Deep Reinforcement Learning

Special Focus: Continuous Control with Deep Deterministic Policy Gradients

John Berroa Felix Meyer zu Driehausen Alexander Höreth

15.06.2017

Institute of Cognitive Science, University of Osnabrück

Reinforcement Learning Recap

Policy Gradients

Deterministic Policy Gradient

Deep Deterministic Policy Gradient

Applications of Reinforcement Learning

Reinforcement Learning Recap

Problem Setting

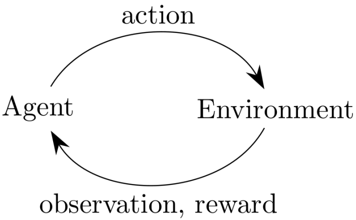


Figure 1: RL General Idea

- An agent is within an environment
- The agent is to complete some task and receive reward
- It solves this task over some amount of time steps

Silver (2015)

Markov Decision Processes

The environment in RL can be described as a Markov Decision Process

This relies on what's called a Markov State:

$$\mathbb{P}[S_{t+1}|S_t] = \mathbb{P}[S_{t+1}|S_1 ... S_t]$$

This means:

- The future is independent of the past, given the present
- The state is a sufficient statistic of the future
- All previous states can be thrown away and the same result will still be calculated

Note: For the Markov property to hold, the environment must be fully observable.

MDP: Observability

- In a *fully observable environment*, the agent's internal state is the same as the environment's internal state
  - i.e., the agent knows how the environment works exactly, and can therefore predict what each of its action will do with 100% accuracy
  - Put formally, the observation at time t is the same as both the agent's and environment's internal representations $S_t^e = O_t = S_t^a$
  - Can be represented with an MDP
- In a *partially observable environment*, the agent only indirectly observes the environment's state
  - The agent must construct it's own internal state based on its belief/construction of the environment state
  - Can be represented with a *Partially Observable Markov Decision Process*, POMDP

MDP: Now to the Markov Decision Process

"Decision" in *Markov* ***Decision*** *Process* means that actions need to be chosen–therefore, we add a policy to choose actions

An MDP can be represented as a tuple:

$$\langle S, A, P, R, \gamma \rangle$$

where:

S is the (finite) state space

A is a finite set of actions

P is the state transition matrix (matrix of state transition probabilities)

R is a reward function

γ is a discount factor, $\gamma \in [0, 1]$

MDP: Value Functions

The *State-Value Function* depends on the policy, and determines how good it is to be in a given state:

$$V^{\pi}(s) = \mathbb{E}_{\pi}(R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ...|S_t)$$

"The expectation when we sample all actions according to this policy π"; the value of a state

The *Action-Value Function* is defined as how good it is to take a particular action when the agent is in a particular state:

$$Q^{\pi}(s,a) = \mathbb{E}_{\pi}\left(\sum_{k=0} \gamma^k R_{t+k+1} \middle| S_t = s, A_t = a\right)$$

"The expected return starting from state s, taking action a, and then following policy π"; the value of an action

MDP: Solving Reinforcement Learning

We want to maximize the value of our actions based on future reward:

$$V^*(s) = \max_a Q^*(s, a)$$

$$Q^*(s, a) = R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a V^*(s')$$

We can nest these to get:

$$Q^*(s, a) = R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a \max_a Q^*(s, a)$$

This is the *Bellman Optimality Equation* (note: it can be nested in the other direction too to solve for $V^*(s)$)

Solve this, and the reinforcement learning problem is solved.

