Deep Reinforcement Learning

Eduardo Morales, Hugo Jair Escalante

INAOE

Outline

Function Approximation

- So far, we have assumed an explicit representation of the value function in the form of a table, which works well in small spaces, but is unfeasible in domains like Chess (10^{120}) or in continuous spaces, like in robotics
- An alternative is to use an implicit representation, i.e., a function
- For instance, in games an estimated utility function can be represented by a weighted linear function over a set of attributes (f_i's):

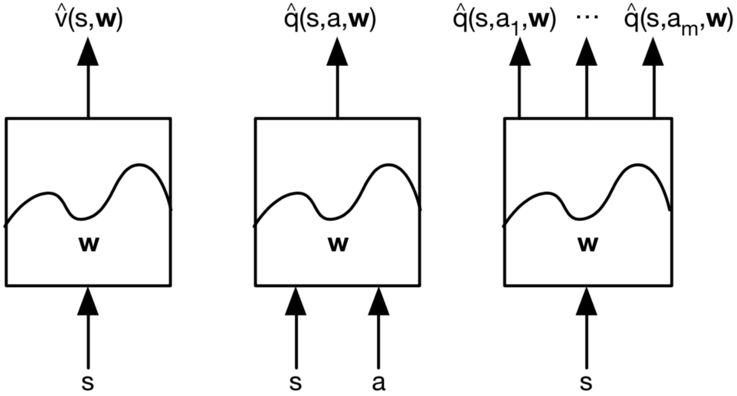
$$V(i) = w_1 f_1(i) + w_2 f_2(i) + \ldots + w_n f_n(i)$$

- In Chess there are approximate 10 weights which is clearly a significant compression

Learning Functions

- The compression obtained with an implicit representation allows the learning system to generalize over states which were not visited
- There is a large number of options that can be used, in RL, researchers have used NN, SVM, decision trees, Gaussian processes, etc.
- As in any learning system, there is a balance between the hypotheses space and the reasoning process
- RL setting poses some challenges to traditional supervised learning: Non stationary, delayed rewards, bootstrapping, on-line learning, non independent samples

Alternatives for Value functions


$\hat{v}(s,\mathbf{w})$
$\mathbf{w}$
s
$\hat{q}(s,a,\mathbf{w})$
$\mathbf{w}$
s
a
$\hat{q}(s,a_1,\mathbf{w})$
$\cdots$
$\hat{q}(s,a_m,\mathbf{w})$
$\mathbf{w}$
s

Learning Functions

- With large state-action spaces, we want to estimate a policy or value function which is close to the real function, e.g., $Q_\theta(s, a) \sim Q(s, a)$ with parameters θ (or $V_\theta(s) \sim V(s)$ or $\pi_\theta(a|s) \sim \pi(a|s)$)
- The objective is to find the parameters θ to minimize the loss between the estimated $Q_\theta(s, a)$ and the real $Q(s, a)$
- Generally the loss function is the mean square error: $J(\theta) = \mathbb{E}[(Q(s, a) - Q_\theta(s, a))^2]$.

Learning Functions

- We can update θ in the direction of the gradient to find a local minimum: $\theta \leftarrow \theta - \frac{1}{2}\alpha\nabla_\theta J(\theta)$.
- In the case where we are approximating the Q function ($Q_\theta(s,a)$), θ can be updated as:

$$\theta \leftarrow \theta + \alpha(Q(s,a) - Q_\theta(s,a))\nabla Q_\theta(s,a)$$

- Again, we do not know $Q(s,a)$ and we have to approximate it
- This is a *semi-gradient* because we are calculating the error with an approximation of the real function, and not with the real function.

A Basic Algorithm (one-step TD function approximation algorithm)

```
Initialize the parameters ($\theta$) of the value function arbitrarily
**repeat** {for each episode}
    Initialize $s$
    **repeat** {for each step in the episode}
        Select an action $a$ in $s$ using a policy derived from $Q$ (e.g., $\epsilon$–greedy)
        Take action $a$, observe $r, s'$
        $\theta \leftarrow \theta + \alpha[r + \gamma Q_\theta(s', a') - Q_\theta(s, a)]\nabla Q_\theta(s, a)$
        $s \leftarrow s'$
    **until** $s$ is terminal
**until** convergence
```

A Basic Algorithm

Again, $Q(s, a)$ can take different forms to evaluate the *TD-error*, for instance:

- One-step SARSA: $r + \gamma Q_\theta(s', a')$
- One-step Q-Learning: $r + \gamma max_{a'} Q_\theta(s', a')$
- Monte Carlo: G_t
- n-step RL: $G_{t:t+n}$ where $G_{t:t+n} = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} \ldots \gamma^n Q_\theta(s_{t+n}, a_{t+n})$

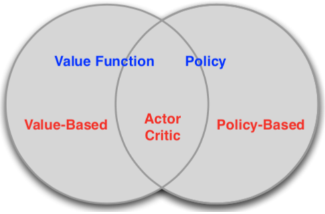
We can plug-in any of these forms into the previous learning algorithm

Learning Functions

- When learning functions we have to be careful since we do not have the real function and the error is evaluated with the function that we are learning!!
- The data distribution changes as we are learning
- Subsequent examples are correlated which breaks the assumption of independent samples
- For *off-policy* algorithms convergence is not always possible

Options to Learn Functions

- Value Functions (V o Q)
- Policy (π)
- Actor-Critic: Both



Actor-Critic Algorithms

- Actor ($\pi(a|s;\Theta)$): Controls how the agent behaves
- Critic ($Q(s,a;\Theta)$): Measures how good are the actions
- Actor-Critic: Runs in parallel, updating the policy and the value function

Deep RL

- Learning directly from high-dimensional data (e.g., images, videos, etc.) has been one the big challenges for RL
- Normally, the user needs to define a suitable representation for the RL algorithm to work
- Recent developments in Deep Learning (DL) have shown that raw data can be directly used as input for learning

Deep RL

However, there are several challenges:

- From a DL point of view, a large number of labelled data is required
- From an RL point of view, we have sparse and delayed rewards, and noisy information
- DL assumes the the data is independent and indentically distributed (i.i.d.) which is not the case for RL
- In RL the data distribution changes during learning, which is challenging for DL which assumes a fix data distribution

DQN

- The breakthrough came with the DQN algorithm which managed to successfully combined Q-learning with Deep Convolutional Networks
- It was originally applied to learn how to play Atari games
- It should be noted that the same architecture (although with different learned parameters) was used to learn all the games
- It achieved expert human level performance in 29 out of 46 games

DQN

It involved two main techniques to mitigate some of the existing problems:

1. *Experience replay*, which stores the experiences of the agent at each step ($e_t = (s_t, a_t, r_t, s_{t+1})$) in a database $\mathcal{D} = e_1, \ldots, e_N$
   Updates to the Q-function are done by sampling $\mathcal{D}$
2. Uses two networks for learning, one with fixed weights that is used as reference to the other network which is updating its parameters
   After a fixed number of steps the networks are interchanged

Experience Replay

- Advantages:
  1. Each step can be used in several updates
  2. Learning from subsequent samples is inefficient due to strong correlations in the samples
  3. Taking the average over several data samples helps to smooth the learning process and prevents from oscillations and divergences in the parameters
- Disadvantages:
  1. It stores the last N samples, and do not distinguishes between relevant transitions
  2. Requires a large storage capacity

Copy of the Value Function

- Another form to reduce variance was to have two networks, one which has fixed weights and serves as reference to the other network which is updated during the learning process
- After M steps the most recent updated network is used as the fixed network and the learning process continues
- We still have a moving target, but now it stays fixed for some time which improves convergence

