Introduction to Machine Learning for Physicists

Problem Set 3
Lecturer: Fabian Ruehle
Github: https://github.com/ruehlef/ML_Oxford_Hilary2020/Problem_set_3/

3.1 Reinforcement learning for the 1D Ising model

In this exercise, we will use Reinforcement Learning to find the ground state of a 1D Ising model from an arbitrary initial configuration. The model assumes a lattice of states, each of which can be spin up or spin down, $\sigma = -1, 1$. Two adjacent lattice sites σ_i and σ_j couple with a coupling strength J_{ij}. Moreover, each site σ_j can interact with an external magnetic field h_j. Hence, the Hamiltonian of the system reads

$$H(\sigma) = -\sum_{i,j\text{adjacent}} J_{ij}\sigma_i\sigma_j - \mu\sum_j h_j\sigma_j\,. \tag{1}$$

For simplicity, we will analyze the system for $h_j = 0$ in the ferromagnetic phase ($J_{ij} = 1$) and the anti-ferromagnetic phase ($J_{ij} = -1$). Actions will consist of flipping one spin at on lattice site.

(a) For N lattice sites, what is the dimension of the state space and of the action space?

We will use the A3C implementation of ChainerRL for this exercise. A simple way of interfacing your own environment with the code is to create an OpenAI gym environment http://gym.openai.com/docs/. You can then define your own environment class which inherits from `gym.Env`. This class should implement the five methods

- `__init__(self)`
- `step(self, action)`
- `reset(self)`
- `render(self, mode, close)`
- `seed(self, seed)`

We won't be using seeding and rendering the environment, so we can just provide any implementation there. Let us focus on the others. As usual, you can find a template file that already has the correct overall structure on GitHub, and you can insert your implementations of the relevant parts into this file.

(b) Implement `__init__(self)`. This function should (at the minimum) initialize a member variable that keeps track of the current internal state, i.e. of the spin configuration. Moreover, you should also define two member variables `self.action_space` and `self.observation_space`. For these you can use

```
self.action_space = spaces.Discrete(number_of_actions)
self.observation_space =
    spaces.Box(-1, 1, [size_of_lattice], dtype=np.float32)
```

If you choose to use this, an action will just be an integer $0, 1, \ldots,$number_of_actions-1. The observation space will be an array with entries between -1 and 1, and of dimension size_of_lattice.

(c) Think about how you want to reward/punish the agent. The goal should be to find the minimum energy configuration. Implement the reward function, which gets the current state as an input and returns a float that encodes the reward or punishment (modeled by a negative reward).

(d) Implement the step function. This will be called by ChainerRL with an action that has been predicted to be the next best action according to the current policy as predicted by a NN. In the function, you need to perform the action (i.e. flip the spin at the lattice site indicated by this action). After that, you should evaluate the reward for the action based on the new state the system is in now and return the following four quantities

- The new state (as a numpy array)
- The reward for the action
- Whether the new state is terminal
- A dictionary with additional information. For our purposes, this can just be empty, i.e. {}.

(e) Implement the reset function. This should bring the environment in a well-defined initial state. This function is called after a terminal state is reached or after a predefinied maximum number of steps (which I set to 100 in `__init__.py` in the gym_a3c folder) is reached. It should return the start configuration (as a numpy array). For the start configuration we can just use a random start configuration which changes every n resets. Try different values for n.

(f) Look at the file train_a3c_gym.py. The NNs for the policy and the state value function are defined in `class A3CFFSoftmax(chainer.ChainList, a3c.A3CModel)`. We use a feed forward NN (called multi-layer perceptron, or MLP, in Chainer) for both. Both get a state as an input, hence the input dimension is set to `ndims_obs`. The policy network is set up as a classifier that returns the likelihood for each action to be the best one, hence it has `n_actions` outputs (and a softmax policy). The state value is just a single real number, hence the output of the state vale NN is just a single node. With `hidden_sizes=`$(h_1, h_2, h_3, \ldots, h_k)$ you can define the size of each of the k hidden layers, and with `nonlinearity` you can define their activation function. Feel free to try different architectures here.

(g) Train 6 agents (or some other number depending on the number of cores of your machine) in parallel for 10^7 steps by running the `python train_a3c_gym.py 6`.

(h) Can you see domain walls forming? Play with the parameters (such as the discount `gamma`) or the number of steps `t-max` after which the sum is truncated and an update is performed. How do they influence the results?