Mukhutdinov, Dmitry; Filchenkov, Andrey; Shalyto, Anatoly; Vyatkin, Valeriy

Multi-agent deep learning for simultaneous optimization for time and energy in distributed routing system

Published in:
Future Generation Computer Systems

DOI:
10.1016/j.future.2018.12.037

Published: 01/05/2019

Document Version
Peer-reviewed accepted author manuscript, also known as Final accepted manuscript or Post-print

Multi-Agent Deep Learning for Simultaneous Optimization for Time and Energy in Distributed Routing System

Dmitry Mukhutdinov[a,*], Andrey Filchenkov[a], Anatoly Shalyto[a], Valeriy Vyatkin[a,b,c]

[a] *Computer Technologies Laboratory, ITMO University, Saint Petersburg, Russia*
[b] *Department of Electrical Engineering and Automation, Aalto University, Finland*
[c] *Department of Computer Science, Computer and Space Engineering, LuleåTekniska Universitet, Sweden*

Abstract

Packet routing problem most commonly emerges in the context of computer networks, thus the majority of routing algorithms existing nowadays is designed specifically for routing in computer networks. However, in the logistics domain, many problems can be formulated in terms of packet routing, e.g. in automated traffic routing or material handling systems. In this paper, we propose an algorithm for packet routing in such heterogeneous environments. Our approach is based on deep reinforcement learning networks combined with link-state protocol and preliminary supervised learning. Similarly to most routing algorithms, the proposed algorithm is a distributed one and is designed to run on a network constructed from interconnected routers. Unlike most other algorithms, proposed one views routers as learning agents, representing the routing problem as a multi-agent reinforcement learning problem. Modeling each router as a deep neural network allows each router to account for heterogeneous data about its environment, allowing for optimization of more complex cost functions, like in case of simultaneous optimization of bag delivery time and energy consumption in a baggage handling system. We tested the algorithm using manually constructed simulation models of computer network and baggage handling system. It outperforms state-of-the-art routing algorithms.

Keywords: Distributed Systems, Multi-Agent Learning, Deep Reinforcement Learning

*Corresponding author
Email addresses: flyingleafe@gmail.com (Dmitry Mukhutdinov), afilchenkov@niuitmo.ru (Andrey Filchenkov), shalyto@mail.ifmo.ru (Anatoly Shalyto), vyatkin@ieee.org (Valeriy Vyatkin)

1. Introduction

Distributed routing problem first emerged in computer science in the context of packet routing in computer networks. Over the years, many network routing algorithms such as *Open Shortest Path First* (OSPF) [1] has been developed and standardized, and now they are widely used in computer networks.

However, the distributed routing problem emerges in contexts different from computer networks. An example of such context is *baggage handling systems* (BHS) primarily used in airports (Figure 1) or material handling systems used in manufacturing enterprises and logistics centers. In [2] it was shown that baggage handling can be approached in a distributed way, by using a simple *distance-vector* routing protocol, which was originally developed for packet routing in computer networks [3].

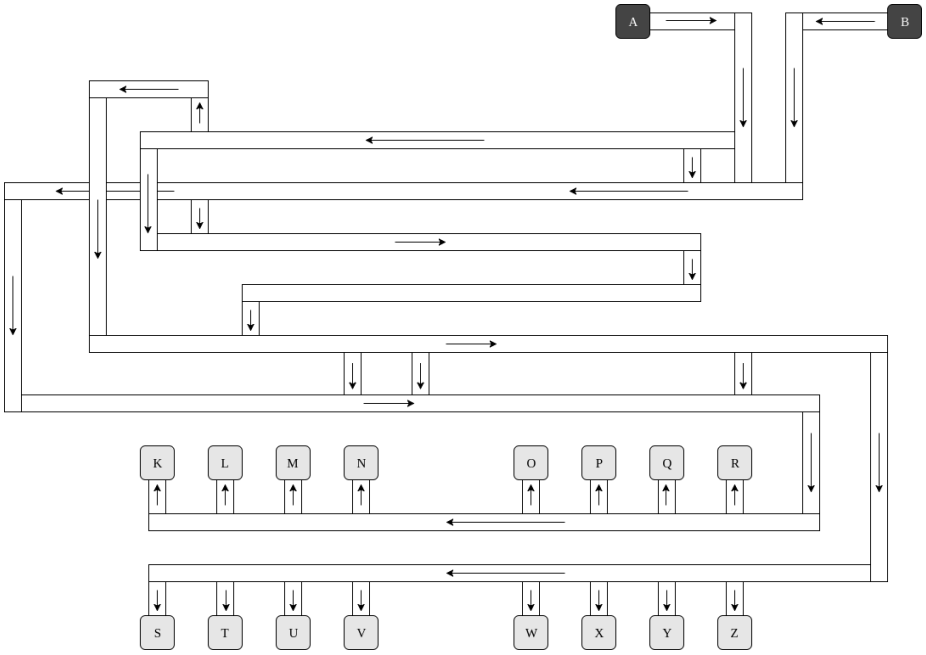


Figure 1: An example scheme of baggage handling system.

Though it is possible to use algorithms for network routing in other contexts, this is not always an optimal solution. Some constraints may exist, which are significantly different from constraints presented in the network routing problem. For example, in the case of BHS, we may aim to minimize energy consumption of conveyors together as well as the speed of baggage delivery. Algorithms designed for network routing do not fit for solving problems with such constraints.

In this paper, we propose a routing algorithm based on *reinforcement learning* approach. The idea of viewing packet routing as a reinforcement learning problem was first implemented in [4] using *Q-routing* algorithm. The algorithm

we propose in this paper is largely based on Q-routing, but has the following differences:

- Value function is approximated via *neural network* (NN) instead of lookup table.
- Learning agents use additional information, such as information about current graph topology in order to make more precise estimations of action values. This information is passed between agents via complementary protocols, such as the link-state protocol.
- Preliminary application of supervised learning on examples of baseline agent behavior is used to avoid divergence.

Using neural networks for value function approximation allows learning agents to consider arbitrary information, which may be related to routing efficiency, thus allowing the algorithm to be applied efficiently for routing in heterogeneous environments (such as baggage handling systems).

The rest of this paper is structured as follows. In Section 2, we specify the packet routing problem in a generalized way, review existing routing algorithms and reinforcement learning techniques and formulate the routing problem in terms of reinforcement learning. Then in Section 3, we describe the algorithm, as well as several considered NN architectures. In Section 4, we provide results of an experimental comparison of the proposed algorithm with link-state based shortest path algorithm (which is basically a simplified version of OSPF, one of the most widely used network routing protocol nowadays) and Q-routing (which is the basis of the proposed algorithm). A comparison is conducted in two different environments: the simulation model of a computer network and the simulation model of BHS. Section 5 is devoted to discussion of the results and drawing conclusions.

2. Background and problem statement

2.1. Formulation of generalized routing problem

In this section, we formally describe packet routing problem in a generalized way, so that resulting definition can be used to describe the problem in a variety of settings.

We model the *network* as a directed graph $G = (V, E)$, where each vertex $v \in V$ corresponds to a network node (e. g. router or switch), and every edge $e \in E \subset V^2$ corresponds to a link between nodes. *Packets* are sent between the nodes in the network: a packet may be sent from one of the *source nodes* $V_s \subseteq V$ and be directed towards one of the *destination nodes* $V_d \subseteq V$.

When a packet p heading towards destination d arrives at node u it proceeds as follows:

- If $u = d$, then the packet has reached its destination and it leaves the network.

- Otherwise, a packet proceeds along one of the edges which begin in u. This edge $e = (u, v)$ is chosen accordingly to some *routing policy* $\text{Route}(p, u, d)$.

Let each edge e has an associated cost $\text{Cost}(e)$. Then the cost of the packet path is defined as sum of costs of the edges included into this path:

$$\text{Cost}(path(p)) = \sum_{e \in path(p)} \text{Cost}(e).$$

In case of the most common instance of packet routing problem, namely network routing, $\text{Cost}(e)$ is usually defined as *link latency*, i. e. the time, which a packet takes to travel along the link e. The cost of a packet path is then total packet's travel time. Note that here we deliberately do not separately define costs for traveling through nodes, despite the fact it makes sense (i. e. in network routing it makes sense to also consider the time a packet spends in router's processing queue). We do so for the sake of simplicity: we assume that the actual cost for traveling through node v is included in costs of all edges ending with v.

