Reinforcement Learning with Reward Shaping for Planning Domains in Jani

Bachelor Thesis

Michael Johannes Kormann

Universität des Saarlandes
Faculty of Mathematics and Computer Science
Foundations of Artificial Intelligence (FAI) Group

September 12, 2022

Acknowledgments

Before I address the actual topic of this bachelor thesis, I would like to thank all employees of the Foundations of Artificial Intelligence (FAI) Group, who helped me with questions, issues, and feedback while working on the topic. I particularly want to thank Prof. Dr. Jörg Hoffmann for making this thesis possible in the first place, Timo Gros, who supported us, especially in matters regarding Reinforcement Learning and Reward Shaping, as well as Marcel Vinzent for supervising my work.

Erklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Statement

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis

Einverständniserklärung

Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

Declaration of Consent

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Saarbrücken,_____ _____

(Datum/Date) (Unterschrift/Signature)

Abstract

Reinforcement Learning tries to obtain optimal policies, according to which an agent can solve given tasks. In our case, the policies will be represented by neural networks. Finding this policy is not always easy, particularly when it comes to large instances of such tasks. In this work, we will try to approach this issue in Planning domains, specified in the framework Jani, by applying Reward Shaping in a setting already specified by Vinzent et al. 2022 [16]. The following work gives a theoretical background on the concepts used, an analysis of the policies obtained, and an outlook on further research topics to follow up on.

Contents

Chapter 1

Introduction

Reinforcement learning became more and more popular as a method to learn policies according to which intelligent agent makes decisions in an environment. In this environment, the agent is usually given a task it is supposed to fulfill. Depending on how the agent performs, it receives a reward. This is then used to optimize the agent's policy and this way its performance. An issue encountered especially in large instances of tasks is that the agent performs poorly since it takes very long to reach the goal state and therefore doesn't receive a reward very often. Without receiving rewards the agent won't improve the policy and is not able to improve the way it performs its task or even fails to solve the task at all. Reward shaping makes it possible for the agent to receive rewards in additional stages of the task.

We apply reward shaping on planning tasks in a framework specified by Vinzent et al. [16]. For this, we use the Jani formalization as well as the code base they used in their work as well.

This work starts with an overview of the background of our work in chapter 2. We therefore address how the state space is formalized. Then a background on reinforcement learning, neural networks, and reward shaping is given. Chapter 3 describes the tasks we are considering in our work, how they are formalized and what reward shaping functions we use depending on the task. Afterward, we display the results accomplished during the work in Chapter 3.3.3. In this section we will give insights on both the progress of learning as well as on how well the learned policies perform, taking both their safety as well as overall performance into account. This chapter is closed with a summary of these results. Finally, in chapter 4.6 we look into future research topics in this area of work.

1.1 Related Work

Since both Reinforcement Learning and AI Planning are very current topics, lots of work has been done in this area, covering topics related to this work.

The general formalization of the state space, independent of the automata language used, was introduced by Vinzent et al. [16] in their work for verifying instead of in this work learning neural network action policies using. We will be using the coding of this project as a basis for our work. Furthermore, the automata language Jani was the basis of their implementation there as well as the implementation of this work. First introduced for qualitative model checking by Budde et al. [4], Jani is the automata language which will also be used for the modeling of the planning domains. Later, the

Python framework *Momba* was created using the JANI-model [11]. Although not directly used for reinforcement learning or the training of agents, *Momba* was used by Gros et al. [7] to create the integrated toolbox MOGYM. MOGYM is meant for the training and verification of such intelligent agents using formal methods.

Regarding the reward shaping part of this work, it originates from Ng et al. [13]. Grzes [8] later publishes further work in this area of research. Since we are considering planning domains in reinforcement Learning for this work, it may be useful to involve heuristics in the learning progress. Gehring et al. [6] try to solve the sparse-rewards issue we encounter in large instances of our tasks as well through applying heuristics as dense reward generators.

Chapter 2

Background

The upcoming chapter gives a background on the technologies and methods used in our work. We start with the formalization of the description of the state space. Afterward, we give a background on reinforcement learning. The last subsection introduces rewards shaping and in particular potential-based reward shaping, which is the method we apply in this work.

2.1 Setting

2.1.1 State Space Formalization

Although we use parts of JANI (Budde et al. 2017 [4]) to implement the following state space, it can also be formalized as follows:

The automata networks used in our work were already defined by Vinzent et al. 2022 [16] for neural network action policy verification via predicate abstraction.

Generic Description

The state space, a three-tuple $\langle \mathcal{V}, \mathcal{L}, \mathcal{O} \rangle$, is defined as follows:

- **state variables** $\mathcal{V}$: each variable $v \in \mathcal{V}$ has a domain $\mathcal{D}_v$, which is a non-empty bounded set of integers
- **action labels** $\mathcal{L}$: the labels of $\mathcal{O}$
- **operators** $\mathcal{O}$: an operator $o \in \mathcal{O}$ is a three-tuple (g, l, u) consisting of the **label** $l \in \mathcal{L}$, the **guard** $g \in \mathcal{C}$ and **update** $u : \mathcal{V} \to Exp$
- with
  - **linear integer expressions Exp**: a polynomial with elements of $\mathcal{V}$ as variables, which all have 1 as an exponent.
    (e.g. $d_r \cdot v_r + ... + d_1 \cdot v_1 + d_0$ with $v_1, ..., v_r \in \mathcal{V}$ and $d_0, .., d_r \in \mathbb{Z}$)
  - **linear integer constraints** $\mathcal{C}$ over $\mathcal{V}$: a linear integer constraint c is:
    - either a comparison of 2 elements e_1, e_2 of Exp: $e_1 \leq e_2, e_1 \geq e_2$ or $e_1 = e_2$

  - or a boolean combination of two linear integer constraints $c_1, c_2 \in C$ (e.g. $c_1 \wedge c_2, c_1 \vee c_2, c_1 \rightarrow c_2$, ...)

Additional Definitions:

- **variable assignment s(v)** over $\mathcal{V}$ (may be a partial variable assignment): takes some variables of $\mathcal{V}$ and assigns each of them a value from their domain $\rightarrow$domain of s $dom(s) \subseteq \mathcal{V}$, codomain of s $codom(s) \in D_v$ for $v \in dom(s)$
- **update of $\mathbf{s_1}$ by $\mathbf{s_2}$** $s_1[s_2]$: $dom(s_1[s_2]) = dom(s_1) \cup dom(s_2)$ with $s_1[s_2](v) = \begin{cases} s_2(v) & v \in dom(s_2) \\ s_1(v) & else \end{cases}$
- **e(s) evaluation of e $\in$ Exp** over s
- ϕ**(s) evaluation of** $\phi \in$ **C**
- **s** $\models \phi$ if $\phi(s)$ evaluates to true

Further, we define the state space of $\langle \mathcal{V}, \mathcal{L}, \mathcal{O} \rangle$ as a labeled transition system (LTS)
LTS $\Theta = \langle S, L, T \rangle$:

- **states S**: the finite set of all complete variable assignments over $\mathcal{V}$
- **transitions** $T \subseteq S \times L \times S$: T contains a transition (s, l, s') if: $\{\exists o \in \mathcal{O} \,|\, s \models g \wedge s' = s[u(s)]\}$
  - $s[u(s)]$ abbreviated by $s[\![o]\!]$; $s \models g$ rewritten to $s \models o$
  - with $u(s)$ as the partial variable assignment induced by u evaluated over s $\rightarrow u(s) = \{v \mapsto u(v)(s) | v \in dom(u)\}$
  - applying the update in s results in s'. The transition is only possible if the guard is satisfied.

Automata Networks [17]

A network of automata $\langle V, \mathcal{L}, A, \Lambda \rangle$ is underlying the generic state space description. It is defined as follows:

- the finite set of **integer state variables** V
- the finite set of **labels** $\mathcal{L}$: includes all labels except for the **silent label** $\tau \notin \mathcal{L}$
- the finite set of **automata A** over V and $\mathcal{L}$
- the finite set of **synchronization constraints** $\mathbf{\Lambda} \subseteq (A \nrightarrow \mathcal{L}) \times \mathcal{L}$

further definitions

- **automaton a** over V and $\mathcal{L}$: tuple $\langle L, E \rangle$
  - non-empty finite set **locations L**

- finite set **edges E**: edge e as a tuple $(l_S, g, l, u, l_d) \in E$ with:
  - source location $l_S \in L$
  - guard $g \in C$ over V
  - $l : \begin{cases} l \in \mathcal{L} & \text{if e is labelled} \\ l = \tau & \text{if e is a silent edge} \end{cases}$
  - update u (may be partial): $V \to Exp$ (over V)
  - destination location $l_d \in L$

Transforming a state space description $\langle \mathcal{V}, \mathcal{L}, \mathcal{O} \rangle$ into an automata network $\langle V, \mathcal{L}, A, \Lambda \rangle$ is defined as follows:

- $\mathcal{V} = V \cup \{v_{loc,a} \,|\, a \in A\}$ with $v_{loc,a}$ as the location variable of a automaton a with $D_{v_{loc,a}} = L(a)$
- $\mathcal{O}$ contains an operator
  - $((v_{loc,a} = l_s) \wedge g, \tau, u[\{v_{loc,a}\} \mapsto l_d])$ for each silent edge $(l_s, g, \tau, u, l_d) \in E(a)$ in each automaton a with $a \in A$
  - (g, l, u) for each synchronization constraint $(\lambda, l) \in \Lambda$ and each combination of edges $e_1 \in E(a_1), ..., e_n \in E(a_n)$ such that:
    - $dom(\lambda) = \{a_1, ..., a_n\}$
    - $e_i = (l_g^i, g^i, l(a_i), u^i, l_d^i)$ for $i \in \{1, ..., n\}$
    - $g = \wedge_{i=1}^{n} (v_{loc,a_i} = l_s^i \wedge g^i)$
    - $u = \bigcup_{i=1}^{n} u^i[v_{loc,a_i} \mapsto l_d^i]$
- There are special naming rules for the the environment modeling silent edges. Taking them is independent of the policy($\pi : \mathcal{S} \to \mathcal{L}$) which controls the agent.

2.1.2 From the State Space to Neural Networks

Deciding which action $l \in \mathcal{L}$ is taken in which state depends on an **action policy** π. Obtaining these policies is our main goal for the bachelor thesis.

Since we will especially be considering large instances of tasks, using a dynamic programming table as often used in reinforcement learning is not possible. The table would take too much space to compute, so instead as already mentioned the action policy is represented by a **neural network** (NN).

2.1.3 Neural Networks

Neural networks have become more and more popular over the last years, especially due to their good performance when used with large amounts of data. Therefore it seems appropriate using them for our task. Benchmarks for this setting already exist, as they were used for neural network policy verification by Vinzent et al. 2022 [16]. We will use the fully connected feed-forward NN, which also was used there.

Structure of the neural network:

- input layer with 1 input for every state variable
- arbitrarily many hidden layers
- output layer with 1 output for each action, from which the policy π is obtained by applying $argmax$

Given this neural network, we define a **NN action policy** for the state space $\langle \mathcal{V}, \mathcal{L}, \mathcal{O} \rangle$ similar to Vinzent et al. [16] as

$$\pi : S \to \mathcal{L}, s \mapsto f_o(f_d(...f_2(f_I(s)))) \tag{2.1}$$

with

- d: the number of layers of the NN
- d_i for $i \in \{1, ..., d\}$: the size of layer i in the NN.

The functions f_m for $m \in \{1, ..., d\} \cup \{I\} \cup \{o\}$:

- $f_i : S \to \mathbb{R}^{d_1}, s \mapsto (s(v_\pi^1), ..., s(v_\pi^{d_1}))$:
  the **input interface** with

  $$v_\pi^j \in \mathcal{V} \text{ for } j \in \{1, ..., d_1\}$$

  as the state variable associated with input neuron j
- $f_i : \mathbb{R}^{d_{i-1}} \to \mathbb{R}^{d_i}, V \mapsto ReLU(W_i \cdot V + B_i)$, for $i \in \{2, ..., d-1\}$:
  the **forwarding function** induced by hidden layer i.
  - $W_i \in \mathcal{Q}^{d_i \times d_{i-1}}$ as the weight of output neuron k in layer $i-1$
    i.e. W_i as a Matrix from layer $i-1$ to layer i
  - $B_i \in \mathcal{Q}^{d_i}$ as the rational bias vector of layer i
- $f_d : \mathbb{R}^{d_{d-1}} \to \mathbb{R}^{d_d}, V \mapsto W_i \cdot V + B_i$:
  the forwarding function induced by the output layer d
- $f_o : \mathbb{R}^{d_d} \to \mathcal{L}, V \mapsto l_\pi^{argmax_{j \in d_d}(V)_j}$:
  the **output interface** with $l_\pi^j \in \mathcal{L}$ for $j \in \{1, ..., d_d\}$ as the action label associated with output neuron j

Real-valued vectors are forwarded from the input to the output layer in a way that each of the *forwarding function* represents one layer in the NN structure.

Due to this definition of NN action policies, it is possible that the action policy selects inapplicable actions for a state. This results from the setting given by Vinzent et al. [16], according to which applicability filtering is non-trivial.

2.2 Reinforcement Learning

2.2.1 Background on Reinforcement Learning

"Reinforcement Learning is learning what to do - how to map situations to actions - so as to maximize a numerical reward signal."(Sutton and Barto 2018 [15]) This is how Sutton and Barto try to define reinforcement learning in one sentence. In general, reinforcement learning aims at finding a policy according to which a task can be solved.

We formally define this task as a Finite Markov Decision Process(MDP)

$$M = (S, \mathcal{L}, T, \gamma, R) \tag{2.2}$$

with

- S: the **finite set of states** (same as S from the definition of LTS)
- $\mathcal{L}$: the **actions** (similar to the action labels $\mathcal{L}$ from the state space description)
- T: the **transition probabilities** $T = P_{sl}(\cdot)|s \in S, l \in \mathcal{L}$, i.e. $P_{sl}(s')$ gives the probability of reaching state s' when applying l in s (in our case $P_{sl}(s')$ is always either 1 or 0, in general it might be in $[0, 1]$ with $\sum_{s' \in S} P_{sl}(s') = 1|\forall s \in S$)
- γ: the **discount factor**
- R: the **reward distributions**

(γ and R are further specified below).

A policy $\pi(s|l)$ for every state $s \in S^{nt}$ (S^{nt} as the set of non-terminal states) is defined as the probability of taking an action $l \in \mathcal{L}$ when in s at time t, i.e. $P(\mathcal{L}_t = l \text{ if } S_t = s)$ with L_t being the applicable actions at this point and S the set of states. The "|" represents that it is a probability distribution over the actions.

To find this policy, the learning agent repeats the task for several iterations, so-called episodes, and learns from what happens during the episodes and especially from what resulting state the episode ends in. This learning progress is guided by rewards, which the agent receives in (intermediate) states. After arriving in a terminal state, the *return* G_t represents all rewards R_i received on the way from the initial state to the terminal state. Usually, the return

$$G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{2.3}$$

is calculated by summing up individual rewards and discounting them with some factor γ. From a mathematical perspective, a *value function*, defined as follows, is used to carry out the search for the optimal policy (we differentiate between a *state-value function*

$$v_\pi(s) \doteq \mathbb{E}_\pi[G_t|S_t = s] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \middle| S_t = s\right] \tag{2.4}$$

for a state and an *action-value function*

$$q_\pi(s, l) \doteq \mathbb{E}_\pi[G_t|S_t = s, \mathcal{L}_t = l] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \middle| S = s, \mathcal{L}_t = l\right] \tag{2.5}$$

for applying an action in a state.

For these functions we use the following:

- $\mathbb{E}_\pi[\cdot]$ being the expectation of a random variable as long as the agent follows policy π
- t being some time step
- G_t theoretically being an arbitrary function of reward sequence received in this task, which is then specified
- R_i being some reward the agent might receive
- γ being the factor by which the rewards are discounted

Further, resolving the sum of G_t leads to the

$$\begin{aligned} G_t &\doteq \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \\ &= R_{t+1} + \gamma \cdot R_{t+2} + \gamma^2 \cdot R_{t+3} + \gamma^3 \cdot R_{t+4} + ... \\ &= R_{t+1} + \gamma \cdot (R_{t+2} + \gamma \cdot R_{t+3} + \gamma^2 \cdot R_{t+4} + ...) \\ &= R_{t+1} + \gamma \cdot G_{t+1} \end{aligned} \tag{2.6}$$

as the recursive definition of the return.
This recursive definition can also be applied to the value function

$$\begin{aligned} v_\pi(s) &\doteq \mathbb{E}_\pi[G_t | S_t = s] \\ &= \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1} | S_t = s] \\ &= \sum_l \pi(l, s) \sum_{s'} \sum_r p(s', r | s, l) \Big[r + \gamma \mathbb{E}_\pi[G_{t+1} | S_{t+1} = s'] \Big] \\ &= \sum_l \pi(l|s) \sum_{s', r} p(s', r | s, l) \Big[r + \gamma v_\pi(s') \Big], \end{aligned} \tag{2.7}$$

leading to the *Bellman equation* defined as

$$v_\pi(s) = \sum_l \pi(l|s) \sum_{s', r} p(s', r | s, l) \Big[r + \gamma v_\pi(s') \Big]. \tag{2.8}$$

The goal of our agent is to solve the task as well as possible, i.e. receive the highest reward achievable in this setting. The optimal policy is supposed to fulfill this criterion. But since there are many actions applicable in each state, there also exist many policies according to which the agent can act. This means, from all those policies we want to determine the policy π_* which has the highest expected return G_t, the *optimal policy*. If we now calculate the state-value functions for the start states or a sample of start states, which this way represents the expected return of a policy, extracting the optimal policy means selecting the state-value function with the highest value. We call this state-value

function the *optimal state-value function*

$$v_*(s) \doteq \max_\pi v_\pi(s) | \forall s \in S. \tag{2.9}$$

Same holds for the *optimal action-value function*

$$q_*(s,l) \doteq \max_\pi q_\pi(s,l) | \forall s \in S \wedge l \in L. \tag{2.10}$$

q_* includes v_* because

$$q_*(s,l) = \mathbb{E}[R_{t+1} + \gamma \cdot v_* s(S_{t+1} | S_t = s, L_t = l)]. \tag{2.11}$$

Intuitively, this is the expected return of applying l in s and then following the optimal policy.

The Bellman equation also holds for the optimal value functions, in form of the *Bellman optimality equation*

$$\begin{aligned} v_*(s) &= max_{l \in L(s)} q_{\pi_*}(s,l) \\ &= \max_l \mathbb{E}_{\pi_*}[G_t | S_t = s, L_t = l] \\ &= \max_l \mathbb{E}_{\pi_*}[R_{t+1} + \gamma G_{t+1} | S_t = s, L_t = l] \\ &= \max_l \mathbb{E}[R_{t+1} + \gamma v_*(S_{t+1}) | S_t = s, L_t = l] \\ &= \max_a \sum_{s',r} p(s', r | s, a)[r + \gamma v_*(s')] \end{aligned} \tag{2.12}$$

$$\begin{aligned} q_*(s,l) &= \mathbb{E}\left[R_{t+1} + \gamma \max_{l'} q_*(S_{t+1}, l') | S_t = s, L_t = l \right] \\ &= \sum_{s',r} p(s', r | s, l) \left[r + \gamma \max_{l'} q_*(s', l') \right]. \end{aligned} \tag{2.13}$$

There are several different methods of reinforcement learning, but for our work, we will be using deep Q learning.

2.2.2 Q-Learning

Temporal-Difference Learning (TD) – Merging Dynamic Programming and Monte Carlo Methods

Two widespread types of reinforcement learning Methods are dynamic programming and Monte Carlo Methods. Both are collections of algorithms to obtain optimal policies to solve a task.

The dynamic programming algorithms in reinforcement learning are such algorithms, which use dynamic programming to calculate value functions and then use these to obtain optimal policies. As mentioned above, one can retrieve an optimal policy from an optimal value function v_* or q_*. Monte Carlo Methods on the other hand do not need complete knowledge of the environment the agent acts in. Instead, it suffices to have

samples of state-action-reward sequences of actual or simulated interaction. The important advantage over dynamic programming is that it is not necessary to have the complete probability distributions of all transitions possible. This means instead of finding the optimal policy over the entire task, only samples are used to obtain this policy. Similar to the Monte Carlo Methods, calculating the policy also does not necessarily need this complete probability distribution. During temporal-difference learning, the estimates are already updated based on intermediate results, which is also done in dynamic programming.

Q-Learning in General

Q-learning is a special case of temporal-difference Learning. It aims at learning an action-value function Q. The update rule works according to

$$Q(S_t, L_t) \leftarrow Q(S_t, L_t) + \alpha \left[R_{t+1} + \gamma \max_l Q(S_{t+1}, l) - Q(S_t, L_t) \right] \tag{2.14}$$

with α being a constant step size parameter.
The respective algorithm for Q-Learning, also known as off-policy TD control, for estimating $\pi \approx \pi_*$ works as follows.

Algorithm 1 Q-Learning (off-policy TD control) for estimating $\pi \approx \pi_*$ [15]

Parameters: step size $\alpha \in (0, 1]$, small $\epsilon > 0$
$Q(s, l) \leftarrow i \,|\, \forall s \in S, l \in L(s)$ for some arbitrary i
$Q(terminal, \cdot) \leftarrow 0$
for each episode **do**
  for each step of episode **do**
    Chose L from S using policy derived from Q (e.g., ϵ-greedy)
    Take action L, observe R,S'
    $Q(S_t, L_t) \leftarrow Q(S_t, L_t) + \alpha \left[R_{t+1} + \gamma \max_l Q(S_{t+1}, l) - Q(S_t, L_t) \right]$
    $S \leftarrow S'$
    if S is terminal **then**
      break loop
    end if
  end for
end for

ϵ-greedy means selecting the action depending on this epsilon. With probability ϵ a random action is selected and with probability $1 - \epsilon$ the action with the highest action value. This way exploring new actions is possible while mostly exploiting the information learned until this point in time.

Q-Learning is an off-policy learning algorithm. This means that at every point during learning there are 2 policies maintained: One policy, Q_q, according to which actions are taken while learning, and one, Q_u which is updated according to the received rewards. Every k-episodes, Q_q is set to Q_u and then the learning continues with the Q_u as the policy according to which the actions are selected.

Q-Learning with Deep Neural Networks [12], [14]

When using Q-Learning without neural networks, the policy is represented by a table such as the one used in dynamic programming algorithms. Since this table is hard to compute for large instances, we instead use deep neural networks in this work. The neural network is used to approximate the optimal Q-Value function, i.e. the neural network receives a state as an input and outputs the Q-Values for the actions applicable in this state.

The updating of the deep neural network works similarly to the approach of Mnih et al. [12]. Since reinforcement learning might be unstable or diverge when using a neural network, some adaptions have to be made in order to use neural networks as policies [12]. This is necessary because neural networks are nonlinear function approximators.

2.2.3 Reward Shaping

Reinforcement learning and in our case Deep Q-Learning aims at learning a policy by repeating a task over and over again. The policy is then adapted according to the reward the agent receives during learning. This means that for the agent to make progress in the process of learning (i.e. adapting the policy towards the optimal one) it needs to receive rewards. If we consider for example a task with a very large state space in which the agent only gets rewarded when reaching a goal state, the agent will not receive rewards very often or even not at all. This happens especially at the beginning of the learning process because by choosing mainly arbitrary actions, it is very unlikely that the agent manages to reach a goal state. Without reaching a goal state, in the cases stated above, the agent does not receive awards and does not make progress towards learning an optimal policy. To counteract this issue, reward shaping is used.

Reward shaping is applied to MDPs as defined above. But instead of then running the reinforcement learning algorithm on the MDP given for the task, we instead run it on a modified version

$$M' = (S, \mathcal{L}, T, \gamma, R') \tag{2.15}$$

of the task.
Furthermore, we define

$$R' = R + F \tag{2.16}$$

as the transformed reward function in M' with

$$F : S \times \mathcal{L} \times S \mapsto \mathbb{R} \tag{2.17}$$

called the **shaping reward function** (a bounded, real-valued function)

So M' is the same task as M, only that the rewards received by the agent are now the sum of the reward from M and some additional value calculated by F.

Since M' is not the formalization to the task we want to obtain an optimal policy for, but an optimized version thereof, using reward shaping only makes sense if the policy learned using M' is equal to the optimal policy π_* for M. To ensure this holds, there are several criteria that have to be met, which were already specified by Ng et al. 1999 [13] and Grzes 2010 [8]. The first characteristic F has to meet is

$$F(s, l, s') = \Phi(s') - \Phi(s) \tag{2.18}$$

with Φ being a function over states. This is necessary to impede positive cycles from being created in π^*_M. Such cycles would mislead the agent from doing what it is supposed to do and instead encourage it to walk through the cycle repeatedly to accumulate rewards. Ensuring the condition above holds leads to $F(s_1, a_1, s'_2) + F(s_2, a_2, s'_3) + ... + F(s_{n-1}, a_{n-1}, s'_n) + F(s_n, a_n, s'_1) = 0$ for any possible cycle.

Including the discounting factor γ into the formula for F, $F : S \times \mathcal{L} \times S \mapsto \mathbb{R}$ is called a **potential-based** shaping function if

$$F(s, l, s') = \gamma\Phi(s') - \Phi(s) \tag{2.19}$$

with

- $\Phi : S \mapsto \mathbb{R}$
- $\forall s \in S - s_0, l \in \mathcal{L}, s' \in S \ (S - s_0 = S \, if \, \gamma < 1)$
- $\Phi(s \in I) = 0$ (I as the set of start states (maybe a sample of start states))
- $\Phi(s \in G) = 0$ (G as the set containing all goal/terminal states)
  (this criterion was not part of the definition by Ng et al. 1999 [13] but instead shown to be necessary by Grzes 2010 [8]).

A potential-based shaping function fulfills the following necessity and sufficiency condition, stating that the obtained optimal policy when learning with $M' = (S, \mathcal{L}, T, \gamma, R+F)$ is equivalent to the optimal policy of $M = (S, \mathcal{L}, T, \gamma, R)$:

- `Sufficiency` If F is a potential-based shaping function, then every optimal policy in M' will also be an optimal policy in M (and vice versa).
- `Necessity` If F is not a potential-based shaping function (e.g. no such Φ exists satisfying $F(s, l, s') = \gamma\Phi(s') - \Phi(s)$, then there exist (proper) transition functions T and a reward function $R : S \times \mathcal{L} \mapsto \mathbb{R}$ such that no optimal policy in M' is optimal in M.

Chapter 3

Specifications of the Tasks and Shaping Functions

The following section gives an overview of what the tasks we are considering look like. First, the tasks at hand are described as well as their modeling in the formalization of the previous chapter. Furthermore, the so-called avoid states and set of initial states are defined according to the work of Vinzent et al. [16]. Then the actual reward shaping functions are defined for each of the instances. Finally, we give some final remarks as well as general specifications.

In our work we look at planning tasks, so they usually are considered in classical planning. The formal representation described in chapter 2 underlies the tasks for our work. All concrete specifications of the tasks are already defined and used by Vinzent et al. [16]. We learn neural networks with two hidden layers of size either 16, 32, or 64 neurons per hidden layer, as defined by Vinzent et al. [16].

3.1 Description of the Tasks

3.1.1 N-puzzle/Sliding Tiles

As already hinted in the title of the task, the n-puzzle task can be scaled to different sizes denoted by n. Aiming at bringing tiles into an intended order, a n-puzzle consists of a $(\sqrt{n+1})^2$ grid with n tiles and one empty field. On this grid, the agent can move the tiles located next to the empty field on the empty field.

For our experiments, we only considered 8-puzzles, so puzzles with 8 tiles and one empty tile on a 3×3 grid.

With regard to the formal representation from chapter 2, n-puzzle has the following formalization:

- variables:
  - one variable for each tile $tile_1, ..., tile_8$, representing the current position of the tile.
  - one variable *empty* as the position of the empty field.

– each variable can have values from 0 to 8 reflecting on one position on the grid. The calculation of this index is given by

$$index(x, y) = x + y \cdot 3 \text{ for } x, y \in \{0, 1, 2\}. \tag{3.1}$$

- action labels and their respective operators
  - *moveleft*: move the tile left of the empty field to the empty field on the grid. This leads to the following assignments:
    - $tile_i = tile_i + 1$ for $i \in \{0, ..., 8\} \wedge tile_i = empty - 1$ ($tile_i$ being the tile left of the empty field)
    - $empty = empty - 1$
  - *moveright*: move the tile right of the empty field to the empty field on the grid. This leads to the following assignments:
    - $tile_i = tile_i - 1$ for $i \in \{0, ..., 8\} \wedge tile_i = empty + 1$ ($tile_i$ being the tile right of the empty field)
    - $empty = empty + 1$
  - *moveup*: move the tile located above the empty field to the empty field on the grid. This leads to the following assignments:
    - $tile_i = tile_i + 3$ for $i \in \{0, ..., 8\} \wedge tile_i = empty - 3$ ($tile_i$ being the tile next above the empty field)
    - $empty = empty - 3$
  - *movedown*: move the tile below the empty field to the empty field on the grid. This leads to the following assignments:
    - $tile_i = tile_i - 3$ for $i \in \{0, ..., 8\} \wedge tile_i = empty + 3$ ($tile_i$ being the tile below the empty field)
    - $empty = empty + 3$

- the goal positions of the tiles and the empty position are the following:

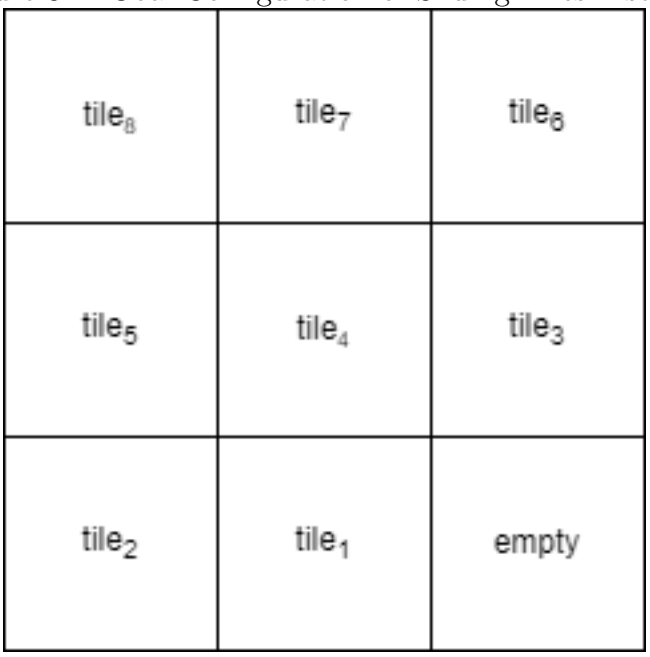
$$goalPositions = [\text{"}empty\text{"} = 8, \text{"}tile_1\text{"} = 7, \text{"}tile_2\text{"} = 6, \text{"}tile_3\text{"} = 5, \\ \text{"}tile_4\text{"} = 4, \text{"}tile_5\text{"} = 3, \text{"}tile_6\text{"} = 2, \text{"}tile_7\text{"} = 1, \text{"}tile_8\text{"} = 0]$$

In addition to the general rules of the task, Vinzent et al. [16] defined unsafe or avoid states for sliding tiles. In 8-puzzle the avoid state is reached if one random tile reaches a certain position.

Solvability of a N-puzzle/Sliding tiles instance [2]

- in opposite to the other tasks considered in this thesis, depending on how the tiles are ordered initially, it is possible that an 8-puzzle is unsolvable. Given an initial state and the goal configuration, it is possible to check whether the goal configuration can be reached:
  The goal configuration from before looked as shown in Figure 3.1:

Figure 3.1: Goal Configuration of Sliding Tiles Instance



Encoding this configuration into a vector and representing each tile by its index and the empty field by 0 gives:

$$(8, 7, 6, 5, 4, 3, 2, 1, 0)$$

In this goal configuration, the tiles are sorted in decreasing order. We can describe every positioning of such tiles as a vector. If the instance is not solved yet, the vector will not be sorted descending. For a vector, we can now count the number of pairs of entries, which are not ordered descending, also known as **Inversions**$(vector)$. The empty tile, so in our case, the position with index 0 is not taken into account. Depending on whether $Inversions$ is an even or an odd number i.e. what the polarity of the vector is, the puzzle is solvable or not.
For example, given the following positioning vector:
$exampleV_1 = (7, 5, 8, 4, 3, 6, 0, 1, 2)$
We get the following not correctly ordered pairs:

- $7 < 8$
- $5 < 8$
- $5 < 6$
- $4 < 6$
- $3 < 6$
- $1 < 2$

So $Inversions(exampleV_1) = 6$ and therefore this instance of the puzzle would be solvable.

Another Example:
$exampleV_2 = (7, 5, 8, 4, 3, 6, 0, 1, 2)$
We get the following not correctly ordered pairs:

- $7 < 8$
- $5 < 8$
- $5 < 6$
- $4 < 6$
- $3 < 6$

$Inversions(exampleV_2) = 5 \rightarrow exampleV_2$ is not solvable.