Copy of the Value Function

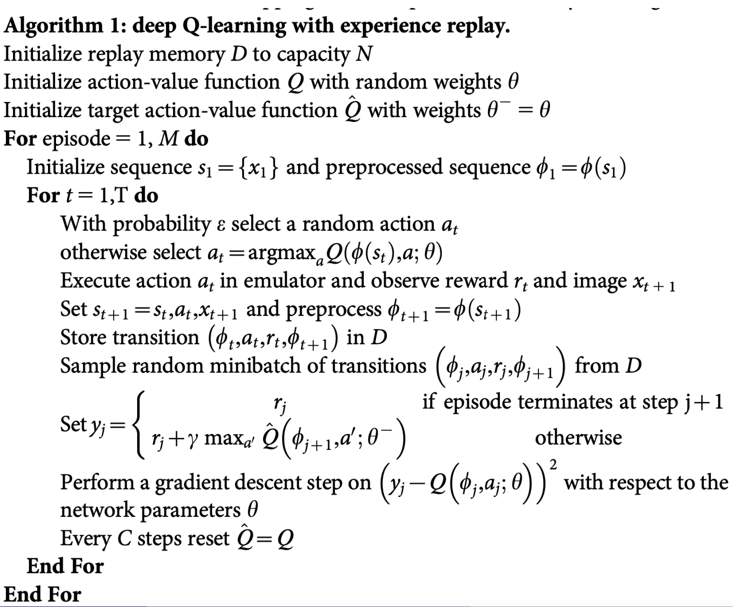
- The gradient of the loss function with respect to its weights is:

$$\nabla_{\theta_i} L(\theta_i) = \mathbb{E}_{s,a,r,s'}$$

$$\left[\left(r + \gamma max_{a'} Q(s', a'; \theta_i^-) - Q(s, a; \theta_i)\right) \nabla_{\theta_i} Q(s, a : \theta_i)\right]$$

- Where θ_i^- refers to the network that estimates Q with fixed weights and θ_i refers to the network that is been updated, and $\mathbb{E}_{s,a,r,s'}$ means that the updates are done using the average of these values taken from the samples of the experience replay
- Every certain number of steps the networks are interchanged

DQN Algorithm

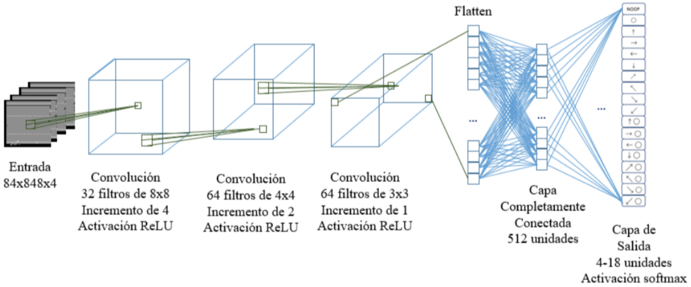


```
**Algorithm 1: deep Q-learning with experience replay.**
Initialize replay memory $D$ to capacity $N$
Initialize action-value function $Q$ with random weights $\theta$
Initialize target action-value function $\hat{Q}$ with weights $\theta^- = \theta$
**For** episode $= 1, M$ **do**
   Initialize 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 $a_t = \mathrm{argmax}_a Q(\phi(s_t), a; \theta)$
      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 $\left(y_j - Q(\phi_j, a_j; \theta)\right)^2$ with respect to the
      network parameters $\theta$
      Every $C$ steps reset $\hat{Q} = Q$
   **End For**
**End For**
```

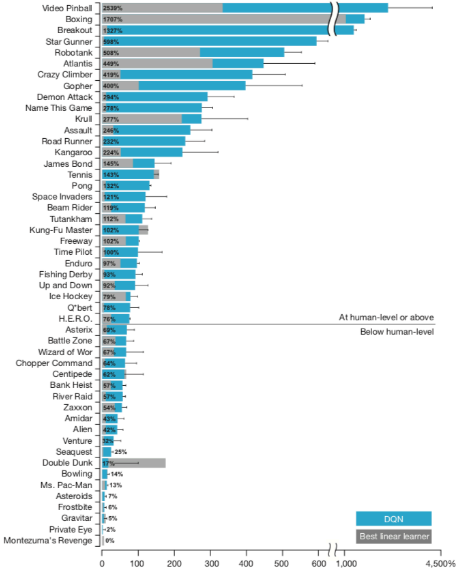
DQN

- It learned each game from 50 million game screens (roughly 38 days of experience per game)
- The screens were converted to black and white images and reduced to arrays of 84×84 pixels and were stacked with the last four frames (i.e., Input = $84 \times 84 \times 4$)
- Network: Three convolutional layers: 32 (20×20), 64 (9×9), and 64 (7×7) feature maps
- The last layer is a dense layer of 512 units connected to up to 18 output units (one for each possible action)

DQN


Entrada
84x848x4
Convolución
32 filtros de 8x8
Incremento de 4
Activación ReLU
Convolución
64 filtros de 4x4
Incremento de 2
Activación ReLU
Convolución
64 filtros de 3x3
Incremento de 1
Activación ReLU
Flatten
Capa
Completamente
Conectada
512 unidades
NOOP
Capa de
Salida
4-18 unidades
Activación softmax

Results


Video Pinball 2539%
Boxing 1707%
Breakout 1327%
Star Gunner 598%
Robotank 508%
Atlantis 449%
Crazy Climber 419%
Gopher 400%
Demon Attack 294%
Name This Game 278%
Krull 277%
Assault 246%
Road Runner 232%
Kangaroo 224%
James Bond 145%
Tennis 143%
Pong 132%
Space Invaders 121%
Beam Rider 119%
Tutankham 112%
Kung-Fu Master 102%
Freeway 102%
Time Pilot 100%
Enduro 97%
Fishing Derby 93%
Up and Down 92%
Ice Hockey 79%
Q*bert 78%
H.E.R.O. 76%
Asterix 69%
Battle Zone 67%
Wizard of Wor 67%
Chopper Command 64%
Centipede 62%
Bank Heist 57%
River Raid 57%
Zaxxon 54%
Amidar 43%
Alien 42%
Venture 32%
Seaquest 25%
Double Dunk 17%
Bowling 14%
Ms. Pac-Man 13%
Asteroids 7%
Frostbite 6%
Gravitar 5%
Private Eye 2%
Montezuma's Revenge 0%
At human-level or above
Below human-level
DQN
Best linear learner
0
100
200
300
400
500
600
1,000
4,500%

Extensions to DQN

After the paper on DQN, several researchers proposed different improvements:

- *Prioritized experience replay*
- DDQN
- *Dueling network*
- *Multi-step learning*: Use eligibility traces in its *forward-view* form
- *Distributional RL*: Learn to estimate a distribution over rewards, instead of a single expected reward value
- *Noisy* DQN: Introduces noise that is gradually reduced to improve the exploration process
- ...