Q-Learning

Method to solve Q^*

- Iteratively act through *episodes*
- *"Backpropagate"* reward in order to calculate Q values which tell the values of actions
- Take the maximum Q value at each time step
- Store Q values into a table

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha_t \cdot \left(R_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t) \right)$$

Q-Learning – Expanded

DQN

- Uses a neural network to approximate Q^*
- Can be thought of as looking at a state row in the Q-table, then taking the argmax

Continuous Actions

- Discretizing the action space almost always leads to combinatorial explosion:
  - Consider discretizing the human arm (7-DoF) into Up/Straight/Down – $3^7 = 2187$ dimensional action space
- If we can't (or don't want to) discretize the action space, the Q-table becomes incalculable (would equate to an infinite length table)
- Therefore, Q-learning (and by extension DQN) will not work when dealing with continuous actions

Q-Learning – Q-Table Example

State	Stay	Left	Right	Forward	Backward
0	0	-1	-1	0	0
1	0	0	-1	-1	10
2	0	0	-1	-1	10
3	0	-1	0	0	-1
4	0	0	0	0	10

Figure 2: Q-Table

Q-Learning – Deterministic Policies

- A deterministic policy can lead an agent into an infinite loop
- Imagine this rule from some policy:
  - "Whenever there is a wall to the north and south, go left"
- If we applied this policy to the problem pictured, the agent would get stuck
- This can be solved by using a stochastic policy and leaving ϵ active during test time

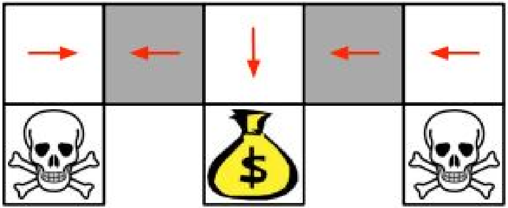


Figure 3: Deterministic Failure

Some Helpful Definitions

To follow everything to come, it is necessary to have a general grasp of the following concepts:

- **Agent**: what performs actions in the environment; wants to maximize future reward
- **Environment**: where the agent resides and what gives observations and reward; interacted with by agent
- **Reward**: R_t; reward at time step t, scalar
- **Observation**: O_t; what the environment shows the agent at step t, after an action
- **Action**: A_t; the action taken at step t, performed by agent
- **History**: sequence of observations, actions, and rewards up to current time step; i.e. $H_t = A_1, O_1, R_1 ... A_t, O_t, R_t$
- **State**: a function of history; $S_t = f(H_t)$; the information used to determine what happens next

Some Helpful Definitions

- **Fully observable**: the environment state equals the agent state; $S_t^a = S_t^e$; the agent knows the complete dynamics of the environment
- **Partially observable**: the agent must make an assumption about the environment because it doesn't know it's dynamics
- **Model**: the agent's internal representation of the environment
- **Policy**: π; what the agent uses to map states to actions; tells the agent what to do
- **Deterministic policy**: a state will always lead to a certain action; $\pi(s) = a$
- **Stochastic policy**: a state will yield a probability of actions to choose from; $\pi(a|s) = \mathbb{P}(A = a|S = s)$
- **State-Value Function**: tells the value of a state based on the expected future reward
- **Action-Value Function**: tells the value of an action based on expected future reward

Some Helpful Definitions: Types of RL Algorithms

- Value Based (e.g. Q-Learning)
  - No policy (implicit)
  - Learnt Value function
- Policy Based (e.g. Policy Gradient)
  - Learnt Policy
  - No value function
- ***Actor-Critic*** (e.g. DDPG)
  - Learnt Policy
  - Learnt Value function
- Model Based/Model Free
  - Learnt Policy and/or Value function
  - Based: has model; Free; no model

Policy Gradients

Policy Objective Function

- Plan: Given policy $\pi_\theta(s, a)$ with parameters θ, find the best θ
- How can we measure the goodness of a policy?
- We consider episodic environments with a start state
- The goodness of policy is the return gained when coming from the start state
- This is denoted by the performance objective:

$$J_1(\theta) = V^{\pi_\theta}(s_1) = \mathbb{E}_\pi(R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \ldots | S_t)$$

We will find the gradient of the performance objective with respect to θ. This allows us to change θ in order to maximize the performance objective.

