`mfpy`

Model-free learning in Python

Advantages and Disadvantages

- Advantages:
  - Very easy to create new representations, algorithms, tasks or policies through inheritance and polymorphism
  - Very easy to debug and to add new functionality
- Disadvantages:
  - Simplicity may make the package not competitive with state-of-the-art implementations

Summary of Current Package

- General framework for model-free tasks
- Value function representations
  - Tabular (hash-table)
  - Deep (keras)
- Learning algorithms
  - Monte-Carlo
  - Q-Learning, Deep-Q, Double Deep-Q
  - Sarsa, Expected Sarsa
  - Sarsa-Lambda
- Exploration policies
  - E.g. epsilon-greedy, Boltzmann, pursuit

How Training Loops are Implemented

Pseudocode

Initialize $Q(s,a)$ arbitrarily
Repeat (for each episode):
  Initialize s
  Repeat (for each step of episode):
    Choose a from s using policy derived from Q (e.g., ε-greedy)
    Take action a, observe r, s'
    $Q(s,a) \leftarrow Q(s,a) + \alpha\big[r + \gamma \max_{a'} Q(s',a') - Q(s,a)\big]$
    $s \leftarrow s'$;
  until s is terminal

Figure 6.12: Q-learning: An off-policy TD control algorithm.

Actual Python Code

```
# initialize state
state = task.initial_state()

# repeat for each step of episode
for t in range(self.episode_length):

    # choose action from state using policy derived from Q
    action = policy.act(Q, task, state)

    # take action and observe reward and new state
    new_state, reward, done = task.transition(state, action)
    rewards[t] = reward

    # update Q
    delta = reward + self.gamma * Q.max_value(new_state) -
              Q.values(state)[action]
    Q.update(state, action, delta)

    # update state
    state = new_state

    # until state is terminal
    if done:
       break
```

Example – Creating a Task

Three methods are to be implemented:

```
initial_state() : state
valid_actions() : int
transition(state, action) : (state, float, bool)
```

Example – Creating a Task

```
from domains.Task import Task

LEFT, UP, RIGHT, DOWN = 0, 1, 2, 3

class Gridworld(Task):

    def __init__(self, maze, initial):
        super().__init__()
        self.initial = initial
        self.maze = maze
        self.height, self.width = maze.shape

    def initial_state(self):
        return self.initial

    def valid_actions(self):
        return 4

   def check(self, row, col, action):
        if (row == 0 and action == UP) or
(row == self.height - 1 and action == DOWN) or \
            (col == 0 and action == LEFT) or \
            (col == self.width - 1 and action ==
RIGHT): return False
        return True
```

```
def transition(self, state, action):

    # perform the movement
    row, col = state
    if self.check(row, col, action):
        if action == LEFT:
            col -= 1
        elif action == UP:
            row -= 1
        elif action == RIGHT:
            col += 1
        elif action == DOWN:
            row += 1
    else:
        return (row, col), -0.1, False

    # compute the new state, status and reward
    grid_x1y1 = self.maze[row, col]
    if grid_x1y1 > 0:
        return (row, col), 50, True
    else:
        return (row, col), -0.1, False
```

Example – Running an Experiment

```
# initialize a cliff-walking domain
maze = np.zeros((4, 20), dtype=np.int32)
maze[3, 1:-1] = -1.0
maze[3, -1] = 1.0
domain = CliffWalking(maze, (3, 0))

# initialize the agent
agent = Tabular(domain.valid_actions(), 0.18)

# initialize the policy
policy = EpsilonGreedy(0.1)

# initialize the algorithm
trainer = ExpectedSarsa(1.0, 200)

# run the experiment 50 times and take the average
lengths, returns = trainer.train_many(agent, domain, policy,
                                      episodes=500, trials=50)
for l in returns: print(l)
```

Future Developments

- Automated tuning of hyper-parameters using various criteria (e.g. final return, area under curve, etc.)
- Automated tools for plot generation
- Decorators to allow adding custom functionality and behavior to existing training algorithms
- Policy algorithms (A3C, policy gradient, DDPG, etc.)
- Object hierarchy for replay (e.g. prioritized replay, etc.)
- Deep RL works only with sequential Keras model (e.g. feedforward, CNN) – extend this to any tensor-flow based model