Prioritized Experience Replay

- Some experiences may be more relevant than others but could rarely occur
- The idea of *Prioritized Experience Replay* is to change the sampling distribution considering:

  $$p_t = |\delta_t| + e$$

  where $|\delta_t|$ is the size of the *TD error* and e is a constant to force that all the samples have a non-zero probability of being selected
- It also included a parameter to regulate the randomness over the sampling process and *importance sampling weights* to gradually change the sampling weight during training

DDQN

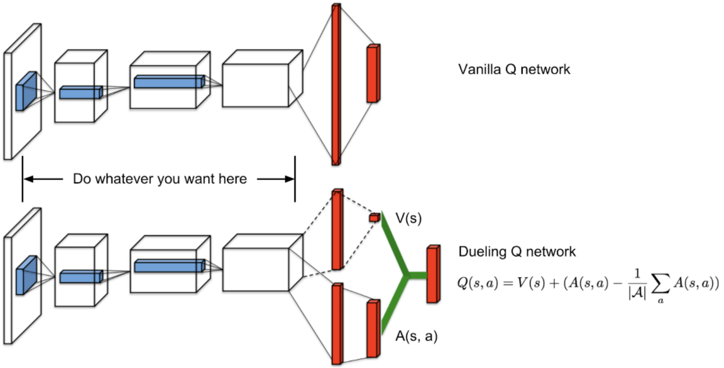
- The Q-learning algorithm can over-estimate the action values under certain conditions
- To deal with this, DDQN decomposes the updating of the Q-function in two steps
- The standard updating of the Q-learning function is:

$$\theta \leftarrow \theta + \alpha(Y^Q - Q_\theta(s,a))\nabla_\theta Q_\theta(s,a)$$

but now Y^Q $(r + \gamma \max_{a'} Q_{\theta^-}(s', a'))$ is changed to $Y^{DoubleQ}$:

$$Y^{DoubleQ} \leftarrow r + \gamma Q_{\theta^-}(s', argmax_a Q_\theta(s', a))$$

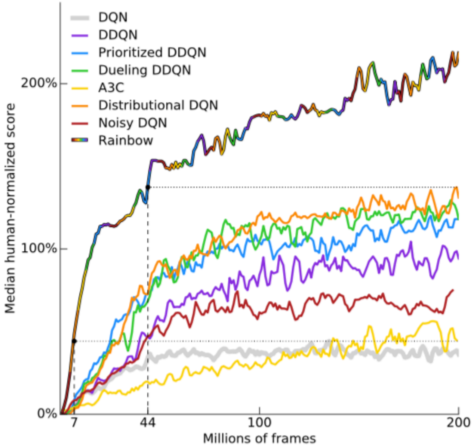
Dueling DQN



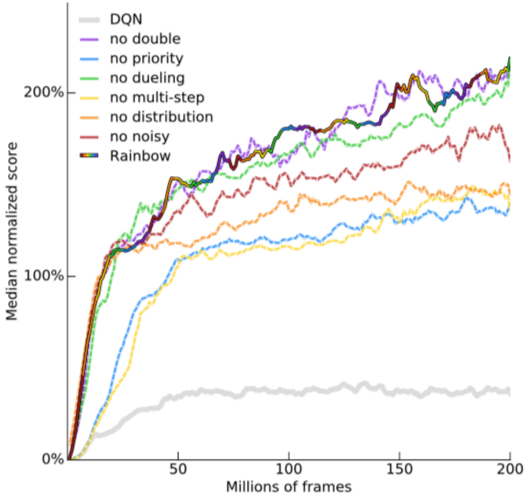
$$Q(s,a) = V(s) + (A(s,a) - \frac{1}{|\mathcal{A}|}\sum_{a} A(s,a))$$

RAINBOW

RAINBOW makes a combination of many of the previous approaches (with some adjustments) and shows that it improves the performance of all the individual approaches



Results


DQN
no double
no priority
no dueling
no multi-step
no distribution
no noisy
Rainbow
Median normalized score
0%
100%
200%
50
100
150
200
Millions of frames

Policy Learning

- Instead of trying to learn a value function (V or Q) we can try to directly learn the policy function (π) or both
- Learning a policy function may be easier than learning a value function, has better convergence properties and stochastic policies may be learned
- We would like to learn a policy that produces the optimal value function under that policy
- An important development is what is known as the *policy gradient theorem*

Policy Gradient

- For convenience, we will denote $V^{\pi_\theta}(s)$ as $V(\theta)$
- Let $\tau = (s_0, a_0, r_0, \ldots, s_{T-1}, a_{T-1}, r_{T-1}, s_T)$ denote a trajectory, where s_T is the terminal state
- So $G(\tau) = \sum_{t=0}^{T} r(s_t, a_t)$ and

$$\begin{aligned} V(\theta) &= \mathbb{E}_{\pi_\theta}\left[\sum_{t=0}^{T} r(s_t, a_t); \pi_\theta\right] \\ &= \sum_{\tau} P_\theta(\tau) G(\tau) \end{aligned}$$

where $P_\theta(\tau)$ denotes the probability over trajectories when executing policy π_θ and $G(\tau)$ is the return we obtain on that trajectory

Policy Gradient

- So our goal is to find the policy parameters θ such that:

$$argmax_\theta V(\theta) = argmax_\theta \sum_\tau P_\theta(\tau) G(\tau)$$

- The policy parameters only appear in the distributions of trajectories, so the gradient with respect to θ is:

$$\nabla_\theta V(\theta) = \nabla_\theta \sum_\tau P_\theta(\tau) G(\tau)$$

can be rewritten as (since $\nabla \log x = \frac{\nabla x}{x}$):

$$\begin{aligned} \nabla_\theta V(\theta) &= \textstyle\sum_\tau G(\tau) \nabla_\theta P_\theta(\tau) \\ &= \textstyle\sum_\tau G(\tau) \frac{P_\theta(\tau)}{P_\theta(\tau)} \nabla_\theta P_\theta(\tau) \\ &= \textstyle\sum_\tau G(\tau) P_\theta(\tau) \nabla_\theta \log P_\theta(\tau) \end{aligned}$$

Policy Gradient

- We are summing over the probabilities of all trajectories, which we can approximate by sampling some m trajectories and averaging uniformly:

$$\nabla_\theta V(\theta) \approx \hat{g} = \frac{1}{m} \sum_{i=1}^{m} G(\tau^{(i)}) \nabla_\theta \log P_\theta(\tau^{(i)})$$

- So we need to evaluate for trajectory i:

$$\nabla_\theta \log P_\theta(\tau^{(i)}) = \nabla_\theta \log \left(\mu(s_0) \prod_{j=0}^{T-1} p(s_{j+1} | s_j, a_j) \pi_\theta(a_j | s_j) \right)$$

where $\mu(s_0)$ is the probability of the initial state s_0 and $\pi_\theta(a|s)$ is the policy that decides which action to take at each state

Policy Gradient

- We can expand the previous expression as:

$$\begin{aligned}\nabla_\theta \log P_\theta(\tau^{(i)}) = \quad & \nabla_\theta \log \mu(s_0) + \\ & \textstyle\sum_{j=0}^{T-1} \nabla_\theta \log p(s_{j+1}|s_j, a_j) + \\ & \textstyle\sum_{j=0}^{T-1} \nabla_\theta \log \pi_\theta(a_j|s_j)\end{aligned}$$

- By taking the derivative, and since only the last term depends on θ:

$$\nabla_\theta V(\theta)) \approx \hat{g} = \frac{1}{m} \sum_{i=1}^{m} G(\tau^{(i)}) \sum_{j=0}^{T-1} \nabla_\theta \log \pi_\theta(a_j|s_j)$$

- Which means that we do not need to know the transition probability, although we still need to evaluate the gradient of the log of the policy

Policy Gradient

- In the last expression $\frac{1}{m}\sum_{i=1}^{m} G(\tau^{(i)})$ can be seen as a Monte Carlo estimate
- We have seen that:

$$\nabla_\theta V(\theta) = \nabla_\theta \mathbb{E}_\tau[G] = \mathbb{E}_\tau[\sum_{t=0}^{T-1} r(s_t, a_t) \sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t)]$$

Policy Gradient

- We can rearrange the summations:

$$\nabla_\theta V^{\pi_\theta} = \mathbb{E}_\tau[\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t) \sum_{i=t}^{T-1} r(s_t, a_t)]$$

- Where the last summation corresponds to the return G_t, which is the sum of the rewards from a particular state until a terminal state

REINFORCE

- REINFORCE is an algorithm that approximates a policy function using the policy gradient theorem [Williams, 1992]
- The algorithm follows a Monte Carlo approach (i.e., updates parameters after completing a full episode)
- Following the previous expression, with the discounted reward, the parameters of the policy function are updated with:

$$\theta_{t+1} = \theta_t + \alpha\gamma^t G_t \nabla_\theta \log \pi_\theta(a_t|s_t)$$

where:

