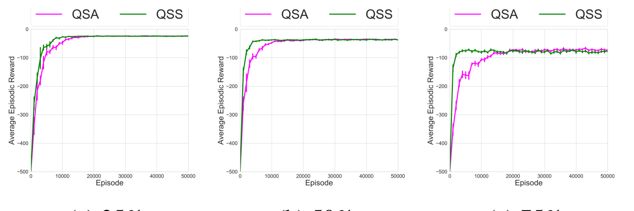
Appendices

A. QSS Experiments

We ran all experiments in an 11x11 gridworld. The state was the agent's $\langle x, y \rangle$ location on the grid. The agent was initialized to $\langle 0, 0 \rangle$ and received a reward of -1 until it reached the goal at $\langle 10, 10 \rangle$ and obtained a reward of 1 and was reset to the initial position. The episode automatically reset after 500 steps.

We used the same hyperparameters for QSA and QSS. We initialized the Q-values to $.001$. The learning rate α was set to $.01$ and the discount factor was set to $.99$. The agent followed an ϵ-greedy policy. Epsilon was set to 1 and decayed to $.1$ by subtracting 9e-6 every time step.

A.1. Additional stochastic experiments



(a) 25% (b) 50% (c) 75%

Figure 9. Stochastic experiments in an 11x11 gridworld. The first three experiments demonstrate the effect of stochastic actions on the average return. Before each episode, we evaluated the learned policy and averaged the return over 10 trials. All experiments were averaged over 10 seeds with 95% confidence intervals.

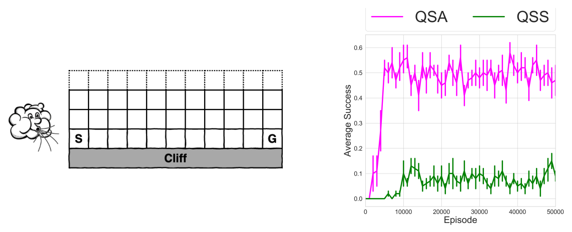


Figure 10. Stochastic experiments in cliffworld. This experiment measures the effect of stochastic actions on the average success rate. Before each episode, we evaluated the learned policy and averaged the return over 10 trials. All experiments were averaged over 10 seeds with 95% confidence intervals.

We were interested in measuring the impact of stochastic transitions on learning using QSS. To investigate this property, we add a probability of slipping to each transition, where the agent takes a random action (i.e. slips into an unintended next state) some percentage of time. Curiously, QSS solves this task quicker than QSA, even though it learns incorrect values (Figure 9). One hypothesis is that the slippage causes the agent to stumble into the goal state, which is beneficial for QSS because it directly updates values based on state transitions. The correct action that enables this transition is known using the given inverse dynamics model. QSA, on the other hand, would need to learn how the stochasticity of the environment affects the selected action's outcome and so the values may propagate more slowly.

We additionally study the case when stochasticity may lead to negative effects for QSS. We modify the gridworld to include a cliff along the bottom edge similar to the example in Sutton & Barto (1998). The agent is initialized on top of the cliff, and if it attempts to step down, it falls off and the episode is reset. Furthermore, the cliff is "windy", and the agent has a 0.5 probability of falling off the edge while walking next to it. The reward here is 0 everywhere except the goal, which has a reward of 1. Here, we see the effect of stochasticity is detrimental to QSS (Figure 10), as it does not account for falling and instead expects to transition towards the goal.

A.2. Additional transfer experiment

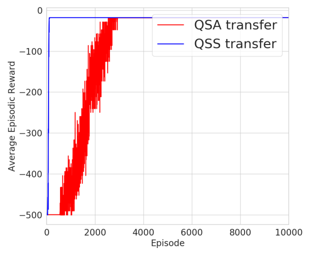


Figure 11. Transfer experiments within 11x11 gridworld. The experiment represents how well QSS and QSA transfer to a gridworld with permuted actions. We now include an additional action that transports the agent back to the start. All experiments shown were averaged over 50 random seeds with 95% confidence intervals

We trained QSA and QSS in a gridworld with an additional *transport* action that moved the agent back to the start. We then transferred the learned values to an environment where the action labels were shuffled. Incorrectly taking the transport action would have a larger impact on the average return than the other actions. QSS is able to learn much more quickly than QSA, as it only needs to relearn the inverse dynamics and avoids the negative impacts of the incorrectly labeled transport action.

