Sergeev Nikita BS20-AI, final report

Description of the environment representation:

For the enviroment representation I implemented class *Environment* with the following component:

- *world_h*, *world_w* - represents total world height and width
- *cargos* - represents dict with *keys* - label of the cargo in the world map. And values are lists with coordinates of the cargo cells.
- *desired space* - list of coordinates of cells in wich situated desired space of the world.
- *world* - numpy array simillar to the given world map, but desired space represented with **-1** instead of **r** in the initial world. It's made so to have number respresentation of the world to make it perceived to the algorithm.
  Also my *Environment* class have following **methods**:
- *move_cargo* - method for moving particular cargo along the enviroment. It takes 2 argiments - cargo which will be moved by this method and action (direction in which cargo will be moved)
- *check_move* - method that return if move of the cargo in the particular direction legal or by this move cargo will crush into the wall.
- *reset* - roll back the environment into it initial state. In other words, if after several moves of cargo we will call this method, positions of all cargos will be returned to the initial.
- *get_cargo_overlaps* - takes cargo as a parameter and return number of other cargo cells which overlaps with the given cargo.
- *get_cargo_overlaps_with_desired* - method, which takes cargo as a parameter and returns number of cells of this cargo, placed in the desired space in the current state of the environment.
- *is_done* - method which returns is game other, according to the rules given in the assignment description (all cargo should be fully in the desired space and don't overlaps).

From the enviroment my model receives current state of the system and from it determine next action. State of the system may be represented as modificated world field of the environment.
State of the system is vary depending on which cargo do we need to move in the current step. State is 2D numpy array, simmilar to the world map from the input file. But desired zone in the state replaces with -1, and other cargos, which are not moveing in the current step are replaced with -2. Other cargos should be considered by the acting cargo identically (as a wall, in which undesireable to collapse).

With these replacement model can interpet all the information from the input world. With this state is also easy for model to differentiate between cargo, wich we should move in the current step(and from which the model answer should be mainly dependent), other cargo, which are act as a wall in this step, and desired zone, in which cargo should ideally move. For example in the step of cargo '1' this replacement will look like this:

$$\begin{bmatrix} 0 & 0 & 2 & 2 & 0 \\ 0 & r & r & r & 0 \\ 0 & r & r & r & 1 \\ 0 & r & r & r & 1 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \rightarrow \begin{bmatrix} 0 & 0 & -2 & -2 & 0 \\ 0 & -1 & -1 & -1 & 0 \\ 0 & -1 & -1 & -1 & 1 \\ 0 & -1 & -1 & -1 & 1 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

Description of the neural network architecture

For the given task I implemented Deep Q Network with the following architecture:

```python
class DQN(nn.Module):
    def __init__(self, output_size):
        super(DQN, self).__init__()
        self.conv1 = nn.Conv2d(1, 16, kernel_size=3, stride=1, padding=1)
        self.conv2 = nn.Conv2d(16, 32, kernel_size=3, stride=1, padding=1)
        self.conv3 = nn.Conv2d(32, 64, kernel_size=3, stride=1, padding=1)
        self.global_pool = nn.AdaptiveAvgPool2d((1, 1))
        self.fc1 = nn.Linear(64, 64)
        self.fc2 = nn.Linear(64, output_size)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = F.relu(self.conv2(x))
        x = F.relu(self.conv3(x))
        x = self.global_pool(x)
        x = x.view(x.size(0), -1)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return x
```

This network have

- 3 convolutional layers with `stride=1` and `padding=1` to not reduce size of the input matrix.
- Global average poolling to convert given input matrix to tensor with size equal to number of layers in the final convolutional layer.
- 2 Fully connected layers on the flatten output of the global average pooling
- And after fully connected layers we have 4 neurons as an output, which represent 1 of 4 actions, which cargo may perform.

Why does this particular architecture was selected:

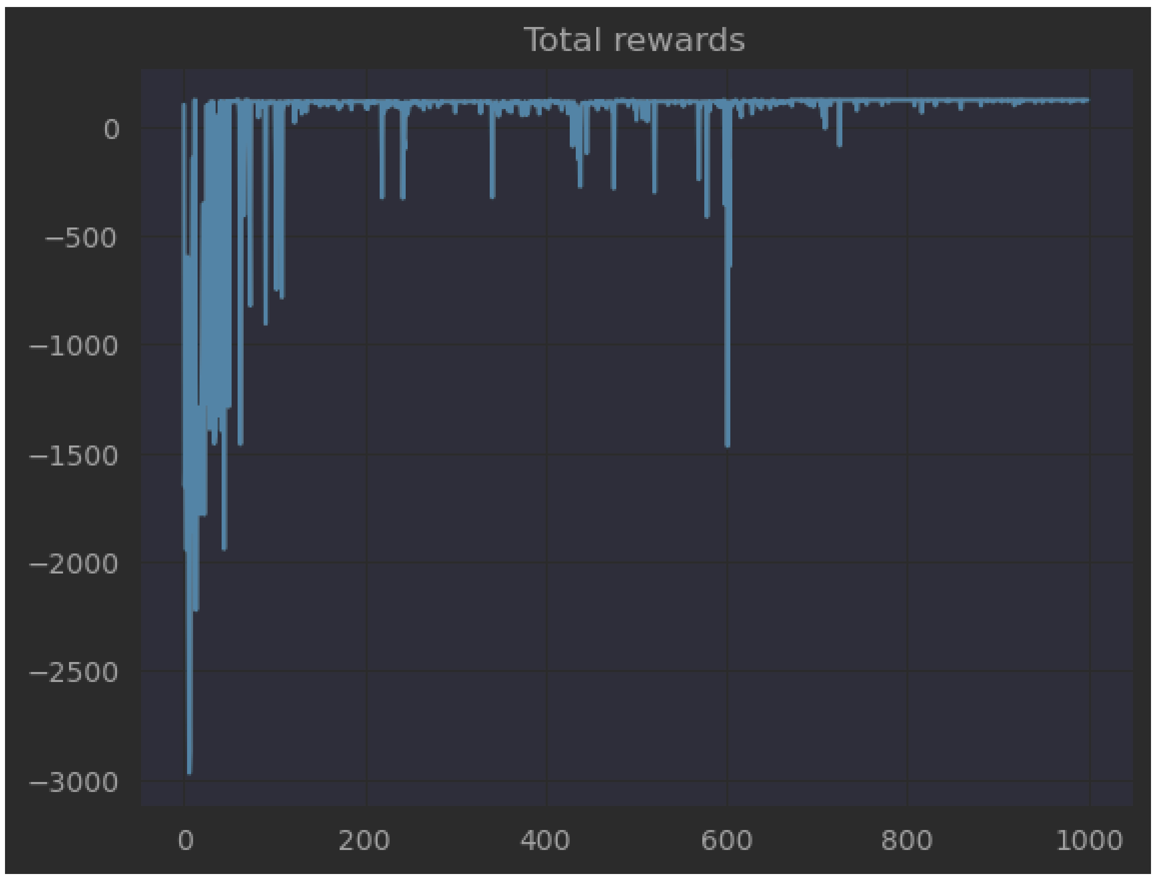
- I used Convolutional NN as feature extractor from the given state of the environmnet (matrix with positions of the objects)
- Global Average Pooling was chosen to make model more general. With it model can work with matrix of any size.
- Fully connected layers at the end needed to find some kind of correlation between output of the Global Average Pooling. With 2 dense layers I transform 64 outputs from the GAP to

4 output, which correspnds to directions (['D', 'U', 'R', 'L'])

The main reason, why this network architecture was chosen is that he just works better then other models, with which I experemented. After some number of episodes this model can find some solution. For example, for the static world:

$$\begin{bmatrix} 0 & 0 & 2 & 2 & 0 \\ 0 & r & r & r & 0 \\ 0 & r & r & r & 1 \\ 0 & r & r & r & 1 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

Graph of the reward function looks the following way:



We can se, that after ~50 episodes of training in this environemnt with ϵ greedy policy, DQN algorithm will show correct path of cargos to the desired zone almost in all the cases.

Description of the neural network training method:

For training my model i do the following:

- Simulate game in the given environment 1000 times
- Make moves for each cargo according to the $\epsilon - greedy$ strategy with the following formula:

  $$\epsilon_{threshold} = \epsilon_{end} + (\epsilon_{start} - \epsilon_{end}) \times \exp(\frac{-steps}{\epsilon_{decay}}), \text{where}$$

  $$\begin{bmatrix} \epsilon_{end} = 0.05 \\ \epsilon_{start} = 0.5 \\ \epsilon_{decay} = 200 \end{bmatrix}$$

  When get random number and if it $> \epsilon_{threshold}$ then select action according to the DQN decision (*exploitation*), otherwise select action randomly(*exploration*) and increase *steps* each time we select action, as we would like to explore at the beginning and exploite at the end.