  - γ^t is a discount factor multiplied by the number of times it reaches the state
  - G_t is the return (total accumulated reward) obtained from that state
  - a_t is the action selected by the policy

REINFORCE

- The policy gradient theorem can be generalize to include a comparison between the action value and a base value (*baseline*) which helps to reduce variance:
- With this, the updates are:

$$\theta_{t+1} = \theta_t + \alpha \left(G_t - b(s_t) \right) \nabla_\theta \log \pi_\theta(a_t|s_t)$$

- A natural candidate for $b(s)$ is the estimate of the value function $\hat{V}(s_t; \phi)$, where ϕ are the parameters of the value function

REINFORCE Algorithm

```
Initialize the weights of the policy ($\theta$) and of the value function ($\phi$)
**repeat**
    Generate an episode $s_0, a_0, r_1, s_1, \ldots, s_{T-1}, a_{T-1}, r_T$ following $\pi$
    **for** each step in the episode $t = 0, 1, \ldots, T-1$ **do**
        $G_t \leftarrow$ return since time $t$
        $\phi \leftarrow \phi + \beta(G_t - \hat{V}(s_t;\phi))\nabla_\phi \hat{V}(s_t;\phi)$
        $\theta \leftarrow \theta + \alpha\gamma^t(G_t - \hat{V}(s_t;\phi))\nabla_\theta \log \pi_\theta(a_t|s_t)$
    **end for**
**until** convergence
```

Actor-Critic

- Although the REINFORCE algorithm learns a policy and a value function, the value function serves as a baseline and not as a critic
- As any Monte Carlo method the learning process tends to be slow
- A temporal difference algorithm can be built by changing the step with the complete return that uses REINFORCE with a single step:

$$\theta_{t+1} = \theta_t + \alpha \left(G_t - \hat{V}(s_t; \phi) \right) \nabla_\theta \log \pi_\theta(a_t | s_t)$$

One-step Actor-Critic Algorithm

- G_t for a single step can be replaced by: $\hat{Q}(s,a;\phi)$ or $r_{t+1} + \gamma \hat{V}(s_{t+1};\phi)$
- In the first case, we end up with the *Advantage function*:

$$\hat{A}(s,a) = \hat{Q}(s,a) - \hat{V}(s)$$

  which tells us how much there is improvement over the average of that state (extra reward obtained with action *a*):

$$\begin{aligned}\theta_{t+1} &= \theta_t + \alpha(\hat{Q}(s_t,a_t) - \hat{V}(s_t))\nabla_\theta \log \pi_\theta(a_t|s_t) \\ &= \theta_t + \alpha \hat{A}(s_t,a_t)\nabla_\theta \log \pi_\theta(a_t|s_t)\end{aligned}$$

- In the second case, we end up with the TD-error, which can be seen as an unbiased estimate of the advantage function:
  $\theta_{t+1} = \theta_t + \alpha(r_t + \gamma \hat{V}(s_{t+1};\phi) - \hat{V}(s_t;\phi))\nabla_\theta \log \pi_\theta(a_t|s_t)$

One-step Actor-Critic Algorithm

```
Initialize the weights of the policy ($\theta$) and of the value function ($\phi$)
**repeat**
    Initialize $s$ (initial state of the episode)
    $I \leftarrow 1$
    **while** $s$ in not a terminal state **do**
        $a \sim \pi_\theta(\cdot|s)$
        Take action $a$, observe $s', r$
        $\delta \leftarrow r + \gamma \hat{V}(s'; \phi) - \hat{V}(s; \phi)$
        $\phi \leftarrow \phi + \alpha^\phi \delta \nabla_\phi \hat{V}(s; \phi)$
        $\theta \leftarrow \theta + \alpha^\theta I \delta \nabla_\theta \log \pi_\theta(a_t|s_t)$
        $I \leftarrow \gamma I$
        $s \leftarrow s'$
    **end while**
**until** Convergence
```

Extensions:Trust-Region Methods

- The policy gradient update can be described as:

$$\nabla_\theta J(\pi_\theta) = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t \nabla_\theta \log \pi_\theta(a_t|s_t) A_{\pi_\theta}(s_t, a_t)\right]$$

- We can use a “surrogate” objective function (TRPO):

$$J(\theta') = \mathbb{E}_{\pi_\theta}\left[\frac{\pi_{\theta'}(a|s)}{\pi_\theta(a|s)} A_{\pi_\theta}(s, a)\right]$$

Extensions:Trust-Region Methods

- One problem with importance sampling is that small differences between the distributions can become large values in the gradient
- We can use a bound that depends on the Kullback-Leibler distance between the two policies to induce $\pi_{\theta'} \sim \pi_\theta$:

$$J(\theta') = \mathbb{E}_{\pi_\theta}\left[\frac{\pi_{\theta'}(a|s)}{\pi_\theta(a|s)} A_{\pi_\theta}(s,a)\right]$$

$$\text{s.t. } \mathbb{E}_{\pi_\theta}\left[KL(\pi_\theta || \pi_{\theta'})\right] \leq \delta$$

Extensions:Trust-Region Methods

- As an alternative, we can constrain the values to be within certain bounds (PPO):

$$J(\theta') = \mathbb{E}_\theta \left[min(r_t(\theta')\hat{A}_t, clip(r_t(\theta'), 1-\epsilon, 1+\epsilon)\hat{A}_t) \right]$$

where $r_t(\theta') = \frac{\pi_{\theta'}(a_t|s_t)}{\pi_\theta(a_t|s_t)}$ and *clip* constrains the value of $r_t(\theta')$ to be within the $1-\epsilon, 1+\epsilon$ limits.

Extensions: Deterministic Policy

- Another possibility is to learn a deterministic policy ($\mu(s)$)
- As previously seen the Policy Gradient Theorem says that:

$$\nabla_\theta J(\pi_\theta) = \mathbb{E}\left[\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t) \sum_{i=t}^{T-1} r(s_t, a_t)\right]$$

- Where the last sum can be replaced by the action-value function Q, expressed in a simplified form as:

$$\nabla_\theta J(\pi_\theta) = \mathbb{E}\left[\nabla_\theta \log \pi_\theta(a|s) Q^\pi(s, a)\right]$$

Deterministic Policy Gradients

- To improve the policy, in general, a maximization greedy strategy is used: $\mu^{t+1}(s) = argmax_a Q^{\mu^t}(s, a)$
- This is problematic in continuous action spaces as we need to do a global optimization at each step
- Instead, we can move the parameters of the policy (θ) in the direction of the gradient of Q: $\nabla_\theta Q^{\mu^t}(s, \mu_\theta(s))$
- Each state may suggest a different direction, so we can average them by taking the expected value with respect to the state distribution $\rho^\mu(s)$:

$$\theta^{t+1} = \theta^t + \alpha \mathbb{E}_{s \sim \rho^{\mu^t}} \left[\nabla_\theta Q^{\mu^t}(s, \mu_\theta(s)) \right]$$

Deterministic Policy Gradients

- Applying the chain rule:

$$\theta^{t+1} = \theta^t + \alpha \mathbb{E}_{s \sim \rho^{\mu^t}} \left[\nabla_\theta \mu_\theta(s) \nabla_a Q^{\mu^t}(s,a)|_{a=\mu_\theta(s)} \right]$$

- The deterministic policy gradient theorem says that:

$$\nabla_\theta J(\mu_\theta) = \mathbb{E}_{s \sim \rho^{\mu}} \left[\nabla_\theta \mu_\theta(s) \nabla_a Q^{\mu}(s,a)|_{a=\mu_\theta(s)} \right]$$

  where $J(\mu_\theta) = \mathbb{E}[\sum_{t=1}^{\infty} \gamma^{t-1} r(s_t, a_t)]$
- This can be used in different actor-critic algorithms

