Reinforcement Learning
Assignement - Function Approximation

In this assignment we will learn how to use function approximation. Before jumping to the exercises, let us briefly introduce a coding framework that will prove to be very useful for this assignment.
In order to efficiently compute derivatives, you can use JAX https://github.com/google/jax, which is a tool for algorithmic differentiation within Python. If you want to familiarize with JAX, you can have a look at https://github.com/probml/pyprobml/blob/master/notebooks/book1/08/autodiff_jax.ipynb
Note that JAX provides its own version of numpy, which is made such that it can be differentiated. This poses some minor limitations, e.g., arrays are immutable, etc.
In order to familiarize with how we can solve optimization problems using JAX, you can use the two files provided with this assignment:

- `jax_basic.py` provides a very simple code to understand how to call JAX to evaluate derivatives and implement gradient descent
- `jax_haiku.py` provides code that solves the same problem using Haiku, a neural network library for JAX: this allows us to define simple tabular function approximators (a specific case of linear FA) as well as deep neural network approximators

In this assignment we will use two MDPs from the OpenAI Gym. In order to familiarize with it, you can have a look at the code in `frozen_lake_sim.py`.

1. **Reimplementing the Tabular Case: Frozen Lake**

   In order to get familiar with the tools that are useful for function approximation, i.e., neural networks and automatic differentiation, let us begin with a simple MDP and let us implement it using a linear NN with one parameter per state-action pair, i.e., a tabular function approximator. To that end, we will use the Frozen Lake environment:

   ```
   env = gym.make('FrozenLake-v1', is_slippery=False)
   ```

   In this MDP, one has to cross a very small lake (a 4×4 grid), in which some states have thin ice which lead to termination with 0 reward. Upon reaching the target, reward 1 is obtained. In your code, modify the reward by introducing a small incentive to move: if the next state coincides with the current one, make the reward -0.01.

   (a) Implement Q-learning using a tabular function approximator. For this you need a shallow NN with a single linear layer (see the hints below). Display how the loss function and the NN parameters evolve throughout learning. Verify that you do obtain an optimal policy at the end of learning. You can use $\gamma = 0.9$, an ε-greedy policy with $\varepsilon = 0.01$, and run 2000 episodes with learning rate $\alpha = 0.1$.

   Hint 1: In order to encode the action-value function $\hat{Q}_\theta$, you can use the one-hot encoding function, so that you do not have to convert the state index given by the environment to a one-hot encoded vector all the time. Moreover, you can let the NN have as many outputs as actions, so that it will be easy to index and use in your code. Two useful lines of code:

   ```
   value = hk.Sequential((hk.Linear(nA, w_init=jnp.zeros, with_bias=False), ))
   return value(jax.nn.one_hot(S,num_classes=nS))
   ```

   Note that we remove the bias since it is not useful and we initialize the Q function to 0.

   This way of implementing a Q function is sometimes called a type-2 Q function, as opposed to type-1, in which the state-action pair is one-hot encoded and the NN has a single output. You can as well try to implement your Q function in this other way.

   Hint 2: be careful of what your policy returns when the optimal action is not unique. The policy should pick randomly with uniform probability among all optimal actions. Some useful functions: `jnp.max`, `jnp.sum`, `jnp.square`. You can also use just-in-time compilation by typing `@jax.jit` above the python functions you call during learning.

   (b) What happens if, e.g., out of laziness, you just pick the first optimal action instead of a random action among all optimal actions? Note that you can do this very simply using `jnp.argmax` in the function defining your policy.

2. **Let us now solve Frozen Lake with a generic NN approximator: Optional (but recommended)**

   Solve the Frozen Lake MDP using a nonlinear FA, e.g., an NN with 8 ReLU neurons (`jax.nn.relu`) followed by a linear layer with as many outputs as actions.

   (a) **Optional** Is this NN architecture adequate? Why?

   (b) **Optional** What happens if you implement the policy by being "lazy", i.e., using `jnp.argmax`? Why? What if you remove the small incentive to move?

   (c) **Optional** Is it a smart move to use a deep NN as the one proposed above to solve Frozen Lake?

3. **A more difficult MDP: Cart Pole**

   Let us now try to solve a more difficult MDP with continuous states and discrete actions. We consider the Cart Pole MDP, in which we only have 3 discrete actions to balance a pendulum mounted on top of a cart. The objective is to keep the pendulum upright as long as possible. The reward is simply $R(s,a) = 1$ if the pendulum is sufficiently close to the upright position and the MDP terminates with 0 reward if the pendulum starts to fall. Because the environment terminates after 200 time steps, we need to account for the fact that if the pendulum is still upright, we might as well keep it so forever. Therefore, in that case you need to modify the reward as $R(s,a) = \sum_{t=0}^{\infty} \gamma^t = \frac{1}{1-\gamma}$.

   (a) Try to solve this MDP by using Q-learning and approximate the action-value function using an NN with 3 layers of 8 ReLU neurons each.

(b) Let us now implement DQN. To that end, we will work with experience replay, sampled from an experience buffer that we will construct; and using fixed targets that we update at a slower pace. Before starting the updates, let us first construct a buffer which contains at least 200 samples, then let us pick 128 samples randomly from the buffer, but making sure that the last observed sample is used. In order to use fixed targets, we need to make a copy of the parameters and update it differently. Let us consider $\gamma = 0.9$ and use an ε-greedy policy with $\varepsilon = 0.1$. Let us use learning rate $\alpha = 10^{-3}$ and an experience replay batch of 128 samples. At every time step, the targets are updated in a "damped" way, i.e., with an additional learning rate α_{tup}. The update could then read

$$p_{\text{t}} = (1 - \alpha_{\text{tup}})p_{\text{t}} + \alpha_{\text{tup}}p.$$

Try to implement this variant using $\alpha_{\text{tup}} = 0.01$.

Hint: if you code your functions properly, you should be able to evaluate the Q function for the whole batch with one single call to the function. This might make your code faster.