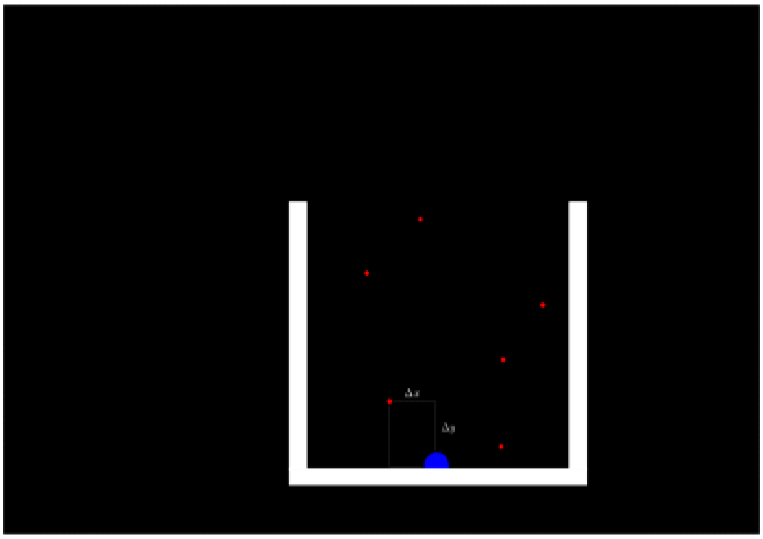
NEURAL NETWORK REINFORCEMENT LEARNING FOR AN AVOIDANCE GAME

TONĆI ANTUNOVIĆ

Abstract. A reinforcement learning algorithm has been implemented to train an AI model to play a new "obstacle avoidance" game. The learning algorithm takes as an input a low-dimensional continuous game state which is initialized and evolves randomly throughout each game. A simple neural network then estimates the cost of each move (Q-learning), which is modeled after the moment generating function of the number of points collected in future. The learning is performed with pure exploration and parameters are adapted using stochastic gradient descent method.

An obstacle avoidance game

In this report we consider a game in which a blue semicircular object has to avoid red falling balls. The blue object is attached to the bottom of a U-shaped white frame, which is always moving either left or right. The direction of the frame is controlled by the player, and is the only player input throughout the game. The whole frame (with the blue object) moves left-right around the vertical 3-dimensional cylinder, in other words, it wraps around the left and right screen edges. The game ends when a red falling ball hits the blue object. A point is scored every time a red ball hits the white frame. At that point, the red ball in question disappears and a new red ball appears at the uniformly random location at the top of the white frame, keeping the number of red balls fixed. The red balls fall vertically with small random vertical and horizontal noise added. Various parameters control the number of red balls, sizes of game objects, relative speed of frame vs balls, amplitude and frequency of the noise ball movement, etc. Note that the parameters have to be adjust properly to avoid certain trivialities, such as balls never being able to reach the blue object when the frame direction is fixed.



Reinforcement Learning

The game module run with N falling objects outputs the vector of relative coordinates of the falling objects with respect to the center of the blue object $z = (\Delta x_i, \Delta y_i)_{i=1}^N$. The coordinates are naturally generated and updated so that elements in vector z are sorted in the increasing order of Δy-coordinates. The vector z is used as the input in a three layer (one hidden layer) neural network (also known as multilayer perceptron) $\Psi\colon \mathbb{R}^{2N} \to \mathbb{R}$ with h hidden nodes as

$$\Psi(z; A, b) = \phi((\phi(A(z^*)))^* \cdot b), \tag{1}$$

where $w^* = (w, 1)$ is adding the bias term (natural embedding $\mathbb{R}^k \to \mathbb{R}^k \times \{1\}$) and $\phi(t) = (1 + e^{-t})^{-1}$ is the logit function being applied to each coordinate. Elements in the matrix $A \in \mathbb{R}^{(2N+1),h}$ matrix and the vector $b \in \mathbb{R}^{h+1}$ are parameters in the model.