- the calculation above was used to generate sets of solvable start states

3.1.2 Transport

The planning task transport can be specified in several ways with very different parameters. In general, a transport task always is specified through:

- **locations**: the positions, at which the trucks can be. Some locations are connected to each other
- **trucks**: the trucks which move from one location to another if the locations are connected
- **packages**: the packages, which the trucks can load and unload at locations

The general task then is specified in a way that given some combination of trucks, locations, and packages, the goal is for the trucks to move the packages to other locations. There can be several adaptions of the task, such as the trucks consuming fuel and driving from one location to another is only possible under some conditions. The task we are considering in this work is specified as follows:

- 10 locations indexed from 0 to 9. A location l_i is connected to another location l_j if $|i - j| = 1$
- driving from location 8 to 9 and from location 9 to 8 has as an additional constraint, that the truck only has 1 package loaded
- there are 15 packages, which are initially distributed at random across the 10 locations
- the goal is that all packages are transported to l_9
- there is one truck driving between the locations
- the truck can load up to 15 packages

Looking at our formal representation from chapter 2, the task is formalized in the following way:

- variables:
  - one variable for each location representing the current amount of packages placed there → $\{locationLoad_0, ..., locationLoad_9\}$
  - one variable for the current location of the truck → *truckPosition*
  - one variable for the current load of the truck → *truckLoad*
  - one variable indicating whether the capacity limit between two locations was violated → *capacityLimit*
- action labels and the respective operators:
  - *driveForward*: increases *truckPosition* by one. Has no applicable operator if $truckPosition = 9$. If $truckposition = 8$ and $truckLoad = 1$, applying this action leads to a violation of safety indicated by *capacityLimit*.
  - *driveBackward*: decreases *truckPosition* by one. Has no applicable operator if $truckPosition = 0$ or $truckLoad > 1$.
  - *pickup*: decreases $locationLoad_i$ with $i = truckPosition$ by one and increases *truckLoad* by one. Has no applicable operator if $locationLoad_i = 0$
  - *drop*: decreases *truckLoad* by one and increases $locationLoad_i$ with $locationLoad_i = truckPosition$ by one. Has no applicable operator if $truckLoad = 0$

The agent reaches an unsafe or avoid state in the transport task if he exceeds the capacity limit of a road. In our formalization, this can only occur if the truck has more than one package loaded and drives from location 8 to location 9 or backward from location 9 to location 8.

3.1.3 Blocks World

Blocks World is one of the most famous planning domains. It in general consists of some number of uniquely identifiable blocks distributed across a table. The learning agent can pick up one block at a time and stack it on another block. At every point in time, there can only be one block stacked directly on top of another. Furthermore, the robot can only pick up those blocks that currently do not have other blocks stacked on top of them. The objective of the agent is to stack the blocks in a certain order onto each other. In our tasks, the blocks are numbered so they can be distinguished. The goal is for the agent to stack them forming one tower with the blocks being stacked in ascending order, i.e. the block with the largest index is at the top and the one with the smallest is standing on the table. We considered several tasks with different numbers of blocks:

- 4 blocks (indexes from 0 to 3)
- 6 blocks (indexes from 0 to 5)
- 8 blocks (indexes from 0 to 7)
- 10 blocks (indexes from 0 to 9)

Specifying the task in the formalization from chapter 2 leads to the following variables, action labels, and operators:

- variables:
  - One variable representing whether the hand of the agent is currently empty
    $\rightarrow$ *handEmpty*
  - One variable counting the number of blocks currently positioned on the table
    $\rightarrow$ *tableCounter*
  - One variable for each block, indicating the current position of a block.
    $\rightarrow block_i | i \in \{0, ..., number\,of\,blocks - 1\}$
    These variables can take values from 0 to the number of blocks, with 0 representing the block β_i being in the hand of the agent, 1 β being on the table, and each other value β_i being stacked on another block β'. The larger this value gets, the larger the index of the β'.
  - One variable for each block, indicating whether the agent can currently pick up the block or if it is blocked by either other blocks stacked on top of it or the block is already in the hand of the agent.
    $\rightarrow clear_i | i \in \{0, ..., number\,of\,blocks - 1\}$
- action labels and their respective operators:
  - *table*: if the agent has a block in his hand ($block_i = 0$), this action label leads to the block being put on the table ($block_i = 1$)
  - $moveBlock_i for i \in \{0, ..., number\,of\,blocks - 1\}$: depending on the state, the following happens:
    - if $clear_i = 1 \wedge handEmpty = 1$, i.e. if the block with index i is not blocked and the hand of the agent is empty, then this action leads to the block being picked up by the agent. ($block_i = 0, clear_i = 0, handEmpty = 0$)
    - if $clear_i = 1 \wedge handEmpty = 0 \wedge$
      $block_j = 0 | j \in \{0, ..., number\,of\,blocks - 1\} \backslash \{i\}$,
      i.e. if the block with index i is not blocked and there is another block with index j in the hand of the agent, then this action leads to the block with index j being stacked on top of the block with index i.
      ($handEmpty = 1, clear_i = 0,$
      $block_j = \begin{cases} 2 + i & if\ i < j \\ 2 + (i - 1) & else \end{cases}$)
- the goal state is encoded as follows:

$$block_i = i + 1 | \forall i \in \{0, ..., number\,of\,blocks - 1\}$$

In Blocks World, an unsafe or avoid state is reached by the Agent if all blocks are lying on the table, so there is no block in the hand of the agent and there are no two or more blocks stacked on each other.

3.2 Start States of the Tasks

The set of start states we particularly want the policies to be safe on and perform well is already defined by Vinzent et al. [16]. The sets look as described in the following sections.

- For **n-puzzle**, the set of start states contains states fulfilling a rule regarding the tiles. We are given all the sets of tiles, which are then partitioned into disjoint subsets. Within each subset, for each tile it must hold that when sorting the indexes of the tiles in ascending order, the positions of the tiles are sorted in ascending order as well.
- For **Transport**, the start states have to fulfill the criterion that initially each package is positioned left of the bridge described above. A package may be loaded.
- For **Blocks World**, the blocks have to be positioned either on the table or on top of blocks with larger indexes than their own. Additionally, the unsafety condition must not be fulfilled, i.e. not more than a certain number of blocks may be placed on the table.

3.3 Reward Shaping Functions

As mentioned in the chapter 2, each potential-based reward shaping function has to fulfill the criteria that the potential of initial and terminal states is 0. Additionally to the definitions in the upcoming section, the functions below fulfill these criteria.

3.3.1 N-puzzle/Sliding Tiles

While both Transport and particularly Blocks World are tasks that are quite simple for people to solve, solving a n-puzzle is not as intuitive for people to solve. Therefore there is no intuitive approach from which we can derive shaping functions for n-puzzle.

The shaping functions we used for learning the networks are:

1. **goal count**: during learning, the agent receives an additional reward of 10 for each tile that is located at its goal position. Encoding this into potential-based rewards leads to the function

$$\Phi_{p_1} = \sum_{i \in \{0,\dots,8\}} (tile_i == goalPositions[\text{"}tile_i\text{"}]) \cdot 10 + (empty == goalPositions[\text{"}empty\text{"}]) \cdot 10. \tag{3.2}$$

2. **editing distance**: during editing, the agent receives an additional reward depending on how much the x and y coordinate of each tile and the empty field differ from the goal position of this tile. Encoding this into potential-based rewards leads to the function

$$\Phi_{p_2} = 36 - \sum_{i \in \{1,\dots,8\}} ed(tile_i) - ed(empty) \tag{3.3}$$

with

$$\begin{aligned} ed(te) = &|goalPositions[te].xCoordinate - te.xCoordinate| \\ &+ |goalPositions[te].yCoordinate - te.yCoordinate|. \end{aligned}$$

3. **solving approach**: when trying to solve an 8-puzzle as a human player, one approach to solving it is the following:

   (a) you move the tile which belongs in the top left corner to the top left corner

   (b) you move the tile which belongs in the top right corner to the top right corner

   (c) you move the tiles belonging in the top row into their goal positions

   (d) you move the tiles belonging in the left column into their goal positions

   (e) you move the leftover tiles into their goal positions

   Trying to guide the agent to try and use this approach leads to the function

$$\Phi_{p_3} = \begin{cases} 26 & \text{if } tile_8 = 0 \wedge tile_6 = 2 \wedge tile_7 = 1 \wedge tile_5 = 3 \wedge tile_2 = 6 \\ 21 & \text{if } tile_8 = 0 \wedge tile_6 = 2 \wedge tile_7 = 1 \\ 15 & \text{if } tile_8 = 0 \wedge tile_6 = 2 \\ 8 & \text{if } tile_8 = 0 \end{cases} \tag{3.4}$$

   for potential-based reward shaping.

4. **solving approach and editing distance**: The agent receives additional rewards during learning through p_2 and p_3. Encoding this into potential-based rewards leads the function

$$\Phi_{p_4} = \Phi_{p_2} + \Phi_{p_3}. \tag{3.5}$$

3.3.2 Transport

For Transport we considered the following reward shaping function:

1. **Number of packages at goal**: During learning, the learning agent receives an additional reward of 10 for each package located at l_9. Encoding this into potential-based rewards leads to the function

$$\Phi_{t_1}(s) = locationLoad_9 \cdot 10. \tag{3.6}$$

2. **Number of packages loaded**: During learning, the learning agent receives an additional reward of 10 if 1 package is loaded and of -5 for each package loaded in addition to this package. Since the truck is not able to enter $locationLoad_9$ while

having more than one package loaded, this shaping function is on the one hand supposed to encourage the agent to load one package and on the other not to load any other packages. Encoding this into potential-based rewards leads to the function

$$\Phi_{t_2}(s) = \begin{cases} 10 & \text{if } truckLoad = 1 \\ (truckLoad - 1) \cdot (-5) & \text{if } truckLoad > 1 \end{cases}. \tag{3.7}$$

3. **Number of packages loaded and number of packages at goal**: A combination of the shaping functions above. Encoding this into potential-based rewards leads to the function

$$\Phi_{t_3} = \Phi_{t_1} + \Phi_{t_2}. \tag{3.8}$$

3.3.3 Blocks World

1. **Blocks at their goal positions**: The learning agent receives a reward of 10 for each block that is at its goal position. Encoding this into potential-based rewards leads to the function

$$\Phi_{b_1} = \sum_{i|i\in\{0,\ldots,number\,of\,blocks-1\}} 10 \cdot (block_i == i+1). \tag{3.9}$$

Possible issues with this shaping function: This shaping function has the issue that the agent may receive high rewards in states, in which it is necessary to first unstack many blocks and then stack them again in a different order to reach the goal. Let us look at an example with 4 blocks Figure 3.2.

The state in the example has a potential of 20 since the blocks with indexes 1 and 2 are positioned at their goal position. In order to reach the goal from this state, the agent would have to first unstack the blocks with indexes 0, 1 and 2 again, leading to it entering states with smaller potential than the example state. The shaping function therefore might lead to the agent learning a policy aiming to get the blocks in the state from the example, although this state is bad if the actual goal of the agent is to reach the goal state.

2. **Blocks at their goal positions recursive checking**: The learning agent receives a reward of 10 for each block that is at its goal position as long as the blocks with smaller indexes are at their goal positions as well. We use

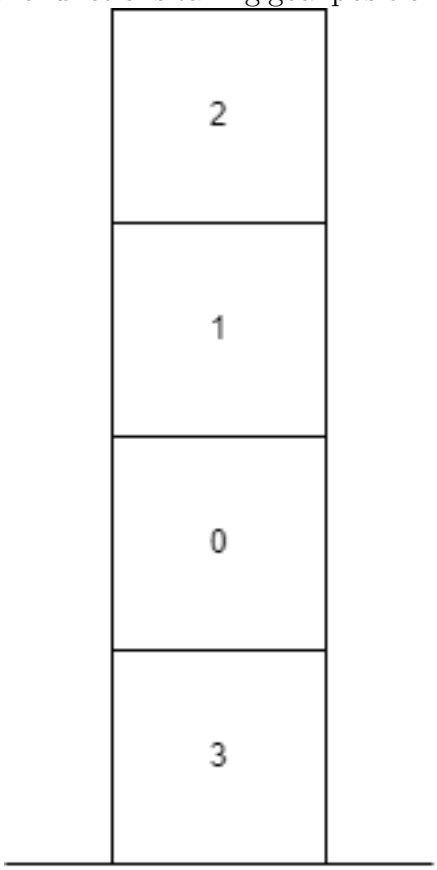
$$\Phi_{b_2} = \sum_{i|i\in\{0,\ldots,number\,of\,blocks-1\}} 10 \cdot belowAtGoal(block_i) \tag{3.10}$$

with

$$belowAtGoal(block_i) = \begin{cases} (block_i = i+1) & \text{if } i = 0 \\ (block_i = i+1) \wedge belowAtGoal(block_{i-1}) & \text{if } i > 0 \end{cases}$$

as the function for reward shaping.

Figure 3.2: Example where the functions taking goal positions into account are misguiding



Possible issues with this shaping function: Looking at Figure 3.3 Φ_{b_2} would encounter the same issue as Φ_{b_1}. Still, there are cases where Φ_{b_2} does not give a high reward on a bad state while Φ_{b_1} does. An example of this is shown in Figure 3.3. Once again, several unstackings would be necessary here to reach the goal state, but Φ_{b_1} assigns this state a potential of 30. By recursively checking the underlying blocks as well, Φ_{b_2} detects the issue in this state and therefore assigns a potential of 0.

3. **Penalties on wrong stacking**: The learning agent receives a negative reward of 10 for blocks positioned on a height, which is not the table, the hand of the robot, or their goal height. Furthermore, the learning agent receives an additional negative reward of 10 for blocks that are stacked on top of a block with a larger index than their own.

   Encoding this into potential-based rewards yields the function

$$\Phi_{b_3} = \sum_{i|i\in\{0,\dots,number\ of\ blocks-1\}} \Big((-10)\cdot(\neg(block_i = i{+}1 \vee block_i < 2)) + (-10)\cdot(block_i > i{+}1)\Big). \tag{3.11}$$

   For example, if we are considering a Blocks World instance with 6 blocks, we are

given the scenario where the block with index 3 is stacked on top of the block with index 5, the block with index 2 is stacked on top of the block with index 3, and the block with index 1 is stacked on top of the block with index 2 (Figure 3.4). All other blocks are lying on the table. Formalized and ignoring the variables *handEmpty* and *tableCounter* this example would look as follows:

$$block_5 = 1, block_3 = 6, block_2 = 4, block_1 = 3, block_0 = 1, block_4 = 1$$
$$clear_5 = 0, clear_3 = 0, clear_2 = 0, clear_1 = 1, clear_0 = 1, clear_4 = 1$$

This state would have a potential of -60, because:

- $\neg(block_1 = 1 + 1 \vee block_1 < 2)$
- $\neg(block_2 = 2 + 1 \vee block_2 < 2)$
- $\neg(block_3 = 3 + 1 \vee block_3 < 2)$
- $block_3 > 4$
- $block_2 > 3$
- $block_1 > 2$

4. **Blocks at their goal positions recursive checking & Penalties for wrong stacking**: Adds up the potential of Φ_{b_2} and Φ_{b_3}. Therefore we use the function

$$\Phi_{b_4} = \Phi_{b_2} + \Phi_{b_3}. \tag{3.12}$$

By combining Φ_{b_2} and Φ_{b_3}, this function aims avoiding the bad estimation given by Φ_{b_2} in Figure 3.2. While the potential of the state in Figure 3.2 calculated by Φ_{b_2} would be 30, Φ_{b_3} recognizes that the block with index 0 is stacked on top of the block with index 3 and therefore assigns a penalty of -20. This leads to a total potential of 10, which gives a better estimate than 30.

Another advantage of the combination of the shaping functions is that it guides larger parts of the search space. Φ_{b_2} on its own only guides the agent towards building the tower with the blocks in the right way, but if the blocks are stacked in the wrong order, Φ_{b_2} gives no feedback on what good actions might be. For example in Figure 3.5, Φ_{b_2} assigns both example 1 and example 2 the same potential 0, although for solving example 1 several unstackings are necessary before stacking them again in the right order to reach the goal. Vice versa, Φ_{b_3} on its own only punishes the learning agent if the blocks are stacked in the wrong order. In Figure 3.6 the upper state shows a scenario, where the goal is almost reached, while in the lower state several additional actions have to be applied. Still, according to Φ_{b_3}, they both have the potential 0. The combination by building the sum of Φ_{b_2} and Φ_{b_3} is able to differentiate the states in Figure 3.5 and in Figure 3.6. This way it should guide the learning of the agent in both scenarios towards learning policies that choose good actions.

Figure 3.3: Example where Φ_{b_2} gives a better estimate than Φ_{b_1}

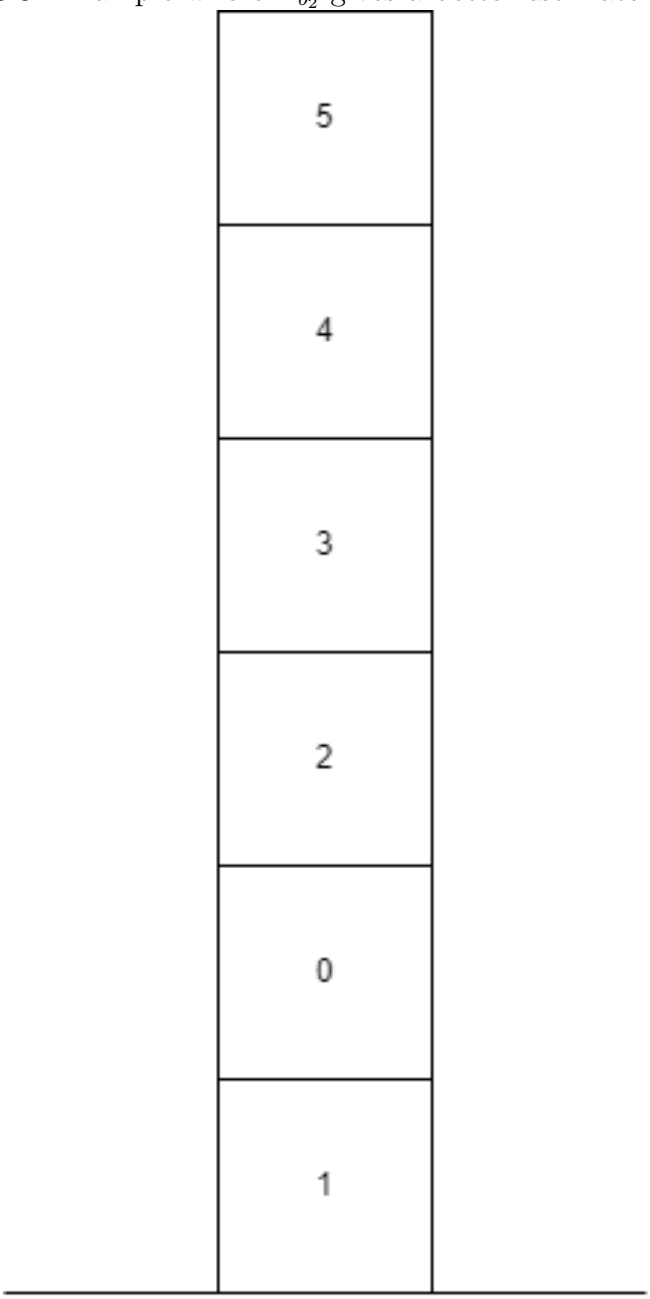


Figure 3.4: Example for Φ_{b_3}

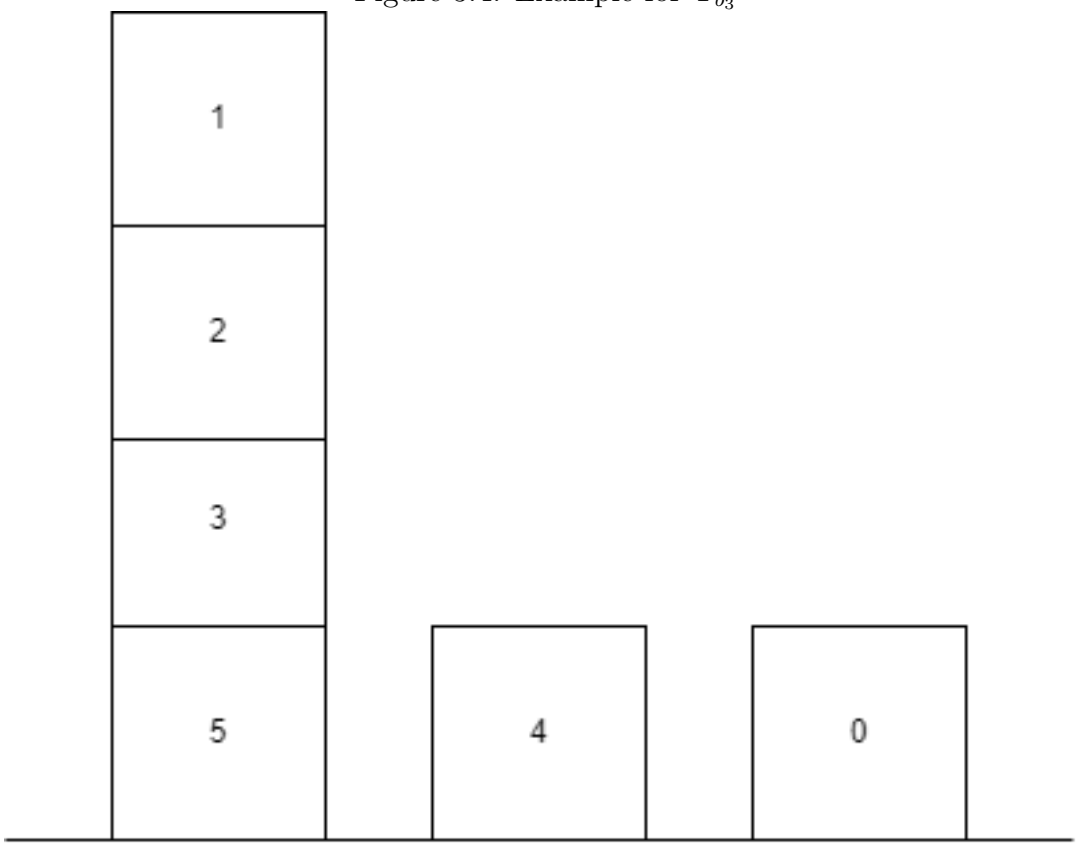
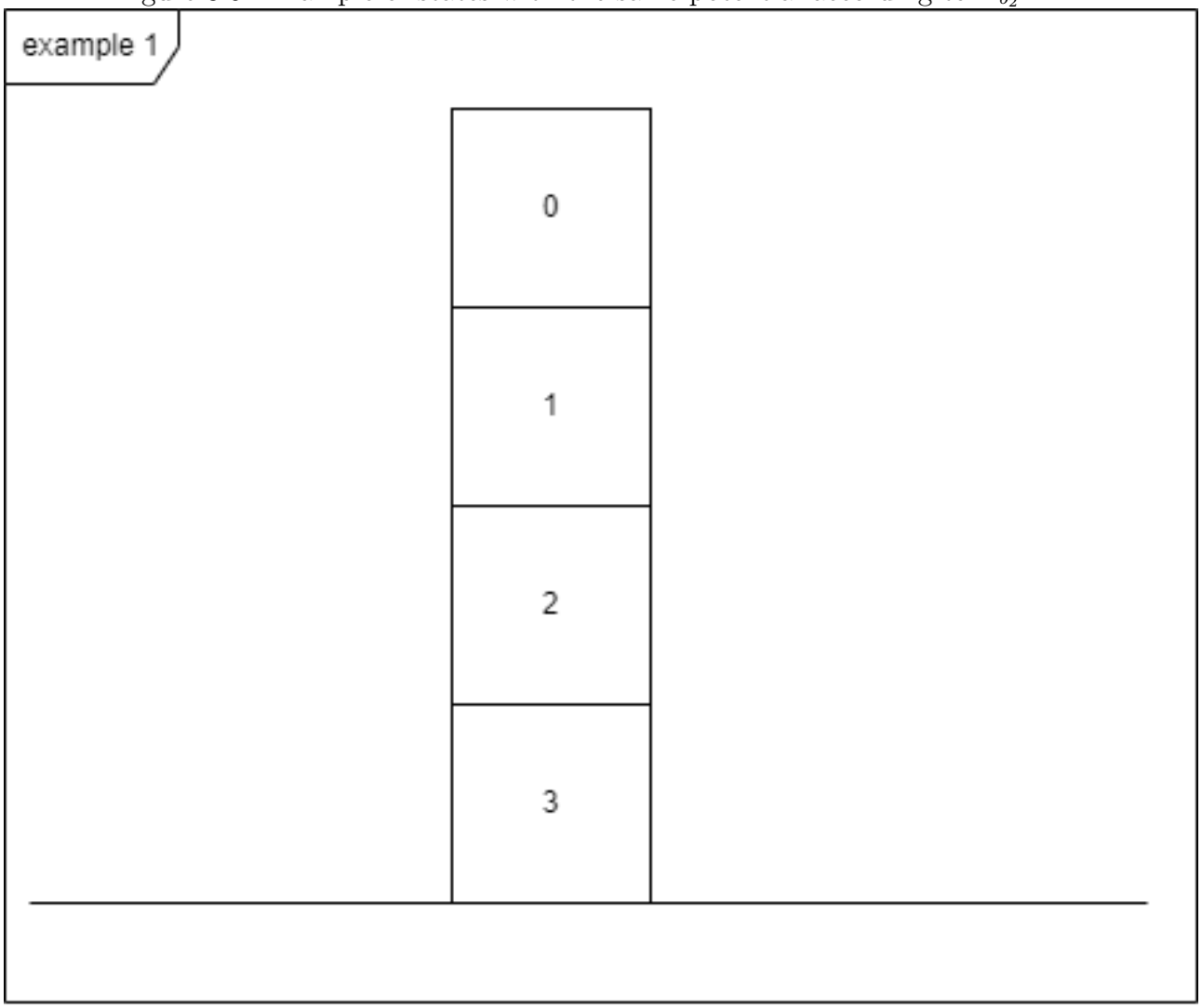


Figure 3.5: Example of states with the same potential according to Φ_{b_2}



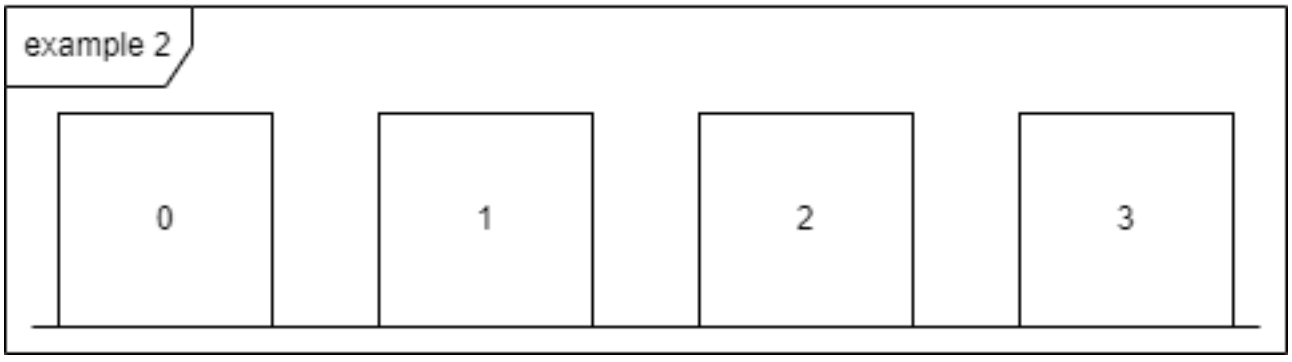
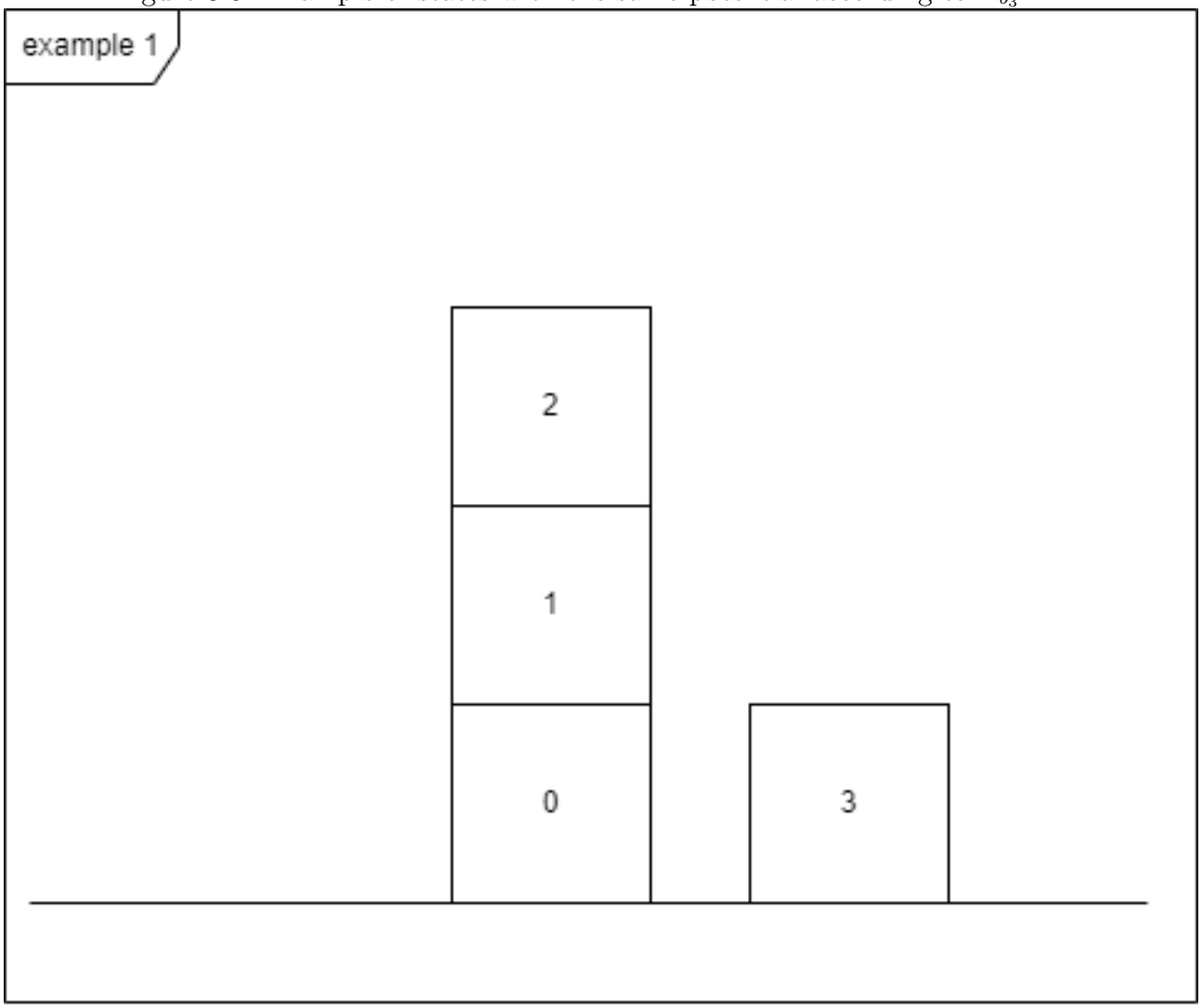
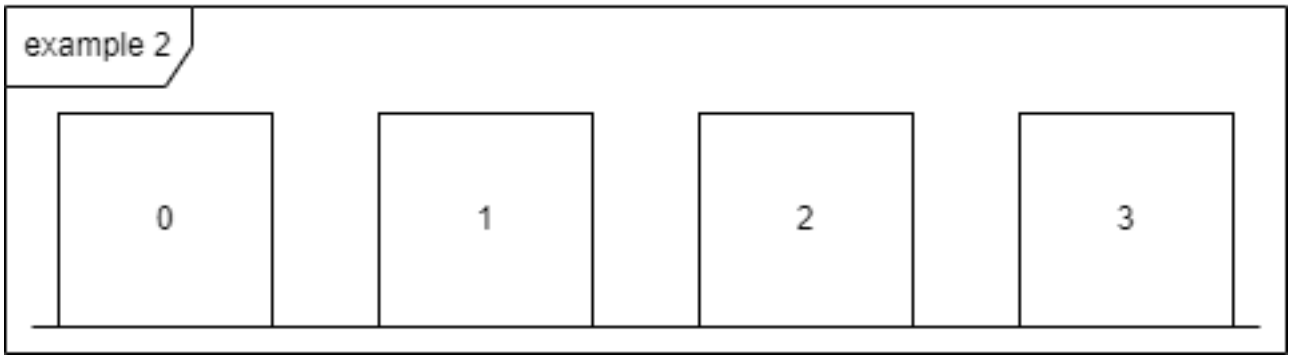


Figure 3.6: Example of states with the same potential according to Φ_{b_3}





Chapter 4

Results

This chapter states the results retrieved from learning policies and evaluates them on different sets of start states. We start with a description of the framework which was used and what general conditions were met during learning. Afterward, we evaluate the performance of the learned networks and compare different shaping functions to each other and to other approaches. The last section sums up these results.

4.1 Specifications on the Implementation

As already mentioned in chapter 1, the implementation used for policy learning is the same as used by Vinzent et al. [16]. This implementation already supported a basic version of reward shaping, not yet fulfilling the criteria necessary for potential-based reward shaping. Therefore it was adapted to meet the respective criteria.

4.2 Specifications on Learning and Evaluation

For the learning of each task, the number of episodes is limited to 100000. Exceptions from this are mentioned below. Furthermore, the time limit for both learning and evaluation is set to 4000 seconds. During the evaluation, cycles are terminated when occurring. In the tasks n-puzzle and Blocks World, there are two configurations of networks, one which takes the action cost into account and one which does not. The shaping functions are applied for learning both cases. Learning and evaluation are conducted for each instance for each number of neurons per hidden layer.

When the agent reaches a goal state, he receives an award of 100, when reaching a avoid state, the award is -100. These rewards are independent of the reward shaping applied.

N-puzzle/sliding tiles: Since we were still interested in finding safe policies on the benchmarks given by Vinzent et al. [16], the evaluation was conducted on both a set of only solvable start states and the set given by Vinzent et al., which does not guarantee to only contain solvable states. The sets of data used for training and evaluating the networks in addition to the set from Vinzent et al. [16] contain 120000 randomly sampled states each. The set from Vinzent et al. [16] contains 4900 states.

