**Homework 04**

**AA203: Optimal and learning-based Control**

Ricardo Luiz Moreira Paschoeto

[rp304154@stanford.edu](mailto:rp304154@stanford.edu)

HW02, 03 and 04 was done in colaboration with my group partner Srikanth.

**Problem 1:** Deep reinforcement learning

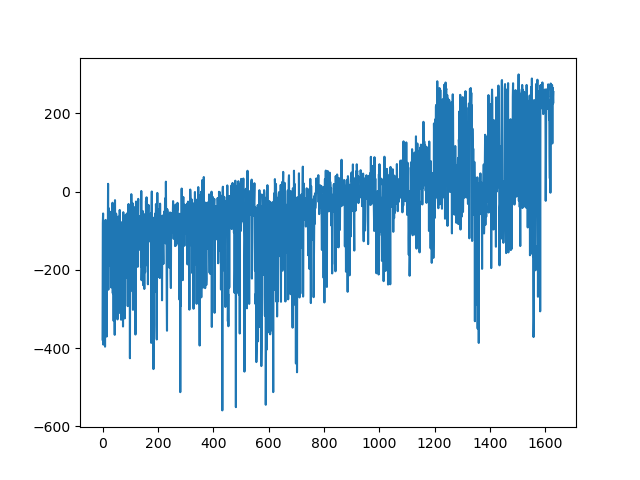


Figura 1 - episodic\_rewards x episodes

Python code:

    def forward(self, x):

        """

        forward of both actor and critic

        """

        # TODO map input to

        # mean of action distribution

        # variance of action distribution (pass this through a non-negative function)

        # state value

        x = F.relu(self.affine1(x))

        x = F.relu(self.affine2(x))

        a\_mean = self.action\_mean(x)

        a\_var = torch.exp(self.action\_var(x))

        s\_values = self.value\_head(x)

        return 0.5\*a\_mean, 0.5\*a\_var, s\_values

def select\_action(state):

    state = torch.from\_numpy(state).float()

    mu, sigma, state\_value = model(state)

    # create a normal distribution over the continuous action space

    m = Normal(loc=mu,scale=sigma)

    # and sample an action using the distribution

    action = m.sample()

    # save to action buffer

    model.saved\_actions.append(SavedAction(m.log\_prob(action), state\_value))

    # the action to take (left or right)

    return action.data.numpy()

def finish\_episode():

    """

    Training code. Calculates actor and critic loss and performs backprop.

    """

    R = 0

    saved\_actions = model.saved\_actions

    policy\_losses = [] # list to save actor (policy) loss

    value\_losses = [] # list to save critic (value) loss

    returns = [] # list to save the true values

    # calculate the true value using rewards returned from the environment

    for r in model.rewards[::-1]:

        # TODO compute the value at state x

        # via the reward and the discounted tail reward

        R = r + args.gamma \* R

        returns.insert(0, R)

    returns = torch.tensor(returns)

    returns = (returns - returns.mean()) / (returns.std() + eps)

    # whiten the returns

    returns = torch.tensor(returns).float()

    returns = (returns - returns.mean()) / (returns.std() + eps)

    for (log\_prob, value), R in zip(saved\_actions, returns):

        # TODO compute the advantage via subtracting off value

        adv = R - value.item()

        # TODO calculate actor (policy) loss, from log\_prob (saved in select action)

        # and from advantage

        policy\_losses.append(-log\_prob \* adv)

        # append this to policy\_losses

        # TODO calculate critic (value) loss

        value\_losses.append(F.mse\_loss(value, R))

    # reset gradients

    optimizer.zero\_grad()

    # sum up all the values of policy\_losses and value\_losses

    loss = torch.stack(policy\_losses).sum() + torch.stack(value\_losses).sum()

    # perform backprop

    loss.backward()

    optimizer.step()

    # reset rewards and action buffer

    del model.rewards[:]

    del model.saved\_actions[:]

**Problem 2:** Extremal curves

Where:

Applying Euler Equation,

Where:

,

Computing:

Replacing in (I):

Homogenous Equation:

.

Then:

Particular Solution:

,

Replacing in (II):

.

Overall Solution:

Using conditions:

Solving the system we have:

Then the extremal curve:

**Problem 3:** Minimum control effort

Hamiltonian:

Necessary Optimality:

Computing:

From (I):

From (II):

From (III):

Using boundary condition,:

Computing:

Replacing in (I):

Where:

Integrating (V):

Using

Finally:

**Problem 4:** Zermelo’s ship

a)

Using Necessary Optimality Conditions:

Then:

From (I):

Where :

b)

From state equation y(t):

=