FLATLAND

Giulia Cantini

Challenge presentation

How can trains learn to automatically coordinate among themselves, so that there are minimal delays in large train networks?

https://www.aicrowd.com/challenges/flatland-challenge
https://gitlab.aicrowd.com/flatland/flatland

Vehicle Rescheduling Problem (VRSP)

Li, Mirchandani, Borenstein 2007:

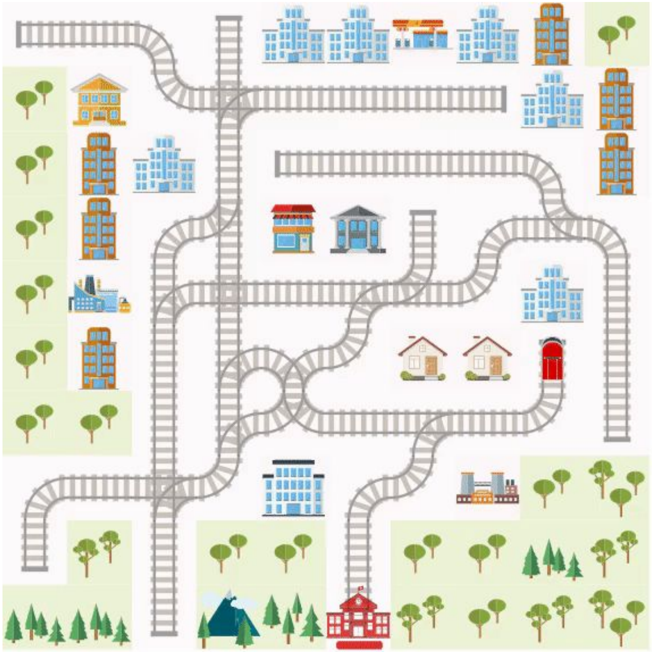
“The vehicle rescheduling problem (VRSP) arises when a previously assigned trip is disrupted. A traffic accident, a medical emergency, or a breakdown of a vehicle are examples of possible disruptions that demand the rescheduling of vehicle trips. The VRSP can be approached as a dynamic version of the classical vehicle scheduling problem (VSP) where assignments are generated dynamically.”

- ➔ real world problem
- ➔ affects transportation and logistics of many complex systems (not only railways)

Round 0: Learn to navigate (beta)

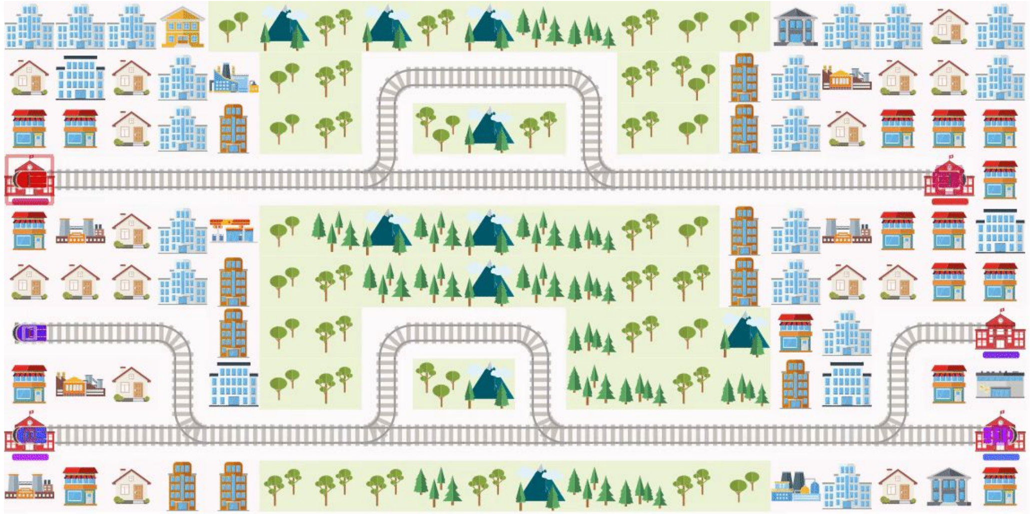
One agent must navigate from starting point to target given a random infrastructure

(shortest path).



