

Deep Reinforcement Learning

Lecture 7: Learning with Value Function Approximation

Instructor: Chongjie Zhang

Tsinghua University

Function Approximation

- Never enough training data!
 - Must **generalize** what is learned from one situation to other “similar” new situations
- Idea:
 - Instead of using large table to represent V or Q , use a parameterized function
 - The number of parameters should be small compared to number of states (generally exponentially fewer parameters)
 - Learn parameters from experience
 - When we update the parameters based on observations in one state, then our V or Q estimate will also change for other similar states
 - i.e., the parameterization facilitates generalization of experience

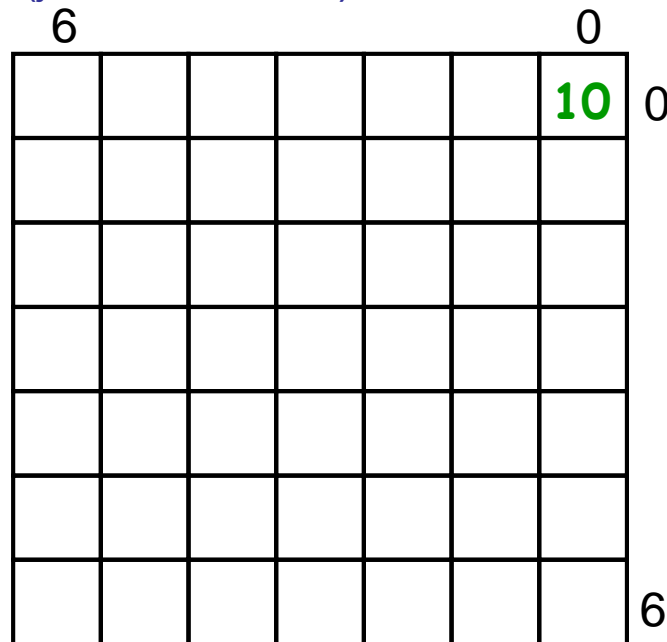
Linear Function Approximation

- Define a set of state features $f_1(s), \dots, f_n(s)$
 - The features are used as our representation of states
 - States with similar feature values will be considered to be similar
- A common approximation is to represent $V(s)$ as a weighted sum of the features (i.e., a linear approximation)

$$\hat{V}_\theta(s) = \theta_0 + \theta_1 f_1(s) + \theta_2 f_2(s) + \dots + \theta_n f_n(s)$$

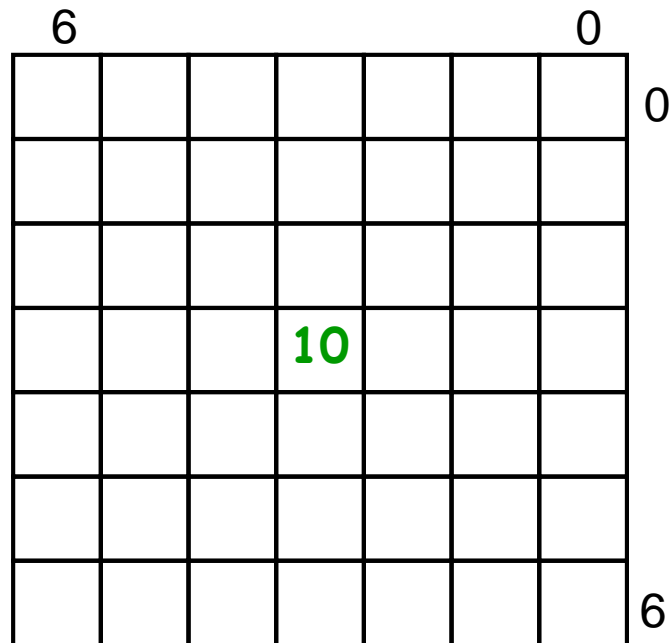
- The approximation accuracy is fundamentally limited by the information provided by the features
- Can we always define features that allow for a perfect linear approximation?
 - Yes. Assign each state an indicator feature. (i.e., i 'th feature is 1 iff i 'th state is present and θ_i represents value of i 'th state)
 - Of course this requires far too many features and gives no generalization.

- Grid with no obstacles, deterministic actions U/D/L/R, no discounting, -1 reward everywhere except +10 at goal
 - Features for state $s=(x, y)$: $f_1(s)=x$, $f_2(s)=y$ (just 2 features)
 - $V(s) = \theta_0 + \theta_1 x + \theta_2 y$
 - Is there a good linear approximation?
 - Yes.
 - $\theta_0=10$, $\theta_1 = -1$, $\theta_2 = -1$
 - (note upper right is origin)
 - $V(s) = 10 - x - y$
subtracts Manhattan dist from goal reward
-



But What If We Change Reward ...

- $V(s) = \theta_0 + \theta_1 x + \theta_2 y$
- Is there a good linear approximation?
 - No.



But What If...

- $V(s) = \theta_0 + \theta_1 x + \theta_2 y + \theta_3 z$

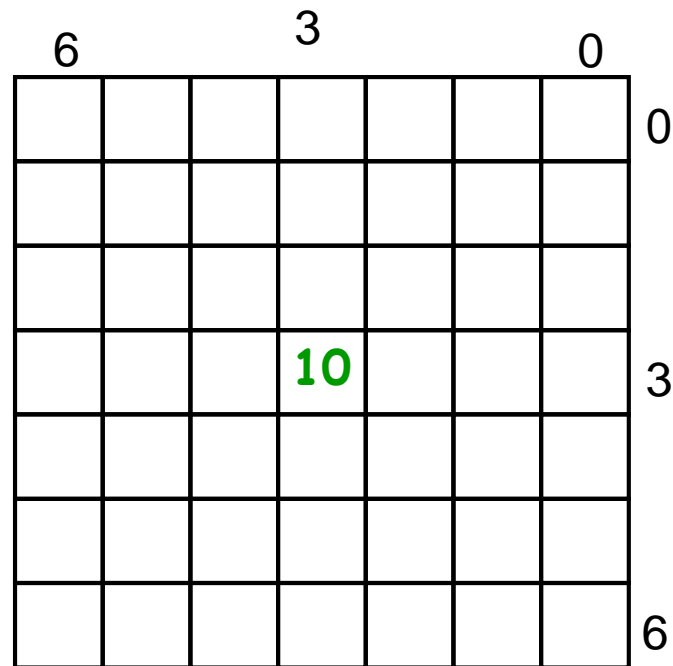
- Include new feature z

- ▲ $z = |3-x| + |3-y|$

- ▲ z is dist. to goal location

- Does this allow a good linear approximation?