Reward policy redefining - FIrstly for the effective learning I decided to change reward policy to the following:

- Each cargo get -1 to reward for every move
- To reward addeded overlaps of cargo with desired space
- To reward added **difference in overlaps with desired in the state before move of cargo and after move**. It is needed to prevent cargo from abusing reward by entering desired zone and leave it repeatevely
- From reward of the step substracted number of cargo cells, which are overlaped with another cargo.
  Code of getting reward looks like this:

```python
def get_reward(environment, possible, cargo, diff):
    reward = 0
    reward += environment.get_cargo_overlaps_with_desired(cargo)
    reward += diff
    reward -= environment.get_cargo_overlaps(cargo)
    reward -= 1
    if not possible:
        reward = -10
    return torch.tensor([reward], dtype=torch.float)
```

- and if cargo by the current step achieved final state of the game it recieves huge reward for this $reward+ = env.world_h * env.world_w * get\_cargo\_cells(env)$

After determining action of the cargo and reward for this action

We add transition(current_state, action, next_state, reward) to the **ReplayMemory**(deque of transitions) and with help of it, optimize our model.

To optimize my DQN algorithm I do the following steps:

- From the ReplayMemory we choice random batch of given size(in my case $batchSize = 128$)
- I have 2 CNN - *policy_net* and *target_net*
- Policy net computing $Q(s_t, a)$ - the model computes $Q(s_t)$, then we select the columns of actions taken. These are the actions which would've been taken for each batch state according to policy net
- Target net computing $V(s_{t+1})$ for all next states. Expected values of actions for non_final_next_states are computed based on the "older" target_net; selecting their best reward with $max(1)[0]$.
- Expected Q values from the target net $expected\_state\_action\_values = (next\_state\_values * \gamma) + reward\_batch$, where γ is a constant discounting factor. In my case $\gamma = 0.999$
- Then, loss of the DQN in the current step may be calculated as $SmoothL1Loss$ from PyTorch applied to state_action_values - computed by policy net and expected_state_action_values - computed with help of target net.

- Then we as usual perform loss backward and step of the optimizer (Adam) to optimize parameters of policy net. I don't update parameters of target net, but Every 10 episodes of training I update it state dict to state dict of the polity net.
- The last step we do, while optimizing parameters of the network in to clamp gradients of the policy net between -1 and 1 , so they don't explode.

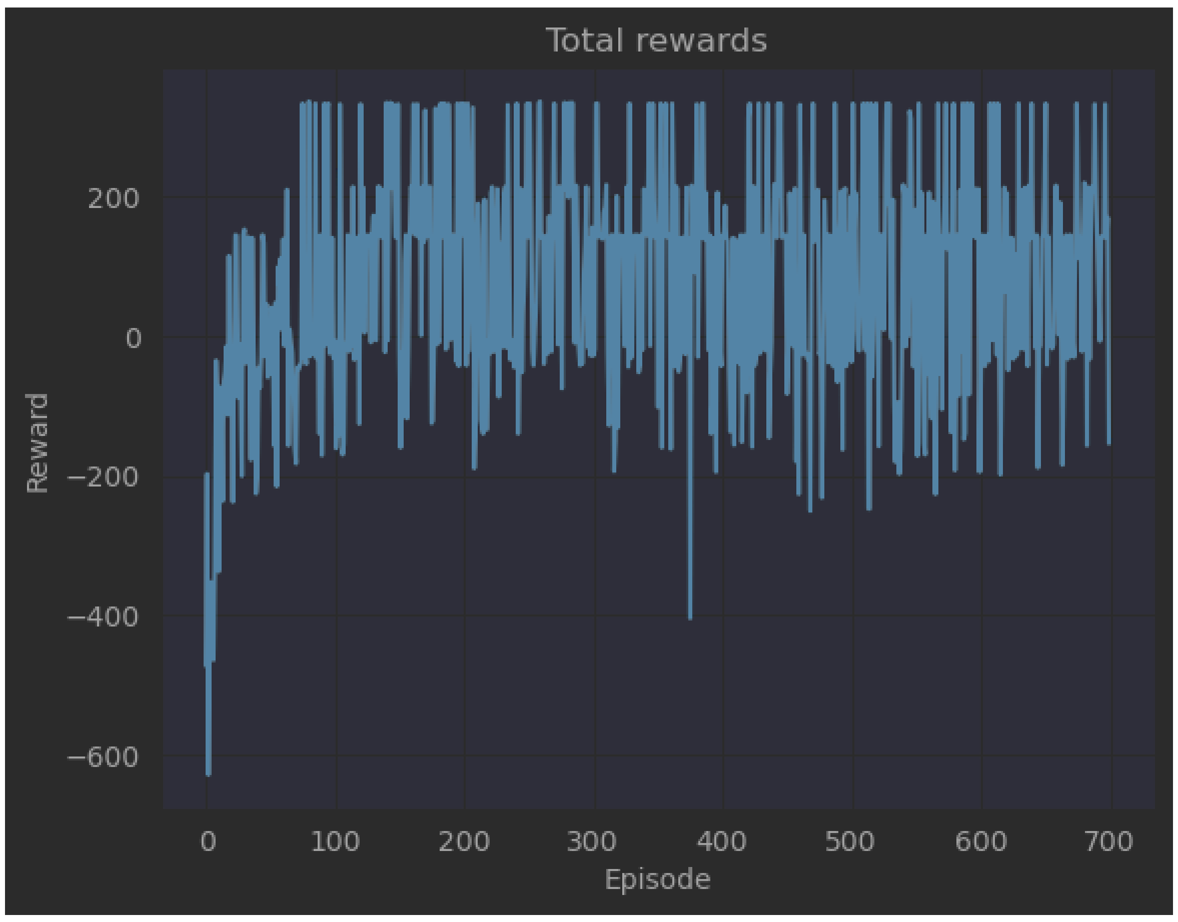
Environment configuration used in the training process:

For the training process of the model to not overfit it, I created 7 sample environments:

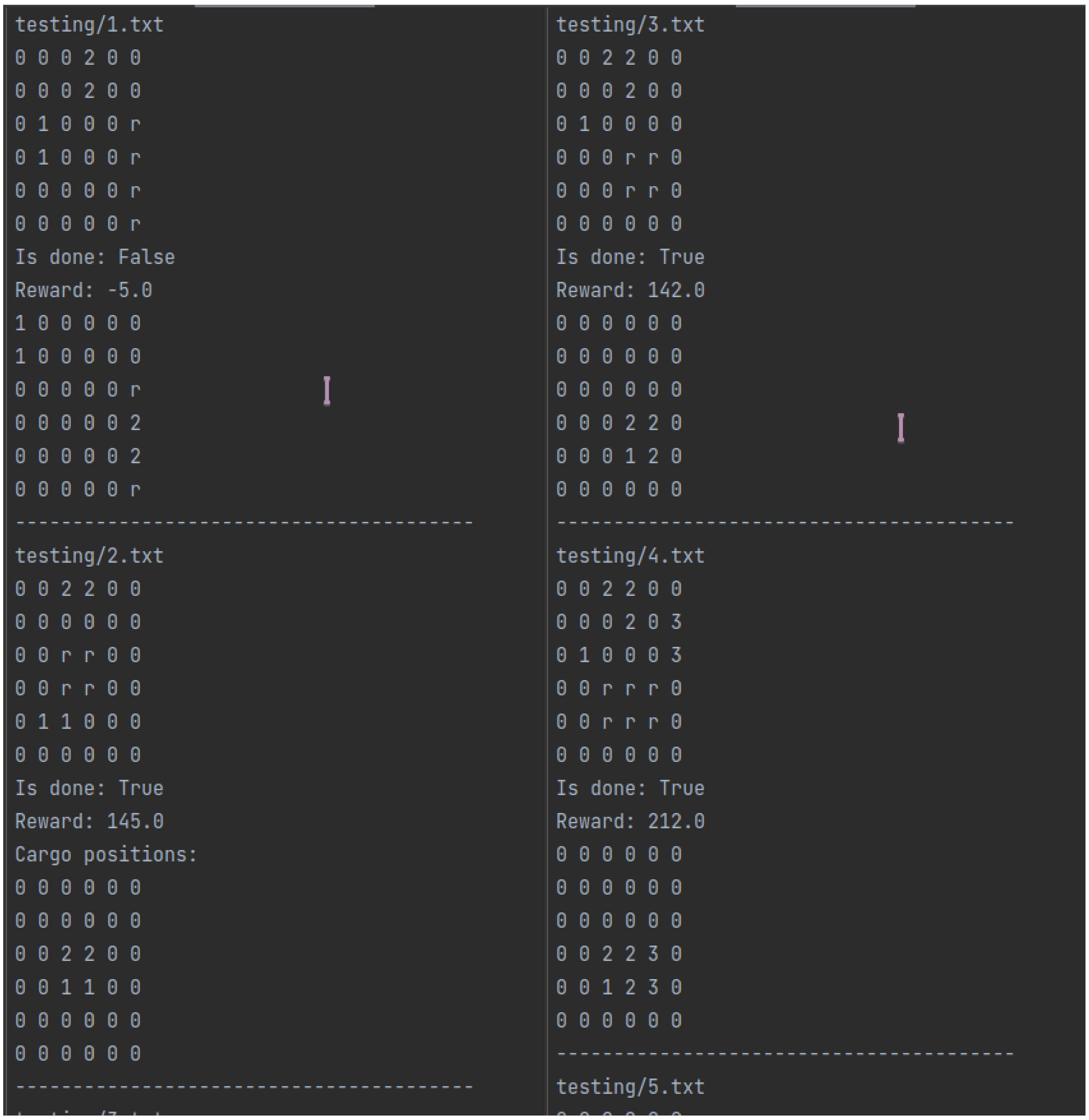
$$\begin{bmatrix} 0 & 0 & 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 2 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & r \\ 0 & 1 & 0 & 0 & 0 & r \\ 0 & 0 & 0 & 0 & 0 & r \\ 0 & 0 & 0 & 0 & 0 & r \end{bmatrix}, \begin{bmatrix} 0 & 0 & 2 & 2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & r & r & 0 & 0 \\ 0 & 0 & r & r & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}, \begin{bmatrix} 0 & 0 & 2 & 2 & 0 & 0 \\ 0 & 0 & 0 & 2 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & r & r & 0 \\ 0 & 0 & 0 & r & r & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}, \begin{bmatrix} 0 & 0 & 2 & 2 & 0 & 0 \\ 0 & 0 & 0 & 2 & 0 & 3 \\ 0 & 1 & 0 & 0 & 0 & 3 \\ 0 & 0 & r & r & r & 0 \\ 0 & 0 & r & r & r & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix},$$