Deterministic Policy Gradients

- For instance, using SARSA as critic, involves the following steps:

$$\delta_t = r_t + \gamma Q_\rho(s_{t+1}, a_{t+1}) - Q_\rho(s_t, a_t) \text{ (TD-error)}$$
$$\rho_{t+1} = \rho_t + \alpha_\rho \delta_t \nabla_\rho Q_\rho(s_t, a_t) \text{ (update } Q\text{)}$$
$$\theta^{t+1} = \theta^t + \alpha_\theta \nabla_\theta \mu_\theta(s_t) \nabla_a Q^\mu(s_t, a_t)|_{a=\mu_\theta(s)} \text{ (update } \mu\text{)}$$

- Using Q-learning as critic, involves replacing the TD-error by:

$$\delta_t = r_t + \gamma Q_\rho(s_{t+1}, \mu_\theta(s_{t+1})) - Q_\rho(s_t, a_t)$$

Deep Deterministic Policy Gradients

- DQN works with discrete actions, however, in many domains, it is more natural to have continuous actions
- Deep Deterministic Policy Gradients (DDPG), is a model-free RL algorithm for continuous actions that combines DPG with DQN
- It follows the same strategy as DQN (experience replay and a frozen target network) but in an actor-critic scheme
- It keeps four networks (2 for actor and 2 for critic):
    1. *Policy* (actor): $\pi : S \rightarrow A$
    2. *Action-value function approximator* (critic) $Q : S \times A \rightarrow R$

Deep Deterministic Policy Gradients

- Episodes are generated following a behavior policy, which is a noisy version of the objective function: $\pi_b(s) = \pi(s) + \mathbb{N}(0, 1)$
- The critic is trained with DQN but the objectives (y_t) are evaluated using the actions generated by the actor, i.e.: $y_t = r_t + \gamma Q(s_{t+1}, \pi(s_{t+1}))$
- The actor is trained with a mini-batch gradient descend with a deterministic policy function

DDPG Algorithm

```
Randomly initialize critic $Q_\theta(s, a)$, actor $\mu_\phi(s)$ and target networks $Q'_{\theta'}(s, a)$ and $\mu'_{\phi'}(s)$ with weights $\theta' \leftarrow \theta$ and $\phi' \leftarrow \phi$
for episode=1 to M do
    Receive initial observation state $s_t$
    for t=1 to T do
        Select $a_t = \mu_\phi(s_t) + \mathcal{N}_t$ according to current policy
        Execute $a_t$ and observe $r_t$ and $s_{t+1}$
        Store transition $(s_t, a_t, r_t, s_{t+1})$ in $B$
        DDPG Algorithm (continue)
    end for
end for
```

DDPG Algorithm (continue)

Sample a random minibatch of N transitions (s_i, a_i, r_i, s_{i+1})
Set $y_i = r_i + \gamma Q'_{\theta'}(s_{i+1}, \mu'_{\phi'}(s_{i+1}))$
Update critic by minimizing the loss:
$L = \frac{1}{N}\sum_i (y_i - Q_\theta(s_i, a_i))^2$
Update the actor policy using the sampled policy gradient:

$$\nabla_\phi J(\mu_\phi) \approx \frac{1}{N}\sum_i \nabla_\phi \mu_\phi(s) \nabla_a Q^\mu_\theta(s,a)|_{a=\mu_\phi(s)}$$

Update the target networks:

$$\theta' \leftarrow \tau\theta + (1-\tau)\theta' \text{ and } \phi' \leftarrow \tau\phi + (1-\tau)\phi'$$

Extensions: Maximum Entropy

- In RL the randomness in the selection of the actions defines the exploration mechanism, which can be defined in terms of a probability distribution and measured with entropy
- As the policy function converges, the entropy decreases
- The maximum entropy approach adds an entropy term called the *entropy bonus*:

$$\nabla_\theta \log \pi_\theta(a_t|s_t)(G_t - \hat{V}(s_t;w) + \eta \nabla_\theta \mathcal{H}(\pi_\theta(a_t|s_t)))$$

which prevents the agent to converge too fast and promotes the agent to take less predictive actions, where:

$$\mathcal{H}(\pi(a|s)) = -\sum_a \pi(a|s) \log \pi(a|s) = \mathbb{E}_{a\sim\pi(\cdot|s)}[-\log \pi(a|s)]$$

Soft-Actor Critic (SAC)

- An off-policy algorithm for continuous spaces that uses an entropy regularizer in π and Q
- Tries to tackle two of the main problems of DRL: (i) High sample complexity and (ii) Brittle convergence that needs a careful tuning of hyperparameters
- The next state actions are taken from the current policy
- Uses the reparameterization trick with a squashed Gaussian policy

Soft-Actor Critic (SAC)

- The Bellman equation changes from:

$$Q^{\pi}(s,a) = \mathbb{E}_{s' \sim P, a' \sim \pi}\left[r(s,a,s') + \gamma Q^{\pi}(s',a')\right]$$

- To:

$$Q^{\pi}(s,a) = \mathbb{E}_{s' \sim P, a' \sim \pi}\left[r(s,a,s') + \gamma(Q^{\pi}(s',a') + \alpha \mathcal{H}(\pi(\cdot|s')))\right]$$

$$Q^{\pi}(s,a) = \mathbb{E}_{s' \sim P, a' \sim \pi}\left[r(s,a,s') + \gamma(Q^{\pi}(s',a') - \alpha \log \pi(a'|s'))\right]$$

Soft-Actor Critic (SAC)

- The policy maximizes V^π which becomes:

$$V^\pi(s) = \mathbb{E}_{a' \sim \pi}\left[Q^\pi(s,a) - \alpha \log \pi(a|s))\right]$$

- Which makes use of the *reparamaterization trick*: Rewrite the expectation so the distribution from which we take the gradient is independent of the parameter θ
- Squashed Gaussian policy: Uses `tanh` to ensure that actions are bounded to a finite range

SAC Algorithm

```
Initialize parameters $\theta$, $\phi_1$, $\phi_2$
**repeat**
    Observe state and select action $a \sim \pi_\theta(a|s)$
    Execute $a$, observe next state $s'$, reward $r$, and terminal signal $ts$
    Store in replay buffer $D \leftarrow D \cup \{(s, a, r, s', ts)\}$
    **for** each gradient step **do**
        Randomly sample a minibatch of transitions, $B = \{(s, a, r, s', ts)\}$ from $D$
        (1) Compute targets for the $Q$ functions
        (2) Update $Q$ functions by one step gradient descent
        (3) Update policy function by one step gradient ascent
        (4) Update target networks
    **end for**
**until** convergence
```

SAC Algorithm (recent) - details

1. Compute targets for the Q functions (uses the min of two Q's):

$$y(r, s', ts) = r + \gamma(1 - ts)\left(\min_{i=1,2} Q_{\phi_{targ,i}}(s', \hat{a}') - \alpha \log \pi_\theta(\hat{a}'|s')\right)$$

   where $\hat{a}' \sim \pi_\theta(\cdot|s')$

2. Update Q functions by one step gradient descent (MSE):

$$\nabla_{\phi_i} \frac{1}{|B|} \sum_{(s,a,r,s',ts)\in B} \left(Q_{\phi_i}(s,a) - y(r,s',ts)\right)^2, \text{ for } i = 1,2$$

SAC Algorithm (recent) - details

1. Update policy function by one step gradient ascent:

$$\nabla_\theta \frac{1}{|B|} \sum_{s, \in B} \left(\min_{i=1,2} Q_{\phi_i}(s, \hat{a}_\theta(s)) - \alpha \log \pi_\theta(\hat{a}_\theta(s)|s) \right.$$

   where $\hat{a}_\theta(s)$ is a sample from $\pi_\theta(\cdot|s)$ with reparameterization and squashed values

