Reinforcement Learning - Final Report

Vsevolod Klyushev

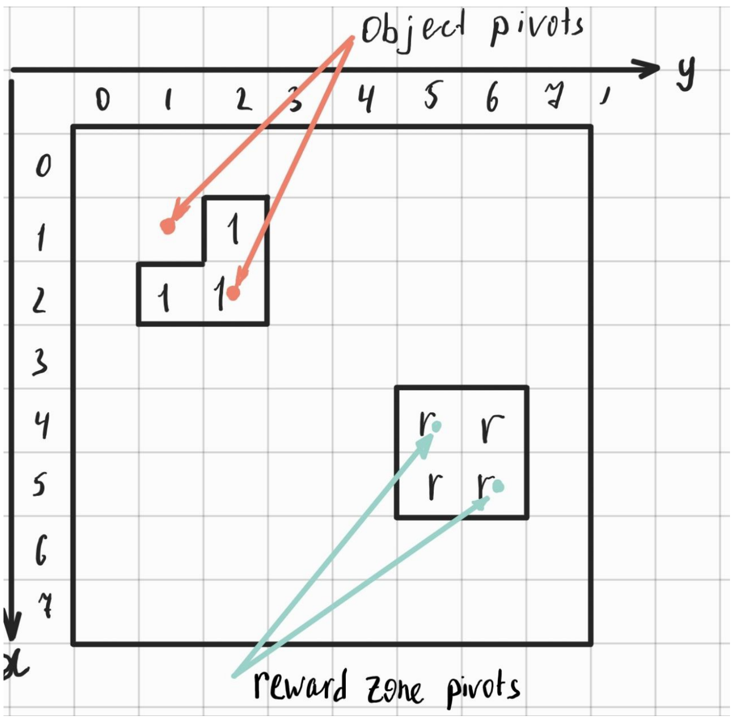
Description of the environment representation:

Since we have an environment without a fixed number of states, I decided to use several coordinates and the value of the reward and penalty received as a representation of the environment for the object (cargo) in our model.

In more detail - each object contains 2 pairs of coordinates that determine its position. The first coordinate is the smallest coordinates of its components on both axes, the second coordinate is the largest coordinates of its components (i.e. if the object is a corner with cells having coordinates (2 1), (2,2), (1,2), then, according to the representation described earlier, it can be represented as (1,1), (2,2)). Thus, any shape of the cargo is represented through 2 coordinates. Let’s call smaller coordinate as **low** and greater coordinate as **high.**

In addition, in the same way, you can set the zone with rewards that we want to achieve, because according to the condition of the task, it is always the same and has the shape of a rectangle.

Let's call such pairs of coordinates as pivots. It’s obvious, that if pivots of object lies between pivots of reward zone, then entire object is in the reward zone.



The reward and penalty values are calculated according to the task description - for each part of the cargo in the reward zone +1 to the rewards and for each intersection with other cargo -1 to the penalty counter.

Therefore, every object of the environment might be represented as object pivots x coordinates, object pivots y coordinates, reward zone pivots x coordinates, reward zone pivots y coordinates, current reward of cargo, current penalty of the cargo. 10 values in total: $(x_{cargo_{low}}, y_{cargo_{low}}, x_{cargo_{high}}, y_{cargo_{high}}, x_{reward_{low}}, y_{reward_{low}}, x_{reward_{high}}, y_{reward_{high}}, r, p)$

For example, for cargo 1 from picture we would have the following environment representation - $(1,1,2,2,4,5,5,6,0,0)$.

Description of the neural network architecture:

Since I decided to predict action value function based on a quite small input vector (10 features), and relation between thus variables is not very complex either, in my opinion, usual neural network with relatively small number of linear layers and a sigmoid activation function is sufficient.

```
HIDDEN_SIZE = 6
INPUT_SIZE = 10
n_actions = 4

class MyModel(nn.Module):
    def __init__(self):
        super(MyModel, self).__init__()
        self.input_layer = nn.Linear(INPUT_SIZE, HIDDEN_SIZE)
        self.output_layer = nn.Linear(HIDDEN_SIZE, n_actions)

    def forward(self, x):
        x = F.sigmoid(self.input_layer(x))
        x = self.output_layer(x)
        return x
```

I also tried to use several linear layers and the relu activation function, however the current configuration showed better results.