The value $\Psi(z)$ is used to model the cost $C_R(z)$ of the state z when the current frame direction is to the right. By symmetry, the cost of the left action $C_L(z) = C_R(z')$ where z' is the same as the vector z with reversed sign of the Δy coordinates in z. In other words, from the current game state z the Q-values for the actions “right” and “left” are given by $\Psi(z; A, b)$ and $\Psi(z'; A, b)$ respectively. The cost $C_R(z)$ is modeled after the moment generating function (evaluated at some negative number $\lambda < 0$) of the number of points P collected from the current game state if the current move is to go to the right and the play is “optimal”, that is

$$C_R(z) = \mathbb{E}_{z,\text{right}}(e^{\lambda P}) = \mathbb{E}_{z,\text{right}}(\gamma^P),$$

where $\gamma = e^{\lambda}$ is between 0 and 1.

Note the three crucial properties of the functions C_R and C_L which are used for the updates of the Q-values (given for C_R, but the same holds for C_L as well). If from state z we perform the action “right” and end up at the state $\hat{z}$ then

- $C_R(z) = 1$, if at $\hat{z}$ the game is over;
- $C_R(z) = \gamma \min(C_L(\hat{z}), C_R(\hat{z}))$, if at $\hat{z}$ the game is not over, but a point has just been scored;
- $C_R(z) = \min(C_L(\hat{z}), C_R(\hat{z}))$, otherwise.

Note that even though the state space is naturally discrete, and the above transitions reflect this fact, we will consider the the state space to be continuous. The Q-values (1) are then smooth functions of the game state. The Q-value updates in reinforcement learning are performed as follows. Assuming that the current game state is z, select randomly action “left” (L) or “right” (R) with certain probabilities (see below). Set w to be z or z' depending if the selected action is R or L respectively, and set the current estimate $v_{\text{curr}} = \Psi(w; A, b)$. Perform the action chosen and observe the new state $\hat{z}$. Set the future estimate

$$v_{\text{fut}} = \begin{cases} 1 \\ \gamma \min(\Psi(\hat{z}; A, b), \Psi(\hat{z}'; A, b)), \\ \min(\Psi(\hat{z}; A, b), \Psi(\hat{z}'; A, b)), \end{cases}$$

using the same parameters in A and b used to compute v_{curr}. More generally, we use the discount damping parameter $0 < \delta < 1$, which (due to the multiplicative Q-value updates) goes into the exponent, and the updates become

$$v_{\text{fut}} = \begin{cases} 1 \\ \gamma \min(\Psi(\hat{z}; A, b), \Psi(\hat{z}'; A, b))^{\delta}, \\ \min(\Psi(\hat{z}; A, b), \Psi(\hat{z}'; A, b))^{\delta}. \end{cases}$$

The elements in matrix A and vector b are adapted using the stochastic gradient descent method, with the error function from the max-likelihood classifier

$$\mathcal{E}(v_{\text{curr}}, v_{\text{fut}}) = -v_{\text{fut}} \log v_{\text{curr}} - (1 - v_{\text{fut}}) \log(1 - v_{\text{curr}}).$$

This choice has benefit of having gradient of larger amplitude for values of v_{curr} close to 0 and 1. For each of the elements θ of A and b we simultaneously perform the update

$$\theta \mapsto \theta - \epsilon \frac{\partial \mathcal{E}(v_{\text{curr}}, v_{\text{fut}})}{\partial \theta} = \theta + \frac{\epsilon(v_{\text{fut}} - v_{\text{curr}})}{v_{\text{curr}}(1 - v_{\text{curr}})} \frac{\partial \Psi(w; A, b)}{\partial \theta}.$$

Here ϵ is the gradient learning parameter which can be adapted as the learning progresses (see below). This updated is performed by a standard backpropagation algorithm.

