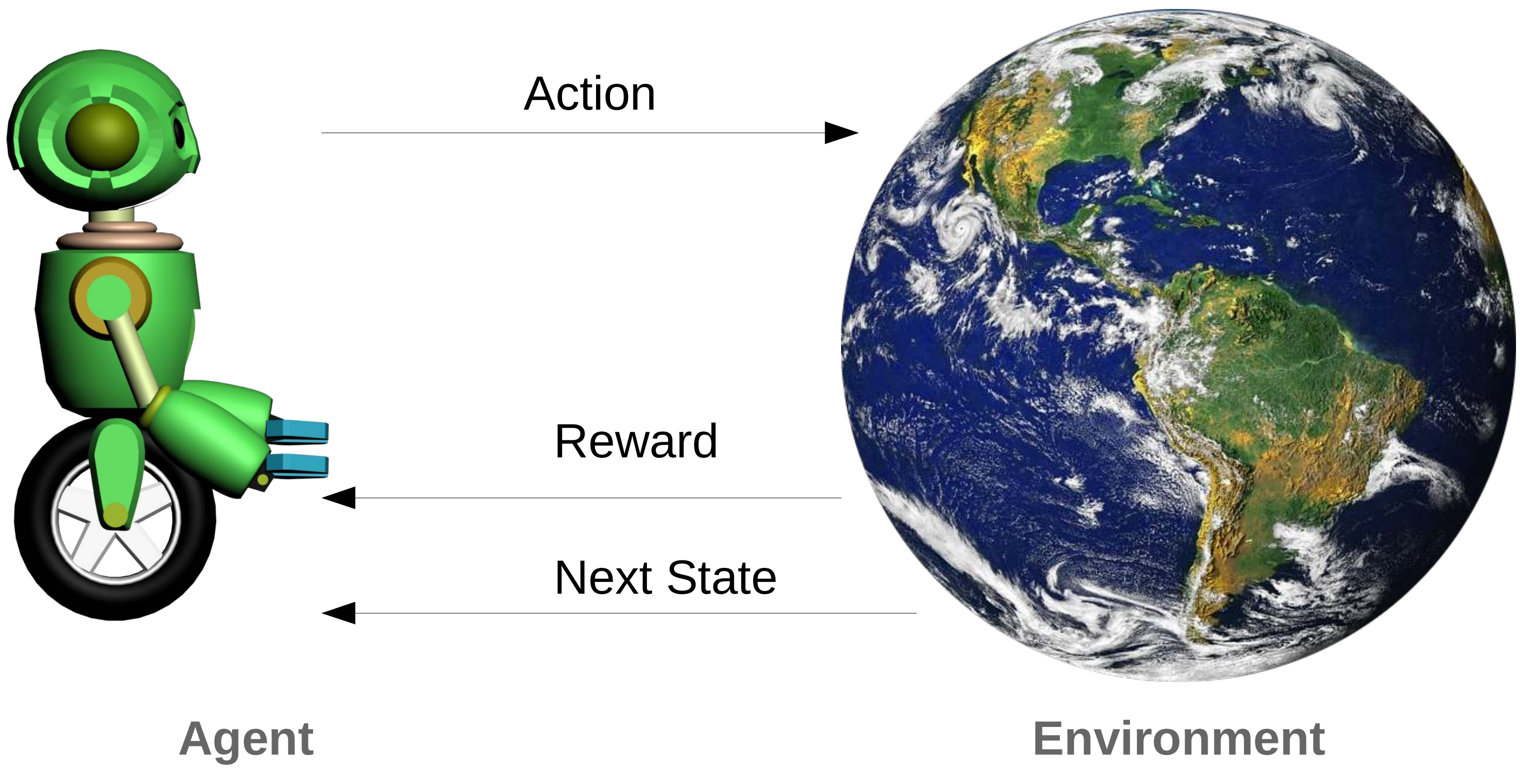
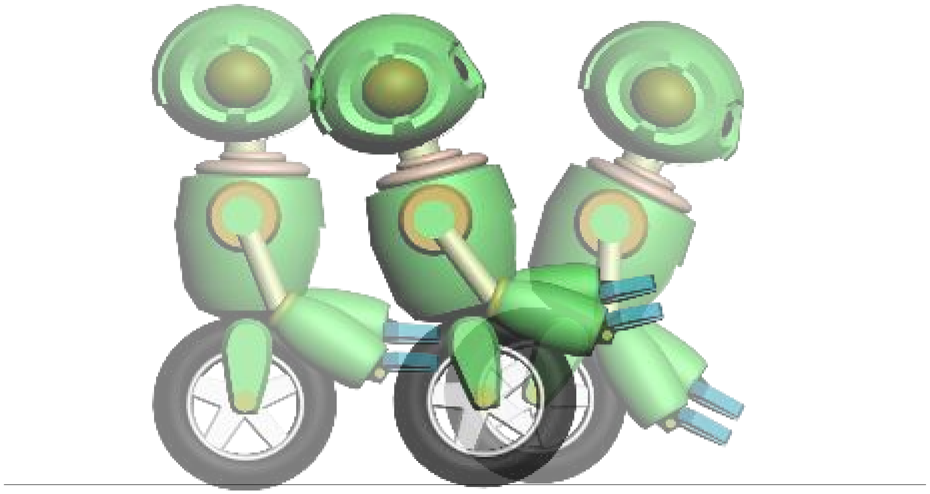
Reinforcement Learning

Reinforcement Learning



Example: Balancing

- Our robot needs to learn critical control routines
- In particular, balancing on the uniwheel
- Control problem

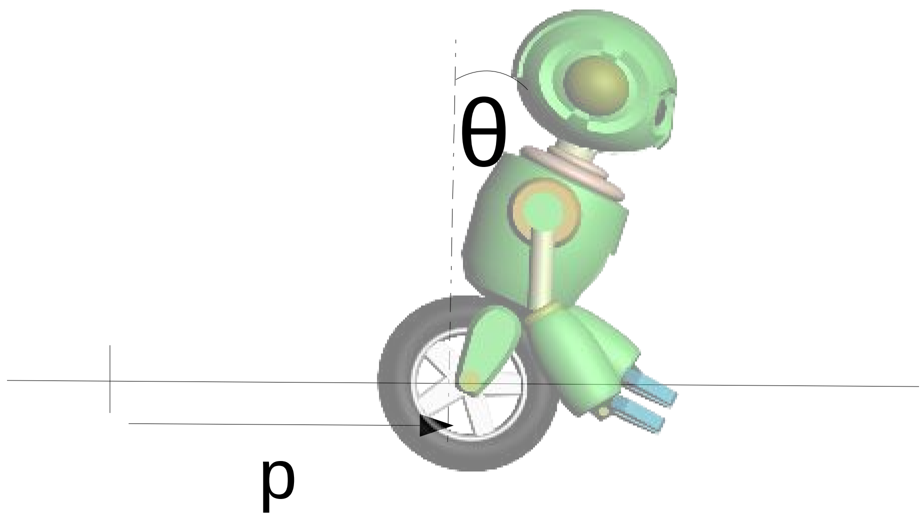


Learning Balancing Policies

State of the System $\mathbf{s}$

Action to execute $\mathbf{a}$

Policy linking states to actions $\mathbf{a} = \pi(\mathbf{x})$



State:

$$\mathbf{x} = \left[\theta, \dot{\theta}, p, \dot{p},\right]^T$$

Action:

$\mathbf{a}$ = Force

Policy:

$$\mathbf{a} = \pi(\mathbf{x})$$

What are Policies?

