Project Report

Deep Learning (Flatland Challenge)

by Jia Liang Zhou, Hanying Zhang

July, 2021

1 The Flatland problem

1.1 Introduction

Flatland[1] challenge aims to solve the problem of efficiently manange dense traffic on complex railway networks. It is a competition to foster progress in multi-agent reinforcement learning for any *re-scheduling problem (RSP)*. The challenge addresses a real-world problem faced by many transportation and logistics companies around the world (such as the *Swiss Federal Railways, SBB*. Different tasks related to RSP on a simplified 2D multi-agent railway simulation must be solved.

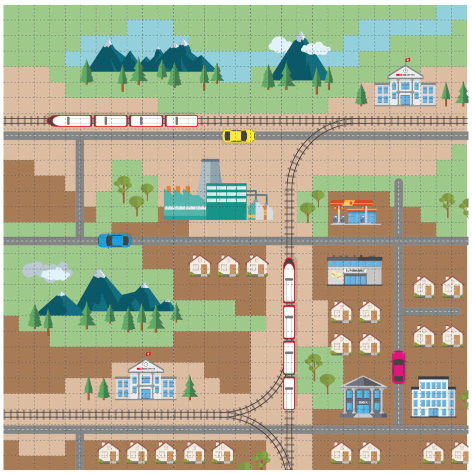

Figure 1. The Flatland problem

1.2 Environments

1.2.1 Observations

A central question while designing an agent is the observations(states) used to take decisions. There are three provided observations, the Global Observation, the Local Grid Observation and the Tree Observation. We can either work with one of the provided observations or design an improved one ourselves.

1.2.2 Actions

The trains in Flatland have strongly limited movements, as you would expect from a railway simulation. This means that only a few actions are valid in most cases. There are five possible actions:

1. DO_NOTHING

1. https://www.aicrowd.com/challenges/flatland

2. MOVE_LEFT
3. MOVE_RIGHT
4. MOVE_FORWARD
5. STOP_MOVING

1.2.3 Rewards

At each time step, each agent receives a combined reward which consists of a local and a global reward signal.

Locally, the agent receives $r_l = -1$ for each time step, and $r_l = 0$ for each time step after it has reached its target location. The global reward signal $r_g = 0$ only returns a non-zero value when all agents have reached their targets, in which case it is worth $r_g = 1$.

Every agent i receives a reward:

$r_i(t) = \alpha r_l(t) + \beta r_g(t)$

where α and β are factors for tuning collaborative behavior. This reward creates an objective of finishing the episode as quickly as possible in a collaborative way.

In this challenge, the values used are: $\alpha = 1.0$ and $\beta = 1.0$.

1.3 Tasks and Metrics

The goal of this challenge is making all the trains arrive at their destinations with minimal travel time, i.e., minimizing the number of stps that it takes for each agent to reach its destination. As the difficulty increases, the agents have to be able to plan ahead.

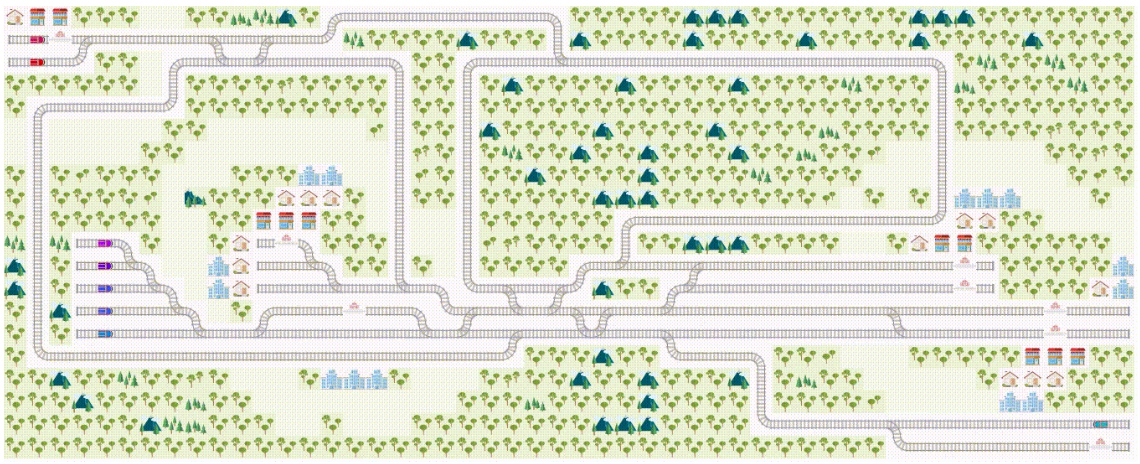

Figure 2. One problem example

The primary metric is the **normalized return** from the agents - the higher the better. For each agent, the minimum possible value is 0.0, which occurs when none of the agents reach their goal. The maximum possible value is 1.0, when all agents reach their targets.