B. D3G Experiments

We used the TD3 implementation from `https://github.com/sfujim/TD3` for our experiments. We also used the "OurDDPG" implementation of DDPG. We built our own implementation of D3G from this codebase. We used the default hyperparameters for all of our experi-

Θ	**D3G**	**TD3**	**DDPG**	**BCO**
Critic lr	3e-4	3e-4	3e-4	–
Actor lr	–	3e-4	3e-4	–
BC lr	–	–	–	3e-4
$\tau(s)$ lr	3e-4	–	–	–
$f(s, \cdot)$ lr	3e-4	–	–	–
$I(s, s')$ lr	3e-4	–	–	3e-4
β	1.0	–	–	–
η	0.005	0.005	0.005	–
Optimizer	Adam	Adam	Adam	Adam
Batch Size	256	256	256	256
γ	0.99	0.99	0.99	–
Delay (d)	2	2	–	–

Table 2. Hyperparameters Θ for D3G experiments.

ments, as described in Table 2. The replay buffer was filled for 10000 steps before learning. All continuous experiments added noise $\epsilon \sim \mathcal{N}(0, 0.1)$ for exploration. In gridworld, the agent followed an ϵ-greedy policy. Epsilon was set to 1 and decayed to .1 by subtracting 9e-6 every time step.

B.1. Gridworld task

We ran these experiments in an 11x11 gridworld. The state was the agent's $\langle x, y \rangle$ location on the grid. The agent was initialized to $\langle 0, 0 \rangle$ and received a reward of -1 until it reached the goal at $\langle 10, 10 \rangle$ and obtained a reward of 0 and was reset to the initial position. The episode automatically reset after 500 steps.

B.2. MuJoCo tasks

We ran these experiments in the OpenAI Gym MuJoCo environment `https://github.com/openai/gym`. We used gym==0.14.0 and mujoco-py==2.0.2. The agent's state was a vector from the MuJoCo simulator.

B.3. Learning from Observation Experiments

We used TD3 to train an expert and used the learned policy to obtain demonstrations D for learning from observation. We collected $1e6$ samples using the learned policy and took a random action either 0, 25, 50, 75, or 100 percent of the time, depending on the experiment. The samples consisted of the state, reward, next state, and done condition.

We trained BCO with D for 100 iterations. During each iteration, we collected 1000 samples from the environment using a Behavioral Cloning (BC) policy with added noise $\epsilon \sim \mathcal{N}(0, 0.1)$, then trained an inverse dynamics model for 10000 steps, labeled the observational data using this model, then finally trained the BC policy with this labeled data for 10000 steps.

We trained D3G with D for $1e6$ time steps without any environment interactions. This allowed us to learn the model $\tau(s)$ which informed the agent of what state it should reach. Similarly to BCO, we used some environment interactions to train an inverse dynamics model for D3G. We ran this training loop for 100 iterations as well. During each iteration, we collected 1000 samples from the environment using the inverse dynamics policy $I(s, m(s))$ with added noise $\epsilon \sim \mathcal{N}(0, 0.1)$, then trained this model for 10000 steps.

C. Architectures

D3G Model $\tau(s)$:

$s \to fc_{256} \to relu \to fc_{256} \to relu \to fc_{len(s)}$

D3G Forward Dynamics Model:

$\langle s, a \rangle \to fc_{256} \to relu \to fc_{256} \to relu \to fc_{len(s)}$

D3G Forward Dynamics Model (Imitation):

$\langle s, q \rangle \to fc_{256} \to relu \to fc_{256} \to relu \to fc_{len(s)}$

D3G Inverse Dynamics Model (Continuous):

$\langle s, s' \rangle \to fc_{256} \to relu \to fc_{256} \to relu \to fc_{len(a)} \to tanh \cdot$ max action

D3G Inverse Dynamics Model (Discrete):

$\langle s, s' \rangle \to fc_{256} \to relu \to fc_{256} \to relu \to fc_{len(a)} \to softmax$

D3G Critic: $\langle s, s' \rangle \to fc_{256} \to relu \to fc_{256} \to relu \to fc_{l}$

TD3 Actor:

$s \to fc_{256} \to relu \to fc_{256} \to relu \to fc_{len(a)} \to tanh \cdot$ max action

TD3 Critic:

$\langle s, a \rangle \to fc_{256} \to relu \to fc_{256} \to relu \to fc_{l}$

DDPG Actor:

$s \to fc_{400} \to relu \to fc_{300} \to relu \to fc_{len(a)} \to tanh \cdot$ max action

DDPG Critic:

$\langle s, a \rangle \to fc_{400} \to relu \to fc_{300} \to relu \to fc_{l}$

BCO Behavioral Cloning Model:

$s \to fc_{256} \to relu \to fc_{256} \to relu \to fc_{len(a)} \to tanh \cdot$ max action

BCO Inverse Dynamics Model:

$\langle s, s' \rangle \to fc_{256} \to relu \to fc_{256} \to relu \to fc_{len(a)} \to tanh \cdot$ max action