Solving Euclidean Markov Decision Processes with Neural Networks

Department of Computer Science
Aalborg University
June 10, 2021

Alexander C. Eilertsen
aeiler16@student.aau.dk

Abstract

When dealing with machine learning on cyber-physical systems, one problem is to train the models without extensive cost or harm to the system or its surroundings as the method learns. One method is to use Priced Timed Markov Decision Processes over a Euclidean state space to define a formal model for these systems and train on. We attempt to use Neural Networks to find optimal strategies for such models. We do this by implementing Deep Q-Network in Uppaal Stratego, make a sweep over possible hyperparamters for DQN, select three candidates and test these against the current state of the art optimization algorithm in Uppaal Stratego. Our results show that DQN can with the right hyperparameters find the optimal strategy for simple models in fewer runs than the current method, and find better strategies on some of the more complex models. However, we could not find improved strategies for all models within the tested set hyperparameter configuration.

I. Introduction

For cyber-physical systems it is often either dangerous or costly to train machine learning controllers directly on the system as the system might be at risk of damaging it self or its surroundings while the controller learns such domains could be the railway[7] or satellite systems[13]. However, if accurate formal models of the systems are available, then preliminary controllers can safely be learned on data generated from these models.

Model checking tools such as Uppaal Stratego [5] facilitates the learning of near optimal strategies for model that can be presented as a Priced Timed Markov Decision Processess (PTMDPs)[6][4]. In [6] they develop a reinforcement learning strategies based on Q-learning and online partition refinement techniques, which has show great experimental results when it comes finding strategies for Markov Decision Processs (MDPs) over a Euclidean state space. They test this method on several case studies each derived from one of three different models. The tests are done using Uppaal Stratego. An alternative could be Neural Networks (NNs) which has shown promising results in similar reinforcement learning environments such as games[12][9] which could be modelled as PTMDPs. We therefore propose to use NNs to learn strategies for PTMDPs.

In this paper we investigate the use of Deep Q-Network (DQN) to find strategies of PTMDPs. This investigation is divided into two series of experiments. The first deals with finding the optimal hyperparameters to use on the models. The second compares DQN using three sets of hyperparameters found in the first series of experiments against the current state of the art Q-learning method in Uppaal Stratego[6].

II. Neural Networks

Artificial Neural Networks are computational systems inspired by how the brain use a system consisting of billions of small simple processing units called neurons to make fast and complex computations. In comparison with the brain, artificial neural networks are far smaller, often consisting of a few hundred neurons instead of billions. However, the idea remains the same with having a set of small computational units, processing and sending information between each other to tackle complex problems.

A. Network Structure

In a neural network, the neurons are organized in a graph structure with weighted directional connections between them. These neurons are usually divided into layers. The layers determine when a neuron is activated and process the information available to it, since the neurons are activated one layer at the time going from the input layer to the output layer. A common type of structure is the fully connected feed forward Neural Network. This structure is defined by each neuron in a non-input layer being connected to each neuron in the previous layer. The number of hidden neurons and how they are connected are one of elements that determines the possible functions that a neural network can approximate. An example of this is *xor*, a logical operation that checks whether boolean inputs are the same. A result table of *XOR* can be seen in Table I. In order for a neural network to be able to fully express a function that behaves as *XOR* it needs at least two hidden neurons as seen in Figure 1. In this example x_0^1 serves as an *OR* operator activating if either x_0^0 or x_1^0 is 1, x_1^1 functions as an *NAND* operator activating if not both x_0^0 and x_1^0 is active, and x_0^2 approximates an *AND* operator activating if both x_0^1 and x_1^1 is active. Using the tahn function as the activation function for x_0^1 and x_1^1 and the sigmoid function as the activation function for x_0^2, this neural network will have the input output relation seen in Table II. Then by applying an interpreter that classify results above 0.5 as $\top$ and results less or equal than 0.5 as $\bot$, this network can be used to emulate *XOR*.

XOR	$I_0 = \top$	$I_0 = \bot$
$I_1 = \top$	$\bot$	$\top$
$I_1 = \bot$	$\top$	$\bot$

Table I: XOR logic operator, with I_0 and I_1 being the inputs for XOR

XOR Neural Network	$x_0^0 = 1$	$x_0^0 = 0$
$x_1^0 = 1$	0.06	0.60
$x_1^0 = 0$	0.60	0.06

Table II: Outputs of example neural network for xor, for all possible states

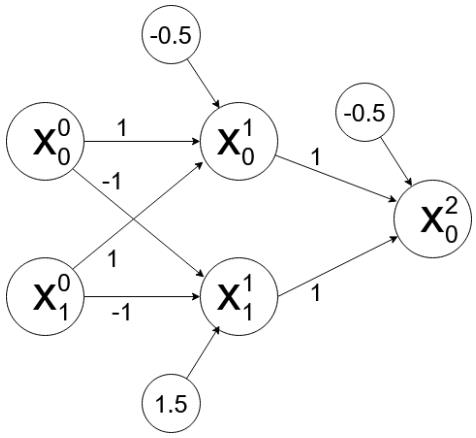


Figure 1: The smallest neural network capable of solving xor, with x_0^0 and x_1^0 being the input neurons, x_0^1 and x_1^1 being two hidden neuron, x_0^2 being the output neuron, and the small circles being the bias for the non-input neurons

B. Neurons

Neurons are small relatively simple processing units that when combined in multi-layer feed-forward structure becomes a universal function approximator, assuming enough hidden neurons are available[3]. The neuron it self can be defined as the function composition of two other functions. An input function λ, which converts the weighted outputs of the previous layer into a numerical value, and an activation function ϕ which further process this value. We will denote the ith neuron in layer l as x_i^l and the output of that neuron as o_i^l. The relation between these variables can be seen in Equation 1.

$$o_i^l = x_i^l(O^{l-1}, W^l, b_i^l) = \phi_i^l(\lambda_i^l(O^{l-1}, W^l, b_i^l)) \tag{1}$$

In this function neuron x_i^l takes as input the output vector of the neurons in layer $l-1$, O^{l-1}, the weight vector for the connections from these neuron to neuron x_i^l denoted as W_i^l, and a bias b_i^l. λ_i^l and ϕ_i^l is the specific input and activation function for neuron x_i^l. One of the most common input functions, and the one we will be using, is the sum function shown in Equation 2

$$\lambda_i^l(O^{l-1}, W_i^l, b_i^l) = \sum_{j=0}^{j}(o_i^{l-1} \cdot w_{i,j}^l) + b_i^l \tag{2}$$

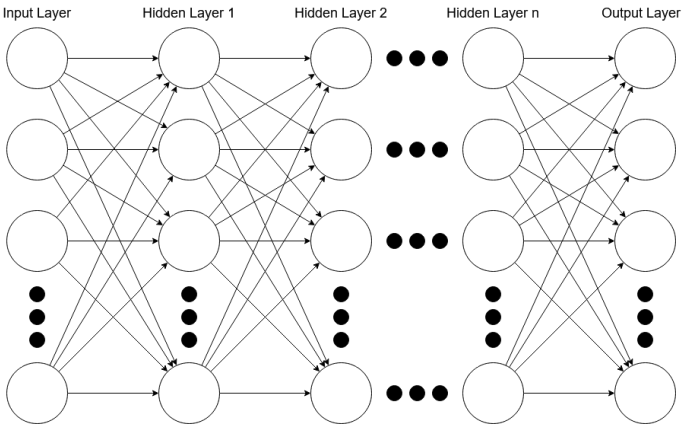


Figure 2: Fully Connected Feed Forward Neural Network Example

This function sums the weighted output of the neurons in the previous layer and adds the bias to it. We use the notation $w^l_{i,j}$ as the weight of the connection from x^{l-1}_j to x^l_i, and J as the size of layer $l-1$. Table III shows four commonly used activation functions, their output boundaries, and their derivatives.

It worth noting that if we use the sum input function, and let Φ^l be a function that applies the activation to all neurons in layer l then calculating the output of a whole layer can be done efficiently through the matrix operations in Equation 3.

$$O^l = \Phi^l(W^l \cdot O^{l-1} + B^l) \tag{3}$$

In this function W^l is the weight matrix for the connection from layer $l-1$ to l and B^l is a vector consisting of the biases for layer l.

Using this equation, we can see how the neural network in Figure 1 derived at the results in Table II. We will do this for the data sample $x^0_0 = 1$ and $x^0_1 = 0$. The first step will be update the input layer to $O^0 = \binom{1}{0}$. Then we want to update O^1, for this we need W^1 and B^1, which is defined by the network to be $\left(\begin{smallmatrix} 1 & 1 \\ -1 & -1 \end{smallmatrix}\right)$ and $\binom{-0.5}{1.5}$ respectively. Plotting these values into Equation 3 we get:

$$O^1 = \Phi^1_T\left(\left(\begin{smallmatrix} 1 & 1 \\ -1 & -1 \end{smallmatrix}\right) \cdot \binom{1}{0} + \binom{-0.5}{1.5}\right) = \binom{\phi_T(0.5)}{\phi_T(0.5)} \approx \binom{0.46}{0.46}$$

The final step is to calculate X^2, which we again do based on Equation 3, where we derive $W^2 = \begin{pmatrix} 1 & 1 \end{pmatrix}$ and $B^2 = \begin{pmatrix} -0.5 \end{pmatrix}$ from the network. Resulting in: $X^2 = \Phi_S(\begin{pmatrix} 1 & 1 \end{pmatrix} \cdot \binom{0.46}{0.46} + \begin{pmatrix} -0.5 \end{pmatrix}) \approx (0.6)$

Which corresponds to the results in Table II. For fulling out the table simply go trough this process but for the other states.