- A policy π is a function that returns an action given a state
- In a Q-learning setup π may be defined as:

$$\pi = \begin{cases} \max_a Q(s_t, a) & \text{if } \epsilon > k \\ a \text{ random} & \text{otherwise} \end{cases}$$

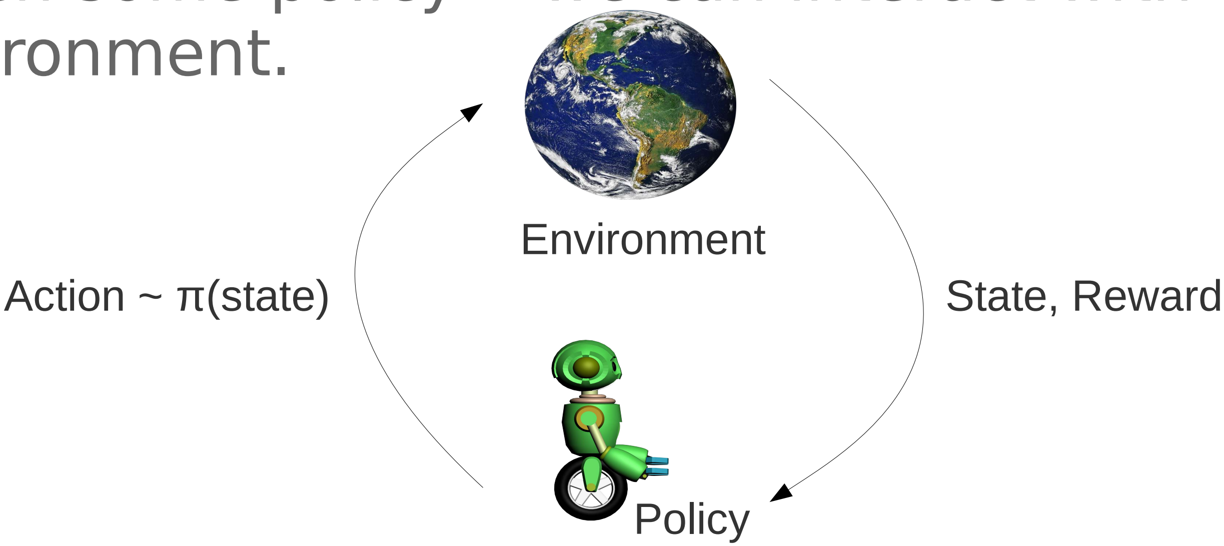
where ϵ is a parameter controlling exploration

- In policy gradient methods the policy directly estimates the action: $a \sim \pi(a|s)$

where π encodes a distribution over actions.

Sampling from a Policy

- Given some policy π we can interact with the environment.



- This results in a set of visited states, their corresponding rewards, and the actions that took the agent there.
- This sequence is a trajectory or **policy rollout**.

Approximating a Policy

- When the state or action spaces are large approximations are necessary.
- In discrete spaces with n actions:

$$\pi = \text{Multinomial}(\theta_1, \ldots, \theta_n)$$

  The parameters are estimated from the current state. In practice, this means actions are sampled from a softmax output.

- In continuous action spaces:

$$\pi = \mathcal{N}(\mu, \sigma^2 I)$$

  Estimate the mean (and sometimes variance) of a continuous distribution, often Gaussian.

Example for a Discrete Policy

- These estimates may be performed by any function approximator, including linear models or a neural network.
- Concretely, if we employ a linear model in an environment with discrete actions:

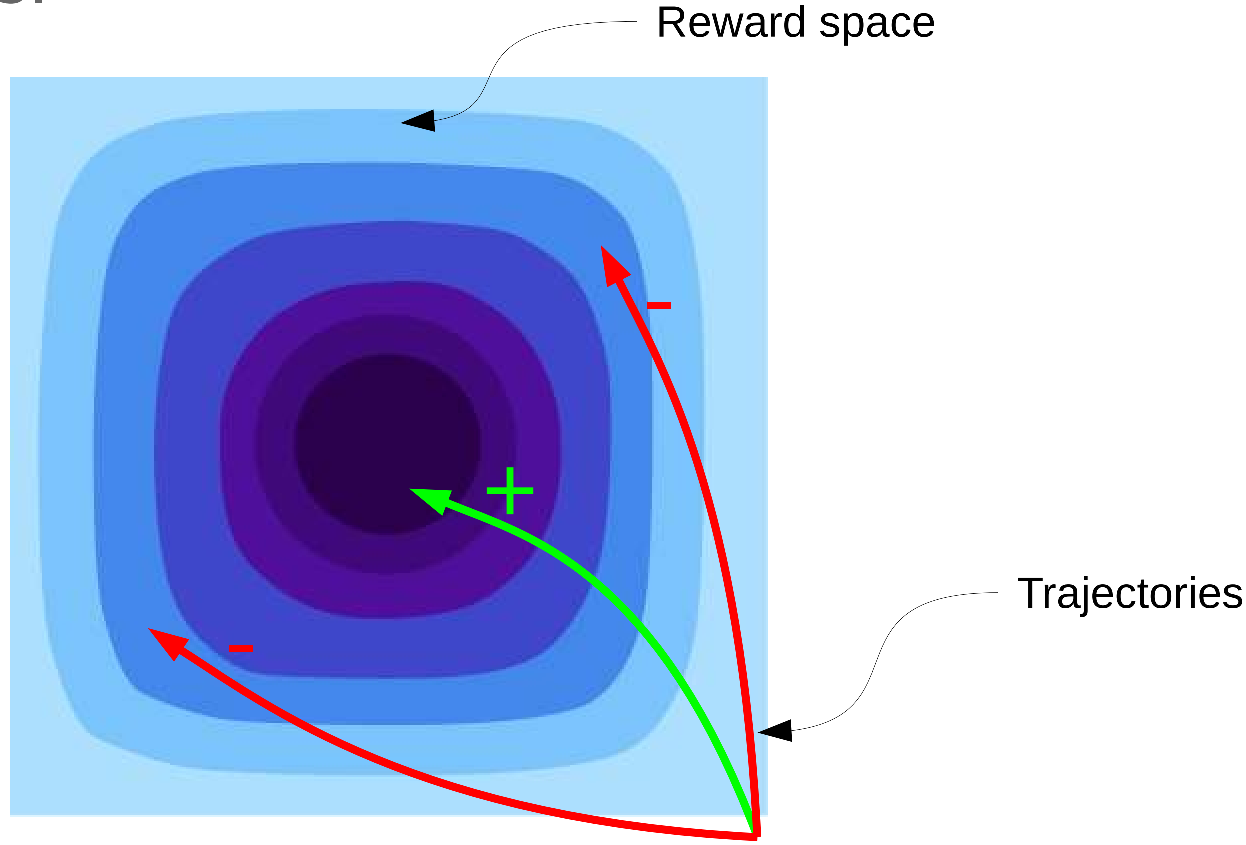
$$\pi = \text{softmax}(w^T s + b)$$

where w, b are model parameters and s is a state.

- A policy parameterized by w will generally be written, $\pi(a|s; w)$ and represents the probability distribution over actions given the parameters.

Policy Gradient Intuition

- Increase the probability of trajectories that give good returns.



Policy Gradient Overview

1) Sample data from policy,

$$D = \{s_1, a_1, r_1, s_2, a_2, r_2, \ldots, s_n, a_n, r_n\}$$

2) Compute the probability of the samples given the policy

3) Determine the “goodness” of sequence

4) Update the policy to increase the probability of good trajectories

The Policy Gradient In Detail

- Let τ be a trajectory under policy π with parameters w
- From the Markov assumption follows, that
$$P(\tau; w) = \prod_t p(s_t, a_t, s_{t+1})$$
is the probability of the trajectory and $R(\tau)$ is a measure of its "goodness."
- We can define an objective,
$$\mathcal{L} = E_\pi[R(\tau)] = \sum_\tau P(\tau; w)R(\tau)$$
and peform gradient ascent on this objective

The Policy Gradient In Detail

The objective is the expected return from the policy:

$$\mathcal{L} = \underbrace{E_\pi[R(\tau)]}_{\text{maximize expected return under } \pi}$$

$$= \sum_\tau P(\tau; w) R(\tau)$$

The Policy Gradient In Detail

Find the gradient of the policy w.r.t. its parameters,

$$\begin{aligned}
\nabla_w \mathcal{L} &= \nabla_w \sum_\tau P(\tau; w) R(\tau) \\
&= \sum_\tau \nabla_w P(\tau; w) R(\tau) \\
&= \sum_\tau \frac{P(\tau; w)}{P(\tau; w)} \nabla_w P(\tau; w) R(\tau) \\
&= \sum_\tau P(\tau; w) \frac{\nabla_w P(\tau; w)}{P(\tau; w)} R(\tau)
\end{aligned}$$

Interactive Robotics Lab

The Policy Gradient In Detail

$$\text{Now use log-derivative trick; } \frac{\nabla a}{a} = \frac{1}{a}\nabla a = \nabla \log a$$

$$\nabla_w \mathcal{L} = \sum_{\tau} P(\tau; w) \frac{\nabla_w P(\tau; w)}{P(\tau; w)} R(\tau)$$

$$= \sum_{\tau} P(\tau; w) \nabla_w \log P(\tau; w) R(\tau)$$

This is progress, but let’s examine the gradient term further.

