

# Deep Q-Networks

**Volodymyr Mnih**

**Deep RL Bootcamp, Berkeley**



**DeepMind**

# Recap: Q-Learning

- Learning a parametric Q function:  $Q_{\theta}(s, a)$ 
  - Remember:  $\text{target}(s') = R(s, a, s') + \gamma \max_{a'} Q_{\theta_k}(s', a')$
  - Update:  $\theta_{k+1} \leftarrow \theta_k - \alpha \nabla_{\theta} \mathbb{E}_{s' \sim P(s'|s,a)} [(Q_{\theta}(s, a) - \text{target}(s'))^2] \big|_{\theta=\theta_k}$
  - For tabular function,  $\theta \in \mathbb{R}^{|S| \times |A|}$ , we recover the familiar update:
$$Q_{k+1}(s, a) \leftarrow (1 - \alpha)Q_k(s, a) + \alpha [\text{target}(s')]$$
    - Converges to optimal values (\*)
- Does it work with a neural network Q functions?
  - Yes but with some care

# Recap: (Tabular) Q-Learning

Algorithm:

Start with  $Q_0(s, a)$  for all  $s, a$ .

Get initial state  $s$

For  $k = 1, 2, \dots$  till convergence

    Sample action  $a$ , get next state  $s'$

    If  $s'$  is terminal:

$$\text{target} = R(s, a, s')$$

        Sample new initial state  $s'$

    else:

$$\text{target} = R(s, a, s') + \gamma \max_{a'} Q_k(s', a')$$

$$Q_{k+1}(s, a) \leftarrow (1 - \alpha)Q_k(s, a) + \alpha [\text{target}]$$

$$s \leftarrow s'$$

# Recap: Approximate Q-Learning

Algorithm:

Start with  $Q_0(s, a)$  for all  $s, a$ .

Get initial state  $s$

For  $k = 1, 2, \dots$  till convergence

Sample action  $a$ , get next state  $s'$

If  $s'$  is terminal:

$$\text{target} = R(s, a, s')$$

Sample new initial state  $s'$

else:

$$\text{target} = R(s, a, s') + \gamma \max_{a'} Q_k(s', a')$$

$$\theta_{k+1} \leftarrow \theta_k - \alpha \nabla_{\theta} \mathbb{E}_{s' \sim P(s'|s,a)} [(Q_{\theta}(s, a) - \text{target}(s'))^2] \big|_{\theta=\theta_k}$$

$$s \leftarrow s'$$

Chasing a nonstationary target!

Updates are correlated within a trajectory!

# DQN

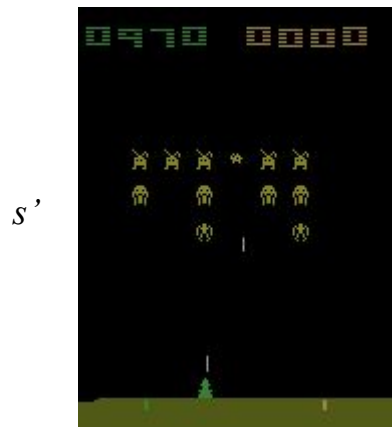
- High-level idea - **make Q-learning look like supervised learning**.
- Two main ideas for stabilizing Q-learning.
- Apply Q-updates on batches of past experience instead of online:
  - **Experience replay** (Lin, 1993).
  - Previously used for better data efficiency.
  - Makes the data distribution more stationary.
- Use an older set of weights to compute the targets (**target network**):
  - Keeps the target function from changing too quickly.

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

# Target Network Intuition

- Changing the value of one action will change the value of other actions and similar states.
- The network can end up chasing its own tail because of bootstrapping.
- Somewhat surprising fact - bigger networks are less prone to this because they alias less.