Activation Function	Equation	output boundaries	Derivatives
Linear	$\phi_L(X) = X$	$(-\infty, \infty)$	$\frac{d\phi_L}{dX} = 1$
Relu	$\phi_R(X) = max(0, X)$	$[0, \infty)$	$\frac{d\phi_R}{dX} = \begin{cases} 1, & \text{if } X > 0 \\ 0, & \text{if } X < 0 \\ undefined, & \text{if } X = 0 \end{cases}$
Sigmoid	$\phi_S(X) = \frac{1}{1+e^{-X}}$	$(-1, 1)$	$\frac{d\phi_S}{dX} = \phi_S(X) \cdot (1 - \phi_S(X))$
Tahn	$\phi_T(X) = \frac{e^X - e^{-X}}{e^X + e^{-X}}$	$(0, 1)$	$\frac{d\phi_T}{dX} = 1 - \phi_T(X)^2$

Table III: Four common activation functions, their equations, result values, and derivatives

C. Training

The goal training a Neural Network is to adjust the weights and biases in order to increase the accuracy of the network. This is usually done through an optimization function and backpropagation. One of the simpler optimization functions are gradient decent which functions by repeatedly taking steps in the opposite direction of the gradient of a cost function C, that describes to what extent our Neural Network solves the problem, with a higher cost indicating worse accuracy. The reason we move in opposite direction of the gradient is because we want to move towards a local minimum of the cost function. Taking these steps, i.e. updating the weights and biases of the neural network, is done based on Equation 4 and Equation 5 for weights and biases respectively. Where we get the new weight and bias w' and b'

$$w' = w - \alpha \cdot \frac{dC}{dw} \tag{4}$$

$$b' = b - \alpha \cdot \frac{dC}{db} \tag{5}$$

In the above functions α is a constant variable called learning rate, which determines the size of the steps. $\frac{dC}{dw}$ and $\frac{dC}{db}$ are the partial derivatives of cost function C with respect to weight w and bias b.

Backpropagation is the algorithm used to compute $\frac{dC}{dw}$ and $\frac{dC}{db}$. We will describe backpropagation by using our *xor* example from before, but instead of having a Neural Network already capable of emulating *xor* we have a fresh Network with randomly initialized weights and biases as seen in Figure 3.

Before we can get started on backpropagation We have to define our dataset and our cost function. The data set we define in Table IV with one data sample for each possible state, and for the cost function we will use the quadratic function in Equation 6,

$$C(N) = \frac{1}{2n} \sum_s ||y(s) - O^L(s)||^2 \tag{6}$$

Where L is the number of layers in the network, n is the total number of samples in our training data, the sum is done over individual training samples s, $y(s)$ is the desired

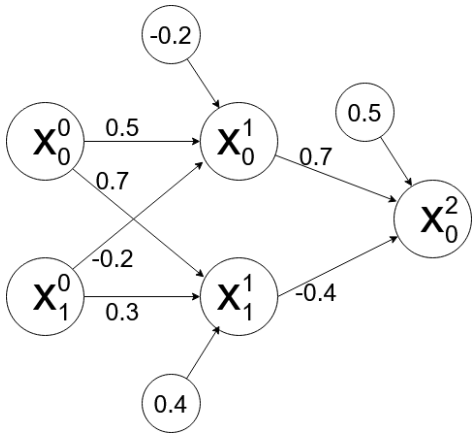


Figure 3: New xor neural network with random initialized weights.

output vector of the network for sample s, and $O^L(s)$ is the actual output vector of the network when given sample s.

The cost function in Equation 6 is the average cost for all data samples in the data set. For the purpose of backpropagation we want to rewrite this into $C(N) = \frac{1}{n}\sum_s C(N)_s$, where $C(N)_s$ is the cost for a single data sample, i.e. $C(N)_s = \frac{1}{2}||y(s) - O^L(s)||^2 = \frac{1}{2}\sum_i (y_i(s) - o_i^L(s))^2$. The reason we want the cost to be for single data sample is because backpropagation function by finding $\frac{dC_s}{dw}$ and $\frac{dC_s}{db}$ and then average those to get $\frac{dC}{dw}$ and $\frac{dC}{db}$[10].

In order to calculate $\frac{dC_s}{dw_{i,j}^l}$ and $\frac{dC_s}{db_i^l}$ we will first rewrite them into $\frac{dC_s}{dw_{i,j}^l} = \frac{d\lambda_i^l}{dw_{i,j}^l} \cdot \frac{dC_s}{d\lambda_i^l}$ and $\frac{dC_s}{db_i^l} = \frac{d\lambda_i^l}{db_i^l} \cdot \frac{dC_s}{d\lambda_i^l}$. The reason behind the rewrites is that it splits the derivatives into simpler derivatives that easier to calculate and allows for reuse of $\frac{dC_s}{d\lambda_i^l}$ when calculating the gradient of every weight and bias going into neuron x_i^l. The gradient $\frac{dC_s}{d\lambda_i^l}$ is often referred to as the error of neuron x_i^l and will from here on be denoted as δ_i^l for simplicity.

Since we are using the sum function in Equation 2 for λ then $\frac{d\lambda_i^l}{dw_{i,j}^l} = o_j^{l-1}$ and $\frac{d\lambda_i^l}{db_i^l} = 1$. The method for calculating δ_i^l depends on whether δ_i^l belongs to a hidden layer or the output layer. If δ_i^l belongs to the output layer, i.e. $l = L$ then $\delta_i^L = \frac{d\phi_i^L}{d\lambda_i^L} \cdot \frac{dC_s}{d\phi_i^L}$, if instead δ_i^l belongs to a hidden layer, i.e. $l \neq L$, then $\delta_i^l = \frac{d\phi_i^l}{d\lambda_i^l} \cdot \sum_j (w_{j,i}^{l+1} \cdot \delta_j^{l+1})$.

The way backpropagation work sis by first calculating the error for the neurons in the output layer, using Equation 7, these are then used to calculate the error of the neurons in the previous layer, as described in Equation 8, until all non-input neurons has been assigned an error. Once the error for each non-input neuron has been computed, then we can use them to compute the gradient of the weights and biases in the network, as seen

in Equation 9 and Equation 10.

$$\delta_i^L = \frac{d\phi_i^L}{d\lambda_i^L} \cdot \frac{dC_s}{d\phi_i^L} \tag{7}$$

$$\delta_i^l = \frac{d\phi_i^l}{d\lambda_i^l} \cdot \sum_j (w_{j,i}^{l+1} \cdot \delta_j^{l+1}) \tag{8}$$

$$\frac{dC_s}{dw_{i,j}^l} = o_j^{l-1} \cdot \delta_i^l \tag{9}$$

$$\frac{dC_s}{db_i^l} = \delta_i^l \tag{10}$$

Now that we have defined the data set and the necessary equations, then let us continue our example, focusing on the data sample s_0.

We First propagate the data and note the output of all neurons in Table V.

The next step is to compute δ_0^2, which we do based on Equation 7. We get the derivative of the sigmoid activation function from Table III, $\frac{d\phi_0^2}{d\lambda_0^2} = o_0^2(s_0) \cdot (1 - o_0^2(s_0))$, the derivative of our cost function becomes, $\frac{dC_s}{d\phi_0^2} = y(s_0) - o_0^2(s_0)$, we can then use these to compute the error of our output neuron as, $\delta_0^2 = (o_0^2(s_0) \cdot (1 - o_0^2(s_0))) \cdot (y(s_0) - o_0^2(s_0)) = (0.59 \cdot (1 - 0.59)) \cdot (1 - 0.59) = 0.1$.

We can now use Equation 8 to compute the errors of the neurons in our only hidden layer. The first step is to convert $\frac{d\phi_i^1}{d\lambda_i^1}$ into $1 - (o_i^1(s_0))^2$ as we are using the Tahn activation function for these neurons. We can then calculate the errors as $\delta_0^1(s_0) = (1 - (o_0^1(s_0)^2) \cdot (w_{0,0}^2 \cdot \delta_0^2) = (1 - 0.29^2) \cdot (0.7 \cdot 0.099) = 0.006$ and $\delta_1^1(s_0) = (1 - (o_1^1(s_0)^2) \cdot (w_{0,1}^2 \cdot \delta_0^2) = (1 - 0.8^2) \cdot (-0.4 \cdot 0.099) = -0.014$. Noting the results in Table V, as we will need them for the next step, which is to compute the gradients.

These computations can be seen in Table VI, where each row corresponds to the calculations for a single weight/bias. The functions used in the second column to calculate the gradient, is Equation 9 for weights and Equation 10 for biases.

The last step of backpropagation is to do all that we just did for data sample s_0, and also do it for the samples s_1, s_2, and s_3 and average the gradients. These results can be seen in Table VII.

Now that we have finished backpropagation and have computed $\frac{dC}{dw}$ and $\frac{dC}{db}$ for all weights and biases, then we are ready to use our gradient decent optimizer to update the weights and biases. We will do this based on Equation 4 for weights and Equation 5 for biases, both with the learning rate α set to 0.1. Table VIII shows the calculations for each weight and bias, as well as the new weights and biases.

For training a neural network, the cycle of backpropagation and optimization is repeated until a stop criteria is met. Common stop criterias are a certain number of epochs, a certain amount of time, or once the network stops improving significantly, i.e. the error does not decrease enough.

