CS294-112 Deep Reinforcement Learning HW5: Advanced Topics

Due November 9th, 11:59 pm

1 Introduction

For this homework, you get to choose among several topics to investigate. Each topic corresponds to a different assignment for HW5. You will implement only **one** of the assignments. You can implement a second assignment as a make-up for a previous homework or for extra credit. Section 2 is for the Soft Actor-Critic assignment.

2 Soft Actor-Critic

In Homework 3, you implemented an actor-critic algorithm based on policy gradients. The main disadvantage of policy gradient methods is that they are *on-policy*—a fresh set of data is needed for every policy gradient update. In this homework, you will implement soft actor-critic [Haarnoja et al., 2018], which is a value-based actor-critic method and can thus make use of *off-policy* data for better sample efficiency.

3 Background

3.1 Maximum Entropy Reinforcement Learning

Soft actor-critic optimizes the maximum entropy reinforcement learning objective

$$J(\pi) = \mathbb{E}_{(\mathbf{s}_0,\mathbf{a}_0,\mathbf{s}_1,\mathbf{a}_1,\ldots,\mathbf{s}_T,\mathbf{a}_T)\sim\rho_\pi}\left[\sum_{t=0}^{T} r(\mathbf{s}_t,\mathbf{a}_t) + \alpha\mathcal{H}(\pi_t(\cdot\,|\mathbf{s}_t))\right], \tag{1}$$

where ρ_π is the joint distribution over states and actions induced by the policy π, α is a temperature parameter, and $\mathcal{H}$ denotes the entropy. The optimal policy with respect to the maximum entropy objective is called the *maximum entropy policy*. One of the intriguing properties of maximum entropy policies is that they automatically trade off between exploration (maximizing entropy) and exploitation (maximizing return). This trade-off is controlled by the temperature parameter α: in the limit as $\alpha \to 0$, we recover the standard maximum expected return objective. From a practical point of view, maximum entropy reinforcement learning provides more stable algorithms due to the regularizing effect of the entropy term, which can help to make learning more efficient, particularly in continuous action spaces. You have already seen how the maximum entropy objective can be derived from the connection between optimal control and inference in Lecture 15. For more detailed analysis of maximum entropy reinforcement learning, see Haarnoja et al. [2017], Levine [2018].

3.2 Soft Actor-Critic

Soft actor-critic is a value-based algorithm that maximizes the maximum entropy objective in Equation 1. The algorithm learns both the *soft Q-function*, $Q^\pi : \mathcal{S} \times \mathcal{A} \mapsto \mathbb{R}$, and a *soft policy*, $\pi : \mathcal{A} \times \mathcal{S} \mapsto \mathbb{R}_{\geq 0}$, and we sometimes refer to them simply as the Q-function and policy. The soft Q-function is defined as

$$Q_t^\pi(\mathbf{s}_t, \mathbf{a}_t) \triangleq r(\mathbf{s}_t, \mathbf{a}_t) + \mathbb{E}_{(\mathbf{s}_{t+1}, \mathbf{a}_{t+1}, \ldots) \sim p_\pi} \left[\sum_{l=1}^{T} r(\mathbf{s}_{t+l}, \mathbf{a}_{t+l}) + \alpha \log \pi_t(\mathbf{a}_{t+l} | \mathbf{s}_{t+l}) \right], \tag{2}$$

and it satisfies the soft Bellman equation

$$Q_t^\pi(\mathbf{s}_t, \mathbf{a}_t) = r(\mathbf{s}_t, \mathbf{a}_t) + \mathbb{E}_{\mathbf{s}_{t+1} \sim p(\mathbf{s}_{t+1} | \mathbf{s}_t, \mathbf{a}_t)} \left[V_{t+1}^\pi(\mathbf{s}_{t+1}) \right], \tag{3}$$

where

$$V_t^\pi(\mathbf{s}_t) \triangleq \mathbb{E}_{\mathbf{a}_t \sim \pi_t(\mathbf{a}_t | \mathbf{s}_t)} \left[Q_t^\pi(\mathbf{s}_t, \mathbf{a}_t) - \alpha \log \pi_t(\mathbf{a}_t | \mathbf{s}_t) \right] \tag{4}$$

is the *soft value function*. We can evaluate the soft Q value of a fixed policy π by applying Equation 3 to each time step. This procedure is called *soft policy evaluation*. We can now write the objective in Equation 1 in terms of the soft Q-function as