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



# DQN Training Algorithm

## Algorithm 1: deep Q-learning with experience replay.

Initialize replay memory  $D$  to capacity  $N$

Initialize action-value function  $Q$  with random weights  $\theta$

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  $\varepsilon$  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  $D$

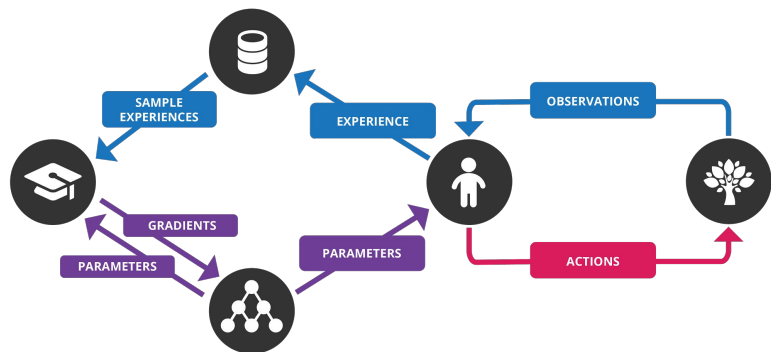
Set  $y_j = \begin{cases} r_j & \text{if episode terminates at step } j+1 \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^-) & \text{otherwise} \end{cases}$

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

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

**End For**

**End For**

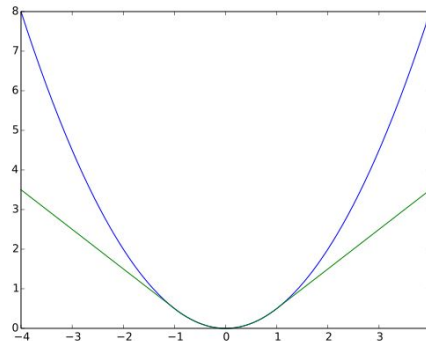


# DQN Details

- Uses Huber loss instead of squared loss on Bellman error:

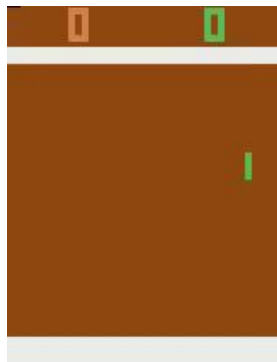
$$L_{\delta}(a) = \begin{cases} \frac{1}{2}a^2 & \text{for } |a| \leq \delta, \\ \delta(|a| - \frac{1}{2}\delta), & \text{otherwise.} \end{cases}$$

- Uses RMSProp instead of vanilla SGD.
  - Optimization in RL really matters.
- It helps to anneal the exploration rate.
  - Start  $\varepsilon$  at 1 and anneal it to 0.1 or 0.05 over the first million frames.





