Learning to Dispatch: A Reinforcement Learning Framework for Train Dispatch Networks

Andres Espinosa

Industrial and Systems Engineering
University of Florida
andresespinosa@ufl.edu

Abstract.

1 Introduction

The Train Dispatching Problem (TDP) concerns the real-time management of train movements across a rail network, ensuring safe and efficient use of infrastructure. This involves deciding when and where trains should move, stop, or wait, based on factors such as schedules, track availability, and priority rules. Dispatching decisions must be made continuously and quickly, especially in high-traffic networks, making the problem both operationally critical and computationally challenging.

Despite significant technological advances in other areas of transportation and logistics, train dispatching remains heavily reliant on human decision-makers or static rule-based systems. This is largely because the problem is highly combinatorial: at any given moment, there are exponentially many possible sequences of decisions that can lead to different network outcomes. Human dispatchers bring experience and intuition to these situations, but they are limited in how much information they can process and how consistently they can manage large-scale disruptions or optimize traffic flow over time.

Improving train dispatching systems has the potential to reduce passenger delays, minimize dispatching errors, and prevent network deadlocks—situations where no train can proceed without violating safety or scheduling constraints. Optimized dispatching can also improve energy efficiency and capacity utilization, making rail transport more competitive and sustainable. In dense urban transit systems or busy freight corridors, even marginal improvements in dispatching can lead to significant gains in overall network performance and reliability.

Reinforcement Learning (RL) offers a new approach to tackling the TDP by framing it as a sequential decision-making problem, where an agent learns to make dispatching decisions through interactions with a simulated environment. RL is particularly well-suited for problems with delayed consequences, dynamic environments, and large state spaces—all of which characterize train dispatching. With recent successes in games, robotics, and supply chain optimization, RL is emerging as a promising tool for learning policies that outperform hand-crafted heuristics in complex, real-world systems.

This paper focuses on the formulation of the Train Dispatching Problem for RL-based approaches, rather than on developing a fully trained solution. We emphasize how the problem can be encoded as an RL task using graph structures to represent rail networks, define meaningful states, actions, and rewards, and manage constraints inherent to railway systems. Our goal is to provide a robust and extensible framework that future researchers and practitioners can build upon when applying RL methods to train dispatching and related infrastructure scheduling problems.

To provide a thorough understanding of our approach, the remainder of this paper is organized as follows. In Section 2, we present the foundational background necessary for our formulation, beginning with a formal description of the Train Dispatch Problem and its mixed-integer programming (MIP) formulation, as well as the DISPLIB benchmark format (2.1). We then outline key reinforcement learning concepts, including Markov Decision Processes (2.2) and Deep Q-Networks, highlighting their relevance to sequential decision-making in dispatching tasks. This is followed by an introduction to Graph Neural Networks (2.3), which are critical for representing the structure of railway networks. Section **??** provides a review of related literature, including recent RL applications to combinatorial scheduling, graph-based approaches, and prior work on train dispatching. Section 3 details our proposed formulation, introducing Train Operation Graphs (3.1), Resource Conflict Graphs (3.2), and the definitions of our state and action spaces (3.3, 3.4). In Section **??**, we demonstrate preliminary results from our Deep Graph Q-Network agent (**??**), evaluating its performance on sample instances and visualizing its learned strategies (**??**). Finally, Section 4 outlines our conclusions and discusses future directions for scaling and refining the framework (4.1, 4.2).

2 Problem Background

2.1 Train Dispatch Problem

2.2 Deep Reinforcement Learning

Deep Reinforcement Learning (DRL) is a subfield of machine learning that combines reinforcement learning (RL) with deep learning techniques to solve complex decision-making problems. RL focuses on training agents to make sequential decisions by interacting with an environment, receiving feedback in the form of rewards, and learning to maximize cumulative rewards over time. Deep learning, on the other hand, enables the representation of high-dimensional data through neural networks, making it possible to handle complex state and action spaces.

The seminal work by Mnih et al. [1] introduced the Deep Q-Network (DQN), which demonstrated the ability of DRL to learn policies directly from raw pixel inputs in Atari games. This approach used a convolutional neural network to approximate the Q-value function, enabling the agent to make decisions in high-dimensional state spaces. The success of DQN highlighted the potential of combining deep learning with RL to tackle problems that were previously intractable.