	I	y
s_0	$\{1,0\}$	1
s_1	$\{0,1\}$	1
s_2	$\{0,0\}$	0
s_3	$\{1,1\}$	0

Table IV: Xor data set, I is the input values, and y is the label.

	x_0^0	x_1^0	x_0^1	x_1^1	x_0^2
$o_i^l(s_0)$	1	0	0.29	0.8	0.59
δ_i^l	-	-	0.006	-0.014	0.01

Table V: The output and errors of the neurons with regard to data sample s_0.

	Calculation	Result
$w_{0,0}^1$	$1 \cdot 0.006$	0.006
$w_{0,1}^1$	$0 \cdot 0.006$	0
$w_{1,0}^1$	$1 \cdot -0.014$	-0.014
$w_{1,1}^1$	$0 \cdot -0.014$	0
$w_{0,0}^2$	$0.29 \cdot 0.01$	0.003
$w_{0,1}^2$	$0.8 \cdot 0.01$	0.008
b_0^1	0.006	0.006
b_1^1	-0.014	-0.014
b_0^2	0.01	0.01

Table VI: The gradient of all weights with regard to C_s for s_0, with the equations in the *calculation* column taken from Equation 9 and Equation 10.

	$\frac{dC_{s_0}}{dw}$	$\frac{dC_{s_1}}{dw}$	$\frac{dC_{s_2}}{dw}$	$\frac{dC_{s_3}}{dw}$	$\frac{dC}{dw}$
$w_{0,0}^1$	0.006	0	0	-0.094	-0.022
$w_{0,1}^1$	0	0.075	0	-0.094	-0.005
$w_{1,0}^1$	-0.014	0	0	0.011	-0.001
$w_{1,1}^1$	0	-0.032	0	0.011	-0.005
$w_{0,0}^2$	0.003	-0.048	0.026	-0.014	-0.008
$w_{0,1}^2$	0.008	0.075	-0.052	-0.121	-0.023
b_0^1	0.006	0.075	-0.092	-0.094	-0.026
b_1^1	-0.014	-0.032	0.047	0.011	0.003
b_0^2	0.01	0.125	-0.136	-0.136	-0.034

Table VII: Gradients of all weights and biases with regard to each data sample and the whole datasample.

	old weight	gradient decent function	new weight
$w^1_{0,0}$	0.5	$0.5 - 0.1 \cdot -0.022$	0.498
$w^1_{0,1}$	-0.2	$-0.2 - 0.1 \cdot -0.005$	-0.201
$w^1_{1,0}$	0.7	$0.7 - 0.1 \cdot -0.001$	0.700
$w^1_{1,1}$	0.3	$0.3 - 0.1 \cdot -0.005$	0.301
$w^2_{0,0}$	0.7	$0.7 - 0.1 \cdot -0.008$	0.701
$w^2_{0,1}$	-0.4	$-0.4 - 0.1 \cdot -0.023$	-0.398
b^1_0	-0.2	$-0.2 - 0.1 \cdot -0.026$	-0.197
b^1_1	0.4	$0.4 - 0.1 \cdot 0.003$	0.400
b^2_0	0.5	$0.5 - 0.1 \cdot -0.034$	0.503

Table VIII: The left column contains the initial weights, the middle column the gradient decent function with values, and the last column contains the new weights after one training step.

III. Euclidean Markov Decision Process

In this section we present the system model and controller synthesis objective that we use, both originate from [6].

Definition 1 ($\mathcal{K}$-Dimensional, Euclidean) Markov Decision Processes[6]) *A MDP is a tuple $\mathcal{M} = (\mathcal{S}, Act, s_{init}, T, \mathcal{C}, \mathcal{G})$ where:*

- *$\mathcal{S} \subseteq \mathbb{R}^{\mathcal{K}}$ represents all our possible states, as a bounded and closed subset of the Euclidean space,*
- *Act is a finite set of possible actions,*
- *$s_{init} \in \mathcal{S}$ is the initial state,*
- *$T : \mathcal{S} \times Act \to (\mathcal{S} \to \mathbb{R}_{\geq 0})$ is a probability density function over $\mathcal{S}$ i.e. $\forall (s, \alpha) \in \mathcal{S} \times Act$ then $\int_{t \in \mathcal{S}} T(s, a)(t)dt = 1$,*
- *$\mathcal{C} : \mathcal{S} \times Act \times \mathcal{S} \to \mathbb{R}$ is a cost-function that takes in a state-action-state triple, and returns the cost for moving from the first state to the second state through the action,*
- *$\mathcal{G} \subseteq \mathcal{S}$ is a set of goal states.*

We refer to a run of an MDP as π where π is a sequence of alternating states s and actions α, i.e. $\pi = s_1\alpha_1 s_2\alpha_2 \ldots$ where $s_1 = s_{init}$ and $T(s_i, \alpha_i)(s_{i+1}) > 0$ for all $i > 0$. The set of all runs of MDP $\mathcal{M}$ is denoted as $\Pi_{\mathcal{M}}$ and all finite runs of $\mathcal{M}$ as $\Pi^f_{\mathcal{M}}$. To indicate a run up to s_i we use the notation $\pi|i$, i.e. $\pi|i = s_1 a_1 s_2 a_2 \ldots a_{i-1} s_i$. $|\pi|$ denotes the length of run π such that if $\pi = s_1 a_1 s_2 a_2 \ldots s_n$ then $|\pi| = n$. We define the cost of π as the total cost until a goal state is reached. Let $s_{i_{min}}$ be the first state s in π where $s \in \mathcal{G}$, then we define the cost of π as $\mathcal{C}_{\mathcal{G}}(\pi)$, where $\mathcal{C}_{\mathcal{G}}(\pi)$ is defined in Equation 11[6].

$$\mathcal{C}_{\mathcal{G}}(\pi) = \sum_{s_i a_i s_{i+1} \in \pi | i_{min}} \mathcal{C}(s_i, a_i, s_{i+1}) \tag{11}$$

Definition 2 (Strategy[6]) *We define a strategy for a MDP as a function $\sigma : \mathcal{S} \to (Act \to [0, 1])$, which maps a $\mathcal{S}$ to a probability distribution over Act.*

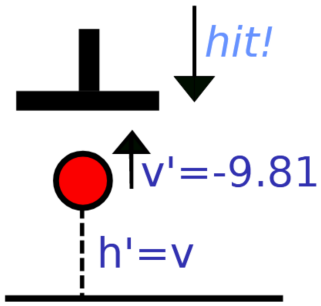


Figure 4: Bouncing Ball model, the red circle is the ball, the black figure is the piston which moves down when the action *hit*! is taken[6]

We evaluate strategies based on their expected cost to reach a goal state $\mathcal{G}$.

Definition 3 (Expected Cost of Strategy[6]) *Given a strategy σ over MDP M, then the expected cost of σ reaching G from a state s, is define by the following system of equations:*

$$\mathbb{E}_{\sigma}^{\mathcal{M}}(\mathcal{C}_{\mathcal{G}}, s) = \begin{cases} \mathbb{E}_{\sigma}^{\mathcal{M}}(\mathcal{C}_{\mathcal{G}}, s) = 0, & \text{if } s \in \mathcal{G} \\ sum_{\alpha \in Act} \sigma(s)(\alpha) \cdot \int_{t \in S} T(s, \alpha)(t) \cdot (\mathcal{C}(s, a, t) + \mathbb{E}_{\sigma}^{\mathcal{M}}(\mathcal{C}_{\mathcal{G}}, t))dt, & \text{if } s \notin \mathcal{G} \end{cases}$$

The problem we address is to find an optimal strategy σ.

Definition 4 (Optimal Strategy) *Strategy σ is an optimal strategy for MDP $\mathcal{M}$ if for any other σ':*

$$\mathbb{E}_{\sigma}^{\mathcal{M}}(\mathcal{C}_{\mathcal{G}}, s_{init}) \leq \mathbb{E}_{\sigma'}^{\mathcal{M}}(\mathcal{C}_{\mathcal{G}}, s_{init}) \tag{12}$$

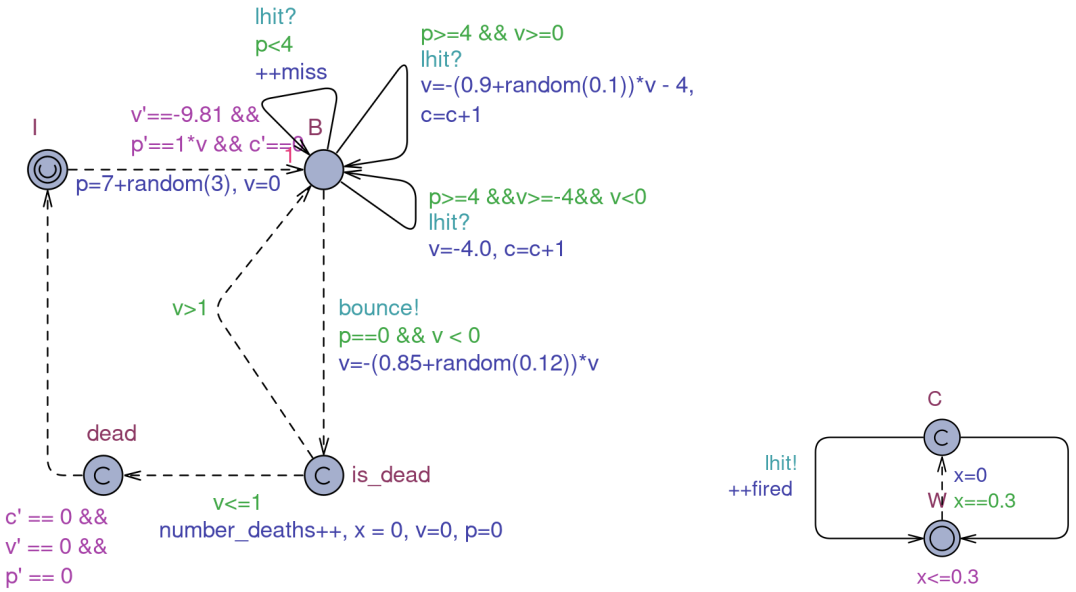
We denote this strategy as σ^*

A. Bouncing Ball Example

An example of a model that can be defined as an EMDP is the bouncing ball model from [6], which we will be using for our experiment.