$$\begin{bmatrix} 0 & 0 & 2 & 2 & 0 & 0 \\ 0 & 0 & 0 & 2 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & r & r & 0 \\ 0 & 0 & 0 & r & r & 0 \\ 3 & 3 & 0 & r & r & 0 \end{bmatrix}, \begin{bmatrix} 0 & 0 & 0 & 2 & 2 & 2 \\ r & r & r & 2 & 0 & 2 \\ r & r & r & 0 & 0 & 0 \\ r & r & r & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 \end{bmatrix}, \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 2 & 0 & 3 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & r & r & 4 \\ 0 & 6 & 0 & r & r & 0 \\ 0 & 0 & 0 & r & r & 5 \end{bmatrix},$$

And then I train model on these environments. To make model decision more general I chose 1 random enviroment of 7 defined above and train it. The result of training with 700 episodes are following:

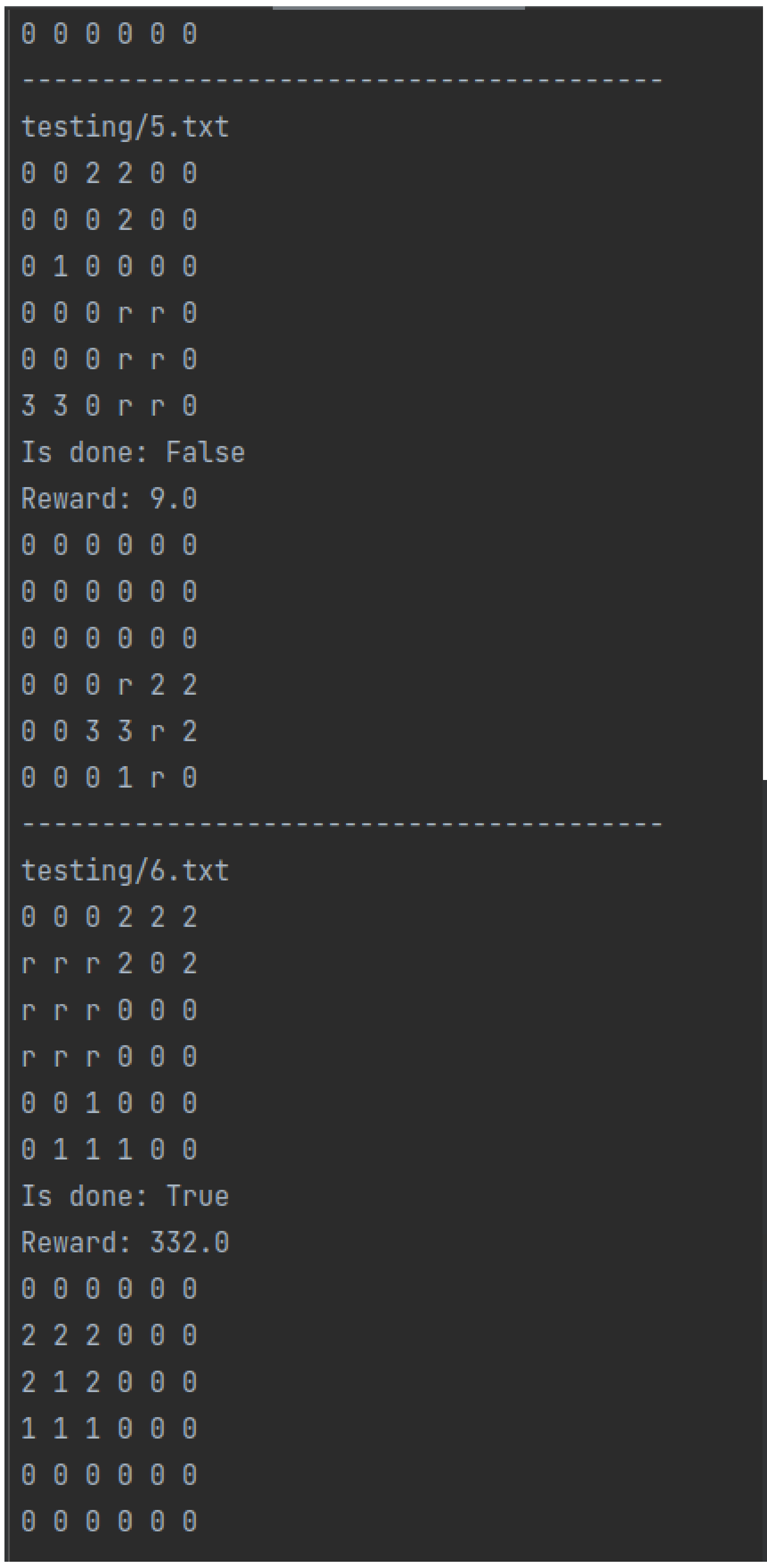


From the graph of rewards per Episode it's visible, that with time performance of the DQN becomes better. And for now, lets test given enviroment on wich we trained DQN without any random action(only with)