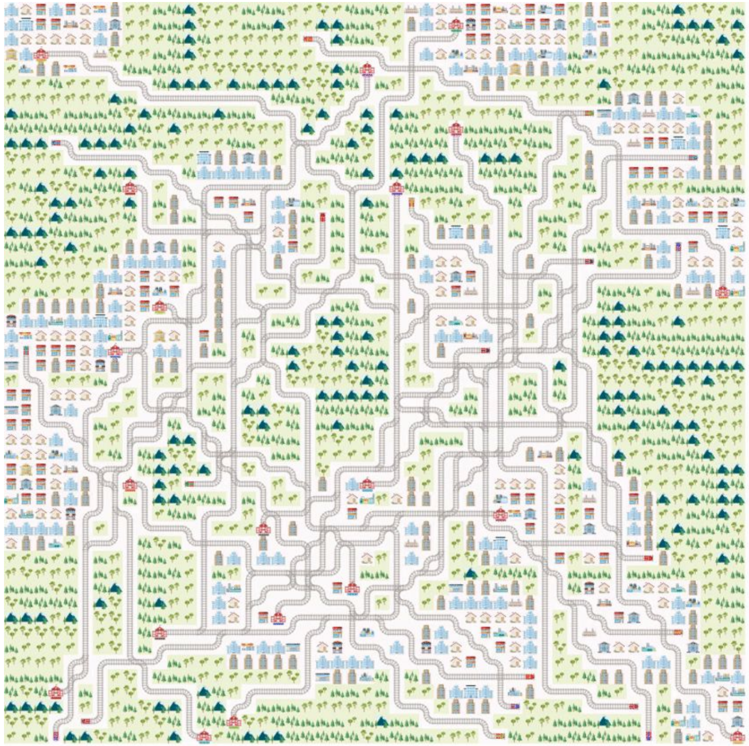
Round 1: Avoid conflicts

Multiple agents have to find a way to reach their targets. In this scenario it is likely to encounter resource conflicts when two or more agents simultaneously plan to occupy the same section of infrastructure (avoid conflicts + shortest path)



Round 2: Optimize train traffic

In the same scenario as Round 1, different agents may have different speeds.
Complexity increases by considering that slower trains could slow down the faster ones.



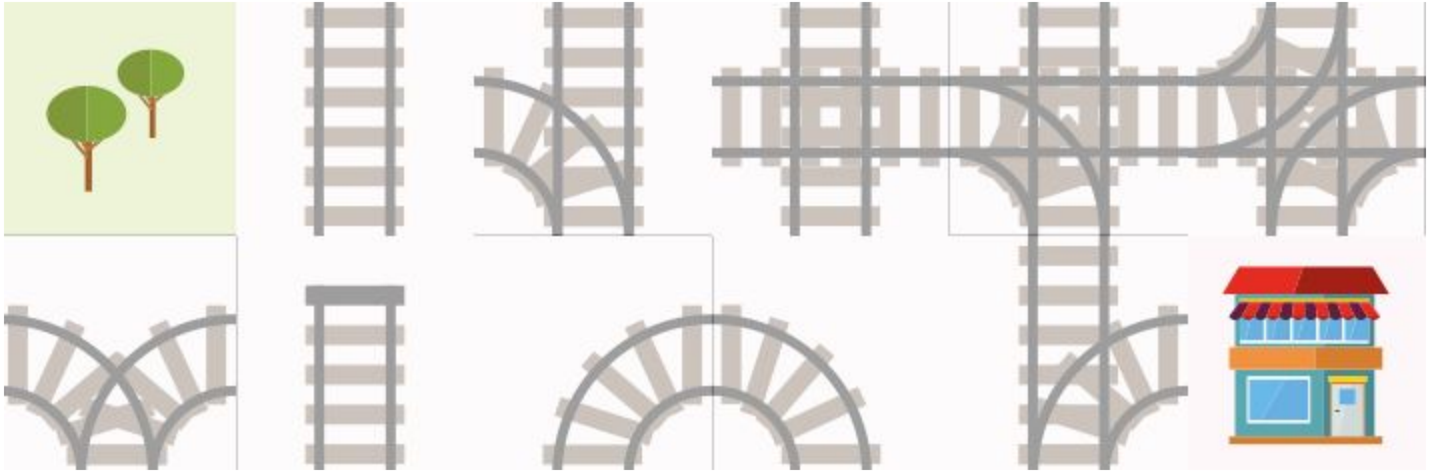
Environment: grid

```python
# Example generate a rail given a manual specification,
# a map of tuples (cell_type, rotation)
# cell_type is in [0..10] where
# 0: empty cell (e.g. a building)
# 1: straight
# 2: simple switch
# 3: diamond crossing
# 4: single slip
# 5: double slip
# 6: symmetrical
# 7: dead end
# 8: turn left
# 9: turn right
# 10: mirrored switch
specs = [[(0, 0), (1, 0), (2, 0), (3, 0), (4, 0), (5, 0)],
         [(6, 0), (7, 0), (2, 180), (9, 0), (2, 270), (0, 0)],
         [(7, 270), (1, 90), (2, 0), (1, 90), (2, 90), (7, 90)],
         [(0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0)]]
```

