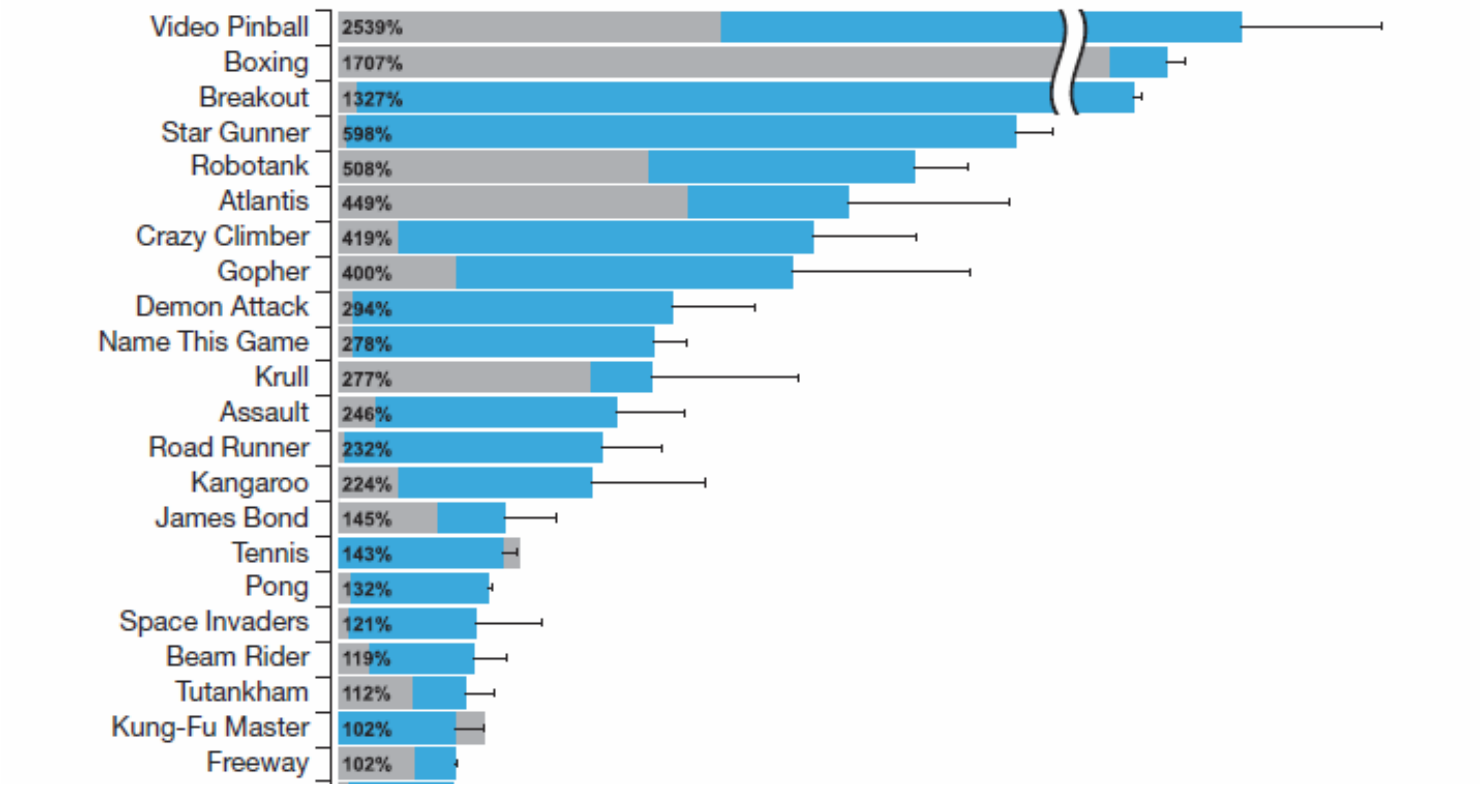
- Agent's experiences at each time-step $e_t = (s_t, a_t, r_t, s_{t+1})$
- Data-set $\mathcal{D} = e_1, \ldots, e_N$ pooled from many episodes into replay memory
- Function $\phi(s)$ produces fixed length representation of history (input to neural network)

Algorithm: Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode $= 1, M$ **do**
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$
  end for
end for