Building on this foundation, Silver et al. [2] showcased the power of DRL in mastering the game of Go, a task long considered a grand challenge in artificial intelligence. Their AlphaGo system employed deep neural networks to evaluate board positions and guide search strategies, achieving superhuman performance. Subsequently, the AlphaZero algorithm [3] extended this approach to generalize across multiple games, such as chess and shogi, using a self-play mechanism and a unified reinforcement learning framework.

These advancements have established DRL as a versatile and powerful tool for solving sequential decision-making problems in diverse domains, including robotics, healthcare, and transportation systems.

Markov Decision Processes Markov Decision Processes (MDPs) provide a mathematical framework for modeling decision-making in environments where outcomes are partly random and partly under the control of a decision-maker. An MDP is defined by the tuple (S, A, P, R, γ):

- S: A finite set of states representing all possible configurations of the environment.
- A: A finite set of actions available to the agent.
- $P(s'|s, a)$: The state transition probability function, which defines the probability of transitioning to state s' given the current state s and action a.
- $R(s, a)$: The reward function, which specifies the immediate reward received after taking action a in state s.
- $\gamma \in [0, 1]$: The discount factor, which determines the importance of future rewards relative to immediate rewards.

The goal in an MDP is to find a policy $\pi(a|s)$, which is a mapping from states to actions, that maximizes the expected cumulative reward, also known as the return:

$$G_t = \mathbb{E}\left[\sum_{k=0}^{\infty} \gamma^k R(s_{t+k}, a_{t+k})\right].$$

MDPs are widely used in reinforcement learning to model environments where agents learn optimal policies through interaction and feedback.

Deep Q-Network Deep Q-Networks (DQN) are a class of reinforcement learning algorithms that approximate the Q-value function using deep neural networks. The Q-value function, $Q(s, a)$, represents the expected cumulative reward of taking action a in state s and following the optimal policy thereafter. The DQN agent learns this function through experience replay and iterative updates.

The key components of a DQN agent are:

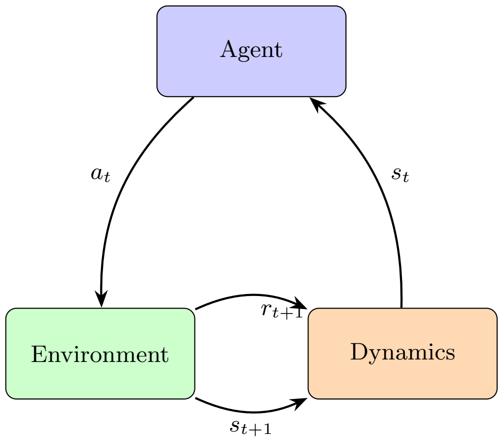


Fig. 1: The simplified markov decision process and reinforcement learning framework.

- **Q-Network:** A neural network that takes the current state s_t as input and outputs Q-values for all possible actions. The network is parameterized by weights θ, which are updated during training.
- **Target Network:** A separate network with weights θ^-, used to stabilize training by providing target Q-values. The target network is periodically updated to match the Q-network.
- **Experience Replay:** A buffer that stores past experiences (s_t, a_t, r_t, s_{t+1}). During training, mini-batches of experiences are sampled randomly from the buffer to break correlations and improve learning stability.
- **Bellman Equation:** The Q-network is trained to minimize the temporal difference (TD) error, defined as:

$$\delta = \left[r_t + \gamma \max_{a'} Q(s_{t+1}, a'; \theta^-) - Q(s_t, a_t; \theta) \right],$$

  where γ is the discount factor.
- **Exploration-Exploitation Tradeoff:** The agent balances exploration (choosing random actions) and exploitation (choosing actions with the highest Q-value) using an ϵ-greedy policy.

The training process involves iteratively updating the Q-network's weights θ using gradient descent to minimize the TD error. The learned Q-values guide the agent in selecting actions that maximize long-term rewards.