$$J(\pi) = \mathbb{E}_{\mathbf{s}_0 \sim p(\mathbf{s}_0),\ \mathbf{a}_0 \sim \pi_0(\mathbf{a}_0 | \mathbf{s}_0)} \left[Q_0^\pi(\mathbf{s}_0, \mathbf{a}_0) - \alpha \log \pi_0(\mathbf{a}_0 | \mathbf{s}_0) \right] \tag{5}$$

$$= -\mathbb{E}_{\mathbf{s}_0 \sim p(\mathbf{s}_0)} \left[\mathrm{D}_{\mathrm{KL}} \left(\pi_0(\cdot | \mathbf{s}_0) \,\middle\|\, \exp\left(\frac{1}{\alpha} Q_0^\pi(\mathbf{s}_0, \cdot) \right) \right) \right] + \textit{constant}. \tag{6}$$

We call minimization of the above objective with respect to the policy the *policy improvement step*.

You will implement soft actor-critic for the discounted, infinite horizon case. Soft actor-critic applies the soft policy evaluation and soft policy improvement steps using stochastic gradient descent as explained in the next section. The exact objective function for that case is slightly more involved, but from an optimization point of view, it amounts to simply dropping the time dependencies, and discounting the value of the next time step in the soft Bellman backup (see Haarnoja et al. [2017]).

3.3 Implementation

You will be asked to implement the loss functions for the Q-function, value function, and the policy, parameterized by θ, ψ, and ϕ, respectively. Similar to Q-learning, we can learn the soft Q-function by minimizing the squared soft Bellman residual

$$J_Q(\theta) = \mathbb{E}_{(\mathbf{s},\mathbf{a},\mathbf{s}')\sim\mathcal{D}}\left[\left(Q_\theta(\mathbf{s},\mathbf{a}) - \left(r(\mathbf{s},\mathbf{a}) + \gamma V_{\bar{\psi}}(\mathbf{s}')\right)\right)^2\right], \tag{7}$$

where $\mathcal{D}$ denotes the replay pool, and $\bar{\psi}$ are the *target* value function parameters. A target network is used to stabilize training, similar to the target network for Q-learning in Homework 3. We can learn the soft value function by using its definition in Equation 4 and least squares regression,

$$J_V(\psi) = \mathbb{E}_{\mathbf{s}\sim\mathcal{D}}\left[\left(V_\psi(\mathbf{s}) - \mathbb{E}_{\mathbf{a}\sim\pi_\phi(\mathbf{a}|\mathbf{s})}\left[Q_\theta(\mathbf{s},\mathbf{a}) - \alpha\log\pi_\phi(\mathbf{a}|\mathbf{s}) \mid \mathbf{s}\right]\right)^2\right], \tag{8}$$

and to learn the policy, we can minimize the KL divergence in Equation 6 (ignoring the constant as it is independent of the policy):

$$J_\pi(\phi) = \mathbb{E}_{\mathbf{s}\sim\mathcal{D}}\left[\mathbb{E}_{\mathbf{a}\sim\pi_\phi(\mathbf{a}|\mathbf{s})}\left[\alpha\log\pi_\phi(\mathbf{a}|\mathbf{s}) - Q_\theta(\mathbf{s},\mathbf{a}) \mid \mathbf{s}\right]\right]. \tag{9}$$

Soft actor-critic minimizes these loss functions using stochastic gradient descent. Note that unlike our actor-critic implementation from Homework 3, we have two critics: a V function and a Q function.

4 Installation

Obtain the code from `https://github.com/berkeleydeeprlcourse/homework/tree/master/hw5/sac`. To run the code, you will first need to create a virtual environment to setup all dependencies. We recommend you use Conda, but you can use any other environment management system too to install all requirements listed in `environment.yml`, as long as you make sure you are using the specified versions of the libraries. To setup a Conda environment, follow first the instruction in `https://conda.io/docs/user-guide/install/index.html`, then in `<homework_root>/hw5/sac/`, do the following:

```
conda env create
source activate hw5-sac
```

This will create and launch a virtual environment which you will you to train the policies. (Note: if you would like to run experiments on GPUs, change `tensorflow==1.10.0` to `tensorflow-gpu==1.10.0` in `environment.yml`.)