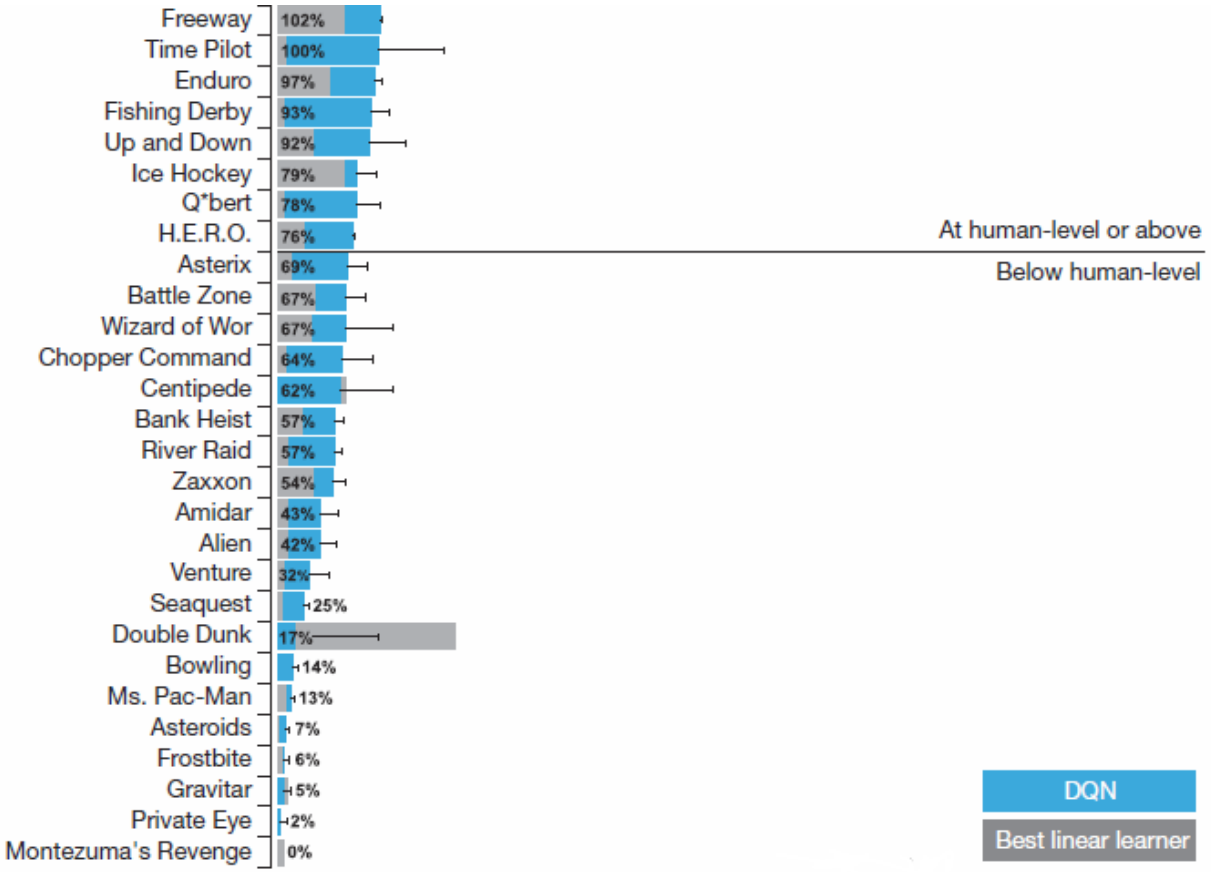
https://arxiv.org/pdf/1312.5602.pdf

Results



doi:10.1038/nature14236

Results



doi:10.1038/nature14236

DQN + RAM

- The idea is to use RAM (128 bytes) instead of game screens
- This makes task easier as input is much smaller
- The information about the game may be hard to retrieve
- The results are comparable (in two games higher, in one lower)

Epsilon-greedy strategy

The first decisions, made with little information, would be reinforced and followed in the future, because we'd stick to these actions for the first states, as their value estimation would be positive (and for the other actions would be nearly zero).

The solution is to use epsilon-greedy strategy: this means that at any time with some small probability ϵ the agent chooses a random action instead of the best action according to Q-value.

Epsilon decay is also useful: we start learning from high $\epsilon \approx 1$ and gradually decrease it to a small value.

Frameskip

- Console generates 60 FPS
- The idea is to repeat one action over N frames and skip $N-1$ frames.
- Makes learning few times faster.
- N can be 4, 8 or even 30 in some games

Reference

- Deepsense.ai blog
- Project at github
- Playing Atari with Deep Reinforcement Learning
- Human-level control through deep reinforcement learning