The environment of this model consist of a ball and a piston, as seen in Figure 4. The goal of the agent in this model is use the piston to make sure that the ball keeps bouncing. It is worth noting that the piston can only hit the ball if it has a height above 4. We model this as a two dimensional state space $\mathcal{S} \subseteq \mathcal{R}^2$, where the first dimension $s^0 \subseteq \mathcal{R} \geq 0$ is the height of the ball and the second dimension $s^1 \subseteq \mathcal{R}$ is the vertical velocity of the ball. The action space $Act = \{a^0, a^1\}$ is of size two, where a^0 signifies that the piston is not activated while a^1 signifies the action that activates the piston. The cost function is as follows, if the piston is activated add one to the cost, if the ball is determined dead i.e. it will never be able to get within reach of the piston, add 1000 to the cost. This means that the goal is to avoid death states while activating the piston as few times as possible.

The transition function $\mathcal{T}$ is modeled in Uppaal Stratego with the models in Figure 5. Figure 5 (b) shows a model of the agent that we want to make a policy σ to control. The agent has two locations, an initial location marked *W*, and a control location marked *C*.



(a) (b)

Figure 5: Uppaal Stratego Models for Bouncing Ball 1, (a) models the environment (b) models the agent.

When the agent is in the control location it has to make a choice of either activating the piston and thus following the left edge to location W, or do nothing and thereby follow the right edge to location W. These choice correspond with action a^1 and a^0 in our action space Act, respectively. The agent has to make this decision once every 0.3 seconds, which is demonstrated by the guarded edge from W to C which must be done when $x = 0.3$. Note in this figure that when the agent takes the left edge from C then $lhit!$ is used to tell the environment that the piston is being activated.

To simulate the environment the model in Figure 5 (a) are used. In this model the variable p is the vertical position of the ball, v is the vertical velocity of the ball and c is a counter for how many times the piston has hit the ball. In the initial location I s_{ini} is set where the position of the ball is set to a random value between 7 and 10, and a velocity set to 0. From this location the environment moves over to location B. This is the primary location used to model the behaviour of the environment. While in this location the state of the ball is continuously updated, with the velocity of the ball being decreased with 9.81 every simulated second and the position of the ball being changed with regard to the velocity. This node has three loop edges, each representing a scenario for when the agent activates the piston. From left to the right the scenarios is as follows, the first scenario is for when the position of the ball is below 4, and thereby out of reach for the piston, in this scenario the simulation continues without the next state, i.e. p or v being affected by the activation. The next edge represents the scenario where the the ball has a positive velocity, and is within range of the piston, in this scenario the velocity of the ball

is adjusted to be downwards with a small random energy loss, and the kinetic energy of the piston is added to it, thereby accelerating the ball further downward. The last scenario is when the ball is already moving downwards and is within range of the piston, in this scenario the velocity of the ball is set to match the -4 velocity of the piston. The last two transitions define the impact our agent can have on the ball.

From location B there is also an edge going to an *is_dead* location which is taken every time the ball bounces and is responsible for checking if the ball died when it bounced, if the balls velocity post bounce is above 1 then we take an edge back location B where the simulation continues. If on the other hand that the balls velocity is below 1 then the ball is declared dead as it will no longer be able to get within reach of the piston. The model goes to the *dead* location and from there back into the initial location I thereby resetting the environment.

In [6] they address the problem of approximating an optimal strategy σ for models such as this bouncing ball example, through Q-learning and and finite partitioning of the state space $\mathcal{S}$. We instead propose to approximate the optimal strategy using NNs as they are universal function approximators[3] and can handle the continues nature of $\mathcal{S}$ without directly dividing into partitions.

IV. Deep Q-Network

For solving the problem of approximating the optimal strategy σ^*, we use the deep reinforcement learning method Deep Q-Network (DQN)[9].

The goal of DQN is to make an agent that interacts with an environment by selecting actions which minimizes future cost [9].

An assumption is made that future cost is discounted by a factor of γ per time step, i.e. the future discounted cost from time step t to the termination step T is defined as $C_t = \sum_{t'=t}^{T} \gamma^{t'-t} c_t$. We then define the optimal action-value function $Q^*(s,a)$, as the minimum expected C_t achievable by following any policy, when taking action a in state s. This function obeys the *Bellman equation*, which is based on the intuition that if the optimal value $Q^*(s_{t+1}, a')$ is known for all possible actions a', then the optimal policy is to select the action a' which minimizes the expected value of $c_t + Q^*(s_{t+1}, a)$[9].

The idea behind DQN is to use a neural network to define an action-value function $Q(s,a;N)$, and then make it approximate $Q^*(s,a)$ by using the equation as an iterative update for the network, i.e. $Q(s,a;N_{i+1}) = c_t + \gamma \min_{a'} Q(s_{t+1}, a'; N_i)$. This means that we can train the network by minimizing the following loss function sequences of loss functions $L_i(N_i)$, where $y_i = c_t + \gamma \min_{a'} Q(s_{t+1}, a'; N_i)$.

$$L_i(N_i) = (y_i - Q(s,a;N_i))^2 \tag{13}$$

It is worth noting that the target value, y_i is dependent on the network, in contrast to how supervised learning utilises a fixed predetermined target value. Differentiating the

loss function in Equation 13 with regard to the network, gives the following gradient:

$$\frac{dL_i}{dN_i}(N_i) = (c_t + \gamma min_{a'} Q(s_{t+1}, a'; N_i) - Q(s_t, a_t; N_i)) \cdot \frac{dQ}{dN_i}(s_t, a_t; N_i) \tag{14}$$

A. Training

When training DQN there are two key elements, the training policy, which is the policy used to select actions during training, and data sampling, which is how we store and sample data.

The training policy defines what action DQN takes during training. The training policy is responsible for balancing exploration and exploitation. To do this we are using an ϵ-greedy policy, which given a state will pick the estimated best action with a probability of $1 - \epsilon$, and has a probability of ϵ to pick a random action.

Data sampling defines how we create and use the data samples that we train our network on. For this we are using experience replay, which means that every time we take action a_t in state s_t, we observe the cost c_t and the resulting state s_{t+1} and store the experience, (s_t, a_t, c_t, s_{t+1}) in our experience replay memory $\mathcal{D}$. For optimizing our network we uniformly sample a minibatch of experiences from our replay memory, and use them to perform a gradient descent step, with Equation 13 as our loss function.

A full overview of how DQN function can be seen in algorithm 1, where line 1 and 2 initializes our memory $\mathcal{D}$ and then the neural network N. Line 5 to 7 represent our ϵ-greedy policy and the last lines describes how we sample and train based on our replay memory.

Algorithm 1: Deep Q-learning with Experience Replay[9]

1 Initialize replay memory $\mathcal{D}$ to capacity $|\mathcal{D}|$
2 Initialize Neural Network N_0 with random weights and biases
3 **for** $run = 0$ *to* R **do**
4 **for** $t = 1$ *to* T **do**
5 with probability ϵ select a random action a_t
6 otherwise select $a_t = min_a Q(s_t, a; N_i)$
7 Execute action a_t in emulator and observe state s_{t+1} and cost c_t
8 store experience (s_t, a_t, c_t, s_{t+1}) in $\mathcal{D}$
9 sample random minibatch of experiences (s_j, a_j, c_j, s_{j+1}) from $\mathcal{D}$
10 Set $y_j = \begin{cases} c_j, & \text{for terminal } s_{t+1} \\ c_j + \gamma min_{a'} Q(Q_{j+1}, a'; N_i), & \text{for non-terminal } s_{t+1} \end{cases}$
11 Update N_i to N_{i+1} by performing a gradient descent step on $(y_j - Q(s_j, a_j; N_i))^2$

Experience	$\{s_t, a_t, c_t s_{t+1}\}$
t_0	$\{(6.8, -7.5), a^0, 0, (4.3, -10.3)\}$
t_1	$\{(2.7, 6.2), a^1, 1, (4.8, 3.1)\}$
t_2	$\{(5.8, 2.4), a^1, 1, (4.6, -6.3)\}$

Table IX: Replay memory $\mathcal{D}$

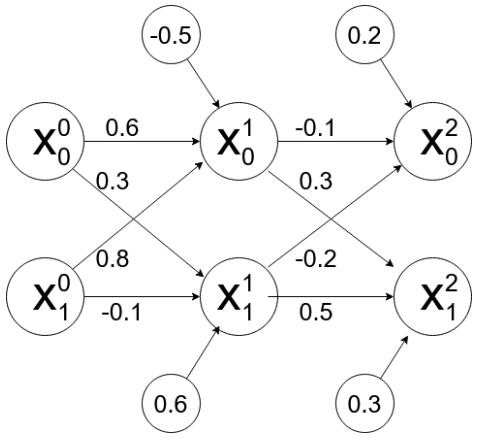


Figure 6: Neural Network used for DQN example, the two input node x_0^0 and x_1^0 corresponds to the state dimensions s^0 and s^1 respectively and the two output neurons x_0^2 and x_1^2 corresponds to the actions a^0 and a^1 respectively, for input functions the sum function is used by all neuron, for activation functions the hidden layer uses tanh and the output layer uses Linear.