The packet routing problem is then a problem of choosing a *routing policy* $\text{Route}(p, u, d)$, which minimizes expected cost of path from u to d for packet p. If costs of all edges in the network are known and constant, then the packet routing problem reduces to a problem of finding the shortest path in a directed graph. However, for real-world problems, this is rarely true. Even in a relatively simple case of computer networks link latencies change over time because of network congestion, link breaks and other reasons. In case of more complex environments costs of edges may change because of a large variety of reasons.

In order to account for this in a most general way possible, let us introduce a notion of *network state* $\mathcal{S}$. $\mathcal{S}$ is a value representing all information about the current state of the network, including constant parameters (e. g. latencies and bandwidths of links) as well as dynamically changing information (e. g. current positions of packets in the network). To account for the current network state in the edge cost function, we may simply make the cost function accept $\mathcal{S}$ as an input parameter: $\text{Cost}(e, \mathcal{S})$. However, in practice, only the small part of the network state influences the cost of particular edge e, e. g. for network routing, it is the properties of link $e = (u, v)$ and router v together with the number and properties of packets currently traveling through link e and router v. Let us denote this small part of state related to edge e as $\mathcal{S}_e$. The cost function then takes the form $\text{Cost}(e, \mathcal{S}_e)$.

If costs of edges depend on network state, then the routing policy $\text{Route}(p, u, d)$ should depend on it too. Ideally, router u should take the whole network state into account in order to precisely determine the optimal routing decision for the current packet. However, we consider a *distributed* routing problem, which means that a single router v cannot possibly be aware of the whole network state. Instead, it has access to some part of the state, which we will call *router observation* and denote as $o(v, \mathcal{S})$. The routing policy then takes the form $\text{Route}(p, v, d, o(v, \mathcal{S}))$.

2.2. Routing as a reinforcement learning problem

Note that routing policies of nodes in network directly influence on how the network state changes over time, because they determine how the positions of packets change. This makes the problem of finding an optimal routing policy even harder, especially in the distributed setting, where a single router cannot observe the whole network state. However, such a problem can be conveniently reformulated in terms of *multi-agent reinforcement learning* with *partially observable Markov decision processes* (POMDPs) [**?**] as follows:

- The *state* of the environment is the *network state* S as defined in Section 2.1.
- Each router is viewed as an independent *agent* which observes only some part of the state. The agent's *observation* is a tuple $(p, v, d, o(v, \mathcal{S}))$, where p is the packet being currently processed, v is the current router, d is the packet's destination and $o(v, \mathcal{S})$ is the *router observation* as defined in Section 2.1.
- The set of possible *actions* is the set of available outgoing edges. When the particular *action* is chosen, the packet is sent over the corresponding edge.
- A *reward* for action is negated *cost* of the edge over which the packet has been sent: $r = -\text{Cost}(e, \mathcal{S}_e)$, where cost of the edge is defined as in Section 2.1.

Such a problem can be solved, for example, via the method of *independent Q-learning* [5]. In the following sections, we will show how we apply ideas of independent Q-learning to packet routing problem.

2.3. Overview of existing routing algorithms

Most of widely used routing protocols belong to one of the two families: *distance-vector* [3] and *link-state* [6]. For example, the most popular network routing protocol nowadays, namely OSPF [1], is a link-state protocol.

Distance-vector protocols are based on Bellman-Ford shortest path algorithm. Every router stores the vector containing costs of shortest paths to every other nodes in a network together with a neighbor from which shortest path starts. Routers periodically broadcast their vectors to their neighbors. After receiving a vector from a neighbor, the router updates its own shortest path estimates considering adjacent link costs. Different distance-vector protocols differ in how exactly routers estimate link costs and exchange their distance vectors.

Link-state protocols are based on Dijkstra's algorithm. Every router stores a graph which models the network and uses Dijkstra's algorithm to determine the shortest paths to every other node. If a link adjacent to some router breaks, the router broadcasts this info to the network and other routers update their network graphs accordingly.

Protocols from both families are well fit for routing network packets and computationally efficient. However, they are still not very good at adapting to substantial changes in the environment, e.g. significant changes in traffic patterns and load [7].

There were attempts to implement an efficient and adaptive routing protocol using ideas of reinforcement learning. The *Q-routing* [4] algorithm, which is based on reinforcement learning, showed good results in conditions of varying traffic patterns, being able to change routing policy on fly. The algorithm itself is remarkably simple, here is its outline:

- Every router x stores two-dimensional table $Q_x(d, y)$, which contains the estimations of minimal time to reach destination node d if going through neighbor y.
- Packet p with destination d is forwarded to neighbor $y = \text{argmin}_{(x,y)\in E} Q_x(d, y)$. Time of packet sending is assigned to $t_{p,s}$.
- When y receives packet p at time $t_{p,r}$, it computes its own estimate for the rest of packet travel time: $t_p = \min_{(y,z)\in E} Q_y(d, z)$, and sends $t_{p,r}$ and t_p back to x.
- After receiving $t_{p,s}$, $t_{p,r}$ and t_p, x updates its own minimal time estimate as follows: $Q_x(d, y) = Q_x(d, y) + \alpha((t_r - t_s) + t - Q_x(d, y))$, where α is *learning rate*, which is a hyperparameter.

There also exist modifications of this algorithm, such as *predictive Q-routing* [8] and *dual Q-routing* [9]. Unfortunately, this algorithm, as well as its modifications, has a significant disadvantage when being applied to computer networks – a router sends a protocol message over network per every received packet, which leads to significant performance overhead. However, in other contexts, protocol messages often do not share the communication channel with "packets", because a "packet" might be a physical object, such as a bag on a conveyor, while service message is a byte sequence sent over a wire. Therefore, in such contexts, sending a service message is negligibly cheap, which allows applying reinforcement learning approaches including *Q-routing*.

In recent years, the reinforcement learning approach has been repeatedly applied to routing problems in complex environments, such as mobile ad hoc networks (MANETs) [10, 11, 12], congestion problems [13, 14] and vehicle routing [15], and software-defined networks [16, 17]. Most of those results feature multi-agent reinforcement learning and some also feature deep reinforcement learning. Those results demonstrate that multi-agent reinforcement learning is a promising approach to a wide variety of routing problems. However, to the best of authors' knowledge, there were no published works on applying multi-agent deep reinforcement learning approach to routing in conveyor networks, such as baggage handling systems, which are the main focus of this paper.

The idea of using neural networks for solving routing problems is not new either. For example, in [18, 19] it was suggested to use hardware implementations

of Hopfield neural networks in routers in order to solve the shortest path problem very quickly. However, this objective is completely different from the one we aim to solve in this paper, thus the way we use neural networks is completely different.

Another original approach is AntNet algorithm [7]. The idea of the algorithm is to use special "agent" packets, which traverse the whole network and collect information about its state and sharing this information with routers they pass through. The algorithm has shown good experimental results but was not widely adopted due to already widespread adoption of distance-vector and link-state protocols.

3. Proposed approach

3.1. Basic algorithm

The method we propose, which we call *DQN-routing*, is largely based on *Q-routing*, which was described in Section 2.3. The method is as follows:

- Every router processes one packet at a time. Processed packed is referred to as *current packet*. A packet p consists of its destination d and its state s_p (s_p may be empty).
- Every router v has a *current state* $s_v = (v, d, s_p, u_1, .., u_m, X_1, .., X_k)$, where d is the destination node of current packet, s_p is the *state* of current packet, $u_1, .., u_m$ are neighbors of v and $X_1, .., X_k$ are pieces of *subsidiary data*, provided by *subsidiary protocols* (k may be zero).
- Every router contains a neural network which approximates value function $Q_v(s_v, u)$, where $-Q_v(s_v, u)$ is the estimation of minimal cost of path from v to destination of current node d through neighbor u.
- Current packet is forwarded to neighbor u, which is chosen randomly with a probability distribution $softmax(\{Q_v(s_v, u)|(v, u) \in E\})$. State of a packet might be changed before sending.
- When u receives packet p from v, it computes a reward $r_{p,v,u}$ as negated cost of packet travel ($r_{p,v,u} = -cost(p, (v, u))$) and estimated value of following actions $q_{p,u} = \max_{(u,w)\in E} Q_u(s_u, w)$, and sends both values back to v.
- After receiving $r_{p,v,u}$ and $q_{p,u}$, v fits its neural network on $(s_v, u, r_{p,v,u} + q_{p,u})$ sample.

Our method differs from *Q-routing* in the following ways:

- Q-function (estimates on full path costs) is approximated via the neural network instead of being stored in a table.