Adjusting with respect to a Score Function

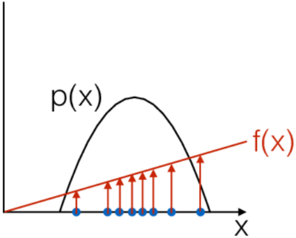


Figure 4: Evaluating f(x)

- Sampling x from $p(x)$
- Evaluating $f(x)$

Adjusting with respect to to a Score Function

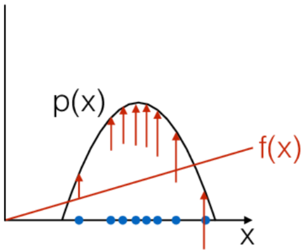


Figure 5: Pushing p(x) w.r.t f(x)

- Manipulate $p(x)$ according to magnitude of $f(x)$
- Re-normalize $p(x)$

Goal: Change p(x) such that it "produces" x which in turn result in high values in the score function. **But how to change** $p(x)$?

Score Function Gradient Estimator

$$
\begin{aligned}
\nabla_\theta \mathbb{E}_x[f(x)] &= \nabla_\theta \sum_x p(x|\theta) f(x) && \text{definition of expectation} \quad (1)\\
&= \sum_x \nabla_\theta p(x|\theta) f(x) && \text{swap sum and gradient} \quad (2)\\
&= \sum_x p(x|\theta) \frac{\nabla_\theta p(x|\theta)}{p(x|\theta)} f(x) && \text{both multiply and divide by } p(x|\theta) \quad (3)\\
&= \sum_x p(x|\theta) \nabla_\theta \log p(x|\theta) f(x) && \text{use the fact that } \nabla_\theta \log(z) = \frac{1}{z}\nabla_\theta z \quad (4)\\
&= \mathbb{E}_x[f(x) \nabla_\theta \log p(x|\theta)] && \text{definition of expectation} \quad (5)
\end{aligned}
$$

Now we need to sample $x_i \sim \; p(x|\theta)$, and compute

$$\hat{g}_i = f(x_i) \nabla_\theta \log(p(x_i|\theta))$$

Score Function Gradient Estimator Intuition

$$\hat{g}_i = f(x_i)\nabla_\theta \log p(x_i|\theta)$$

- $f(x)$ measures how good the sample x is (score function)
- Stepping (ascending) in the direction $\hat{g}_i$ increments the log probability of the x, proportionally to the score
- x which yield good scores in f become more probable

Score Function Gradients in Context of Policies

In the context of policies the random variable x is a whole trajectory $\tau = (s_0, a_0, r_0, s_1, a_1, r_1, ..., s_{T-1}, a_{T-1}, r_{T-1}, s_T)$

Previous slide:

$$\hat{g}_i = f(x_i)\nabla_\theta \log p(x_i|\theta)$$

Now in the context of policies:

$$\nabla_\theta \mathbb{E}_\tau[R(\tau)] = \mathbb{E}_\tau[\nabla_\theta \log\ p(\tau|\theta)R(\tau)]$$

Now we detail $p(\tau|\theta)$:

$$p(\tau|\theta) = \mu(s_0) \prod_{t=0}^{T-1} [\pi(a_t|s_t, \theta)P(s_{t+1}, r_t|s_t, a_t)]$$

Score function Gradients in context of policies II

$$\log\, p(\tau|\theta) = \log\, \mu(s_0) + \sum_{t=0}^{T-1} [\log\, \pi(a_t|s_t,\theta) + \log\, P(s_{t+1}, r_t|s_t, a_t)]$$

Now differentiating with respect to θ

$$\nabla_\theta \log\, p(\tau|\theta) = \nabla_\theta \sum_{t=0}^{T-1} [\log\, \pi(a_t|s_t,\theta)$$

The gradient is not dependent on the state transition distribution $P(s_{t+1}, r_t|s_t, a_t)$.
Inserting back into the expectation yields:

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

Stochastic Policy Gradient Theorem

- When using the state-action value function Q^π for R the policy gradient is:

$$\nabla_\theta \mathbb{E}_\tau[R] = \mathbb{E}_\tau[\sum_{t=0}^{T-1} \nabla_\theta \log \pi(a_t|s_t, \theta) Q^\pi(s_t, a_t)]$$

Silver et al. (2014)

Actor - Critic

Actor - Critic is an Architecture based on the Policy Gradient Theorem.