While learning, the action (direction) taken at each step is chosen to be the optimal (according to the learned policy) with the probability p_{greedy} which can be adapted as the learning progresses (see below). In other words it is choosing the action R if $\Psi(\hat{z}; A, b) < \Psi(\hat{z}'; A, b)$ and L otherwise. Otherwise, with probability $1 - p_{\text{greedy}}$, the action (direction) is not updated from the previous step with conditional probability p_{inertia}, and is uniformly updated with probability $1 - p_{\text{inertia}}$. The reason for the bias to the current direction (possibility of setting $p_{\text{inertia}} > 0$) is to allow the red objects to hit the vertical sides of the white frame. Setting $p_{\text{inertia}} = 0$ the values Δy_i in input vector z perform simple random walk with small variance and have very small probability of hitting the vertical sides of the frame before hitting the bottom. This would presumably make it quite difficult for the model to learn that red object hitting the vertical sides leads to a point scored, especially with pure exploration policy $p_{\text{greedy}} = 0$.

The training is started by initially setting the values in A and b to be independent and distributed uniformly in $(-1, 1)$, performing n_{test} initial testing games, and then performing consecutive training episodes consisting of n_{train} games followed by n_{test} testing games. Assume that the median score in the initial test session is m_0 and in the test part of the i-training episode m_i. In the first training episode we will use the gradient learning rate ϵ_1 which is set in the model, and for $i \geq 2$ in the i-th train episode the learn rate is set to be

$$\epsilon_i = \left(\frac{m_0}{m_{i-1}}\right)^{r_\epsilon} \epsilon_1,$$

where r is a chosen parameter. This way the gradient learning decreases as the median score goes up and increases as the median score goes down.

Policy "greediness" is also adapted according to the rule $p_{\text{greedy}}(i) = p_{\text{greedy}}$ and

$$1 - p_{\text{greedy}}(i) = (1 - p_{\text{greedy}}(i-1))\left(\frac{m_{i-1}}{m_i}\right)^{r_p},$$

with $p_{\text{greedy}}(i) = 0$, if the right hand side above is greater than 1.

To train the model we used the same parameters for both $N = 4, 5, 6$. We always used the neural net with $h = 12$ hidden nodes.

I) Initially the training was performed with pure exploration, that is $p_{\text{greedy}} = 0$ and $r_p = 0$ and without the discount term, that is $\delta = 1$. We also used $\gamma = 0.9995$, $p_{\text{inertia}} = 0.8$, $\epsilon_1 = 0.2$ and $r = 1$.

II) Afterwards the training was performed without scaling the parameter ϵ, that is $r_\epsilon = 0$. Here we used $\gamma = 0.8$, $p_{\text{inertia}} = 0.8$, $\epsilon = 0.05$, the discount $\delta = 0.999$, initial greediness $p_{\text{greedy}} = 0$ and $r_p = 0.5$.

The algorithm runs with either $n_{\text{train}} = 400$ or 1000 train games in each episodes followed by $n_{\text{test}} = 100$ or 400 test games, and learning is stopped when the median score goes over an appropriated threshold (with the added functionality in the code to run for a fixed number of training episodes).

Analysis

Pre-train performance with random parameters is very poor, scoring around 10 points. The performance for the trained model is tested with $N = 4, 5, 6$ red balls.

The average, median score, deviation, and 95% confidence intervals are shown in the following table under the training scheme I) is as follows.

N	median	average	st. deviation	95% conf. interval for average
4	238	342.14	336.46	342.14 ± 9.33
5	180	256.30	252.08	256.30 ± 6.99
6	162	233.62	228.39	233.62 ± 6.33

Under the training scheme there was a serious improvement for $N = 4$ balls case, achieving an average score of 906 points (±10% for 95% confidence interval) with the median score of 656 points.