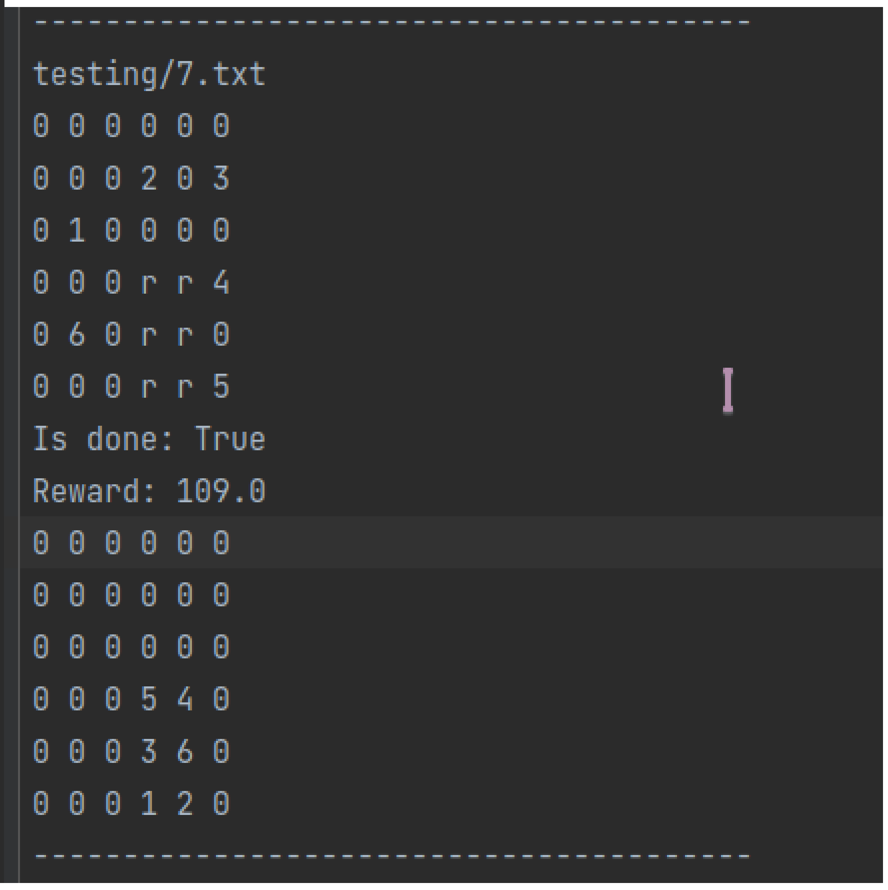


```
testing/1.txt
0 0 0 2 0 0
0 0 0 2 0 0
0 1 0 0 0 r
0 1 0 0 0 r
0 0 0 0 0 r
0 0 0 0 0 r
Is done: False
Reward: -5.0
1 0 0 0 0 0
1 0 0 0 0 0
0 0 0 0 0 r
0 0 0 0 0 2
0 0 0 0 0 2
0 0 0 0 0 r
----------------------------------------
testing/2.txt
0 0 2 2 0 0
0 0 0 0 0 0
0 0 r r 0 0
0 0 r r 0 0
0 1 1 0 0 0
0 0 0 0 0 0
Is done: True
Reward: 145.0
Cargo positions:
0 0 0 0 0 0
0 0 0 0 0 0
0 0 2 2 0 0
0 0 1 1 0 0
0 0 0 0 0 0
0 0 0 0 0 0
----------------------------------------
```

```
testing/3.txt
0 0 2 2 0 0
0 0 0 2 0 0
0 1 0 0 0 0
0 0 0 r r 0
0 0 0 r r 0
0 0 0 0 0 0
Is done: True
Reward: 142.0
0 0 0 0 0 0
0 0 0 0 0 0
0 0 0 0 0 0
0 0 0 2 2 0
0 0 0 1 2 0
0 0 0 0 0 0
----------------------------------------
testing/4.txt
0 0 2 2 0 0
0 0 0 2 0 3
0 1 0 0 0 3
0 0 r r r 0
0 0 r r r 0
0 0 0 0 0 0
Is done: True
Reward: 212.0
0 0 0 0 0 0
0 0 0 0 0 0