# DQN on ATARI



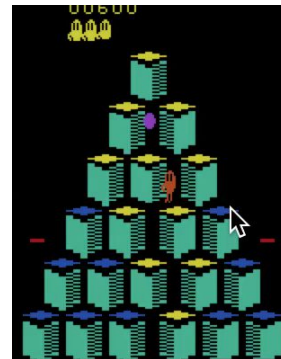
Pong



Enduro



Beamrider

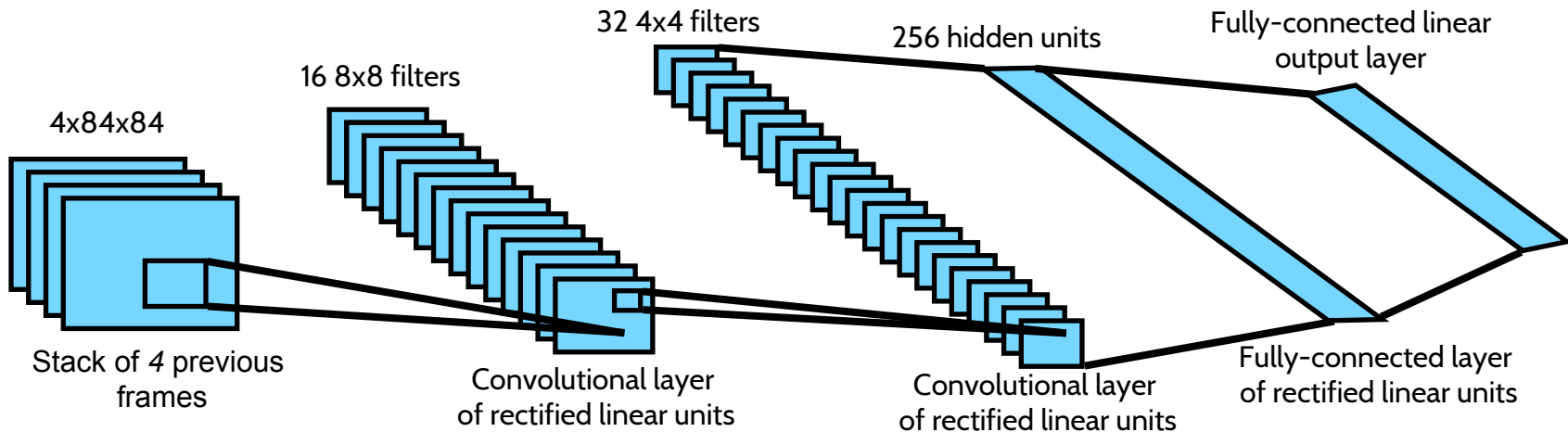


Q\*bert

- 49 ATARI 2600 games.
- From pixels to actions.
- The change in score is the reward.
- Same algorithm.
- Same function approximator, w/ 3M free parameters.
- Same hyperparameters.
- Roughly human-level performance on 29 out of 49 games.