- Usage of a neural network allows Q-function having arbitrary, possibly infinite domain. We use this advantage by gathering *subsidiary data* via *subsidiary protocols* and considering this data as an input to a neural network approximating Q-function in addition to destination ID d and neighbor ID y. An example of a subsidiary protocol is a link-state protocol, which we use to obtain information on current network topology.

- We use the abstract notion of *path cost* instead of strictly optimizing packet travel time, because we aim to be able to optimize arbitrary path cost functions.

- We use the *softmax strategy* for choosing the neighbor in order to obtain a better balance between exploration and exploitation.

We call this approach a *method* rather than an *algorithm*, because different algorithms with different properties can be obtained by adding various subsidiary protocols and choosing various neural network architectures. In the rest of this paper, however, we will consider a particular set of neural network architectures and a particular set of subsidiary protocols. Most importantly, we will use a link-state protocol and data about network topology in every variant of method implementation.

3.2. Neural network architecture

The input of a neural network is the current state of a router $s_n = (n, d, s_p, y_1, .., y_m, X_1, .., X_k)$, where n is the current node in network, d is the destination node of the current packet, s_p is the state of current packet, $y_1, .., y_m$ are available neighbors and $X_1, .., X_k$ are pieces of subsidiary data. Having current node n as a part of the neural network's input allow us to train a single model, which is able to work as any node in given network.

Numbers n, d and set of numbers $y_1, .., y_m$ are passed in *one-hot encoding*. E.g. if we have a network with seven nodes, number 3 is encoded as 0010000 and set $\{4, 6\}$ is encoded as 0001010. One-hot encoding is used in order to avoid depending on the order in which nodes are enumerated.

We consider three different architectures of neural networks. Every NN architecture has a corresponding type of packet state s_p.

First one is a basic feed-forward neural network with two fully-connected hidden layers, which use hyperbolic tangent ($tanh$) as an activation function (Figure 2). This type of architecture uses packets with an empty state.

Other two architectures are recurrent ones, both have LSTM layers [20]. These architectures differ in a way the hidden state of LSTM layer is passed between actions. In the first case (Figure 3), which we denote as "network with router memory", network works as the classic RNN: LSTM hidden state (C_{t-1}, h_{t-1}) after $(t-1)$-th iteration is passed to the same layer of the network on t-th iteration, and the packet state is empty.

In the second case (Figure 4), which we denote as "network with packet memory", the packet state contains last LSTM hidden state of a previously

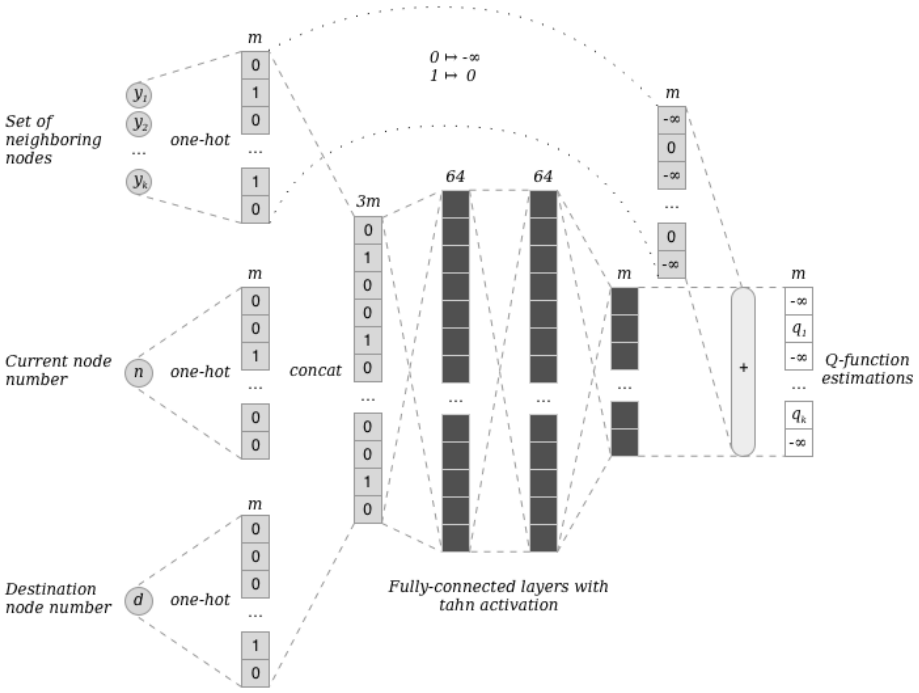


Figure 2: Feed-forward NN architecture. m denotes the number of nodes in graph

encountered router, which is injected into the LSTM layer of the current router during packet processing. Every packet which has just entered the network contains hidden LSTM state initialized to zero.

The intuition behind two types of recurrent architectures is as follows. In case with "router memory", *router* maintains some kind of "memory" of previously routed packets (encoded in LSTM hidden state), while in the case with "packet memory" *packet* has the memory of previously visited nodes.

In every case, the output layer of the neural network contains n neurons with linear activation functions, where i-th neuron corresponds to i-th node in the network. If i-th is not a neighbor of the current node, then $-\infty$ is added to an output of i-th neuron in the output layer (In practice, $-\infty$ is some big negative number, such as -1,000,000). So the output values of NN yield estimations of Q-function $Q(s, a)$, where action determines the neighbor to which we redirect a packet, and for every node b which is not a neighbor $Q(s, b) = -\infty$.

In every considered test scenario and environment we use a *link-state* protocol as a subsidiary protocol to provide information about graph topology. This information is added to the inputs of neural network as flattened upper triangle of *adjacency matrix* A_g (Figure 5).

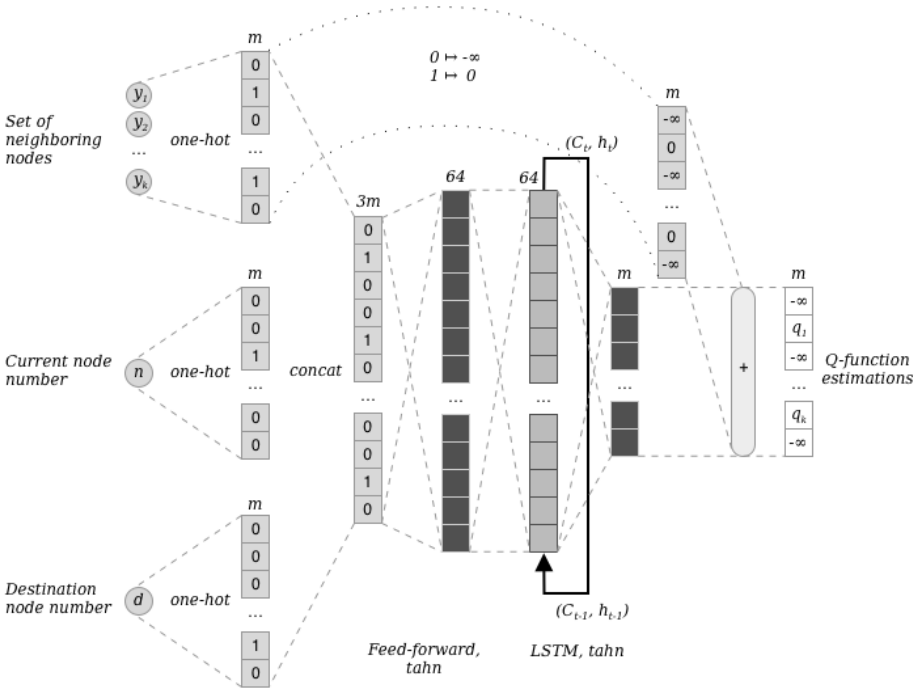


Figure 3: RNN with "router memory".

3.3. Preliminary supervised learning

crossed out *Q-learning* algorithms applied to finding an optimal strategy in reinforcement learning problem successfully converge in case of finite state space [21], but that cannot be generalized, which makes Q-learning unstable when training neural networks. Authors of the DQN algorithm [22] counter this using *experience replay*. This technique is to store previous states with corresponding rewards and actions and to fit on a random sample of episodes from this memory buffer on each learning step. Unfortunately, this approach does not work well in non-stationary environments, as old episodes in experience replay buffer stop representing the actual behavior of the environment over time [23].

Multi-agent environments are non-stationary because other agents learn over time, changing their behavior, rendering experience replay ineffective. Figure 6 demonstrates how the DQN algorithm with randomly initialized neural networks perform in a model of a computer network in the test scenario with static traffic pattern and low traffic intensity. The algorithm is compared with the link-state shortest path algorithm, which provides an optimal strategy for this scenario. It can be seen that the DQN algorithm takes a lot of time to converge, and still converges to a suboptimal strategy.