Each training was performed on Intel's i5-2520M CPU with 4GB of RAM and took several hours. No serious human performance has been explored to compare with the above values, and naturally human performance will be drastically affected by the "game speed". However, for the

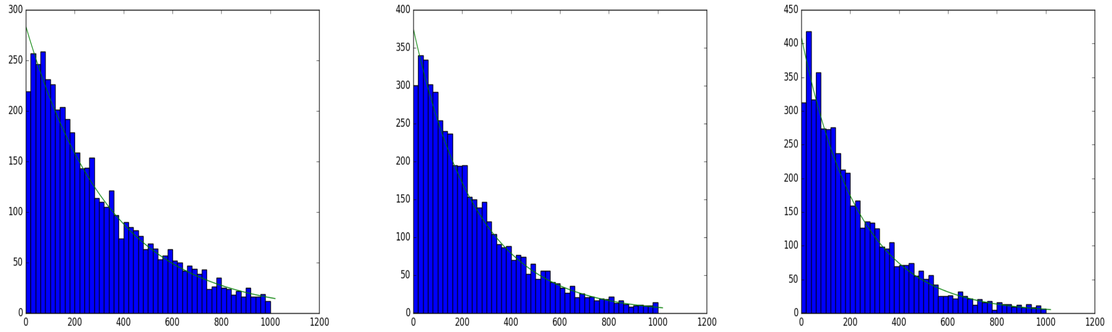


Figure 1. Histogram of trained scores for $N = 4, 5, 6$ (under training scheme I) and the predicted smoothed histogram for a geometric distribution with the appropriate mean. Scores are cutoff at 1000 points. Note that the gap at the beginning is due to game initialization which does not represent a typically sampled game state, in order for the human player to start playing more easily.

only experienced player (the author) it seems to be extremely difficult to consistently achieve the above results when played on the implemented "fast" level.

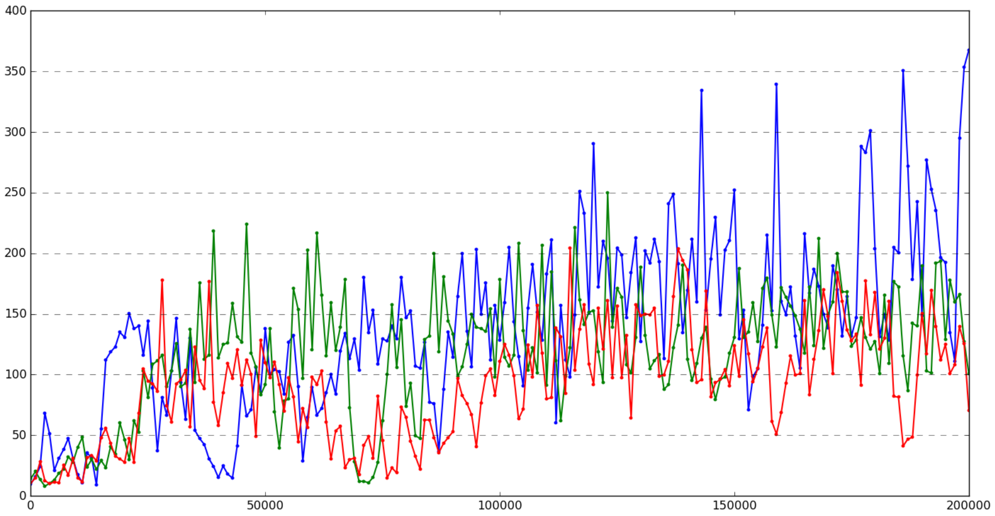
As seen in the above table, the score standard deviation is very close to average value, and the ratio of the median and average is close to $\log(2) \approx 0.693$ (ratios are 0.696, 0.702 and 0.693). Histograms shown give further indication that the scores have (roughly) geometric distribution. This is rather intuitive, since the AI plays the game until it reaches a certain state from which it either can't avoid the loss, or is not trained to handle, and since the balls appear and evolve independently and with identically distributions. The natural dispersion of the geometric distribution makes it quite likely for the model to perform significantly better but also significantly worse from the average values. This could lead to very surprising but also very disappointing results when testing the trained model for performance.