You can train a policy by running

```
python train_mujoco.py --env_name <environment_name> --exp_name
↪  <experiment_name> -e <num_seeds>
```

where `environment_name` can be any valid OpenAI Gym environment name (e.g., `HalfCheetah-v2`). You can tune the rest of the parameters (network sizes, algorithm hyperparams, etc) via a series of dictionaries defined in the `run_experiment` function. Before you can actually run the training script, you will need to implement the algorithm! You will need to edit `sac.py`, which contains the soft actor-critic implementation, and `nn.py` which contains the neural network definitions.

5 Problems

Problem 1: Loss functions

Problem 1.1: Q-Function Loss

In `sac.py`, look for the method `_q_function_loss_for`. The method takes in the Q-function Q_θ and the target value function $V_{\bar{\psi}}$, and outputs a value function loss over a minibatch. The two input arguments are Keras `Network` instances (defined in `nn.py`), and they provide an easy interface to construct neural networks for arbitrary inputs. For example, to obtain a $N \times 1$ tensor of Q values for a minibatch of size N, you can write

```
q_values = q_function((self._observations_ph, self._actions_ph))
```

where `self._observations_ph` is a $N \times |\mathcal{S}|$ and `self._actions_ph` is a $N \times |\mathcal{A}|$ placeholder tensor. All placeholders that you will need for this assignments are created in `_create_placeholders` method. Note that you can pass in any `tf.Tensor` objects as an input to a Keras `Network` object.

Task A: Look for

```
### Problem 1.1.A
### YOUR CODE HERE
```

in `sac.py` and replace it with your implementation of the Q-function loss. Again, the loss is defined as:

$$J_Q(\theta) = \mathbb{E}_{(\mathbf{s},\mathbf{a},\mathbf{s}')\sim\mathcal{D}}\left[\left(Q_\theta(\mathbf{s},\mathbf{a}) - \left(r(\mathbf{s},\mathbf{a}) + \gamma V_{\bar{\psi}}(\mathbf{s}')\right)\right)^2\right] \tag{10}$$

We will estimate this loss via a minibatch of N off-policy $(\mathbf{s},\mathbf{a},\mathbf{s}')$ tuples that come from the replay buffer $\mathcal{D}$. Don't forget to incorporate the discount factor and terminal masks in the loss function (just like we did in actor-critic for Homework 3)!

Problem 1.2: Value Function Loss

Your second task is to implement the value function loss in `_value_function_loss_for`.

Task A: Look for

```
### Problem 1.2.A
### YOUR CODE HERE
```

in `sac.py` and replace it with your implementation of the V-function loss. For now, ignore the input parameter `q_function2`, it will be relevant in a later part. Again, the loss is defined as:

$$J_V(\psi) = \mathbb{E}_{\mathbf{s}\sim\mathcal{D}}\left[\left(V_\psi(\mathbf{s}) - \mathbb{E}_{\mathbf{a}\sim\pi_\phi(\mathbf{a}|\mathbf{s})}\left[Q_\theta(\mathbf{s},\mathbf{a}) - \alpha\log\pi_\phi(\mathbf{a}|\mathbf{s}) \mid \mathbf{s}\right]\right)^2\right], \tag{11}$$

We will estimate the outer expectation via a minibatch of N off-policy states that come from the replay buffer $\mathcal{D}$. Note that the inner expectation over $\pi_\phi(\mathbf{a}|\mathbf{s})$ is on-policy, so you must resample actions from the policy again rather than from the replay buffer. This operation will not "hurt" us in terms of sample efficiency, since we're just resampling from the neural network policy — we're not resampling transitions in the real world, which is much more costly. For the inner expectation over the policy, a single-sample estimate suffices (for a minibatch of N states, there will be N actions sampled, one action for each state). To sample actions from the policy π_ϕ (and evaluate their log-likelihoods), you can call

```
actions, log_pis = policy(self._observations_ph)
```

which creates two tensors: an $N \times |\mathcal{A}|$ dimensional tensor containing actions sampled from the policy, and an N dimensional tensor containing the corresponding log-likelihoods.

Don't forget to incorporate the temperature factor α in the loss function!

Problem 1.3: Policy Loss

Next, you will need to implement the loss function for the policy in `_policy_loss_for`. The policy loss differs from the Q-function and value function losses in that it

involves the expectation under the policy distribution being optimized over. You have learned two ways to evaluate such a gradient: you can either use the REINFORCE gradient estimator (a.k.a. likelihood ratio method, or score function estimator) or reparameterization trick.