To counter this problem, we use *preliminary supervised learning* on dataset of tuples $(o, \{Q'(o,x)\}_{x \in V})$, where o is a router observation containing start

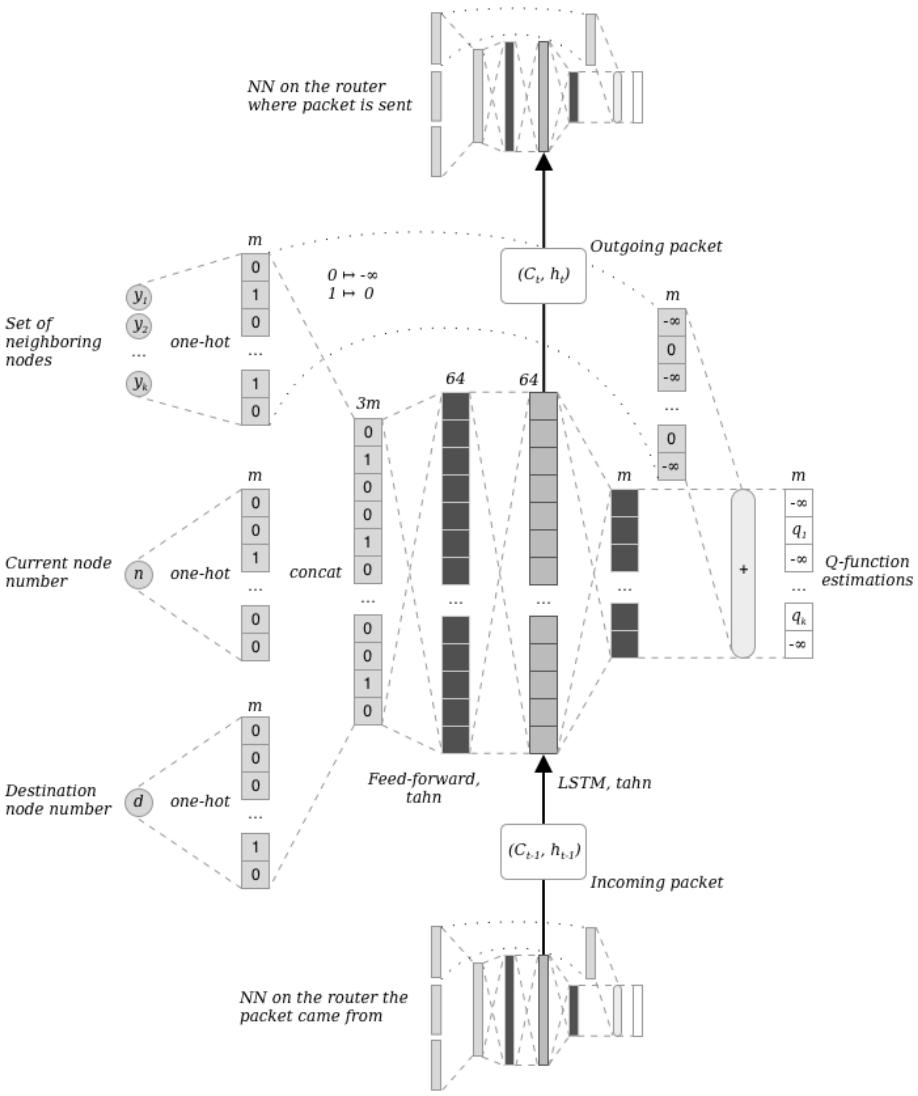


Figure 4: RNN with "packet memory".

node n and destination d, as described in Section 3.2, and $Q'(o, x)$ is the length of the shortest path between n and d, which contains neighbor x ($Q'(o, x) = -\infty$ if x is not a neighbor of n). The lenghts of edges are considered static while generating pre-training dataset, each edge has a length which is a *lower bound* on cost for passing this edge (e. g. *link latency* in case of computer network routing).

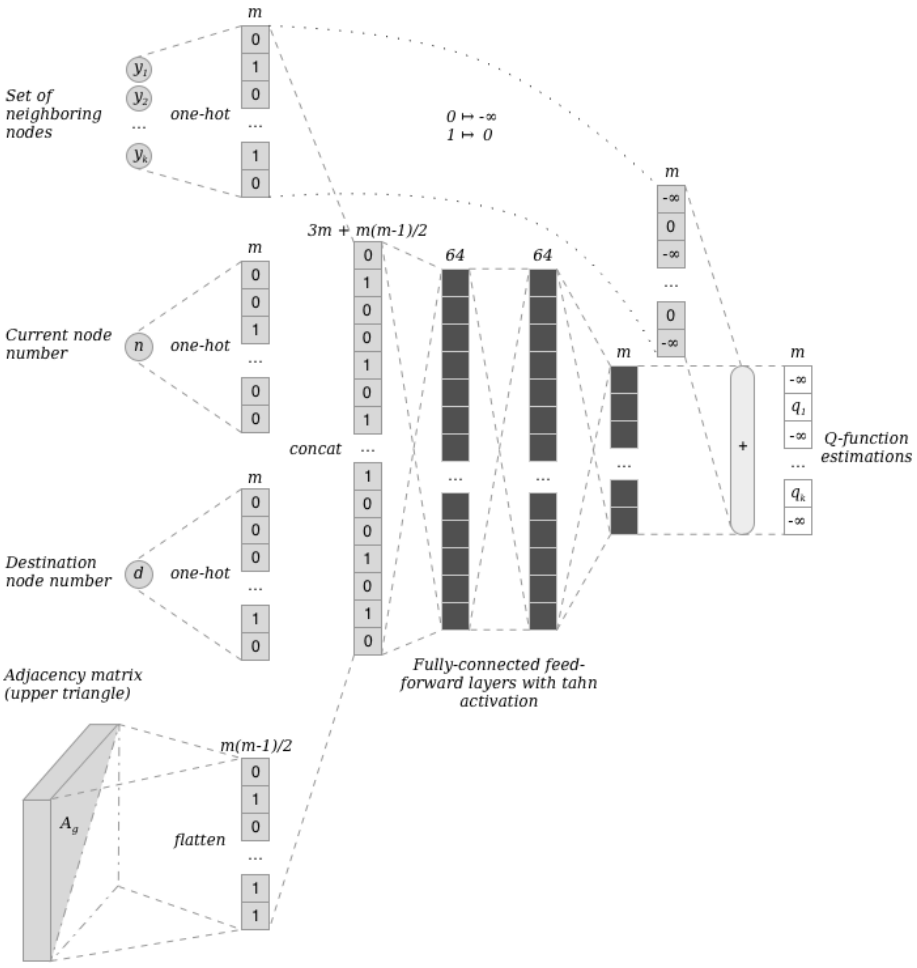


Figure 5: Feed-forward NN architecture with subsidiary data (graph adjacency matrix) passed as a part of input.

4. Experiments and results

We performed experiments in two simulation models: a model of a computer network and a model of a baggage handling system. In both environments, we compared DQN-routing with the shortest path (Dijkstra) algorithm with a link-state protocol and Q-routing algorithm. The link-state shortest path algorithm was chosen because link-state protocols are dominant in computer network routing nowadays, and Q-routing was chosen because DQN-routing is based on it.

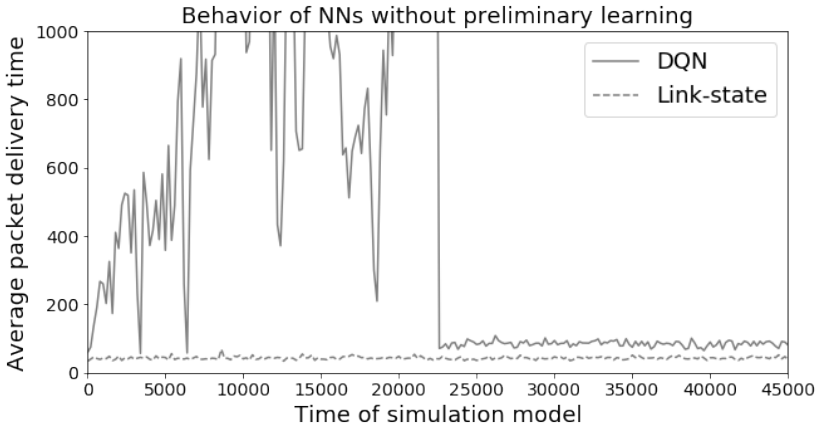


Figure 6: Divergence of DQN routing without preliminary supervised learning.