Transport: Evaluating how the learned networks perform on the state set used by Vinzent et al. [16] is not feasible due to the size of the start set of interest. We instead

use a subset of the set of start states described in chapter 3. Therefore the following sets are used for learning and evaluation of the policies in the Transport benchmark:

- for learning: two sets of start states, one containing 1000 and one containing 5000 states.
- for evaluation: the evaluation was conducted first on the set which was also used for learning the networks and afterward on a newly generated set consisting of 10000 states
- All sets were generated at random using an implementation already specified by Vinzent et al. [16]

Blocks World: For Blocks World, the set of start states used for learning the Neural Neural Network is the one used by Vinzent et al. [16]. The Policies were evaluated on the set used for learning (Training set) as well as on an additionally generated set of random start states (Test set), generated using a generator also provided by Vinzent et al. [16]. The number of states used depending on the number of blocks is as follows:

Number of blocks / Set of start states	Training Set	Test set
4 blocks	14	72
6 blocks	202	4050
8 blocks	4139	100001
10 blocks	115974	100001

For Blocks World instances with 4 blocks, the number of episodes considered was reduced to 40000 instead of 100000.

4.3 Policies in Addition to the Shaping Functions

In addition to comparing the policies applying the shaping functions from chapter 3, we took some additional approaches into account. First of all, we compare the policies to those by Vinzent et al. [16]. Then we learned networks without applying any reward shaping. Furthermore, we conducted learning, during which the agent always acted epsilon greedy. Finally, we also evaluated the performance of an agent, which ignores the policy in the first place and instead always acts epsilon greedy.

4.4 Process of Learning

As can be seen in the evaluation of the neural networks learned as policies for the tasks Transport and n-puzzle (Figures A.1 – A.10),

the policies obtained applying reward shaping perform poorly. The only instances where reward shaping leads to significantly better performances than no shaping are the large instances of Blocks World, i.e. the instances with 8 and 10 Blocks. This advanced performance is also reflected in the learning curve of the shaping functions. When looking at the figures displaying the learning curves, we see that applying reward shaping leads to significantly larger rewards than not. The learning progress made in the first episodes is also higher when reward shaping is applied (Figures 4.1 – 4.4).

Figure 4.1: Learning Curves Blocks World 8 Blocks Cost Aware

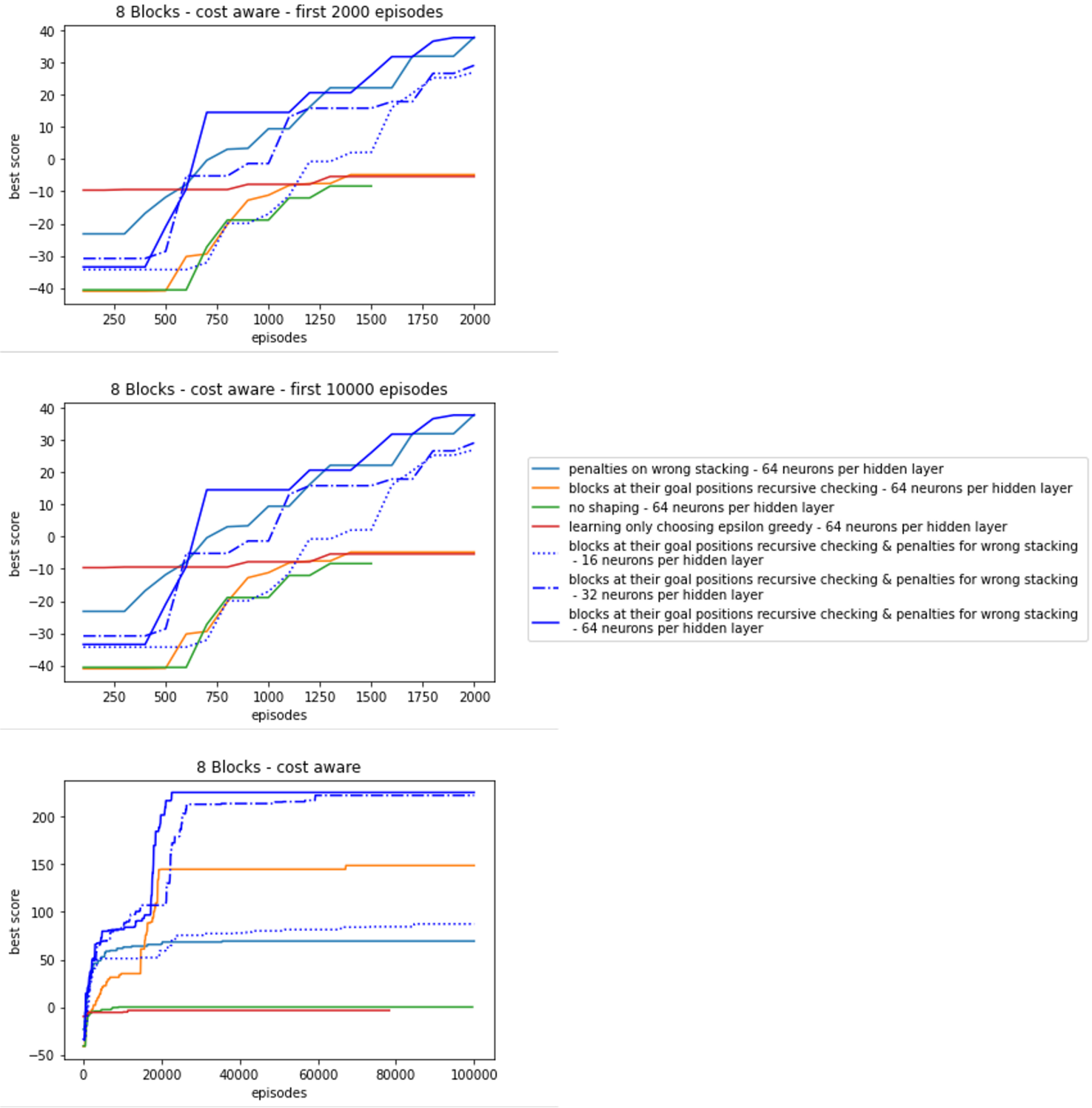


Figure 4.2: Learning Curves Blocks World 8 Blocks Cost Ignoring

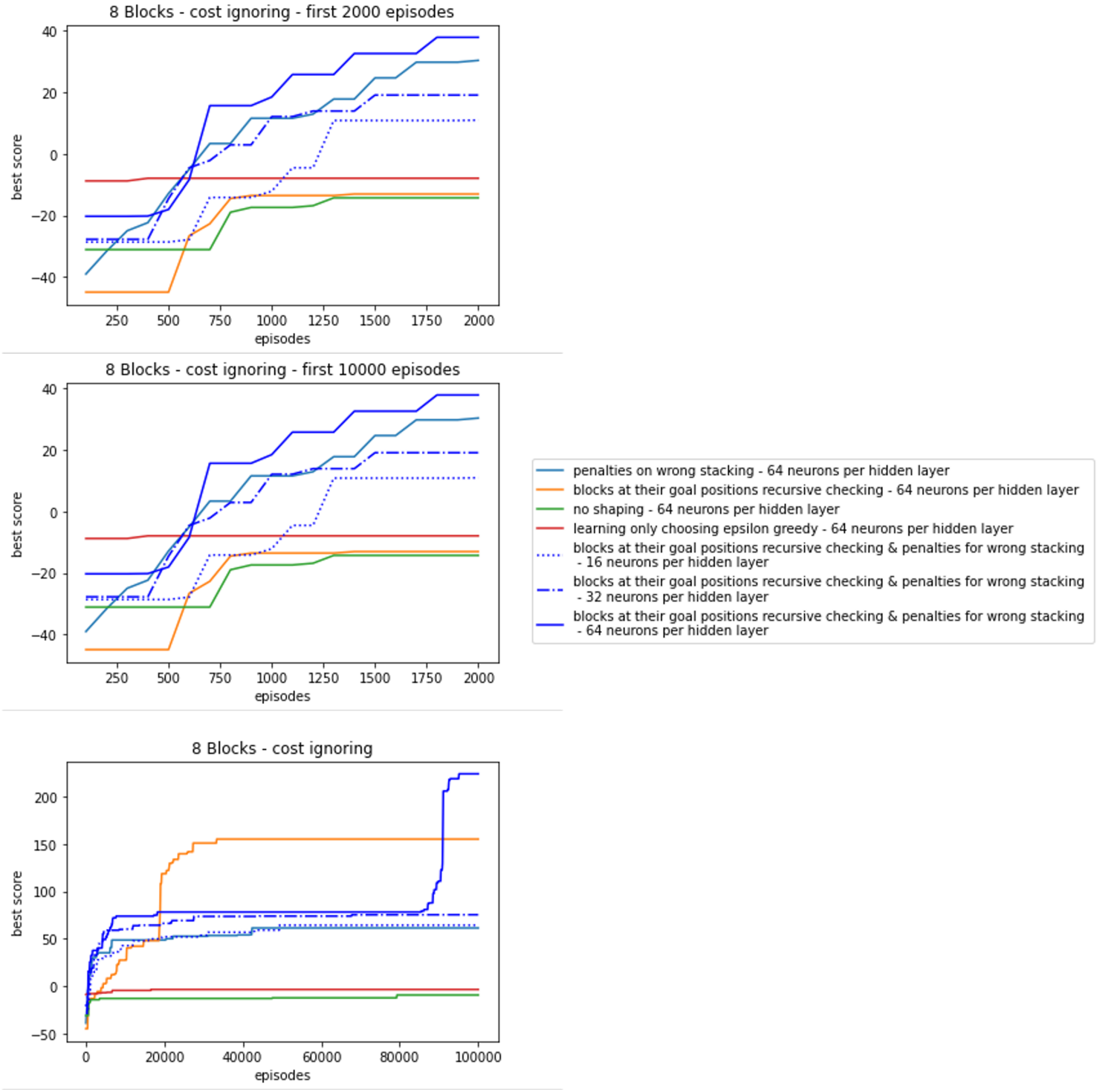


Figure 4.3: Learning Curves Blocks World 10 Blocks Cost Aware

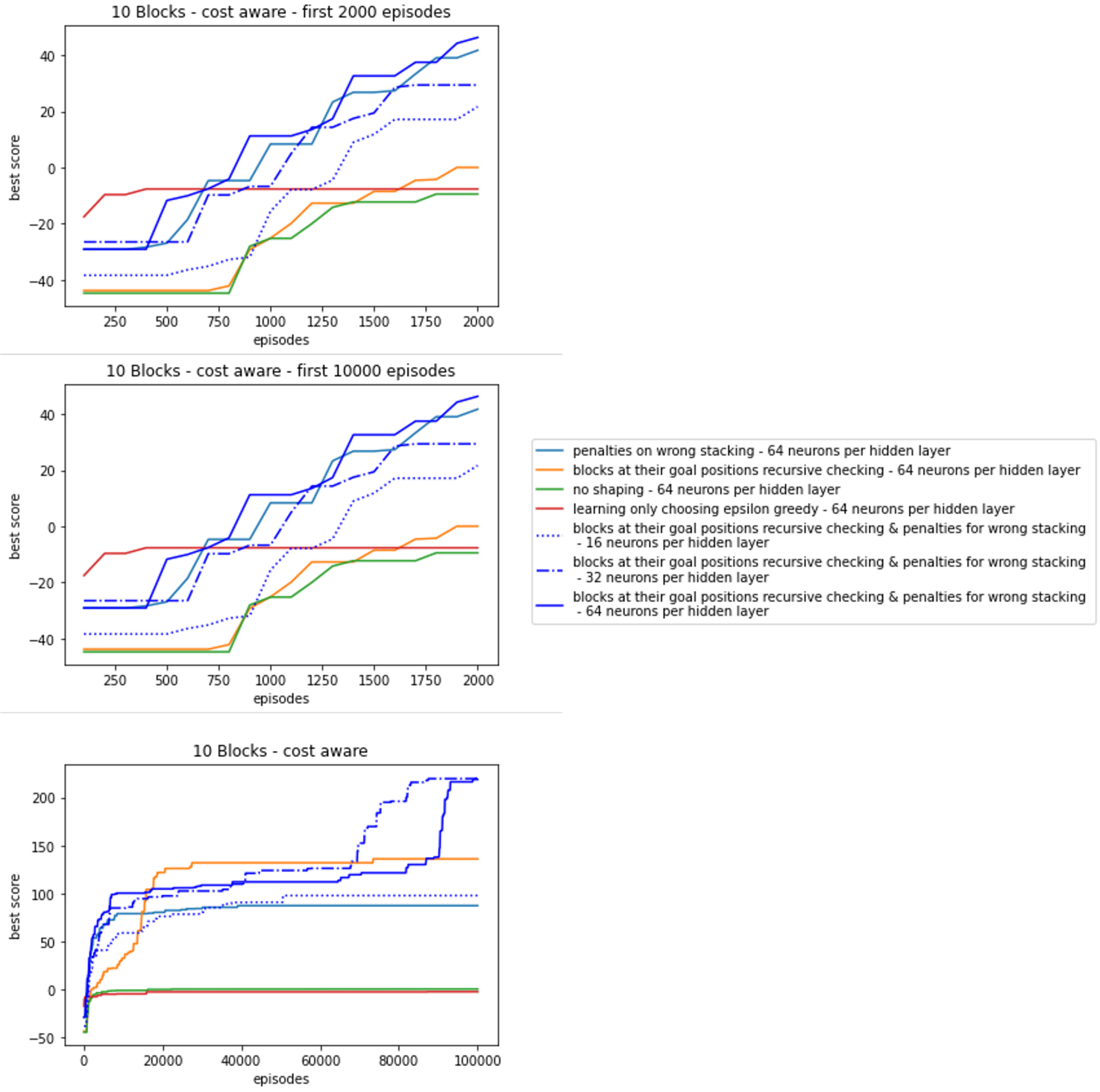
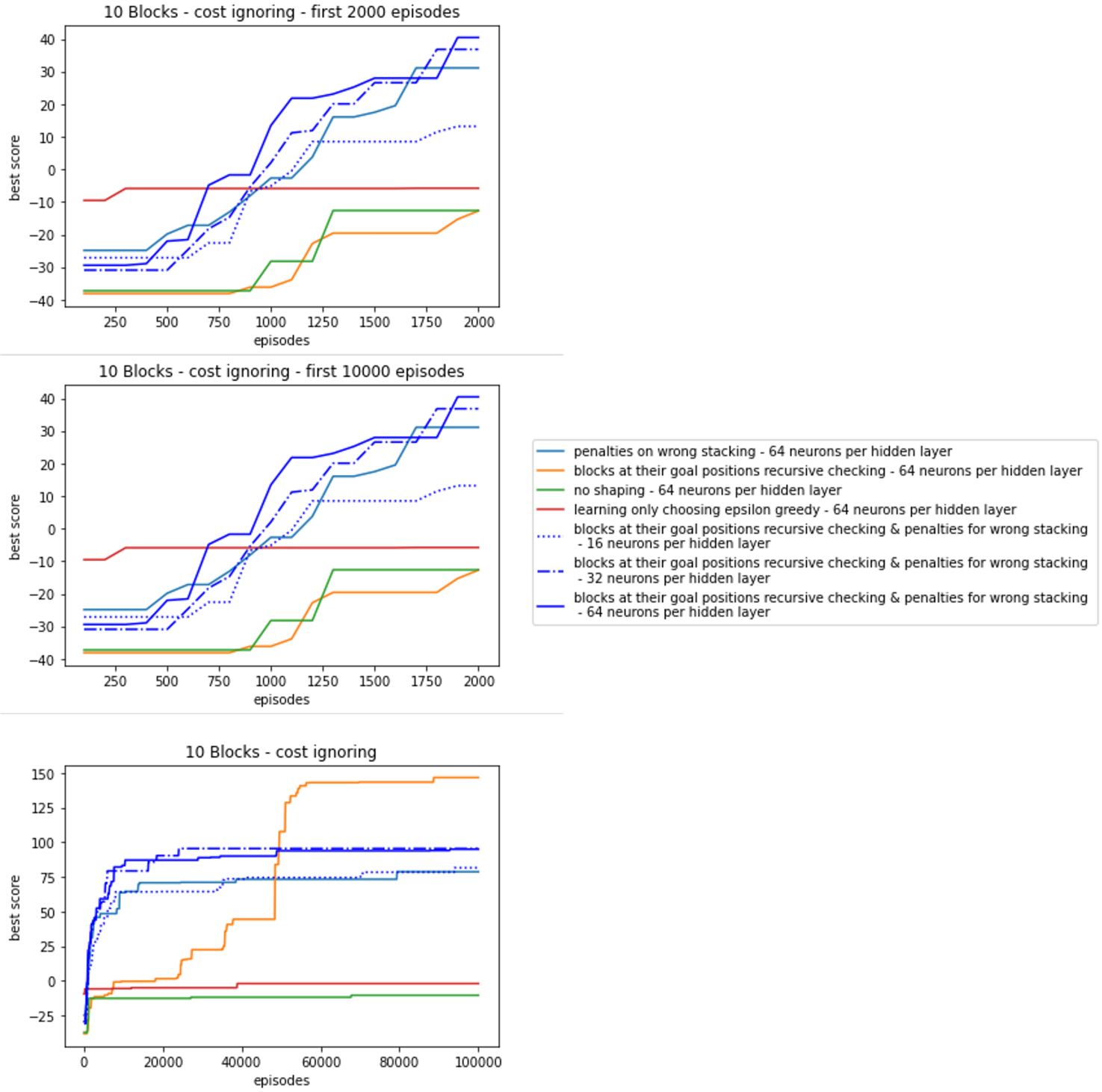


Figure 4.4: Learning Curves Blocks World 10 Blocks Cost Ignoring



4.5 Safety of the Learned Policies

Diagrams giving more detailed information on the safety of the policies can be found in appendix A.

In n-puzzle, applying reward shaping does not lead to a significant advantage with regard to safety. Both the stalling and avoid states are reached similarly often, independent of whether reward shaping is applied or not. The only noticeable fact is, that when learning networks always choose actions epsilon greedily, the policy ends up in dead ends almost all the time (Figures A.1 – A.6).

With regard to transport, the number of reached avoid and stalling states is very little as well. The networks previously learned by Vinzent et al. [16] encounter an increase in dead ends, while the approach always acting epsilon greedy leads to several avoid states reached. As in n-puzzle, the evaluation of the policies learned choosing actions epsilon greedy has an increased number of stalling states reached. Besides the two approaches mentioned before, all policies are mostly able to circumvent both avoid and stalling states (Figures A.7 – A.10).

Throughout Blocks World, the number of reached stalling states when evaluating the policies learned by only choosing actions epsilon greedy is very high as well. Furthermore, no learned policy reached many avoid states, independent of the number of blocks and the evaluation set. Besides very few exceptions, the number of avoid states is 0. The only very noticeable exception from this fact is the evaluation of cost-ignoring neural networks on 4 blocks. Here, the evaluation where the policy is ignored and the network always chooses actions epsilon greedily reaches between 6 and 10 avoid states in the 72 states evaluated in total (Figure A.20).

We now look at the evaluation on the set used for **training**, so the set of start states defined by Vinzent et al. [16]. In Blocks World instances with 4 and 6 blocks, most shaping functions enable the policies to avoid stalling states almost entirely. Only the shaping function punishing actions leading to blocks being stacked in the wrong order has a policy as a consequence, which evaluated encounters some stalling states in the cost ignoring instance with 6 blocks (Figures A.11 – A.14). On the cost-aware training set with 8 blocks, applying reward shaping lets the policies reduce the number of stalling states by about 80 percent compared to the networks given by Vinzent et al. [16]. On the same set ignoring the cost, several networks learned using reward shaping encounter the same number of dead ends as the networks from Vinzent et al. [16] (487 – 1347 reached stalling states in 4139 start states evaluated). Still, there are shaping functions that produce neural networks with the below 200 stalling states reached in evaluation, for example, Φ_{b_2}. Similar to the cost-aware set with 8 blocks, on the set with 10 blocks applying reward shaping during the learning of policies leads to fewer stalling states reached than not applying reward shaping. On the cost-ignoring set, only the network with 64 neurons per hidden layer learned under Φ_{b_2} is able to reach fewer stalling states than the networks trained by Vinzent et al. [16]. The other policies end up in stalling states in about 25 to 30 percent of the evaluated states (Figures A.15 – A.18).

Looking at the evaluation on the separately generated **test** set for block world, we get the following results with regard to reached stalling states:

For 4 blocks during the evaluation of both the cost aware as well as the cost ignoring evaluation, the networks by Vinzent et al. [16] do not reach any of those states. All other

policies evaluated are not able to achieve this (Figures A.19, A.20). For 6 blocks the networks from Vinzent et al. [16] do not achieve this anymore, but they still reach fewer such states than the other policies during evaluation (Figures A.21, A.22). The same holds for the evaluation on 8 and 10 blocks (Figures A.23 – A.26).

4.6 Performance of the Learned Policies – Measured in the Number of Reached Goal States

On n-puzzle, there are almost no goal states reached at all (Figures A.1 – A.6). So solving a n-puzzle using policies learned through the settings given appears to not be possible, independently of whether reward shaping is applied or not.

For the Transport task, the policies are at least able to reach some goal states. One can also see that the number of reached goal states is higher if reward shaping is applied than if it is not. Still, there are never more than 27 goal states reached by a policy in 1000/5000/10000 states the policies are evaluated on (Figures A.7 – A.10). From this, we can say the neural networks still perform poorly on Transport tasks if reward shaping is applied.

The only significant increase in performance of the neural networks learned can be seen in the Blocks World domain. We once again start by looking at the **training** set first.

For 4 blocks, most policies learned by applying shaping functions manage to perform very well and reach goal states in every evaluated start state. But also policies by Vinzent et al. [16] and those obtained not applying reward shaping manage to solve all 14 start states considered (Figures A.11, A.12). For 6 blocks we get similar results. The policies with the same approaches still perform well, although there is no policy able to solve all considered start states. We also see that there are more cost-aware networks able to perform this well than there are cost-ignoring ones (Figures A.13, A.14). Then, when we are looking at the Blocks World instances with 8 blocks, we see that besides one exception, neither the policies by Vinzent et al. [16] nor the policies obtained not applying reward shaping during learning are able to reach a large number of goal states. In opposite to this, there are still several neural networks trained through the usage of Φ_{b_1}, Φ_{b_2} and Φ_{b_4} that are able to get to a high percentage of goal states. We can therefore say that the application of reward shaping here leads to policies superior to those obtained without reward shaping (Figures A.15, A.16). Similar results can be observed for 10 Blocks. Policies without reward shaping are not able to reach any goal states anymore while some of those learned through reward shaping still do. From those ignoring costs, only the neural network learned using Φ_{b_2} is able to perform well and reach a high percentage of goal states (Figures A.17, A.18).

We now look at the evaluation of the policies on the newly generated **test** set of start states.

On the smallest Blocks World instance, the neural networks by Vinzent et al. [16] are able to solve all start states. From the newly learned networks, there are none able to make it to half of the goal states of this set (Figures A.19, A.20). For 6 blocks, some of the networks trained by Vinzent et al. [16] are still able to perform very well, while all other networks perform poorly (Figures A.21, A.22). On all other test sets, all networks are not able to reach a significant amount of goal states (Figures A.23 – A.26).

Chapter 5

Conclusion

This Chapter begins with a summary of the work done in this bachelor thesis. Afterward, an outline of what additional topics may be looked into in this area of research is given.

Our work on reinforcement learning with reward shaping starts with constructing potential-based reward shaping functions for the planning domains we consider. We then learn policies applying these reward shaping functions in learning for different configurations. In addition to comparing the learned policies to each other, we also take other policies into account, obtained without using reward shaping.

With regard to the results from the experiments from Chapter 4, we state that the performance of manual reward shaping depends on the planning domain as well as on the start state set the deep neural networks are evaluated on. Although the main benchmarks we are interested in are the start states compactly described by Vinzent et al [16], the following can be stated concerning the separate test sets. On every domain, when looking at the test states, so those which are not used for learning, the performance of the shaping function is bad. Also on the domains Transport and n-puzzle, manual reward shaping does not lead to a significant improvement over not applying manual reward shaping. In Transport, this bad performance probably originated from the size of the state space combined with the fact that the manual shaping functions provide too little and too imprecise guidance. Although some shaping functions give a reward in every state of a n-puzzle instance, the complexity of solutions of such instances predominates the guidance the shaping functions provide.

With regard to Blocks World and the domain instances from Vinzent et al. [16], the neural networks learned by applying the shaping functions from chapter 3 perform significantly better than the neural networks retrieved without reward shaping on the instances with 8 and 10 blocks. Since the policies by Vinzent et al. [16] already perform very well on Blocks World instances with 4 and 6, applying reward shaping there does not lead to an improvement. Still, the policies learned through reward shaping perform on a similar level to the ones by Vinzent et al. [16] do.

5.1 Future Work

As a consequence of the progress made in the research area of reinforcement learning recently, this technology may hold the potential to bring forth good policies on planning domains.

Due to the fixed set of start states used for learning and evaluation together with the episode and time limit, the results may improve when changing such parameters. Furthermore, there are more planning domains, such as Rovers, Visit All, or Sokoban, where Manual Reward Shaping might be applied. For the instances taken into account in this work, extending the encoding by additional parameters might make it possible to create more complex shaping functions in order to obtain better learning results.

Besides only applying manual reward shaping, there are other possibilities for improving the learned networks. One of them is the possibility of guiding epsilon greediness during the learning process. This means every time the agent chooses an action epsilon greedy during the learning progress, choosing this action according to certain rules instead of epsilon greedy. For the guidance of epsilon greediness, policy greediness might be an option. This means we manually define a (maybe incomplete) policy for solving a task (which for for instance Blocks World or Transport is relatively easy for humans) and then every time the agent chooses an action epsilon greedy, the action is chosen according to this manually defined policy. As another option for guiding epsilon greediness, one can adapt heuristics from classical planning to the task. As a result, every time the agent chooses its action epsilon greedy, the action with the lowest or best heuristic value is selected. Generally, it may be possible to use other reinforcement learning techniques, such as entropy regularization [9], count-based exploration bonuses [3] and prediction based exploration bonuses [1] [5].

Instead of using heuristics only for the guidance of epsilon greediness, they might also be applicable in potential-based reward shaping. Heuristics estimate the distance from a state to some goal state. By using the estimated distance as a negative potential in reward shaping, it might be possible to guide the agent toward learning good policies. Using heuristics particularly yields the advantage that each state would have some potential, so the agent would get feedback and learn from it for every action performed. This risk of misguiding the learning progress through bad estimation exists, but since heuristic search yielded good results in classical planning, for example, the Fast-Forward Planning System [10] using delete relaxations, there should be heuristics leading to good results. Heuristics also have the advantage of being applicable independent of the task, so learning neural networks in other domains should be possible with little effort.

Bibliography

[1] Joshua Achiam and Shankar Sastry. Surprise-based intrinsic motivation for deep reinforcement learning. *CoRR*, abs/1703.01732, 2017.

[2] W.W. ROUSE BALL and H.S.M. COXETER. *MISCELLANEOUS PROBLEMS*, pages 312–337. University of Toronto Press, 1974.

[3] Marc G. Bellemare, Sriram Srinivasan, Georg Ostrovski, Tom Schaul, David Saxton, and Rémi Munos. Unifying count-based exploration and intrinsic motivation. *CoRR*, abs/1606.01868, 2016.

[4] Carlos E. Budde, Christian Dehnert, Ernst Moritz Hahn, Arnd Hartmanns, Sebastian Junges, and Andrea Turrini. Jani: Quantitative model and tool interaction. In Axel Legay and Tiziana Margaria, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 151–168, Berlin, Heidelberg, 2017. Springer Berlin Heidelberg.

[5] Yuri Burda, Harrison Edwards, Amos J. Storkey, and Oleg Klimov. Exploration by random network distillation. *CoRR*, abs/1810.12894, 2018.

[6] Clement Gehring, Masataro Asai, Rohan Chitnis, Tom Silver, Leslie Pack Kaelbling, Shirin Sohrabi, and Michael Katz. Reinforcement learning for classical planning: Viewing heuristics as dense reward generators, 2021.

[7] Timo P. Gros, Holger Hermanns, Jörg Hoffmann, Michaela Klauck, Maximilian A. Köhl, and Verena Wolf. Artifact for the Tool Paper: MoGym: Using Formal Models for Training and Verifying Decision-making Agents. Zenodo, May 2022.

[8] Marek Grzes. Improving exploration in reinforcement learning through domain knowledge and parameter analysis. March 2010.

[9] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. *CoRR*, abs/1801.01290, 2018.

[10] Jörg Hoffmann and Bernhard Nebel. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research*, 14:253–302, 2001.

[11] Maximilian A. Köhl, Michaela Klauck, and Holger Hermanns. Momba: Jani meets python. In Jan Friso Groote and Kim Guldstrand Larsen, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 389–398, Cham, 2021. Springer International Publishing.

[12] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, February 2015.

[13] Andrew Y. Ng, Daishi Harada, and Stuart Russell. Policy invariance under reward transformations: Theory and application to reward shaping. In *In Proceedings of the Sixteenth International Conference on Machine Learning*, pages 278–287. Morgan Kaufmann, 1999.

[14] Martin Riedmiller. Neural fitted q iteration – first experiences with a data efficient neural reinforcement learning method. In *In 16th European Conference on Machine Learning*, pages 317–328. Springer, 2005.

[15] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, second edition, 2018.

[16] M. Vinzent, M. Steinmetz, and J. Hoffmann. Neural network action policy verification via predicate abstraction. 2022.

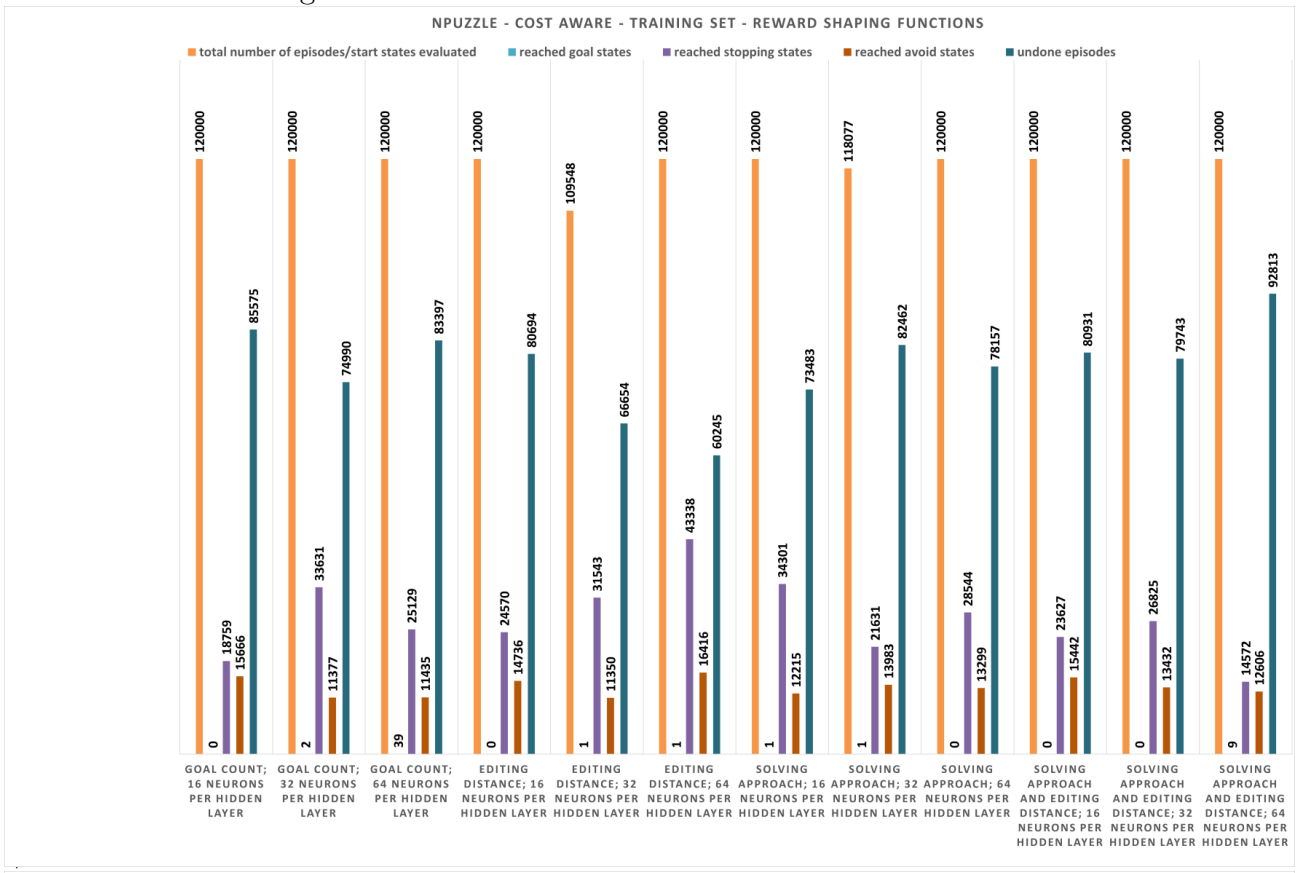
[17] M. Vinzent, M. Steinmetz, and J. Hoffmann. Neural network action policy verification via predicate abstraction technical report. 2022.

Appendix A

Total Results

The following section gives a complete overview of the safety and performance of the learned policies.