Cell types, manual specification

Cell types: visualization

0: Empty | 1: Straight | 2: Simple switch | 3: Diamond crossing | 4: Single slip | 5: Double slip



6: Symmetrical | 7: Dead end | 8: Turn left | 9: Turn right | 10: Mirrored switch

Cell types: rotation

e.g. simple switch

rotation = 0°

rotation = 90°

rotation = 180°

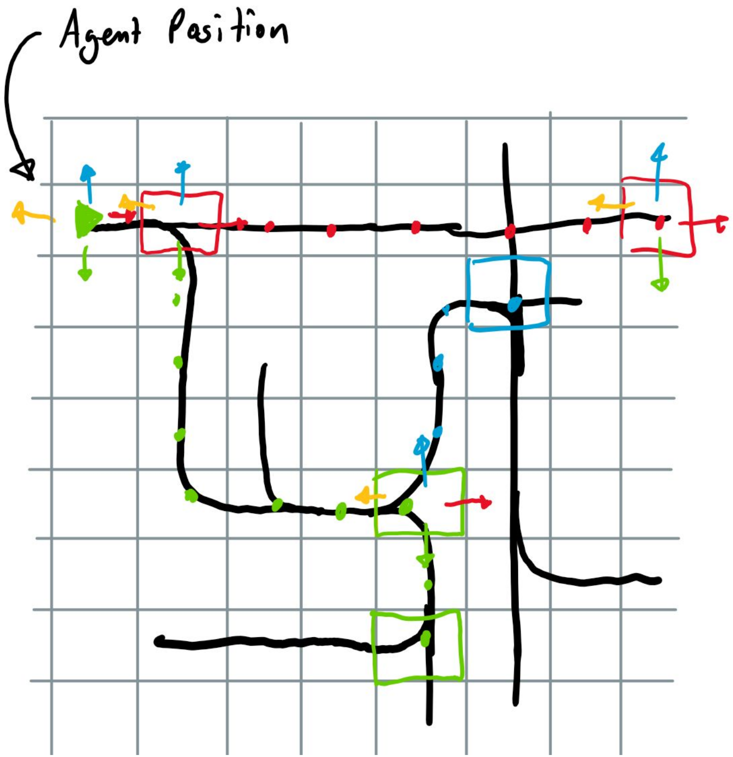
rotation = 270°

Actions

In Flatland there are five possible actions, limited to the railway networks:

1. **Do nothing**: continue moving or stay still
2. **Deviate left**: turn left at switch and move to the next cell; if the agent was not moving, movement is started
3. **Go forward**: move to the next cell in front of the agent; if the agent was not moving, movement is started
4. **Deviate right**: same as “deviate left” but for right turns.
5. **Stop**: stop moving, this is used only in multi-agent setups to avoid conflicts.

