# multiarmed_bandits

Perform a parameter study of various approaches to solving multiarmed bandits. For every hyperparameter choice, perform 100 episodes, each consisting of 1000 trials, and report the average and standard deviation of the 100 episode returns.

Start with the multiarmed_bandits.py (https://github.com/ufal/npfl122/tree/past-1920/labs/01/multiarmed_bandits.py) template, which defines `MultiArmedBandits` environment. We use API based on OpenAI Gym (https://gym.openai.com/) `Environment` class, notably the following two methods:

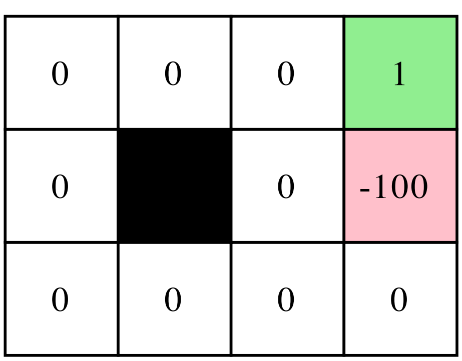
- `reset() → new_state` : starts a new episode
- `step(action) → new_state, reward, done, info` : perform the chosen action in the environment, returning the new state, obtained reward, a boolean flag indicating an end of episode, and additional environment-specific information Of course, the states are not used by the multiarmed bandits ( `None` is returned).

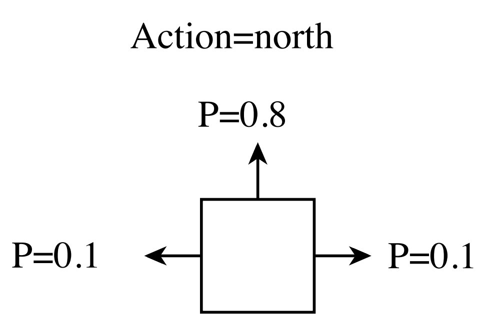
Your goal is to implement the following modes of calculation. You should use multiarmed_bandits_draw.py (https://github.com/ufal/npfl122/tree/past-1920/labs/01/multiarmed_bandits_draw.py) to plots the results in a graph.

- `greedy` *[2 points]*: perform $\varepsilon$-greedy search with parameter `epsilon` , computing the value function using averaging. (Results for $\varepsilon \in \{1/64, 1/32, 1/16, 1/8, 1/4\}$ are plotted.)
- `greedy` and `alpha` $\neq 0$ *[1 point]*: perform $\varepsilon$-greedy search with parameter `epsilon` and initial function estimate of 0, using fixed learning rate `alpha` . (Results for $\alpha = 0.15$ and $\varepsilon \in \{1/64, 1/32, 1/16, 1/8, 1/4\}$ are plotted.)
- `greedy` , `alpha` $\neq 0$ and `initial` $\neq 0$ *[1 point]*: perform $\varepsilon$-greedy search with parameter `epsilon` , given `initial` value as starting value function and fixed learning rate `alpha` . (Results for `initial` $= 1$, $\alpha = 0.15$ and $\varepsilon \in \{1/128, 1/64, 1/32, 1/16\}$ are plotted.)
- `ucb` *[2 points]*: perform UCB search with confidence level `c` and computing the value function using averaging. (Results for $c \in \{1/4, 1/2, 1, 2, 4\}$ are plotted.)
- `gradient` *[2 points]*: choose actions according to softmax distribution, updating the parameters using SGD to maximize expected reward. (Results for $\alpha \in \{1/16, 1/8, 1/4, 1/2\}$ are plotted.)

# policy_iteration

Consider the following gridworld:



Start with policy_iteration.py (https://github.com/ufal/npfl122/tree/past-1920/labs/02/policy_iteration.py), which implements the gridworld mechanics, by providing the following methods:

- `GridWorld.states` : return number of states ( `11` )
- `GridWorld.actions` : return lists with labels of the actions ( `["↑", "→", "↓", "←"]` )
- `GridWorld.step(state, action)` : return possible outcomes of performing the `action` in a given `state` , as a list of triples containing
  - `probability` : probability of the outcome
  - `reward` : reward of the outcome
  - `new_state` : new state of the outcome

Implement policy iteration algorithm, with `--steps` steps of policy evaluation/policy improvement. During policy evaluation, use the current value function and perform `--iterations` applications of the Bellman equation. Perform the policy evaluation synchronously (i.e., do not overwrite the current value function when computing its improvement). Assume the initial policy is "go North" and initial value function is zero.