4.1. Experiments in simulation model of computer network

In the simulation model of a computer network, the optimized cost function is *packet travel time* and reward for single action is time between the moment of packet departure and the moment when the packet is processed on neighbor node. In our simulation model, time is measured in abstract units, but in the context of computer network, we will assume that these units represent *milliseconds* for convenience.

For our set of experiments, we used a model of a network of ten nodes (Figure 7). Every router in the network take 5 ms for processing one packet, every link in the network has a latency of 10 ms and bandwidth of 1024 bytes/ms.

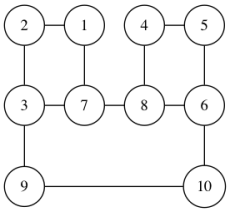


Figure 7: Network graph for experiments in simulation model of computer network.

The only subsidiary protocol used by DQN-routing in these scenarios was the link-state protocol, which provided information about network topology. This information was added to the input of each NN as an adjacency matrix, in the manner described in Section 3.2

Performance of algorithms was calculated in the following way:

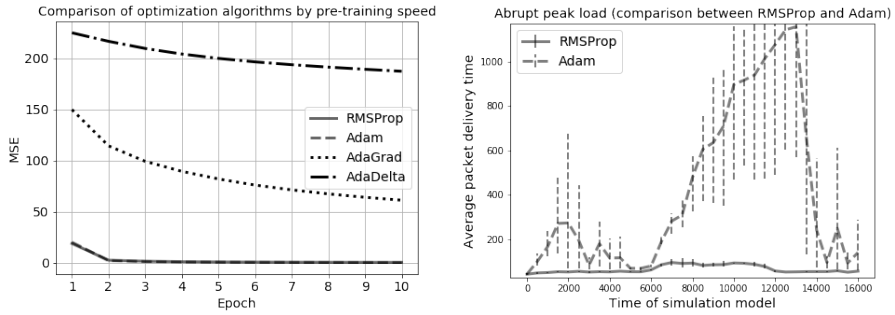
- Packet travel times were collected via evaluation of test scenario. The timeline was split into intervals of 500 time units, and for every interval average value of packet travel time was calculated.

- Test scenario was evaluated three times with different random seeds
- Results from three runs were averaged to obtain final results.

4.1.1. Preliminary application of supervised learning

A neural network was pre-trained on approximately 230,000 actions performed in the graph shown on Figure 7 by the shortest path algorithm. Target values of $Q(o, a)$ were calculated as lengths of the shortest path to packet destination through neighbor a, where link *latencies* were treated as lengths of edges.

During supervised learning, we performed an initial comparison of optimization algorithms for neural networks. We considered four popular gradient optimization algorithms with adaptive learning rate: RMSProp [24], AdaDelta [25], AdaGrad [26] and Adam [27]. We trained a feed-forward NN on the generated training set for multiple epochs using different optimization algorithms and measured *MSE* (mean squared error) on training set after each epoch.



(a) Preliminary supervised learning comparison

(b) Comparison of Adam and RMSProp in a peak load scenario

Figure 8: Comparison of optimization algorithms during preliminary supervised learning and reinforcement learning

Figure 8a shows that AdaGrad and AdaDelta fail to generalize over the training set, while Adam and RMSProp perform well and almost identically.

4.1.2. Changing the traffic load

In this scenario, we imitated suddenly increased traffic load: the period between packet sendings starts with 10 ms, then changes to 3.5 ms, and then changes to 10 ms again. Packets were sent between two parts of the network graph: the first part contains nodes 1, 2, 3 and 7, and the second part contains nodes 4, 5, 6 and 8.

In Section 4.1.1 we determined that Adam and RMSProp performs almost equally well during supervised learning. So at first, we compared these two optimization algorithms in this test scenario. It turned out that Adam performs poorly during reinforcement learning, and agents using Adam cannot adapt to a

changed environment (Figure 8b). On the other hand, RMSProp performs well, thus we chose RMSProp as an optimizer in the final version of the algorithm.

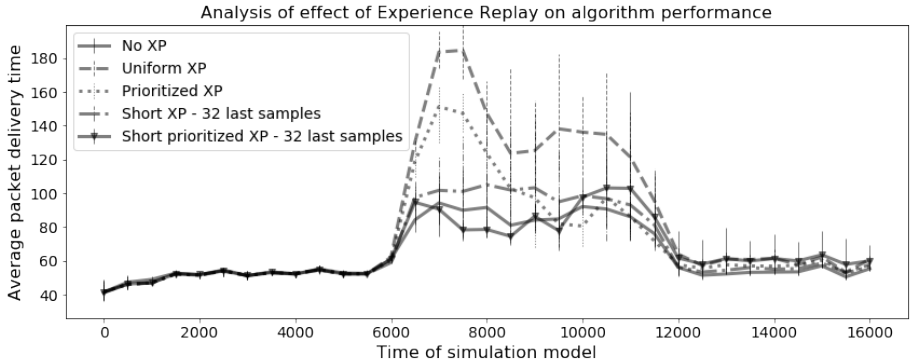


Figure 9: Comparison of different types of experience replay.

Additionally, we compared the performance of feed-forward NNs with different types of experience replay (including no experience replay at all). Figure 9 shows that the introduction of any sort of experience replay impedes agent's ability to adapt to changes in the environment. Therefore we did not use experience replay in the final version of the algorithm.

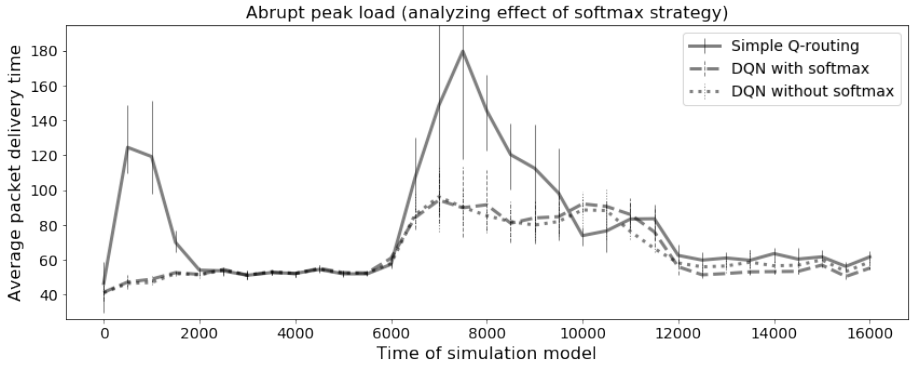


Figure 10: Comparison of agents using naive and softmax strategy.

Furthermore, in this test scenario, we analyzed the effect of using softmax strategy for action selection by comparing the performance of DQN-routing with and without using it. Figure 10 shows that agents struggle to restore optimal strategy after traffic load became low again if they do not use softmax strategy. Therefore, in the final version of the algorithm, we will use the softmax strategy to achieve a balance between exploration and exploitation.

After that, we compared the performance of three considered types of NN architecture. On Figure 11, we can see that in this scenario, DQN-routing with

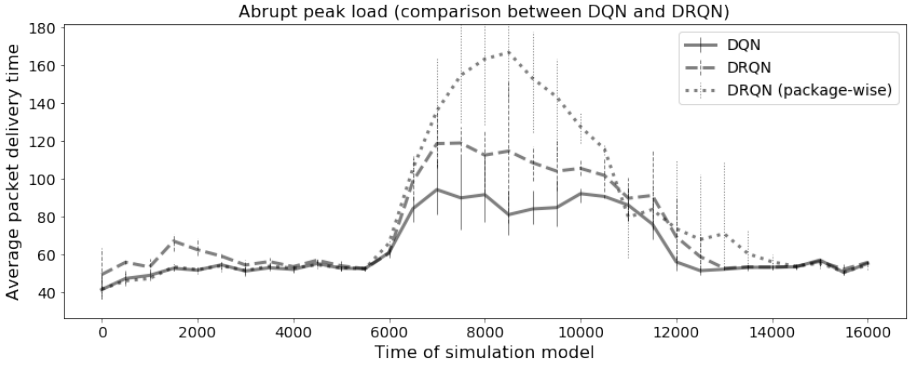


Figure 11: Comparison of different NN architectures in peak load scenario.