(Tree) observations

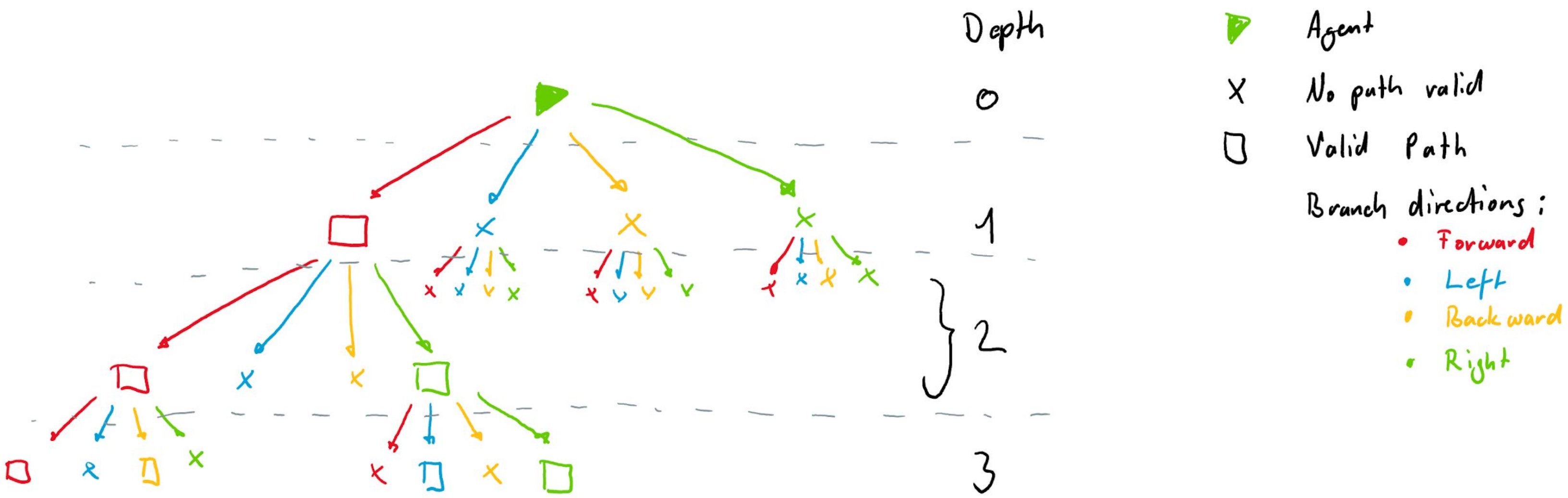


The railway network is a graph → observations are built only along allowed transitions on the graph

As the agent moves along these transitions, a tree is built up, where a new node is created at every cell where the agent has different possibilities (switch), dead-end, or the target is reached.

(Tree) observations: depth

An observation is always built according to the **orientation** of the agent at a given node.
Each node has always 4 branches: left, forward, right, backward.



Observations vector - general

Composed of 4 sequential parts corresponding to to data from the (up to 4) possible movements, sorted relative to the **current orientation** of the agent.

[data from ‘left’] + [data from ‘forward’] + [data from ‘right’] + [data from ‘back’]

Each branch data is organized as:

- [root node information] +
- [recursive branch data from ‘left’] +
- [recursive branch data from ‘forward’] +
- [recursive branch data from ‘right’] +
- [recursive branch data from ‘back’]