Figure A.1: Performance of the learned networks:



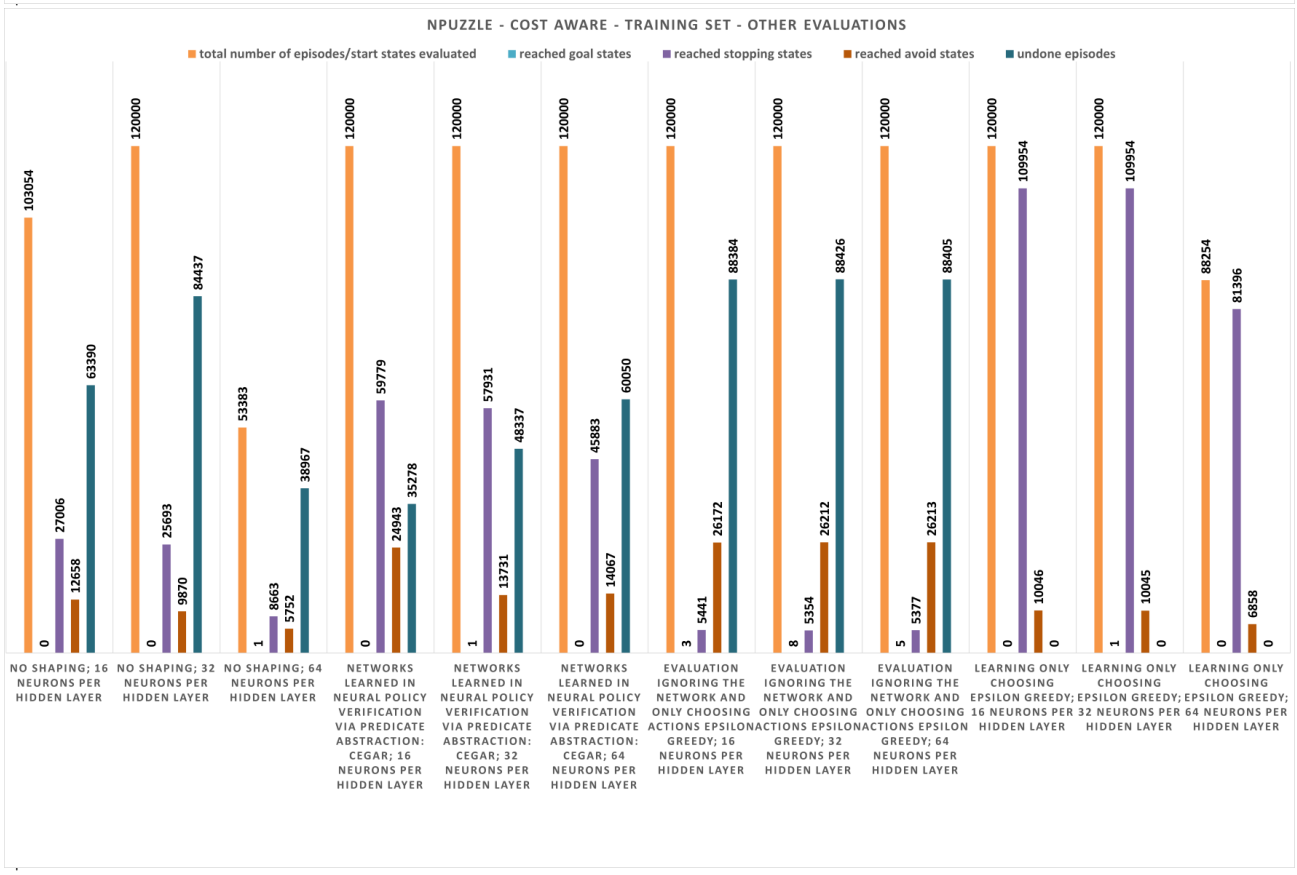
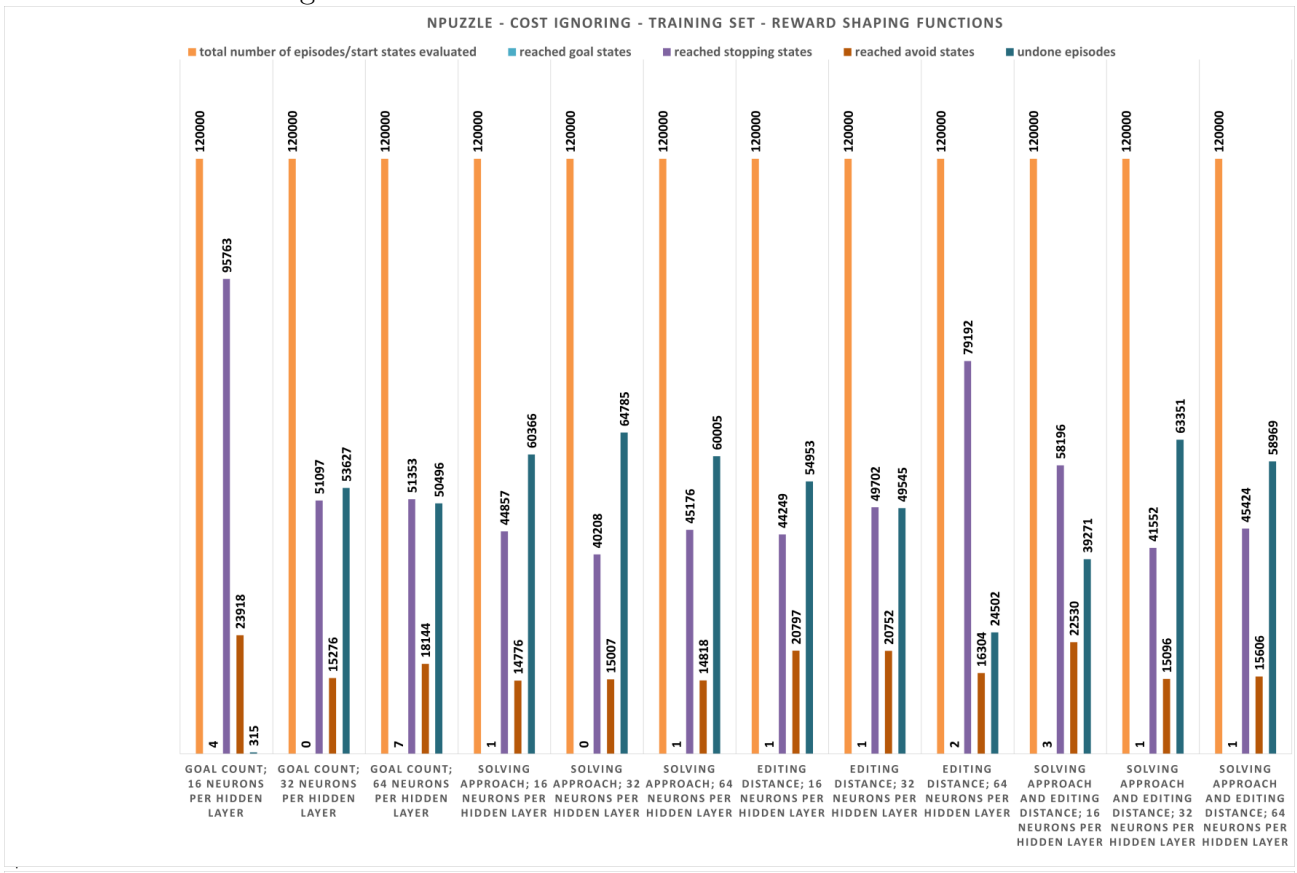


Figure A.2: Performance of the learned networks:



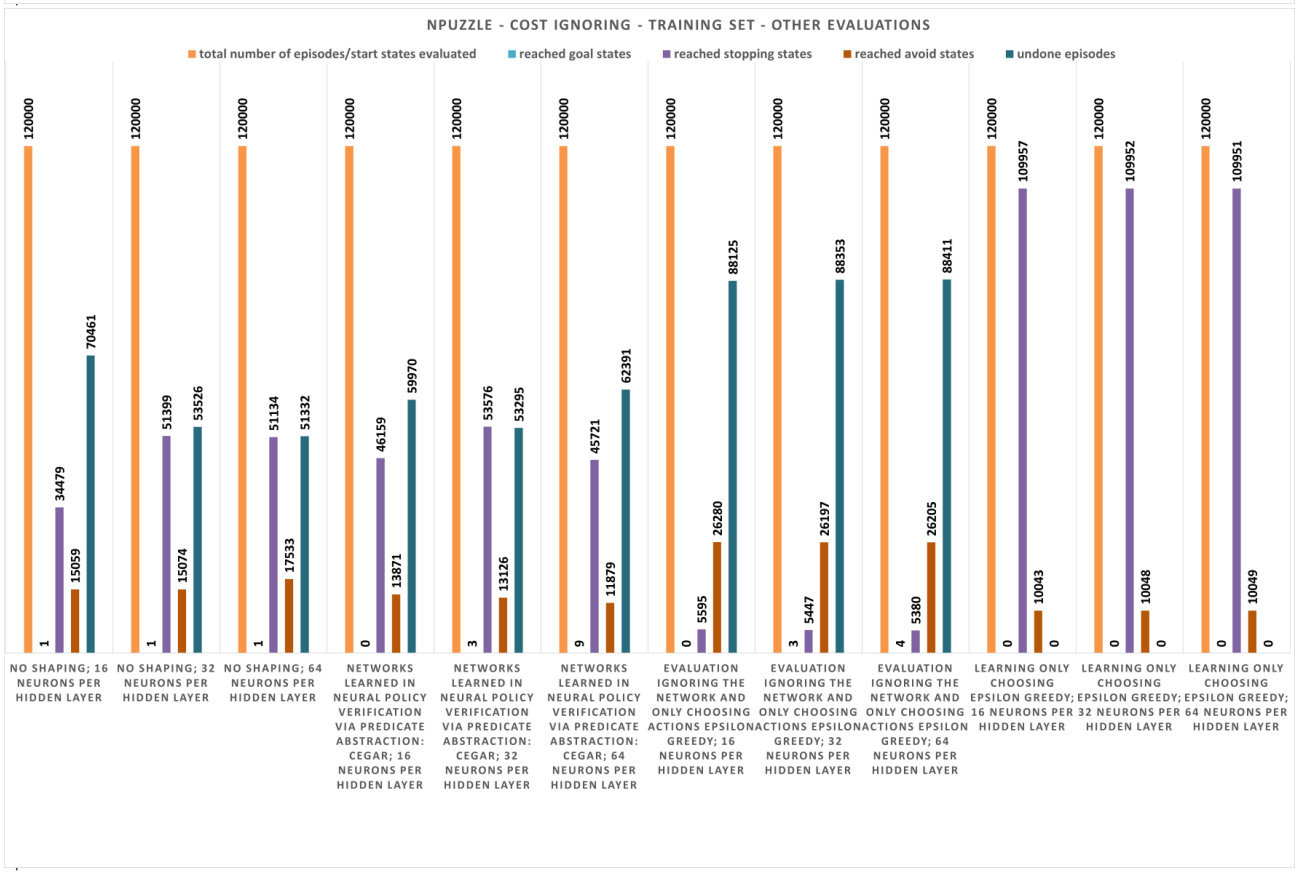
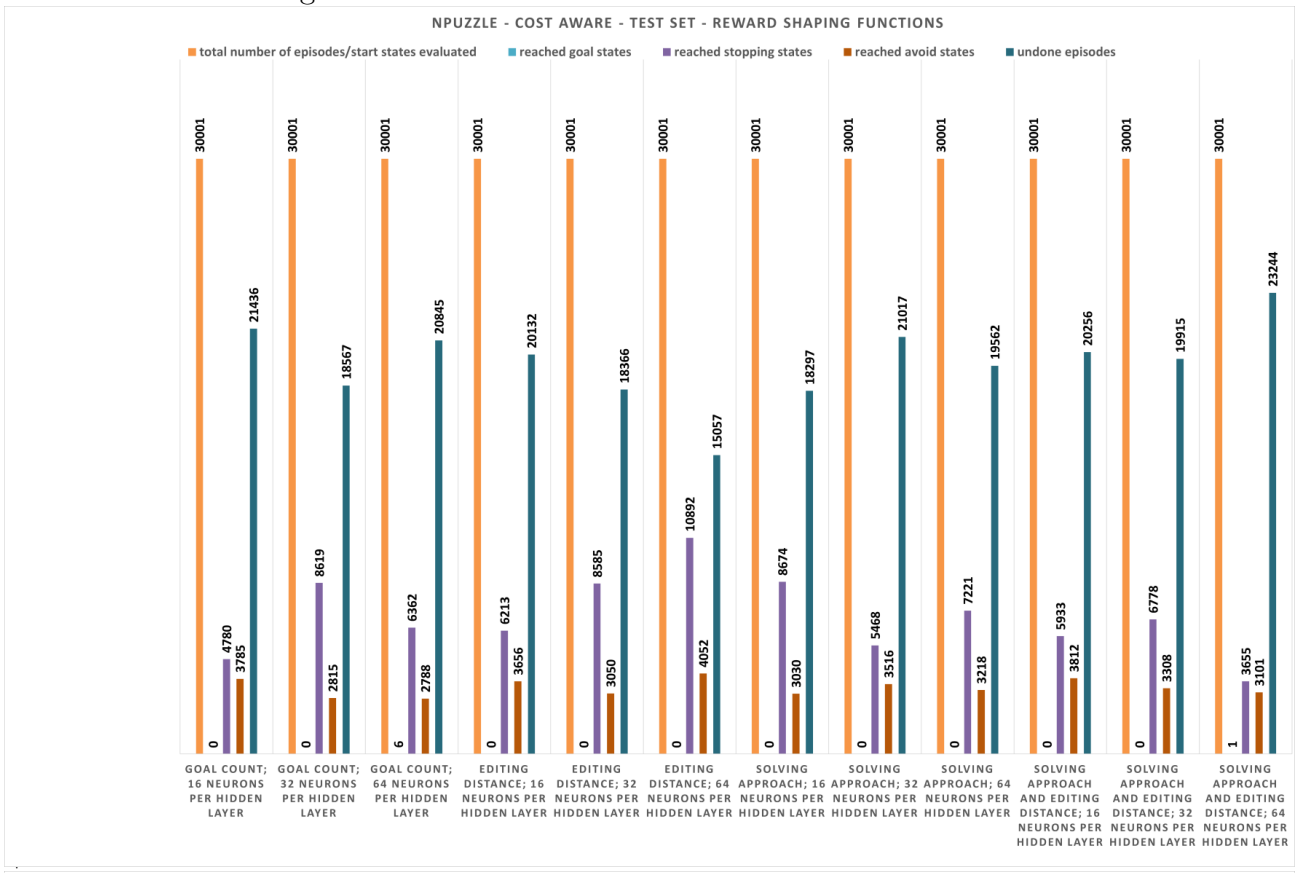


Figure A.3: Performance of the learned networks:



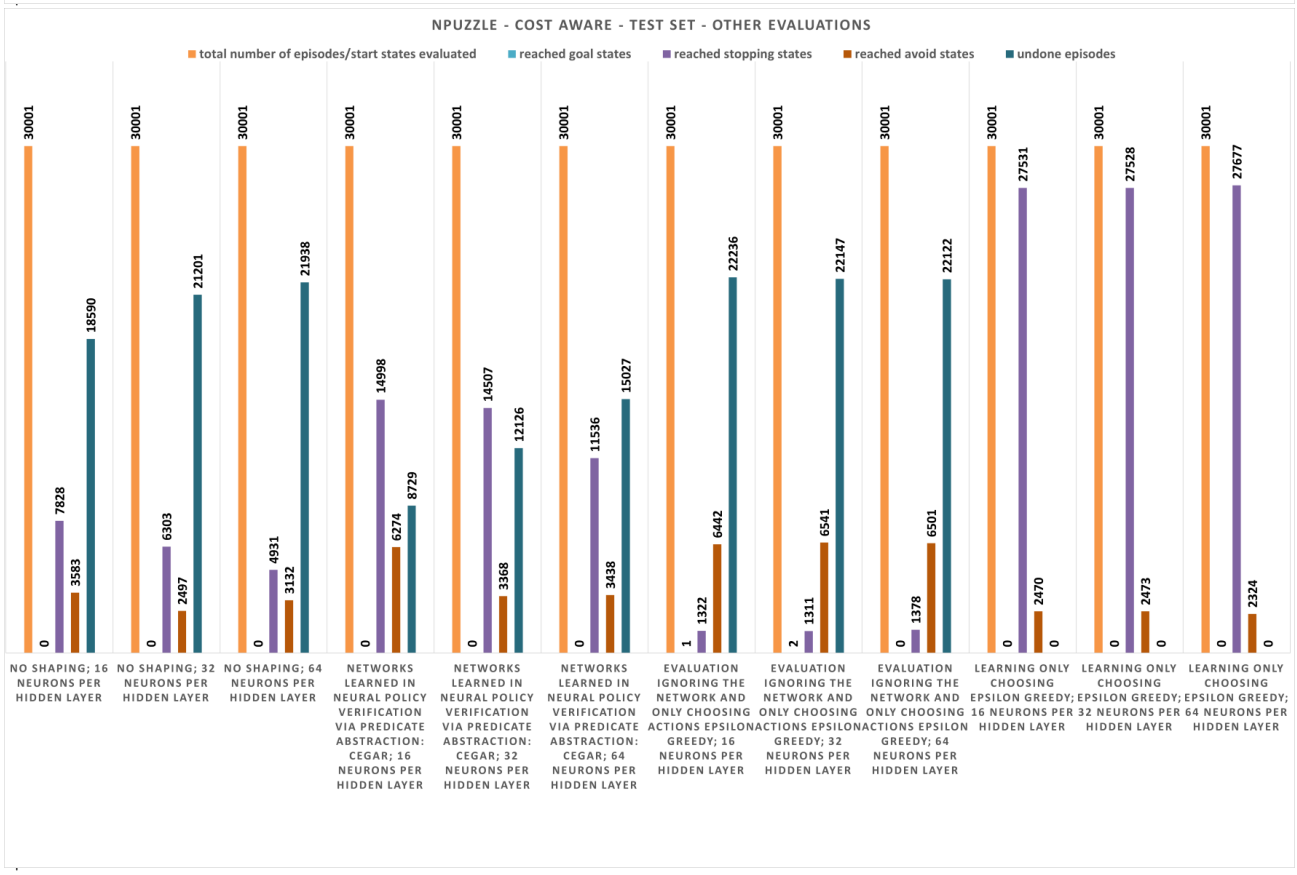
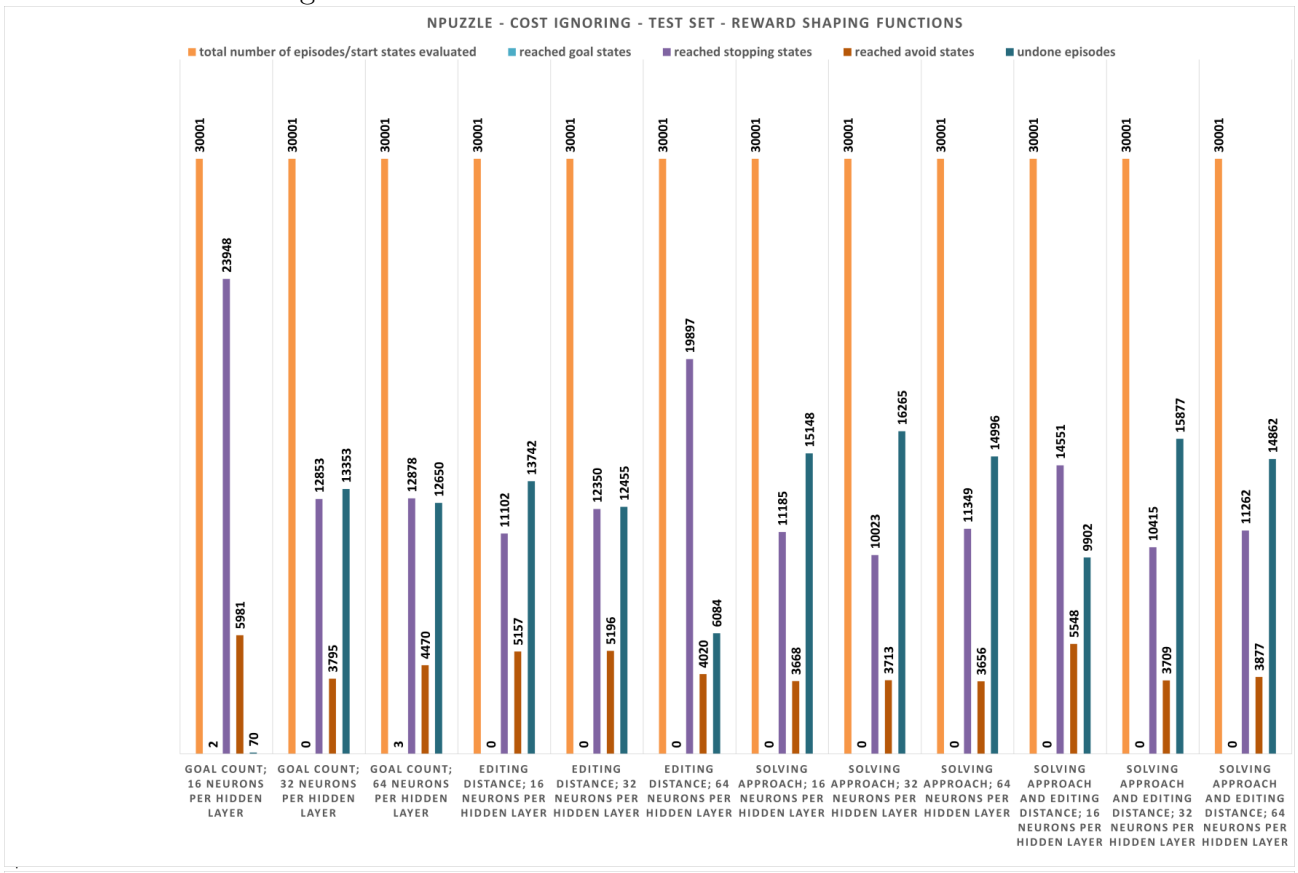


Figure A.4: Performance of the learned networks:



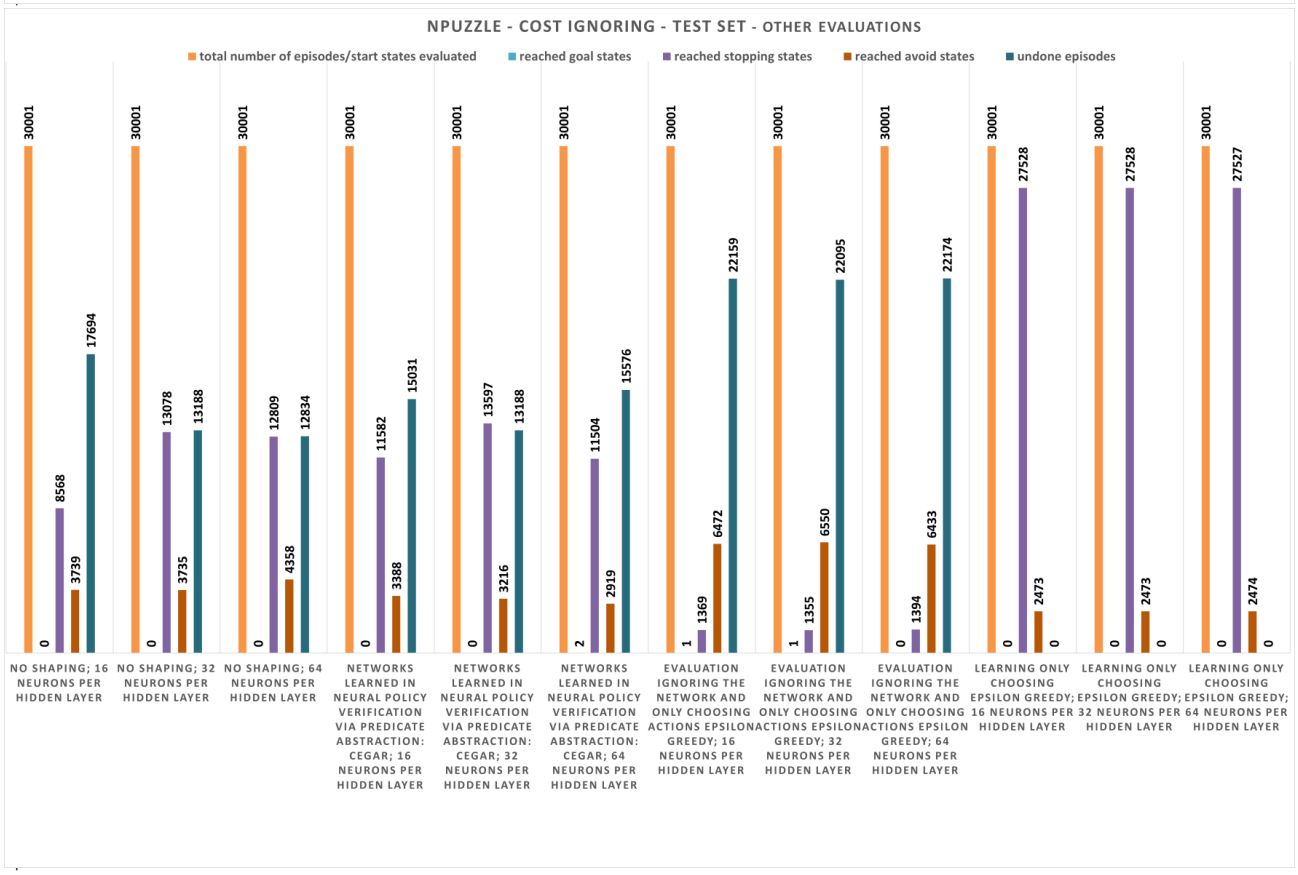


Figure A.5: Performance of the learned networks:

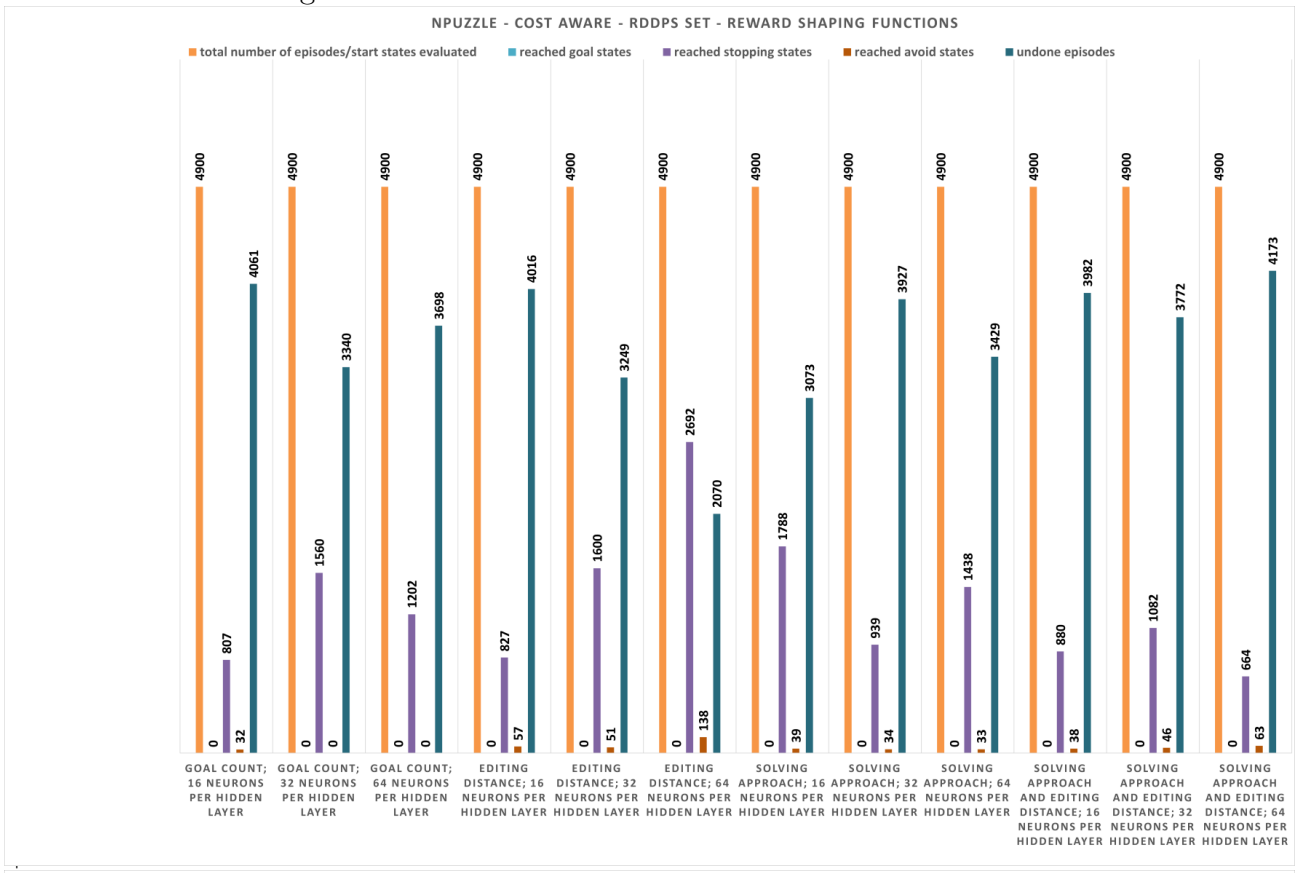
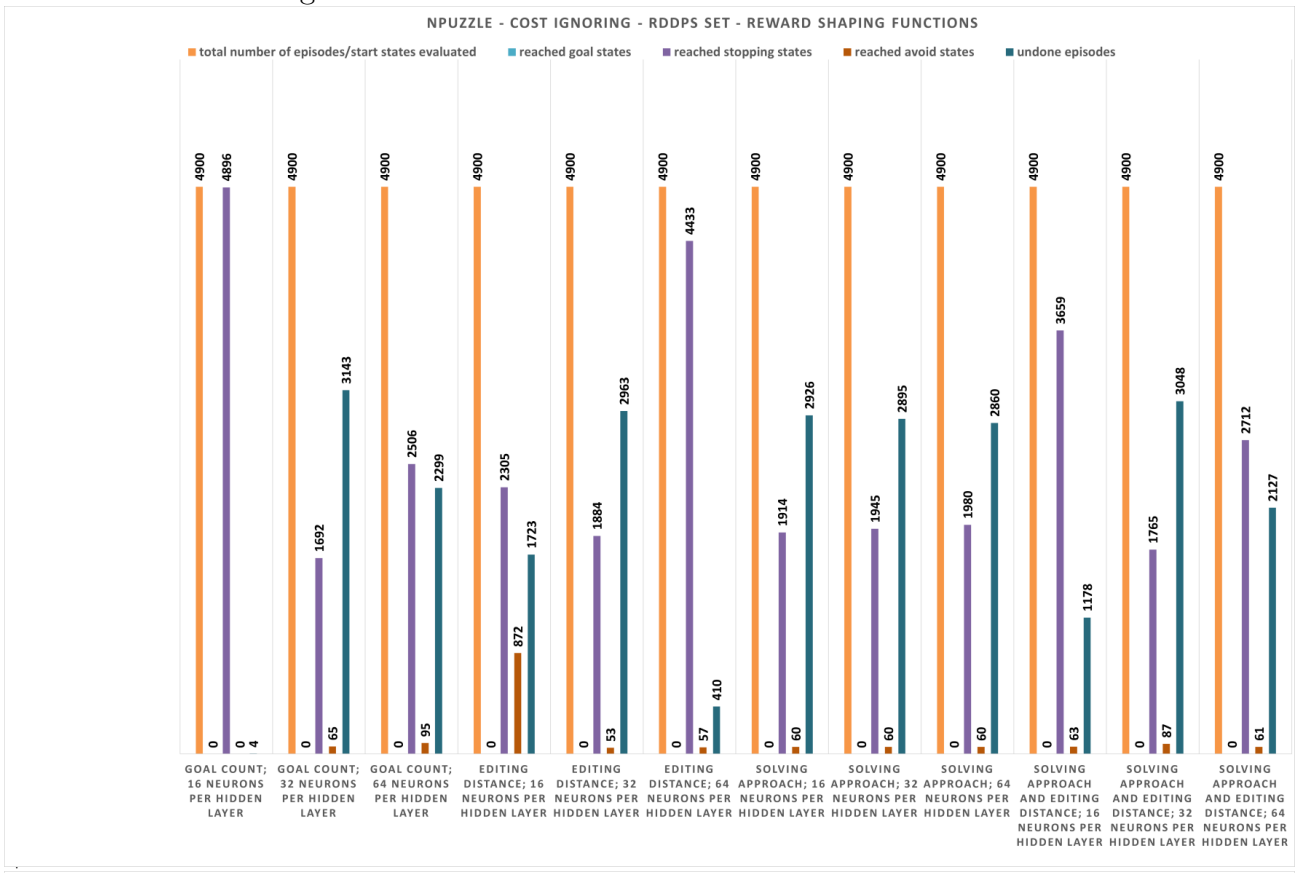


Figure A.6: Performance of the learned networks:



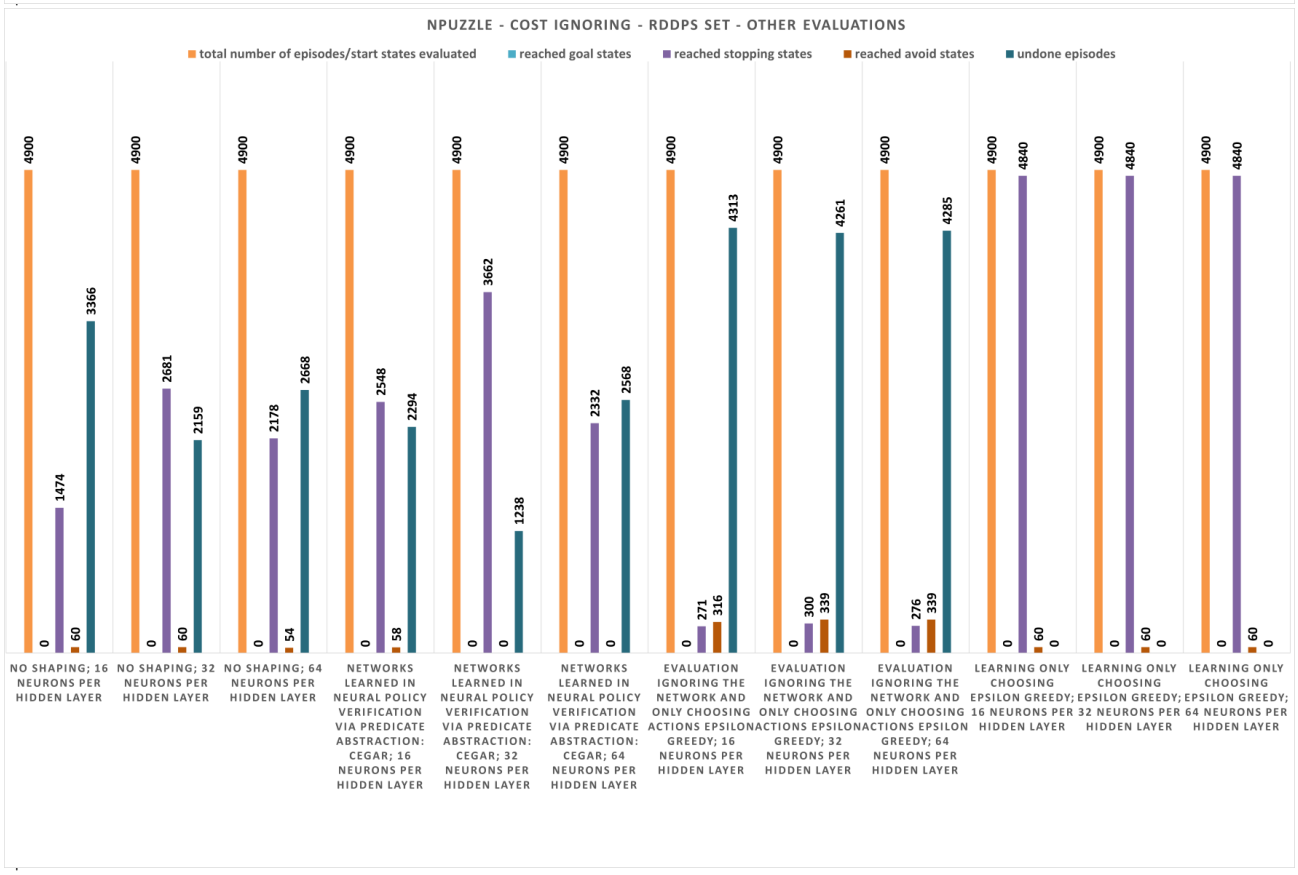
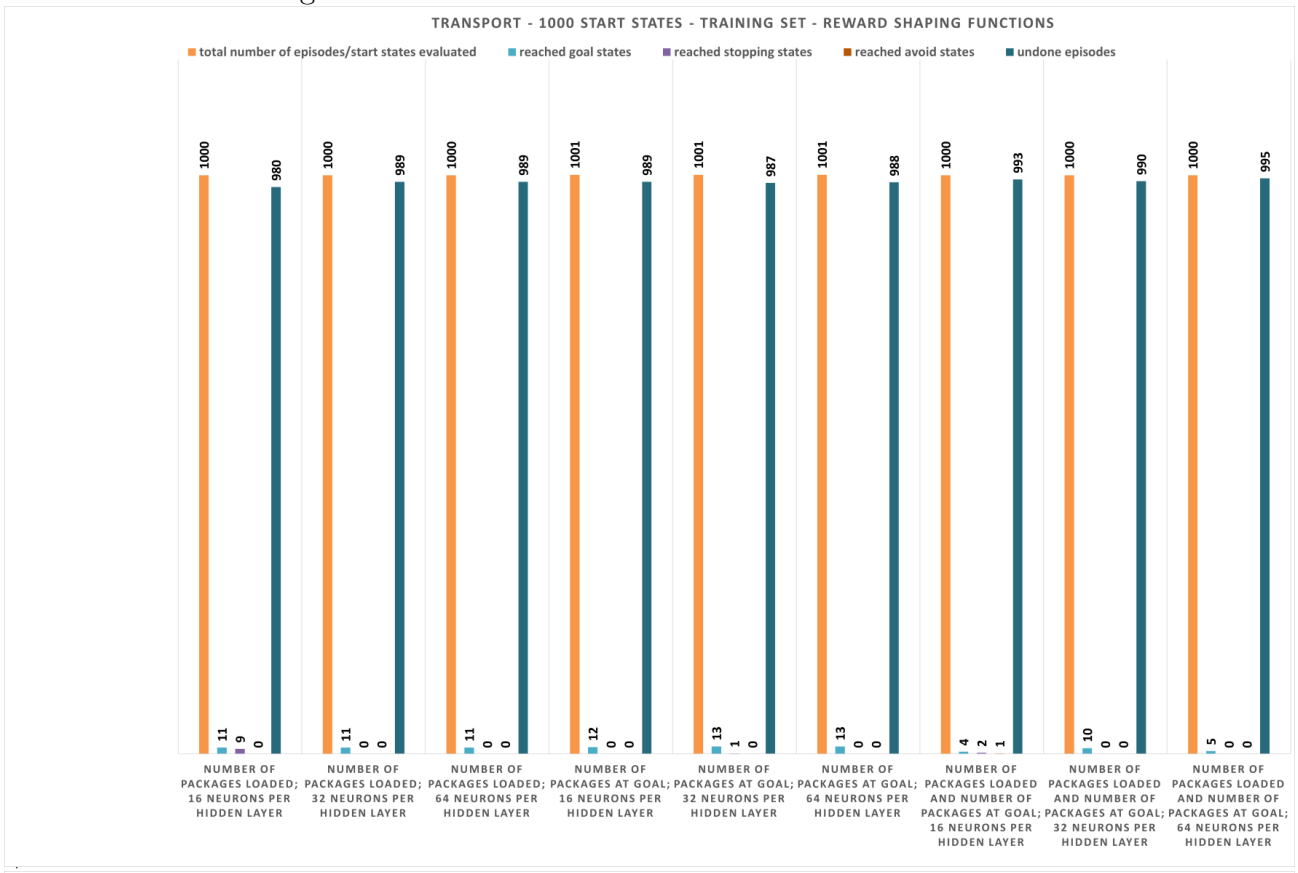


Figure A.7: Performance of the learned networks:



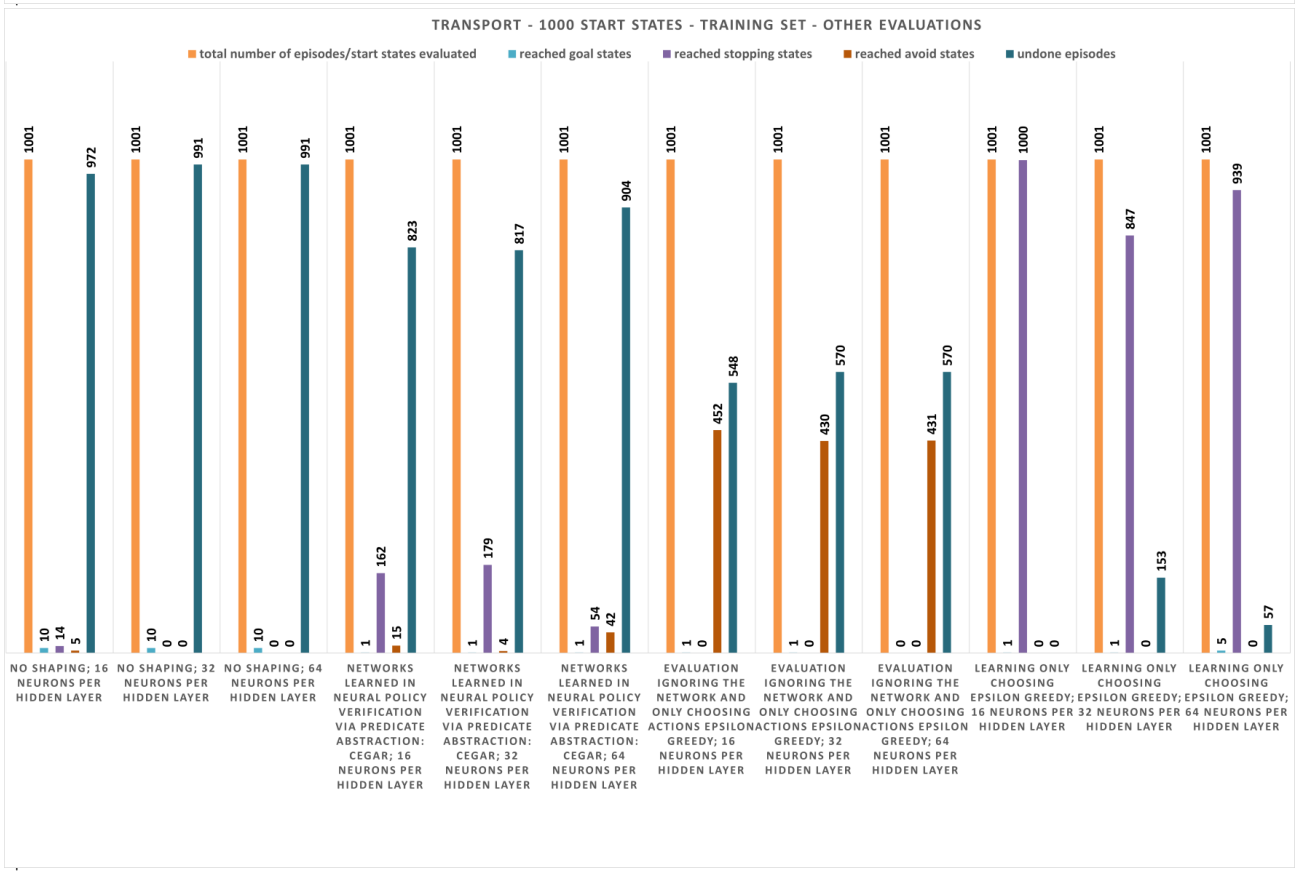
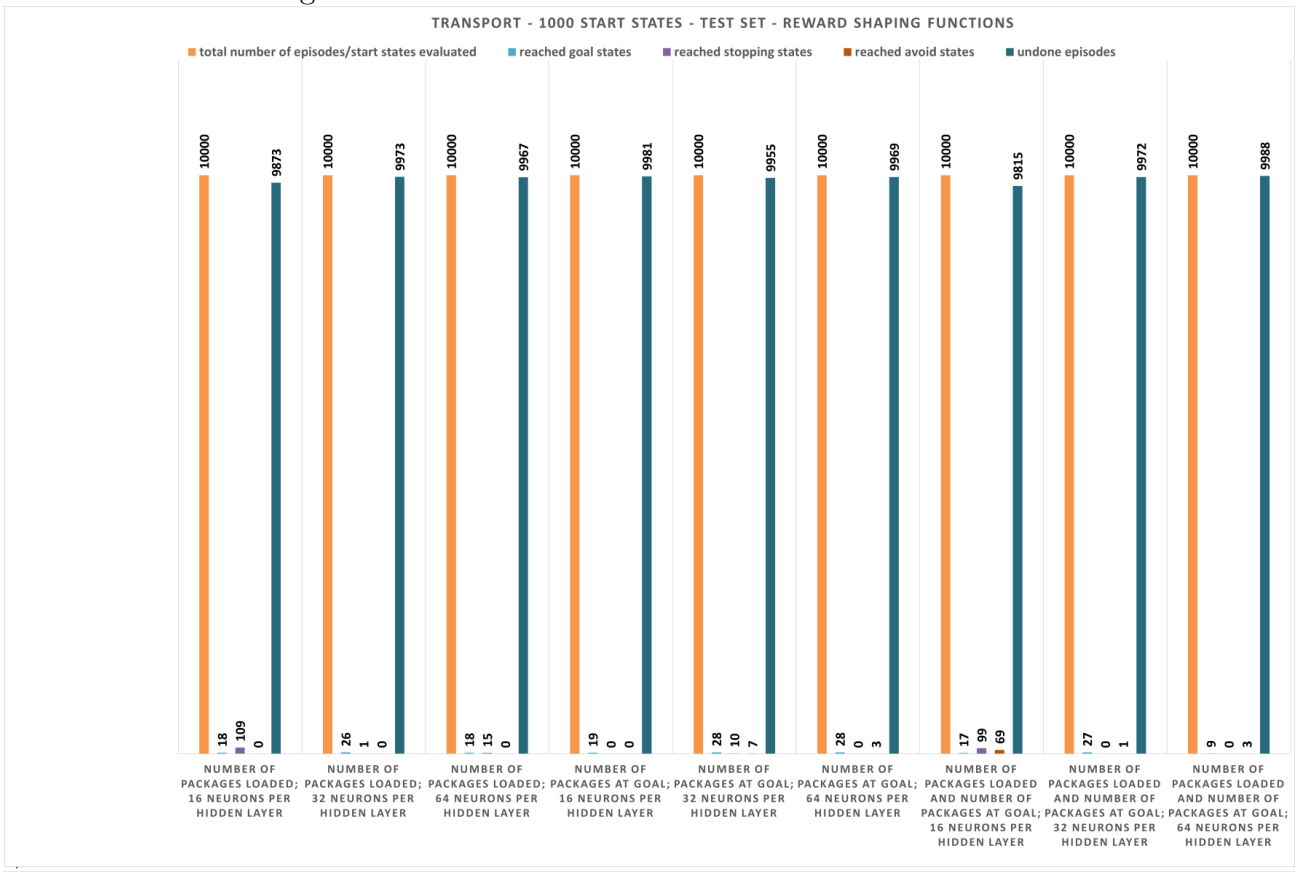


Figure A.8: Performance of the learned networks:



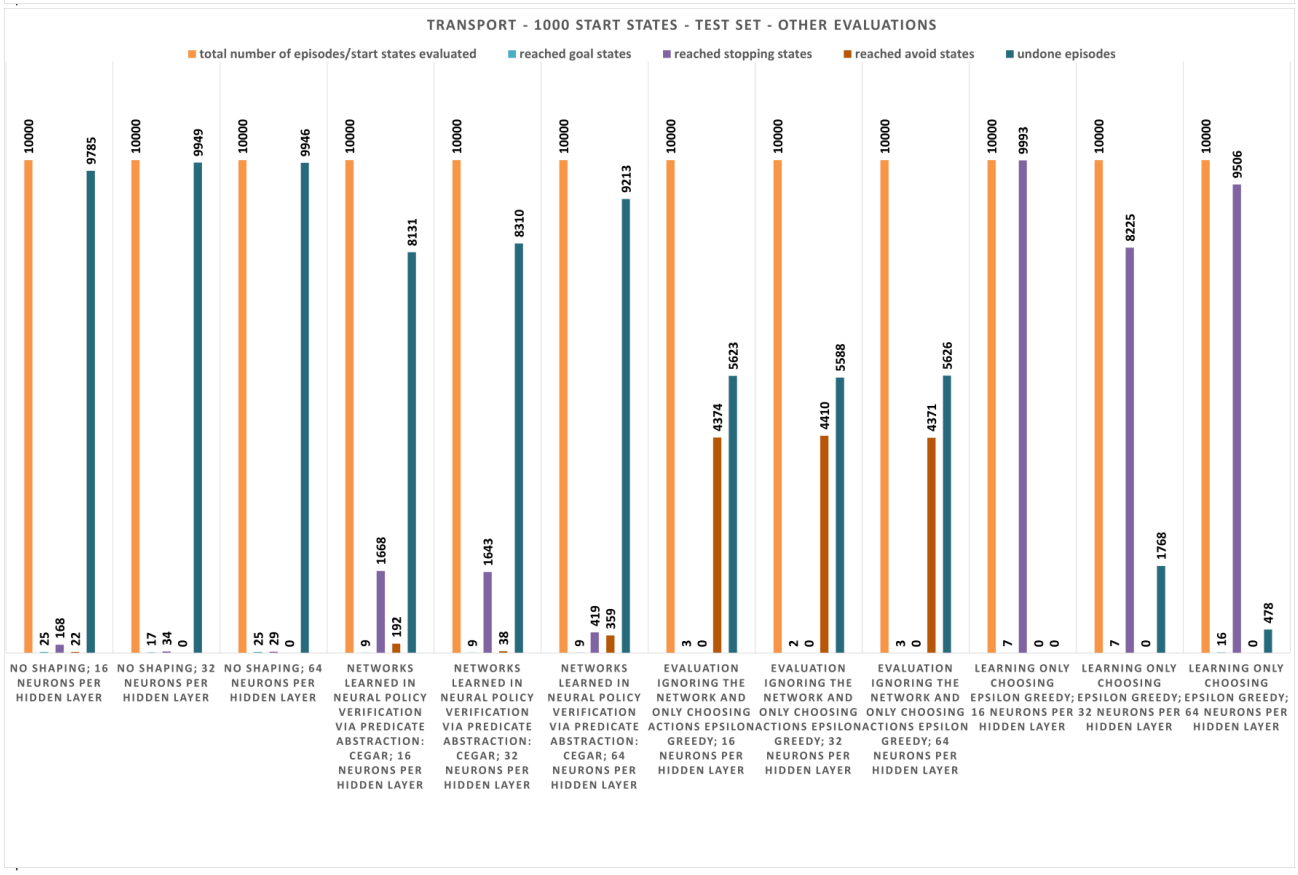


Figure A.9: Performance of the learned networks:

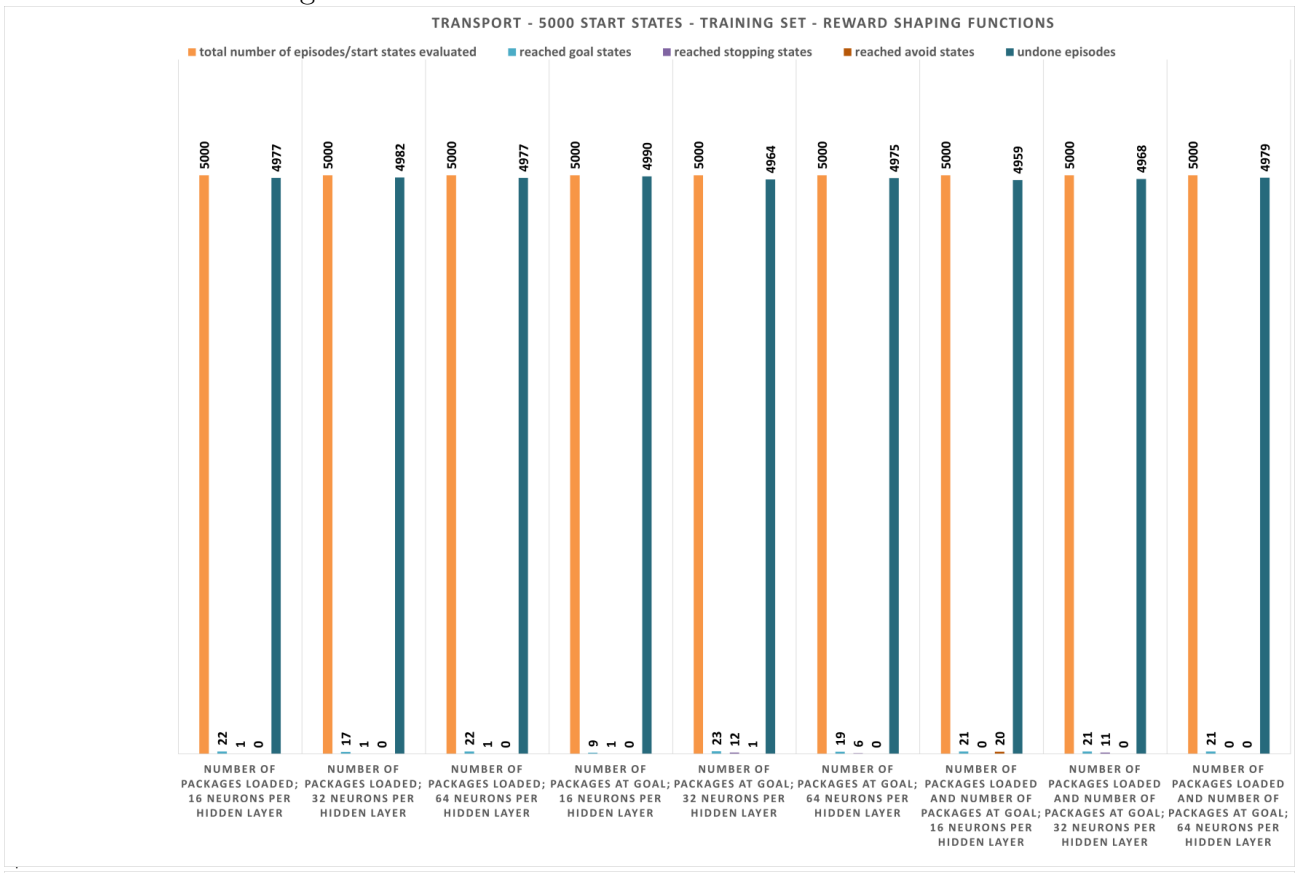


Figure A.10: Performance of the learned networks:

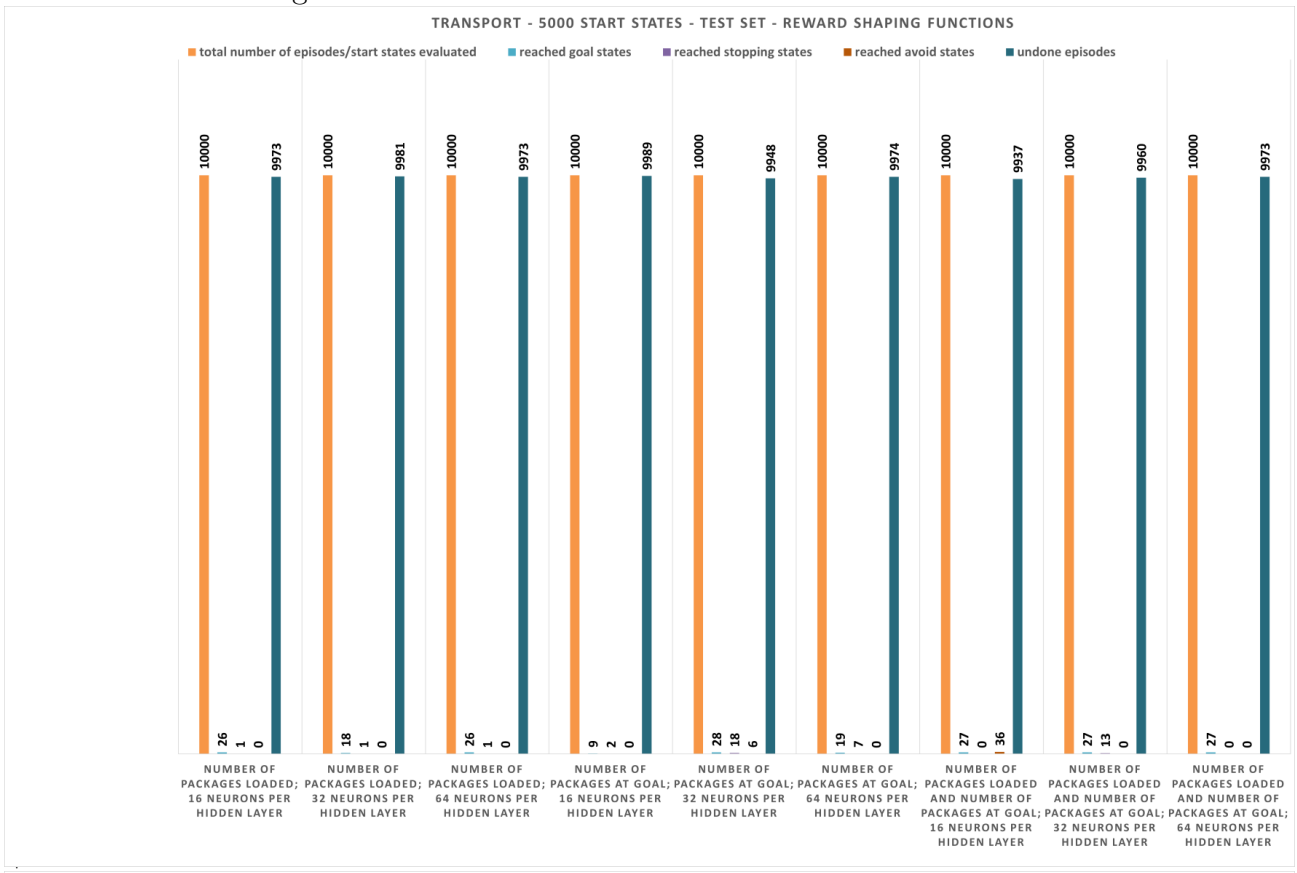
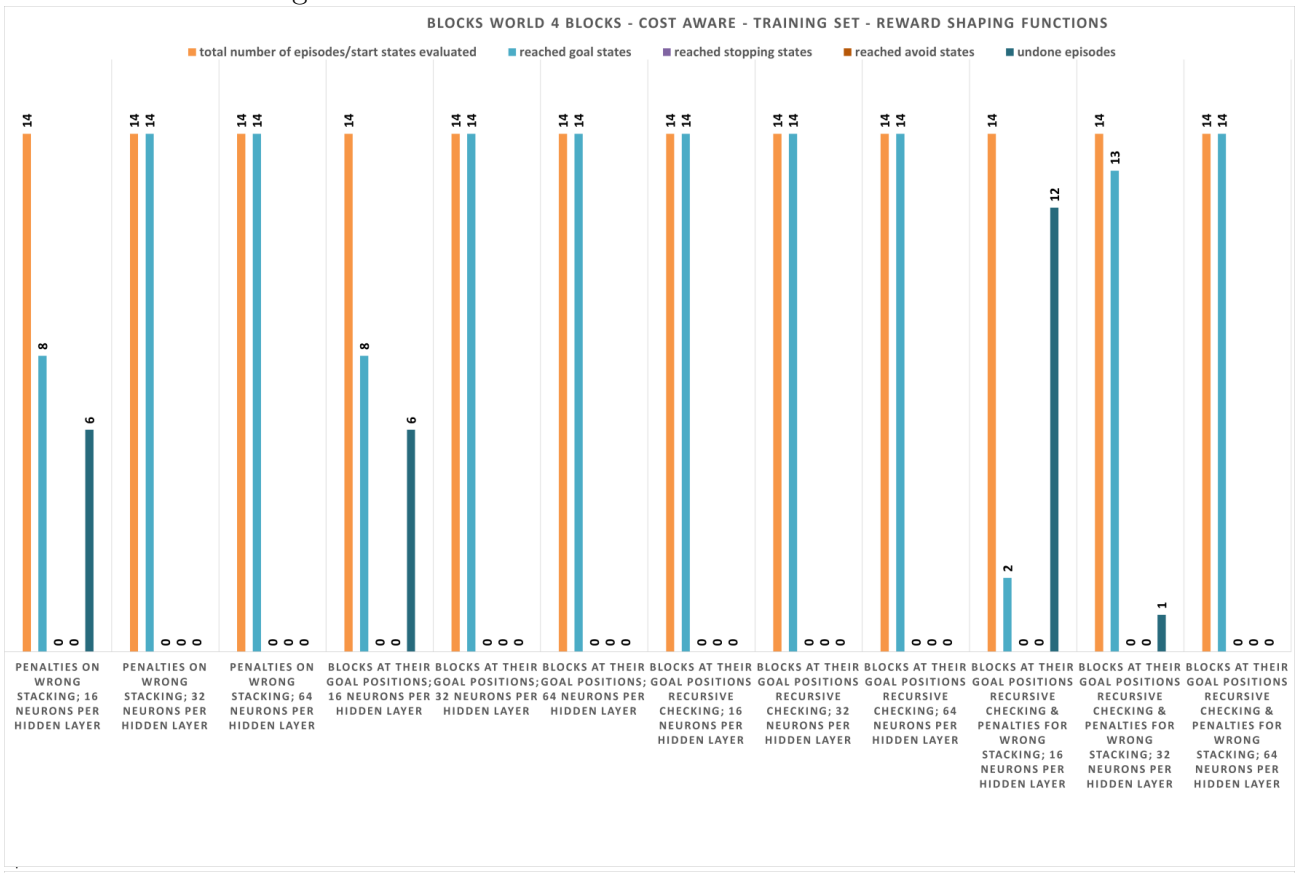


Figure A.11: Performance of the learned networks:



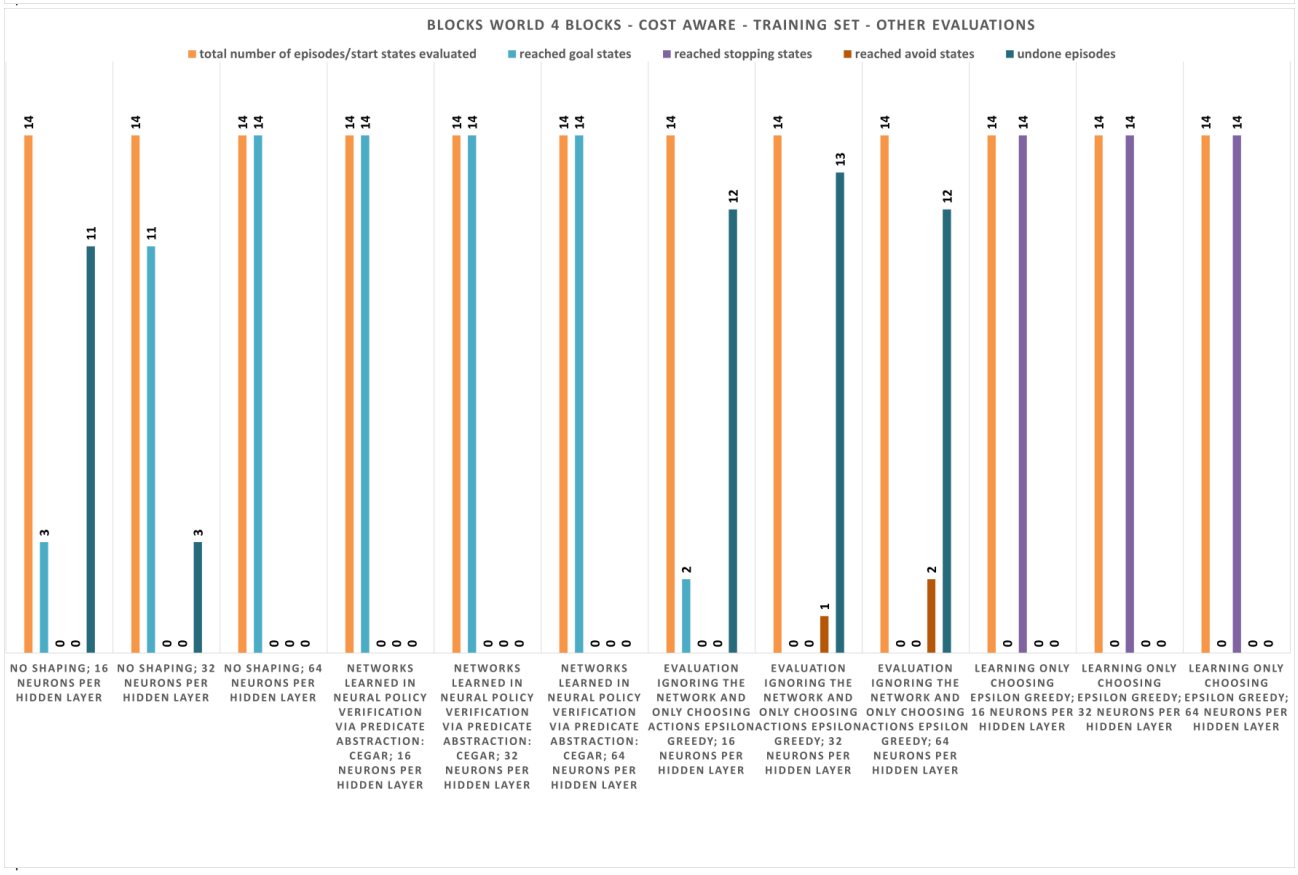
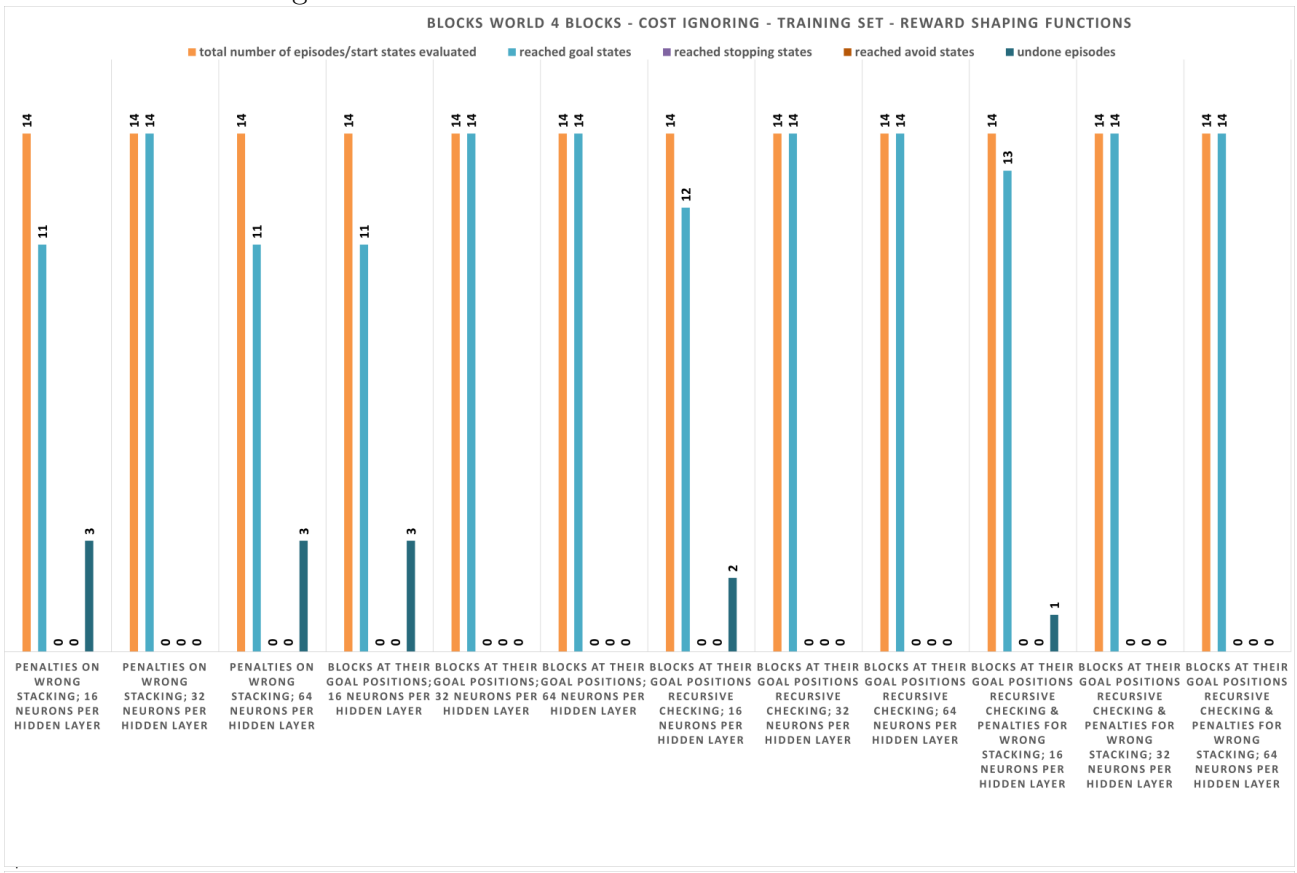


