Addison Wesley Data & Analytics Series

Foundations of Deep Reinforcement Learning

Theory and Practice in Python

Laura Graesser
Wah Loon Keng

Praise for *Foundations of Deep Reinforcement Learning*

"This book provides an accessible introduction to deep reinforcement learning covering the mathematical concepts behind popular algorithms as well as their practical implementation. I think the book will be a valuable resource for anyone looking to apply deep reinforcement learning in practice."
—*Volodymyr Mnih, lead developer of DQN*

"An excellent book to quickly develop expertise in the theory, language, and practical implementation of deep reinforcement learning algorithms. A limpid exposition which uses familiar notation; all the most recent techniques explained with concise, readable code, and not a page wasted in irrelevant detours: it is the perfect way to develop a solid foundation on the topic."
—*Vincent Vanhoucke, principal scientist, Google*

"As someone who spends their days trying to make deep reinforcement learning methods more useful for the general public, I can say that Laura and Keng's book is a welcome addition to the literature. It provides both a readable introduction to the fundamental concepts in reinforcement learning as well as intuitive explanations and code for many of the major algorithms in the field. I imagine this will become an invaluable resource for individuals interested in learning about deep reinforcement learning for years to come."
—*Arthur Juliani, senior machine learning engineer, Unity Technologies*

"Until now, the only way to get to grips with deep reinforcement learning was to slowly accumulate knowledge from dozens of different sources. Finally, we have a book bringing everything together in one place."
—*Matthew Rahtz, ML researcher, ETH Zürich*

Code 4.4 Replay: add experience

```python
# slm_lab/agent/memory/replay.py

class Replay(Memory):
    ...

    @lab_api
    def update(self, state, action, reward, next_state, done):
        ...
        self.add_experience(state, action, reward, next_state, done)

    def add_experience(self, state, action, reward, next_state, done):
        # Move head pointer. Wrap around if necessary
        self.head = (self.head + 1) % self.max_size
        self.states[self.head] = state.astype(np.float16)
        self.actions[self.head] = action
        self.rewards[self.head] = reward
        self.ns_buffer.append(next_state.astype(np.float16))
        self.dones[self.head] = done
        # Actually occupied size of memory
        if self.size < self.max_size:
            self.size += 1
        self.seen_size += 1
        algorithm = self.body.agent.algorithm
        algorithm.to_train = algorithm.to_train or (self.seen_size >
        ↪  algorithm.training_start_step and self.head %
        ↪  algorithm.training_frequency == 0)
```

Replay Memory Sample Sampling a batch involves sampling a set of valid indices and using these indices to extract the relevant examples from each of the memory storage lists to build a batch (see Code 4.5). First, `batch_size` indices are selected by calling the `sample_idxs` function (line 8). The indices are sampled random-uniformly with replacement from a list of indices $\in \{0, \ldots,$ `self.size`$\}$ (line 18). If `self.size == self.max_size`, the memory is full, and this corresponds to sampling indices from the entire memory store. The batch is then assembled using these indices (lines 9–14).

One further detail regarding sampling is worth mentioning. To save space in memory (RAM), the next states are not explicitly stored. They already exist in the state buffer, with the exception of the very latest next state. Consequently, assembling the batch of next states is a little more involved and is handled by the `sample_next_states` function called in line 12. Interested readers can look at the implementation in SLM Lab at `slm_lab/agent/memory/replay.py`.

Code 4.5 Replay: sample

```python
# slm_lab/agent/memory/replay.py

class Replay(Memory):
    ...

    @lab_api
    def sample(self):
        self.batch_idxs = self.sample_idxs(self.batch_size)
        batch = {}
        for k in self.data_keys:
            if k == 'next_states':
                batch[k] = sample_next_states(self.head, self.max_size,
                ↪   self.ns_idx_offset, self.batch_idxs, self.states,
                ↪   self.ns_buffer)
            else:
                batch[k] = util.batch_get(getattr(self, k), self.batch_idxs)
        return batch

    def sample_idxs(self, batch_size):
        batch_idxs = np.random.randint(self.size, size=batch_size)
        ...
        return batch_idxs
```

4.6 Training a DQN Agent

Code 4.6 is an example spec file which trains a DQN agent. The file is available in SLM Lab at `slm_lab/spec/benchmark/dqn/dqn_cartpole.json`.