Node features - node features

In explore_branch():

```python
else:
    observation = [own_target_encountered,
                   other_target_encountered,
                   other_agent_encountered,
                   potential_conflict,
                   unusable_switch,
                   tot_dist,
                   self.distance_map[handle, position[0], position[1], direction],
                   other_agent_same_direction,
                   other_agent_opposite_direction,
                   ]
```

```
<class 'dict'>: {0: [0, 0, 0, 0, 0, 0, 7.0, 0, 0, -inf, -inf, -inf, -inf, -inf, -inf, -inf, -inf, -inf, -inf, -inf,
-inf, -inf, -inf, -inf, -inf, -inf, -inf, -inf, -inf, -inf, -inf, -inf, -inf, -inf, -inf, -inf, -inf, -inf, -inf,
-inf, -inf, -inf, -inf, -inf, -inf, -inf, -inf, -inf, -inf, -inf, -inf, -inf, -inf, -inf, inf, inf, inf, inf, 2, 3,
4.0, 0, 0, 7, inf, inf, inf, inf, 7, 0, 0, 0, -inf, -inf, -inf, -inf, -inf, -inf, -inf, -inf, -inf, inf, inf, inf,
inf, inf, 4, 15.0, 0, 0, -inf, -inf, -inf, -inf, -inf, -inf, -inf, -inf, -inf, -inf, -inf, -inf, -inf, -inf, -inf,
-inf, -inf, -inf, -inf, -inf, -inf, -inf, -inf, -inf, -inf, -inf, -inf, -inf, -inf, -inf, -inf, -inf, -inf, -inf,
-inf, -inf, -inf, -inf, -inf, -inf, -inf, -inf, -inf, -inf, -inf, -inf, -inf, -inf, -inf, -inf, -inf, -inf, -inf,
-inf, -inf, -inf, -inf, -inf, -inf, -inf, -inf, -inf, -inf, -inf, -inf, -inf, -inf, -inf, -inf, -inf, -inf, -inf,
-inf, -inf, -inf, -inf, -inf, -inf, -inf, -inf, -inf, -inf, -inf, -inf, -inf, -inf, -inf, -inf, -inf, -inf, -inf,
-inf, -inf, -inf, -inf, -inf, -inf, -inf, -inf]}
```

Observations vector - node features (part 1)

Each node info is composed of 9 features:

1. distance as number of cells from own target (if detected)
2. distance from another agent target (if detected)
3. distance from another agent (if detected)
4. potential conflict detected
   tot_dist : distance from another agent in a possible conflict (prediction)
   0 : there aren't any predicted conflicts
5. unusable switch

Observations vector - node features (part 2)

6. distance to the next branching (switch, dead-end or target node)

7. minimum distance from node to the agent’s target given the direction
   (if this path is chosen)

8. other agents in the same direction
   0 / n : number of agents in the same direction

9. other agents in the opposite direction
   0 / n : number of agents in the opposite direction

Building an observation (part 1)

```
In case of the root node, the values are [0, 0, 0, 0, distance from agent to target].
In case the target node is reached, the values are [0, 0, 0, 0, 0].
"""

# Update local lookup table for all agents' positions
self.location_has_agent = {tuple(agent.position): 1 for agent in self.env.agents}
self.location_has_agent_direction = {tuple(agent.position): agent.direction for agent in self.env.agents}
if handle > len(self.env.agents):
    print("ERROR: obs _get - handle ", handle, " len(agents)", len(self.env.agents))
agent = self.env.agents[handle]  # TODO: handle being treated as index
possible_transitions = self.env.rail.get_transitions(*agent.position, agent.direction)
num_transitions = np.count_nonzero(possible_transitions)

# Root node - current position
observation = [0, 0, 0, 0, 0, 0, self.distance_map[(handle, *agent.position, agent.direction)], 0, 0]
```

get()

Building an observation (part 2)

```
visited = set()
# Start from the current orientation, and see which transitions are available;
# organize them as [left, forward, right, back], relative to the current orientation
# If only one transition is possible, the tree is oriented with this transition as the forward branch.
orientation = agent.direction

if num_transitions == 1:
    orientation = np.argmax(possible_transitions)

for branch_direction in [(orientation + i) % 4 for i in range(-1, 3)]:
    if possible_transitions[branch_direction]:
        new_cell = self._new_position(agent.position, branch_direction)
        branch_observation, branch_visited = \
            self._explore_branch(handle, new_cell, branch_direction, 1, 1)
        observation = observation + branch_observation
        visited = visited.union(branch_visited)
    else:
        # add cells filled with infinity if no transition is possible
        observation = observation + [-np.inf] * self._num_cells_to_fill_in(self.max_depth)
self.env.dev_obs_dict[handle] = visited
return observation
```