Figure A.12: Performance of the learned networks:



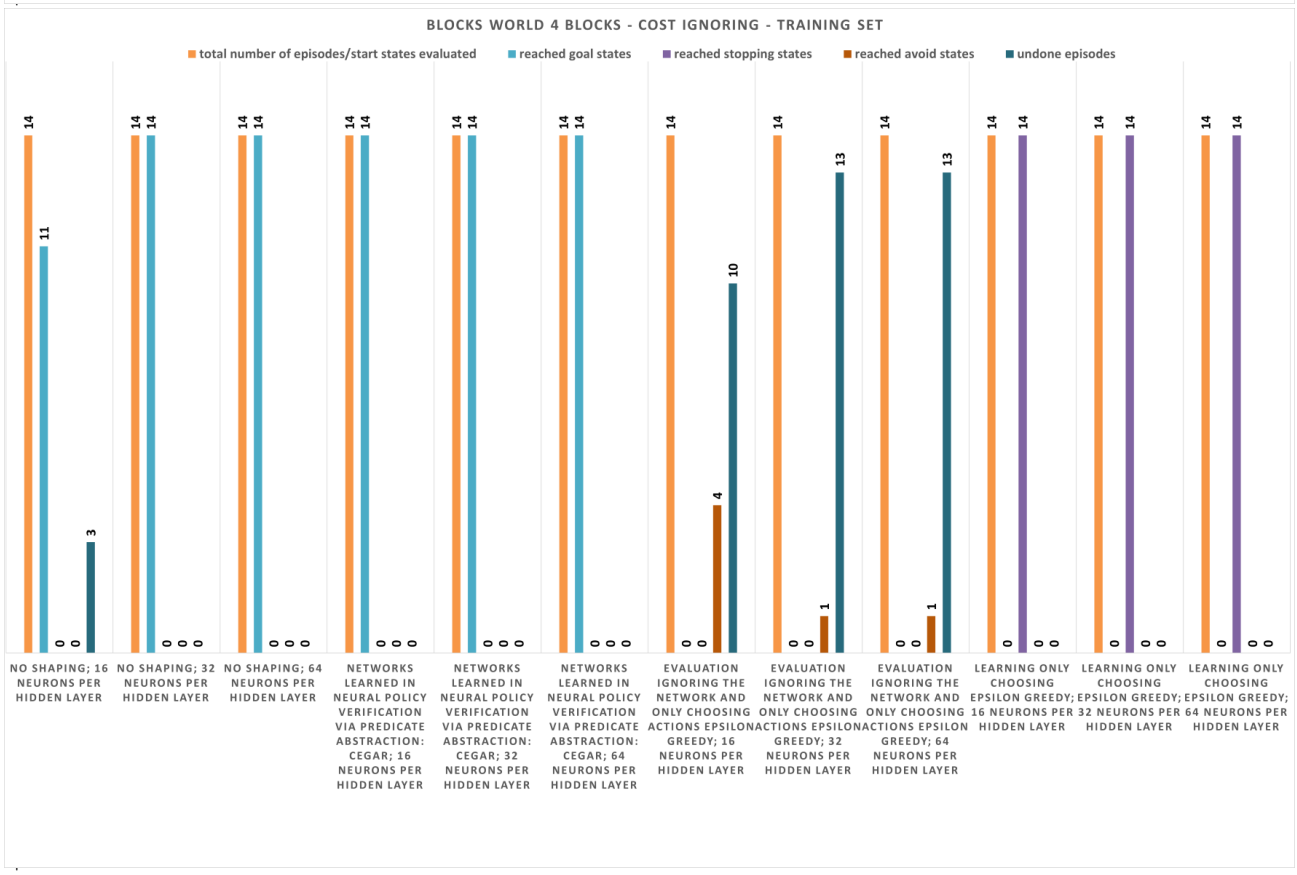
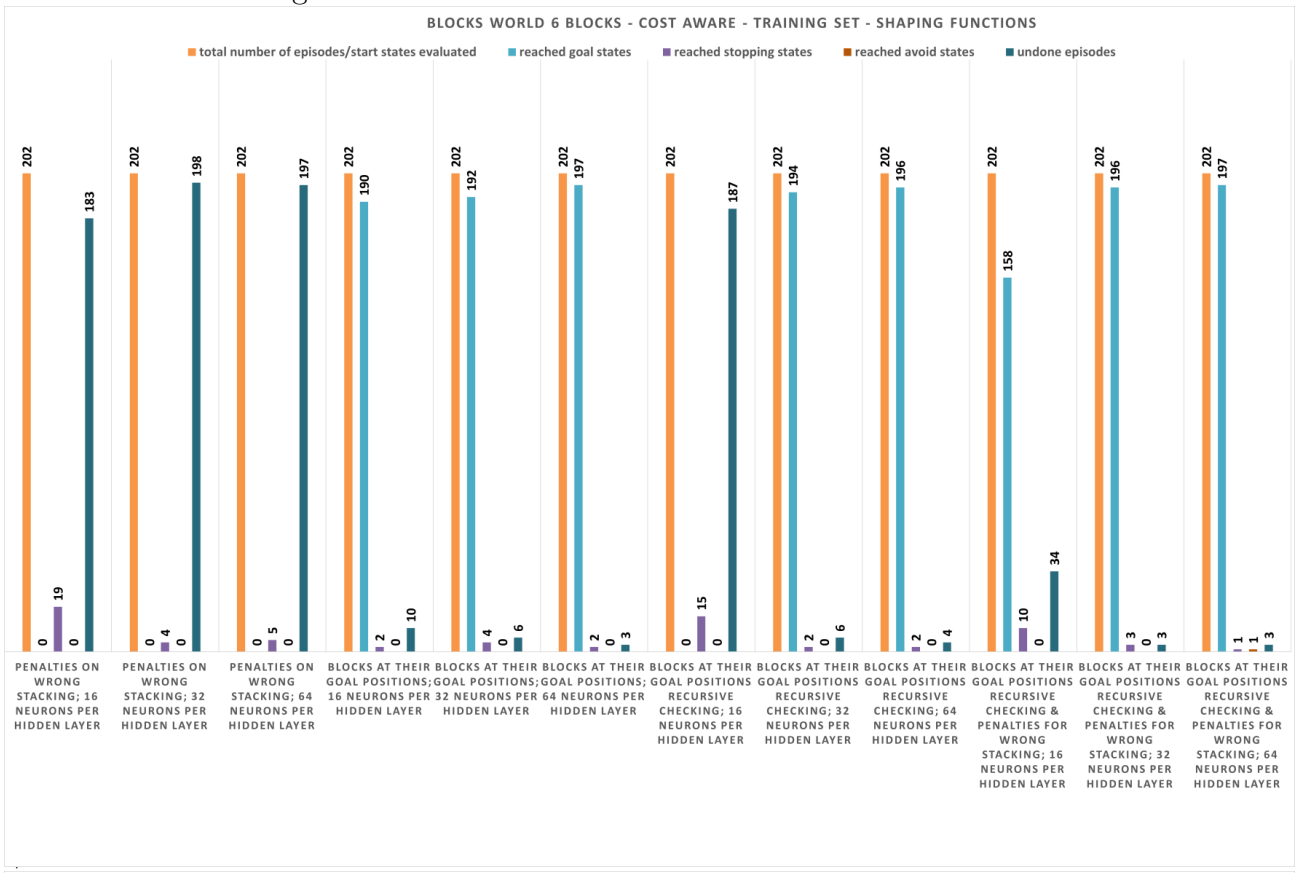


Figure A.13: Performance of the learned networks:



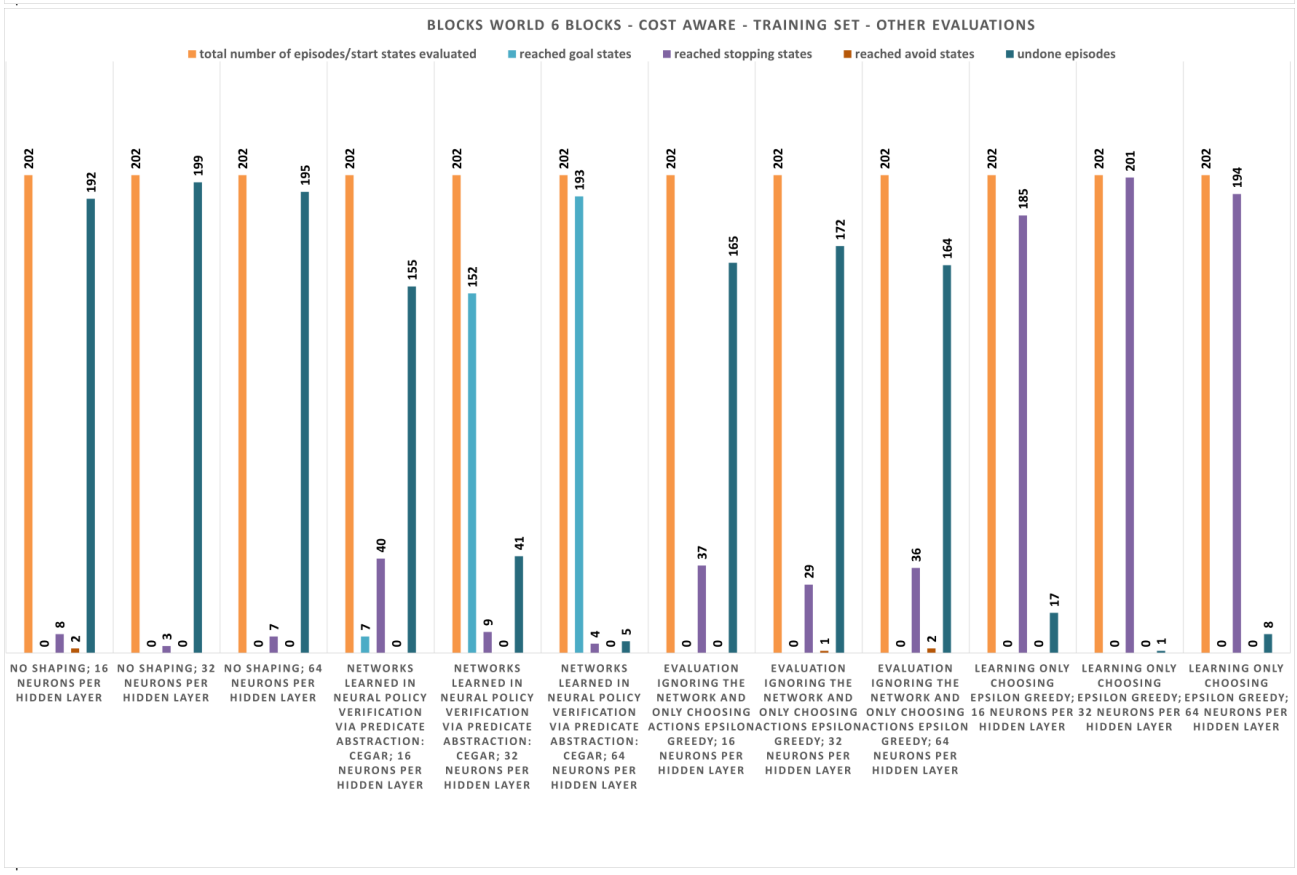
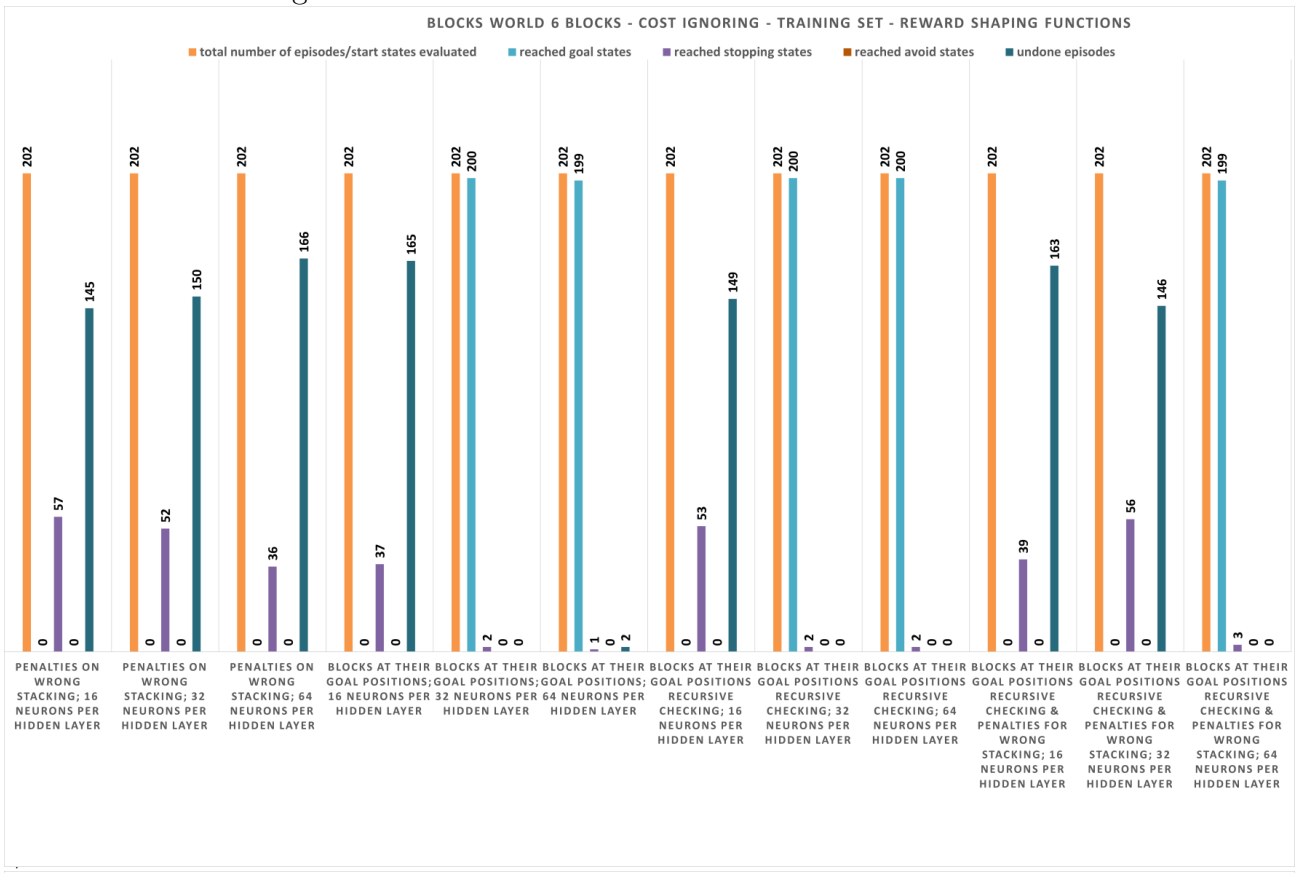


Figure A.14: Performance of the learned networks:



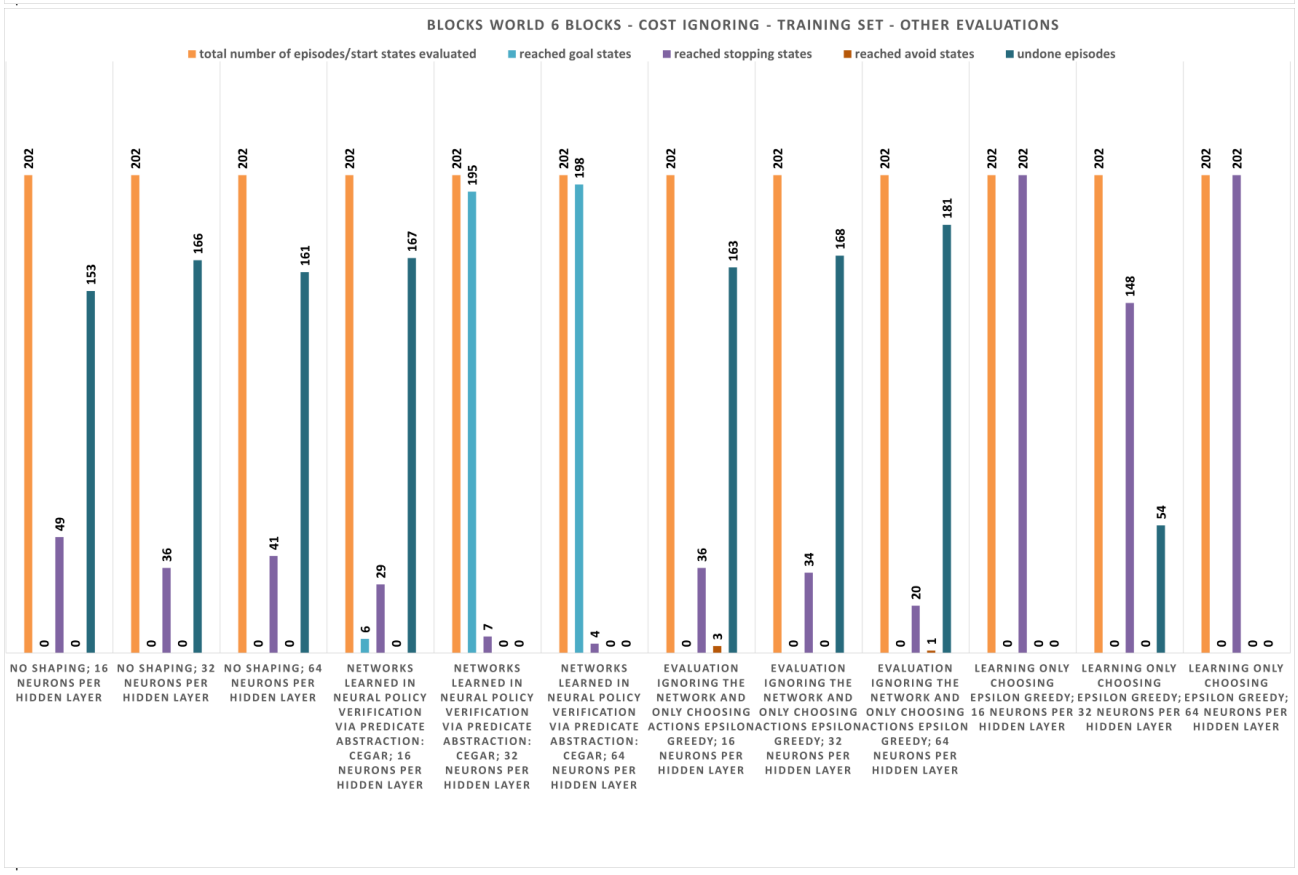
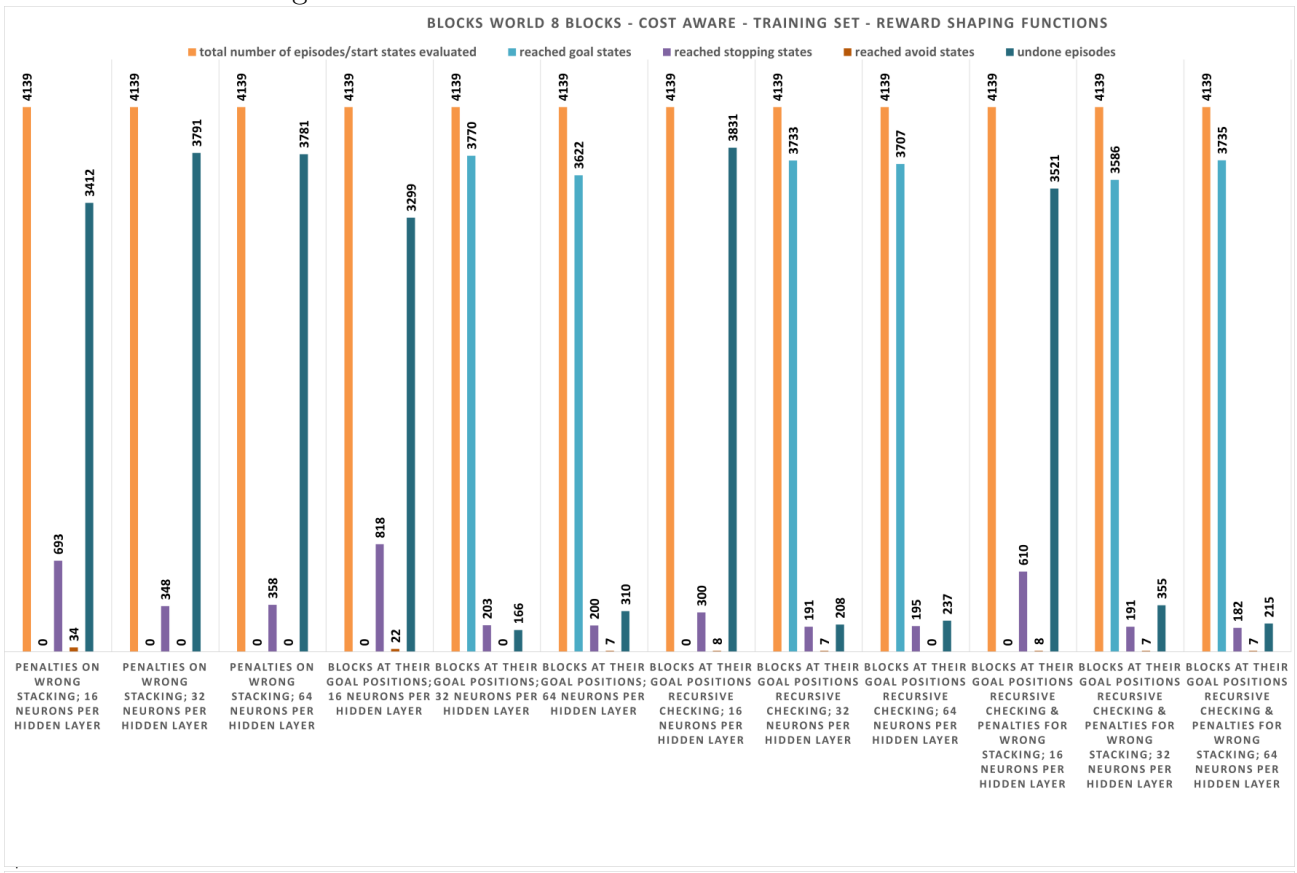


Figure A.15: Performance of the learned networks:



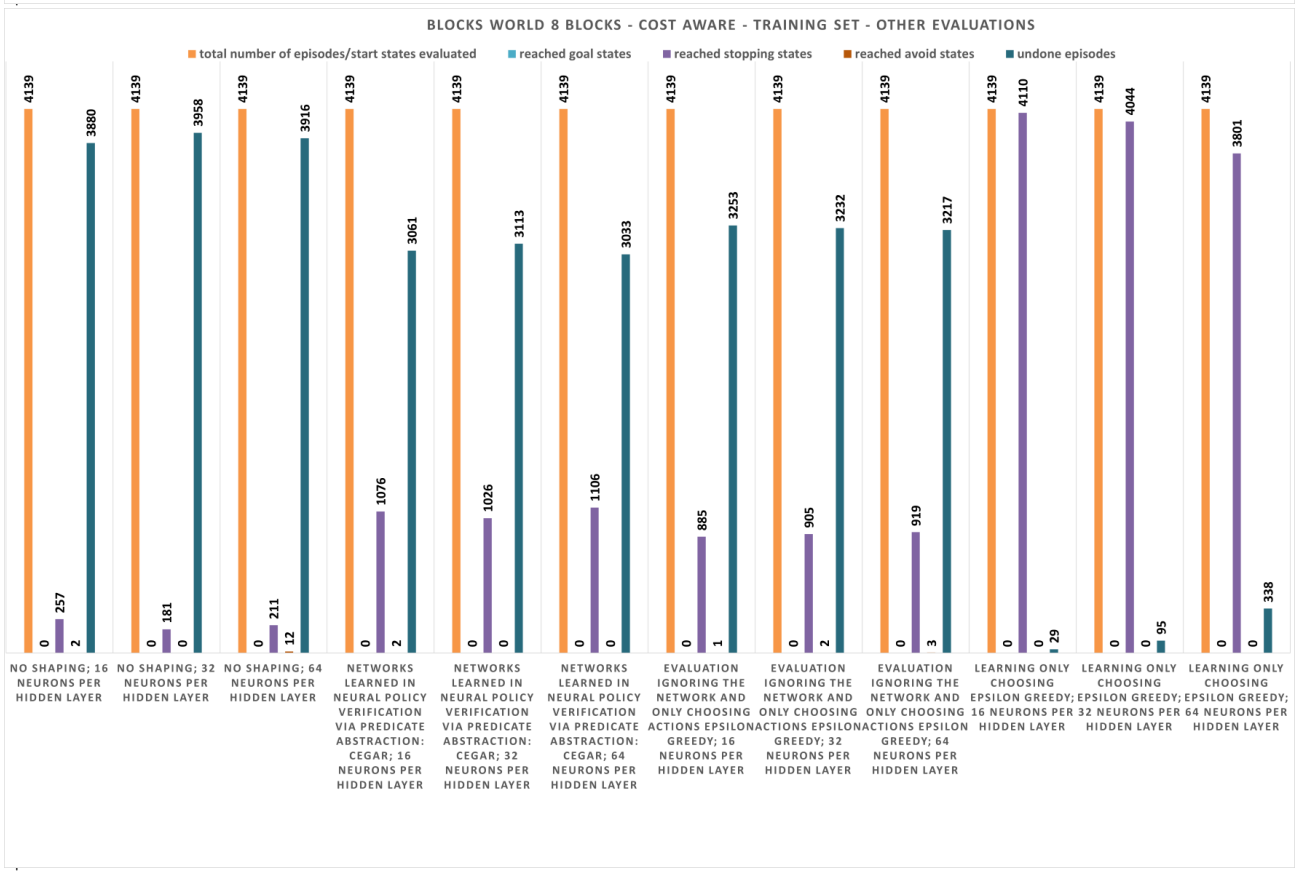
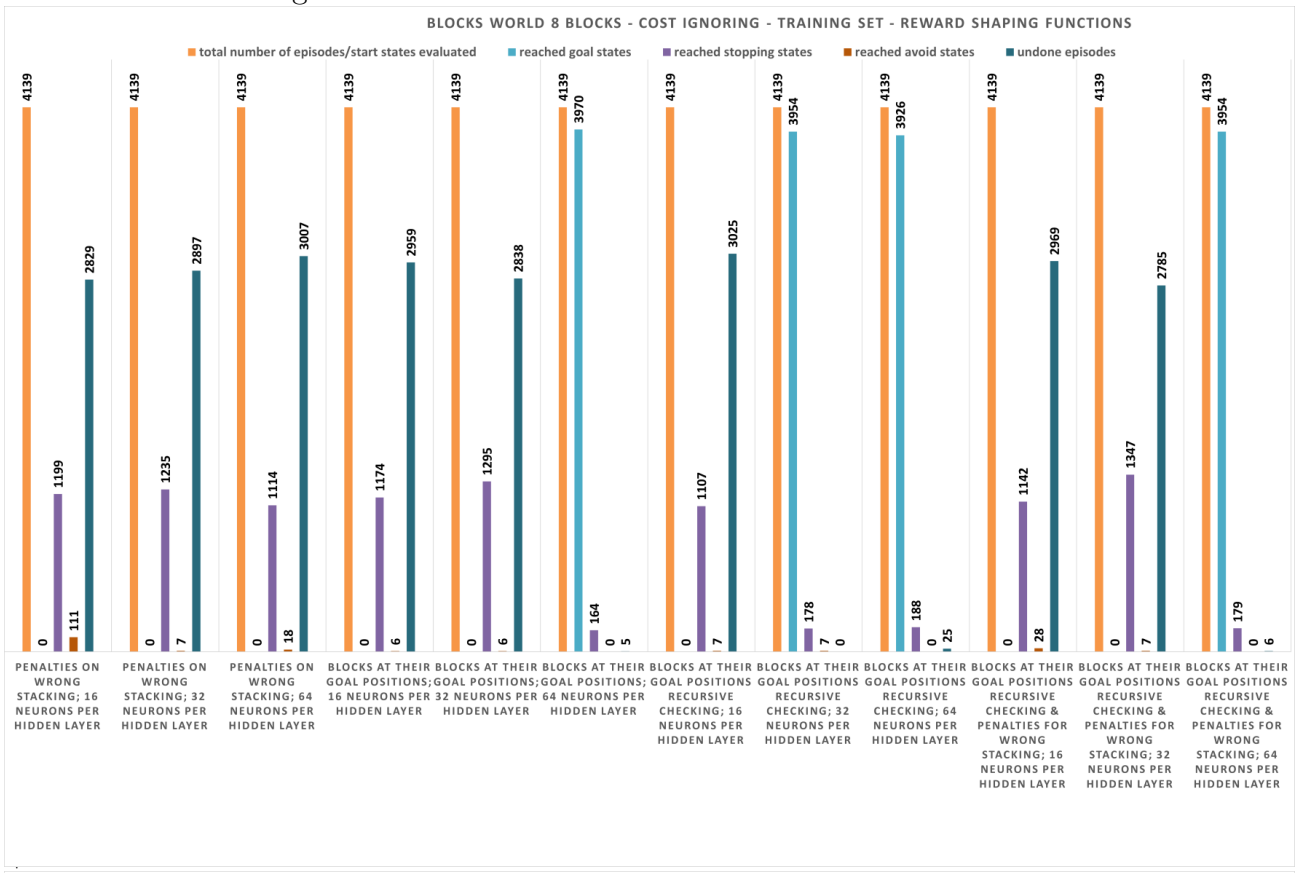


Figure A.16: Performance of the learned networks:



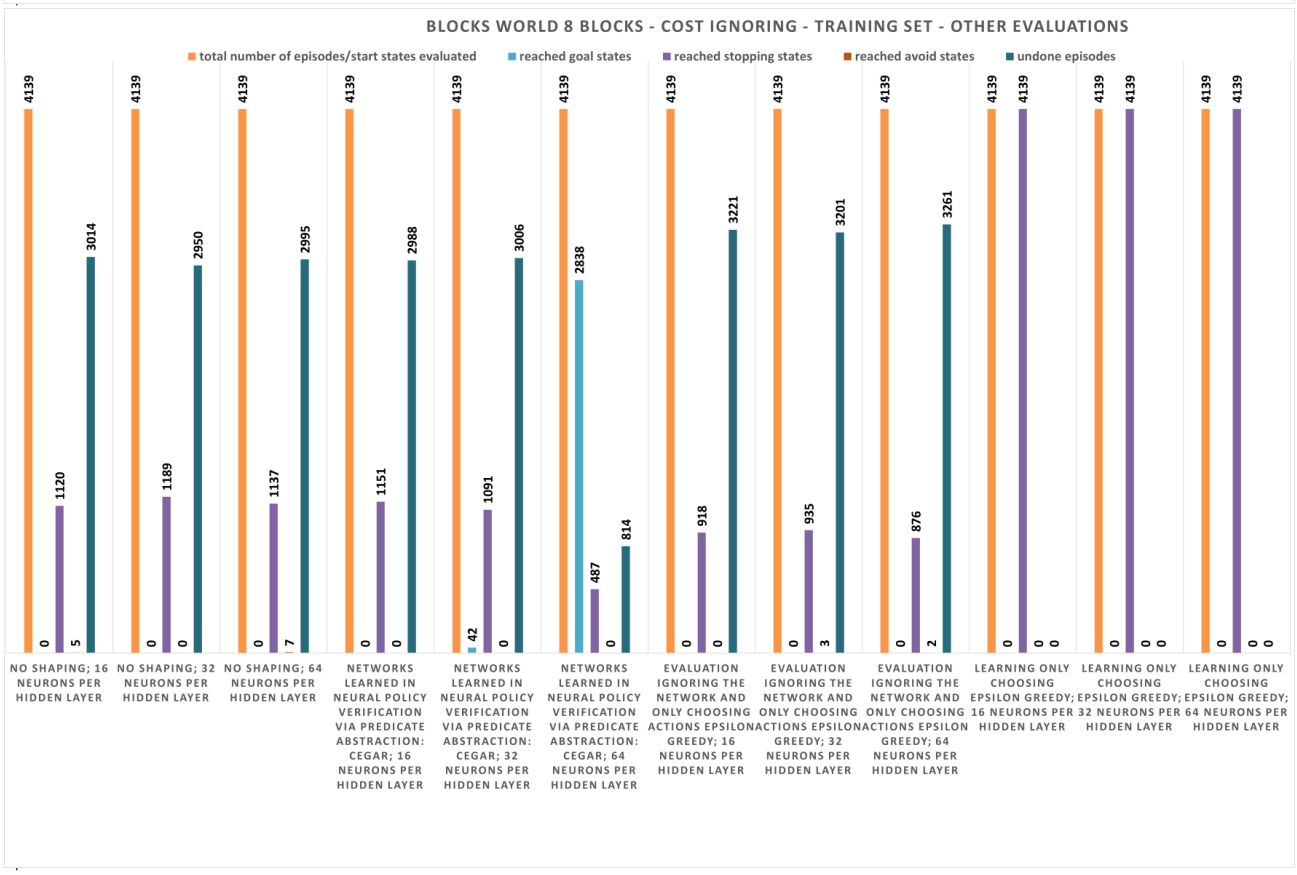
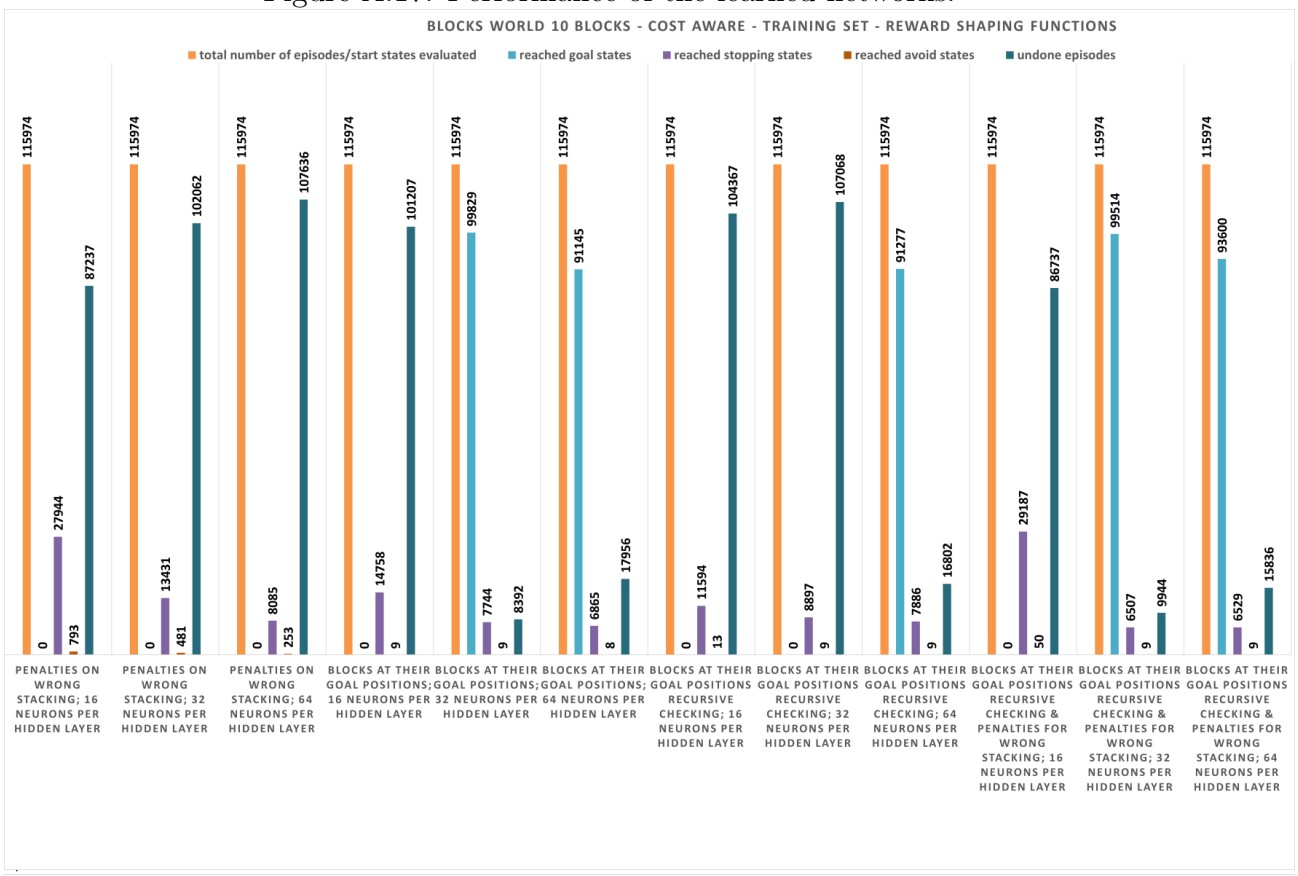


Figure A.17: Performance of the learned networks:



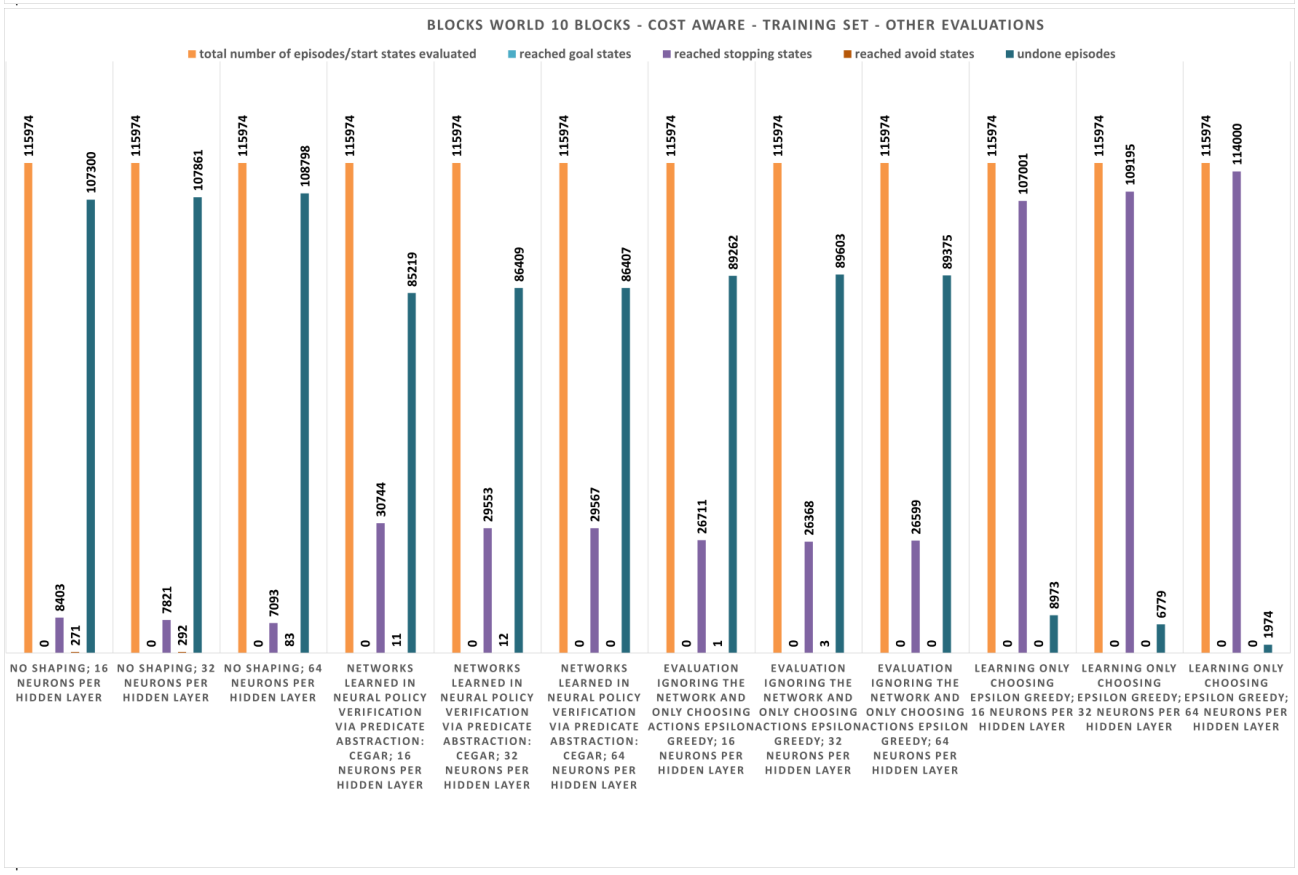
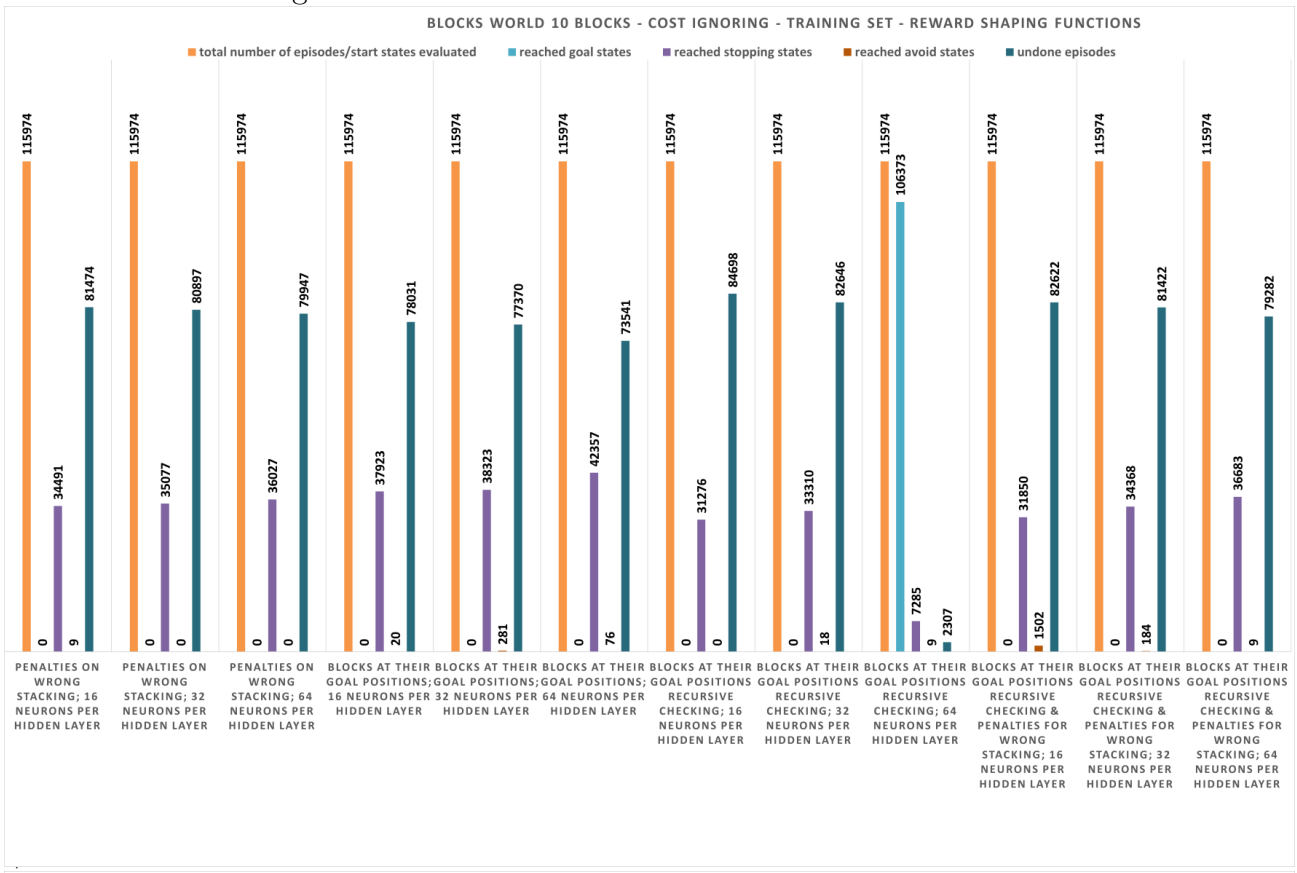


Figure A.18: Performance of the learned networks:



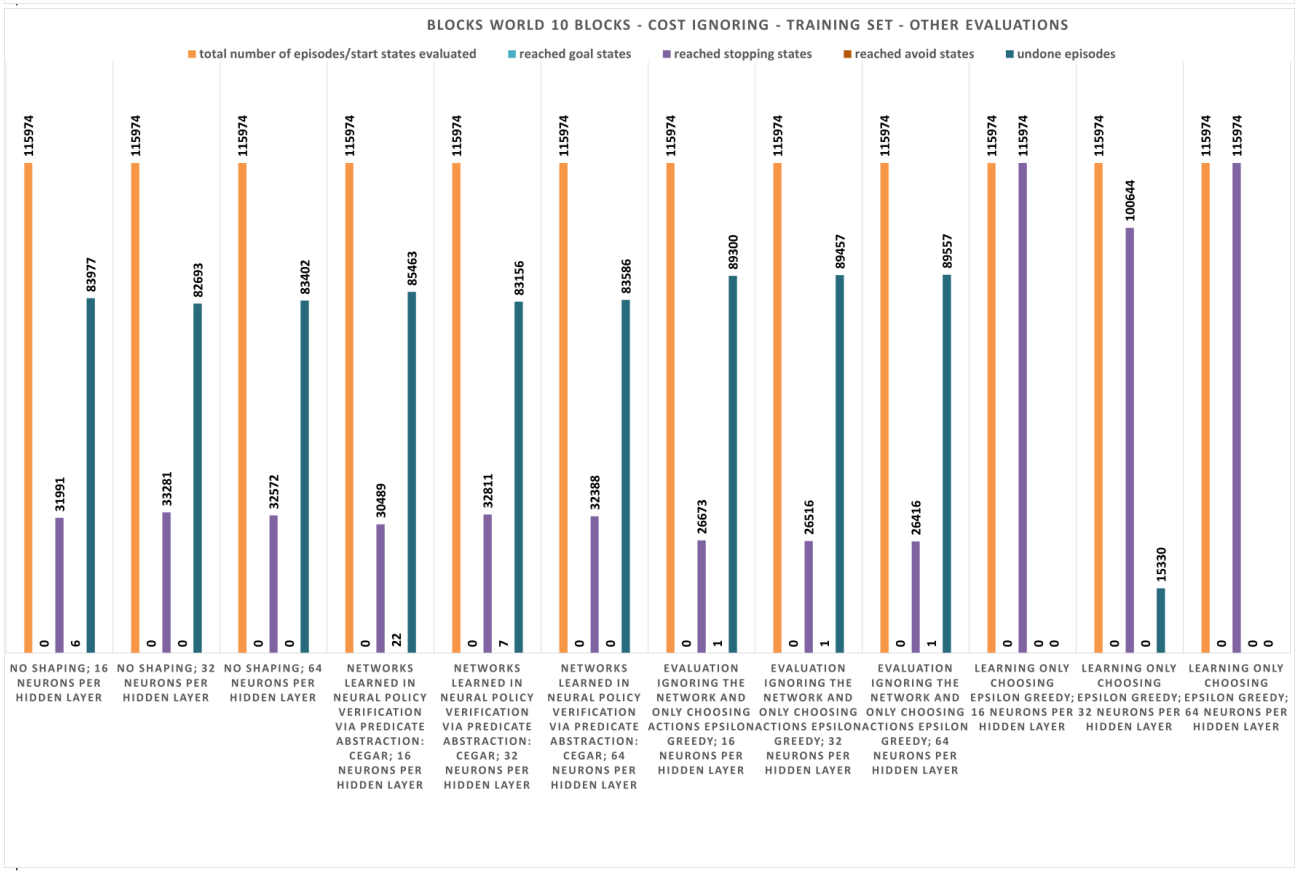
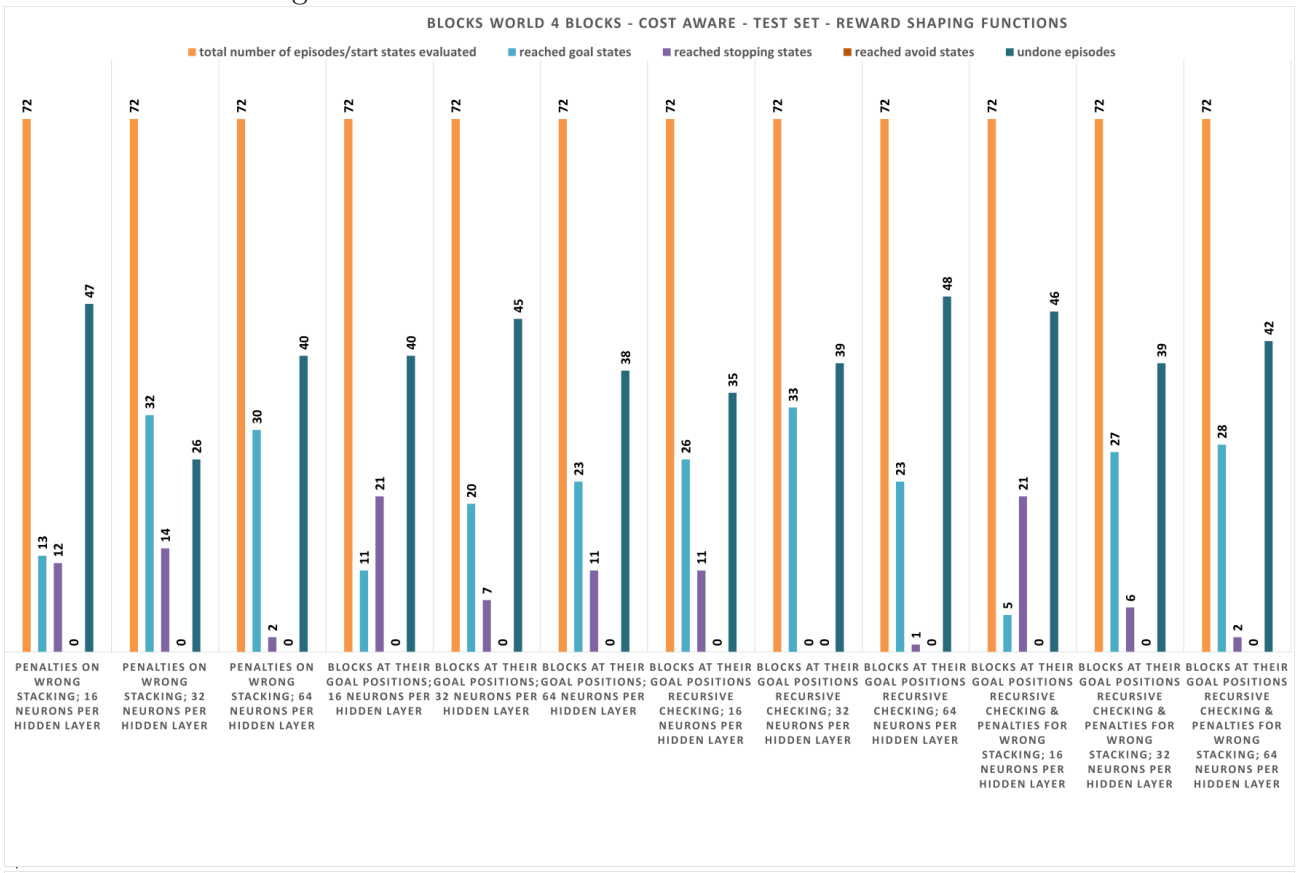


Figure A.19: Performance of the learned networks:



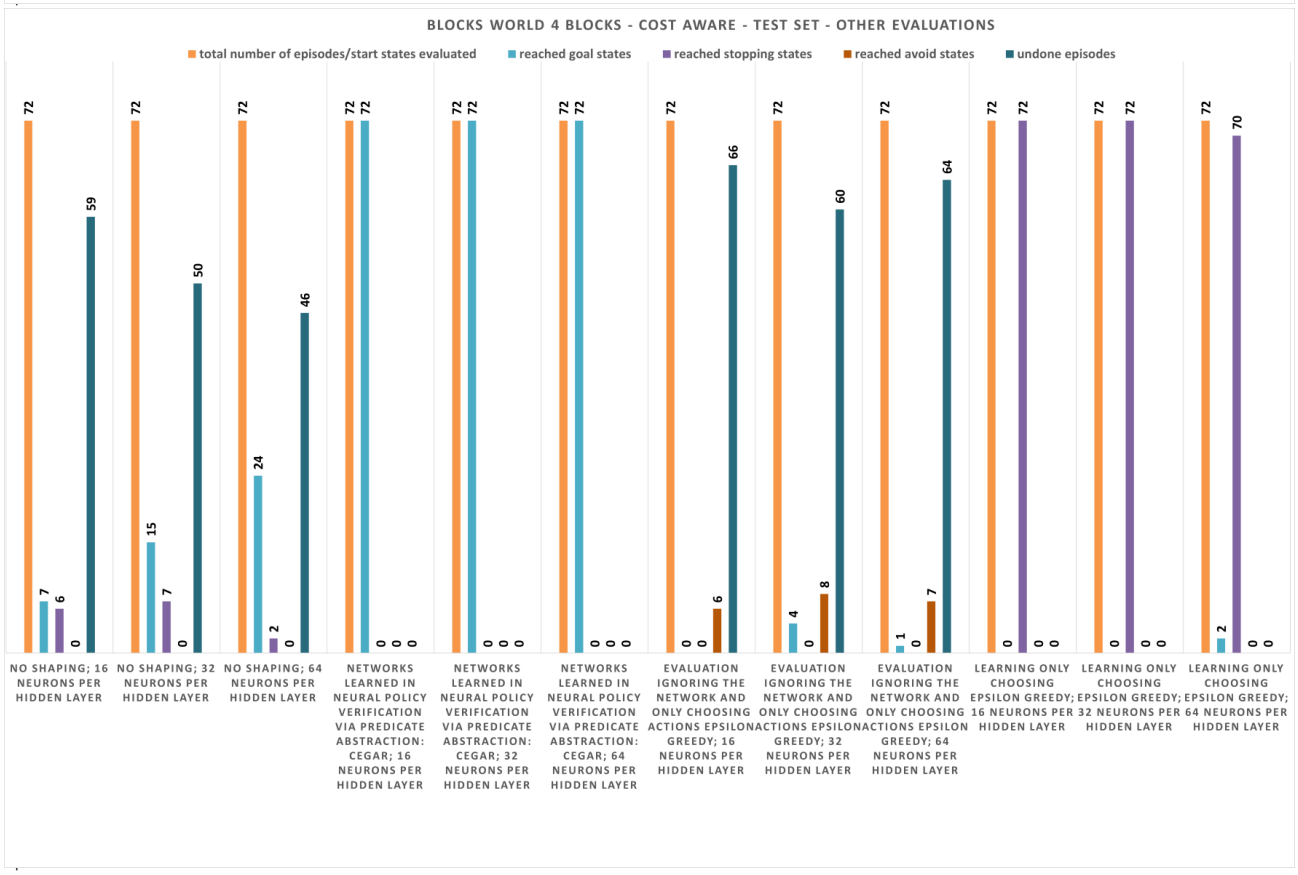
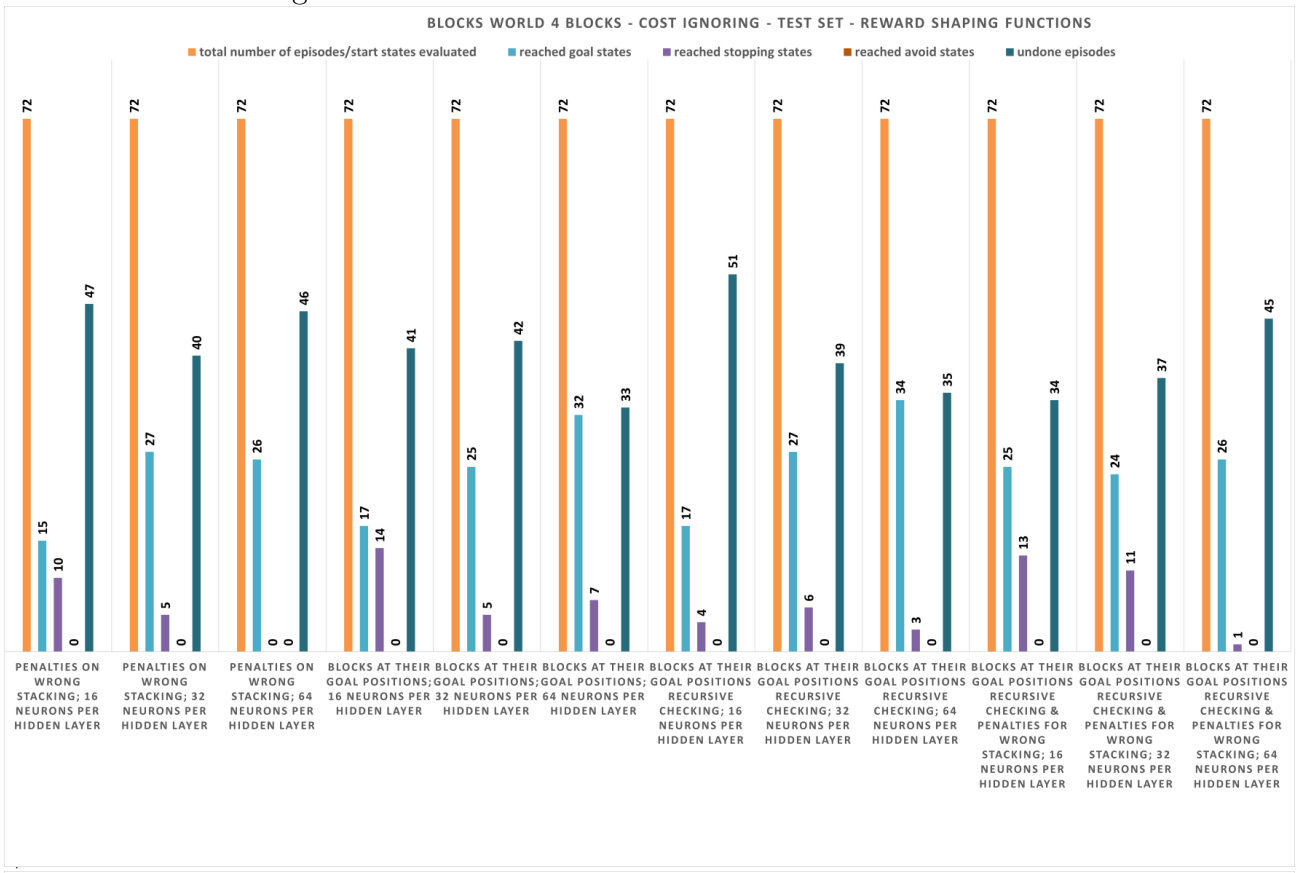


Figure A.20: Performance of the learned networks:



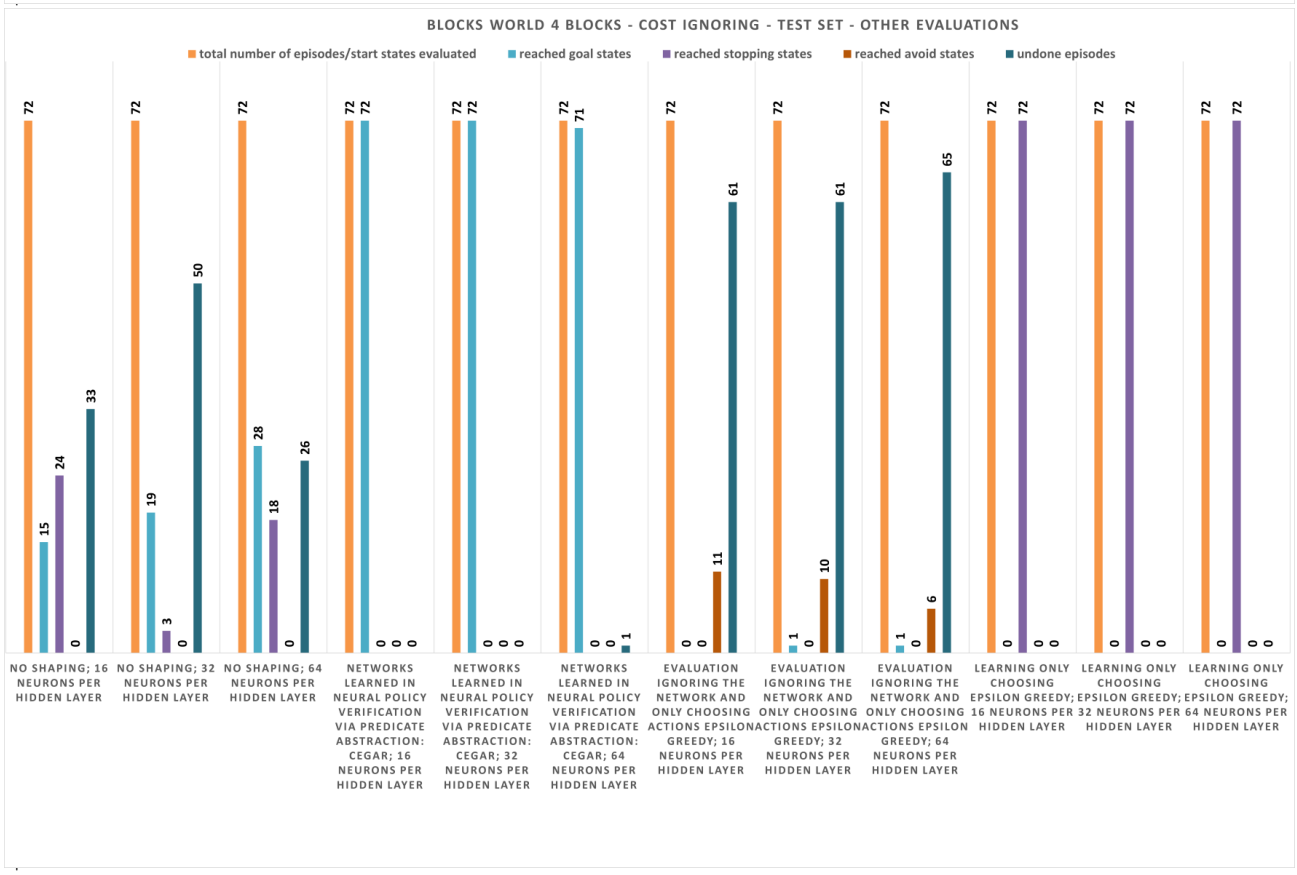
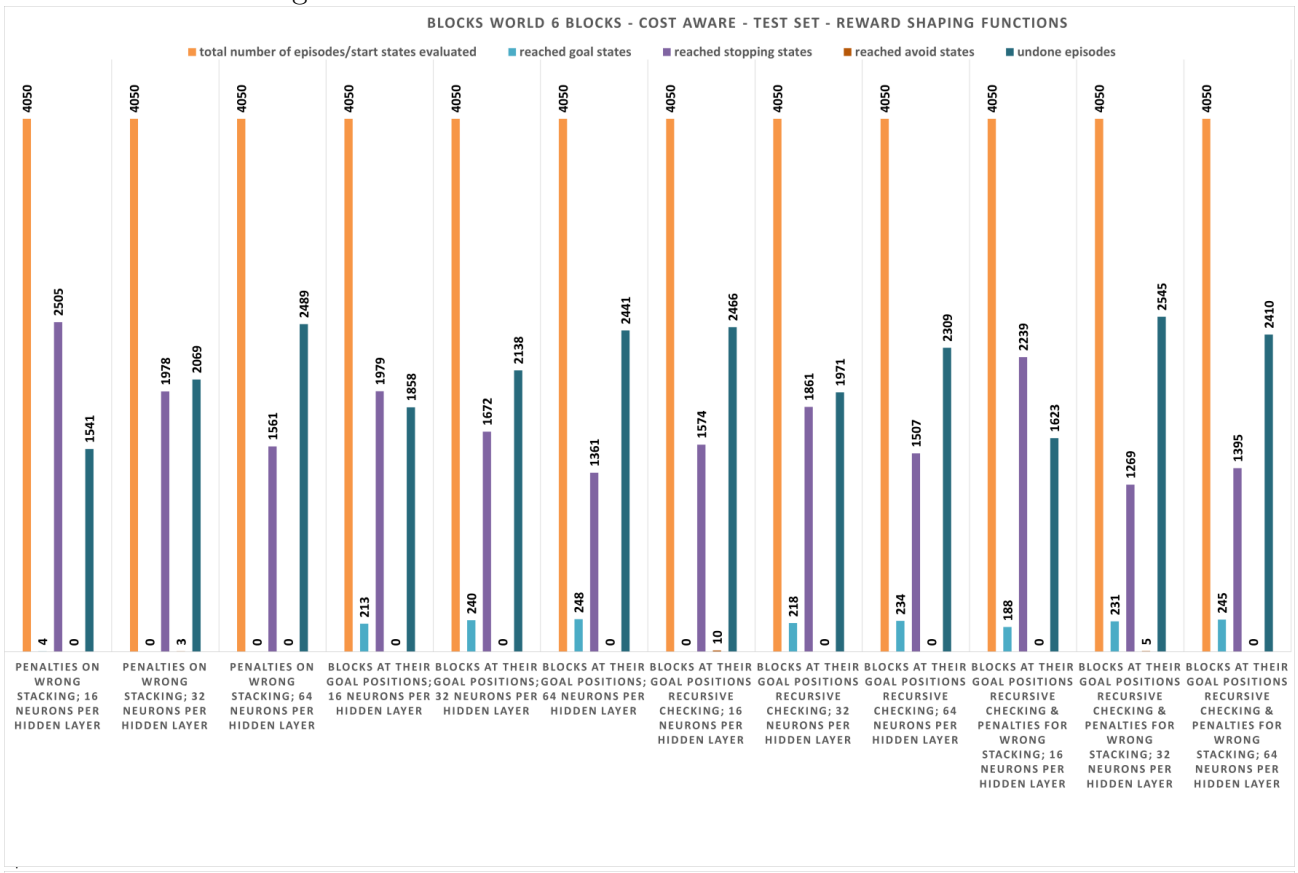