- *Actor* adjusts the parameters θ of the policy π
- This is done by ascent of the policy gradient
- The real $Q^{\pi}(s, a)$ is unknown
- Therefore function $Q^{w}(s, a)$ with parameters w is approximated (e.g with deep neural network)
- A *critic* estimates parameters the action-value function Q^{w}

Deterministic Policy Gradient

Policies in Continuous Action Space

Problem:

- The action value is in $\mathbb{R}$
- At every step this requires to evaluate the action-value function Q globally over (at least a subset of) $\mathbb{R}$
- This is infeasible

Solution:

- Do not maximize over Q
- But move policy in direction of Q
- The policy is now deterministic, giving a real valued number

Policies in Continuous Action Space

Specifically:

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

where $\mu_\theta(s)$ is the deterministic policy

Now applying the chain rule:

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

Determinisitc Policy Gradient Theorem

The deterministic policy gradient now is:

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

Silver et al. (2014)

Deep Deterministic Policy Gradient

From DQN to Deep Deterministic Policy Gradient – Code!

DDPG is very similar to DQN implementation-wise – just with some added bells and whistles. **If you plan to implement DDPG, you might want to start with DQN.**

- Define an environment with observations, rewards and actions.
- Repeatedly act in the environment using the current policy & store experiences.
- Q network as value function approximator, optimized using the Bellman equation.
- **New:** Policy network for continuous actions, optimized using policy gradient.
- Online & target network split. **New:** Soft updates.

```
import tensorflow as tf
```

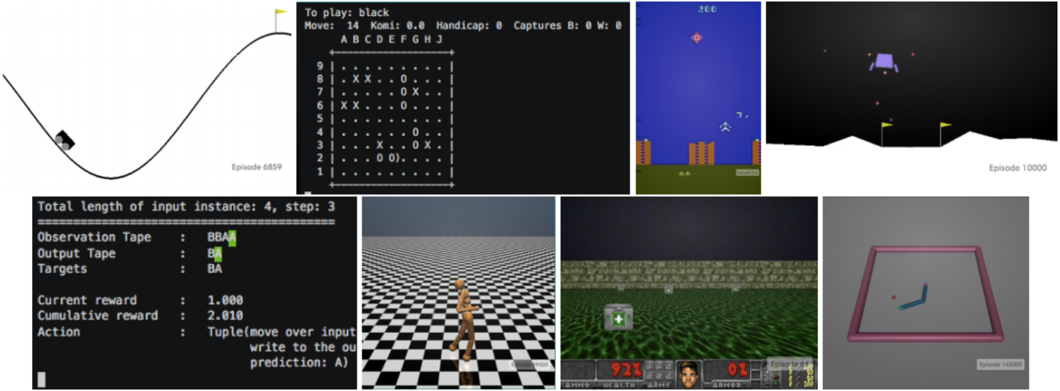


Figure 6: OpenAI Gym Environments (Brockman et al. 2016)[1]

[1]github.com/openai/gym

OpenAI Gym: API

```python
import gym
```

Sensible standardized interface for RL environments. When creating custom environments, building on top of its specifications might make sense.

```python
env = gym.make('LunarLanderContinuous-v2')
env.observation_space  # e.g. float vector, 3D array...
env.action_space  # e.g. integer, float vector...
env.reset()
action = env.action_space.sample()
state, reward, done, info = env.step(action)
env.render()
```

Off-Policy Reinforcement Learning: Generating Samples

Act in the environment following the current policy to generate experiences, store them.

```python
from collections import deque
memory = deque([], maxlen=1e6)  # Note: Random access is O(n)!
policy = lambda state: env.action_space.sample()
done = True
while True:
  if done:
    state = env.reset()
  action = policy(state)
  state_, reward, done, _ = env.step(action)
  memory.append((state, action, reward, state_))
  state = state_
```

The Critic (aka Value Network)

DDPG: $Q(s, a) \to q$

State & action to single Q value.