# ATARI Network Architecture

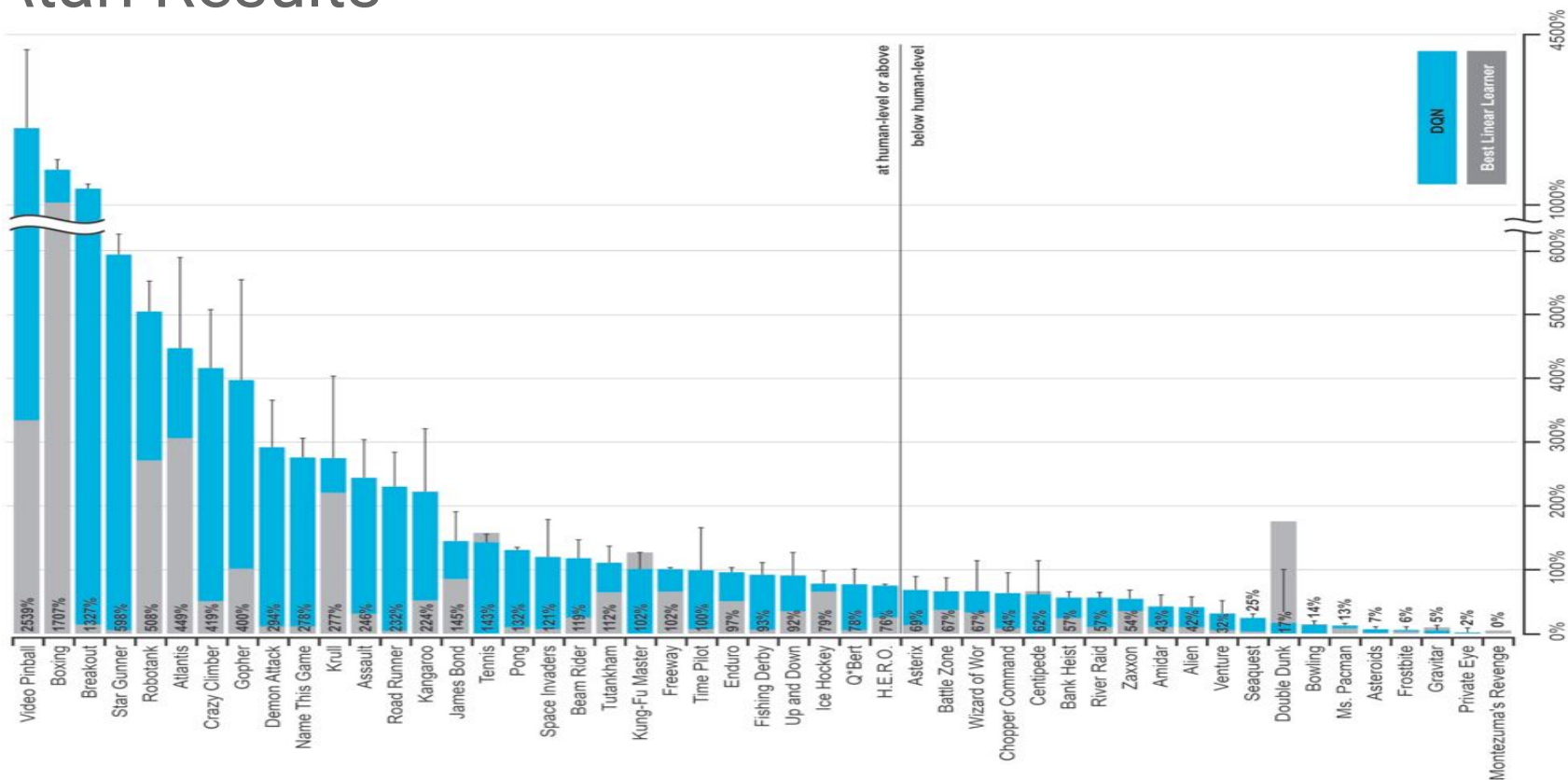
- Convolutional neural network architecture:
  - History of frames as input.
  - One output per action - expected reward for that action  $Q(s, a)$ .
  - Final results used a slightly bigger network (3 convolutional + 1 fully-connected hidden layers).