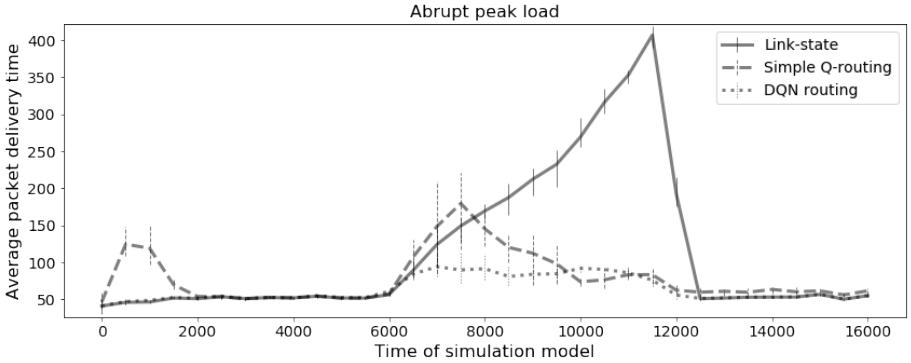


Figure 12: Comparison of DQN-routing with baseline algorithms in peak load scenario.

recurrent NNs perform worse than DQN-routing with feed-forward NNs, so we will again compare only the latter with the baseline algorithms.

As it can be seen on Figure 12, the simple shortest-path algorithm with the link-state protocol is unable to adapt to increased traffic load by redirecting part of the traffic through nodes 9 and 10, instead of overloading nodes 7 and 8. The Q-routing algorithm is able to do this, but it cannot restore the optimal strategy in low-load conditions after the peak is passed. But DQN-routing shows both the ability to quickly adapt to new conditions and restore optimal strategy when conditions change to initial ones again.

4.1.3. Changing the network topology

In this scenario, we imitated successive breakages of links $(7, 8)$, $(1, 2)$ and $(5, 6)$ followed by their restoring in the same order. The traffic load was moderate. As well as in the previous scenario, packets were sent between two parts of the network graph.

Figure 13 shows how three considered neural network architectures compare

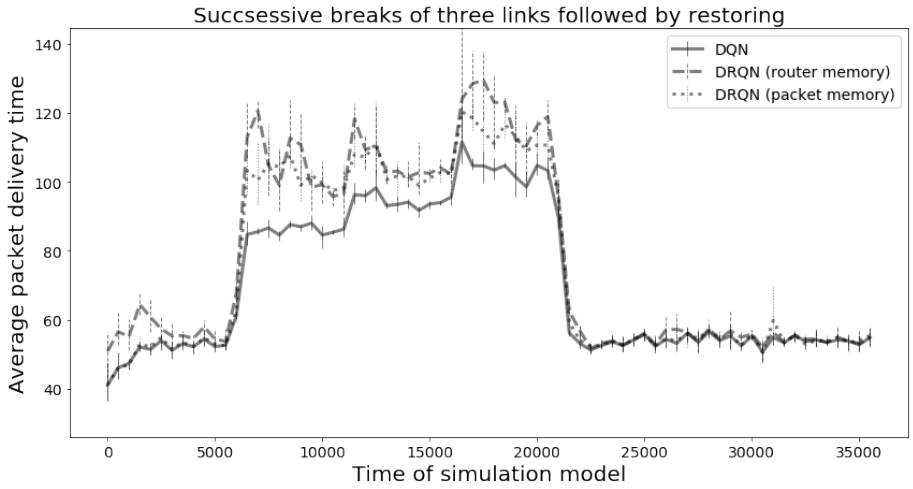


Figure 13: Comparison of different NN architectures in scenario of changing network topology.

with each other in this scenario. It demonstrates that the recurrent neural network architectures perform worse than the simple feed-forward NNs.

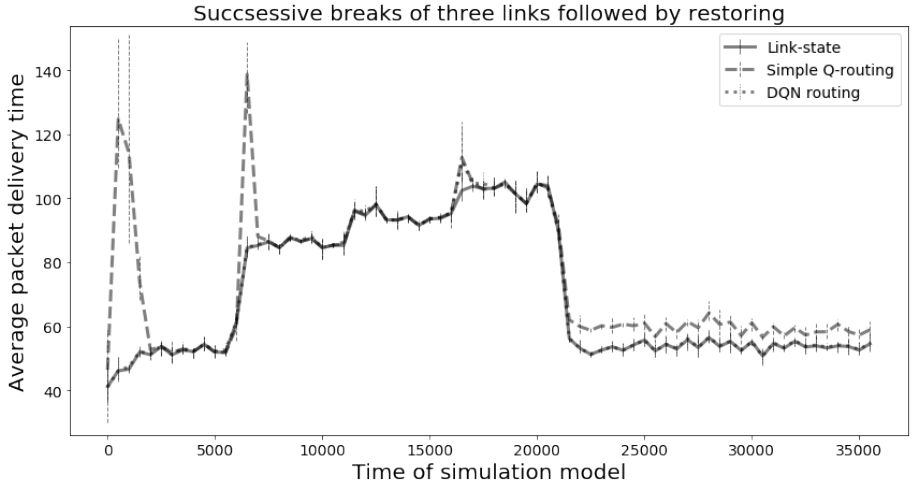


Figure 14: Comparison of DQN-routing and baseline algorithms in scenario of changing network topology.

Figure 14 shows how DQN-routing with feed-forward NNs performs in comparison with the link-state algorithm and Q-routing. Note that in a given scenario (changes in graph topology with moderate traffic load), the link-state algorithm works in an optimal way, as in this conditions travel time between u and v is almost equal to *latency* of link (u, v). The plot shows that DQN-routing

performs equally well with the link-state algorithm throughout almost all scenario, while Q-routing takes time to adapt to link break and cannot restore the optimal strategy when all links are restored.

4.2. Experiments in the simulation model of the baggage handling system

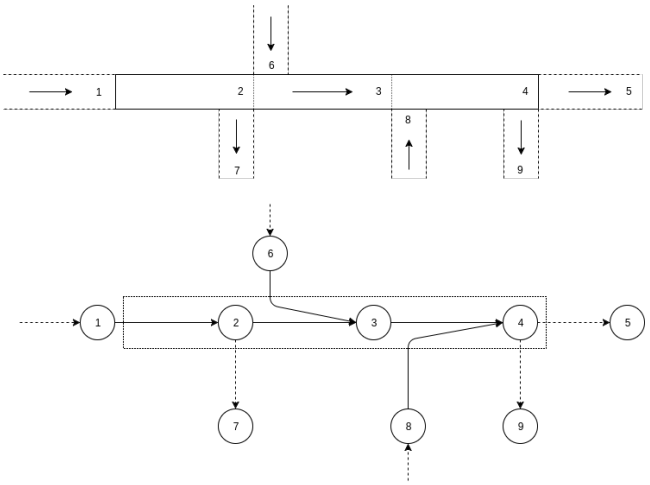


Figure 15: Segment of conveyor network and corresponding segment of graph model.

We model a conveyor network of baggage handling system as an oriented graph $G = (V, E)$, where each *section* of a conveyor is modelled as a node, and edge (u, v) exists if the bag can arrive to section v after leaving section u (Figure 15). Every conveyor has its own *speed*, and every section has its own *length*. Also, we model *entrances* and *exits* of conveyor network as separate nodes.

For BHSs we use a more complex cost function. We simultaneously optimize the average travel time of bags and total the *energy consumption* of the conveyors.

Every conveyor can be in one of two states: *working* and *idle*. In the *working*, state conveyor c consumes e_c *kilowatts* per time unit (for convenience, we will further assume that time units in BHS model are *seconds*). In *idle* state, the conveyor consumes no energy. All conveyors are initially idle. A conveyor goes into the working state when a bag arrives on it. If the conveyor does not transport any bags for d_c seconds while working, it becomes idle.

We achieve simultaneous optimization of bag travel time and energy consumption by counting rewards for actions as $t_p + \alpha e_p$, where t_p is time bag p spent on current section before going to next one, e_p is *energy overhead* caused by bag p entering this section, and α is *energy saving importance coefficient*. Value of e_p is defined as the amount of energy consumed by a conveyor, that transported bag p, which could be saved, if p did not enter the conveyor and conveyor stopped working earlier.