Description of the neural network training method:

First of all we need to use relatively simple map with just one cargo. In that case model would *find* relation between different features (object pivots and reward zone pivots).

Training for one map has the following structure:

- create environment
- for every epoch
    - for each step
        - decide which cargo to move
        - receive action value function values for each of the 4 actions by passing state of the cargo into neural network
        - decide which actions are available in current state
        - for valid actions apply softmax with temperature in order to achieve probabilities of each action (we use softmax with temperature in order to have exploration exploitation trade off)
        - choose action that would be performed
        - save current state as prev_state
        - save reward and penalty for cargo in current state
        - make move
        - store current state
        - check if this state is terminal
        - save in list of transitions tuple which consist of prev_state, action that was preformed and difference in reward between new current state and previous one.
        - if state is terminal, break loop
    - append in list of scores length of transitions list

- for transitions calculate expected return based on Monte Carlo method
- retrieve which values neural network model gave for each state in transition
- calculate loss as sum of MSE values between expected results and model predictions
- append received loss in losses list
- perform backpropagation

Example of training process for 1 map:

```
Horizon = 100
MAX_TRAJECTORIES = 10000
gamma = 0.99
score = []
losses = []
criterion = torch.nn.MSELoss()
env = create_grid_world(INPUT)

for trajectory in tqdm(range(MAX_TRAJECTORIES)):
    env.reset()
    indexes = env.objects_indexes
    done = False
    transitions = []
    curr_states = [decompose(env.get_state(i)) for i in indexes]
    temp = 100 / (trajectory+1)

    for t in range(Horizon):

        unstable_indexes = env.get_unstable_indexes()
        index = unstable_indexes[random.randint(0, len(unstable_indexes)-1)]
        # index = list(indexes)[random.randint(0, len(indexes)-1)]

        act_prob = model(torch.from_numpy(curr_states[index-1]).float())

        l = []
        for i in [0,1,2,3]:
            if env.validate_move((index, i)):
                l.append(i)
        probs = (act_prob/temp).data.numpy()[l]

        action = np.random.choice(l, # U R D L
                p=scipy.special.softmax(probs, axis=0))

        prev_state = curr_states[index-1]

        ps = decompose(env.get_state(index))
        prev_reward_pos = ps[8]
        prev_reward_neg = ps[9]

        env.make_move((index, action))
        curr_state = env.get_state(index)
        size = curr_state[3]
        curr_states[index-1] = decompose(curr_state)
        done = env.get_reward()[1]
        transitions.append((prev_state, action,
                           (curr_states[index-1][8] - prev_reward_pos + curr_states[index-1][9] - prev_reward_neg)))
        if done:
            break

    score.append(len(transitions))

    reward_batch = torch.Tensor([r for (s,a,r) in transitions])
    batch_Gvals = []

    temp_Gval = 0
    for i in range(len(transitions)):
        temp_Gval = gamma * temp_Gval + reward_batch[len(transitions)-i-1].numpy()
        batch_Gvals.append(temp_Gval)

    batch_Gvals = batch_Gvals[::-1]

    expected_returns_batch=torch.FloatTensor(batch_Gvals)
    state_batch = torch.Tensor(np.array([s for (s,a,r) in transitions]))
    action_batch = torch.Tensor(np.array([a for (s,a,r) in transitions]))

    pred_batch = model(state_batch)
    val_batch = pred_batch.gather(dim=1,index=action_batch
                .long().view(-1,1)).squeeze()

    loss= criterion(val_batch, expected_returns_batch)
    losses.append(loss.item())
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```

Faced difficulties:

Initially, I expected that my model would simply predict the most profitable action (there are 4 of them in total - go up, right, down or left) based on the current state for a certain cargo. So model returns probabilities for performing each of the 4 actions for certain state. However, not everything turned out to be so simple.

I was faced with the problem of choosing a way to calculate the loss function. Inspired by a solution from laboratory work, I decided to try the following loss function:

`loss= -torch.sum(torch.log(prob_batch)*expected_returns_batch)`