2.3 Graph Neural Networks

Graph Neural Networks (GNNs) are a class of neural networks designed to operate on graph-structured data, where entities are represented as nodes and their

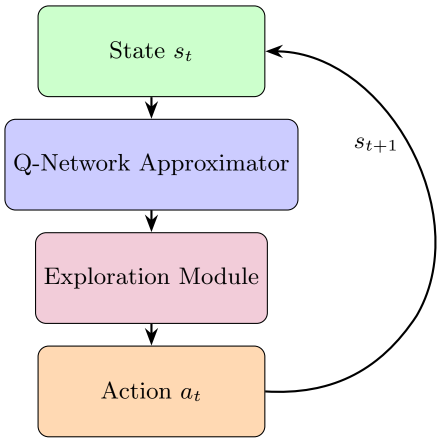


Fig. 2: Simplified Deep Q-Network action selection and feedback loop.

relationships as edges. GNNs leverage the graph topology and node features to learn meaningful representations for tasks such as node classification, link prediction, and graph classification.

Overview of GNNs GNNs extend traditional neural networks by incorporating message-passing mechanisms, where nodes aggregate information from their neighbors iteratively. This process enables the network to capture both local and global graph structures. The key components of GNNs include:

– **Message Passing:** Nodes exchange information with their neighbors through a series of message-passing steps. At each step, a node updates its representation by aggregating messages from its neighbors.
– **Aggregation Functions:** Common aggregation functions include summation, mean, and max-pooling, which ensure permutation invariance in the graph structure.
– **Update Functions:** After aggregation, node representations are updated using learnable functions, often implemented as neural networks.
– **Pooling Layers:** For graph-level tasks, pooling layers summarize the entire graph into a fixed-size representation.

Applications in Reinforcement Learning GNNs have shown significant promise in reinforcement learning (RL) for tasks involving graph-structured environments. Munikoti et al. [4] provide a comprehensive review of how GNNs enhance RL by enabling agents to reason over relational data, such as traffic networks or social graphs. They highlight the ability of GNNs to generalize across varying graph sizes and structures.

In multi-objective optimization problems, such as influence maximization in social networks, GNNs combined with meta-reinforcement learning have demonstrated effectiveness. For instance, the GraMeR framework [5] leverages GNNs to learn transferable policies across different graph instances, optimizing multiple objectives simultaneously.

Devailly et al. [6] introduce IG-RL, an inductive graph reinforcement learning approach for large-scale traffic signal control. By using GNNs, their method efficiently handles massive traffic networks, enabling scalable and adaptive control strategies.

In summary, GNNs provide a powerful framework for learning on graph-structured data, with significant implications for reinforcement learning in domains like transportation, social networks, and beyond.

3 Formulation

3.1 Train Operation Graphs

The Train Operation Directed Acyclic Graph (DAG) format provides a structured representation of train operations and their constraints. Conceptually, each train is modeled as a DAG, called the operations graph, with the following key components:

- **Nodes:** Represent operations, each with:
  - A **minimum duration** δ, specifying the shortest allowable time for the operation.
  - **Start time bounds** $[t_{\min}, t_{\max}]$, defining when the operation may begin.
  - A set of **resources** required exclusively during the operation, each with an optional **release time**.
- **Edges:** Define precedence constraints between operations, ensuring that dependent operations are executed in the correct order.
- **Entry and Exit Nodes:** The graph has a single **entry operation** (no incoming edges) and a single **exit operation** (no outgoing edges).

Resources and Constraints Resources represent entities (e.g., track sections) that cannot be shared between trains simultaneously. Operations requiring the same resource must satisfy the following:

- The **end event** of one operation must precede the **start event** of the next.
- The **release time** of the resource must be respected.

Feasibility Conditions A solution is feasible if:

1. Events are ordered chronologically.
2. Each train's operations form a valid path from the entry to the exit operation.
3. Start times respect the bounds $[t_{\min}, t_{\max}]$.
4. Operation durations meet or exceed the minimum δ.
5. Resource constraints are satisfied across all trains.

Objective Function The objective minimizes the cost of delays, defined as:

$$v_i = c \cdot \max(0, t - t_{\text{threshold}}) + d \cdot H(t - t_{\text{threshold}})$$

where c and d are constants, t is the operation's start time, and H is the Heaviside step function.