The Policy Gradient In Detail

$$\nabla_w \mathcal{L} = \sum_\tau P(\tau; w) \nabla_w \log \left[\prod_{t=0}^{T} p(s_t, a_t, s_{t+1}) \pi(a_t \mid s_t; w) \right] R(\tau)$$

$$= \sum_\tau P(\tau; w) \nabla_w \left[\sum_{t=0}^{T} \log p(s_t, a_t, s_{t+1}) + \sum_{t=0}^{T} \pi(a_t \mid s_t; w) \right] R(\tau)$$

$$= \sum_\tau P(\tau; w) \nabla_w \sum_{t=0}^{T} \pi(a_t \mid s_t; w) \sum_{t=0}^{T} R(s_t)$$

$$= \sum_\tau P(\tau; w) \sum_{t=0}^{T} \nabla_w \pi(a_t \mid s_t; w) R(s_t)$$

Distribution of policy

Gradient direction to increase likelihood of $\pi(a_t | s_t; w)$

How good was visiting this state?

ASU

Interactive Robotics Lab

In more detail

$$\nabla_w \mathcal{L} = \sum_{\tau} P(\tau; w) \sum_{t=0}^{T} \nabla_w \pi(a_t \mid s_t; w) R(s_t)$$

$$= E_{\pi} \left[\sum_{t=0}^{T} \nabla_w \pi(a_t \mid s_t; w) R(s_t) \right]$$

Approximating the Gradient

Approximate this expected value by sampling from the policy,

$$\nabla_w \mathcal{L} = E_\pi \left[\sum_{t=0}^{T} \nabla_w \pi(a_t \mid s_t; w) R(s_t) \right]$$

$$\nabla_w \mathcal{L} \approx \hat{g} = \frac{1}{n} \sum_{i=0}^{n} \left[\sum_{t=0}^{T} \nabla_w \pi(a_t \mid s_t; w) R(s_t) \right]$$

Note there is no need to know the environment dynamics as $p(s_t, a_t, s_{t+1})$

drops out when taking the gradient because it doesn't depend on the parameters!

How should the “goodness” of a state be defined?

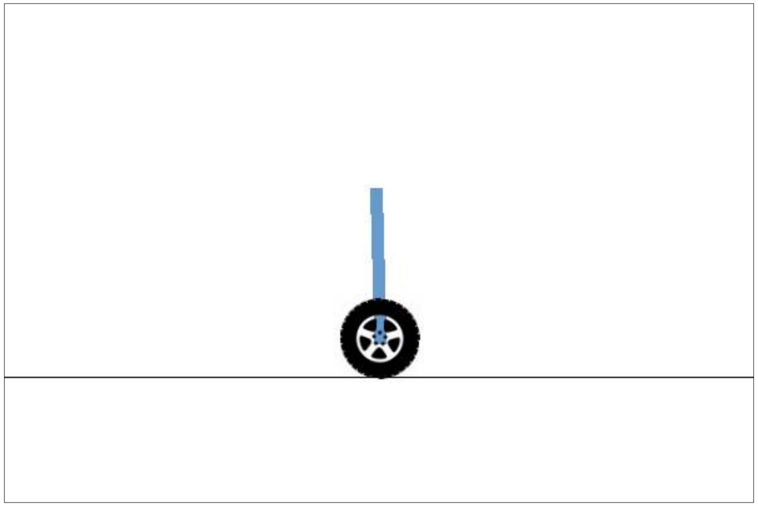
- Attempt 1: The simplest method is to simply use the discounted return received from that state.

$$R(s_t) = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots$$

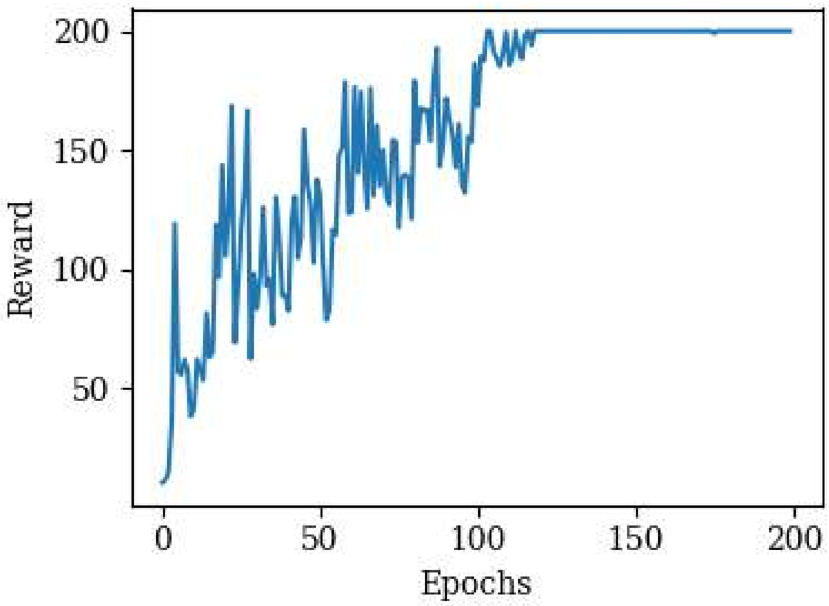
- Demo1: *Vanilla Policy Gradient*. This works! But not sample efficient.

REINFORCE, Williams ‘9

Vanilla Policy Gradient



Cartpole Task



Interactive Robotics Lab

Simple Policy Gradient in Practice: Defining the Policy

```
class Policy(nn.Module):

    def __init__(self):
        super(Policy, self).__init__()
        self.fc1 = nn.Linear(input_dim, hidden_dim)
        self.fc2 = nn.Linear(hidden_dim, n_actions)

    def forward(self, state):
        x = F.tanh(self.fc1(state))
        x = self.fc2(x)
        return x

    def act(state):
        x = self(state)
        probs = F.softmax(x, dim=1)
        log_probs = F.log_softmax(x, dim=1)
        # sample from probs, keep track of log prob
        a = probs.multinomial()
        log_prob_a_s = log_probs[a]
        # return action and its log probability given the policy
        return a, log_prob_a_s
```

Simple Policy Gradient in Practice: Computing the Gradient

```
def compute_policy_gradient(actions, logps, rewards, gamma=0.99):
    # compute vanilla policy gradient
    # assume given actions, log probabilities logps, rewards
    # gamma is the discount factor
    # first compute returns
    returns = [0]*len(actions)
    returns[-1] = rewards[-1]
    for i in reversed(range(len(actions))-1):
        returns[i] = rewards[i] + gamma * returns[i+1]

    # now compute gradient
    grad = torch.mean([logps[i] * returns[i] for i in range(len(actions))])

    # in PyTorch var.backward() does backpropagation
    grad.backward()
```


REINFORCE, Williams ‘92

How should the "goodness" of a state be defined?