- ▲ $\theta_0 = 10, \theta_1 = \theta_2 = 0,$
 $\theta_3 = -1$



Learning with Linear Function Approximation

- Define a set of features $f_1(s), \dots, f_n(s)$
 - The features are used as our representation of states
 - States with similar feature values will be treated similarly
 - More complex functions require more complex features

$$\hat{V}_\theta(s) = \theta_0 + \theta_1 f_1(s) + \theta_2 f_2(s) + \dots + \theta_n f_n(s)$$

- Our goal is to learn good parameter values (i.e. feature weights) that approximate the value function well
 - How can we do this?
 - Use TD-based RL and somehow update parameters based on each experience.

TD-based RL for Linear Approximators

1. Start with initial parameter values
2. Take an action according to an **exploration/exploitation policy** (should converge to a greedy policy, i.e. GLIE)
3. Update estimated model (if model is not available)
4. Perform TD update for each parameter

$$\theta_i \leftarrow ?$$

5. Goto 2

What is a “TD update” for a parameter?

Aside: Gradient Descent

- Given a loss function $E(\theta_1, \dots, \theta_n)$ of n real values $\theta = (\theta_1, \dots, \theta_n)$
 - suppose we want to minimize E with respect to θ
- A common approach to doing this is gradient descent
- The gradient of E at point θ , denoted by $\nabla_{\theta} E(\theta)$, is an n -dimensional vector that points in the direction where f increases most steeply at point θ
- Vector calculus tells us that $\nabla_{\theta} E(\theta)$ is just a vector of partial derivatives

$$\nabla_{\theta} E(\theta) = \left[\frac{\partial E(\theta)}{\partial \theta_1}, \dots, \frac{\partial E(\theta)}{\partial \theta_n} \right]$$

where $\frac{\partial E(\theta)}{\partial \theta_i} = \lim_{\varepsilon \rightarrow 0} \frac{E(\theta_1, \dots, \theta_{i-1}, \theta_i + \varepsilon, \theta_{i+1}, \dots, \theta_n) - E(\theta)}{\varepsilon}$

- Decrease E by moving θ in negative gradient direction

Aside: Gradient Descent for Squared Error

- Suppose that we have a sequence of states and target values for each state
 - E.g. produced by the TD-based RL loop $\langle s_1, v(s_1) \rangle, \langle s_2, v(s_2) \rangle, \dots$
- Our goal is to minimize the sum of squared errors between our estimated function and each target value:

$$E_j(\theta) = \frac{1}{2} \left(\hat{V}_\theta(s_j) - v(s_j) \right)^2$$

squared error of example j our estimated value for j'th state target value for j'th state

- After seeing j'th state the **gradient descent rule** tells us that we can decrease error wrt $E_j(\theta)$ by updating parameters by:

$$\theta_i \leftarrow \theta_i - \alpha \frac{\partial E_j}{\partial \theta_i}$$

learning rate

Aside: continued

$$\theta_i \leftarrow \theta_i - \alpha \frac{\partial E_j}{\partial \theta_i} = \theta_i - \alpha \underbrace{\frac{\partial E_j}{\partial \hat{V}_\theta(s_j)}}_{\hat{V}_\theta(s_j) - v(s_j)} \underbrace{\frac{\partial \hat{V}_\theta(s_j)}{\partial \theta_i}}_{\text{depends on form of approximator}}$$

$$E_j(\theta) = \frac{1}{2} (\hat{V}_\theta(s_j) - v(s_j))^2$$

$$\hat{V}_\theta(s_j) - v(s_j)$$

depends on form of
approximator

- For a linear approximation function:

$$\hat{V}_\theta(s) = \theta_1 + \theta_1 f_1(s) + \theta_2 f_2(s) + \dots + \theta_n f_n(s)$$

$$\frac{\partial \hat{V}_\theta(s_j)}{\partial \theta_i} = f_i(s_j)$$

- Thus the update becomes: $\theta_i \leftarrow \theta_i + \alpha (v(s_j) - \hat{V}_\theta(s_j)) f_i(s_j)$
- For linear functions this update is guaranteed to converge to the best approximation for a suitable learning rate schedule

TD-based RL for Linear Approximators

1. Start with initial parameter values
2. Take an action according to an **exploration/exploitation policy** (should converge to a greedy policy, i.e. GLIE) transitioning from s to s'
3. Update an estimated model
4. Perform TD update for each parameter

$$\theta_i \leftarrow \theta_i + \alpha \left(v(s) - \hat{V}_\theta(s) \right) f_i(s)$$

5. Goto 2

What should we use for “target value” $v(s)$?

