



Student data

| Matrikelnummer | Firstname | Lastname |
|----------------|-----------|----------|
| 12046988 | Alexander | Pichler |
| 11809898 | Alexander | Wieser |
| 12425725 | Urban | Klobcic |

1 Task 1: GridWorld Environment Implementation

In Task 1, a simple GridWorld environment was implemented using JAX. This environment serves as the foundation for the reinforcement learning agents developed in Tasks 2 and 3. The main goal was to design a minimal but fully functional environment that is compatible with `jax.jit` and follows a functional programming style, as required by the assignment.

1.1 Environment Design

The GridWorld consists of a square grid of configurable size `size × size`. The environment state is defined by three components:

- the agent position $(x_{\text{agent}}, y_{\text{agent}})$,
- the target position $(x_{\text{target}}, y_{\text{target}})$,
- the current direction of the agent.

The direction is encoded as an array of arrays:

```
self.direction_to_move = jnp.array(  
    [  
        [1, 0],  # right  
        [0, 1],  # down  
        [-1, 0], # left  
        [0, -1], # up  
    ],  
    dtype=jnp.int32,  
)
```

The state is stored in a `NamedTuple (GridWorldState)`.

The observation returned to the agent is a dictionary containing the agent position, target position, and direction. This structure is used consistently in all later tasks.

1.2 Action Space and Dynamics

The action space is discrete with four possible actions:

- 0: no operation,

- 1: move forward in the current direction,
- 2: rotate left,
- 3: rotate right.

Movement is implemented using a direction-to-movement lookup table, which maps the current direction to a step vector. When moving forward, the new position is clipped using `jnp.clip` to ensure that the agent always remains within the grid boundaries. This avoids invalid states and removes the need for conditional branching.

Rotations are handled using modulo arithmetic, which guarantees that the direction index always stays within the valid range $\{0, 1, 2, 3\}$.

1.3 Reset Function

The `reset` function initializes a new episode. Both the agent position and the target position are sampled uniformly at random using `jax.random.randint`. If the sampled positions are identical, the target position is shifted by $+1 \pmod{\text{size}}$ to ensure that the episode does not terminate immediately.

The initial direction of the agent is also sampled uniformly at random. The function returns both the initial observation and the internal environment state.

All randomness is handled explicitly via PRNG keys, which is required for JAX compatibility.

1.4 Step Function, Reward, and Termination

The `step` function applies the selected action to the current state and returns:

- the new observation,
- the updated state,
- the reward,
- a boolean `done` flag,
- an auxiliary `info` dictionary.

An episode terminates when the agent reaches the target cell. In this case, a reward of 1.0 is returned; otherwise, the reward is 0.0. No additional step penalty is used, which simplifies the learning problem in the later tasks.

As additional diagnostic information, the Manhattan distance between the agent and the target is provided in the `info` dictionary. This value is not used for learning but is helpful for debugging and evaluation.

1.5 JAX Compatibility

Special care was taken to avoid Python control flow inside the environment dynamics. Conditional behavior (e.g., action handling and target correction) is implemented using `jax.lax.cond`, which ensures that the environment functions can be safely JIT-compiled.

The environment was tested by applying random actions in a loop, with the `step` function wrapped using `jax.jit`, confirming correct behavior and compatibility with later reinforcement learning algorithms. Additionally, another python script (`demo.py`) was written to evaluate if the world operates as specified.

2 Task 2

2.1 Index Q-Table

Equation 2.1 describes how our state should look like according to the assignment:

$$S = (x_{\text{agent}}, y_{\text{agent}}, d_{\text{agent}}, x_{\text{target}}, y_{\text{target}}) \quad (2.1)$$

As stated in the assignment, the state therefore has size $|S| = \text{size}^4 \cdot 4$. Considering the dimensions of the individual components in our state, we have 4 coordinate components that are limited by the grid size and the direction limited

by the number of directions that we can turn to. Therefore, our Q-Table corresponds to a $((size \times size \times 4 \times size \times size) \times actions)$ -dimensional matrix. Similar to indexing of flattened matrices with less dimensions, we can multiply the index for each state component (which will be the value of the component itself in our case) by the size of the "lower" dimensions. Repeating this step for each dimension of the state and summing up results in the following formula for the index:

$$S = size \quad (2.2)$$

$$D = 4 \quad (2.3)$$

$$idx = x_a * (S * D * S * S) + y_a * (D * S * S) + d_a * (S * S) + x_t * S + y_t \quad (2.4)$$

Remark 2.1 In the code, the dimension sizes are both set via the environment.

For the initial values, we chose small (in relation to the reward of the target) random numbers in the range [0.001, 0.1] to avoid being stuck in the initial position for long. Additionally, we increased ϵ as this combination yielded the best results. For larger gridsizes it was also necessary to increase the number of training steps, at the cost of a longer training time.