B. Bouncing Ball Example

To illustrate how DQN functions let us do an example of a learning step using the Bouncing Ball model described in Section III-A. To do this we will define a replay memory is Table IX, and a Neural Network in Figure 6. The two input neurons of the network each represents one dimension in our state space, and the two output neurons each represents one of the actions in our action space. For activation function we use the tanh function in the hidden layer, and the linear function in the output layer.

We will in this example process the state $(4.1, -2.1)$. The first step will be to select whether to take a random action or an action decided by our network, as described in algorithm 1 line 5 and 6. Let us continue with the action being selected by out network. We then have to propagate the state through the network, which gives the following output vector $\binom{-0.02}{0.87}$. In accordance with line 7 of the algorithm we then tells the environment to take action a^0 as it has the lowest estimated cost. The environment will then return the next state $(3.4, -5.0)$. We then combine the initial state the chosen action and the resulting state into a transition and store it in $\mathcal{D}$ as follows:

$$\mathcal{D} = \mathcal{D} \uplus \{(4.1, -2.1), a^0, 0, (3.4, -5.0)\} \tag{15}$$

Now that a new transition has been stored in the memory, we make a minibatch of stored transitions and update our network based on those. We will be using a batch size of one for simplicity. With a uniform chance between our four experiences, the original 3 and the new one, we sample the experience $t_1 = \{(2.7, 6.2), a^1, 1, (4.8, 3.1)\}$. We first calculate the loss, it is worth noting that since the action taken in experience t_1 is a^1, then we only consider the loss with regard to that action. The first step of calculating the loss is to calculate the target value y for neuron x_1^2, as this experience does not end in a terminal state. Which means that the target value is equal to c_t plus the discounted estimated future cost:

$$y = 1 + \gamma \cdot \min_{a'} Q(\{4.8, 3.1\}, a'; N) = 1 + 0.9 \cdot -0.09 = 0.92$$

We then compute the cost estimate for action a^1, which through propagation equals 0.43, we therefore has the following loss:

$$L_i(N) = (y - Q((2.7, 6.2), a^1; N))^2 = (0.92 - 0.43)^2 = 0.24$$

The last step is then to back propagate the loss and update the Network through gradient descent, as described in Section II-C.

C. Prioritized Experience Replay

One of the important elements of DQN is experience replay method, i.e. how we sample our memories, currently we have only discussed uniform sampling where each memory has an equal chance of being sampled when training. However, this assumes that all experience has can contribute equally to the learning of the NN, while this is not always the case, e.g. the ball dying in bouncing ball is an important experience has avoiding this is the main goal, but is also a rare experience meaning it will only fill a small portion of the experience replay buffer. In [11] they introduce methods to prioritize which experiences are sampled for replay during training. These methods are prioritized based on TD-error, which DQN already calculates as our loss over a single experience. One of the prioritization methods in [11] is *rank-based prioritization*, which prioritizes relative to the rank of an experience in experience replay buffer sorted on the TD-error. The priority of experience i is calculated in accordance to Equation 16, where $rank(i)$ is the rank of experience i in the sorted experience replay buffer, such that the experience with the highest TD-error has $rank(i) = 1$.

$$p_i = \frac{1}{rank(i)} \tag{16}$$

This priority p_i is then used to calculate the probability that experience i is sampled during learning. This probability is computed based on Equation 17, where ρ is used to determine how much prioritization impacts the sampling, with $\rho = 0$ being equivalent to uniform sampling

$$P(i) = \frac{p_i^\rho}{\sum_k p_k^\rho} \tag{17}$$

Using this prioritizes memory introduces a bias, which when we get close to convergence we would like to avoid. To achieve this [11] introduces importance-sampling weights, which are used to correct for the bias. The importance sampling is calculated in Equation 18, where β determines how much importance sampling compensates for the bias, with $\beta = 1$ fully compensating for the bias.

$$v_i = (\frac{1}{N} \cdot \frac{1}{P(i)})^{\beta} \tag{18}$$

v_i is used under training such that the loss with regard to experience i is weighted with regard to v_i, thereby weighting the impact experience i has on the training update.

As mentioned we are mainly interested in this bias compensation when nearing convergence, as we are interested in this bias to early on learn from significant but rare experiences. To accomplish this β is linearly changed from its initial value β_0 to 1 over the cause of training.

V. Experimental Setup

To test DQN's ability to find optimal strategies for MDPs, we will use the experimental setup form [6] and compare DQN against the Q-learning methods proposed in that paper. This setup consists training and comparing the methods on several case studies, all but one derived from one of three scalable models, the last one derived from XOR.

To test DQN's ability to find optimal strategies for MDPs, we conduct two series experiments. In the first series of experiments we test a variety of hyperparameter setup configurations for DQN on simple case studies from [6] after 250 runs. The goal of this series experiments is to study which hyperparameters have the biggest impact on the strategies DQN can learn for a given case study, and to find the best hyperparameter configuration for each model to use in the second series experiments. In the second series of experiments we conduct an in depth comparison between the best performing DQN configuration for each model from the first series experiments against the Q-learning method proposed in [6]. These experiments consist of comparing the strategies that the models create for several case studies, all but one derived from one of the three scalable models in [6], with the last one being derived from XOR. These models are as follows:

- **Bouncing Ball:**
  - In the Bouncing Ball model the goal is to keep N balls bouncing by activating a piston to hit them. Each activation of the piston has a unit of cost associated with it, and only affects balls above a certain height, as describe in Section III-A.
- **Floor Heating:**
  - The second model is a modified version of the Floor Heating case study from [8], where the outdoor temperature measurements/predictions are replaced with a simple sinusoidal curve.
- **Highway:**

  - For the last model a set of different scenarios for autonomous vehicles on a highway has been modeled. The goal in this model is to control a single vehicle while avoiding collisions[6].

VI. Hyper Parameter Experiments

When using DQN then there are several parameters that define how the method behaves. In this series of experiments we will divide these parameters into two different categores, constant parameters which are set to the same value for all setups, and varying parameters, that for each setup, are given a value from a pre-determined set of possible values.

The constant parameters consist of:

- ϵ for our ϵ-greedy policy,
- γ future discount for cost,
- batch size,
- ρ for prioritized replay,
- Neuron input function,
- Neuron activation function, and
- Optimizier.

For ϵ it starts at 1.0 and then linearly degrades down to 0.1 during the first 10% of the allocated training budget, after which it stays at 0.1. The future discount modifier γ is set to 0.9. The batch size is set to 32. For prioritized experience replay ρ is set to 0.7 as it was found to be best for rank-based prioritization in [11]. Our input and activation functions for our neurons, batch size, and our optimizer. For our input function we use the sum function and for the activation function we use the Tanh function for the hidden layers and the linear function for our output layer. The optimizer we use is the RMSprop[2] as it was used in the original DQN paper[9].

The varying parameters consists of the following:

- Number Of Hidden Layers $\in \{1, 2\}$
- Size of Hidden Layers $\in \{2, 4, 8, 16, 32, 64, 128\}$
- Learning Rate $\alpha \in \{0.005, 0.01, 0.05, 0.1\}$
- Memory Size $|\mathcal{D}| \in \{10.000, 100.000, 1.000.000\}$
- Memory Type $\in \{\mathit{Uniform},\ \mathit{Prioritised}\}$
  - For prioritised we also search for $\beta \in \{-1, 0.0, 0.2, 0.4\}$
- Normalisation $\in \{\top, \bot\}$
- Learning type $\in \{\mathit{online},\ \mathit{semi\text{-}offline}\}$

These parameters refer either directly to the network structure or the learning parameters that has been explained in Section II for number of hidden layers, size of hidden layer, learning rate or in Section IV for memory size, memory type, and $\beta >= 0$. We here only explain what $\beta = -1$, Normalisation, and Learning type. We use $\beta = -1$ to denote when we disable the bias correction. If Normalisation = $\top$, then all input variable and costs are normalised based on the highest observed value for the input and cost respectively after 5

runs. The learning type, describes when we sample and train the network. When learning type = *online*, then we take one training step after each interaction with the environment, as described in algorithm 1. When instead learning type = *semi-offline* then each run is done without updating the network, DQN is then fed the experience in a reverse order, i.e. from the termination state to the initial state, where it takes one training step after each state, action, state tuple it is fed.

To determine the values of our varying parameters we will conduct a test on the simplest case study from each model with a training budget of 250 runs and 25 repetitions. The exception is the highway model, where we increased the complexity of the model two times from `1car` to `3car overtake` to `4car overtake5`, as multiple setups was capable of finding the optimal strategy for both `1car` and `3car overtake` after only 250 runs.

A. Hyperparameter Results

Figure 7 shows the results of our hyper parameter Experiments. To create these graphs we sorted all configuration, on the median of their cost of a run after the 25 repitions, which is what the blue shaded are represents, with the Y-axis denoting the cost and the sorted list of configurations being along the X-axis. The red line shows Q-learnings median cost after 250 runs, and the green line shows its median cost of 10000 runs. The yellow vertical lines denotes points of interest we will go over.

When looking at Figure 7, then there are a few interesting observations to note. For `Bouncing Ball 1`, then all setups that got an average cost below 399, which is the cost for always choosing hit, and is marked by the yellow vertical line, had normalisation = $\top$, Memory Type = *Prioritised*, and the vast majority only used one hidden layer. We speculate that the reason why normalisation is important is due to the high cost value associated with the ball dying, as this will mean that our gradient will be steep and cause big changes to the weights of our network. Thus, making it difficult to develop a more fine tuned policy. We hypothesis that the reason the prioritised memory is needed, is that it allows the network to more quickly find and learn on the rare edge cases, such as when a ball die. As the network is more quickly updated to reflect when a ball dies, it is also more likely to properly trace the reason for the death back to the actual time step where it should have hit, thus allowing for a more fine tuned policy to emerge.