- Use the TD prediction based on the next state s'

$$v(s) = R(s) + \gamma \hat{V}_\theta(s')$$

this is the same as previous TD method only with approximation

TD-based RL for Linear Approximators

1. Start with initial parameter values
2. Take an action according to an **exploration/exploitation policy** (should converge to a greedy policy, i.e. GLIE) transitioning from s to s'
3. Update an estimated model
4. Perform TD update for each parameter

$$\theta_i \leftarrow \theta_i + \alpha \left(R(s) + \gamma \hat{V}_\theta(s') - \hat{V}_\theta(s) \right) f_i(s)$$

5. Goto 2

- Step 2 requires a model to select greedy action
- For some applications (e.g. Backgammon) it is easy to get a compact model representation (but not easy to get policy), so TD is appropriate.
- For others it is difficult to small/compact model representation

Q-function Approximation

- Define a set of features over state-action pairs: $f_1(s, a), \dots, f_n(s, a)$
 - State-action pairs with similar feature values will be treated similarly
 - More complex functions require more complex features

$$\hat{Q}_\theta(s, a) = \theta_0 + \theta_1 f_1(s, a) + \theta_2 f_2(s, a) + \dots + \theta_n f_n(s, a)$$

Features are a function of states and actions.

- Just as for TD, we can generalize Q-learning to update the parameters of the Q-function approximation

Q-learning with Linear Approximators

1. Start with initial parameter values
2. Take action a according to an **explore/exploit policy** (should converge to a greedy policy, i.e. GLIE) transitioning from s to s'
3. Perform TD update for each parameter

$$\theta_i \leftarrow \theta_i + \alpha \left(\underbrace{R(s) + \gamma \max_{a'} \hat{Q}_\theta(s', a')}_{\text{estimate of } Q(s,a) \text{ based on an observed transition}} - \hat{Q}_\theta(s, a) \right) f_i(s, a)$$

4. Goto 2

- TD converges close to minimum error solution
- Q-learning can diverge. Converges under some conditions.

Q-learning w/ Non-linear Approximators

$\hat{Q}_\theta(s, a)$ is sometimes represented by a non-linear approximator such as a neural network


1. Start with initial parameter values
2. Take action according to an **explore/exploit policy** (should converge to greedy policy, i.e. GLIE)
3. Perform TD update for each parameter

$$\theta_i \leftarrow \theta_i + \alpha \left(R(s) + \gamma \max_{a'} \hat{Q}_\theta(s', a') - \hat{Q}_\theta(s, a) \right) \frac{\partial \hat{Q}_\theta(s, a)}{\partial \theta_i}$$

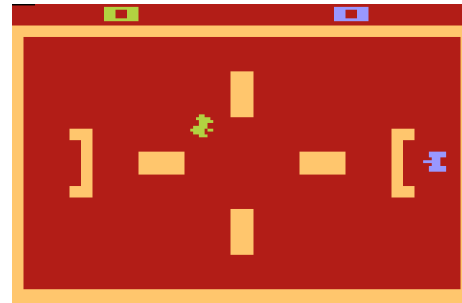
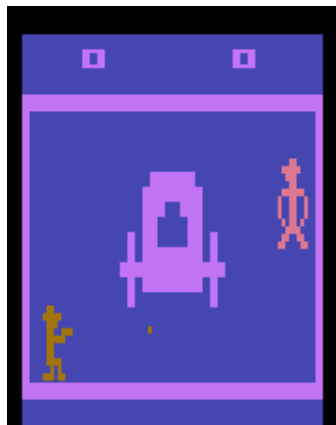
4. Goto 2

- Typically the space has many local minima and we no longer guarantee convergence
- Often works well in practice

calculate
closed-form



Deep Q-Network



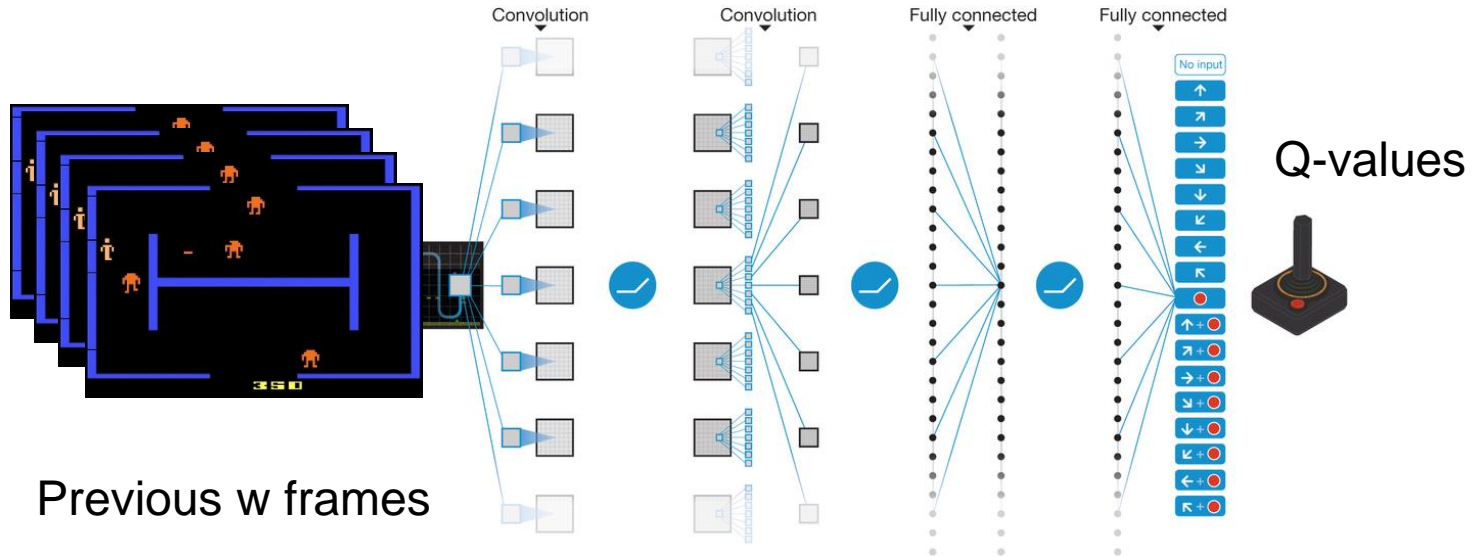
Human-level control through deep reinforcement learning
Nature 2015.