2. Update target networks:

$$\phi_{targ,i} \leftarrow \rho\phi_{targ,i} + (1-\rho)\phi_i \text{ for } i = 1, 2$$

The reparameterization trick (used in VAE)

If we take the gradient, w.r.t. θ of an expectation:

$$\begin{aligned}\nabla_\theta \mathbb{E}_{p(z)}(f_\theta(z)) &= \nabla_\theta \left[\int_z p(z) f_\theta(z) dz\right] \\ &= \int_z p(z) \left[\nabla_\theta f_\theta(z)\right] dz \\ &= \mathbb{E}_{p(z)} \left[\nabla_\theta f_\theta(z)\right]\end{aligned}$$

The gradient of the expectation is equal to the expectation of the gradient.
If the density depends on θ:

$$\begin{aligned}\nabla_\theta \mathbb{E}_{p_\theta(z)}(f_\theta(z)) &= \nabla_\theta \left[\int_z p_\theta(z) f_\theta(z) dz\right] \\ &= \int_z \nabla_\theta \left[p_\theta(z) f_\theta(z)\right] dz \\ &= \int_z f_\theta(z) \nabla_\theta p_\theta(z) dz + \int_z p_\theta(z) \nabla_\theta f_\theta(z) dz \\ &= \int_z f_\theta(z) \nabla_\theta p_\theta(z) dz + \mathbb{E}_{p(z)} \left[\nabla_\theta f_\theta(z)\right]\end{aligned}$$

We may not have an analytic solution for the first term.

The reparameterization trick

- The reparameterization trick allows us to rewrite the expectation over actions, which depends on θ, into an expectation over noise, that does not depend on θ:

$$\mathbb{E}_{a \sim \pi_\theta} \left[Q^{\pi_\theta}(s, a) - \alpha log \pi_\theta(a|s) \right]$$

$$\mathbb{E}_{\xi \sim \mathcal{N}} \left[Q^{\pi_\theta}(s, \tilde{a}_\theta(s, \xi)) - \alpha log \pi_\theta(\tilde{a}_\theta(s, \xi)|s) \right]$$

where

$$\tilde{a}_\theta(s, \xi) = \tanh(\mu_\theta(s) + \sigma_\theta(s) \odot \xi), \xi \sim \mathcal{N}(0, I)$$

Distributed Schemes

- Several approaches have been used to design distributed schemes to scale up the DRL algorithms
- Here we will review some of them:
  1. A3C
  2. Ape-X
  3. R2D2

A3C

- *Experience replay* was used to avoid highly correlated data and create random samples
- However, it requires large storage capacity and it is applicable to *off-policy* algorithms
- A3C proposes to learn from multiple agents in parallel, considering that each agent will have different experiences (no need for a replay memory)
- A3C runs on a “standard” CPU with several cores

A3C

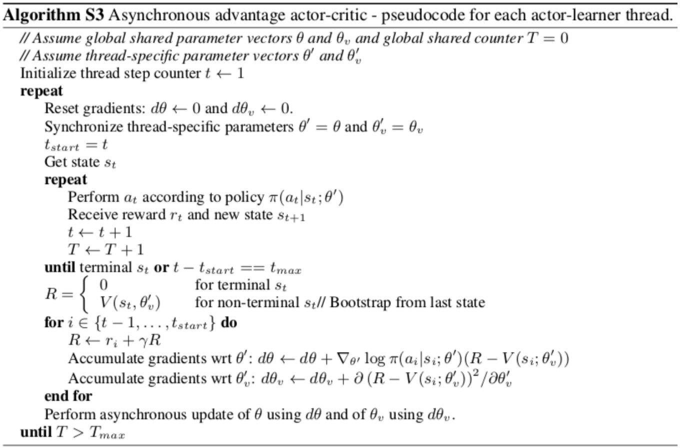
- Different policies are tried in different threads with parallel updating
- Each agent has its own copy of the environment, evaluates its gradient and shares the network that evaluates the loss function
- Eligibility traces are used in *forward view*
- The policy and value functions are updated after t_{max} or when reaching a terminal state

A3C

They tried different options:

- One-step asynchronous Q-learning
- One-step asynchronous SARSA
- N-step asynchronous Q-learning with eligibility traces
- Asynchronous Actor-Critic with *advantage* function (A3C or *asynchronous advantage actor-critic*)

A3C Algorithm



Algorithm S3 Asynchronous advantage actor-critic - pseudocode for each actor-learner thread.

```
// Assume global shared parameter vectors $\theta$ and $\theta_v$ and global shared counter $T = 0$
// Assume thread-specific parameter vectors $\theta'$ and $\theta'_v$
Initialize thread step counter $t \leftarrow 1$
repeat
    Reset gradients: $d\theta \leftarrow 0$ and $d\theta_v \leftarrow 0$.
    Synchronize thread-specific parameters $\theta' = \theta$ and $\theta'_v = \theta_v$
    $t_{start} = t$
    Get state $s_t$
    repeat
        Perform $a_t$ according to policy $\pi(a_t|s_t;\theta')$
        Receive reward $r_t$ and new state $s_{t+1}$
        $t \leftarrow t + 1$
        $T \leftarrow T + 1$
    until terminal $s_t$ or $t - t_{start} == t_{max}$
    $R = \begin{cases} 0 & \text{for terminal } s_t \\ V(s_t, \theta'_v) & \text{for non-terminal } s_t \text{// Bootstrap from last state} \end{cases}$
    for $i \in \{t-1, \ldots, t_{start}\}$ do
        $R \leftarrow r_i + \gamma R$
        Accumulate gradients wrt $\theta'$: $d\theta \leftarrow d\theta + \nabla_{\theta'} \log \pi(a_i|s_i;\theta')(R - V(s_i;\theta'_v))$
        Accumulate gradients wrt $\theta'_v$: $d\theta_v \leftarrow d\theta_v + \partial\left(R - V(s_i;\theta'_v)\right)^2 / \partial\theta'_v$
    end for
    Perform asynchronous update of $\theta$ using $d\theta$ and of $\theta_v$ using $d\theta_v$.
until $T > T_{max}$
```

In practice:
$\nabla_{\theta'} log\pi(a_t|s_t;\theta')(R_t - V(s_t;\theta_v)) + \beta\nabla_{\theta'} H(\pi(a_t|s_t;\theta'))$ where
H means entropy

Ape-X

- Ape-X generates data in parallel, uses *prioritized experience replay* and a single learning agent
- Uses a centralized *experience replay* memory
- Combines data from actors that have different exploration policies, which increases diversity in the examples
- Uses Double Q-Learning with eligibility traces and a *dueling* network architecture
- For all the elements in the batch the loss function is:

$$l_t(\theta) = \frac{1}{2}(G_t - Q(s_t, a_t; \theta))^2$$

Ape-X

With:

$$G_t = \underbrace{r_{t+1} + \gamma r_{t+2} + \ldots + \gamma^{n-1} r_{t+n} + \gamma^n \overbrace{Q(s_{t+n}, argmax_a Q(s_{t+n}, a; \theta); \theta^-)}^{\text{double-Q bootstrap value}}}_{\text{multi-step return}}$$

where:

- θ^- are the parameters of the target network. If the episode ends before the n steps it is truncated. The actors use $\epsilon-$greedy with different ϵ values