Below are learning curves for average game scores, for the model trained with the same parameters stated above and used in the application. Under the scheme I) the curves are the result of 200 learning episodes each having $n_{\text{train}} = 1000$ learning games and $n_{\text{test}} = 400$ test games. Under the scheme I) the curves are the result of 50 learning episodes each having $n_{\text{train}} = 400$ learning games and $n_{\text{test}} = 400$ test games. This is a rather small value for n_{test} since (assuming geometric distribution for game scores), each of the test averages will have confidence intervals of roughly

$$\pm 100 \frac{1.96}{\sqrt{400}} \% = \pm 9.8\%.$$

Unfortunately, larger values n_{test} were not feasible to implement due to somewhat outdated hardware available. This is partly a reason for the large oscillations in the presented learning curves.

See the code at `github.com/Tantun/AIVoid` and a YouTube video at `https://www.youtube.com/watch?v=5_h13nDa6_E`.



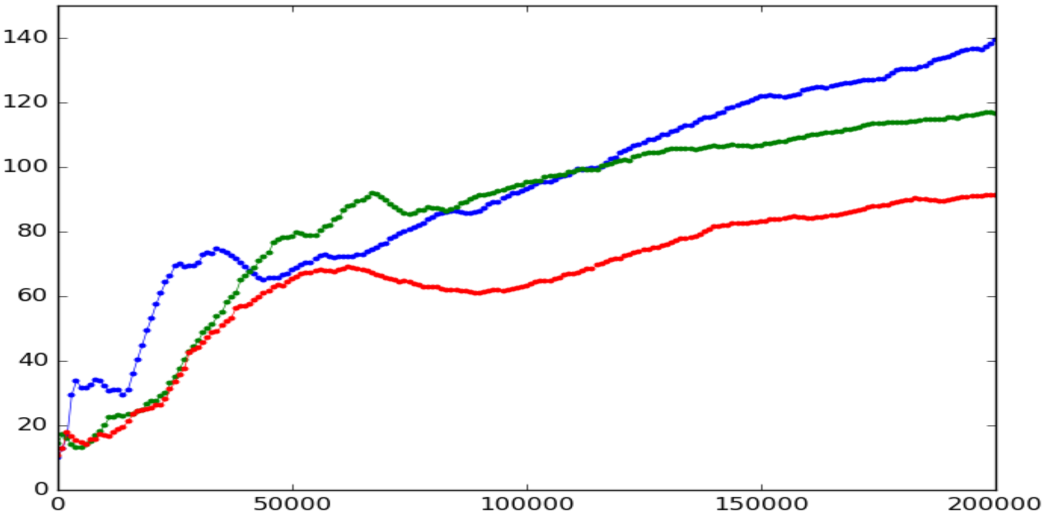


FIGURE 2. Learning curves for $N = 4$ (blue), $N = 5$ (green) and $N = 6$ (red) under the training scheme I). There are 200 train sessions, each with 1000 games. The first figure represents the average test scores of each test episode, while the second one running average across all the test games so far since the beginning of the training. Final average test scores for $N = 4, 5, 6$ are 367.51, 100.7325, and 70.5825 respectively. Best average test scores for $N = 4, 5, 6$ are 367.51, 249.975, and 204.315 respectively.