Deep Q-Network (DQN) = RL + DL

- Seeking a single agent which can solve any human-level task
 - RL defines the objective (Q-value function)
 - DL learns the hierarchical feature representations
- Use deep network to represent value function

Deep Q-Networks for Policies: Atari

- Network input = Observation history
 - Window of previous screen shots in Atari
- Network output = One output node per action (returns Q-value)



Stability Issues of DQN

- Naïve Q-learning oscillates or diverges with neural nets
- Data is sequential and successive samples are correlated
- Policy changes rapidly with slight changes to Q- values
 - Policy may oscillate
 - Distribution of data can swing from one extreme to another
- Scale of rewards and Q-values is unknown
 - Gradients can be unstable when back-propagated

Stable Solutions for DQN

DQN provides a stable solution to deep value-based RL

1. Experience replay
2. Freeze target Q-network
3. Clip rewards to sensible range

Stable Solution 1: Experience Replay

To remove correlations, build dataset from agent's experience

- Take action a_t
- Store transition (s_t, a_t, r_t, s_{t+1}) in replay memory \mathbf{D}
- Sample random mini-batch of transitions (s, a, r, s') from replay memory \mathbf{D}
- Optimize MSE between Q-network and Q-learning targets

$$L_i(\theta_i) = \mathbb{E}_{(s,a,r,s') \sim \mathbf{U}(\mathbf{D})} \left[\left(r + \gamma \max_{a'} Q(s', a'; \theta_i^-) - Q(s, a; \theta_i) \right)^2 \right]$$

Stable Solution 2: Fixed Target Q-Network

To avoid oscillations, fix parameters used in Q-learning target

- Compute Q-learning target w.r.t old, fixed parameters

$$r + \gamma \max_{a'} Q(s', a'; \theta_i^-)$$

- Optimize MSE between **Q-learning targets** and Q-network

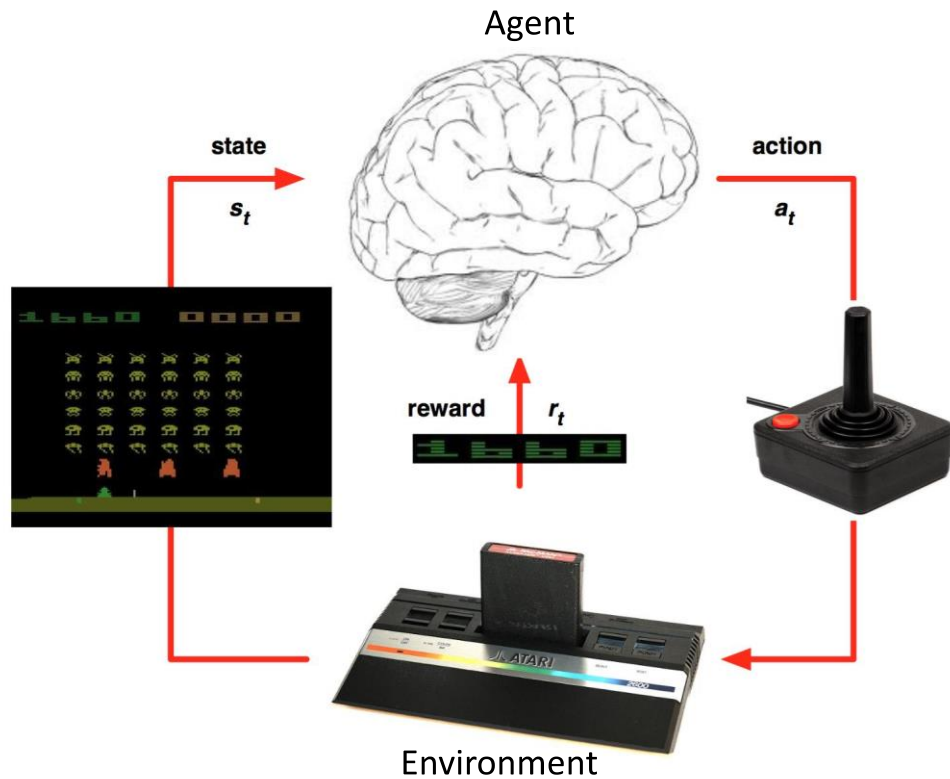
$$L_i(\theta_i) = \mathbb{E}_{(s,a,r,s') \sim U(D)} \left[\left(\boxed{r + \gamma \max_{a'} Q(s', a'; \theta_i^-)} - Q(s, a; \theta_i) \right)^2 \right]$$

- Periodically update fixed parameters

Stable Solution 3: Reward/Value Range

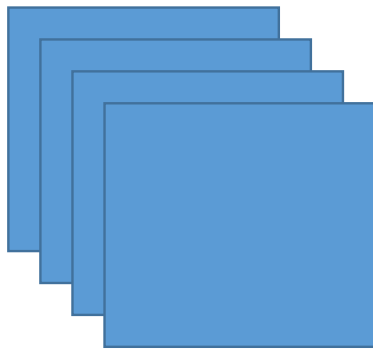
- To limit impact of any one update, control the reward / value range
- DQN clips the rewards to $[-1, +1]$
 - Prevents too large Q-values
 - Ensures gradients are well-conditioned