0 0 0 0 0 0
0 0 2 2 3 0
0 0 1 2 3 0
0 0 0 0 0 0
----------------------------------------
testing/5.txt
```



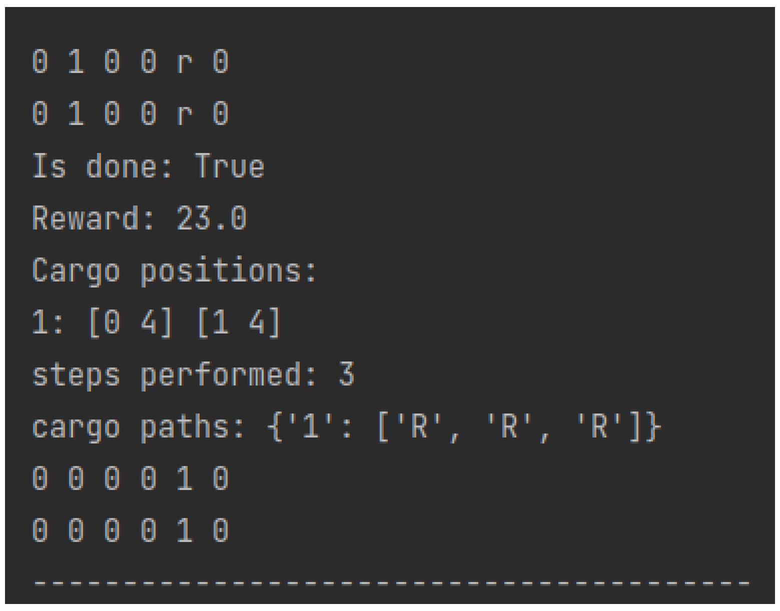
```
0 0 0 0 0 0
----------------------------------------
testing/5.txt
0 0 2 2 0 0
0 0 0 2 0 0
0 1 0 0 0 0
0 0 0 r r 0
0 0 0 r r 0
3 3 0 r r 0
Is done: False
Reward: 9.0
0 0 0 0 0 0
0 0 0 0 0 0
0 0 0 0 0 0
0 0 0 r 2 2
0 0 3 3 r 2
0 0 0 1 r 0
----------------------------------------
testing/6.txt
0 0 0 2 2 2
r r r 2 0 2
r r r 0 0 0
r r r 0 0 0
0 0 1 0 0 0
0 1 1 1 0 0
Is done: True
Reward: 332.0
0 0 0 0 0 0
2 2 2 0 0 0
2 1 2 0 0 0
1 1 1 0 0 0
0 0 0 0 0 0
0 0 0 0 0 0
```



```
----------------------------------------
testing/7.txt
0 0 0 0 0 0
0 0 0 2 0 3
0 1 0 0 0 0
0 0 0 r r 4
0 6 0 r r 0
0 0 0 r r 5
Is done: True
Reward: 109.0
0 0 0 0 0 0
0 0 0 0 0 0
0 0 0 0 0 0
0 0 0 5 4 0
0 0 0 3 6 0
0 0 0 1 2 0
----------------------------------------
```