- *REINFORCE*: This is the gradient estimator used in policy gradient methods. The estimator is given by

$$\nabla_\phi J_\pi(\phi) = \mathbb{E}_{\mathbf{s}\sim\mathcal{D}}\left[\mathbb{E}_{\mathbf{a}\sim\pi_\phi(\mathbf{a}|\mathbf{s})}\left[\nabla_\phi \log \pi(\mathbf{a}|\mathbf{s})\left(\alpha \log \pi_\phi(\mathbf{a}|\mathbf{s}) - Q_\theta(\mathbf{s},\mathbf{a})\right) + b(\mathbf{s}) \mid \mathbf{s}\right]\right], \tag{12}$$

  where $b(\mathbf{s})$ is a baseline that you can choose freely. Again, you can estimate the outer expectation via a minibatch of N states from the replay buffer, and the inner expectation via a single-sample action from the policy. The expression above denotes the *gradient* of the loss function — to implement the loss function itself, you can define

```
policy_loss = ... log_pis * (tf.stop_gradient(...) ....
```

  Where the expression inside the `tf.stop_gradient` call is the target $\alpha \log \pi_\phi(\mathbf{a}|\mathbf{s}) - Q_\theta(\mathbf{s},\mathbf{a})$ — this prevents the gradients of $\alpha \log \pi_\phi(\mathbf{a}|\mathbf{s})$ from being considered in the policy update.

- *Reparameterization trick*: Suppose the policy can be expressed as $\mathbf{a} = f_\phi(\epsilon;\mathbf{s})$, where $\epsilon \sim \mathcal{N}(\mathbf{0},\boldsymbol{I})$. We can now rewrite the loss function in Equation 9 as follows:

$$J_\pi(\phi) = \mathbb{E}_{\mathbf{s}\sim\mathcal{D}}\left[\mathbb{E}_{\epsilon\sim\mathcal{N}(\mathbf{0},\boldsymbol{I})}\left[\alpha \log \pi_\phi(f_\phi(\epsilon;\mathbf{s})|\mathbf{s}) - Q_\theta(\mathbf{s}, f_\phi(\epsilon;\mathbf{s})) \mid \mathbf{s}\right]\right]. \tag{13}$$

  The expectation of the new loss function no longer depends on the policy parameters ϕ, but now the action $\mathbf{a} = f_\phi(\epsilon;\mathbf{s})$ is dependent on the policy parameters. Therefore, for the reparameterized loss we must make sure that the gradients of the sampled actions (w.r.t. the policy parameters) "flow" via backpropagation. By default, Tensorflow assumes that Gaussian distributions are reparameterized and will backpropagate the gradients of the actions. In the REINFORCE gradient estimator however, we treat the sampled actions as fixed constants, so we must disable the gradients of the policy from flowing through the actions. You will need to do this by adding a `tf.stop_gradient` op in `GaussianPolicy` right after sampling actions from the policy when reparameterization is disabled.

Task A: Look for

```
### Problem 1.3.A
### YOUR CODE HERE
```

in `sac.py` and replace it with your implementation of the REINFORCE policy loss. For now, ignore the input parameter `q_function2`, it will be relevant in a later part. The method should return a *surrogate loss*, whose gradient

with respect to the policy parameters is the REINFORCE gradient estimator. You must also make sure to treat the sampled actions as constants for the REINFORCE gradient estimator. Look for

```
### Problem 1.3.A
### YOUR CODE HERE
```

in `nn.py` and add a `tf.stop_gradient` op on the sampled actions.

Task B: Look for

```
### Problem 1.3.B
### YOUR CODE HERE
```

in `sac.py` and replace it with your implementation of the reparameterized policy loss. For now, ignore the input parameter `q_function2`, it will be relevant in a later part.

Bonus Task: Answer the following questions:

1. In Task A, what was your choice of baseline, and why did you choose it?
2. What are the pros and cons of the two types of gradient estimators?
3. Why can we not use the reparameterization trick with policy gradients?
4. We can minimize the policy loss in Equation 9 using off-policy data. Why is this not the case for standard actor-critic methods based on policy gradients, which require on-policy data?