DQN in Atari



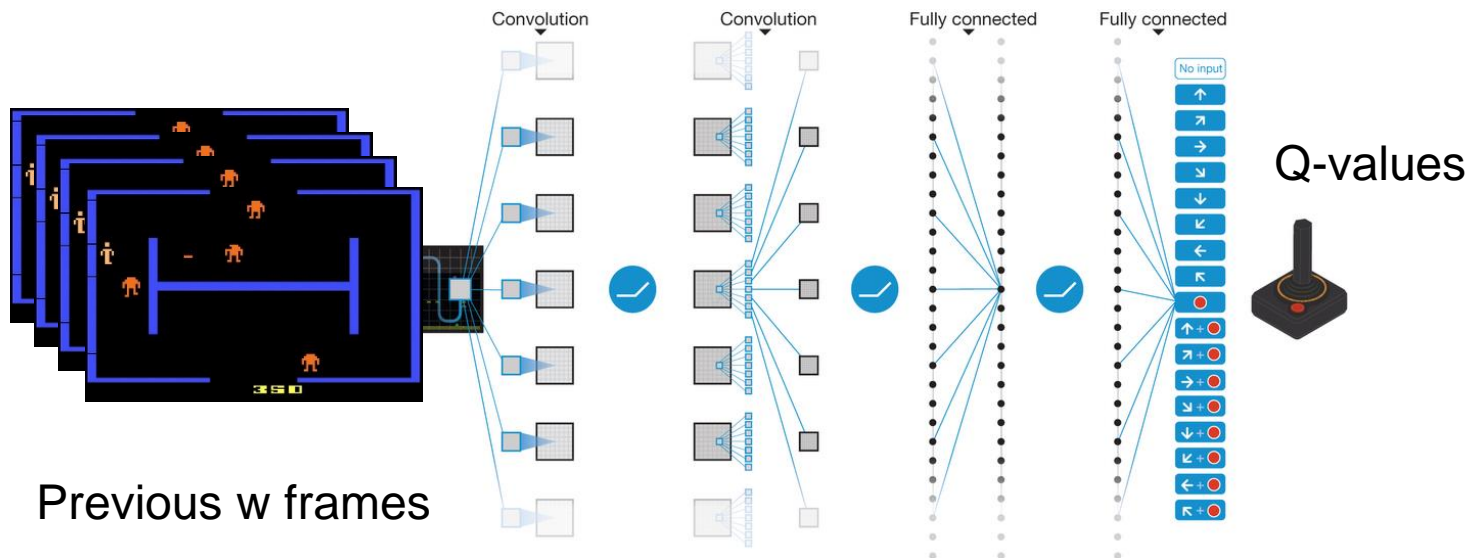
Preprocessing

- Raw images: 210x160 pixel images with 128-color palette
- Rescaled images: 84 x 84
- Input: 84 x 84 x 4 (4 most recent frames)



Deep Q-Networks Architecture: Atari

- Network input = Observation history
 - Window of previous screen shots in Atari
- Network output = One output node per action (returns Q-value)



DQN Algorithm

Initialize replay memory D to capacity N

Initialize action-value function Q with random weights θ

Initialize target action-value function \hat{Q} with weights $\theta^- = \theta$

For episode = 1, M **do**

Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$

For $t = 1, T$ **do**

With probability ε select a random action a_t

otherwise select $a_t = \operatorname{argmax}_a Q(\phi(s_t), a; \theta)$

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 D

Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from D

Set $y_j = \begin{cases} r_j & \text{if episode terminates at step } j+1 \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^-) & \text{otherwise} \end{cases}$

Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ with respect to the network parameters θ

Every C steps reset $\hat{Q} = Q$

End For

End For

DQN source code:

sites.google.com/a/deepmind.com/dqn/

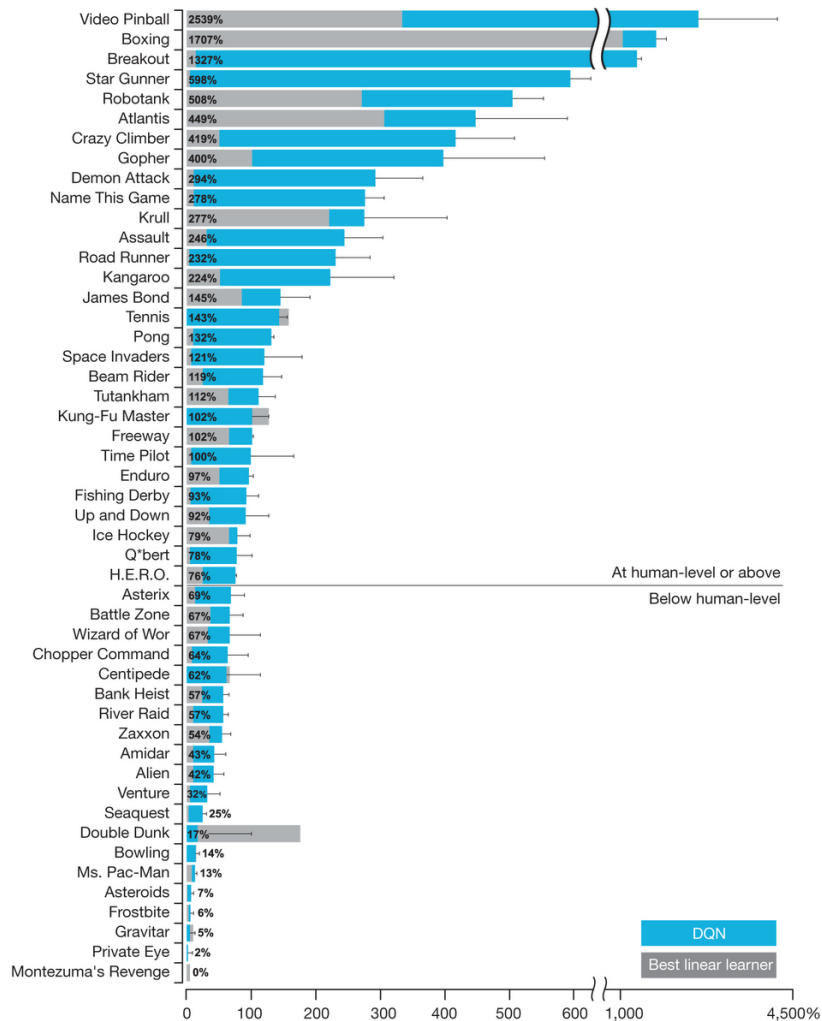
Training Details

- 49 Atari 2600 games
- Use RMSProp algorithms with minibatches 32
- Use 50 million frames (38 days)
- Replay memory contains 1 million recent frames
- Agent select actions on every 4th frames

Evaluation

- Agent plays each games 30 times for 5 min with random initial conditions
- Human plays the games in the same scenarios
- Random agent play in the same scenarios to obtain base-line performance

DQN Performance in Atari



How effective are the stable solutions?