# Stability Techniques

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

# Atari Results



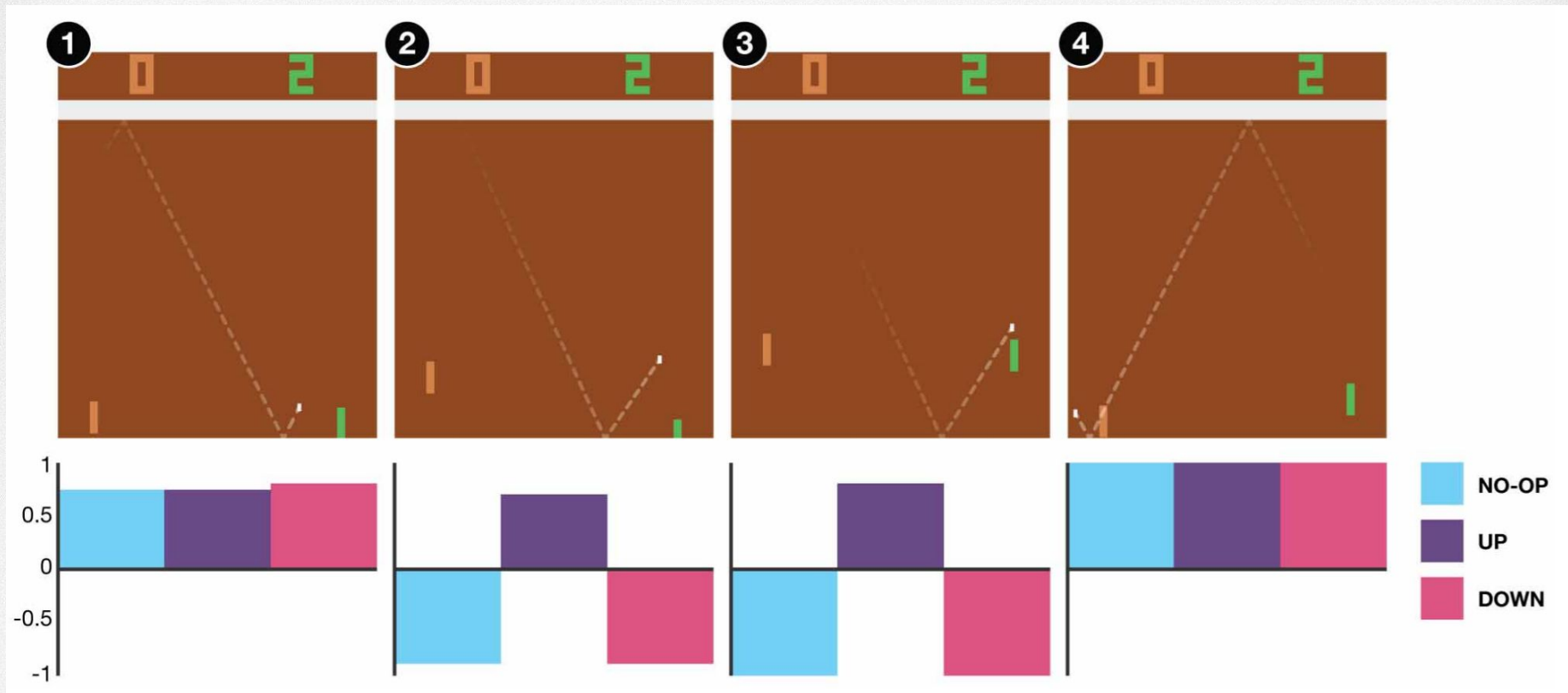
"Human-Level Control Through Deep Reinforcement Learning", Mnih, Kavukcuoglu, Silver et al. (2015)