Ape-X Algorithm

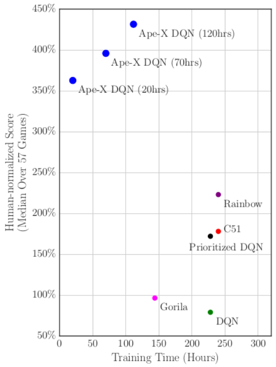
Algorithm 1 Actor

```
procedure ACTOR($B, T$)                          ▷ Run agent in environment instance, storing experiences.
    $\theta_0 \leftarrow$ LEARNER.PARAMETERS( )            ▷ Remote call to obtain latest network parameters.
    $s_0 \leftarrow$ ENVIRONMENT.INITIALIZE( )             ▷ Get initial state from environment.
    for $t = 1$ to $T$ do
        $a_{t-1} \leftarrow \pi_{\theta_{t-1}}(s_{t-1})$            ▷ Select an action using the current policy.
        $(r_t, \gamma_t, s_t) \leftarrow$ ENVIRONMENT.STEP($a_{t-1}$)     ▷ Apply the action in the environment.
        LOCALBUFFER.ADD($(s_{t-1}, a_{t-1}, r_t, \gamma_t)$)    ▷ Add data to local buffer.
        if LOCALBUFFER.SIZE( ) $\geq B$ then     ▷ In a background thread, periodically send data to replay.
            $\tau \leftarrow$ LOCALBUFFER.GET($B$)    ▷ Get buffered data (e.g. batch of multi-step transitions).
            $p \leftarrow$ COMPUTEPRIORITIES($\tau$)   ▷ Calculate priorities for experience (e.g. absolute TD error).
            REPLAY.ADD($\tau, p$)                ▷ Remote call to add experience to replay memory.
        end if
        PERIODICALLY($\theta_t \leftarrow$ LEARNER.PARAMETERS())     ▷ Obtain latest network parameters.
    end for
end procedure
```

Algorithm 2 Learner

```
procedure LEARNER($T$)                    ▷ Update network using batches sampled from memory.
    $\theta_0 \leftarrow$ INITIALIZENETWORK( )
    for $t = 1$ to $T$ do                                   ▷ Update the parameters $T$ times.
        $id, \tau \leftarrow$ REPLAY.SAMPLE( )  ▷ Sample a prioritized batch of transitions (in a background thread).
        $l_t \leftarrow$ COMPUTELOSS($\tau; \theta_t$)          ▷ Apply learning rule; e.g. double Q-learning or DDPG
        $\theta_{t+1} \leftarrow$ UPDATEPARAMETERS($l_t; \theta_t$)
        $p \leftarrow$ COMPUTEPRIORITIES( )      ▷ Calculate priorities for experience, (e.g. absolute TD error).
        REPLAY.SETPRIORITY($id, p$)                          ▷ Remote call to update priorities.
        PERIODICALLY(REPLAY.REMOVETOFIT())        ▷ Remove old experience from replay memory.
    end for
end procedure
```

Ape-X Performance

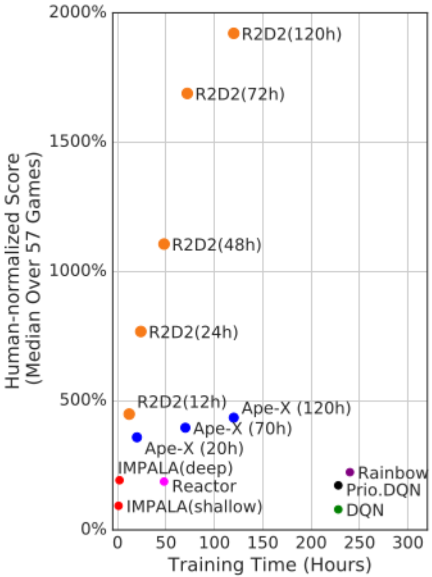


- They found that increasing the number of actors improves performance (more experience)

R2D2

- Recently people have incorporated LSTMs to deal with partial information
- *Recurrent Replay Distributed DQN* (R2D2) trains recurrent networks with a distributed *experience replay*
- Similar to Ape-X (*prioritized distributed replays*) and $n-$step double Q-Learning (with $n = 5$), generating experience from 256 actors, but includes a layer of LSTM after the convolutional layers
- Instead of storing the transition tuples (s, a, r, s'), sequences of fixed size (80) of (s, a, r) tuples are stored with overlaps of adjacent sequences every 40 steps

R2D2 Performance



Go

- Compared to Chess, it has more legal movements per position ($\approx$ 250 vs. $\approx$ 35) and more movements per game ($\approx$ 150 vs. $\approx$ 80)
- Is difficult to define an adequate value function
- AlphaGo combines Deep Learning, Monte Carlo Tree Search (MCTS), Supervised and Reinforcement Learning
- Modifies MCTS using value functions
- Defines (and learns) several networks: the policy network using supervised learning, the Monte Carlo simulations network, and the value function network

AlphaGo Zero

- Learns from self-play through policy iteration: Evaluation and improvement of the policy
- Uses MCTS to select actions and a single CNN
- MCTS runs a simulation until a leaf node of the current search tree (instead to running until a terminal state)
- The inputs to the CNN are tensors of $19 \times 19 \times 17$, representing 17 planes of binary attributes (8 represents the stones of the player, 8 from the opponent player and 1 indicating the color of the player's turn)
- The network learns: $f_\theta(s) = (p, v)$, with $p = Pr(a|s)$ and v = probability of wining from the current position

AlphaGo Zero

- In the search tree each node has the following information: $\{N(s,a), W(s,a), Q(s,a), P(s,a)\}$, where:
    - $N(s,a)$ how many times it has been visited
    - $W(s,a)$ the total value function
    - $Q(s,a)$ the average value function ($Q(s,a) = \frac{W(s,a)}{N(s,a)}$)
    - $P(s,a)$ the *a priori* action probability function (what the network learns)

AlphaGo Zero

- At each step an action is selected:
  $a_t = \text{argmax}_a(Q(s_t, a) + U(s_t, a))$
  where:
  $$U(s,a) = cP(s,a)\frac{\sqrt{\sum_b N(s,b)}}{1 + N(s,a)}$$
- MCTS is used to improve the policy

AlphaGo Zero

- The CNN has 41 convolutional layers which at the end is divided into two:
  1. One branch generates 362 outputs ($19^2 + 1$) that gives the probability of a movement ($Prob(a|s)$) at each place in the board + *pass* (don't do anything)
  2. The other branch generates a single output that estimates the probability of winning from the current positions (v)
- The network was trained using batches of random examples taken from 500 thousand games with the best current policy
- Each 1,000 training steps the new network is tested and if it improves certain threshold it is used

AlphaGo Zero

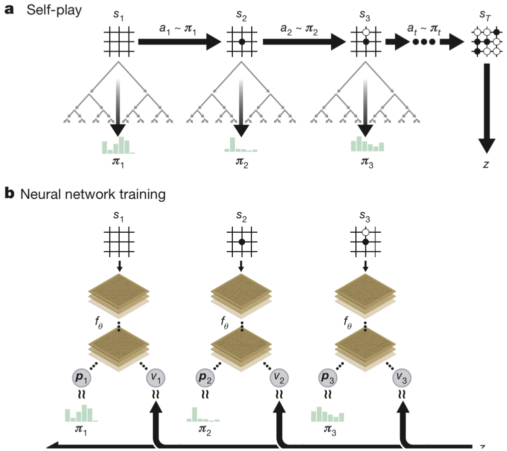
- It was trained with 4.9 millions of self-play and took roughly three days
- Each move from each game is selected by running MCTS for 1,600 iterations (roughly 0.4 seconds per move)
- The network weights are updated over 700,000 batches, each one with 2,048 positions
- At each position MCTS is executed guided by the network
- MCTS outputs a policy (π) with the probabilities of each move which is used to improve the policy