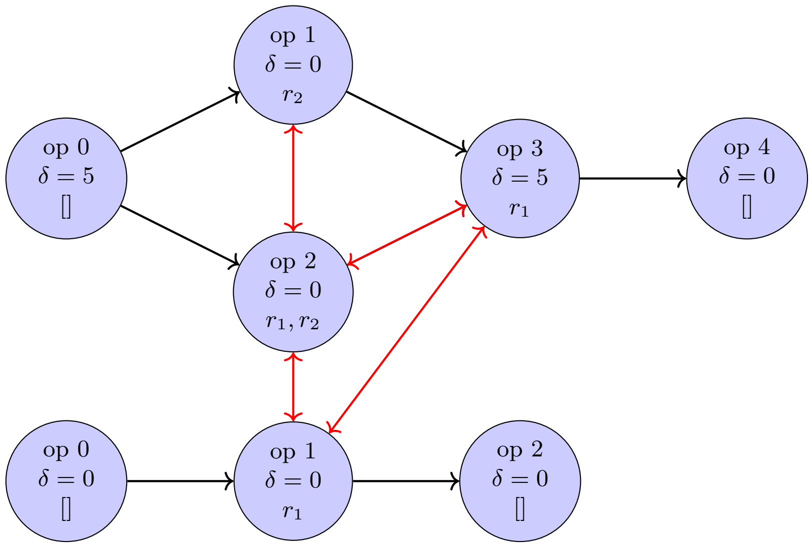


Fig. 3: Example Train Operation DAG with resource and duration constraints.

3.2 Resource Conflict Graphs

3.3 State Space

In this formulation, the state space is represented as a Directed Acyclic Graph (DAG) composed of unconnected operation graphs. Each node in the graph corresponds to an operation and is characterized by a feature vector that captures relevant attributes of the operation. The node feature vector includes the following parameters:

- x_1: Train ID of the operation.
- x_2: Operation ID of the operation.
- x_3: Start lower bound of the operation (0 if not present).
- x_4: Start upper bound of the operation (10^6 if not present).
- x_5: Minimum duration of the operation (0 if not present).
- x_6: Can pick (1 if the operation has not been picked and can be picked, 0 otherwise).

- x_7: Currently picked (1 if a train has selected this operation and no successors exist, 0 otherwise).
- x_8: Picked (1 if the operation has already been picked, 0 otherwise).
- x_9: Not picked yet (1 if the operation has not been picked but can be picked in the future, 0 otherwise).
- x_{10}: Feasible (1 if the operation can be picked, 0 otherwise).
- x_{11}: Cannot be picked (1 if the operation has not been picked and cannot be picked in the future, 0 otherwise).
- x_{12}: Threshold of the operation (0 if not present).
- x_{13}: Haversine component of the operation (0 if not present).
- x_{14}: Linear delay of the operation (0 if not present).
- x_{15}: Operation start time (0 if not started yet).
- x_{16}: Operation end time (0 if not started yet).

Edges in the graph are assigned a feature vector that captures the type of relationship and resource constraints. The edge feature vector is defined as:

- e_1: Operation type (1 if this edge is between two train operations, 0 o/w)
- e_2: Resource type (1 if this edge represents a resource share, 0 o/w)
- e_3: Release time (the release time of the incoming operation's resource, 0 if there is no release time or if it is an operation type edge)

This representation provides a simple yet comprehensive structure for encoding the operations and their relationships, which can be processed by a Graph Neural Network (GNN) to generate a graph embedding for downstream decision-making tasks.

3.4 Action Space

The action space in this formulation consists of three primary actions that the agent can take at each decision step:

- **Add Node:** The agent selects a specific operation node from the graph to be added to the current schedule. This action is only valid if the selected node satisfies all feasibility conditions, such as respecting precedence constraints and resource availability. The agent must choose one node from the set of feasible nodes.
- **Discard Node:** The agent chooses to discard a specific operation node, indicating that the operation will not be scheduled. This action is irreversible and is typically used for operations that are deemed unnecessary or infeasible to include in the schedule.
- **Advance Timestep:** The agent advances the simulation to the next timestep, allowing the environment to update the state of operations and resources. This action is used when no immediate scheduling decisions can be made or when the agent determines that advancing time is the optimal strategy.

Each action is represented as a discrete choice, and the agent's policy maps the current state of the environment to one of these actions. The design of this action space ensures that the agent has the flexibility to make scheduling decisions dynamically while adhering to the constraints of the problem.

4 Conclusion and Future Work

4.1 Future Work

This work opens several avenues for future research and development:

Improved Action Space Design The current action space, while functional, may not fully capture the complexities and nuances of the Train Dispatching Problem. Future work could explore more expressive and flexible action representations, such as hierarchical or multi-step actions, to better align with real-world dispatching scenarios. Additionally, incorporating domain-specific heuristics or constraints directly into the action space could improve the efficiency and feasibility of the learned policies.