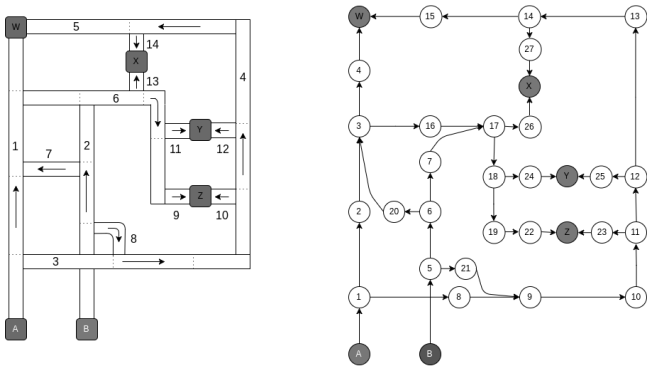
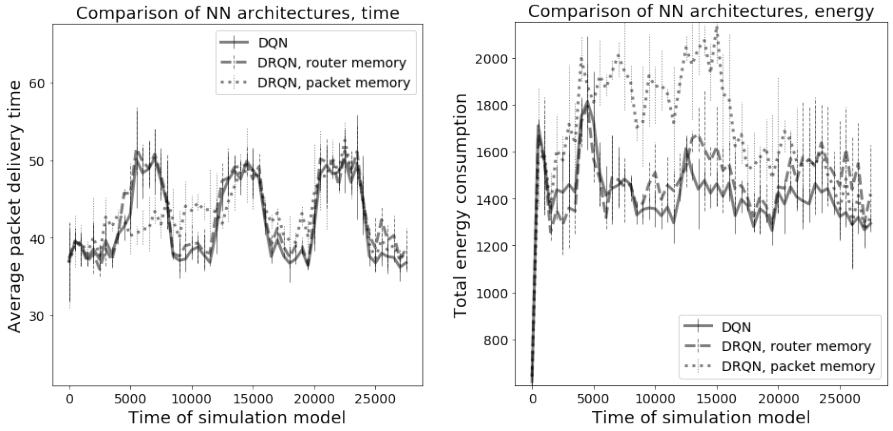


Figure 16: BHS model for tests.

In this experiment, we use a subsidiary protocol in addition to the link-state protocol. This protocol distributes information about working states of the neighboring conveyors. Every conveyor stores a Boolean value for each of its neighbors, which indicates if the neighbor is working or idle. When a conveyor changes its state, it informs its neighbors about it. A vector of neighbor states is added to the input of conveyor neural network similarly to vector y of current neighbors (Section 3.2).

We evaluated test scenarios in BHS model with 14 conveyors consisting of 27 sections in total, two entrances, and four exits (Figure 16). Sections 1-20 have lengths of 10 meters, sections 21-27 have lengths of 2 meters. Every conveyor consumes 1 kW per second, the maximum speed of every conveyor is 1 m/s.



(a) Average travel time

(b) Energy consumption

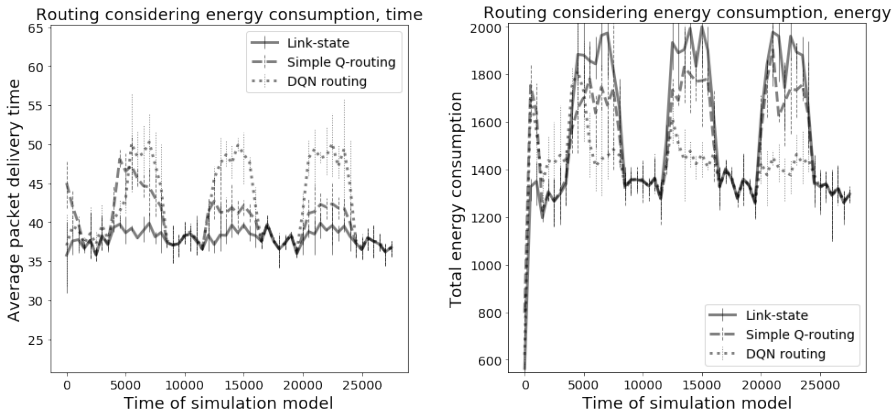
Figure 17: Comparison of different NN architectures in BHS model.

Performance is evaluated in the same way as in Section 4.1.

4.2.1. Uneven traffic to different exits

As it can be seen on Figure 16, the shortest path from entrances to exits Y and Z is via conveyors 3 and 4. But they can be reached via conveyor 6 too, which may be more optimal if conveyors 3 and 4 are idle and we want to minimize the total energy consumption of the system.

In the first testing scenario, traffic pattern switches between bags going only to exits W and X and bags going to all four exits. Energy saving importance coefficient α is set to 1.



(a) Average travel time

(b) Energy consumption

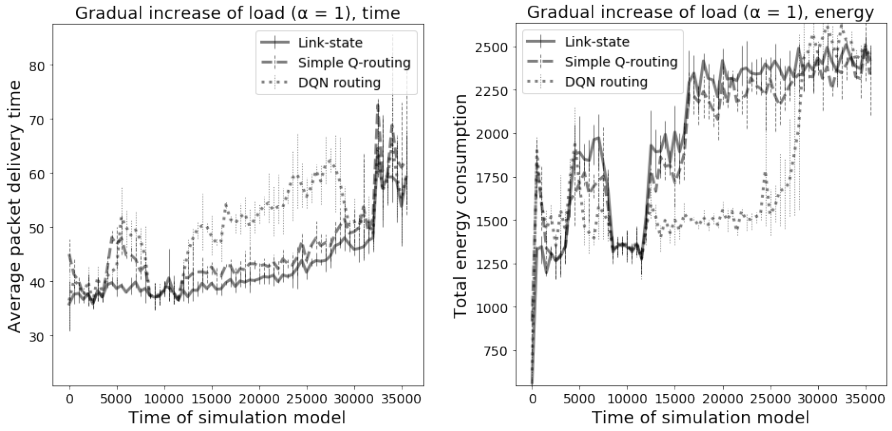
Figure 18: Comparison of DQN-routing with baseline algorithms in BHS model.

Figure 17 shows that in this conditions, DQN-routing with simple feed-forward NNs performs better than variants with recurrent NNs both in terms of average bag travel time and energy consumption. Thus, we will compare the former with baseline algorithms.

As it can be seen on Figure 18, a simple link-state algorithm is unable to optimize for energy consumption because energy consumption overheads cannot be taken into account when assigning static weights for edges before calculating shortest path. On the other hand, the Q-routing algorithm receives the same rewards as DQN-routing but is still unable to optimize energy consumption well enough, while DQN-routing can take info about whether or not neighboring conveyors are idle into account, which allows it to optimize energy consumption better at the expense of average bag travel time.

4.2.2. Gradual increase of traffic load

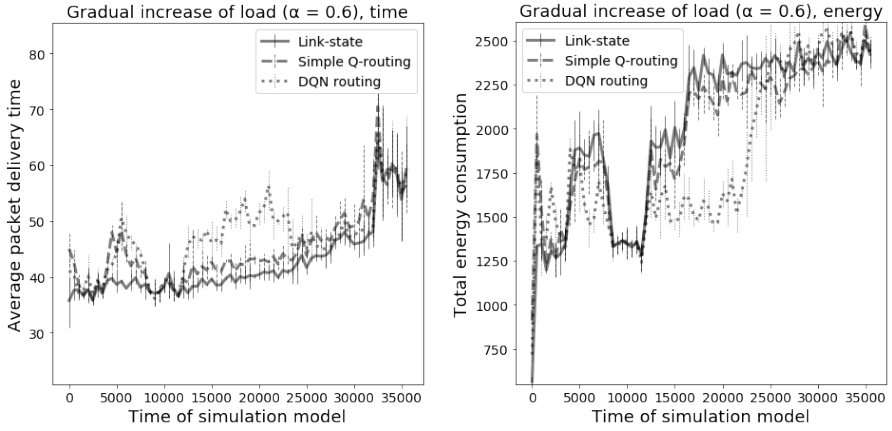
Beginning of this scenario reproduces the previous one. The difference is that the frequency of bags appearing at the entrances grows twice and continues to gradually grow until the end of the scenario. Here we did not compare variants of DQN-routing with different NN architectures, because it was clearly shown



(a) Average travel time (b) Energy consumption

Figure 19: DQN routing compared to baseline algorithms in conditions of increasing traffic load in BHS, $\alpha = 1$.

in the first test scenario that recursive NNs do not perform well in tests for BHS model.



(a) Average travel time (b) Energy consumption

Figure 20: DQN routing compared to baseline algorithms in conditions of increasing traffic load in BHS, $\alpha = 0.6$.

Figure 19 shows that link-state algorithm and Q-routing fail to optimize energy consumption, as in the previous scenario, while DQN-routing prefers to minimize energy consumption at the expense of bag travel time until the end of the scenario, when average bag travel time becomes too high due to increased

traffic load, and traffic finally gets redirected through conveyors 3 and 4.