Lillicrap et al. (2015)

DQN: $Q(s) \to \vec{q}$

State to Q vector, one value per action.

Mnih et al. (2015)

```python
def make_critic(states, actions, name):
  with tf.variable_scope(name) as scope:
    net = tf.layers.dense(states, 400, tf.nn.relu)  # Feature extract
    net = tf.concat([net, actions], axis=1)
    net = tf.layers.dense(net, 300, tf.nn.relu)  # Value estimate
    q = tf.layers.dense(net, 1)  # shape (BATCHSIZE, 1)
    return tf.squeeze(q), get_variables(scope)
```

Training Q-Networks: Bellman Approximation

DQN vs. DDQN vs. DDPG – fine differences in estimating future reward.

$$y^{DQN} = r_t + \gamma max_a Q'(s_{t+1}, a) \qquad \text{Greedy estimate.} \tag{6}$$

$$y^{DDQN} = r_t + \gamma Q'(s_{t+1}, argmax_a Q(s_{t+1}, a)) \qquad \text{Estimate by online policy.} \tag{7}$$

$$y^{DDPG} = r_t + \gamma Q'(s_{t+1}, \mu'(s_{t+1})) \qquad \text{Estimate by detached policy.} \tag{8}$$

Mnih et al. (2015), Van Hasselt, Guez, and Silver (2016), Lillicrap et al. (2015)

Training the DDPG Critic: Bellman Approximation & Mean Squared Error

The critic is optimized to minimize the mean squared error loss between its output and the Bellman approximation.

$$y = r_t + \gamma Q'(s_{t+1}, \mu'(s_{t+1})) \qquad \text{Critic Target} \tag{9}$$

$$\mathbb{L} = \frac{1}{N}\sum^{N}(Q(s_t, a_t) - y)^2 \qquad \text{Critic Loss} \tag{10}$$

```
# critic, _ = make_critic(states, actions, 'online')
# critic_, _ = make_critic(states_, actor_, 'target')
def train_critic(critic, critic_, terminals, rewards):
  targets = tf.where(terminals, rewards, rewards + .99 * critic_)
  mse = tf.reduce_mean(tf.squared(targets - critic))
  return tf.train.AdamOptimizer(1e-3).minimize(mse)
```

The Actor (aka Policy Network)

DDPG: $\mu(s) \rightarrow a$

Vector of continues action values.

Lillicrap et al. (2015)

DQN: $argmax_a Q(s, a) \rightarrow a$

Greedy discrete action selection.

Mnih et al. (2015)

```
def make_actor(states, n_actions, name):
  with tf.variable_scope(name) as scope:
    net = dense(states, 400, tf.nn.relu)
    net = dense(net, 300, tf.nn.relu)
    y = dense(net, n_actions, tf.nn.tanh)  # Action scaling.
    return y, get_variables(scope)
```

Training the Actor (Policy Gradient Ascent)

Ascend the gradients of the critic network with respect to the online actor's actions.

$$\begin{aligned} \Delta_{\theta^\mu} J &\approx \Delta_{\theta^\mu} Q(s_t, a) & a = \mu(s_t|\theta^\mu) \quad (11) \\ &= \Delta_a \; Q(s_t, a) \Delta_{\theta^\mu} a & F'(x) = f'(g(x))g'(x) \quad (12) \end{aligned}$$

```python
# actor, thetaMu = make_actor(states, 4, 'online')
# critic, _ = make_critic(states, actor, 'online')
def train_actor(actor, thetaMu, critic):
  value_gradient, = tf.gradients(critic, actor)
  policy_gradients = tf.gradients(actor, thetaMu, -value_gradient)
  mapping = zip(policy_gradients, thetaMu)
  return tf.train.AdamOptimizer(1e-4).apply_gradients(mapping)
```