Reward function

It costs each agent a **step_penalty** for every time step taken in the environment, independently of the movement. Other penalties such as penalty for stopping, starting and invalid actions are (currently) set to 0.

$$alpha = 1;\ beta = 1$$

Reward function parameters:

- invalid action penalty = 0
- step penalty = -alpha
- global reward = beta
- stop penalty = 0
- start penalty = 0

Training: single agent navigation (part 1)

```
# Reset score and done
score = 0
env_done = 0

# Run episode
for step in range(max_steps):

    # Only render when not triaing
    if not Training:
        env_renderer.renderEnv(show=True, show_observations=True)

    # Chose the actions
    for a in range(env.get_num_agents()):
        if not Training:
            eps = 0

        action = agent.act(agent_obs[a], eps=eps)
        action_dict.update({a: action})

        # Count number of actions takes for statistics
        action_prob[action] += 1

    # Environment step
    next_obs, all_rewards, done, _ = env.step(action_dict)

    for a in range(env.get_num_agents()):
        rail_data, distance_data, agent_data = split_tree(tree=np.array(next_obs[a]),
                                                          num_features_per_node=num_features_per_node,
                                                          current_depth=0)
        rail_data = norm_obs_clip(rail_data)
        distance_data = norm_obs_clip(distance_data)
        agent_data = np.clip(agent_data, -1, 1)
        agent_next_obs[a] = np.concatenate((np.concatenate((rail_data, distance_data)), agent_data))
```

Training: single agent navigation (part 2)

```
        # Update replay buffer and train agent
        for a in range(env.get_num_agents()):

            # Remember and train agent
            if Training:
                agent.step(agent_obs[a], action_dict[a], all_rewards[a], agent_next_obs[a], done[a])

            # Update the current score
            score += all_rewards[a] / env.get_num_agents()

        agent_obs = agent_next_obs.copy()
        if done['__all__']:
            env_done = 1
            break

    # Epsilon decay
    eps = max(eps_end, eps_decay * eps)  # decrease epsilon

    # Store the information about training progress
    done_window.append(env_done)
    scores_window.append(score / max_steps)  # save most recent score
    scores.append(np.mean(scores_window))
    dones_list.append((np.mean(done_window)))

    print(
        '\rTraining {} Agents on ({},{}).\t Episode {}\t Average Score: {:.3f}\tDones: {:.2f}%\tEpsilon: {:.2f} \t Action Probabilities: \t {}'.format(
            env.get_num_agents(), x_dim, y_dim,
            trials,
            np.mean(scores_window),
            100 * np.mean(done_window),
            eps, action_prob / np.sum(action_prob)), end=" ")

    if trials % 100 == 0:
        print(
            '\rTraining {} Agents on ({},{}).\t Episode {}\t Average Score: {:.3f}\tDones: {:.2f}%\tEpsilon: {:.2f} \t Action Probabilities: \t {}'.format(
                env.get_num_agents(), x_dim, y_dim,
                trials,
                np.mean(scores_window),
                100 * np.mean(done_window),
                eps, action_prob / np.sum(action_prob)))
        torch.save(agent.qnetwork_local.state_dict(),
                   './Nets/navigator_checkpoint_tree_depth' + str(tree_max_depth) + '_' + str(trials) + '.pth')
        action_prob = [1] * action_size
```

Dueling Double DQN (“FC”)

```python
class QNetwork(nn.Module):
    def __init__(self, state_size, action_size, seed, hidsize1=128, hidsize2=128):
        super(QNetwork, self).__init__()

        self.fc1_val = nn.Linear(state_size, hidsize1)
        self.fc2_val = nn.Linear(hidsize1, hidsize2)
        self.fc3_val = nn.Linear(hidsize2, 1)

        self.fc1_adv = nn.Linear(state_size, hidsize1)
        self.fc2_adv = nn.Linear(hidsize1, hidsize2)
        self.fc3_adv = nn.Linear(hidsize2, action_size)

    def forward(self, x):
        val = F.relu(self.fc1_val(x))
        val = F.relu(self.fc2_val(val))
        val = self.fc3_val(val)

        # advantage calculation
        adv = F.relu(self.fc1_adv(x))
        adv = F.relu(self.fc2_adv(adv))
        adv = self.fc3_adv(adv)
        return val + adv - adv.mean()
```

