**Stabilization of inverted pendulum using Reinforcement Learning**

­­­­­

**The environment:**

The environment consists of a simple 2D model of a pendulum. The pendulum is made up of a base which is connected to a point-mass trough a stiff rod which is free to rotate. The model assumes that the base can affect the point-mass, but not the other way around: The base is assumed independent of the point-mass and rod. The state-variables are base-position (x: p\_x, p\_y), base-velocity (v: v\_x, v\_y), pendulum angle (theta) and angle-velocity (theta-dot): . The derivative of the state is thus v (velocity), a (acceleration: x,y), theta-dot (angular velocity), theta-dot-dot (angular acceleration): .

The input to the system is an applied force on the pendulum base F. Since the base is independent of the rod, we can calculate from newtons first law that . We can also calculate the torque on the rod as , where is the length of the rod and is the unit vector in the direction of the rod. is the tangential component of the force acting on the point mass and is calculated as the difference between the force on the pendulum base and the force on the point mass (which is translated through the rod): . From the torque we calculate where is the moment of inertia of the rod, defined as , where is the mass of the point mass.

From this we can evolve the state of the system through time by defining the initial conditions and integrating the state using the RK4 method.

**Control of the pendulum using Reinforcement Learning (RL)**

With the environment set up, the next step is to create and train the reinforcement learning agent. Here it is convenient to start simple and thereafter improve the methods. We first define the problem. The goal of the agent will be to stabilize the pendulum in an upright position by applying a force on the pendulum base. To further simplify the problem, we say that the base can only move in the x-direction, and thus that the applied force can also only be parallel to the x-axis. This reduces the number dimensions of the state-space from six to four, since we no longer need the y-position or y-velocity of the pendulum base. We also reduce the action-space from two to one dimension. To start with we also choose a discrete action-space where the agent can only choose to apply a force to the left (0) or to the right (1).

**First model: Deep Q-Network agent**

There are many RL-algorithms to choose from, but it is convenient to start as simple as possible and thereafter explore more complex methods. The first model we explore is the DQN (Deep Q-Network) model. This is an algorithm which aims to estimate the expected discounted future reward for each action , given the state . That is, it aims to estimate the action-value function . This method uses a deep neural network as the function approximator for Q. The network takes as input the current state and returns an approximation of for each action a. After training we can select the best action as the argmax of the output of the network, thereby selecting the action which is expected to give the highest future reward.

For training the network we give the agent access to the environment and let it select actions based on the epsilon-greedy strategy which encourages exploration. For each timestep the agent gets the current state , selects an action and receives a reward , which is +1 if the pendulum stays upright and is some negative value if the pendulum reaches some maximum deviation from the upright position. These values, along with the next state , () is stored to a registry for future learning. When the registry has enough state-transition values for a minibatch we perform learning. The learning is done by selecting a shuffled minibatch of the registry. This eliminates the correlation between following states, making learning more efficient. For a learning step the agent is fed the state and outputs its estimate of the Q-value for the state. We know that the action selected is and that this action leads to . We therefore also feed the state to the network and observe . From this we know that is a better approximate of . We use this information to update the weights of the network through back-propagation, using the mean-squared-error loss function such that the network will output an estimate which is closer to the next time it encounters state . Over time it will refine its estimates of the Q-values.

The network was chosen fairly arbitrarily as the input layer followed by two feed-forward layers using relu activation, followed by the output-layer with a linear activation function. The size of the memory storing encountered state-transitions was 2000 and the size of each minibatch was 32. The discount rate was selected as , and epsilon started as 1 and was reduced by a decay rate of 0.995.

**Policy Gradient Method**

The next step will be to implement the REINFORCE algorithm. Where the network aims to directly approximate the optimal policy. The policy gradient method directly parametrizes the policy. The policy network will then receive the state and output a probability distribution over the possible actions. The goal will be to maximize the expected future discounted reward:

So, the weights of the policy network should be updated along its gradient: .

A problem with this is that the network does not output any rewards, it only outputs probabilities of actions. The observed rewards are only Monte Carlo samples from these actions. The REINFORCE algorithm handles this by playing an entire episode from the policy and only looking at the chosen actions.

**Monday 19 august**

Im now revisiting this project to clean it up and post it on github with proper documentation. First we want to clean the project and properly structure it.

I want to structure it as:

* one file for the environment,
* one file for all models (DQN and REINFORCE),
  + Abstract base class for the model template with models inheriting.
* one file for input parameters (YAML)
  + Train / Run
  + Model type (DQN / REINFORCE)
  + Number of episodes, weight save frequency, episode duration, minibatch size
* one main file running according to various runtypes

We want to store statistics like loss/rewards to make plots and comparisons.

**Saturday 24. August**

Have now gotten well started on cleaning up the project. I would now say im pretty much finished refactoring the code I already had. Some updates include:

* YAML file for input params.
* Env and models separated into separate files.
* Main cleaned up. (Should continue a bit here)

For the next steps I have some thoughts:

* **Documentation**
  + This is a natural point to start creating documentation for the project. We must properly document the code with proper comments and docstrings, as well as writing documentation or a brief report.
* **Continuous action space**
  + One last step I want to include is to add a model acting with a continuous action space, where not only the direction, but also the magnitude of the force is determined by the agent.

I also had an idea for writing the documentation. I’m wondering if it would be feasible/favorable to implement an LLM workflow using the openai API. I was thinking if it could be possible to use an “actor-critic” approach with multiple LLMs where one writes documentation and another critizises the documentation until it converges to a good point. This could potentially automate

**Actor-critic methods:**

**A white sheet with black text

Description automatically generated**

Let’s now implement the actor-critic approach. Here we optimize both a state-value function and the policy. There are also some new aspects, like the eligibility trace vectors. These vectors are exponentially decaying histories of the gradients.

Lets first try without eligibility traces:

A white sheet with black text

Description automatically generated

We need two function approximators, one for the policy and one for the state-value function.

I’ve now implemented the critic, which uses MSE with the bellman equation for updating its estimates based on “seeing” the reward and using the next step expectation.

Next is the actor. From my understanding it will do pretty much the same as the basic REINFORCE algorithm, but now instead of using Gt it will use Gt – v(s) (TD).A math equations and formulas

Description automatically generated with medium confidence

I think this will serve as a baseline for the updates. This should further differentiate the good and bad actions, since for example in cases where both actions would yield high reward, we now will have a larger discrepancy between them, since the baseline would lead to favoring the better action.