Figure A.21: Performance of the learned networks:



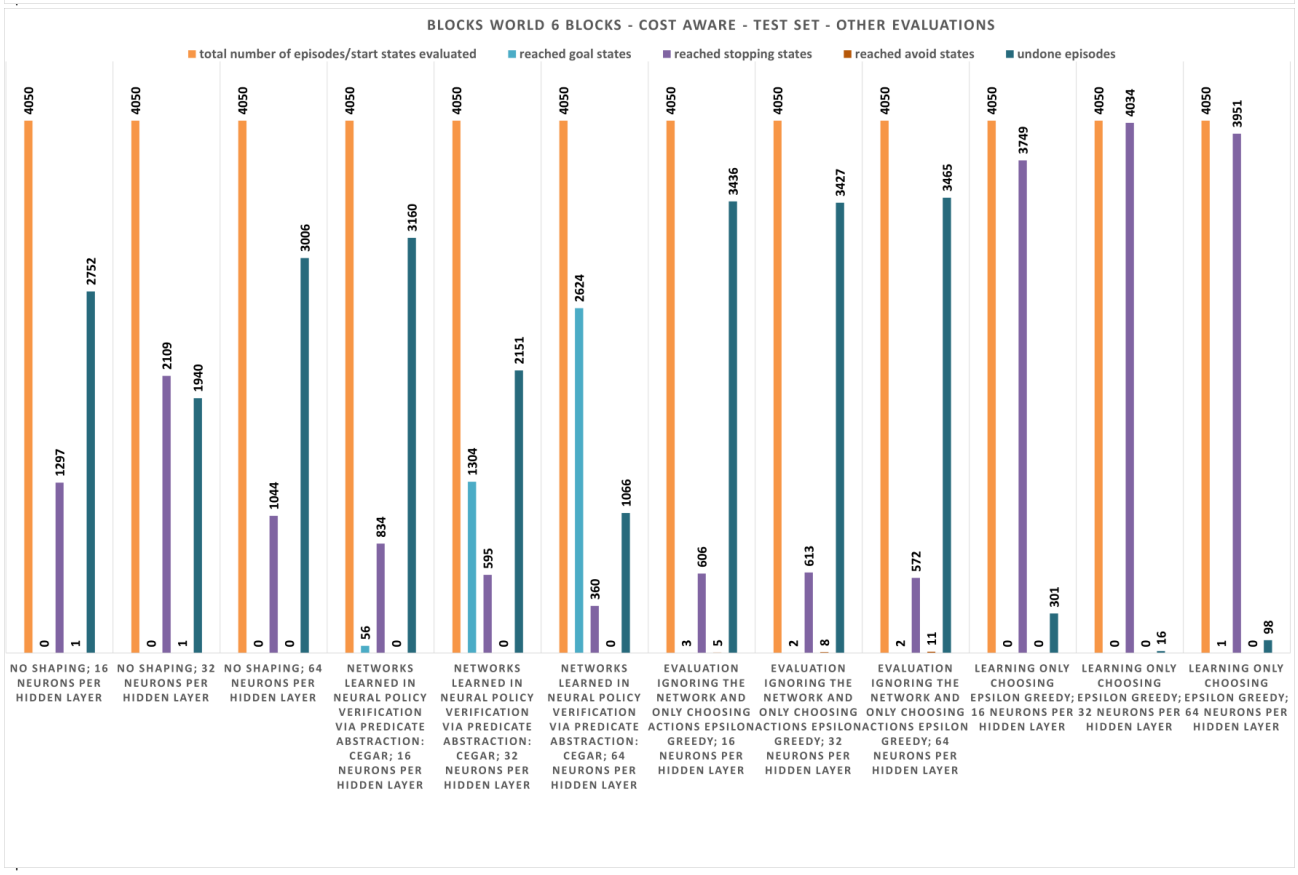
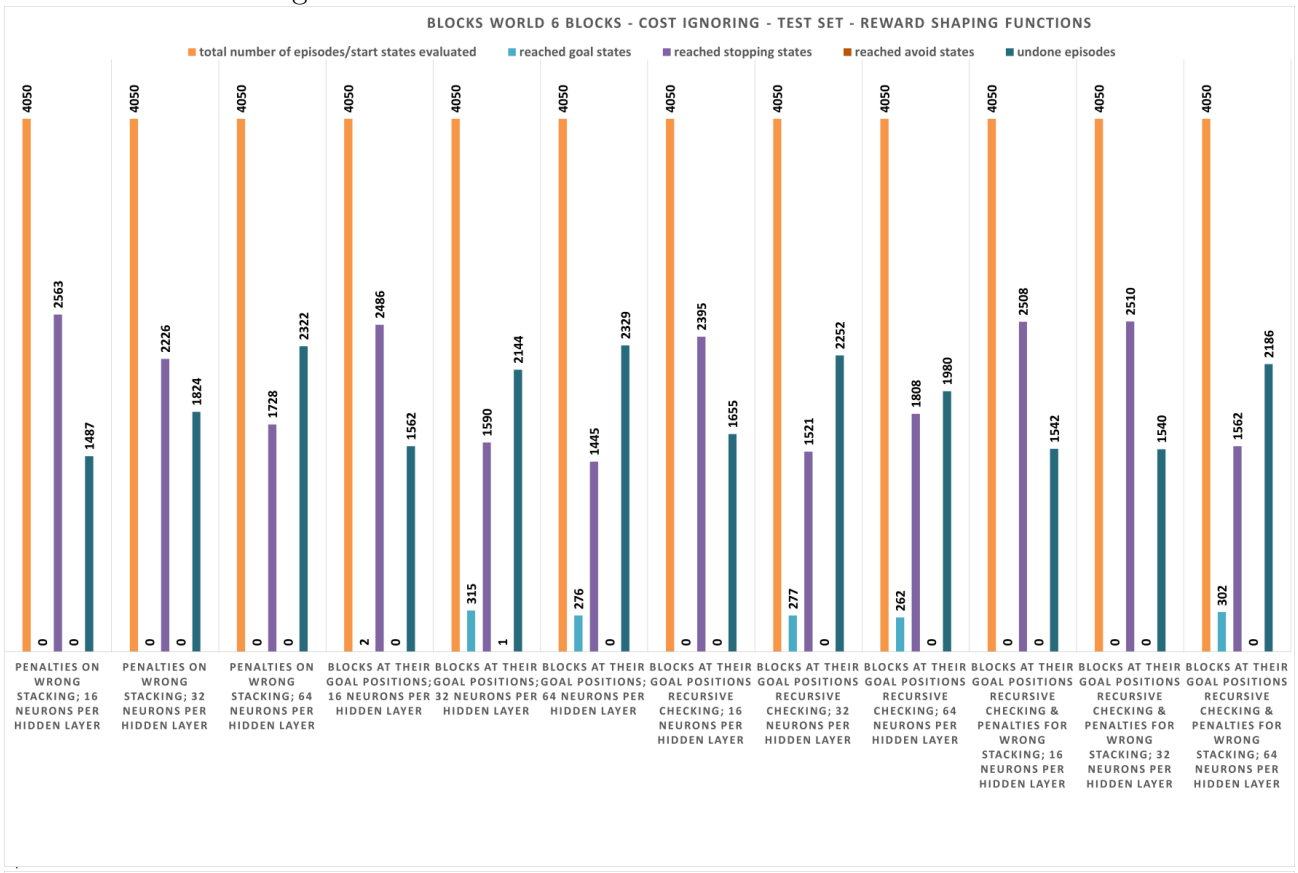


Figure A.22: Performance of the learned networks:



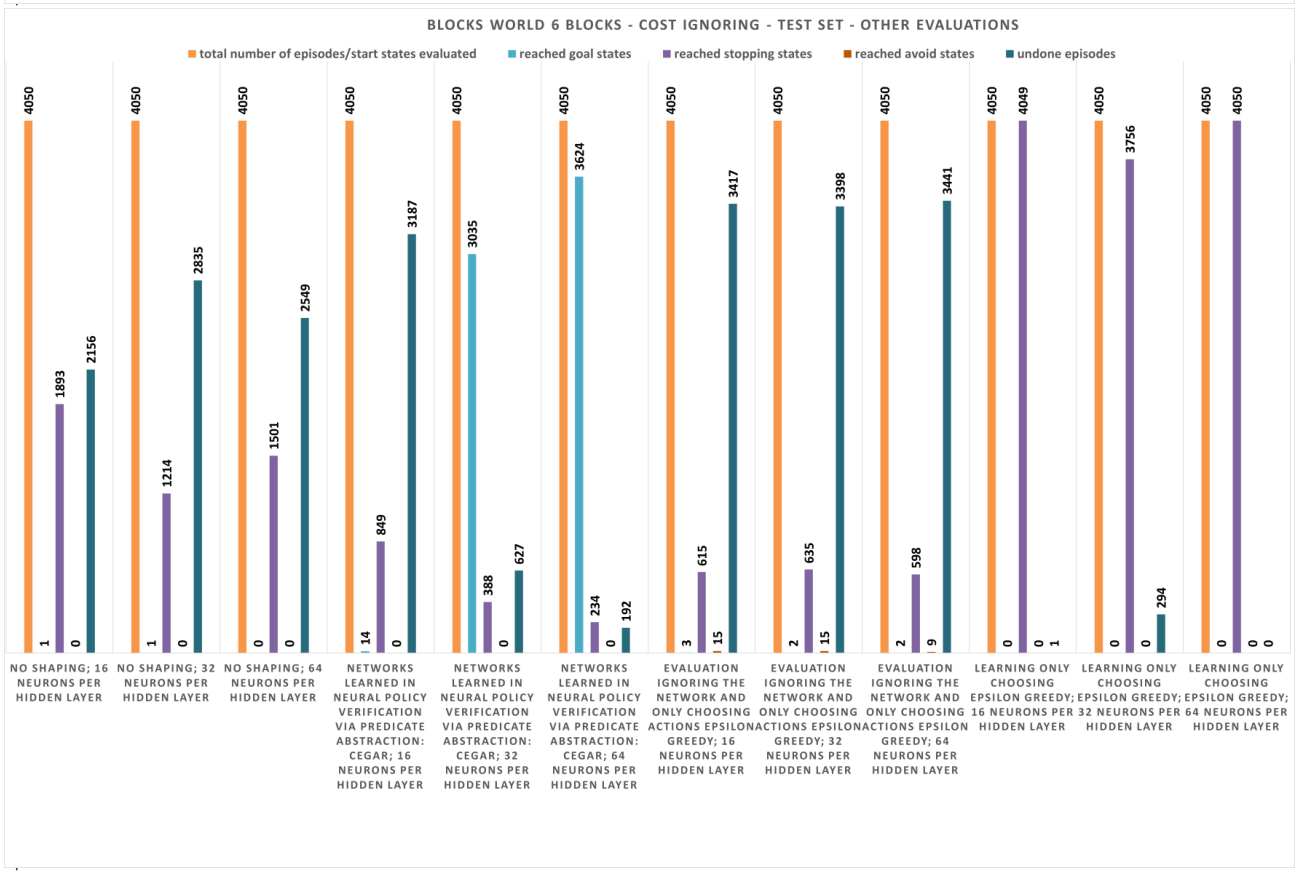
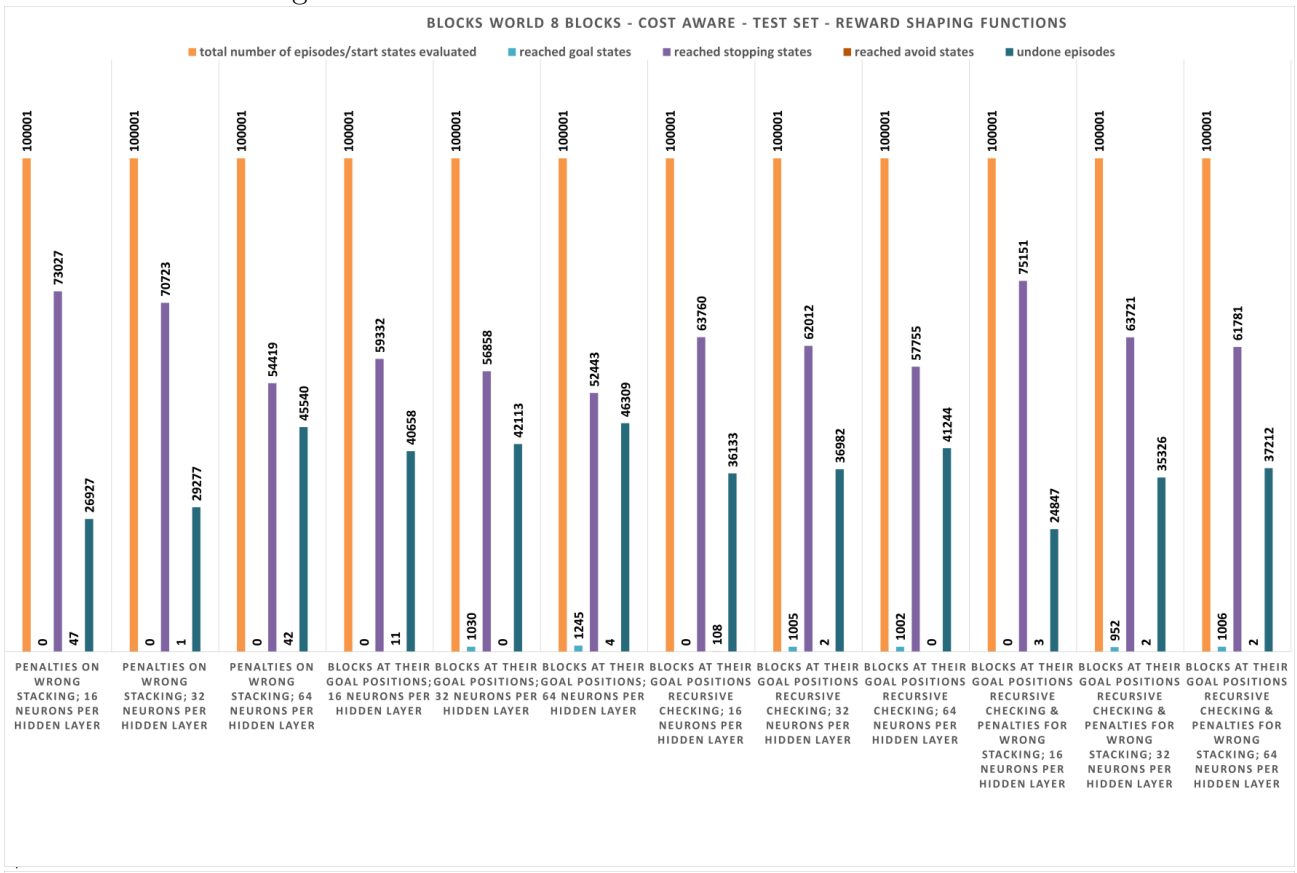


Figure A.23: Performance of the learned networks:



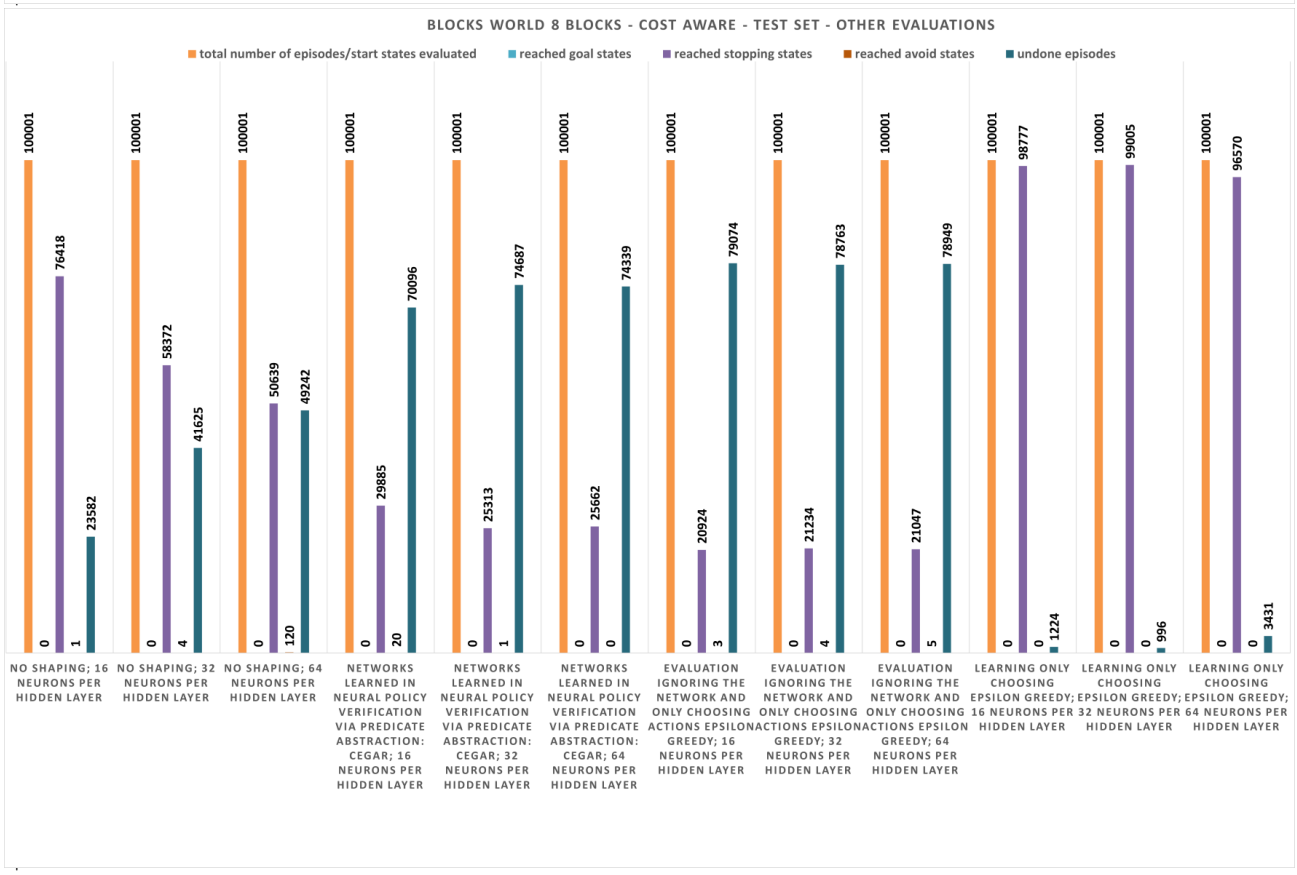
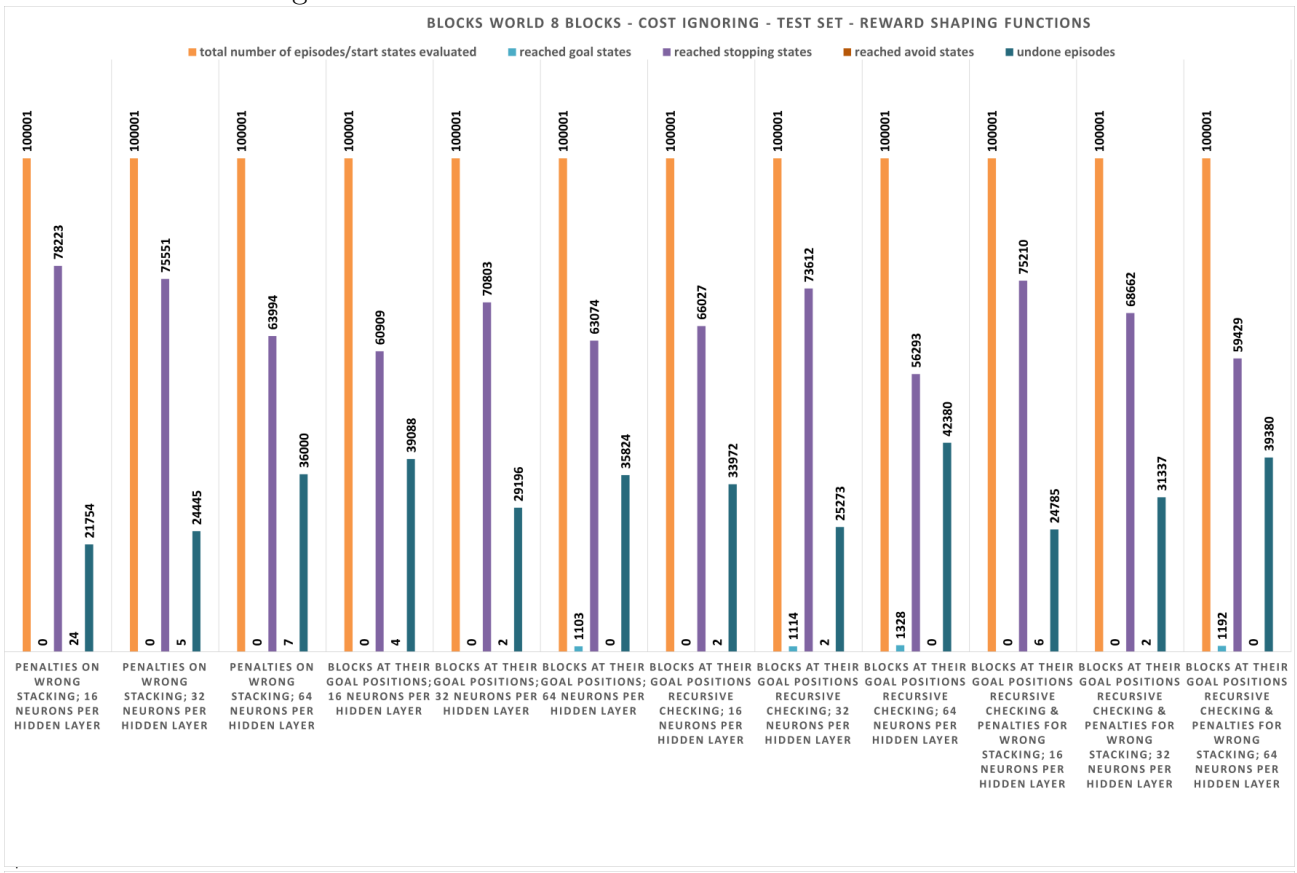


Figure A.24: Performance of the learned networks:



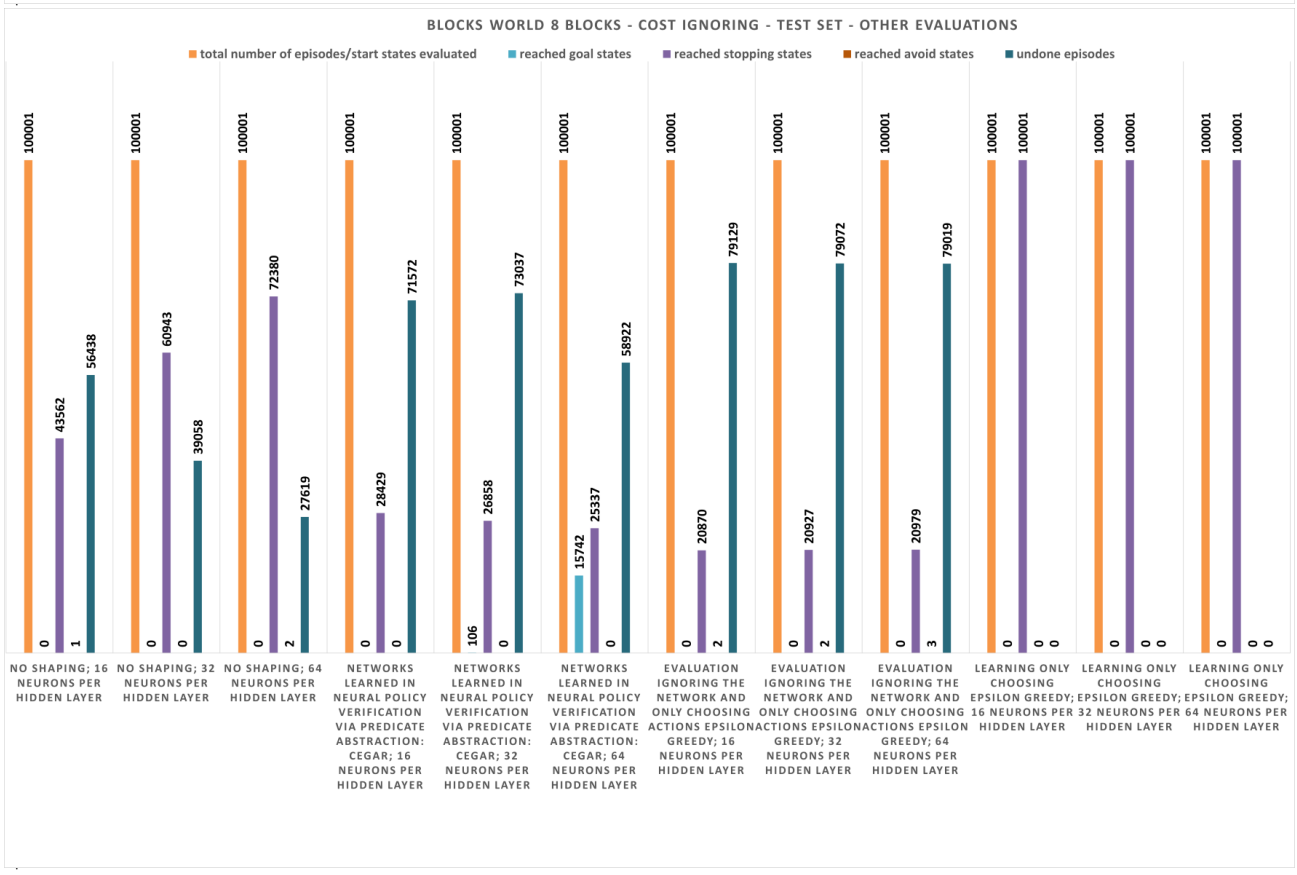
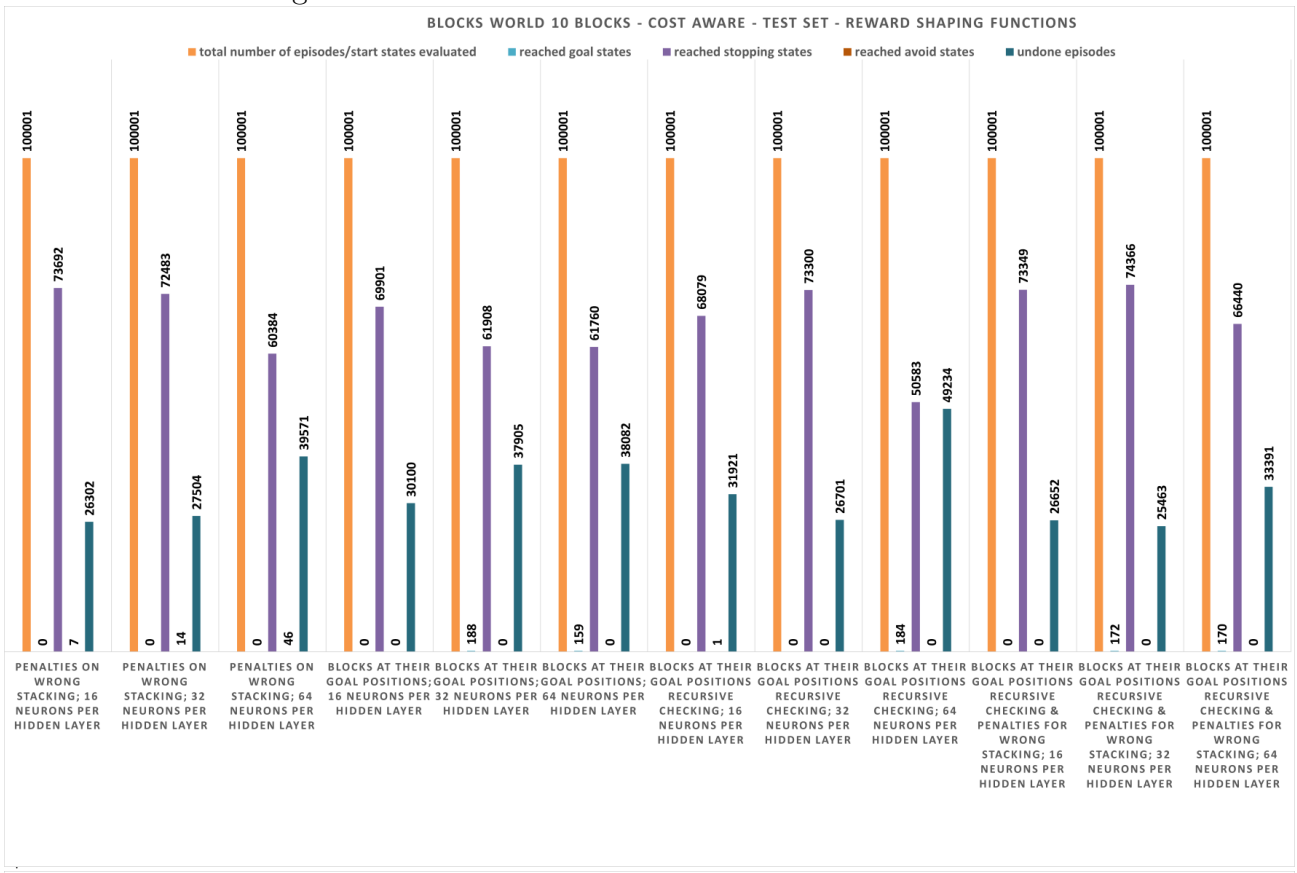


Figure A.25: Performance of the learned networks:



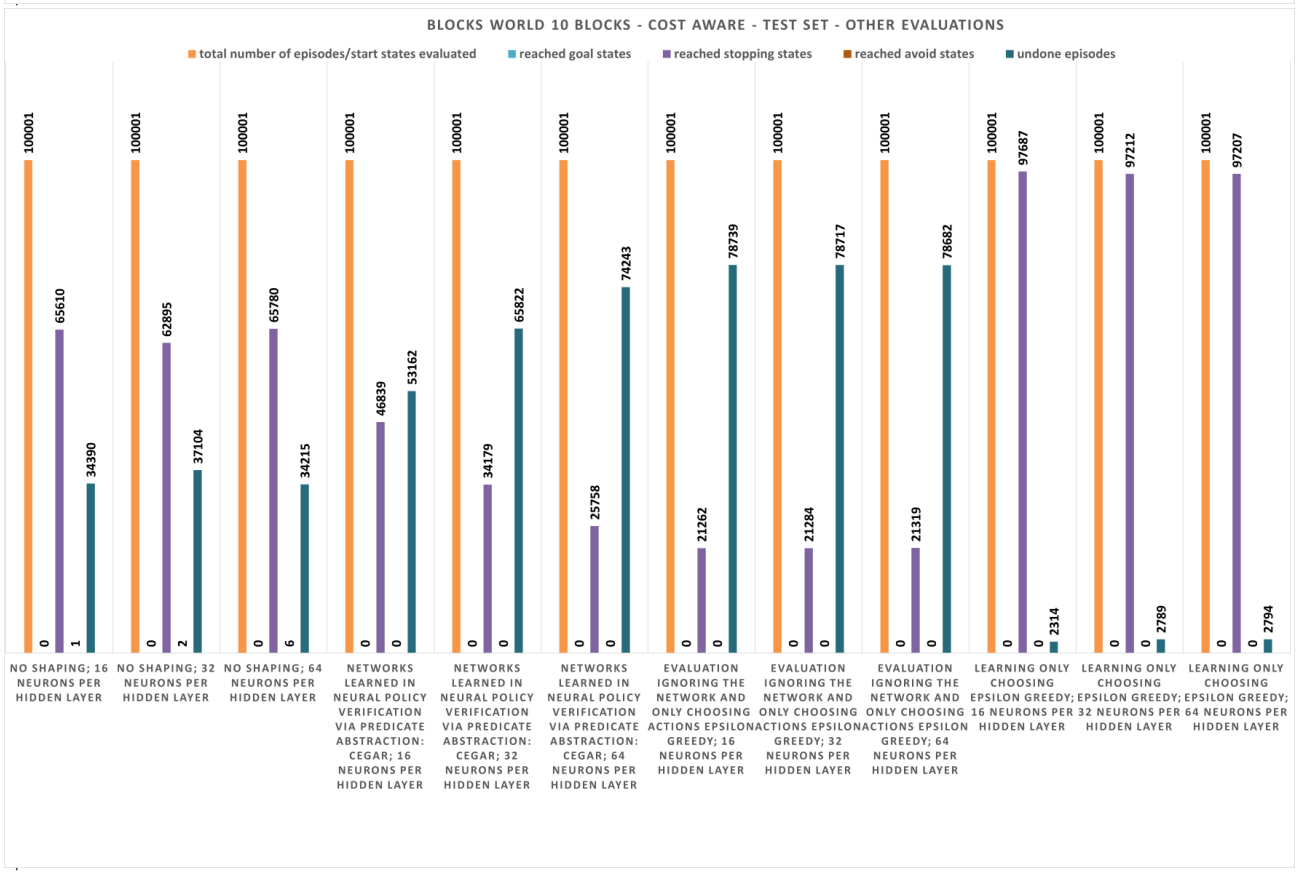


Figure A.26: Performance of the learned networks: