COL864: Special Topics in AI
Semester II, 2021-22

Reinforcement Learning III:
Deep Q-Learning and Policy Gradients

Rohan Paul

Outline

- Last Class
  - Model-Free Reinforcement Learning
- This Class
  - Deep Q-Learning
  - Policy Learning
- Reference Material
  - Please follow the notes as the primary reference on this topic.

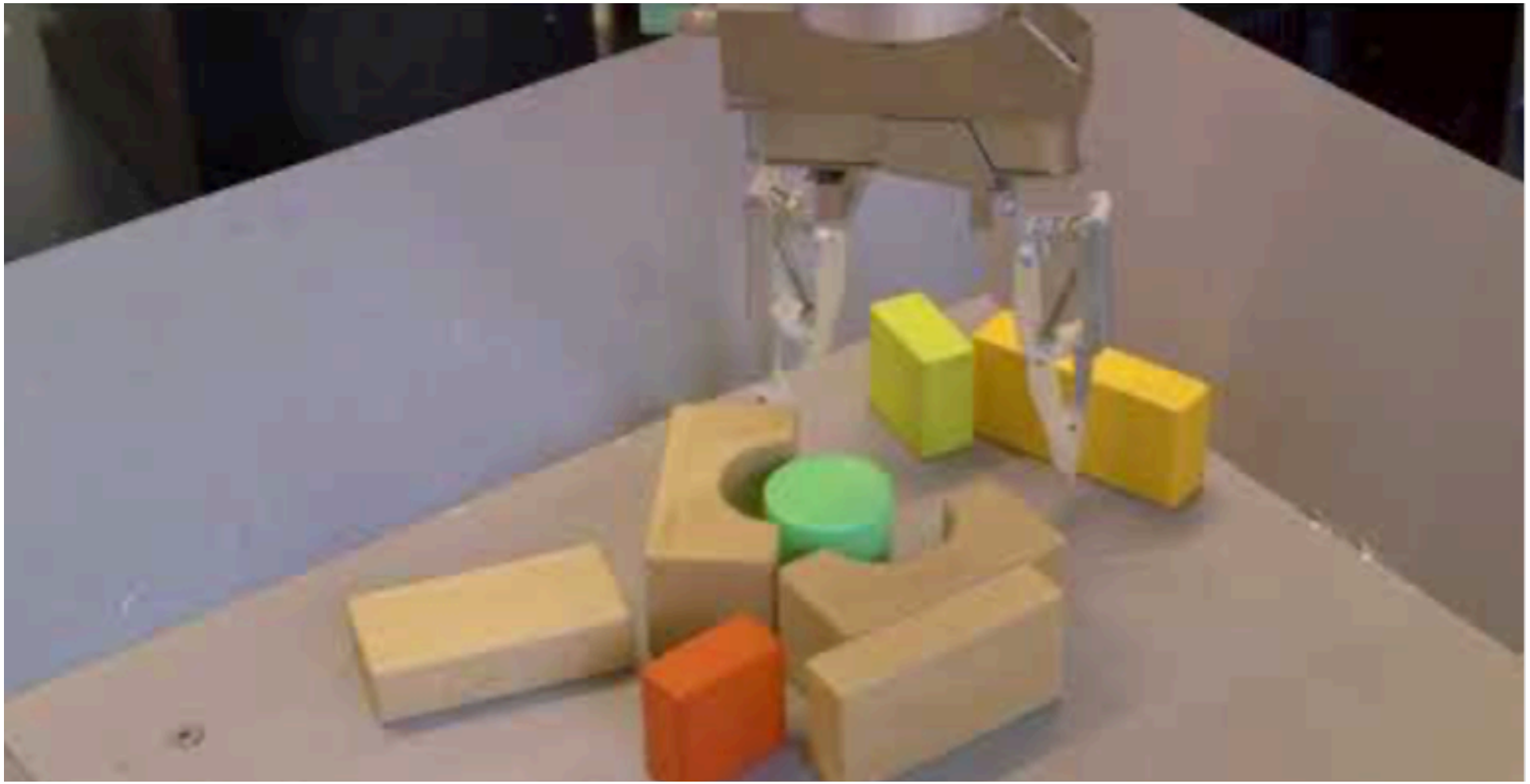
Acknowledgements

These slides are intended for teaching purposes only. Some material has been used/adapted from web sources and from slides by Nicholas Roy, Wolfram Burgard, Dieter Fox, Sebastian Thrun, Siddharth Srinivasa, Dan Klein, Pieter Abbeel, Max Likhachev, Alexander Amini (MIT Introduction to Deep Learning) and others.

Learning to Control

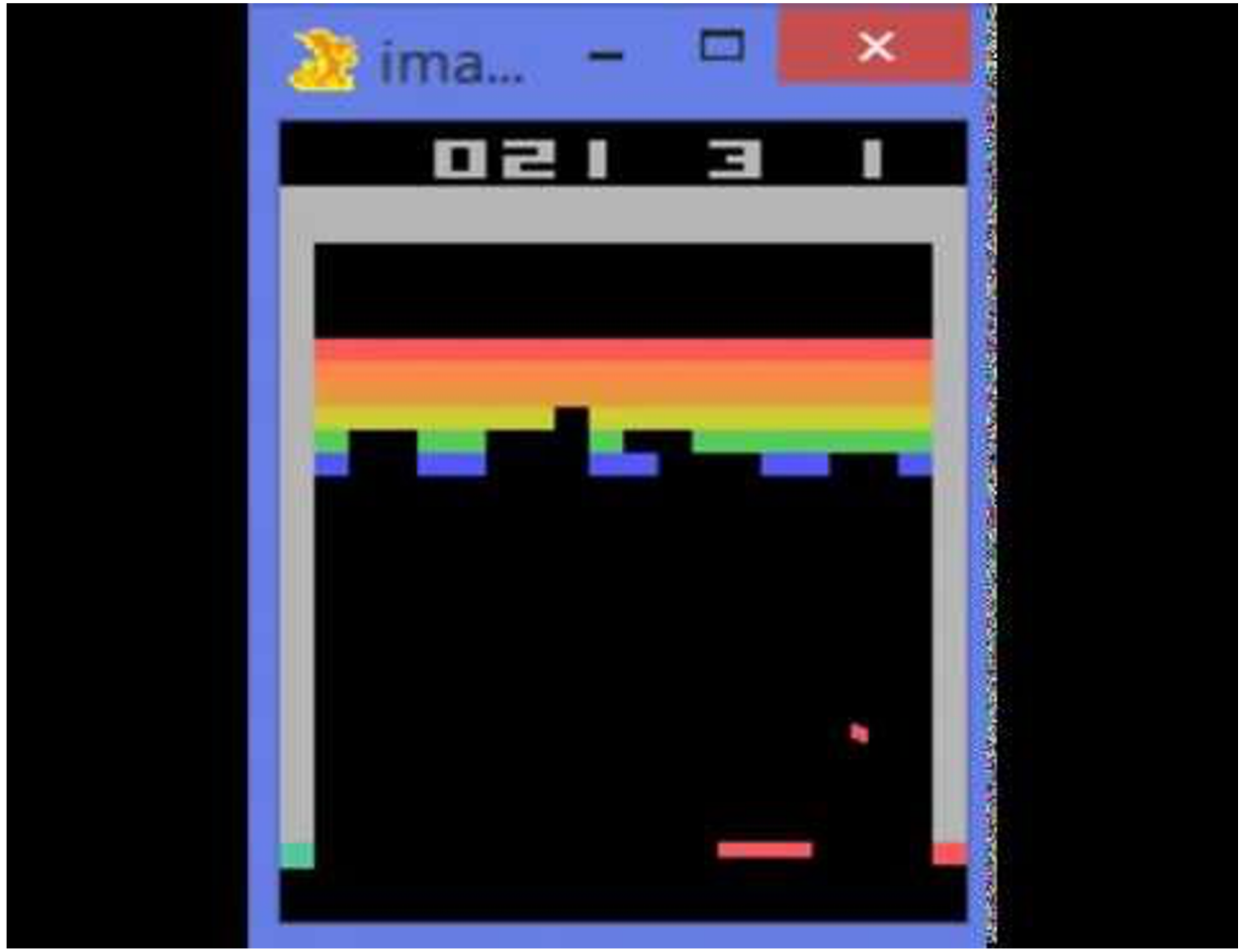
Variational End-to-End Navigation and Localization. Alexander Amini, Guy Rosman, Sertac Karaman, Daniela Rus

Robot Grasping



QT-Opt: Scalable Deep Reinforcement Learning for Vision-Based Robotic Manipulation

Learning to Control: AI Game Play



https://www.youtube.com/watch?v=V1eYniJ0Rnk

Q-Function and Policies

$$Q(s_t, a_t) = \mathbb{E}[R_t | s_t, a_t]$$

(state, action)

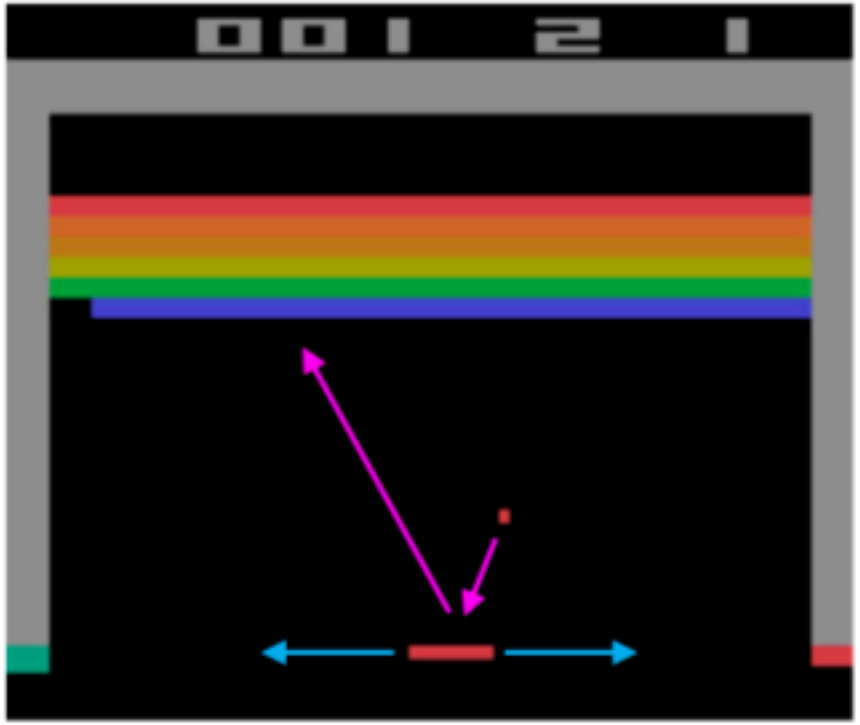
Ultimately, the agent needs a **policy $\pi(s)$,** to infer the **best action to take** at its state, s

Strategy: the policy should choose an action that maximizes future reward

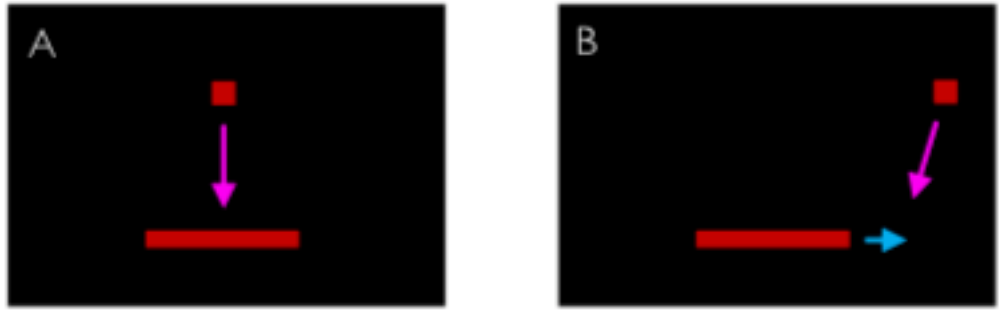
$$\pi^*(s) = \underset{a}{\operatorname{argmax}}\, Q(s, a)$$

The Q-Function

Example: Atari Breakout



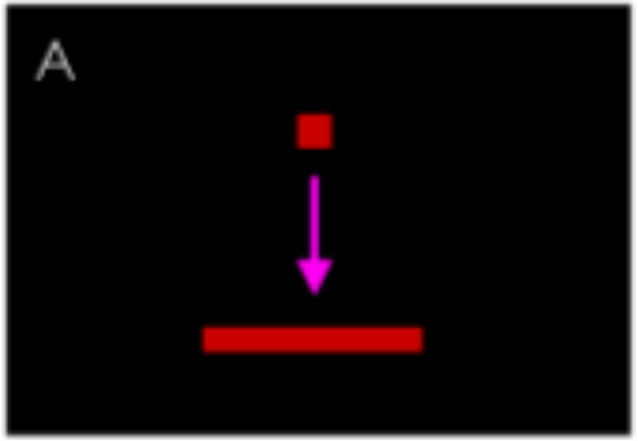
It can be very difficult for humans to accurately estimate Q-values



Which $(\mathbf{s}, \mathbf{a})$ pair has a higher Q-value?