- Attempt 2: Reduce variance by learning a value estimate, and use advantage estimate.

$$R(s_t, a_t) = A(s_t, a_t) = Q(s_t, a_t) - V(s_t)$$

The value function $V(s_t)$ estimates the expected discounted return from state s while acting under policy π

$$V(s) = E_\pi[R(s)]$$

The advantage function $A(s, a)$ estimates the difference between executing action a in state s, the Q -function, and behaving under policy π.

$Q(s, a)$ is estimated from the trajectory rollout.

How should the "goodness" of a state be defined?

- Estimate $Q(s, a)$ from the trajectory rollout. Which is better?

$$Q(s_t, a_t) = \underbrace{r_t + V(s_{t+1}) - V(s_t)}_{\text{low variance; high bias}}$$

$$Q(s_t, a_t) = \underbrace{r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots + V(s_{t+n}) - V(s_t)}_{\text{high variance; low bias}}$$

- Use generalized advantage estimate that interpolate between 1-step and infinite-step returns.

Generalized Advantage Estimation, J. Schulman 2015

Generalized Advantage Estimation

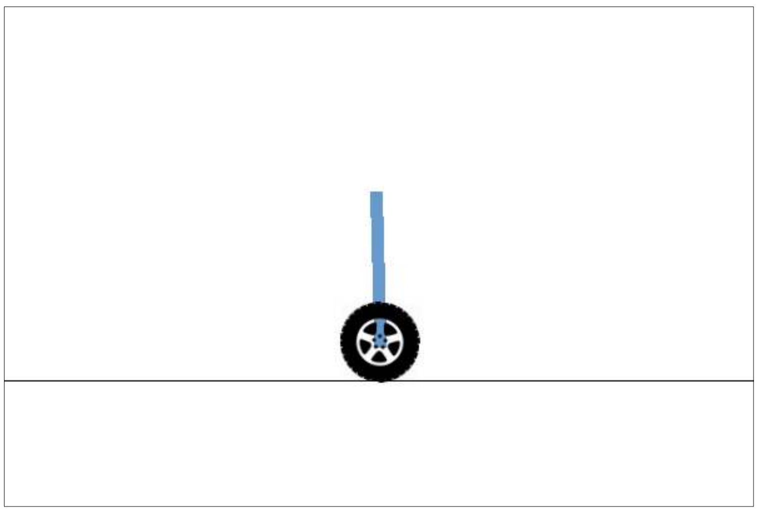
- $A(s_t, a_t)^{\mathrm{GAE}}$ is exponentially weighted average of A^1 through A^∞.

- $A(s_t, a_t) = \sum_{n=0}^{\infty} (\gamma\lambda)^n \delta_{t+n}$ where γ, λ are the discount and exponential GAE parameters, respectively.

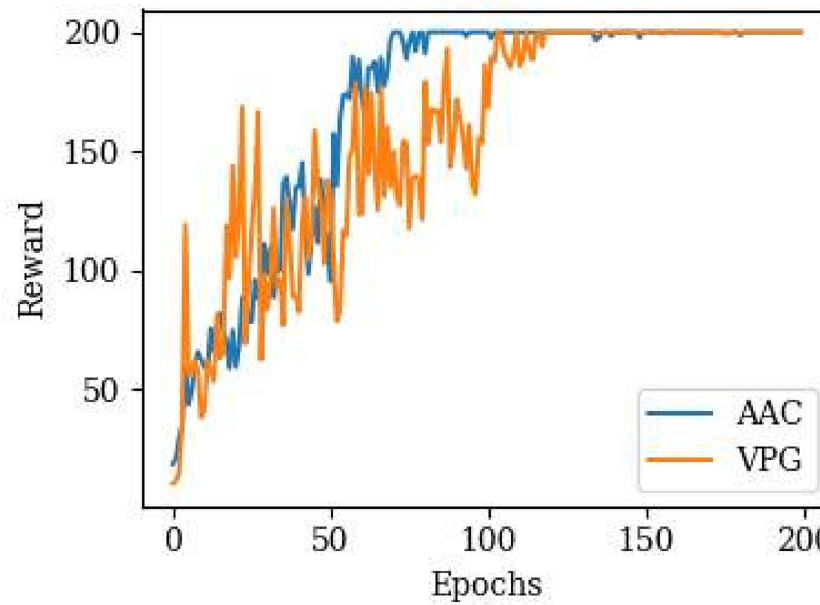
Generalized Advantage Estimation, J. Schulman 2015

Advantage Actor Critic with GAE

- Demo 2: *Advantage Actor Critic*. This is better! But still not particularly sample efficient.



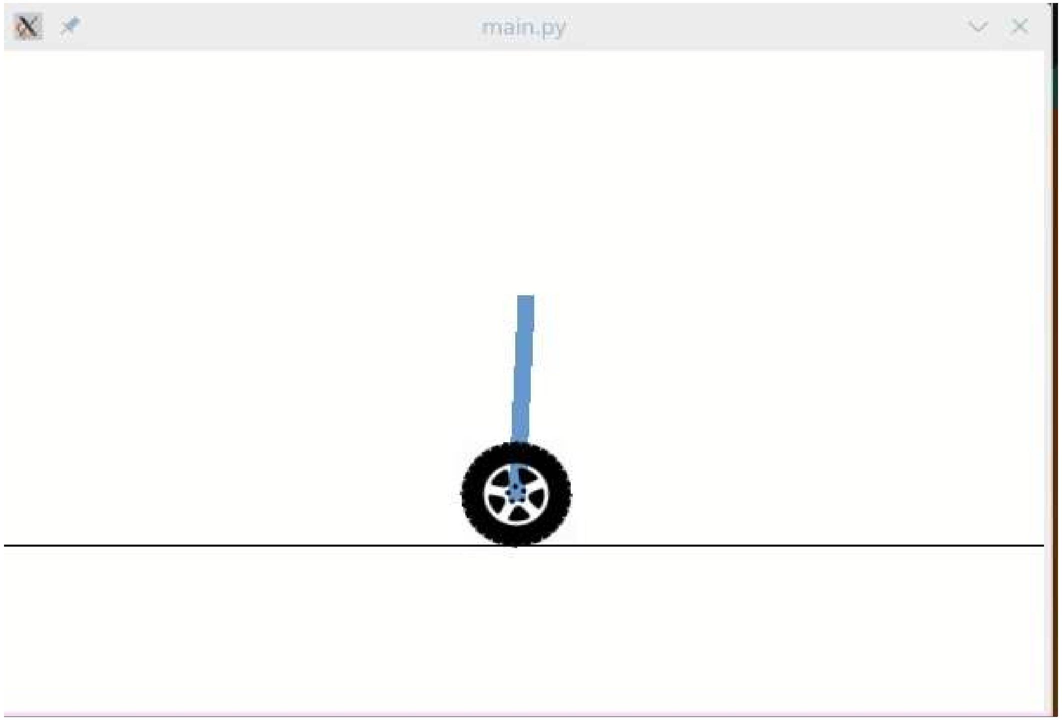
Cartpole Task



Interactive Robotics Lab

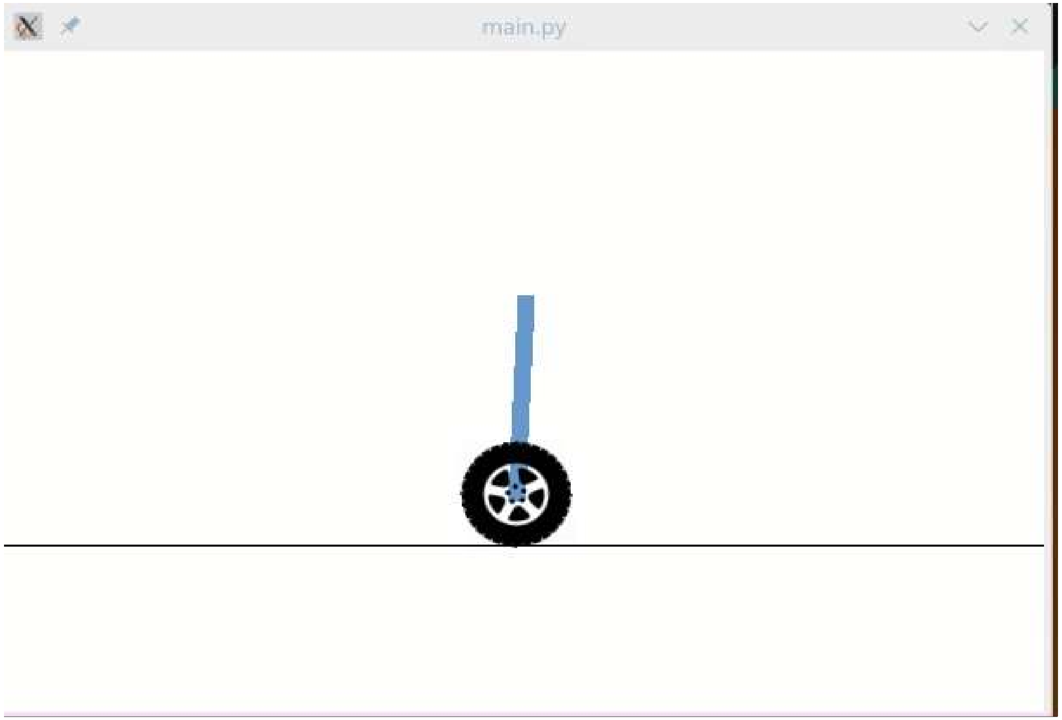
Asynchronous Methods for Deep RL, V. Mnih 2016

Random Policy



Play Video

Advantage Actor Critic Demo



Play Video

Interactive Robotics Lab

Advantage Actor Critic in Practice: Now we need a value estimate

```
class ValueFunction(nn.Module):

    def __init__(self):
        super(ValueFunction, self).__init__()
        self.fc1 = nn.Linear(input_dim, hidden_dim)
        self.fc2 = nn.Linear(hidden_dim, 1)

    def forward(self, state):
        x = F.tanh(self.fc1(state))
        v = self.fc2(x)
        return v

def fit_value_function(vfn, loss_fn, states, returns):
    # given value estimate and inputs / targets
    # loss_fn can be nn.MSELoss for example
    # fit by regression techniques
    loss = 0
    for i in range(len(states)):
        loss += loss_fn(vfn(states[i]), returns[i])
    loss.backward()
```

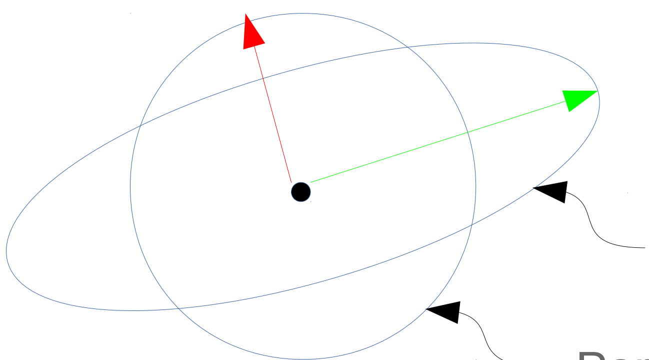
Advantage Actor Critic in Practice

```
def compute_generalized_adv_estim(rewards, values, gamma=0.99, tau=0.95):
    gae = [0]*len(rewards)
    return = values[-1] # initialize to last value
    for I in reversed(range(len(rewards)-1)):
        return = rewards[i] + gamma * return
        delta = -values[i] + return
        gae[i] = gamma * tau * delta

def compute_policy_gradient(states, actions, rewards):
    values = value_fn(states)
    gae = compute_generalized_adv_estim(rewards, values)
    # compute advantage actor critic gradient
    grad = torch.mean([logps[i] * gae[i] for i in range(len(actions))])

    # in PyTorch var.backward() does backpropagation
    grad.backward()
```

Interactive Robotics Lab

Can we still do better?

- RL approaches tend to be unstable
  - Problem: One poor update may lead to divergence
  - Fix: improve gradient and bound the update step
  - Examples: Natural Gradient / Trust Region Methods
- Intuition:



Natural Policy Gradient

- Transform gradient such that movement in parameter space is adjusted for desired movement in probability space
- Use Fisher matrix to approximate local changes in probability distribution given the current parameters.
- Again, estimate by sampling:

$$F = \frac{1}{N}\sum_{i=0}^{N}[\nabla \log \pi(a_i \mid s_i)(\nabla \log \pi(a_i \mid s_i))^T]$$

- Transform gradient by $\hat{g} = F^{-1}\hat{g}$

Natural Policy Gradient, Kakade 2002
Natural Actor Critic, Peters 2008
Trust Region Policy Opt, Schulman 2015
Actor Critic using Kronecker-Factored Trust Region, Wu 2017

What about exploration?

- In value-based methods *off-policy* ϵ-greedy exploration is common.
- PG methods rely on *on-policy* sampling from the distribution $\pi(s|a)$

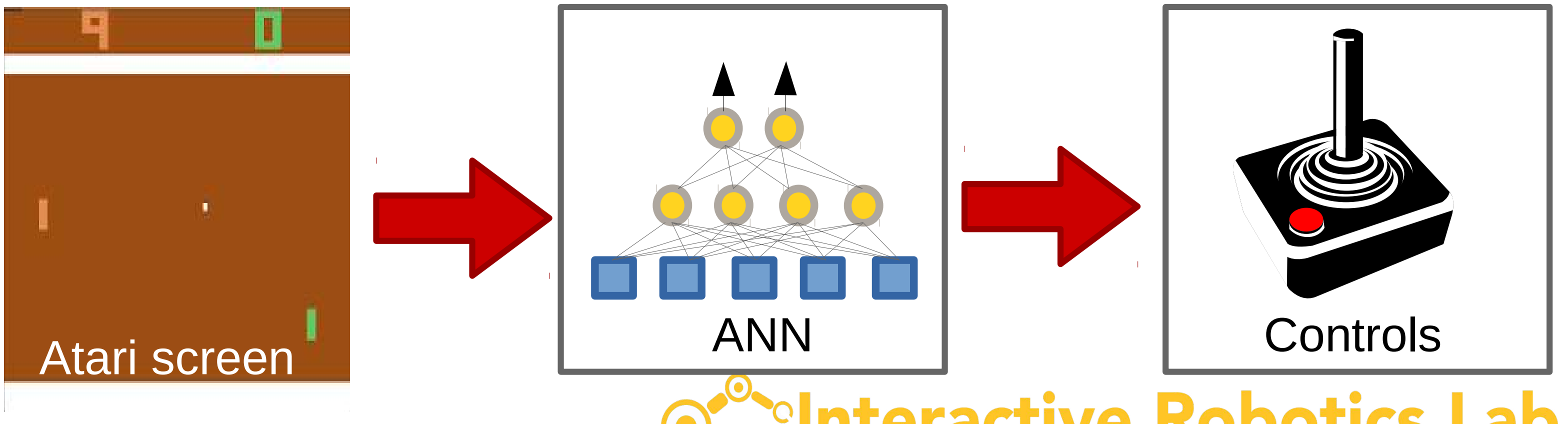
Example Application

- Python code implementing the above examples can be found in folder “Reinforcement”
- The README includes instructions on learning and testing a model

End-To-End Reinforcement Learning

End-to-end Learning

- Some tasks are naturally formed as a mapping from an image to an action
- For example: Atari games or many robotics tasks
- Input = image → output = controls
- Use a neural network to process input image
- ANN **infers state variables** from image



