

RL: studying sequential decision-making

→ sequential decision making

AI: → learning

→ deep computational graph

RL: Study sequential decision-making
(general framework)

The Deep RL Boom

TD-Gammon (Tesauro, 1989-1995)



Slot car driving (Lange & Riedmiller, 2012)



Arcade Learning Environment (Bellemare 2013)



Deep Q-Network (2013, 2015)



Trust region policy optimization (Schulman, 2015)



End-to-end training (Levine 2015)

On real robots



AlphaGo (2015)

RL + DNN ?

- ① Sparse / delayed feedback
- ② data distribution is non-stationary
 - determined by the agent's actions
 - Most of the DNN theory no longer apply
 - Exploration vs exploitation
 - Local minima
- ③ Training DNN with RL was thought to be inherently unstable.

(Tsitsikis & Van Roy 1997)

DQN (Deep Q Networks)

CNN ^{represent} → Action-value function(Q)

S_t, a_t, r_t, S_{t+1}
experience

Tabular Q-learning:

- ① Start with a guess for each $Q(s, a)$
- ② interact with environment using policy based on Q
- ③ updates (based on Bellman Equations)

$$Q(S_t, a_t) \leftarrow Q(S_t, a_t) + \lambda_t$$

$$\left(\underbrace{Y_t + \gamma \max_a Q(S_{t+1}, a)}_{\text{target}} - Q(S_t, a_t) \right)$$

Problems: (DNN/CNN → Q)

- ① Correlation between successive updates
- ② Correlation between $Q(S_t, a_t)$ and the target

High-level idea:

Q-learning
(RL)

look
like

supervised
learning

Apply Q-update on batches of past experience
instead of online

- ① Experience replay (Lin, 1993)
- ② Previously used for better data efficiency
- ③ Makes the data distribution more stationary

Use an old set of weights to compute the
target (target network)

keeps the target function from changing too quickly

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

Neural Fitted Q Iteration

NFQ (Riedmiller, 2005)

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.

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

s

s'



because they can use action repeats but these are two successive

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 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

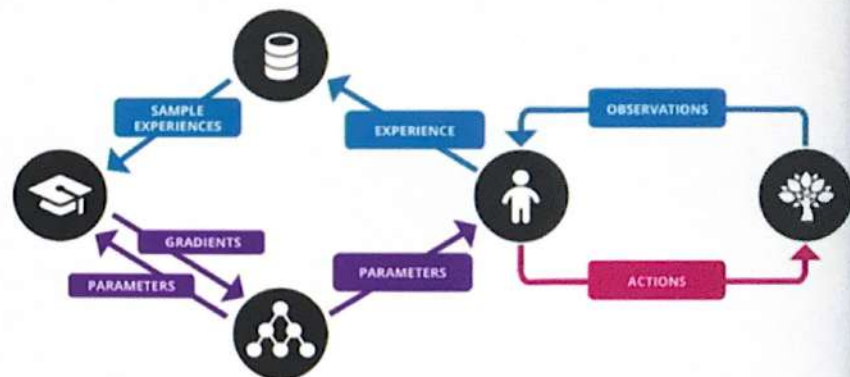
Set $y_j = \begin{cases} r_j & \text{if episode terminates at step } j+1 \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^-) & \text{otherwise} \end{cases}$