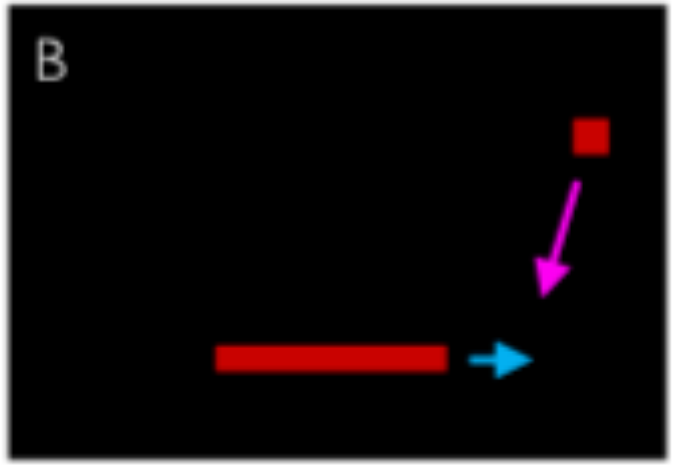
The Q-Function

Example: Atari Breakout - Middle



The Q-Function

Example: Atari Breakout - Side



Its not easy to specify Q(s, a). Can we learn the Q function from data.

Issue of Generalization

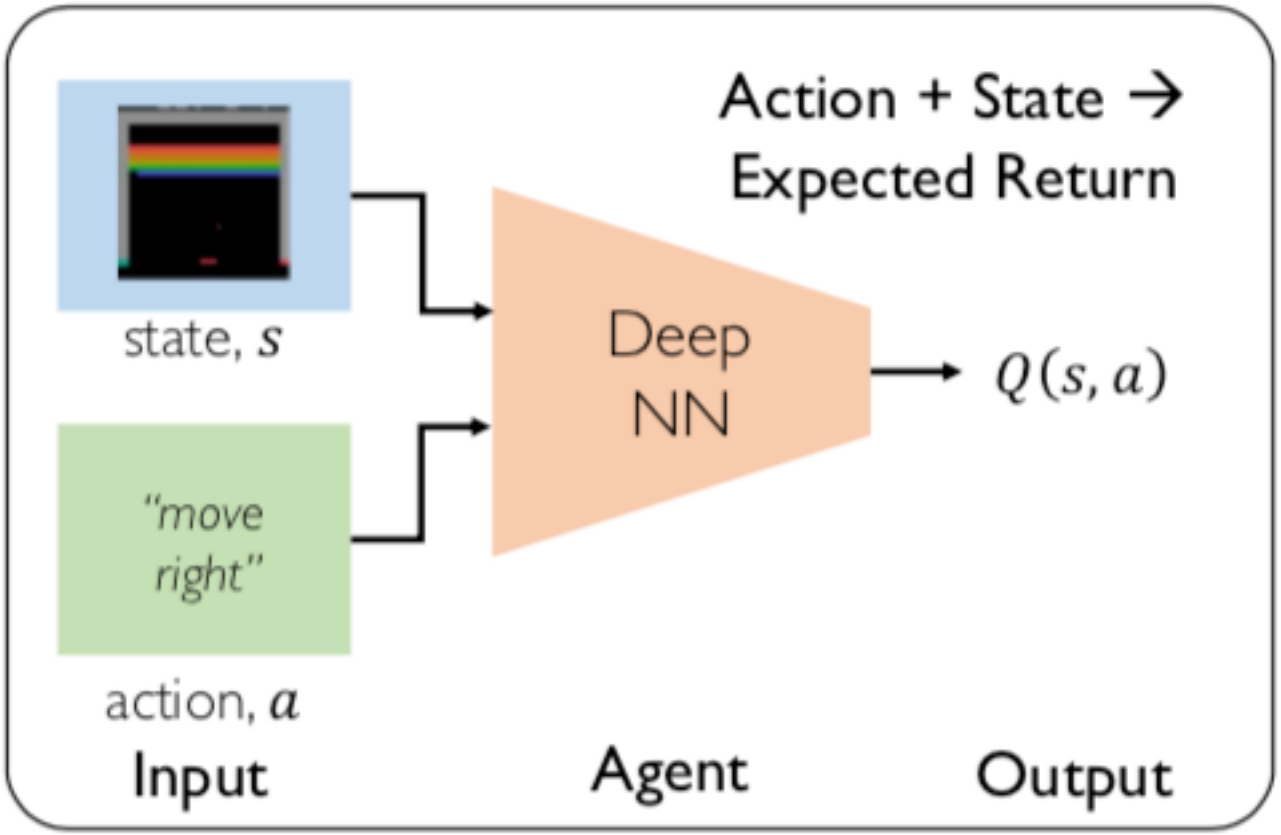
- Till now
  - States were explicitly represented in a tabular format.
- Generalization
  - Feature representation of states.
  - States that share features would share other properties as well.
  - Each state does not have to be dealt with independently.
  - Represent the Q or the V function as a linear combination of features.

Example: Atari Breakout - Side

Example: Atari Breakout - Middle

Deep Q-Networks (DQNs)

Modeling the Q-function via a Neural Network.



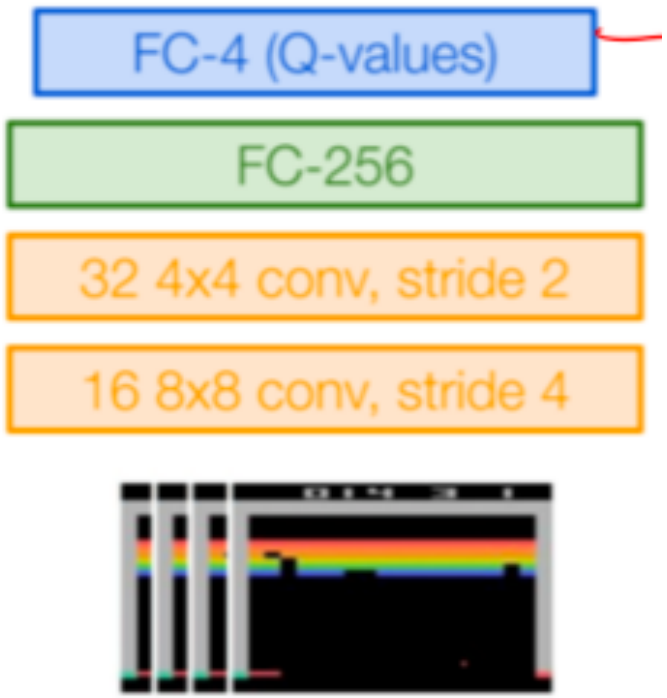
How to Encode an MDP as a NN input?

- Encoding the state-action pair
  - One network for each action a_j, that takes s as input and produces $Q(s,a_j)$ as output
  - One single network that takes s as input and produces a vector $Q(s, \cdot)$, consisting of the Q values for each action
  - One single network that takes s, a concatenated into a vector (if a is discrete, we would probably use a one-hot encoding
- How to decide which encoding to use?
  - First two network structures are only suitable for discrete (and not too big) action sets.
  - The last structure can be applied for continuous actions, but then it is difficult to find $\text{argmax}_A\ Q(s, a)$.
  - Driven by experience

Deep Q-Networks (DQNs)

Output: Q(s, left), Q(s, right), …..

CNNs extensively used in computer vision
If we want to go from pixels to decisions, likely useful to leverage insights for visual input

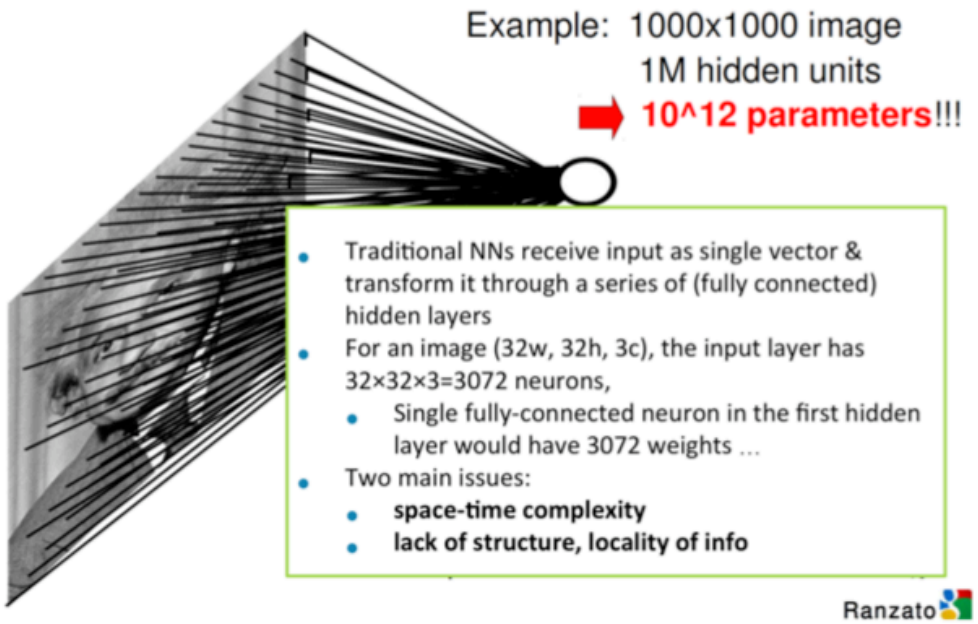


Current state s_t: 84*84*4 stack of last four frames. After RGB-> grayscale conversion, downsampling and cropping.

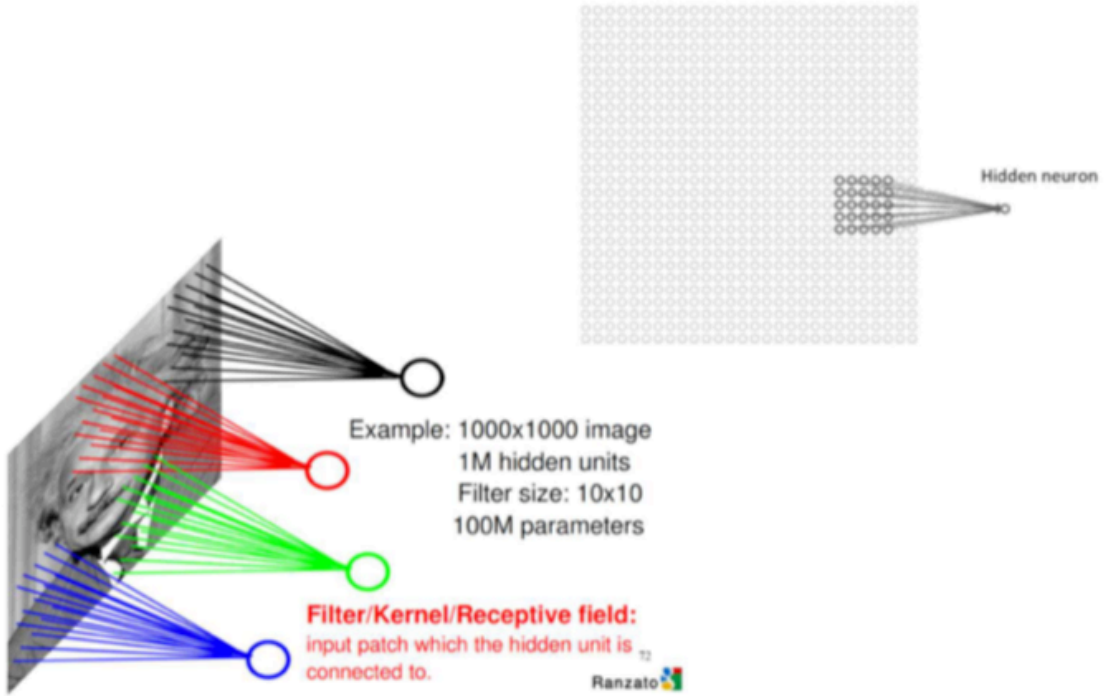
Minh, Nature 2015

Locality of Information: Receptive Fields

Fully connected network.