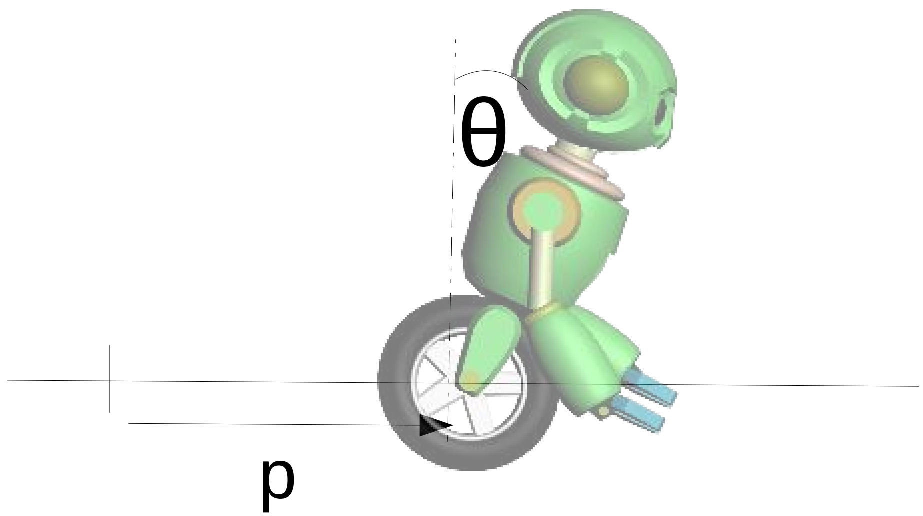
Interactive Robotics Lab

Back to Balancing

State of the System $\mathbf{s}$

Action to execute $\mathbf{a}$

Policy linking states to actions $\mathbf{a} = \pi(\mathbf{x})$



State:
$\mathbf{s} = \mathbb{R}^{X \times Y}$ (Image)

Action:
$\mathbf{a}$ = Force

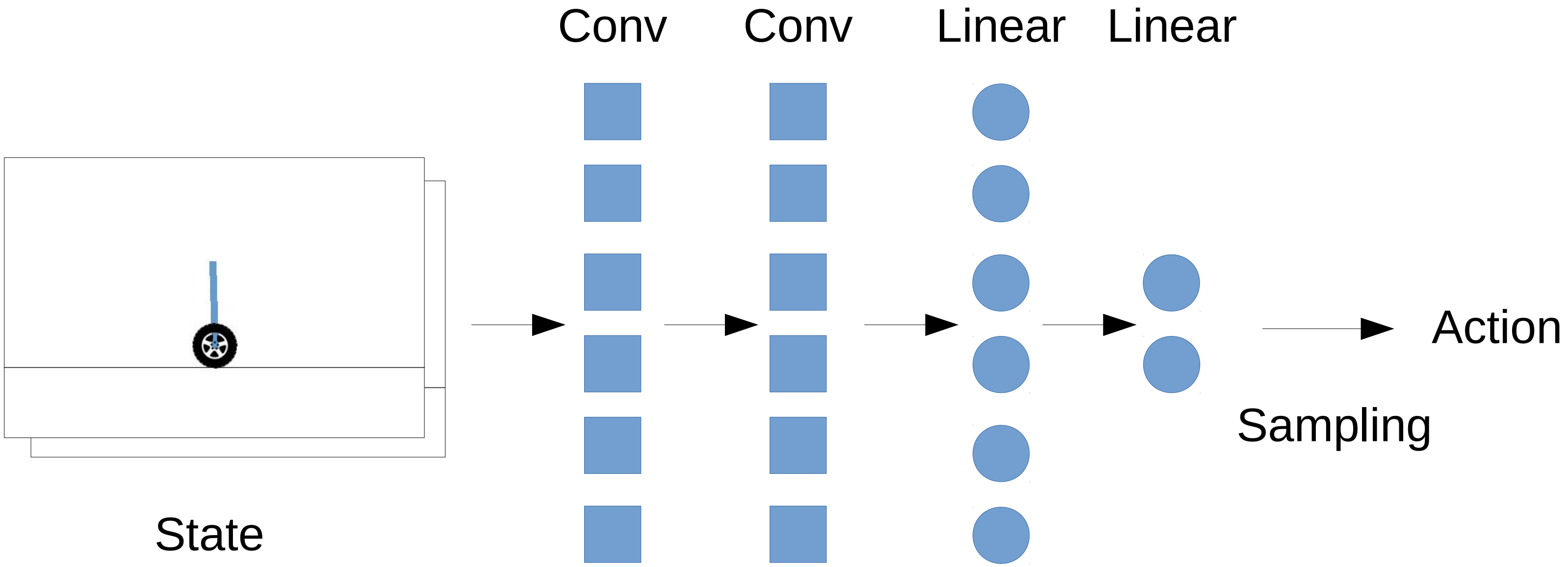
Policy:
$\mathbf{a} = \pi(\mathbf{x})$

Deep Reinforcement Learning for E2E

- The code is the same! Just change the model and the inputs.
- Computes derivatives for all weights in the model by backpropagation.
- Preserve the Markov property: often stack multiple frames in order to infer velocities.

Deep RL in Practice

- Solving the cart pole task from images:



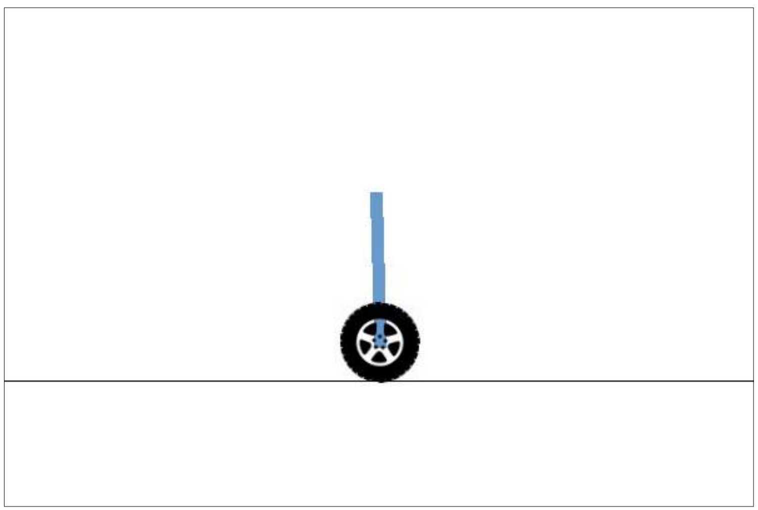
Deep RL in Practice: Defining the Policy

```
class Policy(nn.Module):
    def __init__(self):
        super(Policy, self).__init__()
        self.conv1 = nn.Conv2d(3, 16, kernel_size=5, stride=2)
        self.conv2 = nn.Conv2d(16, 32, kernel_size=5, stride=2)
        self.conv3 = nn.Conv2d(32, 32, kernel_size=5, stride=2)
        self.head_a = nn.Linear(448, 2)
        self.head_v = nn.Linear(448, 1)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = F.relu(self.conv2(x))
        x = F.relu(self.conv3(x))
        v = self.head_v(x.view(x.size(0), -1))
        x = self.head_a(x.view(x.size(0), -1))
        return x, v

    def act(state):
        x, v = self(state)
        probs = F.softmax(x, dim=1)
        log_probs = F.log_softmax(x, dim=1)
        # sample from probs, keep track of log prob
        a = probs.multinomial()
        log_prob_a_s = log_probs[a]
        # return action and its log probability given the policy
        return a, v, log_prob_a_s
```