Target Network Updates

```
# _, theta = make_critic(states, actions, 'online')
# _, theta_ = make_critic(states_, actor_, 'target')
```

Hard Updates: Common in DQN implementations and on initial initialization.

```
def make_hard_update(theta, theta_):
  return [dst.assign(src) for src, dst in zip(theta, theta_)]
```

Soft updates: Slowly follow online parameters, prevents oscillation.

```
def make_soft_update(theta, theta_, tau=1e-3):
  return [dst.assign(tau * src + (1 - tau) * dst)
          for src, dst in zip(theta, theta_)]
```

What kind of monster did we just create?

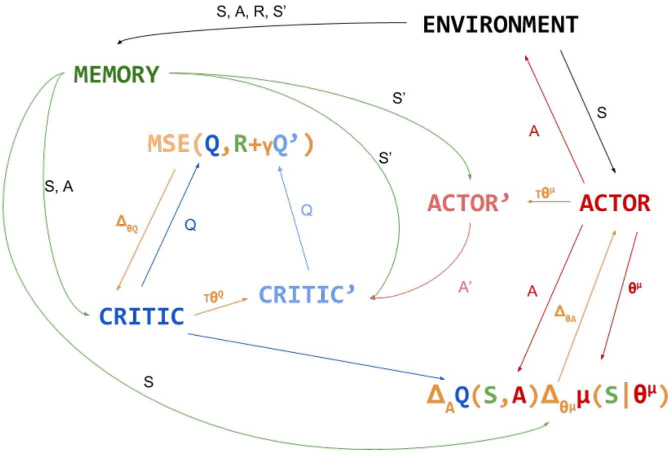


Figure 7: DDPG Dataflow Graph – TensorBoard failed us

Exploration in Continuous Environments

DQN: ϵ-greedy – only for discrete actions.
DDPG: Gaussian continuous through time with friction θ and diffusion σ (Uhlenbeck and Ornstein 1930).

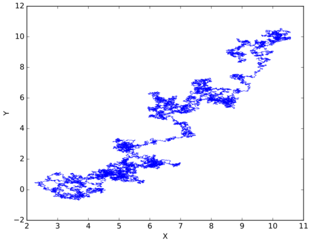


Figure 8: Prototypical Process

```python
def noise(n, theta=.15, sigma=.2):
  state = tf.Variable(tf.zeros((n,)))
  noise = -theta * state + sigma * tf.random_normal((n,))
  return state.assign_add(noise)
```

Do it yourself DDPG

All of the above and more at **github.com/ahoereth/ddpg** $\rightarrow$ **Lander.ipynb**

- Exhaustively documented. Would recommend if you are interested in Deep RL.
- Critic & actor, online & target networks with soft & hard updates.
- Batch normalization – disabled because it didn't improve performance.
- Threaded feeding and training:
    - Main thread can focus on generating new experiences.
    - Some threads feed samples from the memory to the TensorFlow graph.
    - Some threads train the network as scheduled by the agent.
- TensorBoard logs with (not so pretty) graph of whats going on.