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



Neural Fitted Q iteration

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:  
       $input^l = s^l, u^l$ ,  
       $target^l = c(s^l, u^l, s'') + \gamma \min_b Q_k(s'', b)$   
    Rprop_training( $P$ )  $\rightarrow Q_{k+1}$   
    k:= k+1  
  } WHILE ( $k < N$ )
```

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

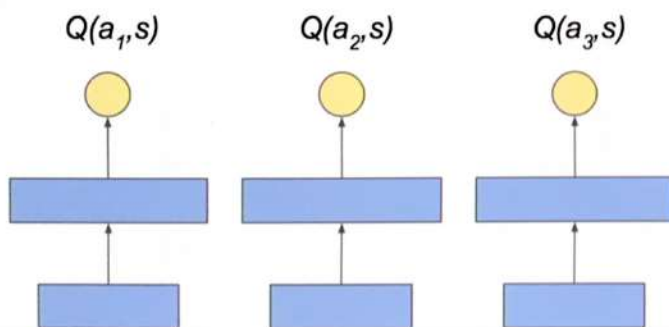


similar method to dqn was neural fitted
cue iteration so

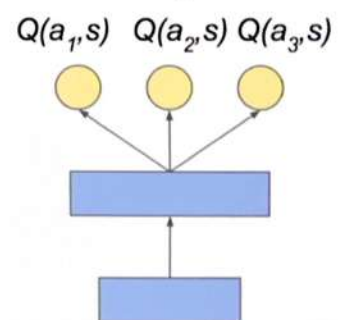
Lin's Networks

- 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



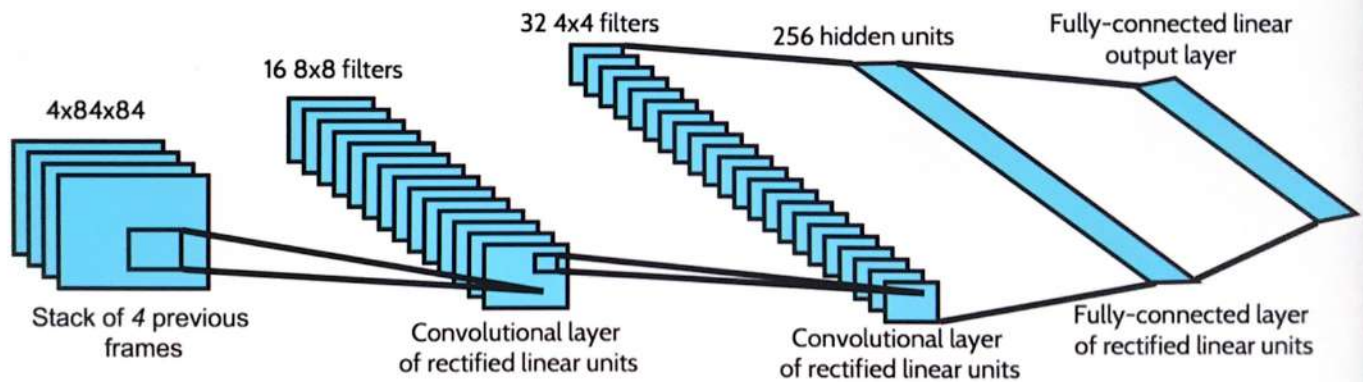
DQN



ATARI Network Architecture

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).



one output per action so not only by
today's standards

DQN / Mini-batch Q-learning

Pros

- Robust
- GPU friendly

Cons:

- Slow
- less RNN friendly (less successful in 3D environment)

Beyond DQN

- ① fast training
- ② on or off-policy methods
- ③ flexibility
 - discrete or continuous actions
 - feedforward
 - recurrent models
- ④ Asynchronous Methods for DRL

(Nknh 2016)

AsyncRL

AsyncRL

Parallel actor-learners (CPU threads)

online asynchronous updates

(Recht 2011, Lian 2015)

RL algorithm: $\left\{ \begin{array}{l} \text{on-policy / off-policy} \\ \text{value-based / policy-based} \end{array} \right.$

1-step Q-learning

Parallel actor-learners compute online 1-step update.

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

$$\Delta \theta \leftarrow \Delta \theta + \frac{\partial (y - Q(s, a; \theta))^2}{\partial \theta}$$

N-step Q-learning

Q-learning with a uniform mixture of backups of length 1 through N.

$r_t, r_{t+1}, r_{t+2}, \dots, r_{t+N}$ $\max_a Q(a, S_{t+N+1})$

$$y \leftarrow \sum_{k=0}^{N-1} \gamma^k r_{t+k} + \gamma^N \max_{a'} Q(S_{t+N}, a'; \theta^-)$$

$$\Delta \theta \leftarrow \Delta \theta + \frac{\partial (y - Q(S_t, a_t; \theta))^2}{\partial \theta}$$

Variation of "Incremental multi-step Q-learning"

(Peng & Williams 1995)

Asynchronous Advantage Actor-critic (A3C)

① Agent learns $\begin{cases} \text{a policy} \\ \text{a state value function} \end{cases}$

② bootstrapped n-step returns

→ reduce variance

over REINFORCE with a baseline

③ Policy gradient multiplied by an estimate of the advantage

$$\nabla_{\theta} \log \pi(a_t | s_t, \theta) \left(\sum_{k=0}^N \gamma^k r_{t+k} + \gamma^{N+1} V(s_{t+N+1}) - V(s_t) \right)$$

④ Critic/value function is trained with n-step

TD learning (by minimizing the MSE)

$$\sum_{k=0}^N \gamma^k r_{t+k} + \gamma^{N+1} V(s_{t+N+1}; \theta^-) - V(s_t; \theta)^2$$

"A3C tends to dominate the value-based methods"

A3C - ATARI Results

Method	Training Time	Mean	Median
DQN	8 days on GPU	121.9%	47.5%
Gorilla	4 days, 100 machines	215.2%	71.3%
D-DQN	8 days on GPU	332.9%	110.9%
Dueling D-DQN	8 days on GPU	343.8%	117.1%
Prioritized DQN	8 days on GPU	463.6%	127.6%
A3C, FF	1 day on CPU	344.1%	68.2%
A3C, FF	4 days on CPU	496.8%	116.6%
A3C, LSTM	4 days on CPU	623.0%	112.6%



Atari and if you train for four days
then you

MacBook Pro

Pros of N-step methods:

- ① faster reward propagation
- ② No need for target network
- ③ easier training of RNN

A3C — Pros (fast, RNN friendly)
(stable, scalable)
Cons (Not GPU friendly)

IMPALA: distributed deep RL, (2018)

Scale up A3C

"V-Trace" algorithm

"DM Lab-30 Task Set"
(Multi-)

DQN is more stable than A3C

off-policy harms actor-critic method



Deep RL

- ① Mini-batch training on GPUs
- ② Deep ResNet and LSTMs
- ③ Adam / RMSProp optimizers

Deep RL

Practical Advice - Getting Started

- Start with a simple problem.
 - Something solvable in under a minute on your local machine.
 - Make it similar to the problem you really want to solve.
 - Ideally it should have knobs for controlling its difficulty.
- Plot the training curves (averaged over multiple episodes).
- Visualize the policy.
- Visualize the value function.
- Visualize everything you can think of.



has some of the properties of the real
problem you want to

Practical Advice - Neural Nets

- Doing early experiments with a small network can help iterate faster.
 - This can also backfire (DQN and target networks).
- Reasonable strategy:
 - Run a few progressively larger nets to find what's sufficient for experimenting.
 - Periodically try larger nets to max out performance and verify assumptions.
- Be careful with initialization:
 - Visualize the initial policy to make sure it gets some rewards.
- Try RMSProp and/or Adam.
- Test deep learning tricks before incorporating them: dropout, batch norm, etc.
- See John Schulman's excellent guide - <http://joschu.net/docs/nuts-and-bolts.pdf>