Game	With replay, with target Q	With replay, without target Q	Without replay, with target Q	Without replay, without target Q
Breakout	316.8	240.7	10.2	3.2
Enduro	1006.3	831.4	141.9	29.1
River Raid	7446.6	4102.8	2867.7	1453.0
Seaquest	2894.4	822.6	1003.0	275.8
Space Invaders	1088.9	826.3	373.2	302.0

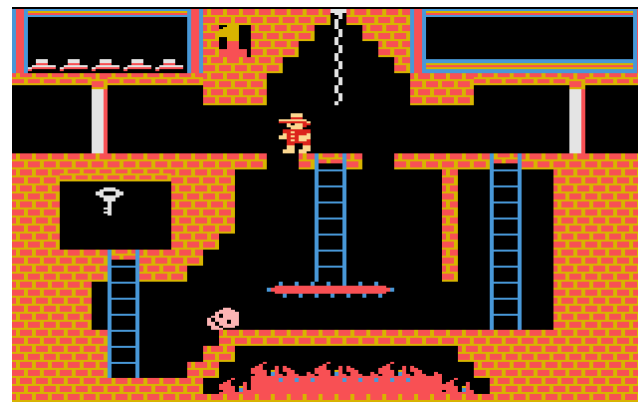
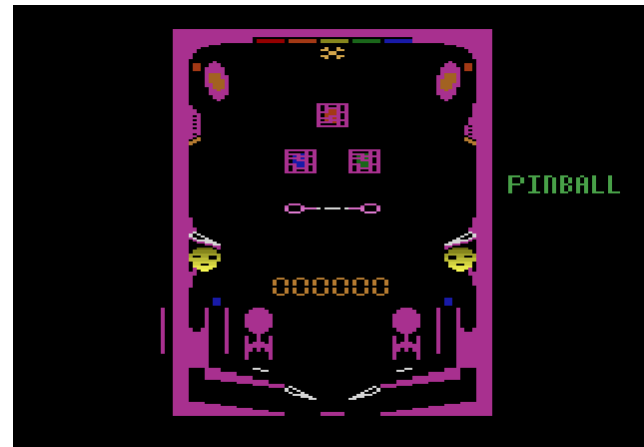
Strengths and weakness

- Strengths:

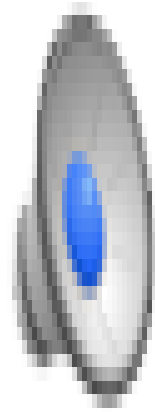
- Quick-moving, short-horizon games
- Pinball (2539%)

- Weakness:

- Long-horizon games that do not converge
- Walk-around games
- Montezuma's revenge



DeepMind Atari (©Two Minute Lectures) approximate Q-learning with neural nets



DQN Summary

- Deep Q-network agent can learn successful policies directly from **high-dimensional** input using end-to-end reinforcement learning
- The algorithm achieve a level surpassing professional human games tester across **49 games**

Extensions of DQN

- Double Q-learning for fighting maximization bias
- Prioritized experience replay
- Dueling Q networks
- Multistep returns
- Value distribution
- Stochastic nets for explorations instead of ϵ -greedy

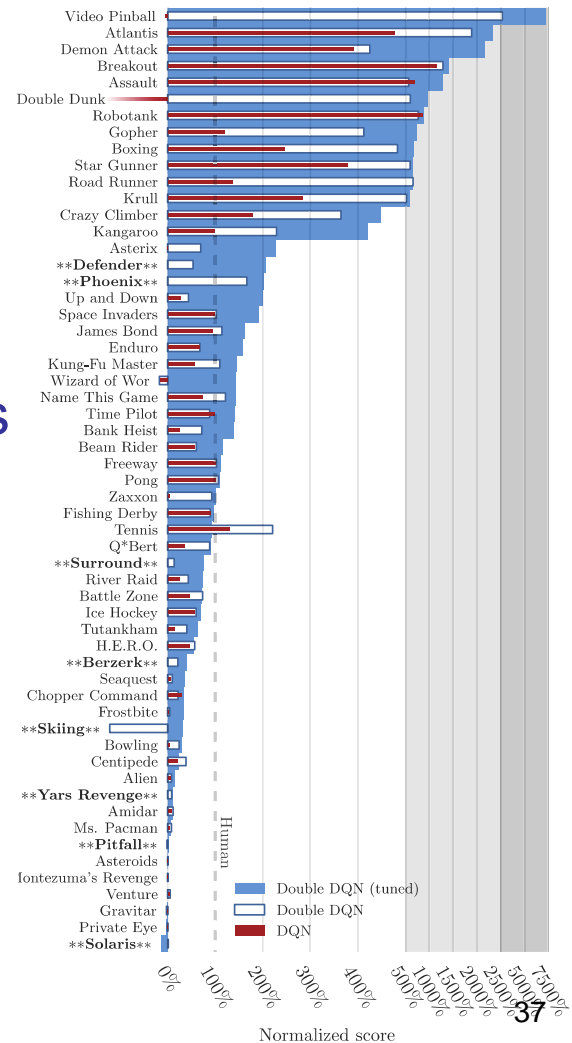
Double DQN

- Dealing with maximization bias of Q-Learning
- Current Q-network w is used to select actions
- Older Q-network w^- is used to evaluate actions

Action evaluation: w^-

$$I = \left(r + \gamma \underbrace{Q(s', \underbrace{\arg\max_{a'} Q(s', a', w)}, w^-)}_{a'} - Q(s, a, w) \right)^2$$

Action selection: w



Prioritized Experience Replay

- Weight experience according to “surprise” (or error)
- Store experience in priority queue according to DQN error

$$\left| r + \gamma \max_{a'} Q(s', a', \mathbf{w}^-) - Q(s, a, \mathbf{w}) \right|$$

- Stochastic Prioritization

$$P(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha}$$

p_i is proportional to
DQN error

- α determines how much prioritization is used, with $\alpha = 0$ corresponding to the uniform case.