Graph Reinforcement Learning Algorithm Development The integration of Graph Neural Networks (GNNs) with reinforcement learning (RL) has shown promise in handling graph-structured data. However, the development of specialized graph RL algorithms tailored to the unique characteristics of train dispatching remains an open challenge. Future research could focus on designing algorithms that leverage the structural properties of train operation and resource conflict graphs, enabling more efficient learning and better generalization across different network configurations.

Extensions to Other Applications While this work focuses on train dispatching, the proposed framework has the potential to be adapted to other domains involving complex scheduling and resource allocation problems. Examples include air traffic management, urban transit systems, and supply chain logistics. Extending the framework to these applications would require domain-specific modifications to the state and action spaces, as well as the underlying graph representations, but could significantly broaden the impact and utility of the approach.

4.2 Conclusion

This honors thesis introduces a proof-of-concept reinforcement learning framework for the Train Dispatching Problem, emphasizing the formulation and potential of graph-based representations and deep learning techniques. By framing the problem as a Markov Decision Process and leveraging Graph Neural Networks, this work provides a foundational approach to train dispatching that is both scalable and extensible.

As a formulation-focused study, this thesis aims to inspire future research in applying reinforcement learning to complex scheduling and resource allocation problems. Future efforts should prioritize refining the action space, developing tailored graph reinforcement learning algorithms, and exploring applications in other domains. These advancements will build upon the groundwork laid here, contributing to the broader understanding and development of intelligent transportation systems.

5 Appendix

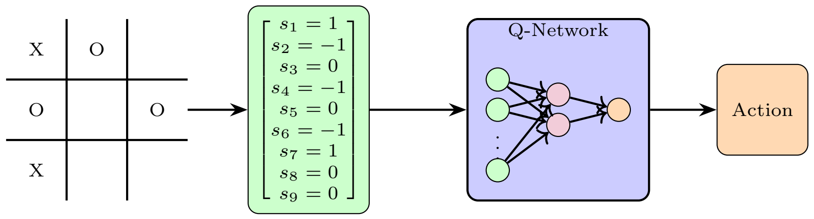
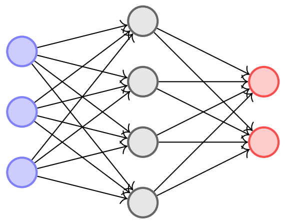


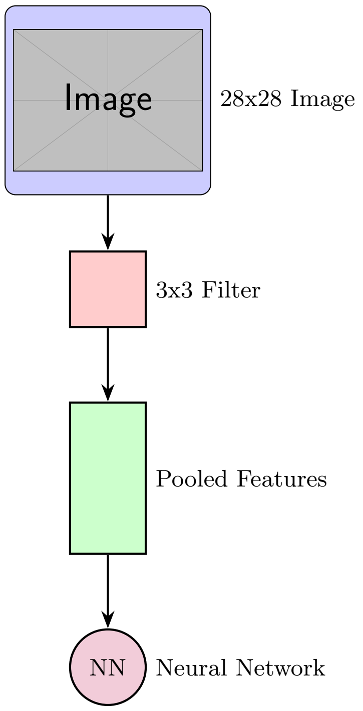
Fig. 4: Illustration of a Q-Network processing a Tic-Tac-Toe board state.