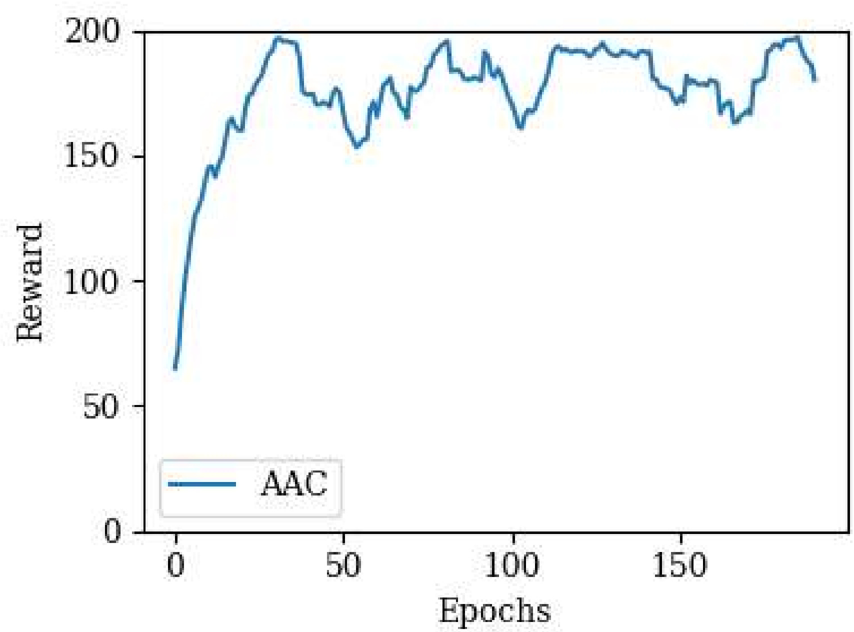
Three convolutions and a linear layer for actions and values

Deep Advantage Actor Critic

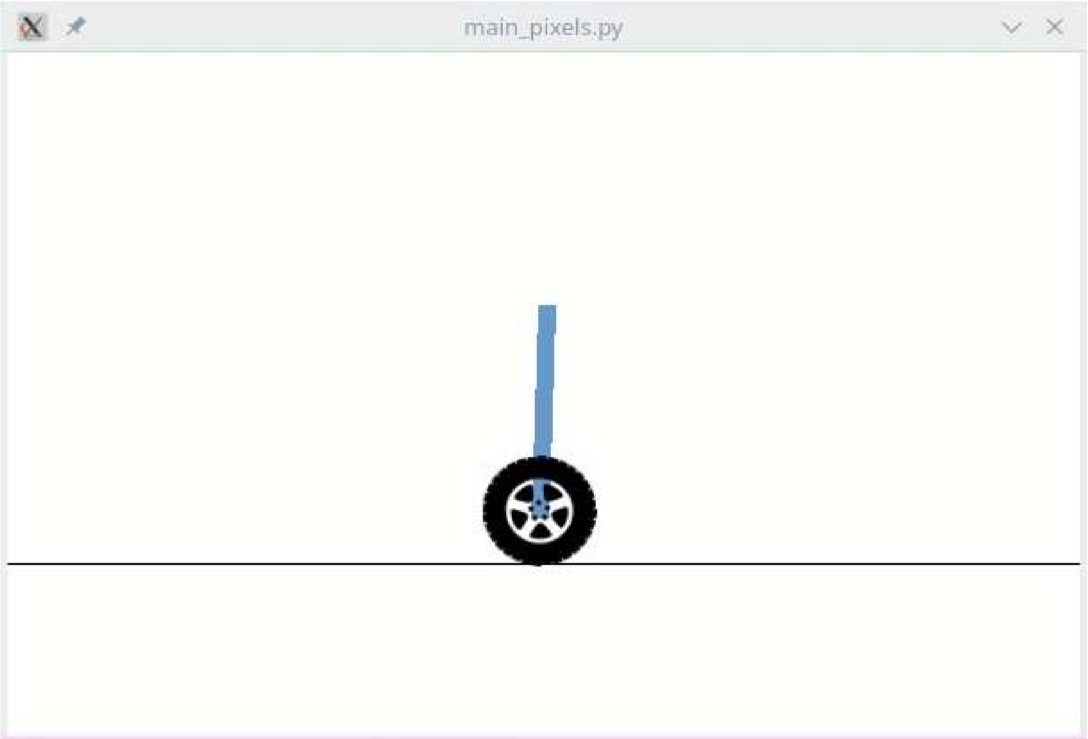
- Demo 3: *Advantage Actor Critic with neural network function approximator*.



Cartpole Task



Advantage Actor Critic Demo



Play Video

ASU

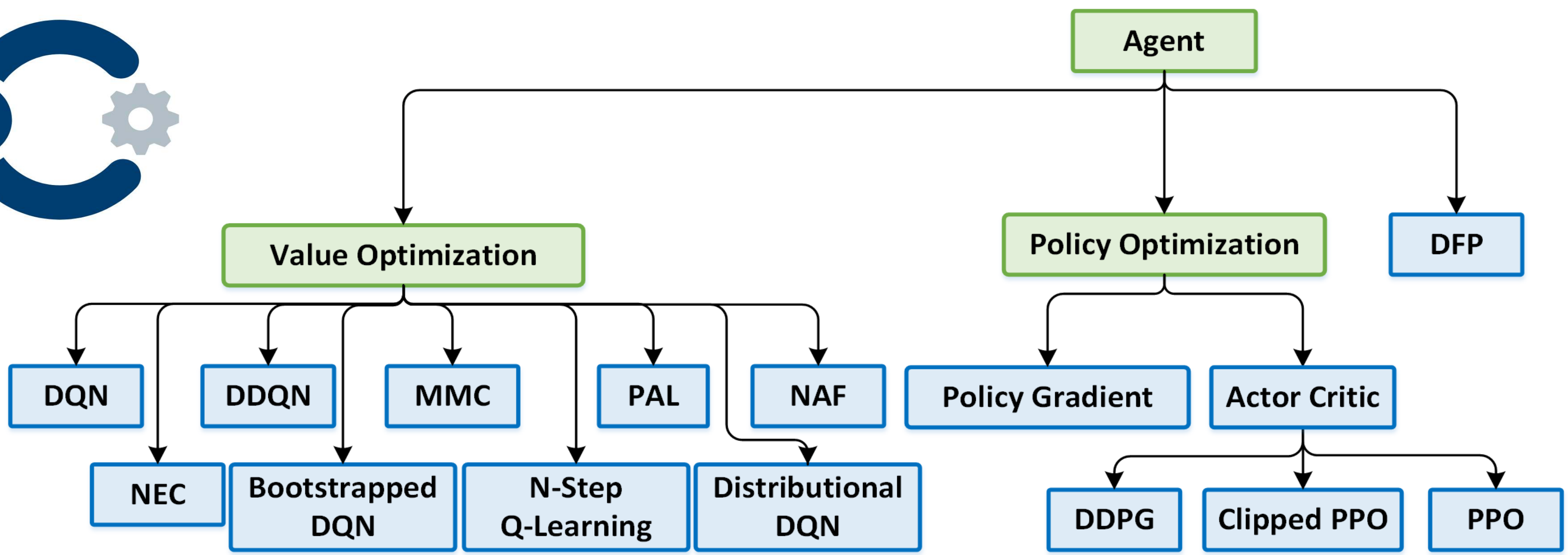
Interactive Robotics Lab

Example Application

- Python code implementing the above examples can be found in folder “Reinforcement”
- The README includes instructions on learning and testing a model

Next Steps: Intel RL Coach

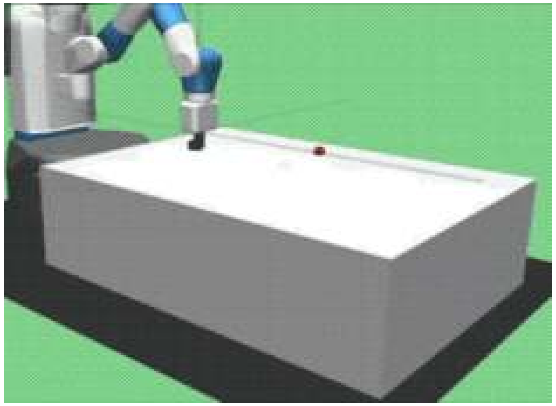
- Intel Reinforcement Learning Coach
- Implements a variety of state-of-the-art methods
- Uses the processing power of multi-core CPUs to enables efficient training of RL agents.