In short, it is calculated as follows: first, for each state visited during epoch, the expected response is calculated using the Monte Carlo method. Then this value is normalized and the logarithm of the probability of choosing this action is multiplied by.

In the lab task, it was necessary to choose one of two actions for a certain state, but in our task the number of actions increased to 4.

In theory, it should have worked, but unfortunately, only in theory. In order to inspire the model to build a path for the pear as short as possible, I added a negative reward for each step that is not performed in the reward zone. This is where all the fun begins.

Since some strange things happened with the length of the created route and the loss function, I started looking for the problem. I couldn't find it until I rewrote the model and the loss function.

I decided to predict with the help of a neural network directly the value of the action value function and consider the error as MSE between the expected results from the model and what it outputs.

`loss= criterion(prob_batch, expected_returns_batch)`

So in order to decide which action to perform I use softmax function on values from model, receive probabilities and perform action according to them.

The main change for the model was that I used softmax with the temperature outside of it. (In both cases, I used softmax with temperature to balance exploration and exploitation.)

After that, I thought about what was going on. It just so happens that the model wants to achieve results along the way that coincide with the expected ones. At first glance, it seems that this is how it should be. But this guy decided that crashing into a wall, thereby receiving a fixed negative reward that converges well to some value that turns out to be greater than the predicted values of the action value function for other actions, is a brilliant idea. Even a very large negative value at the last step not in the reward zone did not help.

In other words, the model was happy to direct the cargo towards the wall and beat it against it, correctly predicting what penalty it would receive, which means that the loss function would give an extremely small value. Wonderful, simply incomparable. **(That's a great plan, Walter. That's fucking ingenious, if I understand it correctly. It's a Swiss fucking watch.)**

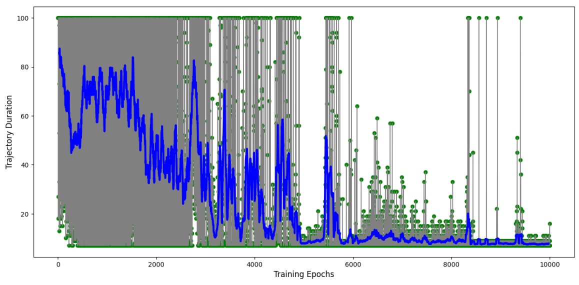
Anyway, I decided that if I forbid him to walk into the wall, then this problem would be solved. But it was too naïve. The situation was repeated with minor changes, in particular, instead of moves to the wall, moves were made to the state that was on the last move. For example, the model said to go right, then left, then right again, and so on.

My remaining nerve cells told me that, in principle, the optimal path is not so important, taking into account the remaining time until deadline, so for all moves except those in the reward zone, a reward of 0 is given. Oh my God, now he behaves as he should, but just not always.

After that, I attended a consultation and learned about how best to reward the model. The new idea was to give out as a reward for the transition the difference between two consecutive states for a certain cargo. Thanks to this, changing of number of layers and activation function in neural network, as well as the correction of several bugs, I have achieved stable performance for maps with a single cargo. (Thanks to Daniil Arapov)

For example, for such map received the following graphs

$$\begin{array}{c}
000000000000000 \\
000000000000000 \\
000rrr000000000 \\
000rrr000000000 \\
000000000000000 \\
000000001100000 \\
000000001100000 \\
000000000000000 \\
000000000000000
\end{array}$$



Model provides such sequence of moves as answer:

$[(1,'L'),(1,'U'),(1,'L'),(1,'U'),(1,'L'),(1,'U'),(1,'L')]$

I also tried to train the model on several maps, but I don't have enough time to implement it well.

Inspiration of the architecture and representation used for the environment:

Inspiration of using coordinates and directions came from lab classes and Assignment 1. In assignment 1 I also used pivot cell in order to simplify all processes. Working with directions was mentioned on lab classes several times. Since my representation works well, I haven’t tried anything else.

Acquired knowledge and skills:

- Better understanding of neural networks and Pytorch
- Different techniques of calculation loss function for policy based and action value based models
- I learned some new tricks in Python
- Implementation of several RL concepts in code
- The fact that dummier than your model is only you