Training algorithm - params and config

```python
BUFFER_SIZE = int(1e5)  # replay buffer size
BATCH_SIZE = 512  # minibatch size
GAMMA = 0.99  # discount factor 0.99
TAU = 1e-3  # for soft update of target parameters
LR = 0.5e-4  # learning rate 5
UPDATE_EVERY = 10  # how often to update the network
double_dqn = True  # If using double dqn algorithm
input_channels = 5  # Number of Input channels
```

```python
class Agent:
    """Interacts with and learns from the environment."""

    def __init__(self, state_size, action_size, net_type, seed, double_dqn=True, input_channels=5):
        """Initialize an Agent object.

        Params
        ======
            state_size (int): dimension of each state
            action_size (int): dimension of each action
            seed (int): random seed
        """
        self.state_size = state_size
        self.action_size = action_size
        self.seed = random.seed(seed)
        self.version = net_type
        self.double_dqn = double_dqn
        # Q-Network
        if self.version == "Conv":
            self.qnetwork_local = QNetwork2(state_size, action_size, seed, input_channels).to(device)
            self.qnetwork_target = copy.deepcopy(self.qnetwork_local)
        else:
            self.qnetwork_local = QNetwork(state_size, action_size, seed).to(device)
            self.qnetwork_target = copy.deepcopy(self.qnetwork_local)

        self.optimizer = optim.Adam(self.qnetwork_local.parameters(), lr=LR)

        # Replay memory
        self.memory = ReplayBuffer(action_size, BUFFER_SIZE, BATCH_SIZE, seed)
        # Initialize time step (for updating every UPDATE_EVERY steps)
        self.t_step = 0
```

Training algorithm - learn step

```
def learn(self, experiences, gamma):
    """Update value parameters using given batch of experience tuples.

    Params
    ======
        experiences (Tuple[torch.Tensor]): tuple of (s, a, r, s', done) tuples
        gamma (float): discount factor
    """
    states, actions, rewards, next_states, dones = experiences

    # Get expected Q values from local model
    Q_expected = self.qnetwork_local(states).gather(1, actions)

    if self.double_dqn:
        # Double DQN
        q_best_action = self.qnetwork_local(next_states).max(1)[1]
        Q_targets_next = self.qnetwork_target(next_states).gather(1, q_best_action.unsqueeze(-1))
    else:
        # DQN
        Q_targets_next = self.qnetwork_target(next_states).detach().max(1)[0].unsqueeze(-1)

        # Compute Q targets for current states

    Q_targets = rewards + (gamma * Q_targets_next * (1 - dones))

    # Compute loss
    loss = F.mse_loss(Q_expected, Q_targets)
    # Minimize the loss
    self.optimizer.zero_grad()
    loss.backward()
    self.optimizer.step()

    # ------------------- update target network ------------------- #
    self.soft_update(self.qnetwork_local, self.qnetwork_target, TAU)
```