After given number of steps and iterations, print the resulting value function and resulting policy. For example, the output after 4 steps and 4 iterations should be:

```
    9.15→   10.30→   11.32→   12.33↑
    8.12↑             3.35←    2.58←
    6.95↑    5.90←    4.66←   -4.93↓
```

# monte_carlo

Solve the CartPole-v1 environment (https://gym.openai.com/envs/CartPole-v1) environment from the OpenAI Gym (https://gym.openai.com/) using the Monte Carlo reinforcement learning algorithm.

Use the supplied cart_pole_evaluator.py (https://github.com/ufal/npfl122/tree/past-1920/labs/02/cart_pole_evaluator.py) module (depending on gym_evaluator.py (https://github.com/ufal/npfl122/tree/past-1920/labs/02/gym_evaluator.py)) to interact with the discretized environment. The environment has the following methods and properties:

- `states` : number of states of the environment
- `actions` : number of actions of the environment
- `episode` : number of the current episode (zero-based)
- `reset(start_evaluate=False) → new_state` : starts a new episode
- `step(action) → new_state, reward, done, info` : perform the chosen action in the environment, returning the new state, obtained reward, a boolean flag indicating an end of episode, and additional environment-specific information
- `render()` : render current environment state

Once you finish training (which you indicate by passing `start_evaluate=True to reset` ), your goal is to reach an average return of 490 during 100 evaluation episodes. Note that the environment prints your 100-episode average return each 10 episodes even during training.

You can start with the monte_carlo.py (https://github.com/ufal/npfl122/tree/past-1920/labs/02/monte_carlo.py) template, which parses several useful parameters, creates the environment and illustrates the overall usage.

# importance_sampling

Using the FrozenLake-v0 environment (https://gym.openai.com/envs/FrozenLake-v0) environment, implement Monte Carlo weighted importance sampling to estimate state value function $V$ of target policy, which uniformly chooses either action 1 (down) or action 2 (right), utilizing behaviour policy, which uniformly chooses among all four actions.

Start with the importance_sampling.py (https://github.com/ufal/npfl122/tree/past-1920/labs/03/importance_sampling.py) template, which creates the environment and generates episodes according to behaviour policy.

For $1000$ episodes, the output of your program should be the following:

```
0.00  0.00  0.00  0.00
0.00  0.00  0.00  0.00
0.00  0.00  0.21  0.00
0.00  0.00  0.45  0.00
```

# q_learning

Solve the MountainCar-v0 environment (https://gym.openai.com/envs/MountainCar-v0) environment from the OpenAI Gym (https://gym.openai.com/) using the Q-learning reinforcement learning algorithm. Note that this task does not require TensorFlow.

Use the supplied mountain_car_evaluator.py (https://github.com/ufal/npfl122/tree/past-1920/labs/03/mountain_car_evaluator.py) module (depending on gym_evaluator.py (https://github.com/ufal/npfl122/tree/past-1920/labs/02/gym_evaluator.py)) to interact with the discretized environment. The environment methods and properties are described in the `monte_carlo` assignment. Your goal is to reach an average reward of -150 during 100 evaluation episodes.

You can start with the q_learning.py (https://github.com/ufal/npfl122/tree/past-1920/labs/03/q_learning.py) template, which parses several useful parameters, creates the environment and illustrates the overall usage. Note that setting hyperparameters of Q-learning is a bit tricky – I usualy start with a larger value of $\varepsilon$ (like 0.2 or even 0.5) an then gradually decrease it to almost zero.

# lunar_lander

Solve the LunarLander-v2 environment (https://gym.openai.com/envs/LunarLander-v2) environment from the OpenAI Gym (https://gym.openai.com/). Note that this task does not require TensorFlow.

Use the supplied lunar_lander_evaluator.py (https://github.com/ufal/npfl122/tree/past-1920/labs/03/lunar_lander_evaluator.py) module (depending on gym_evaluator.py (https://github.com/ufal/npfl122/tree/past-1920/labs/02/gym_evaluator.py) to interact with the discretized environment. The environment methods and properties are described in the `monte_carlo` assignment, but include one additional method:

- `expert_trajectory() → initial_state, trajectory` This method generates one expert trajectory and returns a pair of `initial_state` and `trajectory`, where `trajectory` is a list of the tripples *(action, reward, next_state)*. You can use this method only during training, **not during evaluation**.

You can start with the lunar_lander.py (https://github.com/ufal/npfl122/tree/past-1920/labs/03/lunar_lander.py) template, which parses several useful parameters, creates the environment and illustrates the overall usage.

# q_learning_tiles

Improve the `q_learning` task performance on the MountainCar-v0 environment (https://gym.openai.com/envs/MountainCar-v0) environment using linear function approximation with tile coding. Your goal is to reach an average reward of -110 during 100 evaluation episodes.

Use the mountain_car_evaluator.py (https://github.com/ufal/npfl122/tree/past-1920/labs/03/mountain_car_evaluator.py) module (depending on gym_evaluator.py (https://github.com/ufal/npfl122/tree/past-1920/labs/02/gym_evaluator.py)) to interact with the discretized environment. The environment methods and properties are described in the `monte_carlo` assignment, with the following changes:

- The `env.weights` method return the number of weights of the linear function approximation.
- The `state` returned by the `env.step` method is a *list* containing weight indices of the current state (i.e., the feature vector of the state consists of zeros and ones, and only the indices of the ones are returned). The (action-)value function for a state is therefore approximated as a sum of the weights whose indices are returned by `env.step`.

  The default number of tiles in tile encoding (i.e., the size of the list with weight indices) is `args.tiles=8`, but you can use any number you want (but the assignment is solvable with 8).

You can start with the q_learning_tiles.py (https://github.com/ufal/npfl122/tree/past-1920/labs/05/q_learning_tiles.py) template, which parses several useful parameters, creates the environment and illustrates the overall usage. Implementing Q-learning is enough to pass the assignment, even if both N-step Sarsa and Tree Backup converge a little faster.

# q_network

Solve the CartPole-v1 environment (https://gym.openai.com/envs/CartPole-v1) environment from the OpenAI Gym (https://gym.openai.com/) using Q-learning with neural network as a function approximation.

The cart_pole_evaluator.py (https://github.com/ufal/npfl122/tree/past-1920/labs/02/cart_pole_evaluator.py) module (depending on gym_evaluator.py (https://github.com/ufal/npfl122/tree/past-1920/labs/02/gym_evaluator.py)) can also create a continuous environment using `environment(discrete=False)`. The continuous environment is very similar to the discrete environment, except that the states are vectors of real-valued observations with shape `environment.state_shape`.

Use Q-learning with neural network as a function approximation, which for a given states returns state-action values for all actions. You can use any network architecture, but one hidden layer of 20 ReLU units is a good start.

Your goal is to reach an average return of 400 during 100 evaluation episodes.

You can start with the q_network.py (https://github.com/ufal/npfl122/tree/past-1920/labs/05/q_network.py) template, which provides a simple network implementation in TensorFlow.

# car_racing

The goal of this competition is to use Deep Q Networks and its improvements on a more real-world CarRacing-v0 environment (https://gym.openai.com/envs/CarRacing-v0) environment from the OpenAI Gym (https://gym.openai.com/).

Use the supplied car_racing_evaluator.py (https://github.com/ufal/npfl122/tree/past-1920/labs/06/car_racing_evaluator.py) module (depending on gym_evaluator.py (https://github.com/ufal/npfl122/tree/past-1920/labs/06/gym_evaluator.py) to interact with the environment. The environment is continuous and states are RGB images of size $96 \times 96 \times 3$, but you can downsample them even more. The actions are also continuous and consist of an array with the following three elements:

- `steer` in range [-1, 1]
- `gas` in range [0, 1]
- `brake` in range [0, 1]

Internally you should generate discrete actions and convert them to the required representation before the `step` call. Good initial action space is to use 9 actions – a Cartesian product of 3 steering actions (left/right/none) and 3 driving actions (gas/brake/none).

The car_racing.py (https://github.com/ufal/npfl122/tree/past-1920/labs/06/car_racing.py) template parses several useful parameters and creates the environment. Note that the car_racing_evaluator.py (https://github.com/ufal/npfl122/tree/past-1920/labs/06/car_racing_evaluator.py) can be executed directly and in that case you can drive the car using arrows.

# reinforce

Solve the CartPole-v1 environment (https://gym.openai.com/envs/CartPole-v1) environment from the OpenAI Gym (https://gym.openai.com/) using the REINFORCE algorithm.

The supplied cart_pole_evaluator.py (https://github.com/ufal/npfl122/tree/past-1920/labs/07/cart_pole_evaluator.py) module (depending on gym_evaluator.py (https://github.com/ufal/npfl122/tree/past-1920/labs/07/gym_evaluator.py)) can create a continuous environment using `environment(discrete=False)`. The continuous environment is very similar to the discrete environment, except that the states are vectors of real-valued observations with shape `environment.state_shape`.

Your goal is to reach an average return of 490 during 100 evaluation episodes.

You can start with the reinforce.py (https://github.com/ufal/npfl122/tree/past-1920/labs/07/reinforce.py) template, which provides a simple network implementation in TensorFlow.

# reinforce_baseline

This is a continuation of `reinforce` assignment.

Using the reinforce_baseline.py (https://github.com/ufal/npfl122/tree/past-1920/labs/07/reinforce_baseline.py) template, solve the CartPole-v1 environment (https://gym.openai.com/envs/CartPole-v1) environment using the REINFORCE with baseline algorithm.

Your goal is to reach an average return of 490 during 100 evaluation episodes.

# cart_pole_pixels

The supplied cart_pole_pixels_evaluator.py (https://github.com/ufal/npfl122/tree/past-1920/labs/07/cart_pole_pixels_evaluator.py) module (depending on gym_evaluator.py (https://github.com/ufal/npfl122/tree/past-1920/labs/02/gym_evaluator.py)) generates a pixel representation of the `CartPole` environment as an $80 \times 80$ image with three channels, with each channel representing one time step (i.e., the current observation and the two previous ones).

The cart_pole_pixels.py (https://github.com/ufal/npfl122/tree/past-1920/labs/07/cart_pole_pixels.py) template parses several parameters and creates the environment. You are again supposed to train the model beforehand and submit only the trained neural network.

# paac

Using the paac.py (https://github.com/ufal/npfl122/tree/past-1920/labs/08/paac.py) template, solve the CartPole-v1 environment (https://gym.openai.com/envs/CartPole-v1) environment using parallel actor-critic algorithm. Use the `parallel_init` and `parallel_step` methods described in `car_racing` assignment.

Your goal is to reach an average return of 450 during 100 evaluation episodes.

# paac_continuous

Using the paac_continuous.py (https://github.com/ufal/npfl122/tree/past-1920/labs/08/paac_continuous.py) template, solve the MountainCarContinuous-v0 environment (https://gym.openai.com/envs/MountainCarContinuous-v0/) environment using parallel actor-critic algorithm with continuous actions.

The `gym_environment` now provides two additional methods:

- `action_shape` : returns required shape of continuous action. You can assume the actions are always an one-dimensional vector.
- `action_ranges` : returns a pair of vectors `low` , `high` . These denote valid ranges for the actions, so `low[i]` $\leq$ `action[i]` $\leq$ `high[i]` .

Your goal is to reach an average return of 90 during 100 evaluation episodes.

# ddpg

Using the ddpg.py (https://github.com/ufal/npfl122/tree/past-1920/labs/08/ddpg.py) template, solve the Pendulum-v0 environment (https://gym.openai.com/envs/Pendulum-v0) environment using deep deterministic policy gradient algorithm.

To create the evaluator, use gym_evaluator.py (https://github.com/ufal/npfl122/tree/past-1920/labs/08/gym_evaluator.py) `.GymEvaluator("Pendulum-v0")` . The environment is continuous, states and actions are described at OpenAI Gym Wiki (https://github.com/openai/gym/wiki/Pendulum-v0).

Your goal is to reach an average return of -200 during 100 evaluation episodes.

# walker

In this exercise exploring continuous robot control, try solving the BipedalWalker-v2 environment (https://gym.openai.com/envs/BipedalWalker-v2) environment from the OpenAI Gym (https://gym.openai.com/).

To create the evaluator, use gym_evaluator.py (https://github.com/ufal/npfl122/tree/past-1920/labs/08/gym_evaluator.py) `.GymEvaluator("BipedalWalker-v2")`. The environment is continuous, states and actions are described at OpenAI Gym Wiki (https://github.com/openai/gym/wiki/BipedalWalker-v2).

You can start with the ddpg.py (https://github.com/ufal/npfl122/tree/past-1920/labs/08/ddpg.py) template, only set `args.env` to `BipedalWalker-v2`.

# walker_hardcore

As an extesnion of the `walker` assignment, try solving the BipedalWalkerHardcore-v2 environment (https://gym.openai.com/envs/BipedalWalkerHardcore-v2) environment from the OpenAI Gym (https://gym.openai.com/).