Code 4.6 A DQN spec file

```json
# slm_lab/spec/benchmark/dqn/dqn_cartpole.json

{
  "vanilla_dqn_boltzmann_cartpole": {
    "agent": [{
      "name": "VanillaDQN",
      "algorithm": {
        "name": "VanillaDQN",
        "action_pdtype": "Categorical",
        "action_policy": "boltzmann",
        "explore_var_spec": {
          "name": "linear_decay",
          "start_val": 5.0,
          "end_val": 0.5,
          "start_step": 0,
          "end_step": 4000,
        },
        "gamma": 0.99,
        "training_batch_iter": 8,
        "training_iter": 4,
        "training_frequency": 4,
        "training_start_step": 32
      },
      "memory": {
        "name": "Replay",
        "batch_size": 32,
        "max_size": 10000,
        "use_cer": false
      },
      "net": {
        "type": "MLPNet",
        "hid_layers": [64],
        "hid_layers_activation": "selu",
        "clip_grad_val": 0.5,
        "loss_spec": {
          "name": "MSELoss"
        },
        "optim_spec": {
          "name": "Adam",
          "lr": 0.01
        },
        "lr_scheduler_spec": {
          "name": "LinearToZero",
          "frame": 10000
        },
        "gpu": false
      }
    }],
    "env": [{
      "name": "CartPole-v0",
      "max_t": null,
      "max_frame": 10000
    }],
    "body": {
      "product": "outer",
      "num": 1
```

```
57      },
58      "meta": {
59        "distributed": false,
60        "eval_frequency": 500,
61        "max_session": 4,
62        "max_trial": 1
63      },
64      ...
65    }
66  }
```

As with SARSA, there are several components. All line numbers refer to Code 4.6.

- **Algorithm:** The algorithm is `"VanillaDQN"` (line 8), to distinguish from `"DQN"` with target network from Chapter 5. The action policy is the Boltzmann policy (lines 9–10) with linear decay (lines 11–17) of the temperature parameter τ known as exploration variable `explore_var` (line 11). γ is set on line 18.
- **Network architecture:** Multilayer perceptron with one hidden layer of 64 units (line 32) and SeLU activation functions (line 33).
- **Optimizer:** The optimizer is Adam [68] with a learning rate of 0.01 (lines 38–41). The learning rate is specified to decay linearly to zero over the course of training (lines 42–45).
- **Training frequency:** Training starts after the agent has made 32 steps in the environment (line 22) and occurs every 4 steps from then on (line 21). At each training step, 4 batches are sampled from the `Replay` memory (line 20) and used to make 8 parameter updates (line 19). Each batch has 32 elements (line 26).
- **Memory:** A maximum of 10,000 most recent experiences are stored in the `Replay` memory (line 27).
- **Environment:** The environment is OpenAI Gym's CartPole [18] (line 50).
- **Training length:** Training consists of 10,000 time steps (line 52).
- **Checkpointing:** The agent is evaluated every 500 time steps (line 60).

When training with Boltzmann policies, it is important to set `action_pdtype` to `Categorical` so that the agent samples actions from the categorical probability distribution generated by the Boltzmann action policy. This is different to ε-greedy policies which have an argmax "distribution." That is, the agent will select the action with the maximum Q-value with probability 1.0.