Training algorithm - act step

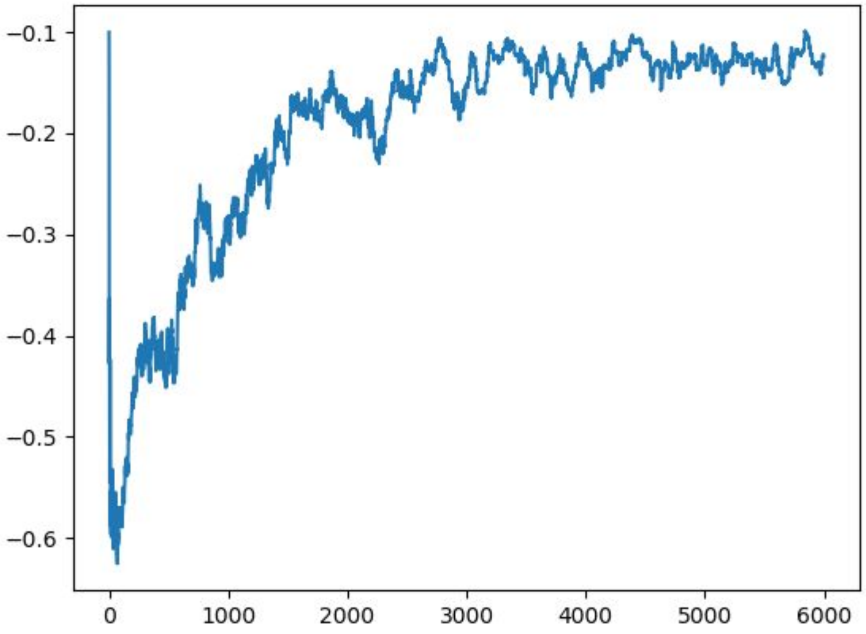
```python
def act(self, state, eps=0.):
    """Returns actions for given state as per current policy.

    Params
    ======
        state (array_like): current state
        eps (float): epsilon, for epsilon-greedy action selection
    """
    state = torch.from_numpy(state).float().unsqueeze(0).to(device)
    self.qnetwork_local.eval()
    with torch.no_grad():
        action_values = self.qnetwork_local(state)
    self.qnetwork_local.train()

    # Epsilon-greedy action selection
    if random.random() > eps:
        return np.argmax(action_values.cpu().data.numpy())
    else:
        return random.choice(np.arange(self.action_size))
```

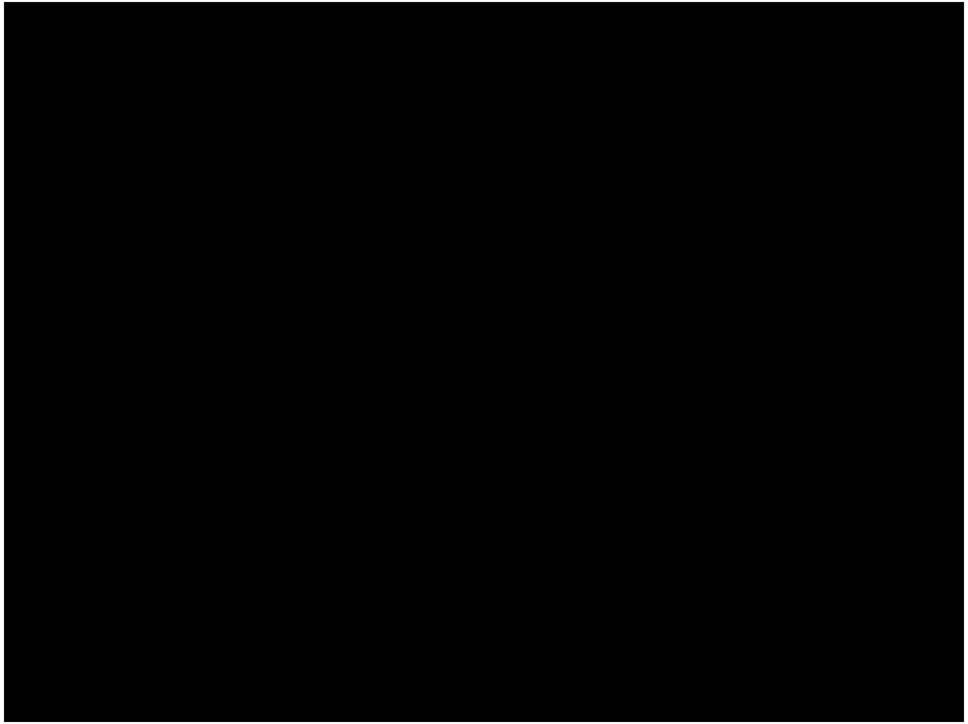
Learning curve

scores



num of trials

Video example



Thank you for your attention!