Dueling Networks

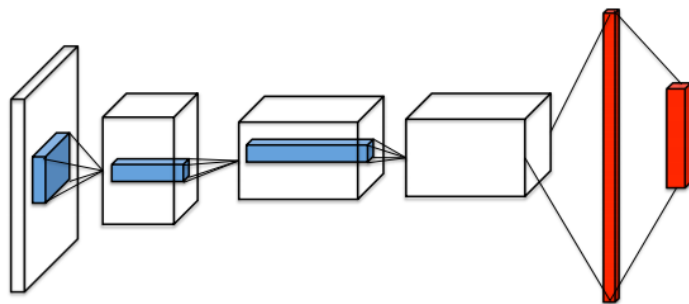
- Split Q-network into two channels
- Action-independent value function $V(s; \mathbf{w})$
- Action-dependent advantage function $A(s, a; \mathbf{w})$

$$Q(s, a; \mathbf{w}) = V(s; \mathbf{w}) + A(s, a; \mathbf{w})$$

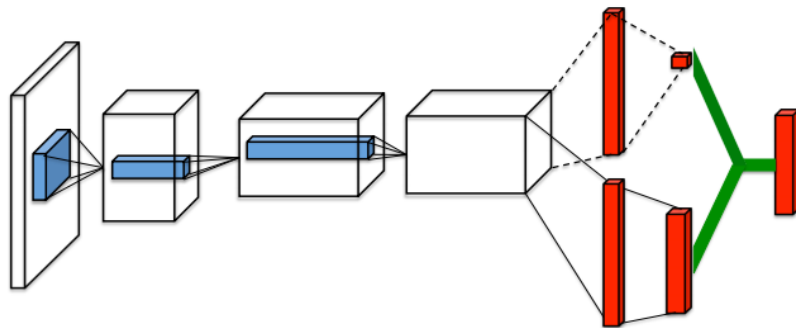
- Advantage function is defined as:

$$A^\pi(s, a) = Q^\pi(s, a) - V^\pi(s)$$

Dueling Networks vs DQN



DQN



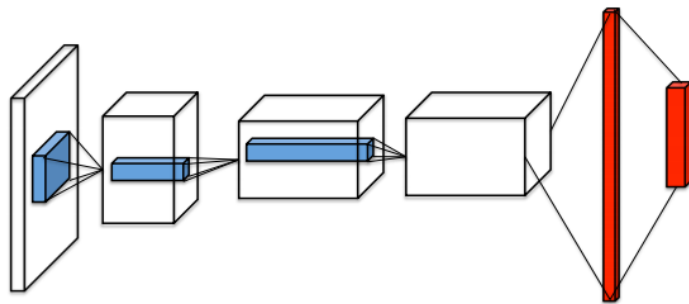
Dueling Networks

$$Q(s, a; \mathbf{w}) = V(s; \mathbf{w}) + A(s, a; \mathbf{w})$$

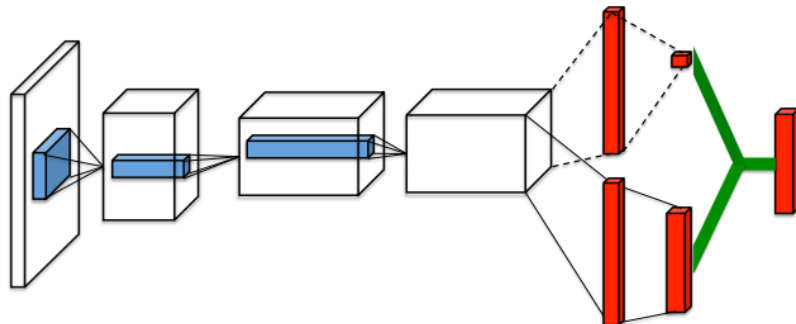
Unidentifiability : given Q, I cannot recover V, A

Wang et.al., ICML, 2016

Dueling Networks vs DQN



DQN



Dueling Networks

$$Q(s, a; \mathbf{w}) = V(s; \mathbf{w}) + \left(A(s, a; \mathbf{w}) - \frac{1}{|\mathcal{A}|} \sum_{a'} A(s, a'; \mathbf{w}) \right)$$

Wang et.al., ICML, 2016

Dueling Networks

- The value stream learns to pay attention to the road
- The advantage stream: pay attention only when there are cars immediately in front, so as to avoid collisions