AlphaGo Zero

- The network is trained to maximize the similarity between the prediction p and the policy π obtained with MCTS and to minimize the error between the prediction of who is going to win v and the actual result z
- The loss function is:

$$l = (z - v)^2 - \pi^T \log p + c||\theta||^2$$

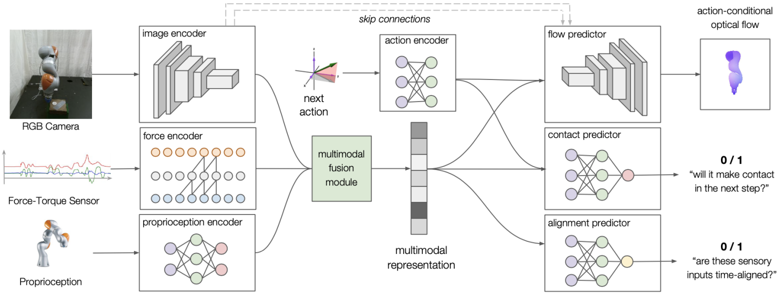
AlphaGo Zero



Deep RL and Robotics

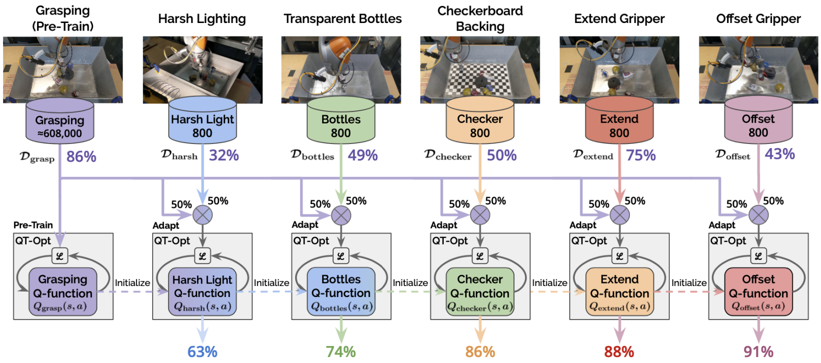
- Besides games, the other area that has received considerable attention with DRL is robotics
- It has been applied to robotic arms, mobile robots, drones, and autonomous vehicles
- We are going to illustrate some examples

Making sense of vision and touch: Self-supervised learning of multi-modal representations for contact-rich tasks



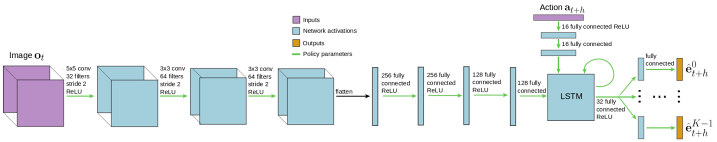
Combines multi-modal information to predict the optical flow contact and alignment

Efficient adaptation for end-to-end vision-based robotic manipulation


Grasping (Pre-Train)
Harsh Lighting
Transparent Bottles
Checkerboard Backing
Extend Gripper
Offset Gripper
Grasping ≈608,000
Harsh Light 800
Bottles 800
Checker 800
Extend 800
Offset 800
$\mathcal{D}_{\text{grasp}}$ 86%
$\mathcal{D}_{\text{harsh}}$ 32%
$\mathcal{D}_{\text{bottles}}$ 49%
$\mathcal{D}_{\text{checker}}$ 50%
$\mathcal{D}_{\text{extend}}$ 75%
$\mathcal{D}_{\text{offset}}$ 43%
50%
50%
Pre-Train
Adapt
QT-Opt
$\mathcal{L}$
Grasping Q-function $Q_{\text{grasp}}(s,a)$
Harsh Light Q-function $Q_{\text{harsh}}(s,a)$
Bottles Q-function $Q_{\text{bottles}}(s,a)$
Checker Q-function $Q_{\text{checker}}(s,a)$
Extend Q-function $Q_{\text{extend}}(s,a)$
Offset Q-function $Q_{\text{offset}}(s,a)$
Initialize
63%
74%
86%
88%
91%

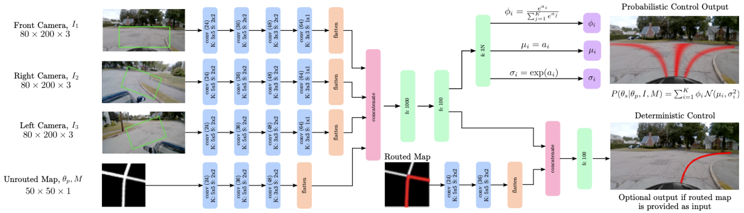
Re-trains a learned model to adjust to the changes in the environment

BADGR: An autonomous self-supervised learning-based navigation system



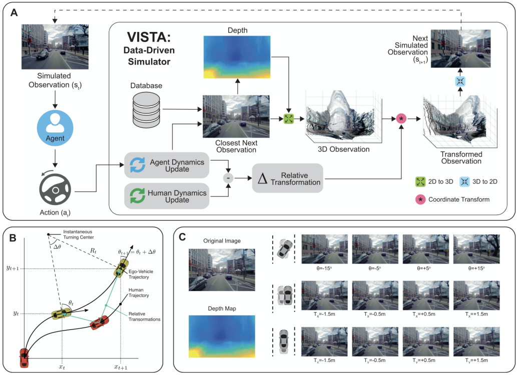
Self-supervised, receives information from the environment and learns a predicted model of relevant events

Variational end-to-end navigation and localization



Uses three cameras and an abstract route and decides how to navigate or how to follow a route

Learning robust control policies for end-to-end autonomous driving from data-driven simulations



Uses a simulator to predict images/scenarios in order to train the system

Deep RL

- Deep Reinforcement Learning is a fast growing research area
- The “natural” applications are games and robotics where there are great expectations for the future
- DRL has been also used in Natural Language Processing
- DRL requires, as DL, robust algorithms, be more data efficient, consume less computational resources, provide explanations of its results, ...

Deep RL

On-going work include:

- Transfer Learning (different rewards, different dynamics, different state-action spaces)
- Use additional knowledge (causal models, reward shaping, curriculum learning)
- Inlude humans in the learning loop (traces, feedback)
- How to deal with sparse rewards
- Life-long learning
- Combine with LLM
- ...

References

- **R. Sutton y A. Barto (2018). Reinforcement Learning: An Introduction. MIT Press (2a. edition).**
- Csaba Szepesvari (2010). Algorithms for Reinforcement Learning. Morgan & Claypool Publishers. Synthesis Lectures on Artificial Intelligence and Machine Learning. R.J. Brachman y T.G. Dietterich (editors).
- D.P. Bertsekas, J.N. Tsitsiklis (1997). Neuro-Dynamic Programming. Athena Scientific.

On-line Courses

- Additional material to the book of Sutton and Barto: http://incompleteideas.net/book/the-book-2nd.html
- Reinforcement Learning. D. Silver: http://www0.cs.ucl.ac.uk/staff/D.Silver/web/Teaching.html
- Deep Reinforcement Learning. S. Levine and others: http://rail.eecs.berkeley.edu/deeprlcourse/

Tools and Resources

- OpenAI Gym:
  1. https://gym.openai.com/
  2. https://github.com/openai/gym
  3. https://towardsdatascience.com/reinforcement-learning-with-openai-d445c2c687d2
  4. https://medium.com/@ashish_fagna/understanding-openai-gym-25c79c06eccb
- Mujoco:
  1. http://www.mujoco.org
  2. http://www.mujoco.org/book/

Tools ans Resources

- Stable baselines:
  1. https://pypi.org/project/stable-baselines/
  2. https://github.com/hill-a/stable-baselines
  3. https://stable-baselines.readthedocs.io/en/master/
- Animal AI:
  1. http://animalaiolympics.com/AAI/
  2. https://github.com/beyretb/AnimalAI-Olympics