Problem 2: Squashing

You will need to implement a "squashing" layer that applies tanh to actions before they are executed in the environment. We need to squash the actions, since typically the action space is bounded, whereas standard tractable policy distributions, for example Gaussian, are unbounded. Tanh is an invertible function from $\mathbb{R}$ to $(-1, 1)$, which makes it ideal for this purpose, as we will see soon. In the other words, a squashed Gaussian policy with state-dependent and learnable mean and variance can be implemented as $\mathbf{a} = \tanh\left(\mathbf{b}_\phi(\mathbf{s}) + \mathbf{A}_\phi(\mathbf{s})\epsilon\right)$, where $\epsilon \sim \mathcal{N}(\mathbf{0}, \boldsymbol{I})$, $\mathbf{b}$ is a learnable bias, and $\mathbf{A}$ is a learnable, full-rank scale matrix, typically diagonal with strictly positive diagonals. We can write this transformation as a composition of two functions (hiding dependency on $\mathbf{s}$ for simplicity): $\mathbf{a} = (f_2 \circ f_1)(\epsilon)$, where $\mathbf{z} = f_1(\epsilon) \triangleq b(\mathbf{s}) + A(\mathbf{s})\epsilon$ and $\mathbf{a} = f_2(\mathbf{z}) \triangleq \tanh(\mathbf{z})$.

Soft actor-critic requires that we evaluate the log-likelihood of actions. Since both f_1 and f_2 are invertible, we can apply the change of variables formula: For

any invertible functions $\mathbf{z}^{(i)} = f_i(\mathbf{z}^{(i-1)})$, we have the identity

$$\mathbf{z}^{(N)} = (f_N \circ \cdots \circ f_1)(\mathbf{z}^0) \;\Leftrightarrow\; \log p(\mathbf{z}^{(N)}) = \log p(\mathbf{z}^{(0)}) - \sum_{i=1}^{N} \log \left| \det \left(\frac{df_i(\mathbf{z}^{(i-1)})}{d\mathbf{z}^{(i-1)}} \right) \right| \tag{14}$$

where $\frac{df_i(\mathbf{z})}{d\mathbf{z}}$ is the Jacobian of f_i, and det is the determinant.

In our case for the tanh layer, since we apply tanh elementwise, the Jacobian is a diagonal matrix with $\frac{d\tanh(z_i)}{dz_i} = 1 - \tanh^2(z_i)$ on the diagonal. Thus, we get

$$\log \left| \det \left(\frac{df_2(\mathbf{z})}{d\mathbf{z}} \right) \right| = \sum_{i=1}^{|\mathcal{A}|} \log \left(1 - \tanh^2(z_i) \right). \tag{15}$$

In the following tasks, you will need to implement squashing in `nn.py`.

Task A: Look for

```
### Problem 2.A
### YOUR CODE HERE
```

in `nn.py` and replace it with your implementation of squashing. You will simply need to apply `tf.tanh` to `raw_actions`, which come from the Gaussian policy $\mathbf{b}_\phi(\mathbf{s}) + \mathbf{A}_\phi(\mathbf{s})\epsilon$.

Task B: Look for

```
### Problem 2.B
### YOUR CODE HERE
```

in `nn.py` The function takes in the raw actions $\mathbf{z}$ and should return the log-det-jac of $\tanh(\mathbf{z})$ given by Equation 15. Note how at the limit $z_i \to \pm\infty$, the argument to the log becomes close to zero, which can cause numerical instabilities. Adding a small positive constant can help mitigate this problem.

Bonus Task: Implement a numerically stable, exact version Equation 15 without adding a positive constant. Your implementation should only make use of multiplication, addition, and softplus (`tf.nn.softplus`). Hint 1: write tanh in terms of exponential function. Hint 2: $\text{softplus}(x) = \log(1 + \exp x)$.

Problem 3: SAC with Two Q-Functions

Value-based methods suffer from positive bias in the Q value updates. In the case of standard Q-learning, the source of this bias can be identified by applying

Jensen's inequality to the Bellman backup:

$$Q(\mathbf{s}, \mathbf{a}) \leftarrow r(\mathbf{s}, \mathbf{a}) + \gamma \max_{\mathbf{a}'} \mathbb{E}_{\mathbf{s}' \sim p(\mathbf{s}'|\mathbf{s},\mathbf{a})} \left[Q(\mathbf{s}', \mathbf{a}') \right] \tag{16}$$

$$\leq r(\mathbf{s}, \mathbf{a}) + \gamma \, \mathbb{E}_{\mathbf{s}' \sim p(\mathbf{s}'|\mathbf{s},\mathbf{a})} \left[\max_{\mathbf{a}'} Q(\mathbf{s}', \mathbf{a}') \right]. \tag{17}$$

In other words, the expectation of the sample estimator of the target Q-value is an upper bound of the true target Q-value. The bias accumulates when we keep applying the Bellman backup operator. A similar problem exists in maximum entropy reinforcement learning, although it is not as explicit as with standard Q-learning. In soft actor-critic, the positive bias is related to the policy improvement step, which corresponds to "soft" maximization of the Q values.