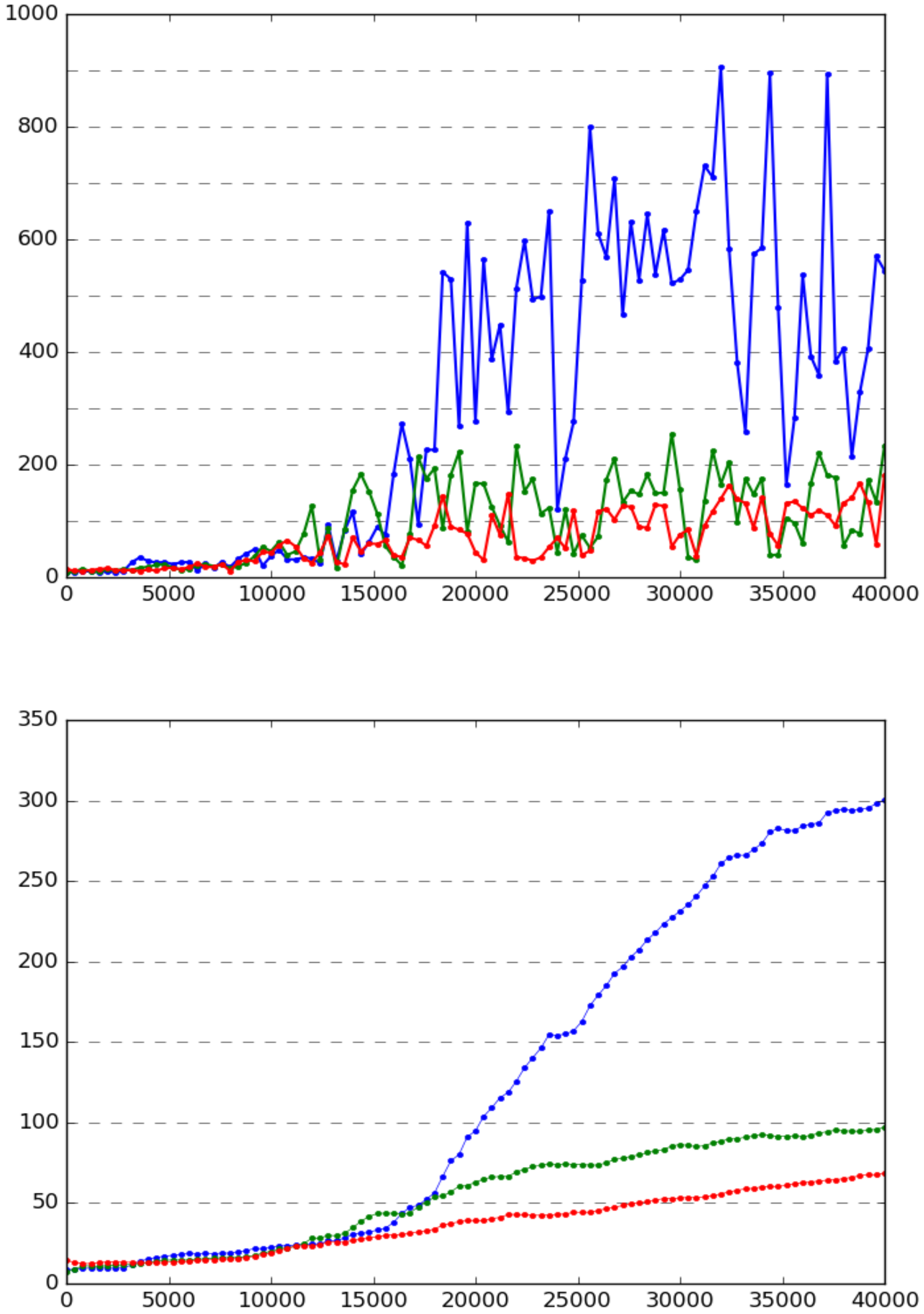


FIGURE 3. Same learning curves for $N = 4$ (blue), $N = 5$ (green) and $N = 6$ (red) under the training scheme II). There are 100 train sessions, each with 400 games. Final average test scores for $N = 4, 5, 6$ are 544.605, 234.2275, and 181.65 respectively. Best average test scores for $N = 4, 5, 6$ are 905.8475, 255.195, and 181.65 respectively.