# DQN Playing ATARI



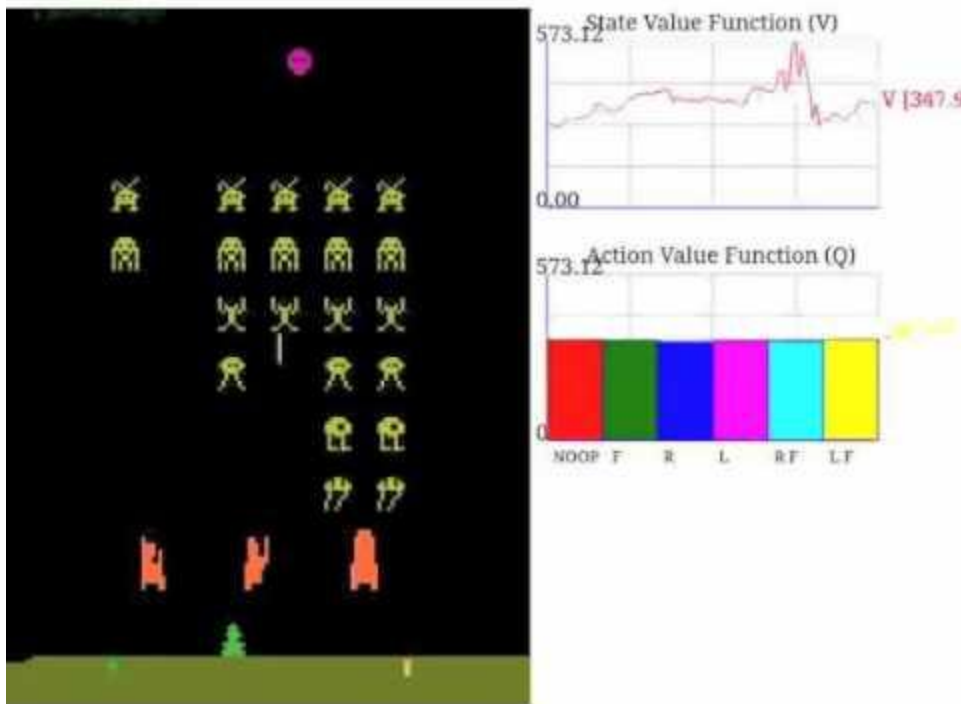


# Action Values on Pong



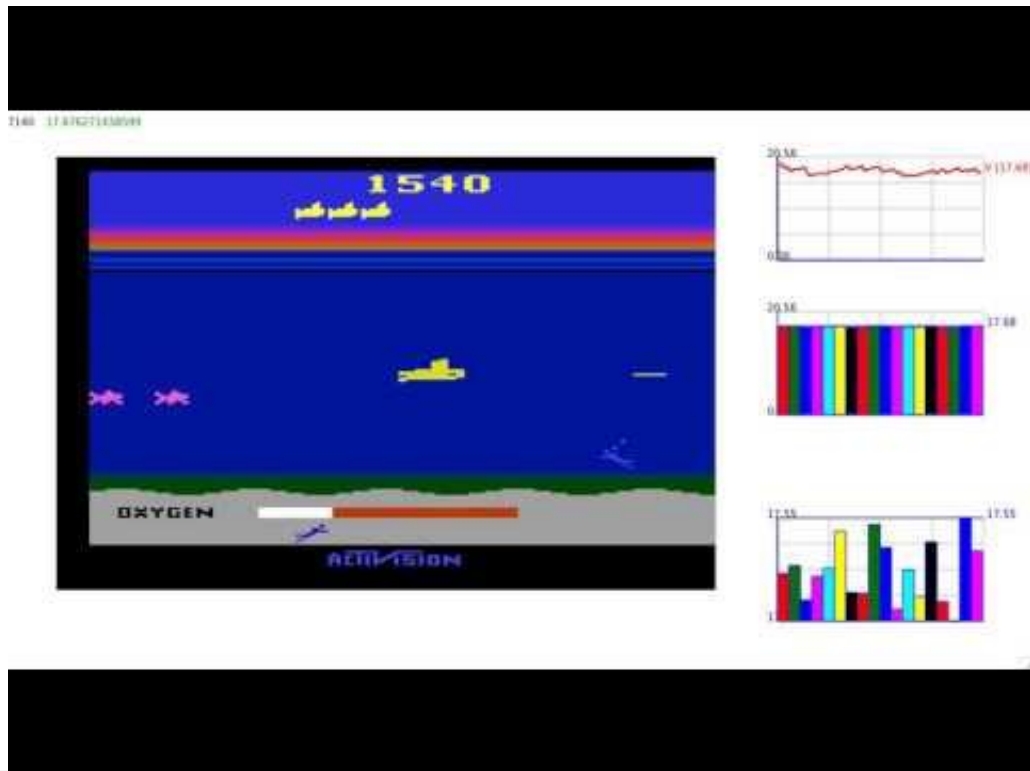


# Learned Value Functions





# Sacrificing Immediate Rewards





# DQN Source Code

- The DQN source code (in Lua+Torch) is available:

<https://sites.google.com/a/deepmind.com/dqn/>



# Neural Fitted Q Iteration

- NFQ (Riedmiller, 2005) trains neural networks with Q-learning.
- Alternates between collecting new data and fitting a new Q-function to all previous experience with batch gradient descent.

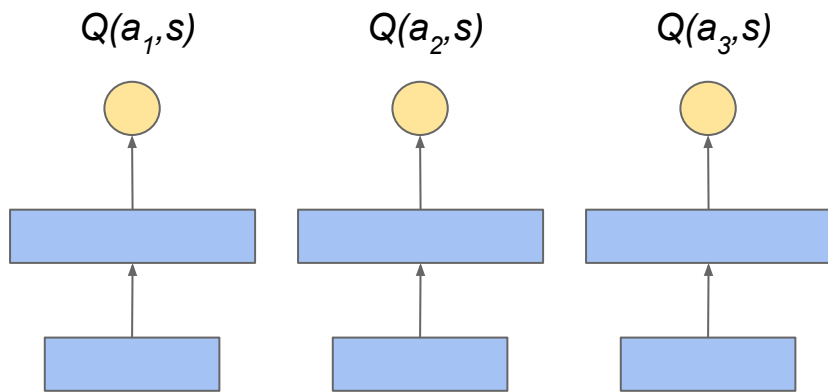
```
NFQ_main() {  
  input: a set of transition samples  $D$ ; output: Q-value function  $Q_N$   
  k=0  
  init_MLP()  $\rightarrow Q_0$ ;  
  Do {  
    generate_pattern_set  $P = \{(input^l, target^l), l = 1, \dots, \#D\}$  where:  
       $input^l = s^l, u^l$ ,  
       $target^l = c(s^l, u^l, s'^l) + \gamma \min_b Q_k(s'^l, b)$   
    Rprop_training( $P$ )  $\rightarrow Q_{k+1}$   
    k:= k+1  
  } WHILE ( $k < N$ )
```

- DQN can be seen as an online variant of NFQ.

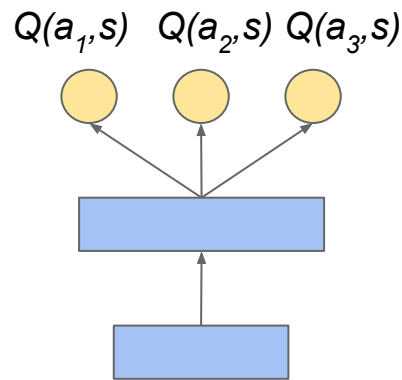
# Lin's Networks

- Long-Ji Lin's thesis "Reinforcement Learning for Robots using Neural Networks" (1993) also trained neural nets with Q-learning.
- Introduced experience replay among other things.
- Lin's networks **did not share parameters among actions**.

**Lin's architecture**



**DQN**

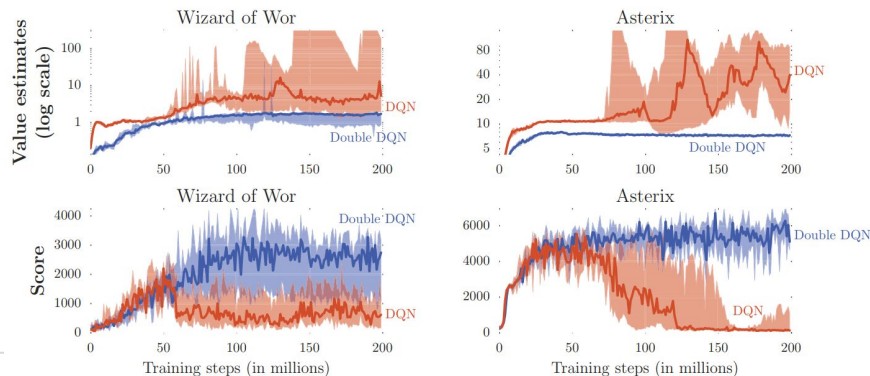


# Double DQN

- There is an upward bias in  $\max_a Q(s, a; \theta)$ .
- DQN maintains two sets of weight  $\theta$  and  $\theta^-$ , so reduce bias by using:
  - $\theta$  for selecting the best action.
  - $\theta^-$  for evaluating the best action.
- Double DQN loss:

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

	no ops		human starts		
	DQN	DDQN	DQN	DDQN	DDQN (tuned)
Median	93%	<b>115%</b>	47%	88%	<b>117%</b>
Mean	241%	<b>330%</b>	122%	273%	<b>475%</b>



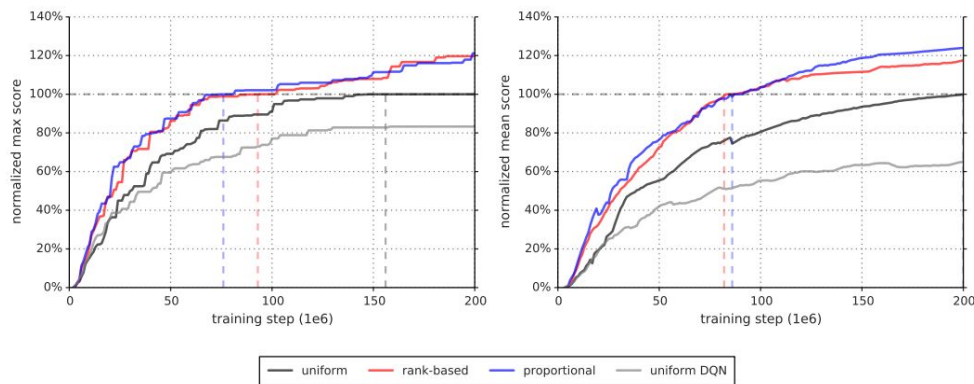
# Prioritized Experience Replay

- Replaying all transitions with equal probability is highly suboptimal.
- Replay transitions in proportion to absolute Bellman error:

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

- Leads to much faster learning.

	DQN		Double DQN (tuned)		
	baseline	rank-based	baseline	rank-based	proportional
<b>Median</b>	48%	106%	111%	113%	128%
<b>Mean</b>	122%	355%	418%	454%	551%
<b>&gt; baseline</b>	–	41	–	38	42
<b>&gt; human</b>	15	25	30	33	33
<b># games</b>	49	49	57	57	57



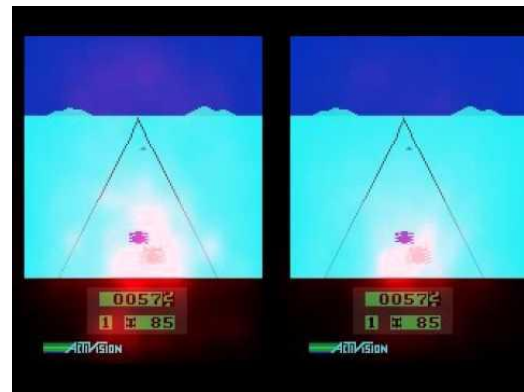
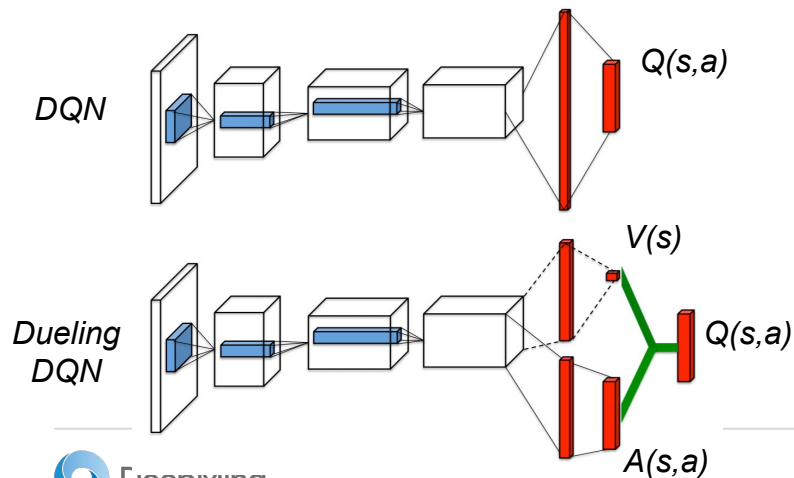
# Dueling DQN

- Value-Advantage decomposition of Q:

$$Q^{\pi}(s, a) = V^{\pi}(s) + A^{\pi}(s, a)$$

- Dueling DQN (Wang et al., 2015):

$$Q(s, a) = V(s) + A(s, a) - \frac{1}{|\mathcal{A}|} \sum_{a=1}^{|\mathcal{A}|} A(s, a)$$



Atari Results

	30 no-ops		Human Starts	
	Mean	Median	Mean	Median
Prior. Duel Clip	<b>591.9%</b>	<b>172.1%</b>	<b>567.0%</b>	<b>115.3%</b>
Prior. Single	434.6%	123.7%	386.7%	112.9%
Duel Clip	<b>373.1%</b>	<b>151.5%</b>	<b>343.8%</b>	<b>117.1%</b>
Single Clip	341.2%	132.6%	302.8%	114.1%
Single	307.3%	117.8%	332.9%	110.9%
Nature DQN	227.9%	79.1%	219.6%	68.5%

"Dueling Network Architectures for Deep Reinforcement Learning", Wang et al. (2016)

# Noisy Nets for Exploration

- Add noise to **network parameters** for better exploration [Fortunato, Azar, Piot et al. (2017)].
- Standard linear layer:  $y = wx + b$
- Noisy linear layer:  $y \stackrel{\text{def}}{=} (\mu^w + \sigma^w \odot \varepsilon^w)x + \mu^b + \sigma^b \odot \varepsilon^b$
- $\varepsilon^w$  and  $\varepsilon^b$  contain noise.
- $\sigma^w$  and  $\sigma^b$  are learned parameters that determine the amount of noise.

	Baseline		NoisyNet	
	Mean	Median	Mean	Median
DQN	213	47	<b>1210</b>	<b>89</b>
A3C	418	93	<b>1112</b>	<b>121</b>
Dueling	<b>2102</b>	126	1908	<b>154</b>