Applications of Reinforcement Learning

Cutting Edge Applications of Reinforcement Learning

We will discuss:

- Safety in Robotics and Reinforcement Learning
- Poker
- Multiple Agents

Fisac et al. (2017); Li (2017); Heinrich and Silver (2016); Lowe et al. (2017)

The General Problem of Robotics

- The world is full of noise
  - Great for neural networks!
- Simulations can't simulate the full range and accuracy of the real world, so training actual robots is best
- Danger to break or destroy robot or property
  - Robots are expensive

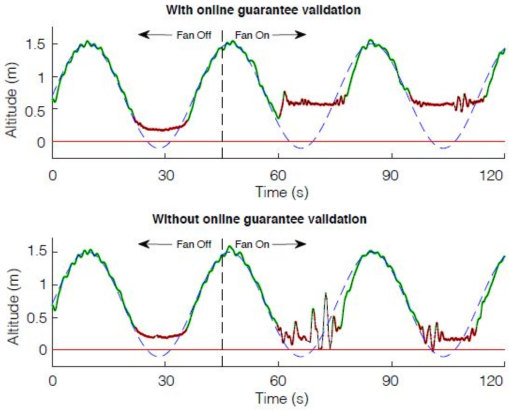
Keeping Robots Safe

- Since neural networks are “black boxes,” it is hard to pinpoint areas where training might lead to dangerous situations based on the weights of the model
- Safety has typically been guaranteed by a manual fallback mechanism or making the environment safe

Safety Framework

Solution:
Combines both a safety net and a bayesian mechanism online to deal with sudden changes in the environment

Youtube link:
https://youtu.be/WAAxyeSk2bw



The Cutting Edge

Let's now look at two applications recently published (last year and last month)

- Poker playing
- Reinforcement learning with multiple agents

Poker as an RL Problem

- Imperfect information game – the hands of other players are unknown, as well as the values of upcoming cards
- Multi-agent zero-sum game – Nash Equilibrium exists, but is incalculable

NFSP (Neural Fictitious Self Play)

- Applying neural networks to the concept of "Fictitious Self Play"
  - FSP = Choose the best response to the opponent's average behavior
- Approaches Nash Equilibrium as it learns

NFSP Poker: Architecture

- Remembers state transitions and the agent's best responses in two separate memories M_{RL} and M_{SL}
  - State transitions used for RL; Best responses used for supervised learning
- M_{RL} uses an off-policy deep RL algorithm to learn the best policy from the state transitions
- M_{SL} uses a feedforward net to learn the average play (in order to do fictitious self play)
- Target network for stability and has an explore parameter

NFSP Poker: Performance

- Comparable to other AIs based on expert knowledge representation (classic AI)
  - e.g. Smooth Upper Confidence Bounds and Counterfactual Regret Minimization+

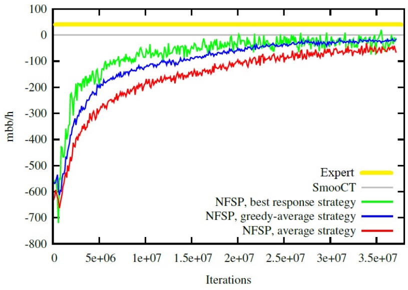


Figure 9: Poker Performance

The Problem

- Multiple agents affect the environment
  - Agent can't accurately predict environment because it is no longer based on its policy alone
  - Significantly increases the variability in policy gradient algorithms – this is because the reward in normal policy gradients is only conditioned on the agent's own actions

The Solution

- Actor-Critic with “centralized” training and “decentralized” execution.
  - The actor can not contain information about the other actors at both training and test time (would require additional assumptions)
  - Solve this by supplying the critic with the policies of all agents (centralized), while the actor remains isolated
  - At test time, only actors are used (decentralized)
  - “Since the centralized critic function explicitly uses the decision-making policies of other agents, we additionally show that agents can learn approximate models of other agents online and effectively use them in their own policy learning procedure”
- Ensemble of policies to make each individual agent robust to changes in other agents’ policies
- Named: MADDPG