2.2 Action selection

The `_epsilon_greedy` function calculates the next state from the current Q-Table (by calculating the `argmax` over the row determined by the current state) and a random action. Then the actual action is chosen to be the random one with probability given by the exploration constant ϵ . The assignment hints us to calculate it in this way such that we do not introduce python control flow structures into the `jit` compiled code.

2.3 Q-Learning Update

To update the table we use the Temporal Difference update rule for Q-Learning that is stated in the assignment. Additionally, in the comments we are instructed to multiply with `1-done`, again to avoid branches in our control flow for the case distinction if we already reached the target.

2.4 Training Loop

The training loop is given in the function `_train_steps` and already implemented in the template. To make it run performantly, it is set up to be `jax.jit` compiled for a static number of iterations. These iterations correspond to the total number of steps in all epochs rather than steps for a single epoch. The loop itself is implemented as `jax.lax.scan` to be compatible with jax. Inside the loop the following steps are executed:

1. Generation of random keys
2. Choosing the next action according to the epsilon greedy algorithm
3. Applying the action to the environment and thus taking a step
4. Updating/learning the new Q values
5. If an epoch is done, we reset the environment before taking the next step, otherwise we continue at the state resulting from the step we took earlier.

2.5 Save/Load Q-Table

Saving and loading is also handled in the template. It is implemented by converting the `jax` arrays to `numpy` arrays (vice versa) and storing/loading them via the builtin `numpy` functionality.

3 Task 3

3.1 Computation of G_t

We compute the discounted cumulative reward at time step t according to the recursive relationship

$$G_t = r_t + \gamma G_{t+1}$$

where r_t is the reward at step t and γ is the discount factor.

We use `jax.lax.scan` with `reverse=True` to compute the rewards backwards as shown in Listing 1.

```

1 def _returns(self, rewards):
2     def step(g_t1, r_t):
3         g_t = r_t + self.gamma * g_t1
4         return g_t, g_t # (carry, return value)
5
6     return jax.lax.scan(step, 0.0, rewards, reverse=True)[1]
```

Listing 1: Function `_returns` to calculate cumulative rewards

3.2 REINFORCE loss

The objective is to maximize expected return:

$$J(\theta) = \mathbb{E}_{\pi_\theta} \left[\sum_{t=0}^{T-1} \gamma^t r_t \right].$$

Using the Policy Gradient Theorem, the gradient is

$$\nabla_\theta J(\theta) \approx \sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t | s_t) G_t.$$

Therefore, the loss function we minimize is

$$\mathcal{L}_{\text{reinforce}} = - \sum_{t=0}^{T-1} \log \pi_\theta(a_t | s_t) G_t M_t$$

where M is a binary mask that indicates whether the episode is active at each time step. We don't want any steps after reaching the target to influence the loss. The calculation is implemented as follows:

```
1 loss = -jnp.sum(logps * jax.lax.stop_gradient(returns) * actives)
```

3.3 Usage of `jax.jit` and `jax.lax.scan`

We use `jax.jit` to just-in-time compile function `_train_n_episodes`. This produces optimized machine code to speed up training compared to the Python interpreter.

The function `jax.lax.scan` takes a function f and repeatedly executes it. The function f should be pure and take `carry` as an input. Further, it should return `(carry, result)` where `carry` is passed as an input to the next call of f and the `results` from all calls are collected. We use it instead of slow Python `for` loops in:

- `_returns` as explained in Section 3.1,
- `_rollout` as explained in Section 3.4,
- `_train_n_episodes` to call `_train_one_episode` for each episode.

3.4 Implementation of `_rollout`

The `_rollout` function generates a full episode trajectory. It uses `jax.lax.scan` to execute function `step_fn` for `max_steps` times.

The function `step_fn` computes a single step in the trajectory as follows:

- Converts the current state to a feature vector

$$[\tilde{x}_{\text{agent}}, \tilde{y}_{\text{agent}}, \tilde{x}_{\text{target}}, \tilde{y}_{\text{target}}, d_0, d_1, d_2, d_3]$$

where coordinates are normalized to $[0, 1]$ (e.g., $\tilde{x} = x/(S-1)$) and d_i represents the one-hot encoded direction.

- Feeds the feature vector to the policy model and get a distribution $\pi(a_t | s_t)$ as output.
- Samples the next action $a_t \leftarrow \pi(a_t | s_t)$.
- Runs `env.step` to get the next state and reward from the GridWorld environment.
- Records (`logp`, `reward`, `active`) for training.

3.5 Implementation of `_train_one_episode`

The function `_train_one_episode` does the following:

- Resets the environment.
- Generates (`logp`, `reward`, `active`) for each step of a full trajectory using `_rollout`.
- Uses JAX auto differentiation `jax.value_and_grad` to compute gradients from the loss function explained in Section 3.2.
- Applies the gradients to the model parameters using `optax` optimizer.
- Returns an updated training state and metrics (loss, episode return).