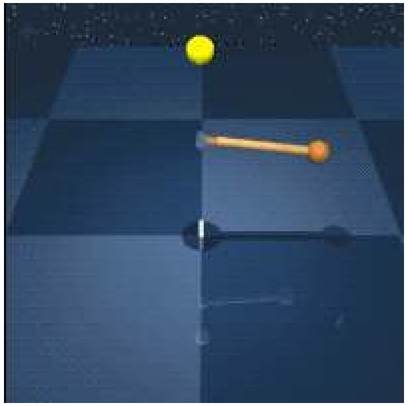


Intel Reinforcement Learning Coach

- Provides a range of benchmark scenarios
- Uses Intel-optimized TensorFlow for computations
- Humanoid control, autonomous driving, StarCraft

Download



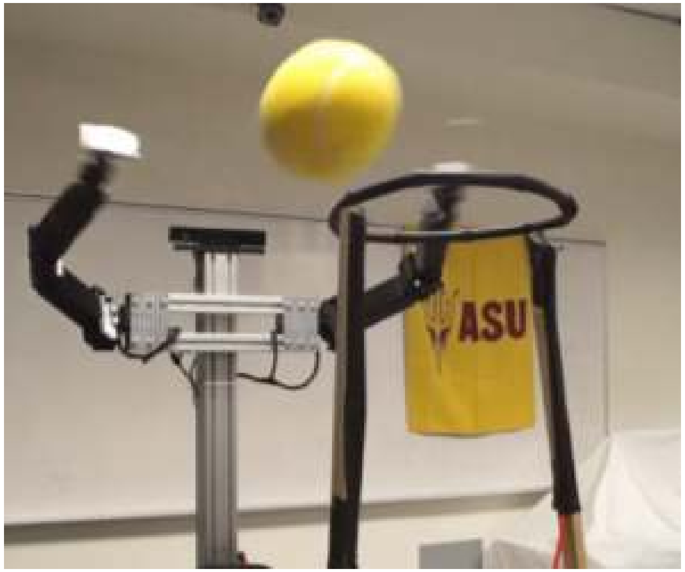


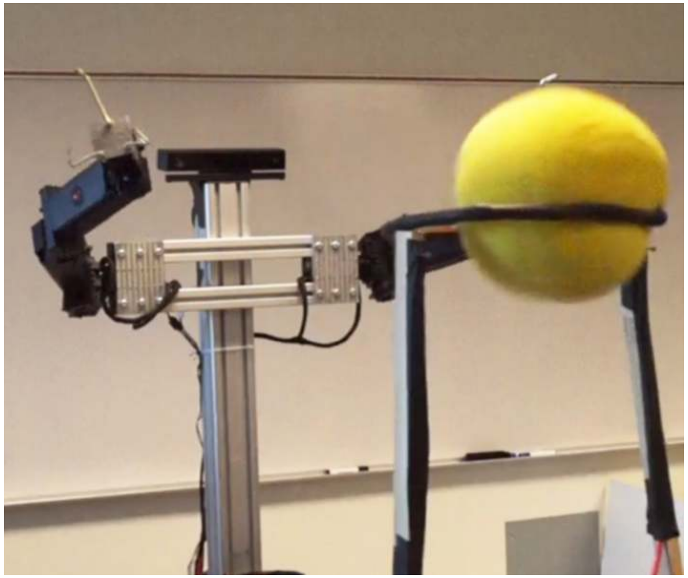
Recent Successes of Policy Gradients and Trends

- Atari / Doom (A3C)
- Simulated Robotic Control (TRPO, NPG, ACKTR, DDPG)
- Combining on- and off-policy learning
  - Off policy tends to be more sample efficient but less stable. How to combine both?
- Exploration strategies
  - Can we do better than Gaussian noise?
    - Intrinsic motivation approaches
    - Parameter space exploration

Robot Basketball with RL

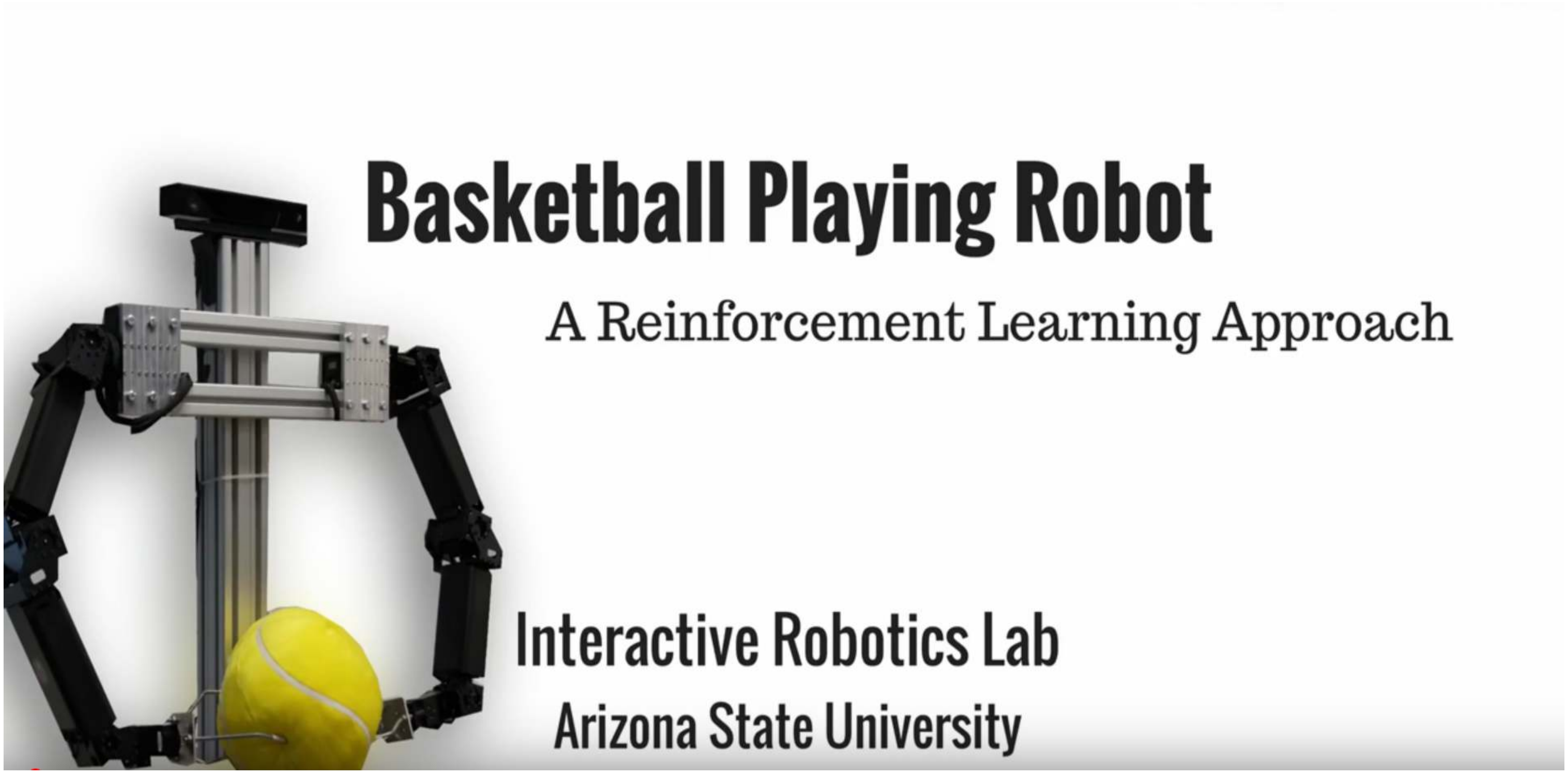
- Task: learn to get ball through hoop
- Reward function: distance to center of hoop
- Reinforcement learning on bimanual robot





ASU Interactive Robotics Lab

Video: Robot Basketball with RL



Play Video

Summary

- We introduced reinforcement learning
- Learning robot control trough trial and error
- Deep networks represent the controller
- We can even go directly from visuals to contols
- End-to-end learning

Interactive Robotics Lab

The development of this course was supported by an Intel AI Academy grant. We thank the sponsor for the continuing support of open-source efforts in research and education.

Interactive Robotics Lab