You can start with the ddpg.py (https://github.com/ufal/npfl122/tree/past-1920/labs/08/ddpg.py) template, only set `args.env` to `BipedalWalkerHardcore-v2`.

# az_quiz

The game itself is implemented in the az_quiz.py (https://github.com/ufal/npfl122/tree/past-1920/labs/09/az_quiz.py) module, using `randomized=False` constructor argument.

Note that az_quiz_evaluator.py (https://github.com/ufal/npfl122/tree/past-1920/labs/09/az_quiz_evaluator.py) can be used to evaluate any two given implementations and there are two interactive players available, az_quiz_player_interactive_mouse.py (https://github.com/ufal/npfl122/tree/past-1920/labs/09/az_quiz_player_interactive_mouse.py) and az_quiz_player_interactive_keyboard.py (https://github.com/ufal/npfl122/tree/past-1920/labs/09/az_quiz_player_interactive_keyboard.py).

For inspiration, use the official pseudocode for AlphaZero (http://science.sciencemag.org/highwire/filestream/719481/field_highwire_adjunct_files/1/aar6404_DataS1.zip). However, note that there are some errors in it.

- On line 258, value of the children should be inverted, resulting in:
  ```
  value_score = 1 - child.value()
  ```
- On line 237, next action should be sampled according to a distribution of normalized visit counts, not according to a *softmax* of visit counts.
- Below line 287, the sampled gamma random variables should be normalized to produce Dirichlet random sample:
  ```
  noise /= np.sum(noise)
  ```

# az_quiz_randomized

Extend the `az_quiz` assignment to handle the possibility of wrong answers. Therefore, when choosing a field, the agent might answer incorrectly.

To instantiate this randomized game variant, pass `randomized=True` to the `AZQuiz` class of az_quiz.py (https://github.com/ufal/npfl122/tree/past-1819/labs/10/az_quiz.py).

The Monte Carlo Tree Search has to be slightly modified to handle stochastic MDP. The information about distribution of possible next states is provided by the `AZQuiz.all_moves` method, which returns a list of `(probability, az_quiz_instance)` next states (in our environment, there are always two possible next states).

## vtrace

Using the vtrace.py (https://github.com/ufal/npfl122/tree/past-1920/labs/10/vtrace.py) template, implement the V-trace algorithm.

You can perform the test of your `Network.vtrace` implementation yourself using the vtrace_test.py (https://github.com/ufal/npfl122/tree/past-1920/labs/10/vtrace_test.py) module, which loads reference data from vtrace_test.pickle (https://github.com/ufal/npfl122/tree/past-1920/labs/10/vtrace_test.pickle) and then evaluates `Network.vtrace` implementation from a given module.

## memory_game

In this exercise we explore a partially observable environment. Consider a one-player variant of a memory game (pexeso), where a player repeatedly flip cards. If the player flips two cards with the same symbol in succession, the cards are removed and the player recieves a reward of +2. Otherwise the player recieves a reward of -1. An episode ends when all cards are removed. Note that it is valid to try to flip an already removed card.

Let there be $N$ cards in the environment, $N$ being even. There are $N+1$ actions – the first $N$ flip the corresponding card, and the last action flips the unused card with the lowest index (or the card $N$ if all have been used already). The observations consist of a pair of discrete values *(card, symbol)*, where the *card* is the index of the card flipped, and the *symbol* is the symbol on the flipped card. The `env.states` returns a pair $(N, N/2)$, representing there are $N$ card indices and $N/2$ symbol indices.

Every episode can be ended by at most $3N/2$ actions, and the required return is therefore greater or equal to zero. Note that there is a limit of at most $2N$ actions per episode. The described environment is provided by the memory_game_evaluator.py (https://github.com/ufal/npfl122/tree/past-1920/labs/10/memory_game_evaluator.py) module.

Your goal is to solve the environment, using supervised learning via provided *expert episodes* and networks with external memory. The environment implements an `env.expert_episode()` method, which returns a fresh correct episode as a list of `(state, action)` pairs (with the last `action` being `None`).

A template memory_game.py (https://github.com/ufal/npfl122/tree/past-1920/labs/10/memory_game.py) is available, commenting a possible use of memory augmented networks.

## memory_game_rl

This is a continuation of the `memory_game` assignment.

In this task, your goal is to solve the memory game environment using reinforcement learning. That is, you must not use the `env.expert_episode` method during training.

There is no specific template for this assignment, reuse the memory_game.py (https://github.com/ufal/npfl122/tree/past-1920/labs/10/memory_game.py) for the previous assignment.