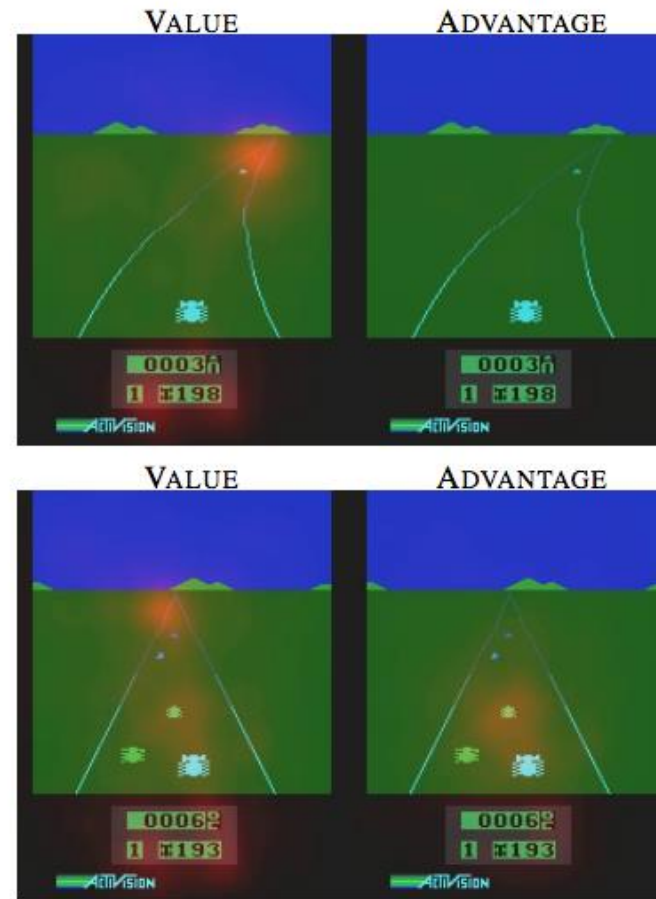
Deep Inside Convolutional Networks: Visualising Image Classification Models and Saliency Maps

Karen Simonyan

Andrea Vedaldi

Andrew Zisserman

Visual Geometry Group, University of Oxford
{karen, vedaldi, az}@robots.ox.ac.uk



Multi-step Returns

- Truncated n-step return from a state s_t :
$$R_t^{(n)} = \sum_{k=0}^{n-1} \gamma_t^{(k)} R_{t+k+1}$$

- Multistep Q-learning update rule:

$$I = \left(R_t^{(n)} + \gamma_t^{(n)} \max_{a'} Q(S_{t+n}, a', \mathbf{w}) - Q(s, a, \mathbf{w}) \right)^2$$

- Singlestep Q-learning update rule:

$$I = \left(r + \gamma \max_a Q(s', a', \mathbf{w}) - Q(s, a, \mathbf{w}) \right)^2$$

Rainbow: Combining Improvements in Deep Reinforcement Learning

Matteo Hessel
DeepMind

Joseph Modayil
DeepMind

Hado van Hasselt
DeepMind

Tom Schaul
DeepMind

Georg Ostrovski
DeepMind

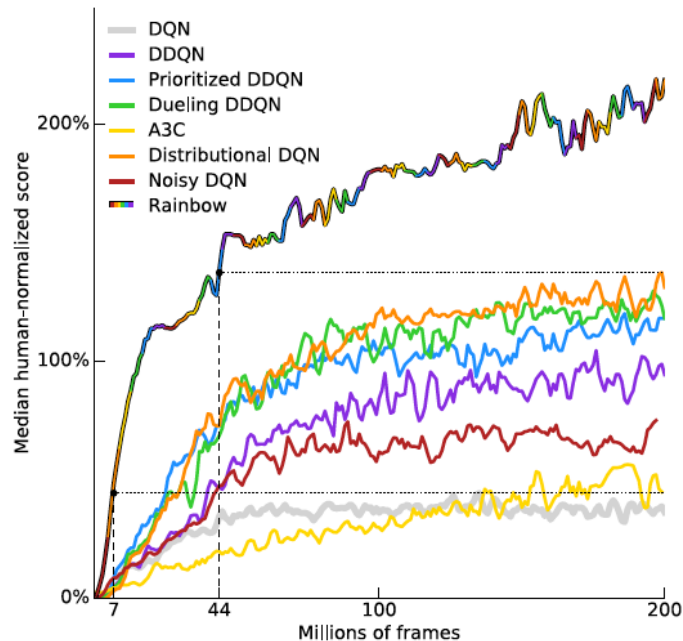
Will Dabney
DeepMind

Dan Horgan
DeepMind

Bilal Piot
DeepMind

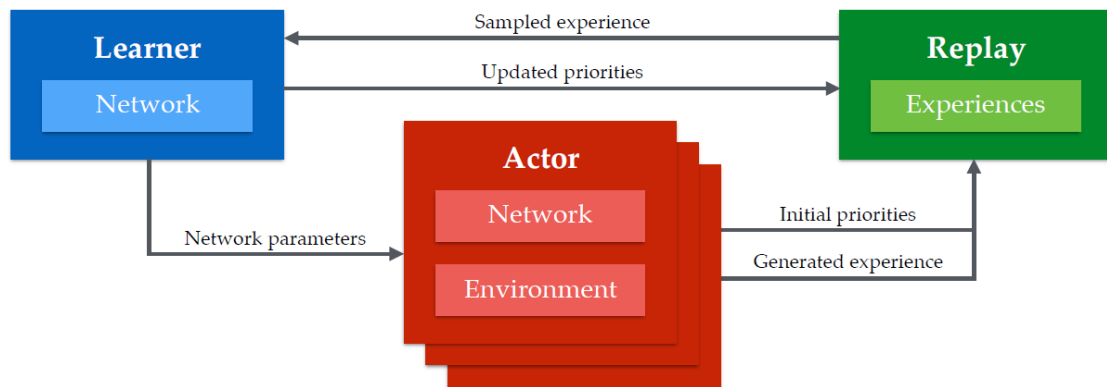
Mohammad Azar
DeepMind

David Silver
DeepMind



Distributed DQN

- Separating Learning from Acting
- Distributing hundreds of actors over CPUs



- Advantages: better harnessing computation, local priority evaluation, better exploration

Distributed DQN with Recurrent Experience Replay (R2D2)

- Providing an LSTM layer after the convolutional stack
 - To deal with partial observability
- Other tricks:
 - prioritized distributed replay
 - n-step double Q-learning (with $n = 5$)
 - generating experience by a large number of actors (typically 256)
 - learning from batches of replayed experience by a single learner

State-of-the-Art DQNs