References

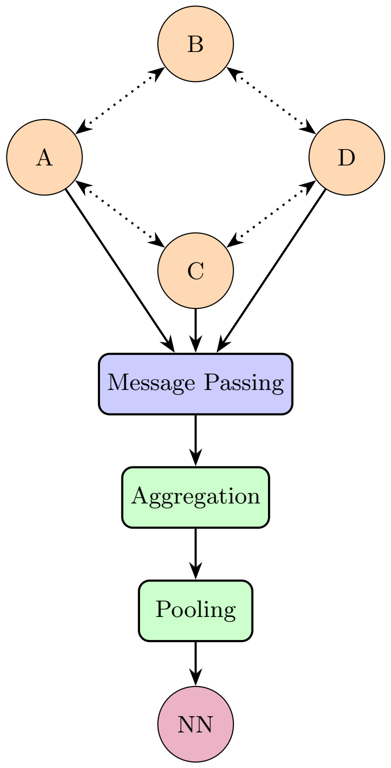
1. V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. A. Riedmiller, "Playing atari with deep reinforcement learning," *CoRR*, vol. abs/1312.5602, 2013. [Online]. Available: http://arxiv.org/abs/1312.5602
2. D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, Y. Chen, T. Lillicrap, F. Hui, L. Sifre, G. van den Driessche, T. Graepel, and D. Hassabis, "Mastering the game of go without human knowledge," *Nature*, vol. 550, pp. 354–, oct 2017. [Online]. Available: http://dx.doi.org/10.1038/nature24270
3. D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T. P. Lillicrap, K. Simonyan, and D. Hassabis, "Mastering chess and shogi by self-play with a general reinforcement learning algorithm," *CoRR*, vol. abs/1712.01815, 2017. [Online]. Available: http://arxiv.org/abs/1712.01815
4. S. Munikoti, D. Agarwal, L. Das, M. Halappanavar, and B. Natarajan, "Challenges and opportunities in deep reinforcement learning with graph neural networks: A comprehensive review of algorithms and applications," 2022. [Online]. Available: https://arxiv.org/abs/2206.07922
5. S. Munikoti, B. Natarajan, and M. Halappanavar, "Gramer: Graph meta reinforcement learning for multi-objective influence maximization," 2022. [Online]. Available: https://arxiv.org/abs/2205.14834
6. F.-X. Devailly, D. Larocque, and L. Charlin, "Ig-rl: Inductive graph reinforcement learning for massive-scale traffic signal control," *IEEE Transactions on Intelligent Transportation Systems*, vol. 23, no. 7, p. 7496–7507, Jul. 2022. [Online]. Available: http://dx.doi.org/10.1109/TITS.2021.3070835



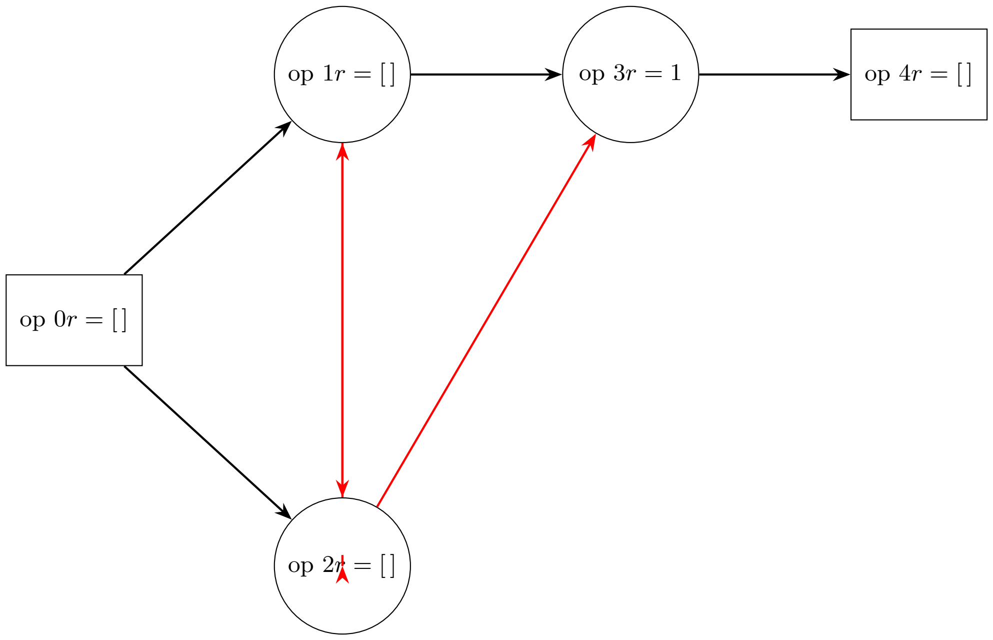
(a) Fully Connected ANN



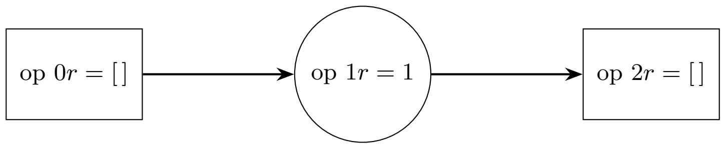
(b) Convolutional Neural Network (CNN)



Train A



Train B


op $0r = [\,]$
op $1r = 1$
op $2r = [\,]$