For discrete action spaces, we can mitigate the positive bias via double Q-learning [Hasselt, 2010], as you saw in Homework 3. We can also use similar techniques when the action space is continuous [Fujimoto et al., 2018]. To implement soft actor-critic with two Q-functions, you will need to

1. Construct two independent Q-functions, with parameters θ_1 and θ_2, and learn them by minimizing the loss in Equation 7. Use the same target value function $V_{\bar{\psi}}$ as a target for both Q-functions.
2. In the value function loss (Equation 8) and policy loss (Equation 9), replace $Q_\theta(\mathbf{s}, \mathbf{a})$ with $\min\{Q_{\theta_1}(\mathbf{s}, \mathbf{a}),\ Q_{\theta_2}(\mathbf{s}, \mathbf{a})\}$.

Task A: Implement SAC with two Q-functions. You will need to go back to `_value_function_loss_for` and `_policy_loss_for` in `sac.py`, and use the minimum of `q_function` and `q_function2` to calculate the Q values.

Problem 4: Experiments

Your implementation should now be complete, and you should be able to start running experiments.

Task A: Compare the REINFORCE and reparameterized policy gradients on HalfCheetah. To run REINFORCE, set the parameter `reparameterize` to `False` in `train_mujoco.py` and run

```
python train_mujoco.py --env_name HalfCheetah-v2 --exp_name
↪  reinf -e 3
```

To run reparameterized policy gradients, set the parameter `reparameterize` to `True` in `train_mujoco.py` and run

```
python train_mujoco.py --env_name HalfCheetah-v2 --exp_name
↪  reparam -e 3
```

Run both experiments for 500 epochs, over 3 seeds. Compare both experiments by plotting them (use `plot.py`). In a short paragraph, elaborate on your findings in a short paragraph.

Task B: Compare using one Q function versus two Q functions on HalfCheetah. For these experiments, set the parameter `reparameterize` to `True` in `train_mujoco.py`. To run one Q function, set the parameter `two_qf` to `False` in `train_mujoco.py` and run

```
python train_mujoco.py --env_name Ant-v2 --exp_name reparam -e 3
```

To run two Q functions, set the parameter `two_qf` to `True` in `train_mujoco.py` and run

```
python train_mujoco.py --env_name Ant-v2 --exp_name reparam_2qf
↪  -e 3
```

Run both experiments for 500 epochs, over 3 seeds. Compare both experiments by plotting them (use `plot.py`). In a short paragraph, elaborate on your findings.

Bonus Problem: The 1M (dollar!) Challenge

Experiment with different hyperparameters and the versions of SAC you have implement in this homework—or develop your own ideas. Train on 1M steps (1000 epochs). For starters, you can play around with the temperature of the policy. As mentioned in Section 3, the temperature parameter controls the amount of exploration: a higher temperature policy is more stochastic (exploratory), whereas a lower temperature policy is more deterministic. You can change the temperature by adjusting `alpha` parameter in `train_mujoco.py`. What is the best score you can reach on Ant-v2 in 1M environment steps (1000 epochs)? List the parameters you found worked the best and include a figure depicting at least three independent training runs for the chosen parameters.

6 Submission

Turn in both parts of the assignment on Gradescope as one submission. Upload the zip file with your code to **HW5 SAC Code**, and upload the PDF of your report to **HW5 SAC**.

References

Scott Fujimoto, Herke van Hoof, and Dave Meger. Addressing function approximation error in actor-critic methods. In *International Conference on Machine Learning*, 2018.

Tuomas Haarnoja, Haoran Tang, Pieter Abbeel, and Sergey Levine. Reinforcement learning with deep energy-based policies. In *International Conference on Machine Learning*, pages 1352–1361, 2017.

Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In *International Conference on Machine Learning*, 2018.

Hado V Hasselt. Double Q-learning. In *Advances in Neural Information Processing Systems*, pages 2613–2621, 2010.

Sergey Levine. Reinforcement learning and control as probabilistic inference: Tutorial and review. *arXiv preprint arXiv:1805.00909*, 2018.