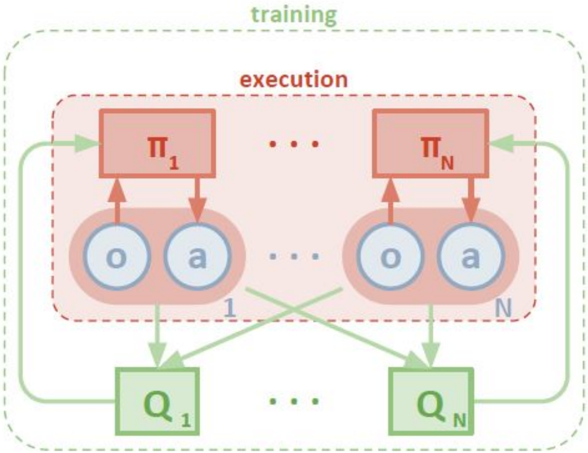


Figure 10: MultiNetwork

Multiagent Reinforcement Learning: Performance

- Trained on a battery of cooperative and competitive multi-agent tasks
- Outperformed DDPG significantly
- Youtube link: youtu.be/QCmBo91Wy64 (1:55)

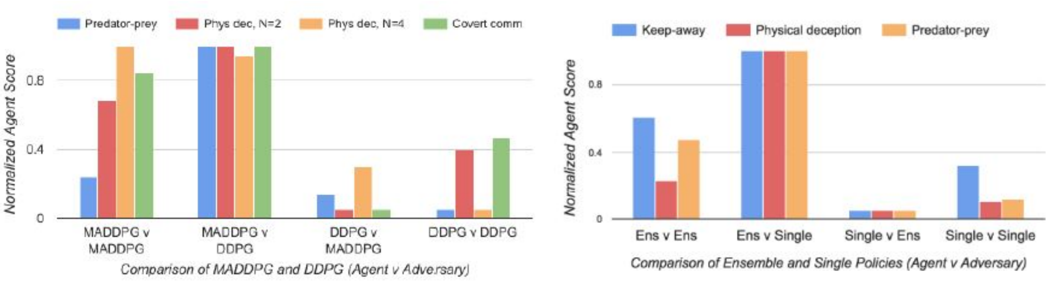


Figure 11: MADDPG Performance

References i

Brockman, Greg, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. 2016. "OpenAI Gym." *arXiv Preprint arXiv:1606.01540*.

Fisac, Jaime F., Anayo K. Akametalu, Melanie N. Zeilinger, Shahab Kaynama, Jeremy Gillula, and Claire J. Tomlin. 2017. "A General Safety Framework for Learning-Based Control in Uncertain Robotic Systems." *arXiv Preprint arXiv:1705.1292v2*.

Heinrich, Johannes, and David Silver. 2016. "Deep Reinforcement Learning from Self-Play in Imperfect-Information Games." *arXiv Preprint arXiv:1603.01121v2*.

Li, Yuxi. 2017. "Deep Reinforcement Learning: An Overview." *arXiv Preprint arXiv:1701.07274*.

Lillicrap, Timothy P, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. 2015. "Continuous Control with Deep Reinforcement Learning." *arXiv*

Preprint arXiv:1509.02971.

Lowe, Ryan, Yi Wu, Aviv Tamar, Jean Harb, Pieter Abbeel, and Igor Mordatch. 2017. "Multi-Agent Actor-Critic for Mixed Cooperative-Competitive Environments." *arXiv Preprint arXiv:1706.02275v1*.

Mnih, Volodymyr, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, et al. 2015. "Human-Level Control Through Deep Reinforcement Learning." *Nature* 518 (7540). Nature Research: 529–33.

Silver, David. 2015. "University College London: Reinforcement Learning Course." http://www0.cs.ucl.ac.uk/staff/d.silver/web/Teaching.html.

Silver, David, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, and Martin Riedmiller. 2014. "Deterministic Policy Gradient Algorithms." In *Proceedings of the 31st International Conference on Machine Learning (Icml-14)*, 387–95.

Uhlenbeck, George E, and Leonard S Ornstein. 1930. "On the Theory of the Brownian Motion." *Physical Review* 36 (5). APS: 823.

Van Hasselt, Hado, Arthur Guez, and David Silver. 2016. "Deep Reinforcement Learning with Double Q-Learning." In *AAAI*, 2094–2100.