2 DQN and its Improvements

2.1 DQN

The DQN (Deep Q-Network) algorithm was developed by DeepMind in 2015. It was able to solve a wide range of Atari games (some to superhuman level) by combining reinforcement learning and deep neural networks at scale. The algorithm was developed by enhancing a classic RL algorithm called Q-Learning with deep neural networks and a technique called *experience replay*.

For most problems, it is impractical to represent the Q-function as a table containing values for each combination of s and a. Instead, we train a function approximator, such as a neural network with parameters θ, to estimate the Q-values, i.e. $Q(s,a;\theta) \approx Q^*(s,a)$.

The loss function at each time step is:

$$L(\theta) = \mathbb{E}\Big[\Big(r + \gamma \max_{a'} Q(s',a';\theta) - Q(s,a;\theta)\Big)^2\Big]$$

where r is the reward of this time step and γ is the discount factor.

The new Q value is:

$$Q'(s',a;\theta) = Q(s,a;\theta) + \alpha\Big[r + \gamma \max_{a'} Q(s',a';\theta) - Q(s,a;\theta)\Big]$$

where α is the learning rate.

Alternatively we could take only state as input and output the Q-value for each possible action. This approach has the advantage, that if we want to perform a Q-value update or pick the action with highest Q-value, we only have to do one forward pass through the network and have all Q-values for all actions immediately available.

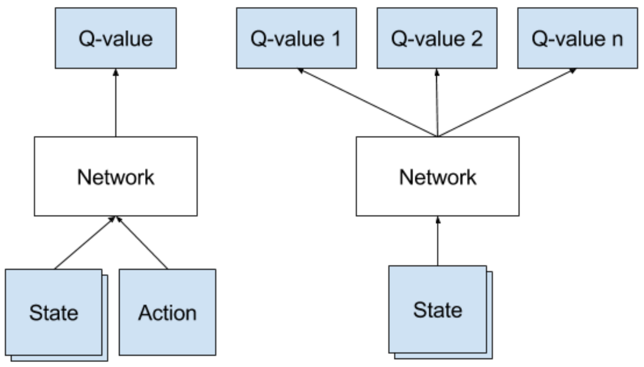

Figure 3. Optimal architecture of deep Q-network

2.2 Improvements on DQN

2.2.1 Fixed Q-targets

The problem of standard DQN is that we using the same parameters (weights) for estimating the target and the Q value. As a consequence, there is a big correlation between the TD target and the parameters we are changing.

Therefore, it means that at every step of training, our Q values shift but also the target value shifts. So, we're getting closer to our target but the target is also moving. It's like chasing a moving target. This lead to a big oscillation in training.

We can use the idea of fixed Q-targets introduced by DeepMind:

- Using a separate network with a fixed parameter(θ^-) for estimating the TD target.
- At every τ steps, we copy the parameters from our DQN network to update the target network.

2.2.2 Double DQN

Double DQN, or double Learning, was introduced by *Hado van Hasselt*[2]. This method handles the problem of the overestimation of Q-values.

At the beginning of the training we don't have enough information about the best action to take. A maximum over estimated values is used implicitly as an estimation of the maximum (true) value, which could lead to a significant positive bias. If non-optimal actions are regularly given a higher Q value than the optimal best action, the learning will be complicated.

The solution is: when we compute the Q target, we use two networks to decouple the action selection from the target Q value generation. We:

- use our DQN network to select what is the best action to take for the next state (the action with the highest Q value).
- use our target network to calculate the target Q value of taking that action at the next state.

$$\underbrace{Q(s,a)}_{\text{TD target}} = r(s,a) + \gamma \underbrace{Q(s', \underbrace{argmax_a Q(s',a)}_{\text{DQN Network choose action for next state}})}_{\text{Target network calculates the Q value of taking that action at that state}}$$

Figure 4. Double DQN

2.2.3 Dueling DQN

Dueling DQN was introduced by Ziyu Wang.et al[3]. The Q-values in DQN correspond to how good it is to be at that state and taking an action at that state Q(s,a). So we can decompose Q(s,a) as the sum of:

- V(s): the value of being at that state
- A(s,a): the advantage of taking that action at that state (how much better is to take this action versus all other possible actions at that state).

$$Q(s,a) = A(s,a) + V(s)$$

With Dueling DQN, we want to separate the estimator of these two elements, using two new streams:

- one that estimates the state value V(s)

2. https://papers.nips.cc/paper/3964-double-q-learning
3. https://arxiv.org/abs/1511.06581

- one that estimates the advantage for each action A(s,a)

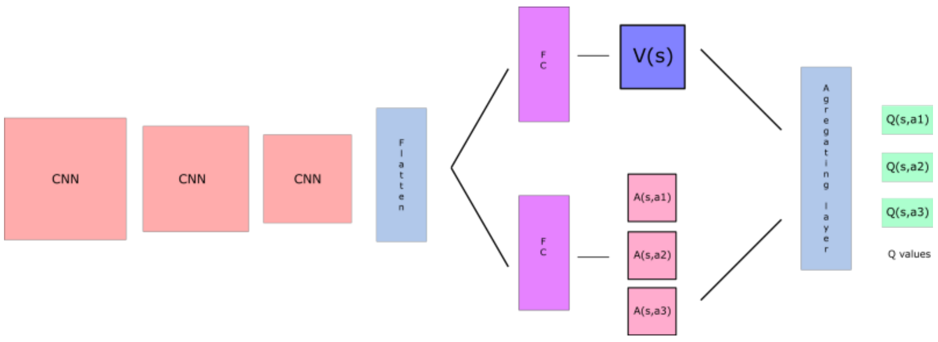

Figure 5. Dueling DQN structure

By decoupling the estimation, intuitively our Dueling DQN can learn which states are (or are not) valuable without having to learn the effect of each action at each state (since it's also calculating V(s)).

3 GNN

Gori et al. (2005)[4] proposed a novel neural network model capable of processing graph structure data–graph neural network in 2005. Graph Neural Networks (GNNs) are a class of deep learning methods designed to perform inference on data described by graphs, i.e. data is generated from non-Euclidean domains. GNNs are neural networks that can be directly applied to graphs, and provide an easy way to do node-level, edge-level, and graph-level prediction tasks.

The intuition of GNN is that nodes are naturally defined by their neighbors and connections. To understand this we can simply imagine that if we remove the neighbors and connections around a node, then the node will lose all its information. Therefore, the neighbors of a node and connections to neighbors define the concept of the node.

The related concepts are introduced as follows: let the input graph be $G=(V,E,\mathbf{X}_V,\mathbf{X}_E)$, $V=\{v_1,v_2,\ldots,v_n\}$ represents the set of nodes, and $E=\{(i,j)|\text{when } v_i \textit{ is adjacent to } v_j\}$ is the set of edges. $\mathbf{x}_i$ denotes the feature vector of node v_i, and $\mathbf{X}_V=\{\mathbf{x}_1,\mathbf{x}_2,\ldots,\mathbf{x}_n\}$ is the set of feature vectors of all nodes. $\mathbf{x}_{(i,j)}$ denotes the feature vector of edge (i,j), and $\mathbf{X}_E=\{\mathbf{x}_{(i,j)}|(i,j)\in E\}$ is the set of feature vectors of all edges.

The input graph G is converted into a dynamic graph $G^t=(V,E,X_V,X_E,H^t)$ in the graph neural network model, where $t=1,2,\ldots,T$ represents time and $H^t=\left(h_1^{(t)},h_2^{(t)},\ldots,h_n^{(t)}\right)$, $h_i^{(t)}$represents the state vector of node v_i at time t, which depends on the graph G^{t-1} at time t–1. The equation of $h_i^{(t)}$ is as follows:

$$h_i^{(t)} = f_w(x_i, x_{\mathrm{co}(i)}, h_{\mathrm{ne}(i)}^{t-1}, x_{\mathrm{ne}(i)})$$

where $f_w(\cdot)$ denotes the local transformation function with parameter w, $x_{ne(i)}$ is the set of feature vectors of all nodes adjacent to node v_i, $x_{co(i)}$ is the set of feature vectors of all edges connected to node v_i, and $h_{\mathrm{ne}(i)}^{(t)}$ is the set of state vectors of all nodes adjacent to node v_i at time t.

GNN updates the node status in an iterative manner, and this process is shown in the figure below, where *A* shows the input graph, *B* shows each node iteration from time *t-1* to *t*, *C* shows the overall iteration process.