For `Floor heating1_5`, the right side of yellow vertical line consist only of setups where normalisation = $\bot$, and the 14 best setups, including the only 4 that outperformed Q-learning after 250 runs, all had a network size of 2 hidden layers with 128 neurons in each. The reason for the size is likely due to the high input/output dimension and the complex function between them. Which means that the network needs a certain level complexity to properly map the correlations. For the normalisation it is interesting that even though floor heating has a higher cost value than bouncing ball, then in contrast to bouncing ball, it needs to have it disabled. This is likely an artefact of the normalization which normalises an input variable based on the observed values of all input variables, rather than normalising based only on the observed values for the given

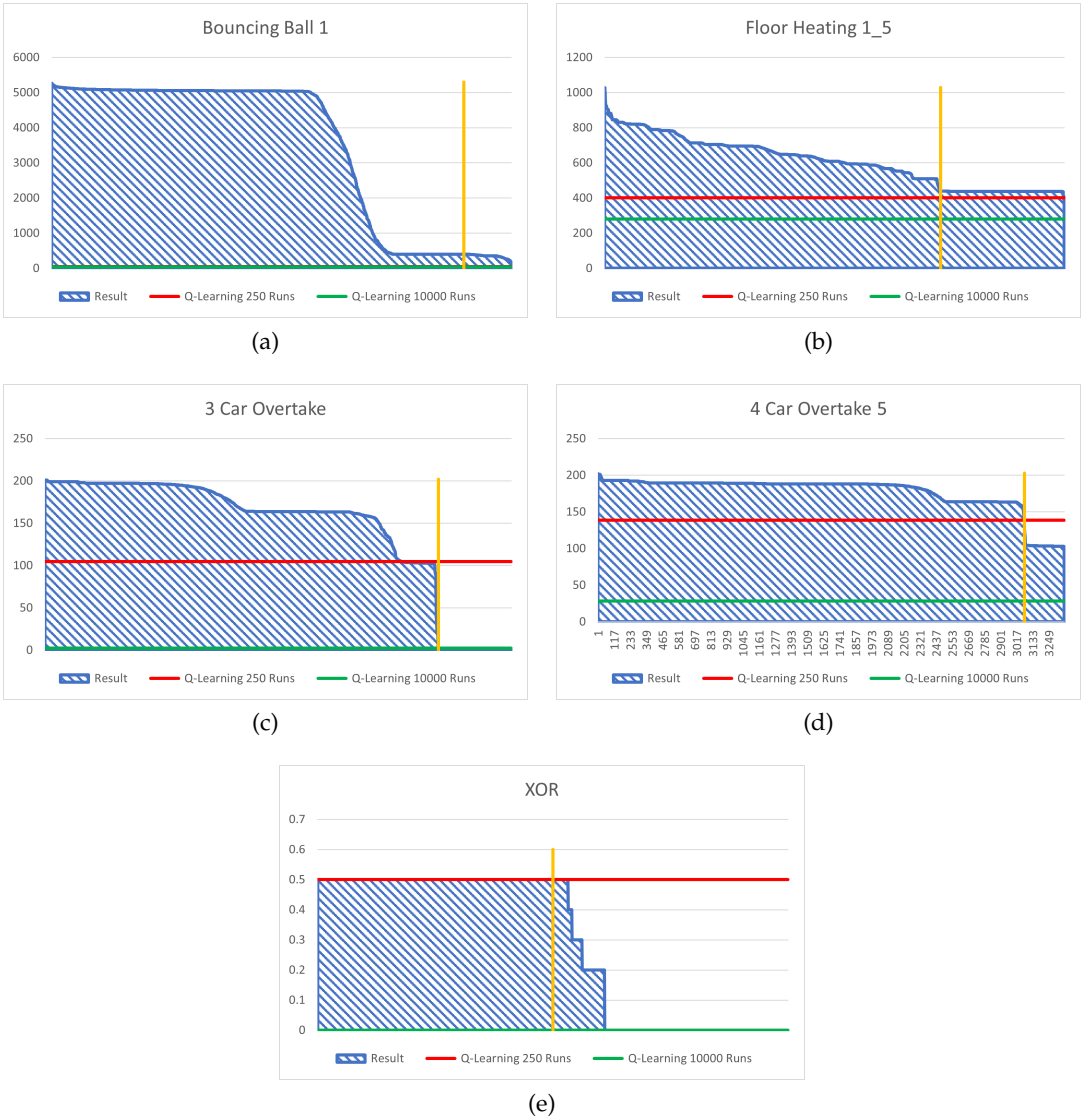


Figure 7: Hyper Parameter Results

input variable. This can be a problem nn floor heating there as there is a variable that tracks time in seconds, the value of this variable greatly out range the possible values of any other variable, thus when normalising, it is possible that the other values becomes too small to have any significant impact on the network. More experiments with a different normalisation method could test this hypothesis.

For highway, it is worth noting that out off the 527 setups that solved `3car overtake`, marked by the yellow vertical line, only 16 of them used Memory Type = *Unitform*, a trend that is also reflected in the group of setups that outperformed Q-learning after 250 runs

in `4car overtake5`, also marked by the yellow vertical line. This is likely due to the cost function, which only has two cost for state-action-state triple, one if the car crashed and one if it did not crash. Therefore, learning on or near the experience where the car crashed are crucial for learning good strategies on the model. However each run has at most one experience of a crash, thereby making it a relatively rare occurrence. Which could explain why uniform sampling has difficulties learning good strategies for highway cases.

For `XOR`, the deciding factor appear to be whether the data is normalised. This is evident by all configurations where normalisation = $\bot$, being to the left of the yellow vertical line with an average cost of 0.5, which corresponds to random play.

From this series of experiments we will take the best performing configuration from each model and use in the comparison experiments. The configurations we use are as follows:

- **Configuration 1 (1x16), derived from `Bouncing Ball 1`:**
  - Number of Hidden Layers = 1
  - Size of Hidden Layers = 16
  - $\alpha = 0.01$
  - $|\mathcal{D}| = 100.000$
  - Memory Type = *Prioritised*
  - $\beta = 0.4$,
  - Learning type = *semi-offline*
  - normalisation = $\top$.
- **Configuration 2 (2x2), derived from `4car overtake5`:**
  - Number of Hidden Layers = 2
  - Size of Hidden Layers = 2
  - $\alpha = 0.005$
  - $|\mathcal{D}| = 100.000$
  - Memory Type = *Prioritised*
  - $\beta = 0.0$,
  - Learning type = *semi-offline*
  - normalisation = $\bot$.
- **Configuration 3 (2x128), derived from `floor heating 1_5`:**
  - Number of Hidden Layers = 2
  - Size of Hidden Layers = 128
  - $\alpha = 0.005$
  - $|\mathcal{D}| = 1.000.000$
  - Memory Type = *Prioritised*
  - $\beta = 0.4$,
  - Learning type = *online*
  - normalisation = $\bot$.

VII. Comparison Experiments

In our comparison experiments we compare the three DQN configurations found in Section VI against the Q-learning method from [6]. These experiments are done over the several case studies from [6], and a model of XOR. For each case study we test the strategy that the methods can create after utilising a training budget ranging from 100 runs to 10000 runs, each experiment is repeated 25 times as to accommodate for stochastic nature of the experiments.

A. Comparison Results

We here present the results of our comparison experiments. They are presented as the 25% quantile, the 50% quantile of the 25 repetitions of each experiment. The results of all the Bouncing Ball, and Floor Heating experiments as well as a selected subset of the Highway experiments can be found in Table X. The remaining Highway results can be found in Appendix A. In these tables, Q, denotes the results of the Q-learning method, while the DQN configurations are named based on their network size, e.g. $1x16$ is the DQN configuration with one hidden layer of size 16.

The results show that for XOR both Q-learning and setup $1x16$ are capable of solving it, though configuration $1x16$ does this much faster than Q-learning, after only 250 runs, while Q-learning show little sign of improvement prior to 5000 runs.

For Bouncing Ball Q-learning outperforms all setups of DQN, with the best performing DQN configuration always being $1x16$, though noticeably not always after 10000 runs. When studying the Bouncing Ball results for the DQN configurations, then a few interesting observations can be made. The first being that for `Bouncing Ball 2` and `Bouncing Ball 3` the configuration $2x2$ always has a median value of 399 and 25% quantile value of no more than 391.2. This interesting as the 399 mark denotes the strategy of always choosing the hit action. This means that for this configuration the network quickly learns to be afraid of dropping the ball, but it appears to be almost impossible for it to improve upon this strategy. One possible explanation for why it cannot improve might be the high cost associated with a non-normalised costs in this model, which means that the changes in weights can often be relative big, thereby making it hard to take the small learning steps necessary to find a fine tuned strategy. A second interesting observation is the small difference between the best achieved strategy by DQN for `Bouncing Ball 2` and `Bouncing Ball 3`. For `Bouncing Ball 2` this strategy is achieved after 5000 runs with a median cost of 326.6, while for `Bouncing Ball 3` a similar median cost of 325.9 is achieved after 10000 runs. This is in contrast to Q-learning where the difference in cost is much higher with 80.2 for `Bouncing Ball 2` and 136.8 for `Bouncing Ball 3`. This could indicate that DQN are better at handling the scaling issue of this task. However, a simpler explanation could also be that since both the best performing strategies for DQN is relative close to the always hitting strategy with a cost of 399, then it simply does not require much to improve to that state for either case study, and DQN might notice similar scaling issues as Q-learning if it achieves a sufficiently well performing strategy.