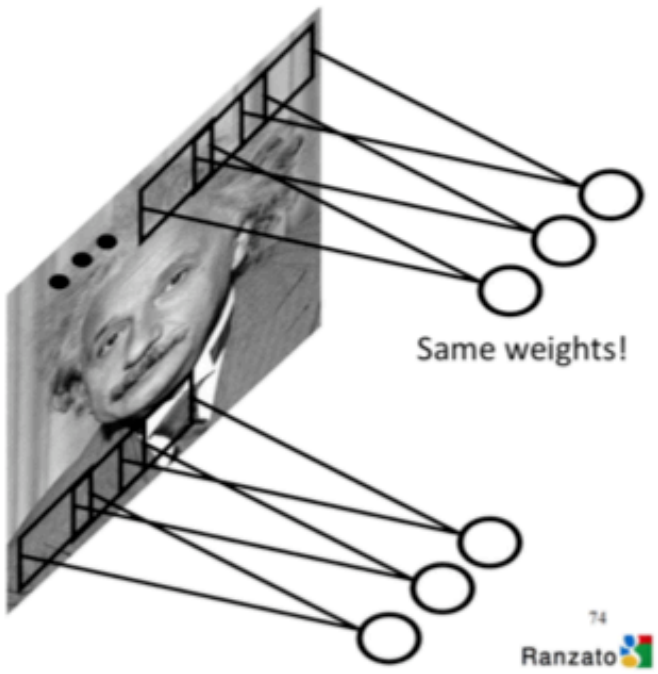


Convolutional NN

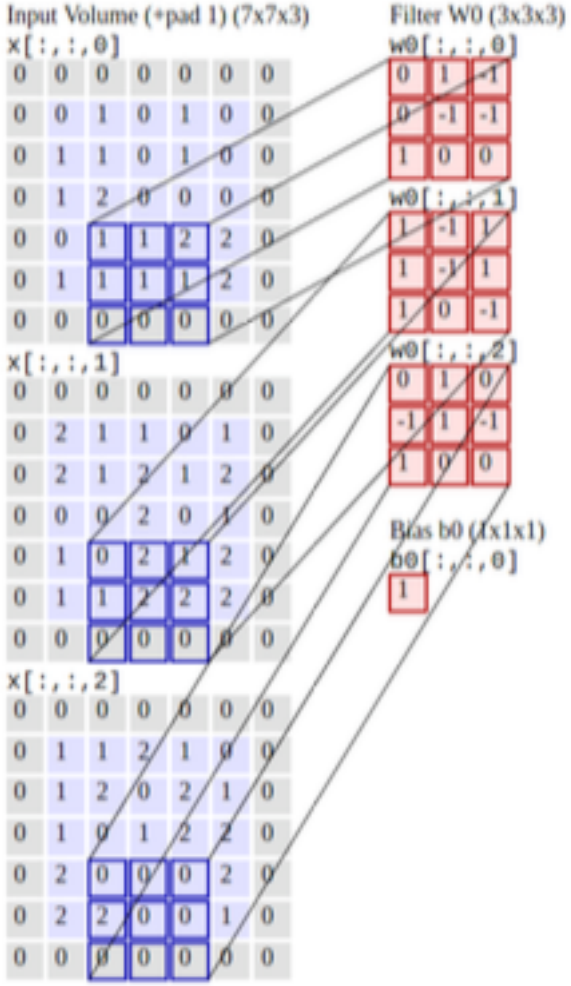


Convolutional NN

Feature Maps



- The map from the input layer to the hidden layer is therefore a feature map: all nodes detect the same feature in different parts
- The map is defined by the shared weights and bias
- The shared map is the result of the application of a convolutional filter (defined by weights and bias), also known as convolution with learned kernels



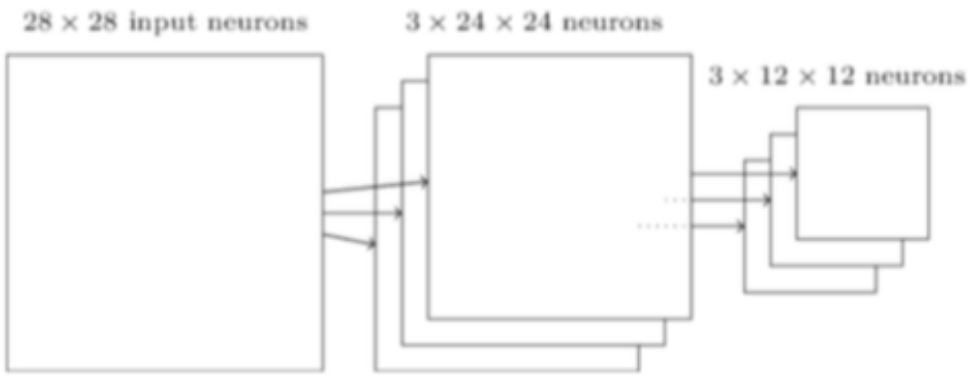
Convolutional NN

Pooling Layers

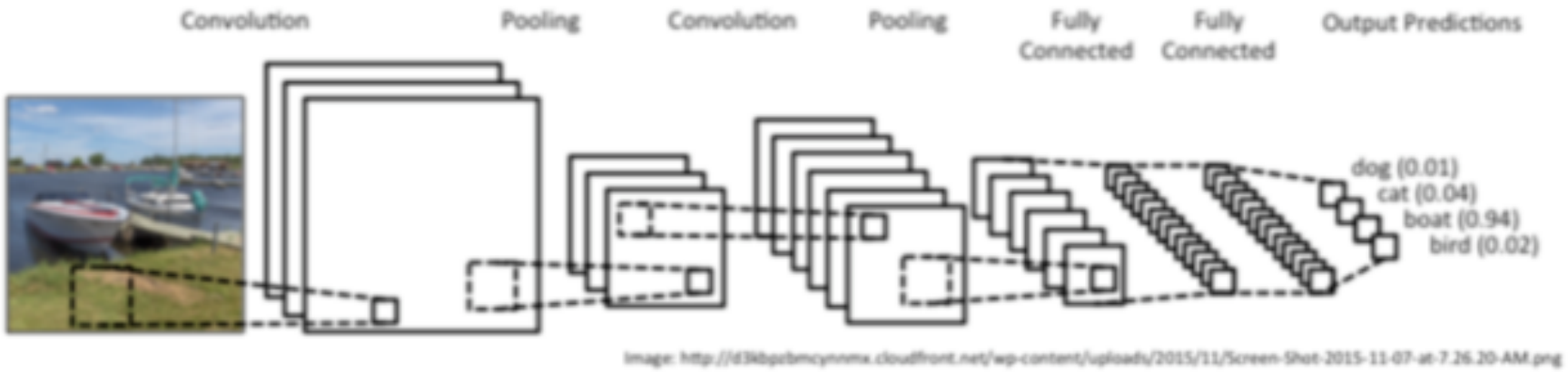
Pooling layers are usually used immediately after convolutional layers.

Pooling layers simplify / subsample / compress the information in the output from convolutional layer

A pooling layer takes each feature map output from the convolutional layer and prepares a condensed feature map

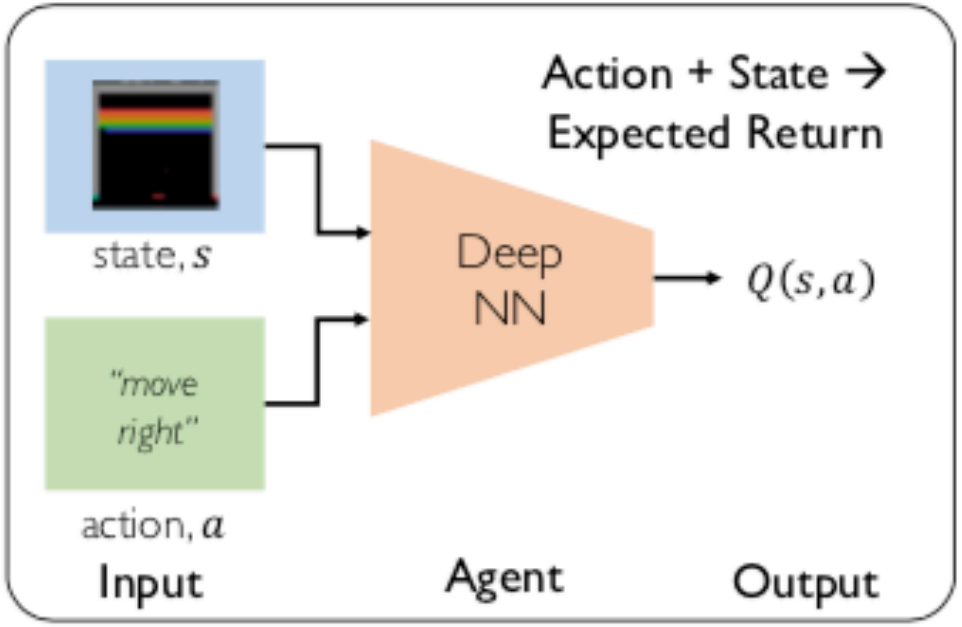


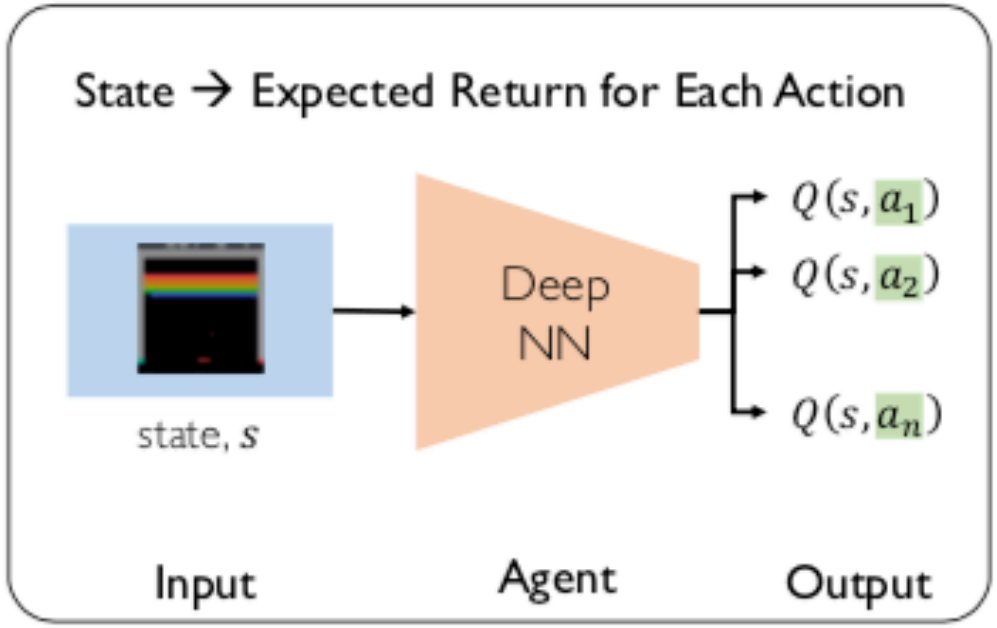
Convolution NN



- Consider local structure and common extraction of features
- Not fully connected. Locality of processing
- Weight sharing for parameter reduction
- Learn the parameters of multiple convolutional filter banks
- Compress to extract salient features & favor generalization

DQN Training: Target Value



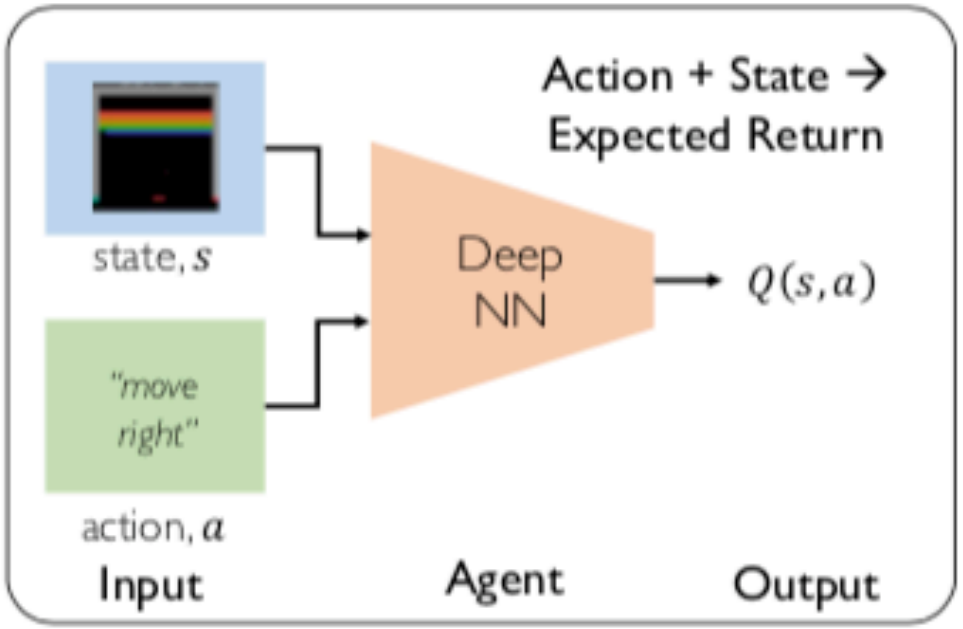


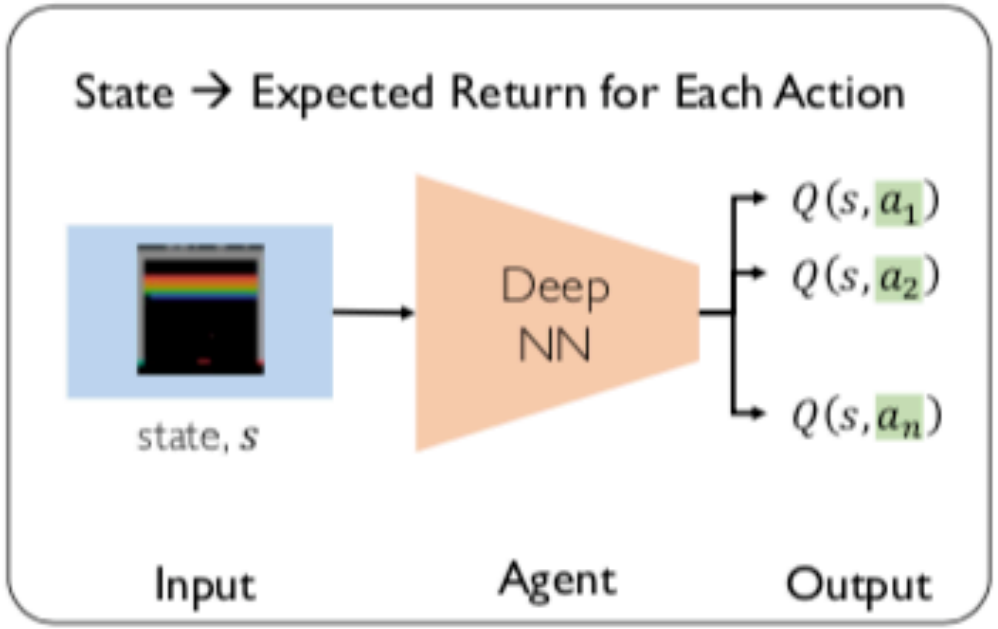
Training data: Experiences of the agent interacting with the environment. These are the values the agent is observing in the environment.

$$\overbrace{\left(r + \gamma \max_{a'} Q(s',a')\right)}^{\text{target}}$$

Take all the best actions → target return

DQN Training: Network Predictions



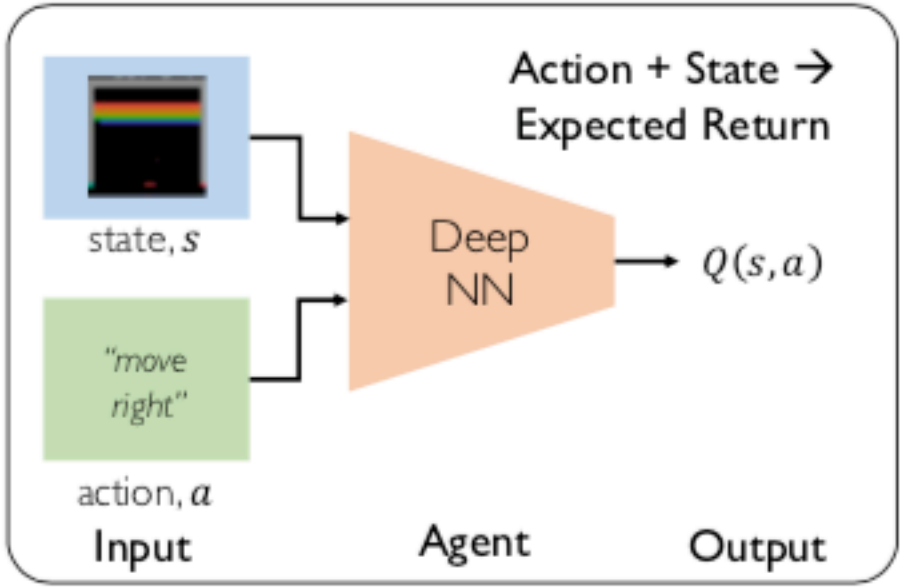


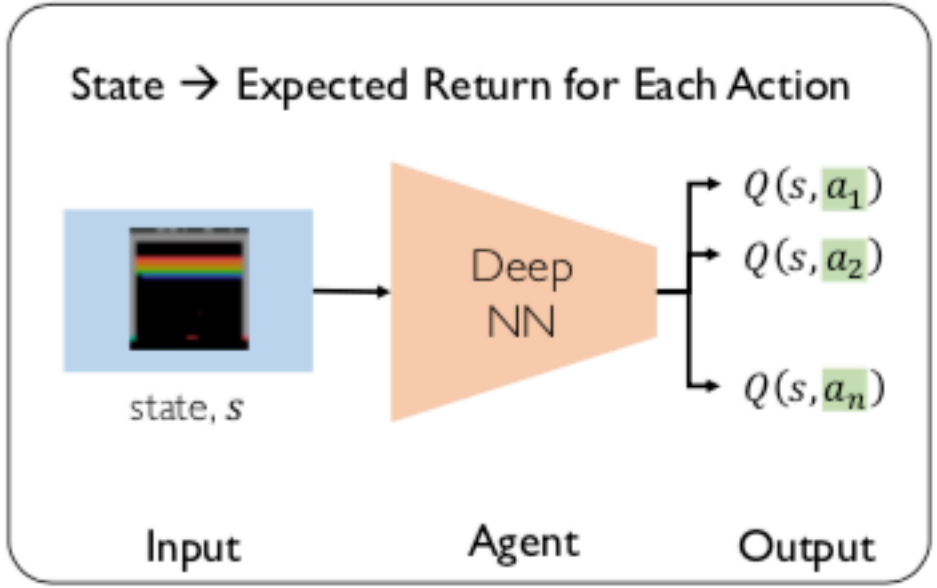
Prediction of the network is a Q (s, a) value.

$$\overbrace{\left(r + \gamma \max_{a'} Q(s', a')\right)}^{\text{target}} \qquad \overbrace{Q(s,a)}^{\text{predicted}}$$

Network prediction

DQN Training: MSE Loss Minimization



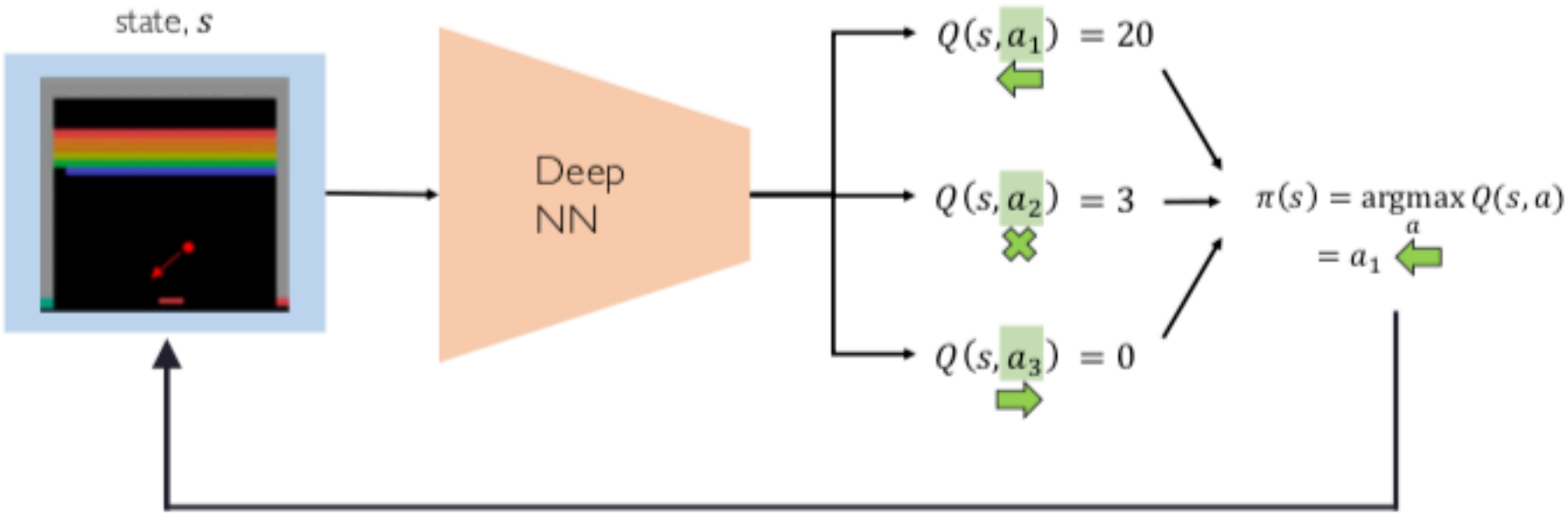


$$\mathcal{L} = \mathbb{E}\left[\left\|\overbrace{\left(r + \gamma \max_{a'} Q(s',a')\right)}^{\text{target}} - \overbrace{Q(s,a)}^{\text{predicted}}\right\|^2\right] \qquad \text{Q-Loss}$$

Given the target and the prediction, we can define a loss. Use the loss to backprop and learn the network parameters.

From Q-function to Policy

Use NN to learn Q-function and then use to infer the optimal policy, $\pi(s)$



Fitted Q-Learning (Control Setting)

- Q-learning seen so far works only for “discrete” actions spaces.
- How to extend Q-learning to problems where the actions space is continuous.
  - This is also called the control setting (e.g., controlling a car with a steering wheel) where the action space is not discrete.
- Fitted Q-learning: Enables Q-learning in domains with “continuous actions”
  - Idea is to use a parametric model for a Q-function: $Q_\theta(x_t, u_t)$
  - Optimize the parameters.
  - Note that action u is continuous.
  - The parametric model can be a neural network.

Fitted Q-Learning

- Fitted Q-learning (Ernst, Geurts, and Wehenkel 2005)
- Approximates the Q-function by breaking down the problem into a series of regression tasks.
- Each iteration of Fitted-Q iteration is similar to essentially solving a supervised regression problem.
- At every iteration of Fitted-Q iteration, the empirical Bellman operator is applied to the current Q-function approximation.
- Each Regression model then tries to find the best parameters to minimize the regression error.

Use parametric model for Q function: $Q_\theta(x, u)$

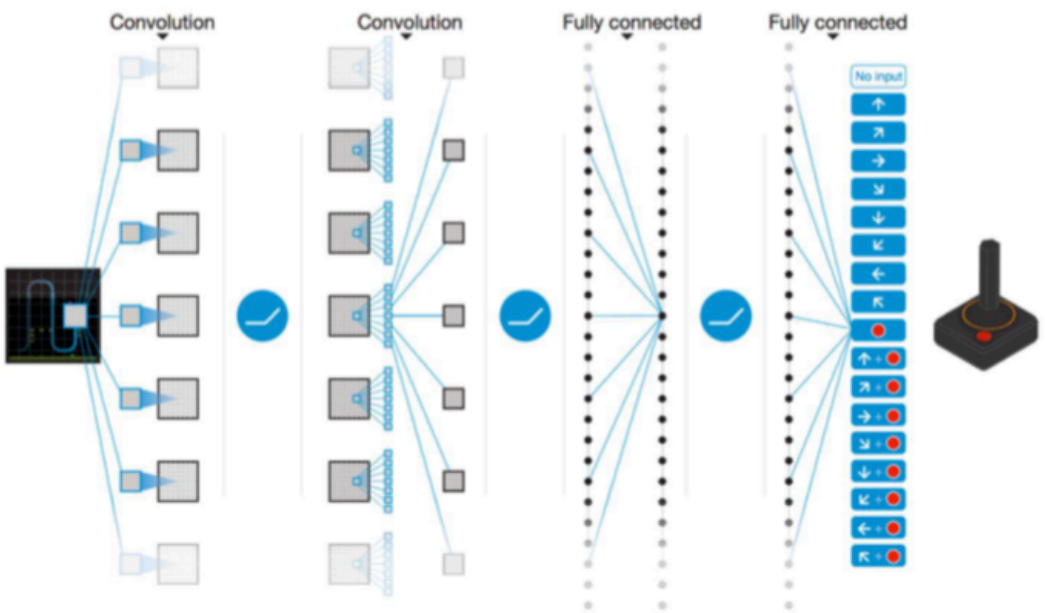
Gradient ascent on θ:

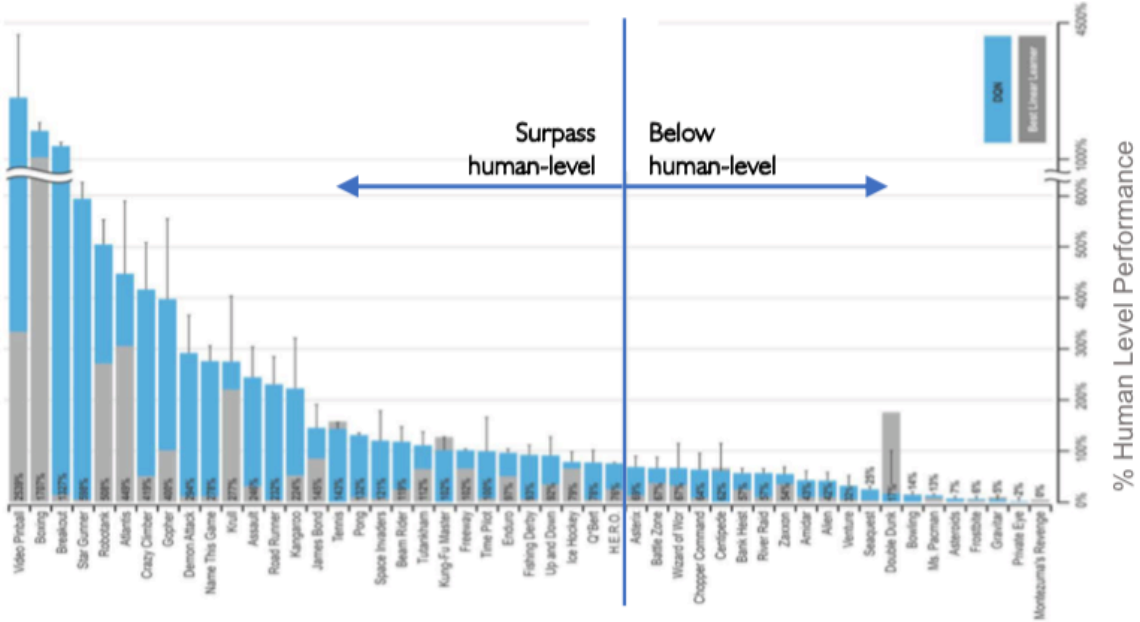
$$\theta \leftarrow \theta + \alpha \left(r_t + \gamma \max_u Q_\theta(x_{t+1}, u) - Q_\theta(x_t, u_t) \right) \nabla_\theta Q_\theta(x_t, u_t)$$

learning rate $\qquad \frac{d(Squared\ TD\ Error)}{dQ} \qquad \frac{dQ}{d\theta}$

https://arxiv.org/pdf/2006.05595.pdf

DQN successes in game play

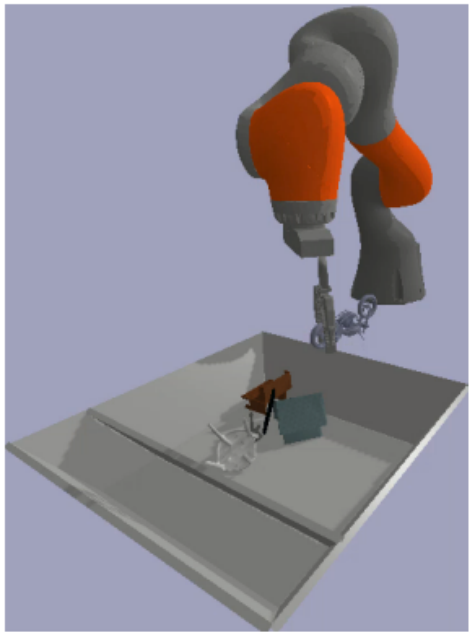




Use of CNNs to extract a state representation. Output is a movement of the agent in the grid.

Robot Grasping: QT-Opt

- State: Image, gripper status, height
- The agent is given a reward of 1 at the end of an episode if it has successfully grasped an object from the bin. In all other cases the reward is 0. Positive reward if the grasp is successful.
- The agent's action comprises a gripper pose displacement, and an open/close command. The gripper pose displacement is a difference between the current pose and the desired pose.



- https://sites.google.com/view/qtopt

QT-Opt: Scalable Deep Reinforcement Learning for Vision-Based Robotic Manipulation

QT-Opt

Figure 1: Seven robots are set up to collect grasping episodes with autonomous self-supervision.

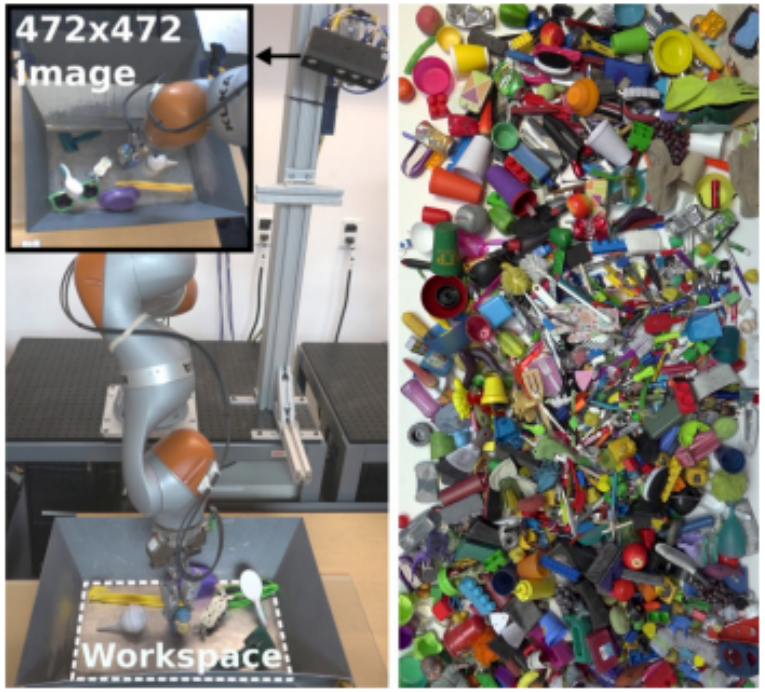
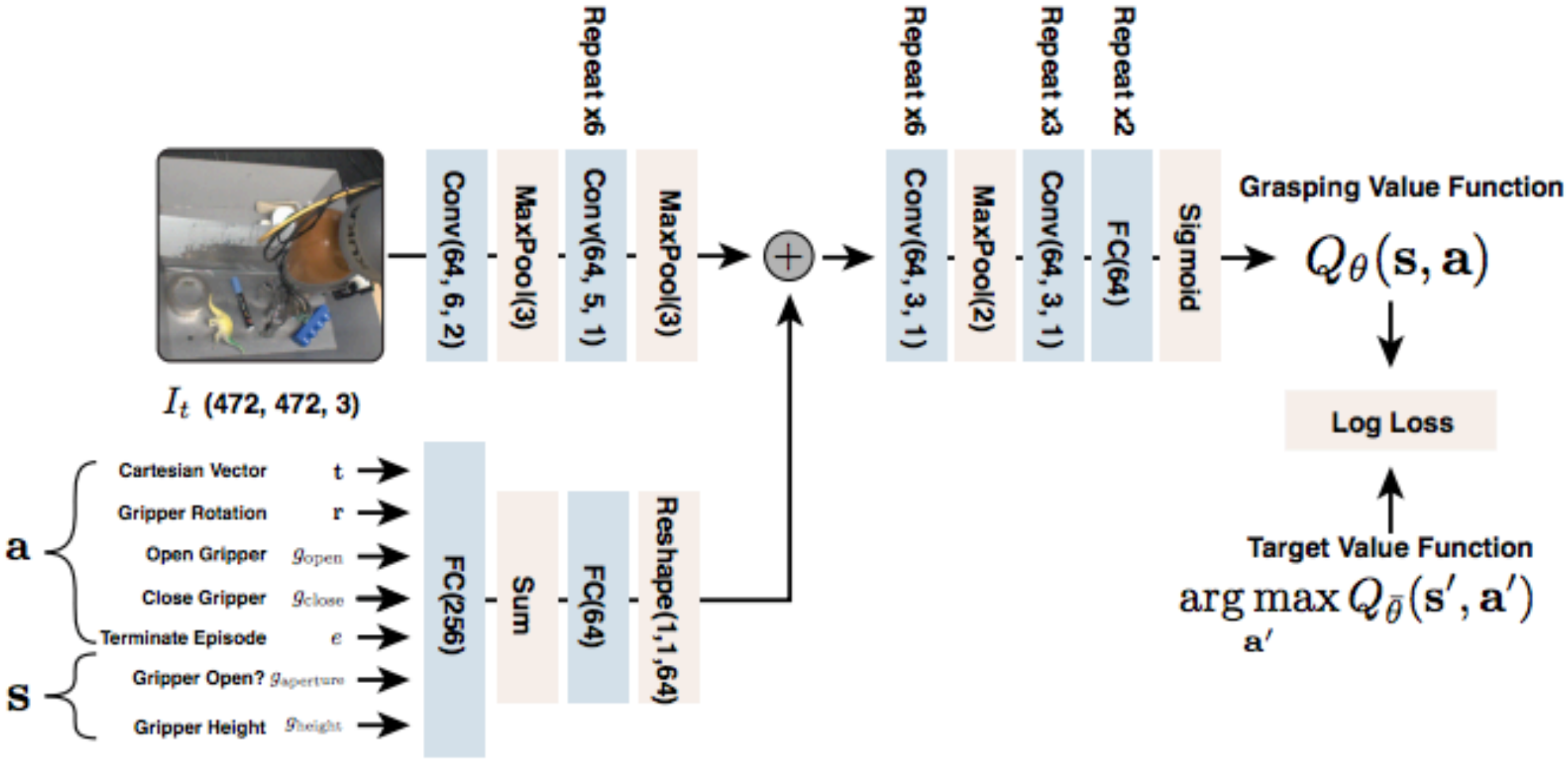


Figure 2: Close-up of a robot cell in our setup (left) and about 1000 visually and physically diverse training objects (right). Each cell (left) consists of a KUKA LBR IIWA arm with a two-finger gripper and an over-the-shoulder RGB camera.

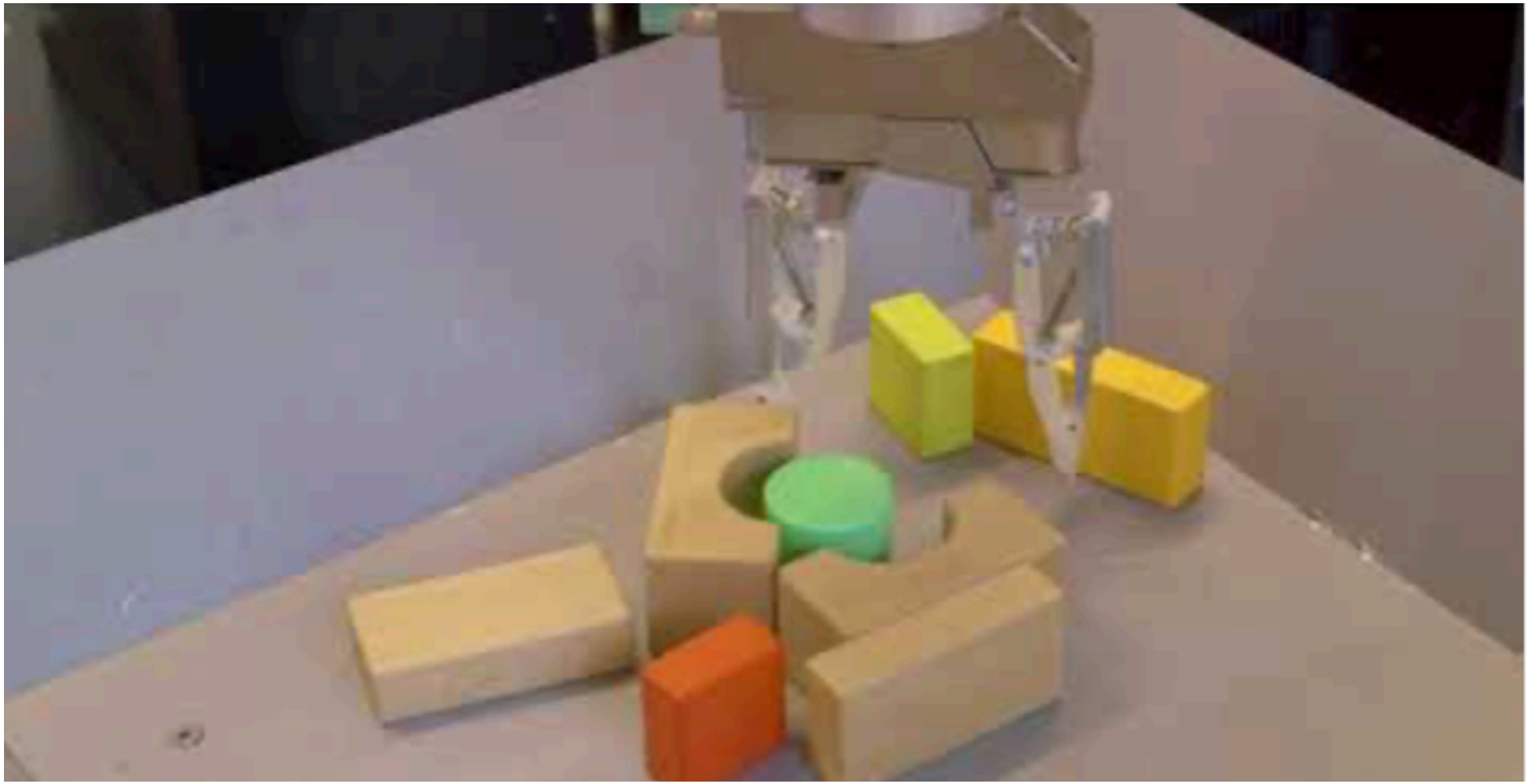
QT-Opt: Scalable Deep Reinforcement Learning for Vision-Based Robotic Manipulation

QT-Opt



QT-Opt: Scalable Deep Reinforcement Learning for Vision-Based Robotic Manipulation

QT-Opt



QT-Opt: Scalable Deep Reinforcement Learning for Vision-Based Robotic Manipulation

QT-Opt

Figure 4: Eight grasps from the QT-Opt policy, illustrating some of the strategies discovered by our method: pregrasp manipulation (a, b), grasp readjustment (c, d), grasping dynamic objects and recovery from perturbations (e, f), and grasping in clutter (g, h). See discussion in the text and Appendix A.