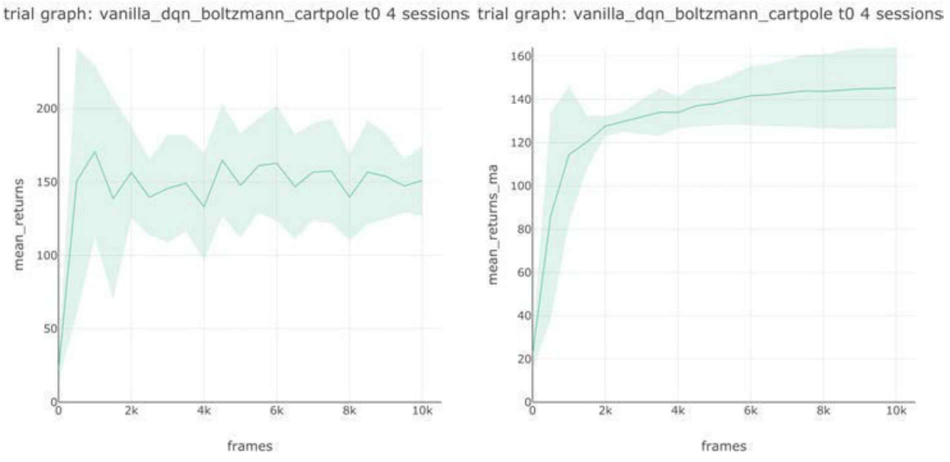
Notice also that the maximum value for τ in the Boltzmann policy (line 14) is not constrained to 1 as it is for ε-greedy policies. The exploration variable in this case, τ, does not represent a probability but a temperature, so it can take any positive value. However, it should never be zero, as this will lead to division-by-0 numerical errors.

To train this DQN agent using SLM Lab, run the commands shown in Code 4.7 in a terminal.

Code 4.7 Training a DQN agent to play CartPole

```
conda activate lab
python run_lab.py slm_lab/spec/benchmark/dqn/dqn_cartpole.json
↪  vanilla_dqn_boltzmann_cartpole train
```

This will use the spec file to run a training `Trial` with four `Session`s to obtain an average result and plot the trial graphs shown in Figure 4.2.



(a) Trial graph (b) Trial graph with moving average

Figure 4.2 DQN trial graphs from SLM Lab averaged over four sessions. The vertical axis shows the total rewards (`mean_return` for evaluation is computed without discount) averaged over eight episodes during checkpoints, and the horizontal axis shows the total training frames. The graph on the right is a moving average with a window of 100 evaluation checkpoints.

4.7 Experimental Results

This section will look at how changing the neural network architecture affects the performance of DQN on CartPole. We will use the experimentation feature of SLM Lab to perform a grid search over network architecture parameters.

4.7.1 Experiment: The Effect of Network Architecture

The architecture of neural networks affects their capability to approximate functions. In this experiment, we will perform a simple grid search over a number of hidden layer configurations, then plot and compare their performance. The experiment spec file, extended from Code 4.6, is shown in Code 4.8. Lines 8–15 specify a grid search over the architecture of the Q-network's hidden layers `net.hid_layers`. The full spec file is available in SLM Lab at `slm_lab/spec/benchmark/dqn/dqn_cartpole.json`.

Code 4.8 DQN spec file with search spec over different network architectures

```
# slm_lab/spec/benchmark/dqn/dqn_cartpole.json

{
  "vanilla_dqn_boltzmann_cartpole": {
    ...
    "search": {
      "agent": [{
        "net": {
          "hid_layers__grid_search": [
            [32],
            [64],
            [32, 16],
            [64, 32]
          ]
        }
      }]
    }
  }
}
```

Code 4.9 shows the command to run this experiment. Note that we are using the same spec file as training, but now we replace the mode `train` with `search`.

Code 4.9 Run an experiment to search over different network architectures as defined in the spec file.

```
conda activate lab
python run_lab.py slm_lab/spec/benchmark/dqn/dqn_cartpole.json
↪  vanilla_dqn_boltzmann_cartpole search
```

This will run an `Experiment` which spawns four `Trials` by substituting different values of `net.hid_layers` into the original spec of DQN with a Boltzmann policy. Each `Trial` runs four repeated `Sessions` to obtain an average. Figure 4.3 shows the multitrial graphs plotted from this experiment.

Figure 4.3 shows that networks with two hidden layers (trials 2 and 3) perform better than networks with a single layer (trials 0 and 1) thanks to their increased learning capacity. When the number of layers is the same, networks with fewer units (trials 0 and 2),