As can be seen on Figure 20, this behavior can be regulated by changing coefficient α. When α is lowered down to 0.6, redirection of traffic through conveyors 3 and 4 happens earlier.

4.3. Summary of experimental results

In order to present a more clear and compact representation of experimental results, we provide a numeric performance comparison of considered routing algorithms. For every test scenario, we compare the algorithms by average packet travel time, and in case of testing in the model of BHS we also compare them by total energy consumed.

For sake of brevity, in summary tables we use the following abbreviations for routing algorithms:

- "LS" is shortest path algorithm using the link-state protocol
- "QR" is Q-routing algorithm
- "DQN" is simple DQN routing using feed-forward neural networks
- "DRQN-p" is DQN routing using RNNs with packet memory
- "DRQN-r" is DQN routing using RNNs with router memory

Best result within every test scenario is marked bold.

Table 1: Summary of experimental results in the model of computer network.

	Abrupt peak load, average time, ms	Changing the topology, average time, ms
LS	114.8	**70.0**
QR	79.3	75.0
DQN	**63.9**	70.3
DRQN-p	80.3	75.4
DRQN-r	72.8	78.2

Table 1 clearly shows how DQN routing outperforms other algorithms in a network environment with changing traffic load while adapting to topology changes almost as good as a simple link-state algorithm.

Table 2 also shows that DQN routing optimizes energy consumption in the best way possible. Moreover, as we can see in Table 3, the behavior of DQN routing can be regulated by changing the importance coefficient α, and that its behavior changes more easily in comparison with Q-routing.

Overall, we can conclude that DQN-routing which uses feed-forward NNs as learning agents is able to efficiently solve the routing problem in different environments and with a differently defined cost function.

Table 2: Summary of experimental results in the model of baggage handling system: scenario of uneven traffic to different exits.

	Average time, s	Total energy, kW
LS	**38.0**	87,183
QR	40.1	84,273
DQN	42.0	**78,947**
DRQN-p	42.9	82,237
DRQN-r	42.9	94,231

Table 3: Summary of experimental results in the model of baggage handling system: scenario of gradually increasing load.

	$\alpha = 1$		$\alpha = 0.6$	
	Time, s	Energy, kW	Time, s	Energy, kW
LS	**42.3**	145,112	**42.3**	145,112
QR	45.3	140,146	44.8	141,613
DQN	49.8	**123,984**	46.8	**132,193**

5. Conclusion

We presented a novel distributed routing approach that is based on machine learning and can be applied both in communication networks and in physical systems, such as baggage handling systems. The unique strength of the proposed method is its ability to optimize simultaneously the travel time of the routed entities and energy consumption. Comparison with contemporary routing algorithms confirms substantial gain of the proposed method.

However, the proposed method has certain limitations. The necessity of performing a preliminary supervised learning makes the method inconvenient for application in real-world industrial environments, and the dependency of neural network input size on the graph size limits the scalability of the method. Future work is planned to address these issues, possibly with the help of model-based reinforcement learning and neural graph embeddings [28].

Future research directions also include further improvements of the methods performance on account of using different learning techniques, and investigating practical implementation techniques, where it could be implemented straight in the distributed industrial automation platforms.

6. Acknowledgements

Authors would like to thank Arip Asadulaev and Ivan Smetannikov for useful comments. The results were obtained under the research project supported by the Ministry of Education and Science of the Russian Federation, Project No 2.8866.2017/8.9.

7. References

[1] J. Moy, Ospf version 2, RFC 1058 (April 1998).

[2] J. Yan, V. Vyatkin, Distributed software architecture enabling peer to peer communicating controllers, IEEE Transactions on Industrial Informatics 9 (4) (2013) 2200–2209.

[3] J. M. McQuillan, D. C. Walden, The arpa network design decisions, Computer Networks 1 (5) (1977) 243–289.

[4] J. A. Boyan, M. L. Littman, Packet routing in dynamically changing networks: a reinforcement learning approach, Advances in Neural Information Processing Systems (6) (1994) 671–678.

[5] M. Tan, Multi-agent reinforcement learning: Independent vs. cooperative agents, in: Proceedings of the tenth international conference on machine learning, 1993, pp. 330–337.

[6] J. M. McQuillan, I. Richer, E. C. Rosen, The new routing algorithm for the arpanet, IEEE Trans. on Comm. 28 (5) (1980) 711–719.

[7] G. Di Caro, M. Dorigo, Antnet: Distributed stigmergetic control for communications networks, Journal of Artificial Intelligence Research 9 (1998) 317–365.

[8] S. P. . M. Choi, D.-Y. Yeung, Predictive q-routing: A memory-based reinforcement learning approach to adaptive traffic control, Advances in Neural Information Processing Systems (8) (1996) 945–951.

[9] S. Kumar, R. Miikkulainen, Dual reinforcement q-routing: An on-line adaptive routing algorithm, Artificial Neural Networks in Engineering (7) (1997) 231–238.

[10] H. A. Al-Rawi, M. A. Ng, K.-L. A. Yau, Application of reinforcement learning to routing in distributed wireless networks: a review, Artificial Intelligence Review 43 (3) (2015) 381–416.

[11] A. Ghaffari, Real-time routing algorithm for mobile ad hoc networks using reinforcement learning and heuristic algorithms, Wireless Networks 23 (3) (2017) 703–714.

[12] R. M. Desai, B. Patil, Prioritized sweeping reinforcement learning based routing for manets, Indonesian Journal of Electrical Engineering and Computer Science 5 (2) (2017) 383–390.

[13] K. Malialis, S. Devlin, D. Kudenko, Resource abstraction for reinforcement learning in multiagent congestion problems, in: Proceedings of the 2016 International Conference on Autonomous Agents & Multiagent Systems, International Foundation for Autonomous Agents and Multiagent Systems, 2016, pp. 503–511.

[14] R. Rădulescu, P. Vrancx, A. Nowé, Analysing congestion problems in multi-agent reinforcement learning, in: Proceedings of the 16th Conference on Autonomous Agents and MultiAgent Systems, International Foundation for Autonomous Agents and Multiagent Systems, 2017, pp. 1705–1707.

[15] M. Nazari, A. Oroojlooy, L. V. Snyder, M. Takáč, Deep reinforcement learning for solving the vehicle routing problem, arXiv preprint arXiv:1802.04240.

[16] G. Stampa, M. Arias, D. Sanchez-Charles, V. Muntés-Mulero, A. Cabellos, A deep-reinforcement learning approach for software-defined networking routing optimization, arXiv preprint arXiv:1709.07080.

[17] S.-C. Lin, I. F. Akyildiz, P. Wang, M. Luo, Qos-aware adaptive routing in multi-layer hierarchical software defined networks: a reinforcement learning approach, in: Services Computing (SCC), 2016 IEEE International Conference on, IEEE, 2016, pp. 25–33.

[18] M. K. M. Ali, F. Kamoun, Neural networks for shortest path computation and routing in computer networks, IEEE Transactions on Neural Networks 4 (6) (1993) 941–953.

[19] F. Araujo, B. Ribeiro, L. Rodrigues, A neural network for shortest path computation, IEEE Transactions on Neural Networks 12 (5) (2001) 1067–1073.

[20] S. Hochreiter, J. Schmidhuber, Long short-term memory, Neural computation 9 (8) (1997) 1735–1780.

[21] C. Watkins, Learning from delayed rewards, Ph.D. thesis, King's College, Cambridge (1989).

[22] V. Mnih, et al., Human-level control through deep reinforcement learning, Nature (518) (2015) 529–533.

[23] J. Foerster, Y. M. Assael, N. de Freitas, S. Whiteson, Learning to communicate with deep multi-agent reinforcement learning (2016) 2137–2145.

[24] T. Tieleman, G. Hinton, Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude, COURSERA: Neural networks for machine learning 4 (2).

[25] M. D. Zeiler, Adadelta: an adaptive learning rate method, arXiv preprint arXiv:1212.5701.

[26] J. Duchi, E. Hazan, Y. Singer, Adaptive subgradient methods for online learning and stochastic optimization, Journal of Machine Learning Research 12 (Jul) (2011) 2121–2159.

[27] D. Kingma, J. Ba, Adam: A method for stochastic optimization, arXiv preprint arXiv:1412.6980.

[28] B. P. Chamberlain, J. Clough, M. P. Deisenroth, Neural embeddings of graphs in hyperbolic space, arXiv preprint arXiv:1705.10359.