Deep Learning Agents

Boi Faltings

Laboratoire d'Intelligence Artificielle
boi.faltings@epfl.ch
http://moodle.epfl.ch/

Deep Reinforcement Learning

Reinforcement learning does not scale well:

- For most real problems, state space is huge and requires too much memory.
- Exploring all state/action pairs is impossible because states are difficult to reach.

$\Rightarrow$ represent states as embeddings in a deep neural network.

$\Rightarrow$ learn policy directly as output of a neural net.

Example: Atari games

- Arcade Learning Environment: simulation of Atari videogames

Deep Q-learning

- Games: state is fully observable from the screen image.
- Use convolutional neural net with the image pixels and possible actions as input
- Output = function $Q(s, a)$
  = expected reward after taking action a.
- Choose action as in Q-learning (maximize Q).

See: `Mnih et al.: Playing Atari with Deep Reinforcement Learning, arXiv 1312.5602, 2013`

From MDP to Reinforcement Learning

- Reactive agents based on a **model**: Markov Decision Processes.
- Reward and state transition functions usually not known a priori, must be learned from observations.
- Strategies for optimal exploration are difficult to apply when there are many states: not only are there many states, but they also may be difficult to reach.

Model-free Reinforcement Learning

- Policy = function from state to optimal action.

$\Rightarrow$ approximate directly using a deep neural net.

- State is directly observable: feedforward neural net encodes environment into an embedding.
- State is not directly observable: use recurrent neural net to approximate state transition function.
- Neural net learns mapping from state embedding to optimal action.

Generalization

- Advantage of neural net: generalization over the space of states.

$\Rightarrow$ no need to systematically explore state/action pairs.

- However, still need to explore different actions to learn the reward in different situations.

Exploration Strategies

- Systematic exploration (UCB, regret matching) not feasible.
- Place agent in real-life training situations and record observations.
- Experience replay: data augmentation by replaying earlier experiences in random order to avoid dependencies among that hurt convergence of stochastic gradient descent.
- Curriculum learning: choose training situations that train particular skills as parts of the policy.
- Play against adversaries to train agents for competitive scenarios.

Delayed Rewards

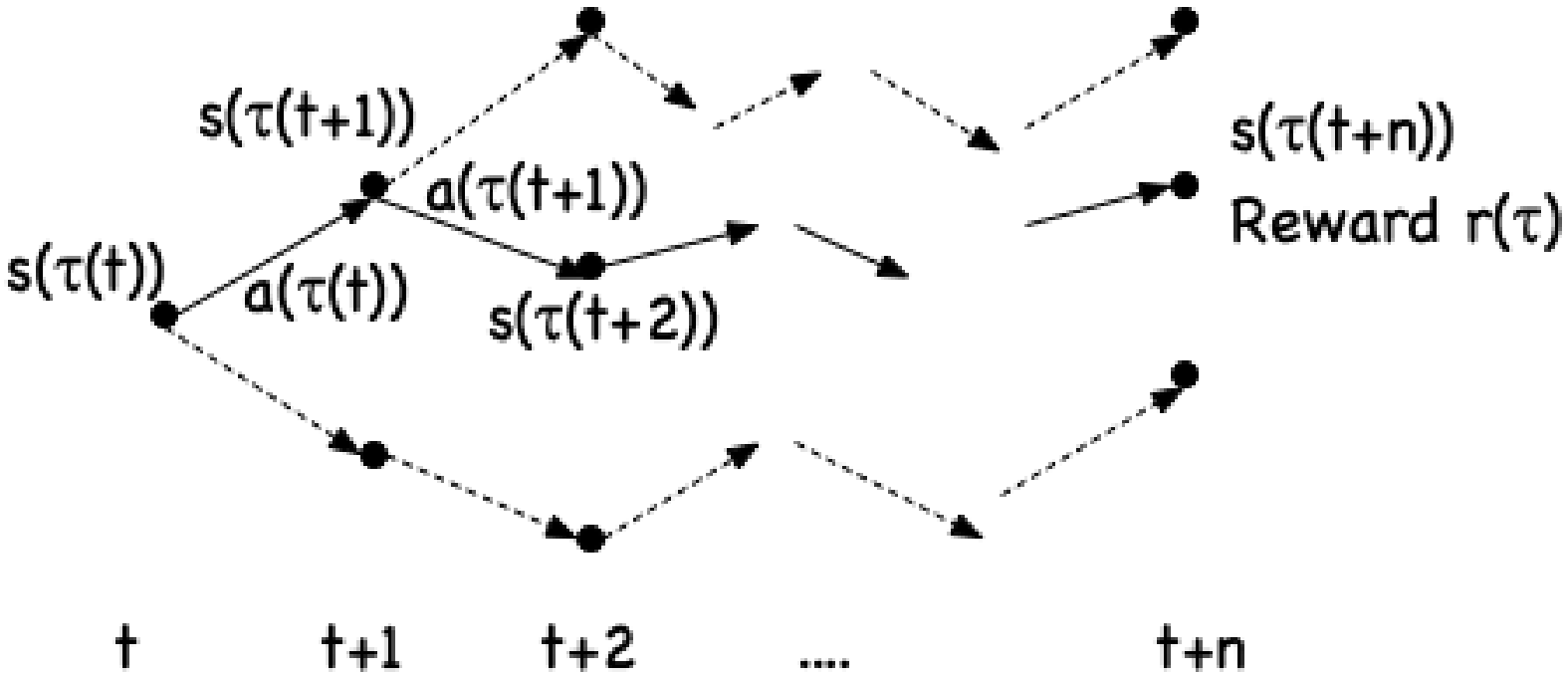
- Reward for the best action may only come later.
- Example: traveling incurs cost but reward only comes at the destination.

$\Rightarrow$ choosing the best immediate action requires evaluating a *sequence* of actions.

- Let $\tau(s)$ be a particular *trajectory* from the current state s at time t:
  - $a(\tau(t+i))$ = action taken at the i-the state of the trajectory.
  - $s(\tau(t+i))$ = i-th state of the trajectory.

  Learn using rewards from the entire trajectory.
- Use Monte-Carlo search (as in deliberative agents) when simulation is possible.

Trajectories



- Reward is known at the end of the trajectory
- Probability of a trajectory determined by policy π.

Trajectories in real life

Example video of learning to park a self-driving car:

`https://www.youtube.com/watch?v=VMp6pq6_QjI`

Difficulty of delayed reward: learning has to propagate reward to the many intermediate states.

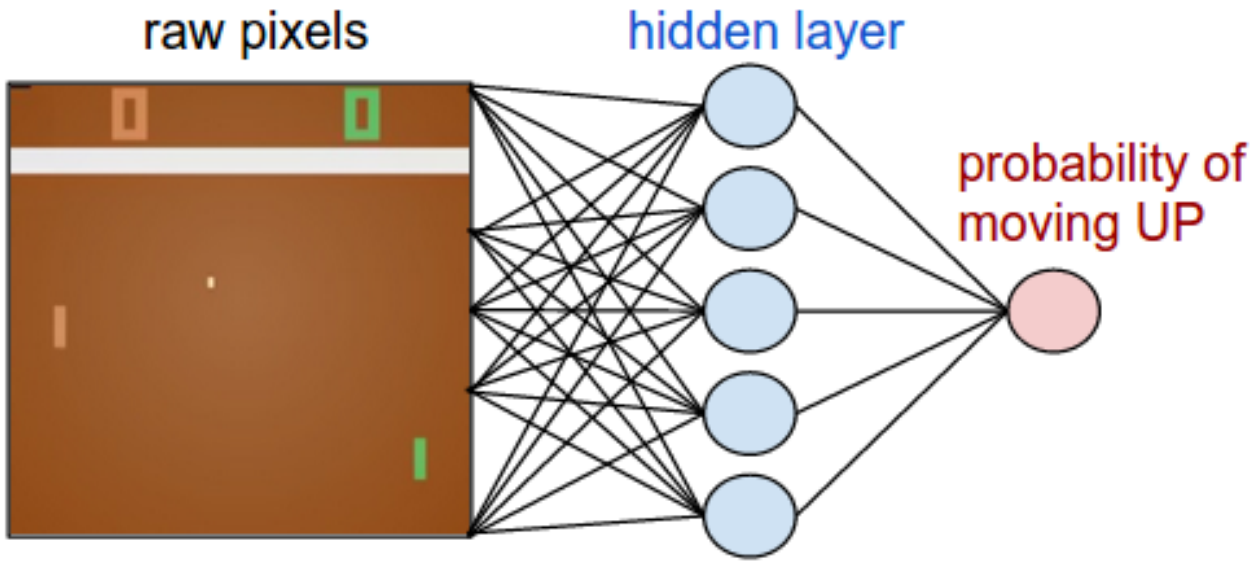
Mixed Policies

- Policy needs to be differentiable for neural net to learn using gradient descent.
- Use policy $\pi(a|s)$ = probability of playing action a in state s
- Implemented as neural net with parameters θ.
- Uncertain rewards and state transitions $\Rightarrow$ π should maximize *expected* reward over a future trajectory τ:

$$J(\theta) = E_\pi[r(\tau)]$$

- Find optimal policy using stochastic gradient descent.

Atari pong with policy gradient



- Actions = move up/move down
- Input = pixel image

See:

`https://github.com/omerbsezer/PolicyGradient_PongGame`

Optimizing the policy

- Gradient descent:

$$\theta_{t+1} = \theta_t + \alpha \nabla_\theta J(\theta_t)$$

- Expectation requires multiple experiences (or simulations):

$$J(\theta_t) = \int \pi(\tau) r(\tau) d\tau$$

For finite set of transitions, reward

$$r(\tau) = \sum r(a(\tau(t+i)), s(\tau(t+i)))$$

but probability of a trajectory τ is a product:

$$\pi(\tau) = \prod \pi(a(\tau(t+i)) | s(\tau(t+i)))$$

and the gradient of $\int \pi(\tau) r(\tau)$ is very complex.

Policy gradient theorem

- Using the fact that:

$$\nabla_\theta \log \pi(\tau) = \frac{\nabla_\theta \pi(\tau)}{\pi(\tau)}$$

rewrite gradient of expectation:

$$\begin{aligned}
\nabla_\theta E_\pi[r(\tau)] &= \nabla_\theta \int \pi(\tau) r(\tau) d\tau \\
&= \int \nabla_\theta \pi(\tau) r(\tau) d\tau \\
&= \int \pi(\tau) \nabla_\theta \log \pi(\tau) r(\tau) d\tau \\
&= E_\pi[r(\tau) \nabla_\theta \log \pi(\tau)]
\end{aligned}$$

Policy gradient theorem (2)

- Now $\log \pi(\tau)$ is a sum over the trajectory and we can compute the policy gradient as:

$$\begin{aligned} \nabla_\theta E_\pi[r(\tau)] &= E_\pi[r(\tau)\nabla_\theta \log \pi(\tau)] \\ &= \sum r(\tau(t+i))\nabla_\theta \log \pi(a(\tau(t+i))|s(\tau(t+i))) \end{aligned}$$

  which is a sum of local gradients for each state of the trajectory τ.

How to sample trajectories

- Recall from Q-learning: converges to optimal policy *only* if observations are an i.i.d. sample of the distribution in the application.
- But if agent only plays according to its policy, the best actions may never be discovered!
- Generating the right sample trajectories requires careful simulation of different scenarios.

Accuracy of reward estimates

- Complexity of simulating a trajectory increases exponentially with its length $\Rightarrow$ must be very limited.

$\Rightarrow$ compute an estimate V of the rewards obtained from the following state under current policy.

- Use this estimate as the expectation of the rewards from the next state onwards:

$$r(\tau) = r(s(\tau(t))) + \gamma V(s(\tau(t+1)))$$

similar to value iteration for MDP.

$\Rightarrow$ only observations of the immediate reward are needed.

- Lowers the variance of reward estimates: faster learning convergence.

Actor-Critic

- Estimate of the rewards $V(s)$ from the next state $\tau(s)$ onwards allows comparing different actions from the current state s.
- Expected reward of action a_j in state $s = s(\tau(t))$ as:

$$Q(a_j, s) = r(a_j, s) + \gamma V(s(\tau(t+1)))$$

- Define Advantage of action a_j as:

$$\mathcal{A}(a_j, s) = Q(a_j, s) - \sum_a \pi(a, s) Q(a, s) = Q(a_j, s) - V(s)$$

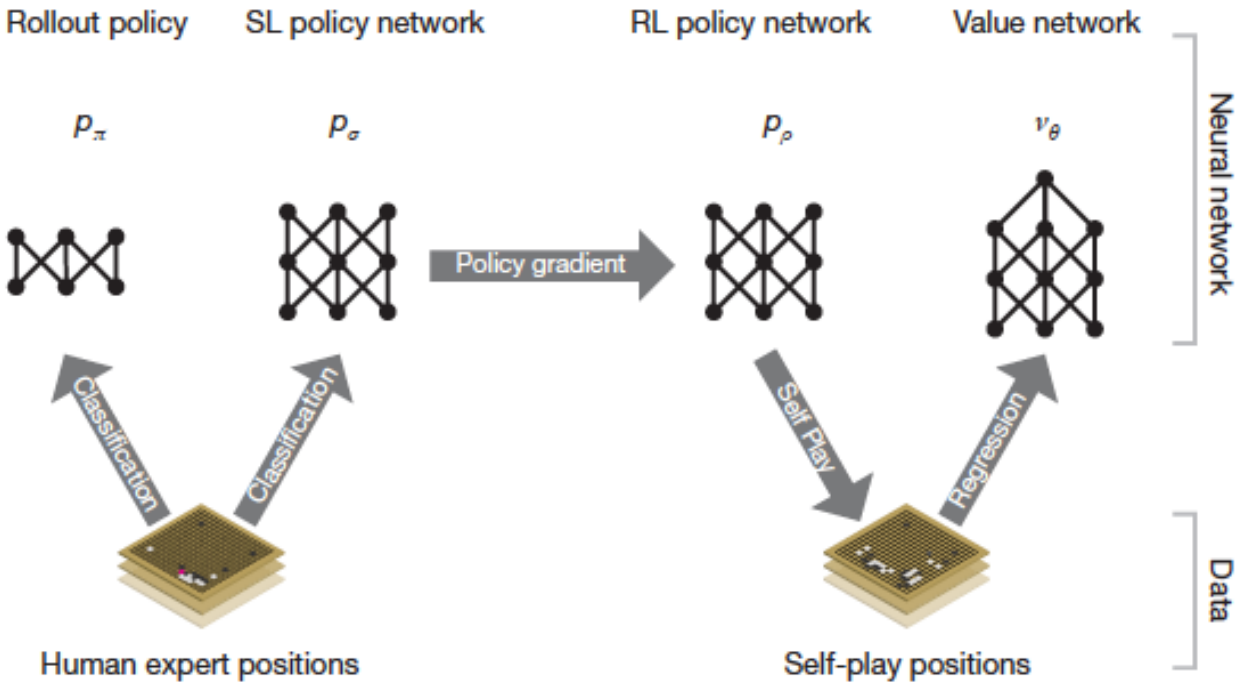
- Replace reward by Advantage = relative quality of action a_j.
- Actor = agent that executes current policy
- Critic = module for value estimate that computes advantage of each action to allow policy modification.

Estimating the value function

- State is observable: train a feedforward neural net, using the observations and chosen action as input to characterize the next state.
- Dynamics may also be important: use a sequence of the k last observations and actions as input.
- State is partially observable: use recurrent neural net (but not many good results so far).
- Generate data using the current policy.

Example: Alpha Go

- Alpha Go (first program to beat human champion at Go) used 2 networks: policy and value network:



Multitask Learning

- Predicting value and next move both use the same inputs: observations and actions.
- Performance can be improved using *multitask learning*: both objectives are learned simultaneously.
- Leads to dramatic performance improvements.
- Example Alpha Go $\Rightarrow$ Alpha Zero: general algorithm to learn any board game.

Convergence of SGD for policy gradients

Stochastic gradient descent for policy often does not converge:

- gradient can get very large.
- gradient strength varies a lot

$\Rightarrow$ difficult to set a good learning rate α.

Solution: Optimize a surrogate objective

- Replace gradient:

$$E_\pi[\nabla_\theta \log \pi(a,s)\mathcal{A}(a)]$$

by objective to find $\pi(a,s)$ that maximizes:

$$L^{PG} = E_\pi\left[\frac{\pi(a,s)}{\pi_{old}(a,s)}\mathcal{A}(a)\right]$$

- Trust region policy optimization (TRPO): add constraints for all actions a_i:

$$E_\pi[KL(\pi_{old}(a_i,s),\pi(a_i,s)] < \delta$$

- Limits change in π.

Proximal Policy Optimization (PPO)

- Let

$$d = \frac{\pi(a, s)}{\pi_{old}(a, s)}$$

and $clip(d)$ the function that clips d to $[1 - \epsilon..1 + \epsilon]$.

- Define surrogate objective:

$$L^{CLIP} = E_{\pi} min(d\mathcal{A}(a), clip(d(a)\mathcal{A}(a))$$

- Limits steps to improve objective, but guards against steps that cause large degradation.

Exploration-Exploitation tradeoff

Training data is gathered with a policy that avoids bad states.
Reasons:

- safety (self-driving car driving off the road)
- credibility (recommender showing irrelevant items)
- efficiency (many more bad than good states)

Resulting policy needs to generalize to off-policy situations with little or no training data.

Off-policy learning

Different degrees of generalization:

- different state distribution: importance sampling.
  (e.g.: driving in warmer/colder climate)
- unseen states: extrapolation of value function and policy.
  (e.g. car taking a turn too fast)

Importance sampling

- Policy is based on *expected rewards*:

$$\mathcal{R}(\pi) = \sum p(\tau) r(\tau)$$

- Distribution of states changes $\Rightarrow$ optimal policy changes as well.

$\Rightarrow$ reweight rewards for differences in distribution between observation Q and application P:

$$\mathcal{R}_P(\pi) = \frac{p(\tau)}{q(\tau)} \sum q(\tau) r(\tau)$$

- Works well as long as there is enough data about all states and actions ($r(a, s)$ is known).

Issues in importance sampling

- For rare τ, only few samples so variance of $q(\tau)$ is very high.

$\Rightarrow$ $\frac{p(\tau)}{q(\tau)}$ can become much too large: clip to avoid excessive influence.

- Optimize sampling so that variance of reweighted samples is minimized.

State interpolation

- Reward $r(a, s)$ not explored for all states and actions.

$\Rightarrow$ interpolate between known values.

- Simplest form: linear/polynomial models; least-squares approximation.
- More powerful: deep neural networks.

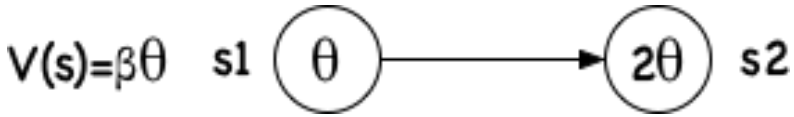
Runaway errors

- If interpolation overestimates rewards for some s, a, optimal policy will give higher weight to this state.

$\Rightarrow$ increase in value function estimate for s and neighbouring states.

$\Rightarrow$ even higher weight in policy.

Example



- At time t, $V(s1) = \theta = 10 \Rightarrow V(s2) = 2\theta = 20$; learning rate $\alpha = 0.1$

$\Rightarrow$ at time t+1, $V(s1) = \theta = 10 + \alpha(20 - 10) = 11$ and $V(s2) = 2\theta = 22$

$\Rightarrow$ at time t+2, $V(s1) = 11 + \alpha(22 - 11) = 12.1$, ...

- Even if true rewards for both states and actions are 0!

No convergence guarantees with value function/policy approximations!

Multiagent Reinforcement Learning

- Many real scenarios involve multiple agents.
- Multiagent learning: learn a combination of policies for all agents to obtain optimal results.
- Many agents $\Rightarrow$ number of action combinations explodes.

Settings

- Competitive: each agent receives an individual reward and acts to maximize this reward.
- Cooperative: reward is attributed to all agents, goal is to maximize combined welfare.
- Adversarial: agent(s) reward is the loss of the opponent(s).

Competitive Setting

- Agent rewards depend on each others' actions.

$\Rightarrow$ optimal policy depends on other agents and forms equilibria.

- Often solved using centralized intermediaries = game-theoretic mechanisms.
- Analyzed using game theory, treated later in the course.

Cooperative Setting

- Find a *joint* policy that optimizes the a function of all agents' rewards.
- Often uses sum, but this can be unfair: some agents may suffer very poor rewards in order to improve those of others.

$\Rightarrow$ combination often optimizes a fairness criterion, e.g. minimum reward of any agent.

- Usually computed by a central agent that computes a policy for all agents and communicates it to them.

Learning in cooperative settings

- Simplest: reward signal to each agent = sum (or fair function) of all agent rewards.
- Learning signal is imprecise: agents get the same reward even if their action had a negative effect.

$\Rightarrow$ convergence is very slow.

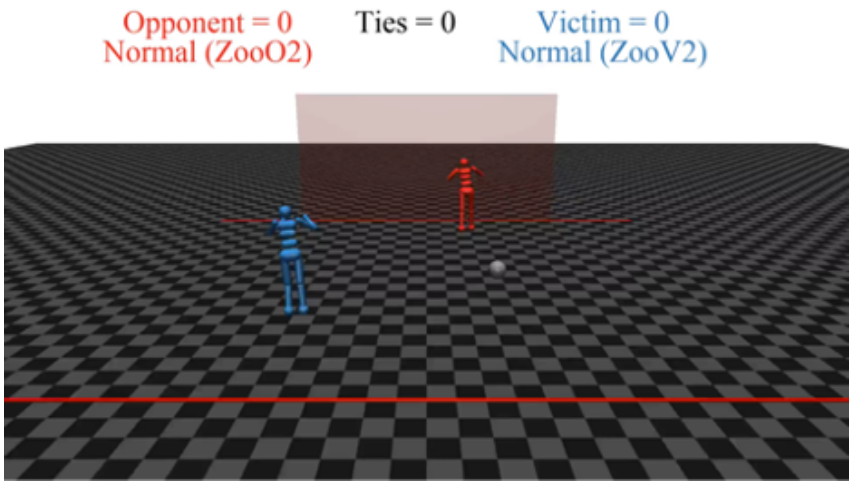
- Assigning credit to individual agents difficult because of combinatorics: contribution depends on other agents' actions.

Uniform policies

- Simplify by forcing every agent to use the same policy: combinatorics disappears.
- Can have additional inputs to identify individual roles, e.g. different positions of soccer players.
- agents can take each others' roles.
- agents can learn from each other.

Self-play

- Agents can learn to cooperate by simulated play against each other's policies.
- Advantage: unlimited amount of data.

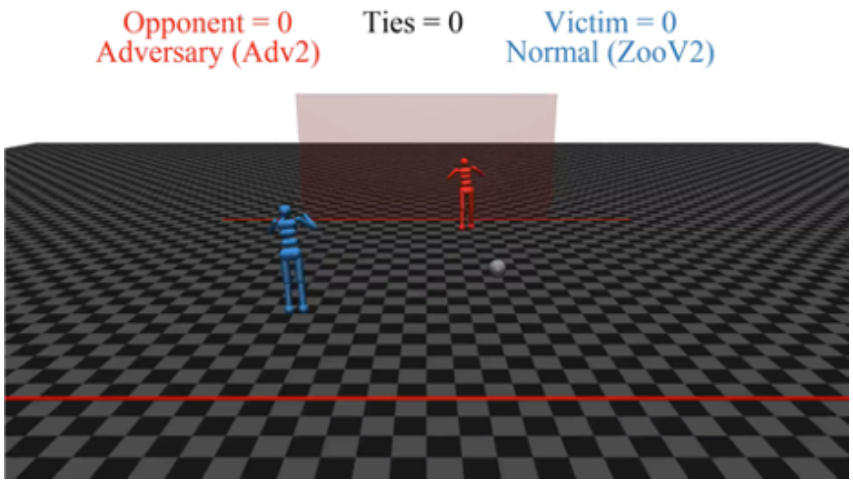


Adversarial Setting

- simpler form of competitive setting (zero-sum).
- requires mixed policies (see lecture on learning agents).
- learned strategy often tailored to particular opponent; policy learned against good soccer player fails completely when played against poor soccer player.

Exploitable policies

- Learned policies have an implicit dependence on that learned by adversaries.
- Can be exploitable when adversary uses a suboptimal policy.



Summary

- Q-table, values and policies can be approximated with neural networks.
- Allows reinforcement learning without a model.
- Policy gradient theorem.
- Actor-critic architecture, multi-task learning.
- Proximal policy optimization.
- Generalization allows learning even for unseen states and actions (off-policy learning).
- Can be extended to multiagent settings; training through simulations can leave exploitable weaknesses when agents do not follow policies used in training.