4. https://www.frontiersin.org/articles/10.3389/fgene.2021.690049/full#B30

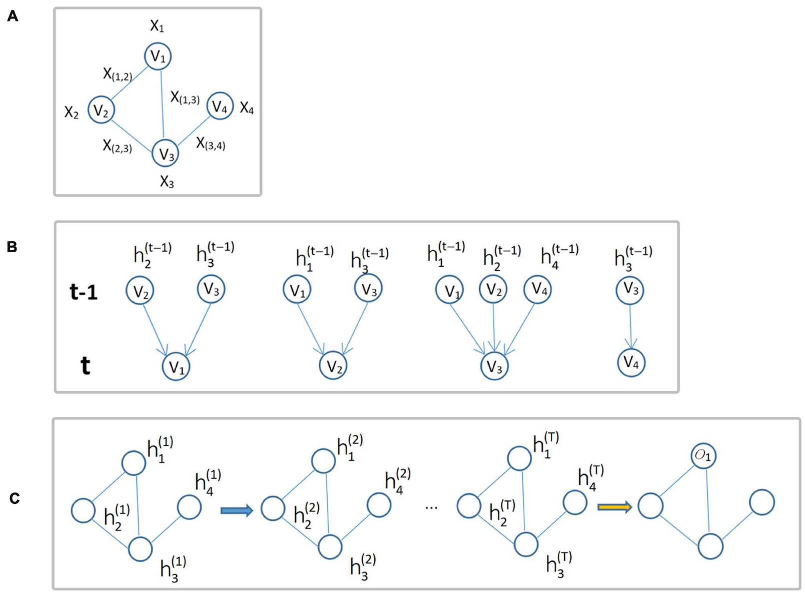

Figure 6. GNN - update of node status

3.1 GCN

Graph Convolutional Networks developed by Thomas Kipf and Max Welling[5]. GCNs perform similar operations of CNNs where the model learns the features by inspecting neighboring nodes. The major difference between CNNs and GNNs is that CNNs are specially built to operate on regular (Euclidean) structured data, while GNNs are the generalized version of CNNs where the numbers of nodes connections vary and the nodes are unordered (irregular on non-Euclidean structured data).

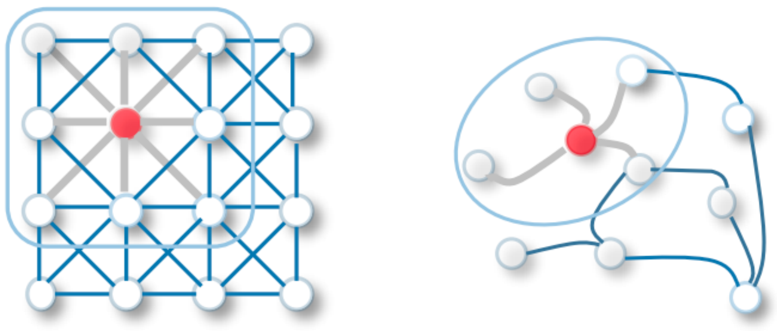

Figure 7. CNN and GCN

We will focus on Specture-based GCN here. Consider the following very simple form of a layer-wise propagation rule:

$$f(H^{(l)}, A) = \sigma(\mathrm{A}\mathrm{H}^{(l)}W^{(l)})$$

where $W^{(l)}$ is a weight matrix for the l-th neural network layer and $\sigma(\cdot)$ is a non-linear activation function like the ReLU.

5. https://arxiv.org/pdf/1609.02907.pdf

But there are two limitations of this simple model: multiplication with A means that, for every node, we sum up all the feature vectors of all neighboring nodes but not the node itself (unless there are self-loops in the graph). We can "fix" this by enforcing self-loops in the graph: we simply add the identity matrix to A.

The second major limitation is that A is typically not normalized and therefore the multiplication with A will completely change the scale of the feature vectors (we can understand that by looking at the eigenvalues of A). Normalizing A such that all rows sum to one, i.e. $\mathrm{A}D^{-1}A$, where D is the diagonal node degree matrix, gets rid of this problem. Multiplying with $\mathrm{A}D^{-1}A$ now corresponds to taking the average of neighboring node features. In practice, dynamics get more interesting when we use a symmetric normalization, i.e. $D^{-\frac{1}{2}}\mathrm{AD}^{-\frac{1}{2}}$ (as this no longer amounts to mere averaging of neighboring nodes). Combining these two tricks, we get the propagation rule:

$$f(H^{(l)}, A) = \sigma\left(\hat{D}^{-\frac{1}{2}}\hat{A}\hat{D}^{-\frac{1}{2}}H^{(l)}W^{(l)}\right)$$

with $\hat{A} = A + I$, where I is the identity matrix and $\hat{D}$ is the diagonal node degree matrix of $\hat{A}$.

3.2 GAT

A Graph Attention Network (GAT)[6] is a neural network architecture that operates on graph-structured data, leveraging masked self-attentional layers to address the shortcomings of prior methods based on graph convolutions or their approximations. By stacking layers in which nodes are able to attend over their neighborhoods' features, a GAT enables (implicitly) specifying different weights to different nodes in a neighborhood, without requiring any kind of costly matrix operation (such as inversion) or depending on knowing the graph structure upfront.

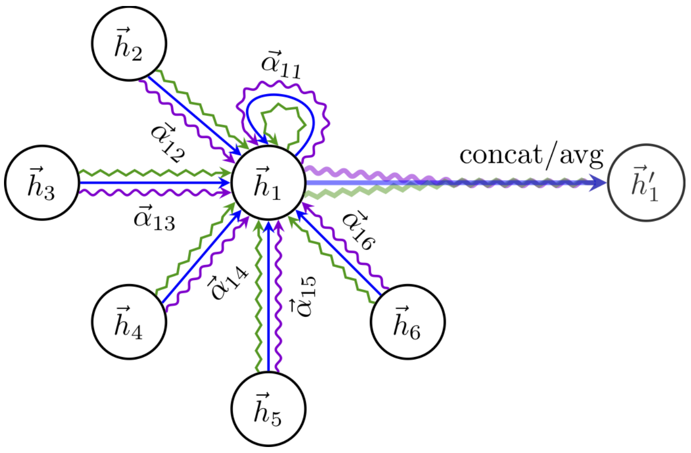

Figure 8. GAT structure

Generally, we let α_{ij} be computed as a byproduct of an *attentional mechanism*, $a: \mathbb{R}^N \times \mathbb{R}^N \to \mathbb{R}$, which computes unnormalised coefficients e_{ij} across pairs of nodes i, j, based on their features:

$$e_{\mathrm{ij}} = a\left(\overrightarrow{h_i}, \overrightarrow{h_j}\right)$$

6. https://arxiv.org/abs/1710.10903

We inject the graph structure by only allowing node i to attend over nodes in its neighbourhood, $j \in N_i$. These coefficients are then typically normalised using the softmax function, in order to be comparable across different neighbourhoods:

$$\alpha_{\text{ij}} = \frac{\exp(e_{\text{ij}})}{\sum_{k \in N_i} \exp(e_{\text{ik}})}$$

3.2.1 PyTorch Geometric

The GNN library we used in our project is PyTorch Geometric[7]. It is a geometric deep learning extension library for PyTorch.

It consists of various methods for deep learning on graphs and other irregular structures, also known as geometric deep learning, from a variety of published papers.

In addition, it consists of an easy-to-use mini-batch loader for many small and single giant graphs, a large number of common benchmark datasets (based on simple interfaces to create your own), and helpful transforms, both for learning on arbitrary graphs as well as on 3D meshes or point clouds.

Figure 9. PyTorch Geometric

Generalizing the convolution operator to irregular domains is typically expressed as a *neighborhood aggregation* or *message passing* scheme. With $x_i^{(k-1)} \in R^F$ denoting node features of node i in layer $(k-1)$ and $e_{j,i} \in R^D$ denoting (optional) edge features from node j to node i, message passing graph neural networks can be described as

$$x_i^{(k)} = \gamma^{(k)}\left(x_i^{(k-1)}, \square_{j \in N(i)} \phi^{(k)}\left(x_i^{(k-1)}, x_j^{(k-1)}, e_{j,i}\right)\right)$$

where $\square$ denotes a differentiable, permutation invariant function, *e.g.*, sum, mean or max, and γ and ϕ denote differentiable functions such as MLPs (Multi Layer Perceptrons).

PyTorch Geometric provides the MessagePassing base class, which helps in creating such kinds of message passing graph neural networks by automatically taking care of message propagation. The user only has to define the functions ϕ , i.e. message(), and γ , i.e. update(), as well as the aggregation scheme to use, i.e. `aggr="add"`, `aggr="mean"` or `aggr="max"`.

4 Our approach

5 Results

6 Analysis & future work

[1]

7. https://github.com/rusty1s/pytorch_geometric