Runs	100		250		500		1000		2500		5000		10000	
Conf:	25%	50%	25%	50%	25%	50%	25%	50%	25%	50%	25%	50%	25%	50%
XOR:														
Q	0.5	0.5	0.4	0.5	0.5	0.5	0.5	0.5	0.0	0.4	0.0	0.0	0.0	0.0
1x16	0.0	0.2	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
2x2	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
2x128	0.5	0.5	0.5	0.5	0.5	0.5	0.4	0.5	0.5	0.5	0.4	0.5	0.5	0.5
Bouncing Ball 1:														
Q	74.2	88.2	44.3	49.4	42.7	46.0	42.4	50.3	39.6	41.6	40.2	41.9	39.6	39.7
1x16	106.4	1198.4	135.6	842.7	83.6	194.2	101.4	108.5	88.9	131.3	159.4	227.9	149.6	452.4
2x2	399.0	5046.4	5053.0	5063.8	399.0	5046.0	5053.0	5062.0	399.0	5056.0	5042.0	5056.0	399.0	5041.0
2x128	5058.0	5078.2	5053.0	5068.1	5044.0	5065.2	5058.0	5078.9	5048.0	5053.0	5057.8	5061.6	5061.0	5061.0
Bouncing Ball 2:														
Q	182.2	250.6	114.9	183.9	96.8	241.8	92.0	137.5	93.2	117.5	78.3	95.1	70.4	80.2
1x16	353.3	695.7	323.4	2509.7	258.1	810.8	210.5	786.1	167.2	817.6	216.0	326.6	303.3	374.3
2x2	391.2	399.0	397.4	399.0	399.0	399.0	399.0	399.0	399.0	399.0	399.0	399.0	399.0	399.0
2x128	5110.2	6676.4	4101.0	7082.3	6130.3	7969.3	6248.6	8226.0	9598.4	9636.2	5167.7	7634.9	9547.6	9547.6
Bouncing Ball 3:														
Q	217.8	331.5	240.9	505.4	188.1	313.5	180.1	218.5	144.0	239.4	117.7	149.0	108.5	136.8
1x16	332.6	951.9	193.9	857.0	301.3	811.6	226.0	443.5	298.3	526.7	266.1	345.7	274.6	325.9
2x2	399.0	399.0	399.0	399.0	399.0	399.0	399.0	399.0	399.0	399.0	399.0	399.0	399.0	399.0
2x128	350.8	556.4	450.0	4506.3	4914.3	7195.0	7720.3	11070.8	11339.6	12449.2	7786.6	10095.4	14225.2	14225.2
Floor Heating 1_5:														
Q	490.5	567.9	361.3	402.0	327.7	405.1	285.5	308.6	284.3	288.0	275.9	286.9	273.1	280.7
1x16	437.5	437.7	437.4	437.9	437.0	440.1	348.4	428.0	284.4	314.1	277.3	309.4	270.7	279.9
2x2	638.7	845.1	510.5	648.3	437.2	437.7	437.3	437.6	437.6	437.9	438.2	567.2	510.5	567.8
2x128	437.5	437.9	369.9	424.8	319.4	327.7	302.7	324.0	294.5	305.2	282.6	316.9	292.1	317.4
Floor Heating 6_11:														
Q	867.6	1305.2	645.4	731.8	440.3	536.6	346.7	408.2	291.4	305.3	287.3	292.5	276.4	286.9
1x16	737.0	1221.8	640.4	1157.2	599.1	612.2	530.3	530.8	-	-	-	-	-	-
2x2	642.2	862.4	912.8	1179.0	598.9	1145.9	658.9	991.1	-	-	-	-	-	-
2x128	612.5	861.7	530.5	586.9	530.8	585.4	530.9	592.1	481.6	530.6	434.7	530.9	316.4	392.1
3 Car Overtake:														
Q	0.0	12.2	0.1	45.6	0.0	0.0	0.0	4.0	3.0	5.7	1.2	3.8	1.3	2.5
1x16	58.5	111.4	20.4	61.5	1.4	3.4	0.1	18.3	0.3	1.5	0.2	2.3	0.0	0.4
2x2	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
2x128	152.6	163.5	103.1	195.9	105.9	189.3	109.1	163.3	103.9	196.7	102.3	163.6	101.8	163.7
4 Car Overtake 3:														
Q	113.5	146.5	61.8	104.7	13.1	45.6	21.7	52.5	20.4	43.5	11.0	15.3	3.8	6.4
1x16	113.3	125.2	36.4	67.8	2.6	19.5	10.3	36.0	21.3	59.9	5.2	11.1	3.4	16.9
2x2	103.1	103.5	102.6	103.0	102.9	103.3	102.8	103.1	102.7	103.4	102.9	103.3	102.5	102.8
2x128	173.7	189.4	164.2	197.0	189.3	198.9	163.6	183.7	104.3	163.4	103.6	163.5	163.6	189.3
4 Car Overtake 5:														
Q	116.3	174.0	114.6	138.7	96.0	110.1	98.4	113.9	44.1	57.9	30.1	40.5	18.1	28.4
1x16	142.7	157.0	46.1	122.5	26.3	31.2	10.9	53.2	22.6	61.9	11.0	50.4	3.0	9.5
2x2	102.2	102.7	102.5	102.8	102.4	103.2	102.3	102.7	103.0	103.7	103.0	163.5	102.6	102.9
2x128	163.6	187.2	163.6	188.0	103.3	188.0	163.7	189.4	163.5	188.0	103.3	188.0	163.4	188.0

Table X: Results from XOR, all the Bouncing Ball, all Floorheating experiments, and a selected subset of the Highway experiment results

For `floor heating 1_5`, Q-learning and DQN configuration $1x16$ performs similarly well, though it should be noted that configuration $2x128$ outperforms $1x16$ until it starts to worsen in performance after 2500 runs. So if it had been able to continue to improve it might have outperformed both configuration $1x16$ and Q-learning.

For `floor heating 6_11`, Q-learning significantly outperforms all DQN configurations, the missing values for configuration $1x16$ and $2x2$ is due to error in the code causing the program to crash. An interesting observation for the two floor heating cases, is that Q-learning does not appear to be suffer a significant penalty in the cost after 2500 when scaling from `floor heating 1_5` to `floor heating 6_11`, which is in contrast to configuration $2x128$.

Generally the right DQN configuration seems to be on pair with or outperform Q-learning for most highway cases, with the exception being `4car overtake3`. More specifically, for `3car overtake` Q-learning solves it after 500 runs, but then seems to degress in performance as it after 10000 runs has a median cost of 2.5, while configuration $2x2$ finds the optimal strategy already after 100 runs and does not degress. For `4car overtake5` performs significantly better than Q-learning, with Q-learning best achieved median cost being 28.4, while the $1x16$ configuration achieves a median cost of 9.5, both after 1000 runs. Finally for `4car overtake3` Q-learning outperforms the best DQN configuration. However, with a noticeably smaller margin than the right DQN configuration outperforms Q-learning with in `4car overtake5`. Overall our highway results indicates that for this model, the right configuration of DQN hyperparameters are capable of outperforming Q-learning, but if the hyperparameters are not optimal then DQN performs significantly worse. An example of this is for both `4car overtake3` and `4car overtake5` neither configuration $2x2$ or $2x128$ manage to get bellow a cost of 100.

One observation that can be seen in several cases is that DQN seems prone to start degress significantly in performance as the training budget increases. One example of this is `Bouncing Ball 1` configuration $1x16$, which with a training budget of 1000 runs achieve a median average cost of 108.5 which then steadily increases to 452.4 at 10000 runs. This is notably above the 399 mark, which is the always hit strategy. This means that the strategy has started to drop the ball sometimes. In [1] they found similar behaviour of a deep reinforcement learning method and tested the cause of it. Through extensive experimental testing, they concluded that the reason behind this is duo to old memories being replaced with new ones which are less diverse as our ϵ has decreased and our policy has settled. This means that as the network trains on these samples, which has a limited coverage of the state-action space, it forgets the knowledge that it obtained through the early samples which had more exploration. Furthermore, [1] illustrated that since our exploration has decreased significantly at this point, then the algorithm does not explore enough to create a diverse enough experience replay memory to recover from this loss in performance.

Overall our results shows that within the limitation of our DQN configurations Q-learning significantly outperforms any tested DQN configuration on all bouncing ball cases and `floor heating 6_11`, while the right configuration of DQN is on pair with Q-learning for `floor heatin 1_5`. For the highway model, the right configuration of DQN is either on pair with Q-learning if they both solves it, is better than Q-learning, or in the case of `4car overtake3` and `4car overtake2` performs slightly worse.

Furthermore the Q-learning method appears to continue to converge to a better solution as run budget is increased, while our current experience replay memory method prevents this behaviour for DQN, aligning with the observations of [1].

VIII. Conclusion

In this paper we hypothesised that by using NNs we could improve upon the current state of art methods used to find strategies in Uppaal Stratego.

We tested this hypothesis by first sweeping over a set of hyperparameters, selecting three configurations of hyperparameters and then do an extensive comparison of them against the Q-learning method from [6].

The results of our experiments shows that within the limitations of our hyperparameter selection, we were not able to find a single set of hyperparameters suiting all problem domains. However, if the right set of hyperparameters is found DQN does appear able to find the solution within less runs, as it did in XOR where the right DQN configuration solved it within 250 runs, where as Q-learning needed up to 5000 runs. DQN can also with the right hyperparameters find better strategies, as was the case for the highway model. Lastly similarly to [1] we aw that a better method is needed to decide which memories we keep in the experience replay buffer and which are forgotten, in order to keep diversity in the experience replay buffer, as without it DQN seems to forget what it learned early on.