Algorithm 1 Grasping control-loop

```
Pick a policy policy.
Initialize a robot.
while step < N and not terminate_episode do
    s = robot.CaptureState()
    a = policy.SelectAction(s)
    robot.ExecuteCommand(a)
    terminate_episode = e {Termination action e is either learned or decided heuristically.}
    r = robot.ReceiveReward()
    emit(s, a, r)
end while
```

QT-Opt: Scalable Deep Reinforcement Learning for Vision-Based Robotic Manipulation

Catastrophic Forgetting

- Problem: Neural network regression assumes that training data are “IID”
  - The data are drawn Independently from the Identical Distribution.

- Temporal Correlations (*independence assumption*)
  - When a learning agent is moving through an environment, the sequence of states it encounters will be temporally correlated.

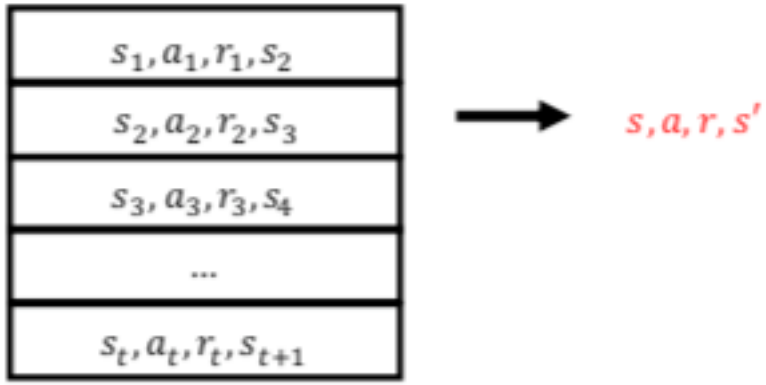
- Day and Night Robot Example (*identically distributed*)
  - Imagine a robot that spends 12 hours in a dark environment and 12 hour in a light one”.
  - Can mean that while it is in the dark, the neural-network weight-updates will make the Q function “forget” the value function for when it’s light.
  - Example from Leslie Kaelbling and Lozano-Perez.

- Use of Experience Replay
  - To avoid catastrophic forgetting
  - To break temporal correlations/ remove correlations

Experience Replay

- Solution: use experience replay, where we save our (s_t, a_t, r_t, s_{t+1}) experiences in a replay buffer.
  - Whenever we take a step in the world, we add the (s_t,a_t,r_t,s_{t+1}) to the replay buffer and use it to do a Q-learning update.
  - Then we also randomly select some number of tuples from the replay buffer, and do Q-learning updates based on them, as well.

Maintain a replay buffer



- To perform experience replay, repeat the following:
  - $(s, a, r, s') \sim \mathcal{D}$: sample an experience tuple from the dataset
  - Compute the target value for the sampled s: $r + \gamma \max_{a'} \hat{Q}(s', a'; \boldsymbol{w})$
  - Use stochastic gradient descent to update the network weights

$$\Delta \boldsymbol{w} = \alpha(r + \gamma \max_{a'} \hat{Q}(s', a'; \boldsymbol{w}) - \hat{Q}(s, a; \boldsymbol{w}))\nabla_{\boldsymbol{w}} \hat{Q}(s, a; \boldsymbol{w})$$

QT-Opt

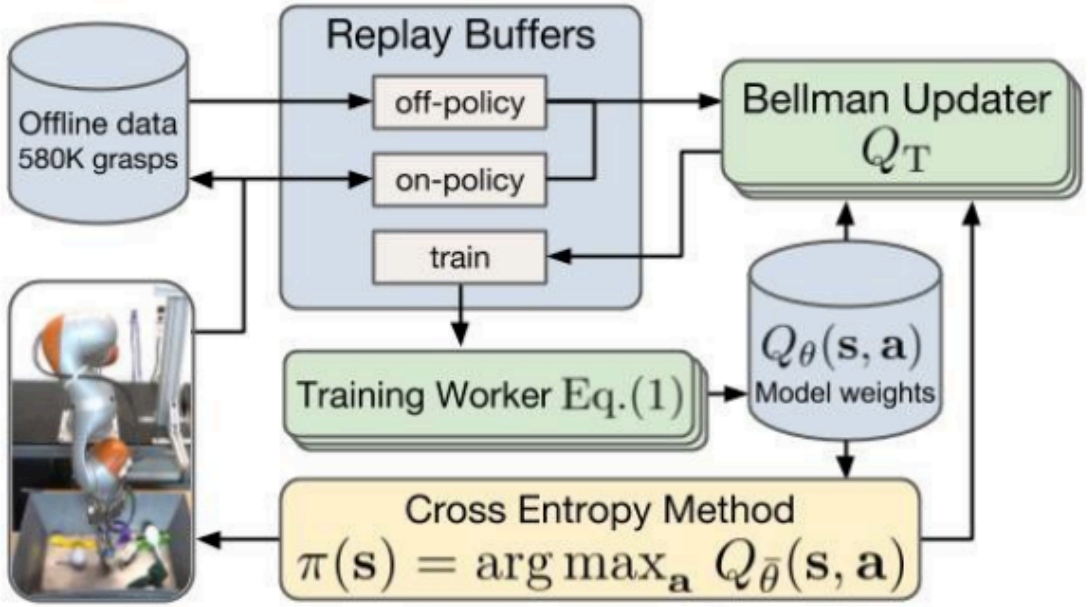


Figure 3: Our distributed RL infrastructure for QT-Opt (see Sec. 4.2). State-action-reward tuples are loaded from an offline data stored and pushed from online real robot collection (see Sec. 5). Bellman update jobs sample transitions and generate training examples, while training workers update the Q-function parameters.

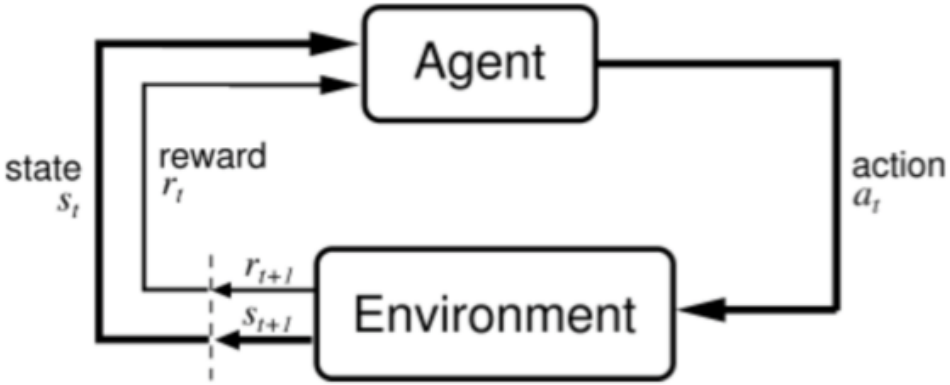
QT-Opt: Scalable Deep Reinforcement Learning for Vision-Based Robotic Manipulation

Limitations

- Works well when the the action space is discrete and small. Need to optimize over actions.
  - Hard to apply to continuous action spaces
- Policy is deterministically computed from the Q function by maximizing the rewards. I.e., there is one policy that is output.
  - Cannot model stochastic policies (distribution over actions for a state)
- The optimal Q-function may be harder to learn in comparison to the optimal policy
  - Can we learn the policy directly?

Policy Gradient (PG)

- Key Idea
  - Model-free. Policy-based.
  - Instead of learning the Q function, learn the policy directly.
  - Define a class of policies π_θ where θ are the parameters of the policy.
- Goal
  - Learn the optimal θ from interaction?
  - Model the objective of the policy (maximize future expected reward).
  - Use trajectories to estimate a gradient of policy performance w.r.t parameters θ.



$$\tau = (x_0, u_0, r_0, x_1, \dots, u_{H-1}, r_{H-1}, x_H)$$

Given episodes (trajectories) estimate the optimal policy directly without first estimating the Q-function.
Note the notation we will use in the derivation above.

Credit: Some slides on Policy Gradients are adapted from Principles of Robot Autonomy taught by Jeanette Bohg, Dorsa Sadigh and Marco Pavone.

PG: Setting up the objective J(θ)

A particular value of θ induces a distribution of possible trajectories

Objective function:

$$J(\theta) = E_{\tau \sim p(\tau;\theta)}[r(\tau)]$$

$$J(\theta) = \int_{\tau} r(\tau) p(\tau;\theta) d\tau$$

where $r(\tau)$ is the total discounted cumulative reward of a trajectory

The objective function: Sample a trajectory given the policy π_θ and estimate its cumulative reward.

PG: Gradient of J(θ) w.r.t. policy parameter θ

Gradient of objective w.r.t. parameters:

$$\nabla_\theta J(\theta) = \int_\tau r(\tau) \nabla_\theta p(\tau;\theta) d\tau$$

Trick: $\nabla_\theta p(\tau;\theta) = p(\tau;\theta) \frac{\nabla_\theta p(\tau;\theta)}{p(\tau;\theta)} = p(\tau;\theta) \nabla_\theta \log p(\tau;\theta)$

$$\nabla_\theta J(\theta) = \int_\tau (r(\tau) \nabla_\theta \log p(\tau;\theta)) p(\tau;\theta)\, d\tau$$

$$\nabla_\theta J(\theta) = E_{\tau \sim p(\tau;\theta)} [r(\tau) \nabla_\theta \log p(\tau;\theta)]$$

Gradient of the objective has a useful form.

Policy Gradient

$$\nabla_\theta J(\theta) = E_{\tau \sim p(\tau;\theta)}[r(\tau)\nabla_\theta \log p(\tau;\theta)]$$

$$\log p(\tau;\theta) = \log\left(\prod_{t\geq 0} T(x_{t+1}|x_t,u_t)\pi_\theta(u_t|x_t)\right)$$

$$= \sum_{t\geq 0} \log T(x_{t+1}|x_t,u_t) + \log \pi_\theta(u_t|x_t)$$

Transition model does not affect. Not dependent on theta.

$$\nabla_\theta \log p(\tau;\theta) = \sum_{t\geq 0} \nabla_\theta \log \pi_\theta(u_t|x_t)$$

Expand out the likelihood of the trajectory given the policy. This is for one single trajectory.

Policy Gradient

If we use π_θ to sample a trajectory, we can approximate the gradient:

Policy Gradient

$$\nabla_\theta J(\theta) \approx \sum_{t \geq 0} r(\tau) \nabla_\theta \log \pi_\theta(u_t | x_t)$$

Intuition: adjust θ to:

- Boost probability of actions taken if reward is high
- Lower probability of actions taken if reward is low

Learning by trial and error

Policy Gradient and REINFORCE algorithm

Learn (update) the policy parameter theta through gradient ascent.

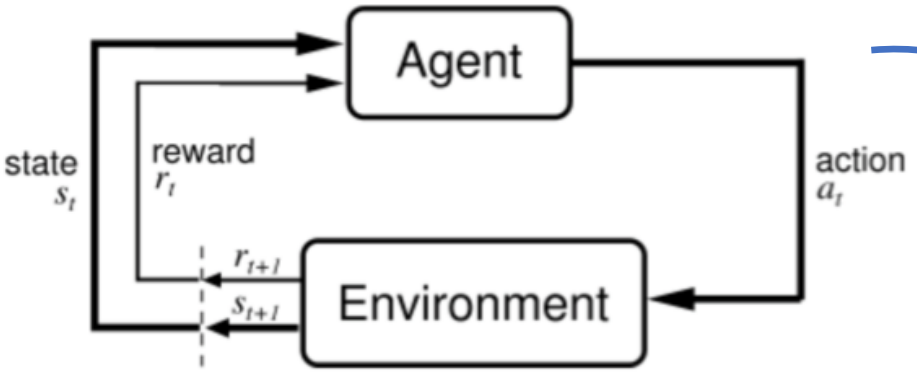
$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha \widehat{\nabla J(\boldsymbol{\theta}_t)}$$

Interleave policy execution and policy optimization steps. Estimate the policy gradient using several sampled trajectories.

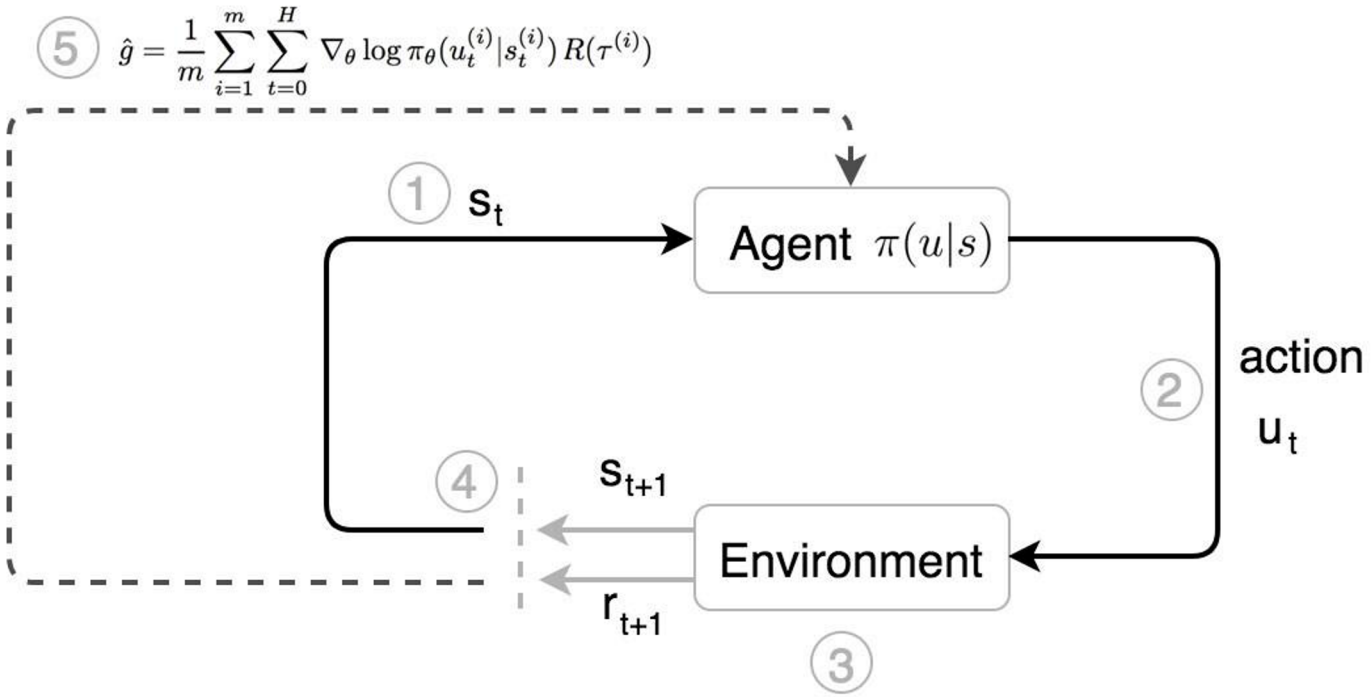
REINFORCE algorithm:

1. sample $\{\tau^i\}$ from $\pi_\theta(\mathbf{a}_t|\mathbf{s}_t)$ (run the policy)
2. $\nabla_\theta J(\theta) \approx \sum_i \left(\sum_t \nabla_\theta \log \pi_\theta(\mathbf{a}_t^i|\mathbf{s}_t^i)\right)\left(\sum_t r(\mathbf{s}_t^i, \mathbf{a}_t^i)\right)$
3. $\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta)$

Policy Gradients: In Summary

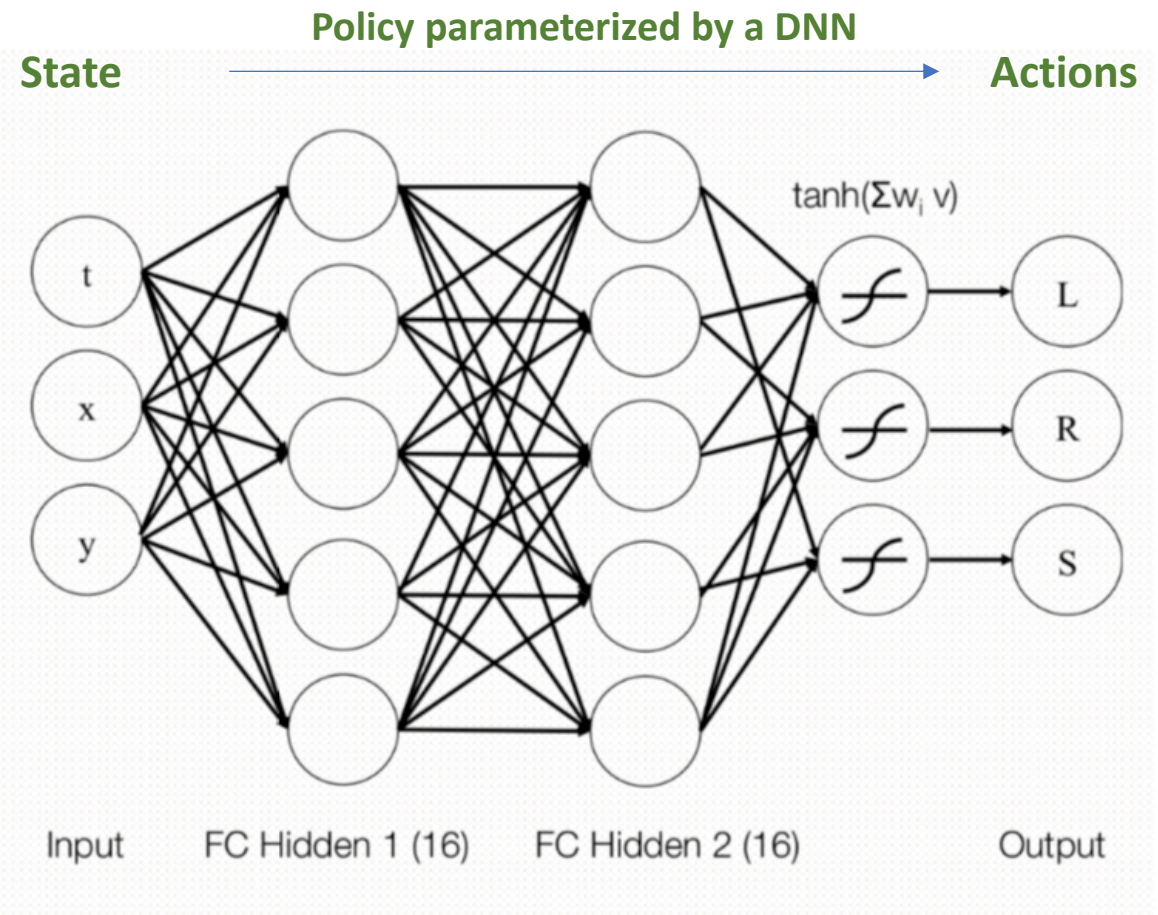


The agent learns to optimize the policy to best explain its experience.



“Deep” Policy Gradients

- Deep Policy Gradient:
  - Parameterize policy as deep neural network
  - Policy can act on high dimensional input, e.g. directly from visual feedback
  - Note that our derivation needed a policy parameterized by a set of parameters. We can plug in a neural network there.



Case Study: Learning to Drive

Reinforcement Learning Loop:

Case Study – Self-Driving Cars

Agent: vehicle

State: camera, lidar, etc

Action: steering wheel angle

Reward: distance traveled

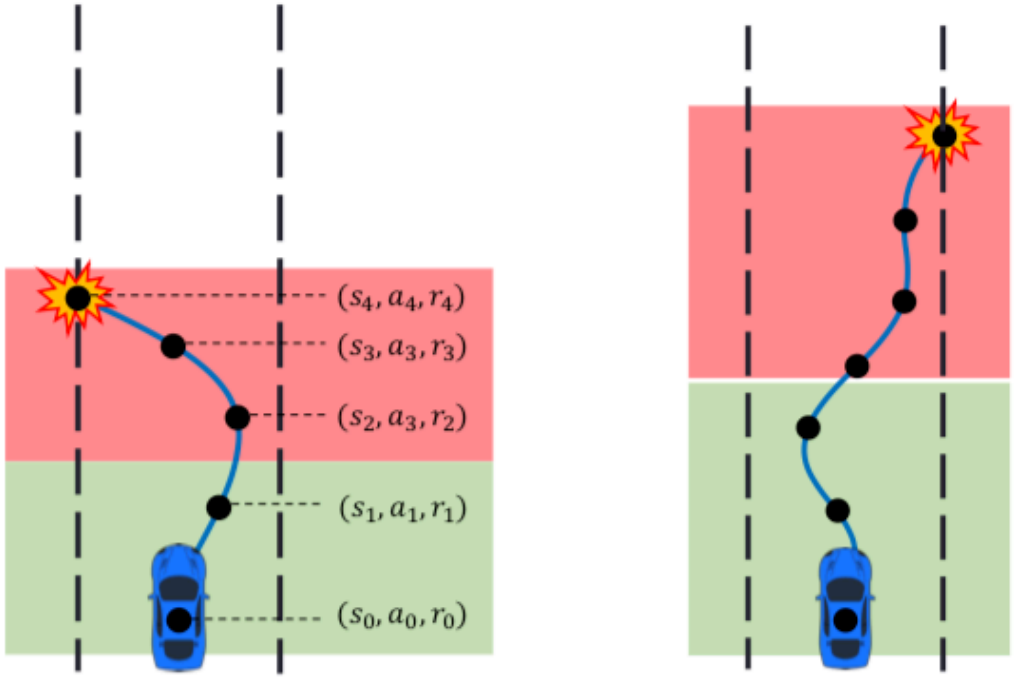
without collisions

Example from Alexandar Amini (Introduction to Deep Learning)

Training Policy Gradients

Training Algorithm

1. Initialize the agent
2. Run a policy until termination
3. Record all states, actions, rewards
4. Decrease probability of actions that resulted in low reward
5. Increase probability of actions that resulted in high reward

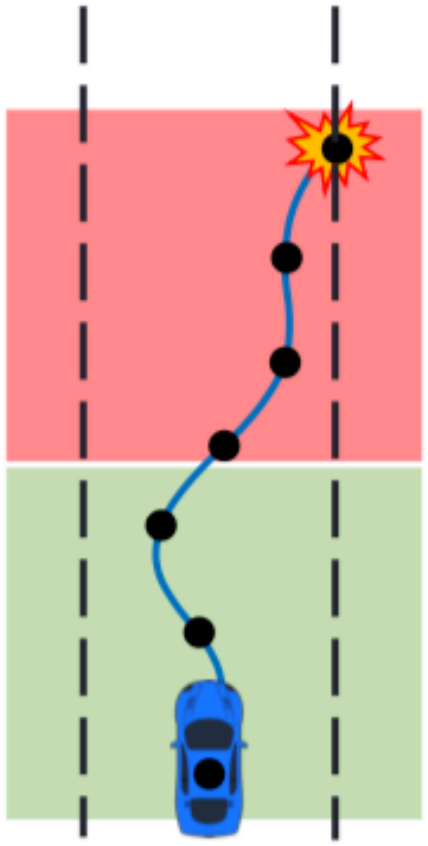


Run the policy to obtain different episodes.

Training Policy Gradients

Training Algorithm

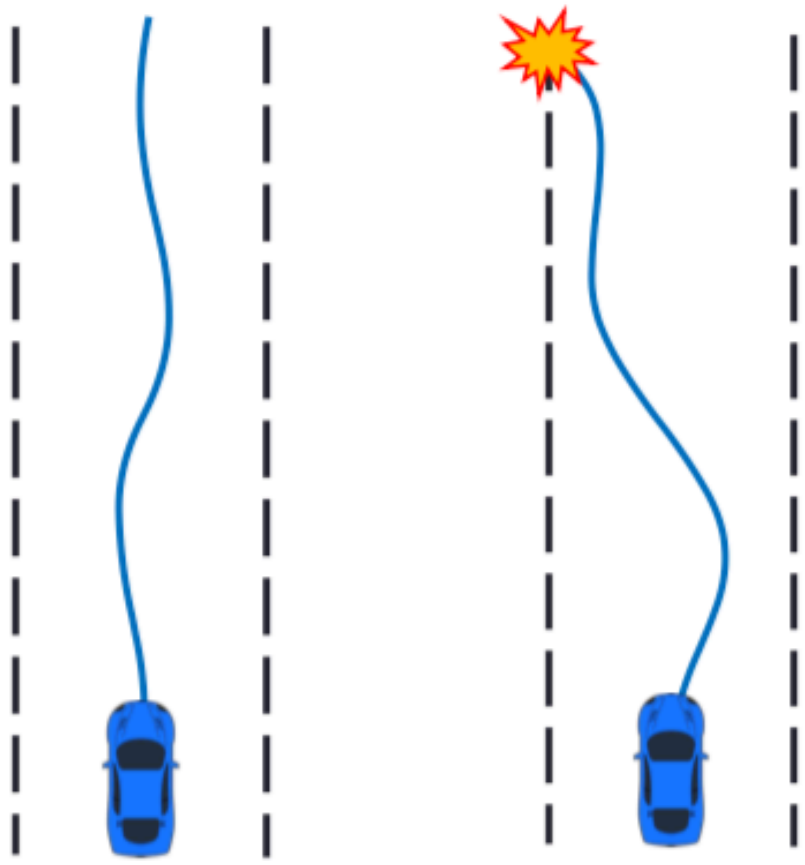
1. Initialize the agent
2. Run a policy until termination
3. Record all states, actions, rewards
4. Decrease probability of actions that resulted in low reward
5. Increase probability of actions that resulted in high reward



Training Policy Gradients

Training Algorithm

1. Initialize the agent
2. Run a policy until termination
3. Record all states, actions, rewards
4. Decrease probability of actions that resulted in low reward
5. Increase probability of actions that resulted in high reward



Increase probabilities of these actions

Decrease probabilities of these actions

Training Policy Gradients

Training Algorithm

1. Initialize the agent
2. Run a policy until termination
3. Record all states, actions, rewards
4. Decrease probability of actions that resulted in low reward
5. Increase probability of actions that resulted in high reward

log-likelihood of action

$$\mathbf{loss} = -\log \mathrm{P}(a_t|s_t)\, R_t$$

reward

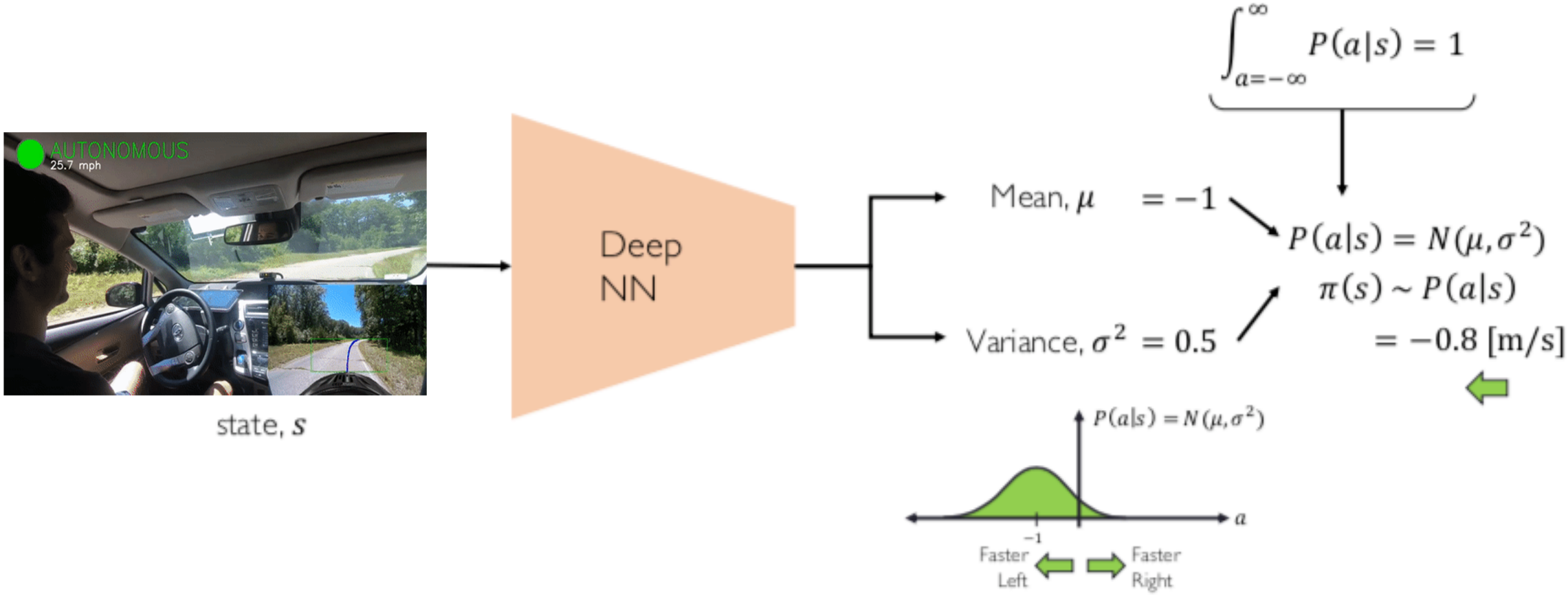
$$w' = w - \nabla\, \mathbf{loss}$$

$$w' = w + \boxed{\nabla \log \mathrm{P}(a_t|s_t)\, R_t}$$

Policy gradient!

Variational End-to-End Navigation and Localization. Alexander Amini, Guy Rosman, Sertac Karaman, Daniela Rus

Handling continuous action spaces

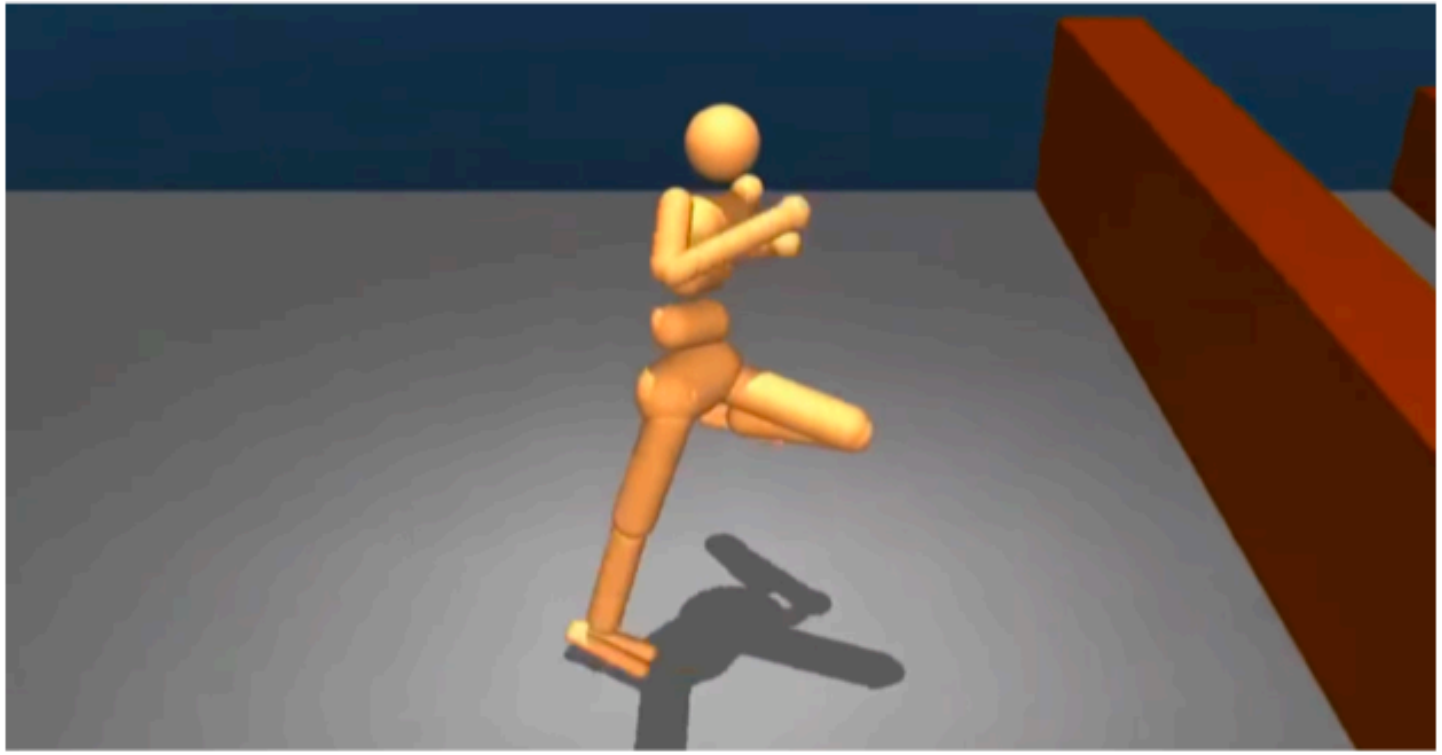


Robotics: Real world experiments are not easy

Real experiments may be dangerous

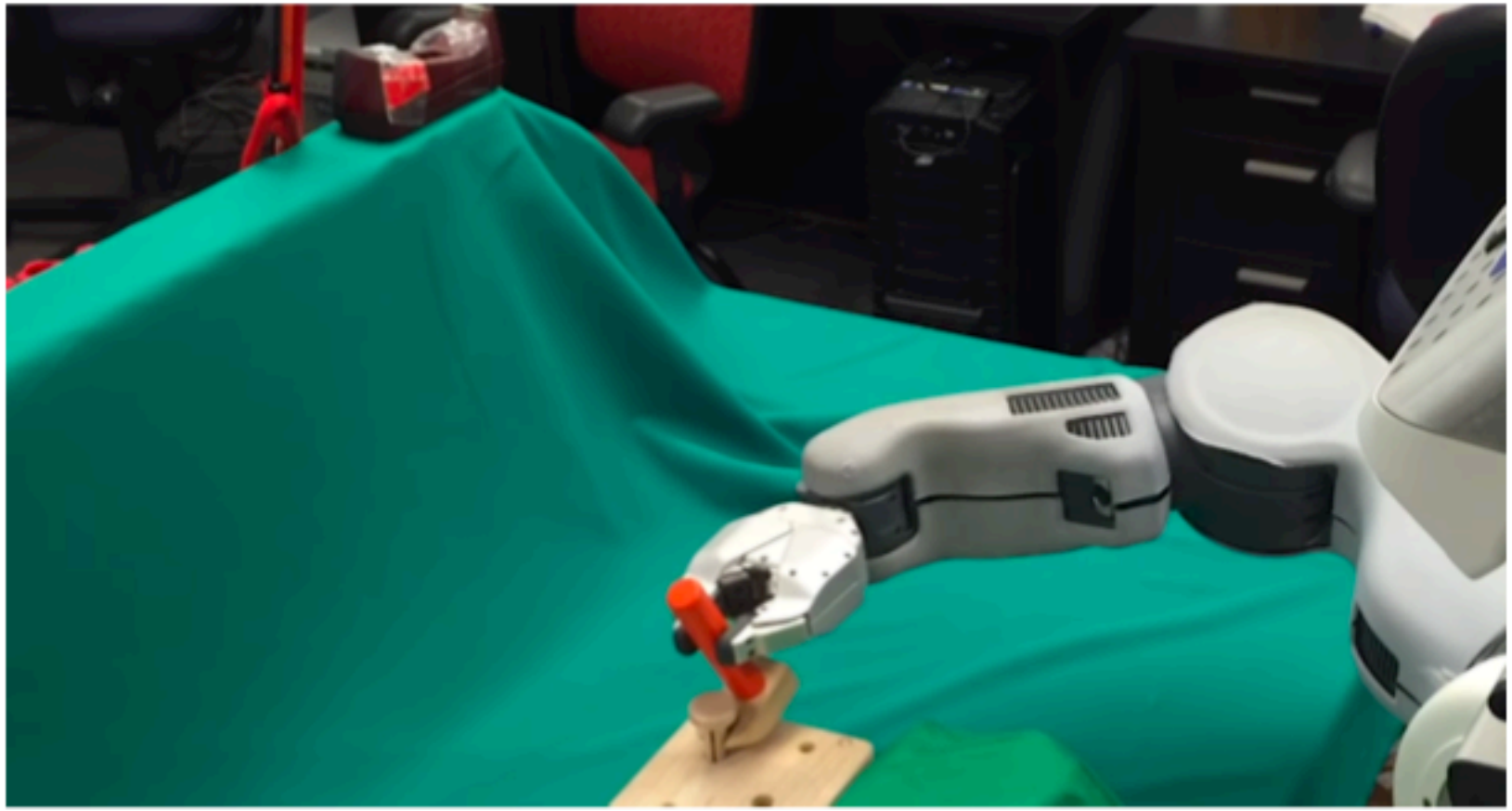
Perform roll outs in simulation.

PG Examples: Learning Locomotion for Humanoids (Sim)



Heess et al., "Emergence of Locomotion Behaviours in Rich Environments"
https://www.youtube.com/watch?v=hx_bgoTF7bs

Learning Robot Manipulation Skills



Levine et al., "End-to-End Training of Deep Visuomotor Policies"
https://sites.google.com/site/visuomotorpolicy/

Further Advances

- Reducing variance in the policy gradient estimation
  - Formulation of PG that uses an advantage function
  - Actor-Critic Methods
- Others advances
  - DDPG
  - Cross Entropy methods
  - TRPO