After training, DQN can solve 5 out of 7 environments. It is not perfect result, but this is the best result I could achieve.

And now, lets check it on some environment, which our model didn't see before:



```
0 1 0 0 r 0
0 1 0 0 r 0
Is done: True
Reward: 23.0
Cargo positions:
1: [0 4] [1 4]
steps performed: 3
cargo paths: {'1': ['R', 'R', 'R']}
0 0 0 0 1 0
0 0 0 0 1 0
----------------------------------------
```

As we can see, this trained DQN works even in some environment, which it didn't saw before. That mean, that this model has potential to solve given task in general case.

Difficulties I encounter during the process of solving:

* The first problem I faced during the solution process is how to generalize the model so that it could interpret an environment of varying size, with a varying number of cargos and its shapes, and the size of the desired zone. I overcame that using CNN with the global average pooling. With this architecture the model didn't care about the size of the input matrix size.

- How to generate a random environment to make DQN learn problems in the general way, not just solve the given environment because of the random steps. To overcome this problem I generated 7 random environments, in which I trained my DQN. This is not the most general solution that can be proposed for the given task. But this is at least more general, than learning a model just in 1 static environment as I did firstly.

- How to optimize the DQN model to make it learn. This problem was solved by taking the code of training DQN for the cart pole in the 4'th lab and rewriting it according to the given task.

- Another one problem was to choose an appropriate reward function for the given game. To overcome this problem I just tried modifying the reward function until my DQN started learning at least of the static single environment.

- That state of the environment should I give to the model to make it learn something. To solve this problem i do the same as in the previous point. Just try different state representations that make sense, make experiments with the model and compare results. At the end choice variant with the best result.

Model architecture and environment representation, which i tried:

In my assignment solution I tried lots of different neural network architectures, state of the environment representations, parameters to calculate the reward. But generally, all my ideas and attempts can be divided into 2 parts, which I, after that tries to imporve:

1) Solution with the fully connected architecture of DQN:

In this class of my experiments I tried to make things work using NN with only dense layers (with Batch Normalization in some cases). But working with a fully connected network requires some specific state representation of the environment. I used several of them:

- 2 corner cells, which can represent particular cargo, 2 corner cells, which can represent desired space and Manhattan distance between current cargo and desired space.
  * 2 corner cells, which can represent particular cargo, 2 corner cells, which can represent desired space and total size of the Environment.
  Motivation to use this approach is that it was the first way which I came up with to deal with the varying environment.

Problem with this representation is that it don't know anything about other cargos in the environment. So, this solution can't deal with the overlapping cargos, during movement. One more problem, which makes game solutions impossible in some cases. In the situation, then all the cargos are already in the desired zone, but should resolve overlaps to complete the game. In this situation our model can't deal with this kind of situation, because it's dont know anything about other cargos. So, we should think about something else

2) Solution with the CNN model:

This idea was used in my final solution of the problem, because it works better in almost any case. Idea behind this type of architecture of state representation seems to me much more proper for the given task.
In this Idea I use CNN architecture of the NN with global average pooling and several fully connected layers at the end. And for the given model, state representation may be whole environment, but slightly changed to be perceived by the model (I already described above the way I change environment to make in input for CNN model).
My motivation to use this type of solution is that, from the state given to the model, it can differ current cargo, which it would move in the current step, other cargos, which are perceived as walls and desired zone. So I suppose, this idea have a potential to solve given task in general case.

What knowledge and sklls do I get:

- Experience of training DQN in the real task
- Learned about policy net and target net in details
- Learned about Global average pooling and other NN generalization techniques
- Learned about attention in neural networks. But had no time to understand how does it works and try it in the given task.