IX. Future Work

Two issues that this investigation has revealed when using Neural Networks to find strategies in Uppaal Stratego is firstly setting the right hyperparameters, and secondly overcoming diversity loss in our experience replay memory as our training budget increases. We therefore suggest the following, finding and implementing automated hyperparameter search that is capable of finding both simple parameters such as learning rate and more complex parameters such as network structure. Finding and implementing a method that decide which memories to keep in our experience replay memory, such that that network does not forget the knowledge gained early on.

References

[1] Tim de Bruin et al. "The importance of experience replay database composition in deep reinforcement learning". In: Jan. 2015 (cit. on pp. 23, 24).

[2] Vitaly Bushaev. *Understanding RMSprop — faster neural network learning*. 2018. URL: https://towardsdatascience.com/understanding-rmsprop-faster-neural-network-learning-62e116fcf29a (visited on 05/20/2021) (cit. on p. 17).

[3] Balázs Csanád Csáji. "Approximation withArtificial Neural Networks". In: (2001) (cit. on pp. 3, 12).

[4] Alexandre David et al. "On Time with Minimal Expected Cost!" In: *Automated Technology for Verification and Analysis*. Ed. by Franck Cassez and Jean-François Raskin. Cham: Springer International Publishing, 2014, pp. 129–145. ISBN: 978-3-319-11936-6 (cit. on p. 1).

[5] Alexandre David et al. "Uppaal Stratego". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Christel Baier and Cesare Tinelli. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 206–211. ISBN: 978-3-662-46681-0 (cit. on p. 1).

[6] Manfred Jaeger et al. "Teaching Stratego to Play Ball: Optimal Synthesis for Continuous Space MDPs". In: *Automated Technology for Verification and Analysis*. Ed. by Yu-Fang Chen, Chih-Hong Cheng, and Javier Esparza. Cham: Springer International Publishing, 2019, pp. 81–97. ISBN: 978-3-030-31784-3 (cit. on pp. 1, 2, 9, 10, 12, 16, 17, 21, 24).

[7] Shyam Lal Karra et al. "Safe and Time-Optimal Control for Railway Games". In: *Reliability, Safety, and Security of Railway Systems. Modelling, Analysis, Verification, and Certification*. Ed. by Simon Collart-Dutilleul, Thierry Lecomte, and Alexander Romanovsky. Cham: Springer International Publishing, 2019, pp. 106–122. ISBN: 978-3-030-18744-6 (cit. on p. 1).

[8] Kim G. Larsen et al. "Online and Compositional Learning of Controllers with Application to Floor Heating". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Marsha Chechik and Jean-François Raskin. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 244–259. ISBN: 978-3-662-49674-9 (cit. on p. 16).

[9] Volodymyr Mnih et al. *Playing Atari with Deep Reinforcement Learning*. 2013. arXiv: 1312.5602 [cs.LG] (cit. on pp. 1, 12, 13, 17).

[10] Michael Nielsen. *How the backpropagation algorithm works*. Apr. 2021. URL: http://neuralnetworksanddeeplearning.com/chap2.html (cit. on p. 6).

[11] Tom Schaul et al. *Prioritized Experience Replay*. 2016. arXiv: 1511.05952 [cs.LG] (cit. on pp. 15–17).

[12] Kun Shao et al. *A Survey of Deep Reinforcement Learning in Video Games*. 2019. arXiv: 1912.10944 [cs.MA] (cit. on p. 1).

[13] Erik Ramsgaard Wognsen et al. "A Score Function for Optimizing the Cycle-Life of Battery-Powered Embedded Systems". In: *Formal Modeling and Analysis of Timed Systems*. Ed. by Sriram Sankaranarayanan and Enrico Vicario. Cham: Springer International Publishing, 2015, pp. 305–320. ISBN: 978-3-319-22975-1 (cit. on p. 1).

Appendix A
Highway Results

Runs	100		250		500		1000		2500		5000		10000	
Conf:	25%	50%	25%	50%	25%	50%	25%	50%	25%	50%	25%	50%	25%	50%
1 Car														
Q	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
1x16	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
2x2	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
2x128	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	188.0	0.0	0.0	0.0	0.0
2 Car Fast Back:														
Q	0.0	2.1	0.0	0.5	1.8	35.3	0.2	4.4	1.0	5.6	0.2	1.8	0.8	1.2
1x16	7.0	142.8	0.0	5.6	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.7	0.0	0.0
2x2	0.3	0.4	0.3	0.4	0.3	0.3	0.3	0.4	0.2	0.4	0.1	0.4	0.3	0.3
2x128	132.5	191.5	0.3	188.0	0.4	191.5	0.2	0.5	0.0	0.7	0.0	0.3	0.0	0.0
2 Car Slow Front														
Q	0.0	157.6	0.0	0.0	0.0	0.0	0.0	1.5	0.0	3.7	0.0	1.2	1.1	1.9
1x16	5.3	133.1	0.5	15.4	0.0	0.4	0.0	0.1	0.0	1.8	0.2	3.8	0.5	13.9
2x2	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
2x128	157.0	157.6	0.0	178.1	0.0	157.2	0.0	178.2	0.0	157.4	0.0	157.6	0.0	157.6
2 Car Overtake:														
Q	0.0	0.8	0.0	0.0	0.0	2.8	0.1	4.4	1.9	4.2	2.0	3.9	0.4	0.6
1x16	2.5	104.8	0.0	1.8	0.0	0.3	0.0	0.0	0.0	0.0	0.1	25.1	0.0	0.7
2x2	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
2x128	179.7	188.0	0.0	188.0	0.0	44.9	41.6	188.0	0.0	22.6	0.0	0.0	0.0	0.0
3 Car:														
Q	65.6	153.4	73.3	111.2	43.0	71.1	16.5	29.5	6.7	9.7	6.6	13.5	5.9	6.5
1x16	95.4	136.4	22.6	102.3	1.1	10.1	1.6	11.3	7.5	27.8	2.2	10.5	1.9	5.6
2x2	102.3	103.1	102.5	103.1	102.4	103.0	102.6	103.0	102.7	103.0	102.8	104.1	102.7	102.8
2x128	163.5	186.5	163.3	189.3	103.1	188.0	163.6	189.3	163.8	189.4	102.9	163.4	104.2	189.3
3 Car Slow Front:														
Q	125.0	170.0	44.0	121.0	93.0	125.0	54.0	109.0	0.0	0.0	0.0	0.0	0.0	0.0
1x16	68.0	137.0	45.0	84.0	0.0	48.0	0.0	37.0	0.0	0.0	0.0	0.0	0.0	37.0
2x2	125.0	125.0	125.0	125.0	125.0	125.0	125.0	125.0	125.0	125.0	125.0	125.0	125.0	125.0
2x128	125.0	172.0	125.0	189.0	125.0	172.0	125.0	172.0	172.0	185.0	187.0	190.0	172.0	189.0
3 Car Fast Back:														
Q	125.0	170.0	44.0	121.0	93.0	125.0	54.0	109.0	0.0	0.0	0.0	0.0	0.0	0.0
1x16	68.0	137.0	45.0	84.0	0.0	48.0	0.0	37.0	0.0	0.0	0.0	0.0	0.0	37.0
2x2	125.0	125.0	125.0	125.0	125.0	125.0	125.0	125.0	125.0	125.0	125.0	125.0	125.0	125.0
2x128	125.0	172.0	125.0	189.0	125.0	172.0	125.0	172.0	172.0	185.0	187.0	190.0	172.0	189.0
4 Car Overtake 1:														
Q	108.7	130.5	81.4	104.5	48.7	96.4	33.3	59.6	24.4	58.1	8.4	14.9	4.7	7.9
1x16	109.1	145.3	7.3	58.6	3.6	22.3	1.1	5.5	3.2	7.3	0.3	3.6	2.4	23.9
2x2	102.4	102.7	102.3	102.7	102.7	102.9	102.9	103.1	102.5	103.0	102.8	104.1	102.6	103.5
2x128	187.0	189.3	120.4	163.7	163.9	188.0	163.8	189.4	188.0	201.0	102.8	163.5	163.6	188.0
4 Car Overtake 2:														
Q	93.0	128.1	58.8	83.9	46.3	96.5	12.2	59.5	28.2	34.9	8.0	24.1	4.6	6.2
1x16	93.4	148.9	17.5	37.2	8.8	15.0	16.2	39.7	1.2	12.5	26.4	84.1	2.1	9.7
2x2	102.6	103.1	102.7	102.9	102.5	103.2	102.7	103.2	102.9	103.5	102.9	103.2	102.3	102.6
2x128	163.6	188.3	163.1	196.8	163.1	199.0	108.4	188.0	118.6	197.0	103.2	163.4	107.0	163.5
4 Car Overtake 4:														
Q	161.5	175.8	138.7	161.1	87.2	112.8	108.4	120.6	45.3	88.8	40.2	53.3	12.3	25.5
1x16	133.8	141.1	62.4	75.0	26.4	69.6	6.8	65.6	27.9	58.7	8.6	19.6	6.9	29.7
2x2	102.6	103.1	102.3	103.0	102.8	103.3	102.5	102.8	102.9	103.1	102.6	103.3	103.0	103.7
2x128	187.2	197.9	169.0	197.0	154.5	189.4	163.7	189.3	163.5	163.7	163.7	199.0	163.3	163.6

Table XI: Results from all highway cases