Game Playing for Modified 1010!

Kaan Ertas
Department of Computer Science
Stanford University
kertas@stanford.edu

Neval Cam
Department of Computer Science
Stanford University
nevalcam@stanford.edu

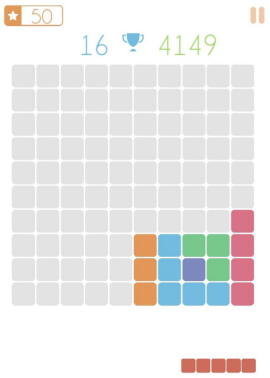
Jung-Won Ha
Department of Computer Science
Stanford University
jwha23@stanford.edu

Abstract

1010! is a single-player game similar to Tetris, where random pieces are given to the player to place on the board, where rows and columns are cleared once complete. Modeling the game as a Markov Decision Process, we propose the use of Q-Learning, with an exponentially decaying ϵ-greedy policy and a Q value approximating function to tackle the large state space. Our findings show that a Convolutional Neural Network Q-Learner with a target network performs best.

1 Introduction

We are considering a modified version of the game 1010!, which is a one player game played on a 10x10 grid. At each turn, the player randomly receives a piece to place on the board. Once that shape is placed, another shape is given, and the game continues until there is no space on the board to place the piece. The types of shapes that may be provided are fixed in advance. When the player completes a column or row with the pieces they have placed, the row/column is cleared just like Tetris. However, there is no gravity, and blocks stay in place after being placed. The scoring is based on the size of the pieces placed and the number of rows/columns cleared, and extra score is awarded if multiple rows/columns are cleared simultaneously.



The horizontal 5-piece, when placed on the lowest row, clears out the row. Once the piece is placed, we receive a new piece.

33rd Conference on Neural Information Processing Systems (NeurIPS 2019), Vancouver, Canada.

2 Task Definition

We want to create a game playing agent for the game 1010!. When given the current state of the board and a piece to be placed, the agent should be able to place that piece in an optimal location that will maximize its score. Through playing multiple games, the agent's performance should improve as it eventually learns an optimal strategy that will consistently earn a high score when playing the game.

3 Literature Review

3.1 Q-Learning

Q-Learning is an off-policy reinforcement learning algorithm introduced by Watkins in 1989 which evaluates state-action pairs to quantify the quality of an action for a given state, and utilizes these values in policy decisions [1]. Q-Learning with deterministic action selection runs the risk of getting stuck in a local optimum. In order to strike a balance between exploration and exploitation, an ϵ-greedy algorithm is often employed, where for a given state s, the action given by $\underset{a}{argmax} Q_{opt}(s, a)$ is chosen with probability $1 - \epsilon$ and a random action is selected with probability ϵ.

3.2 Deep Q-Learning and Target Networks

Mnih et al. (2015) proposed the deep Q-Network (DQN) that uses deep convolutional neural networks and Q-learning to play visual games [2]. The algorithm would be input 4 consecutive frames and an action, and output estimated Q-value for the state-action pair. The algorithm notably used experience replay, which will be discussed later, and a target network.

It is important to note that during on-line training (train as you play), the (state,action,reward,next state) tuples arrive in a temporal fashion as the game is played out, meaning that the data used to train the neural network will not be independent and identically distributed as there is very high temporal correlation between two consecutive states. Furthermore, due to the recursive nature of Q-Value calculation (which uses the network's approximation to calculate V_opt), the network has non-stationary target values. To tackle this problem, Mnih et al. propose the use of target networks, which keeps two separate networks N_t and N_v. The two are initialized to be the same, but the training process uses N_t for loss and gradient calculations to update N_v, and sets $N_t \leftarrow N_v$ at every T updates.

3.3 Experience Replay

While target networks alleviate the problem of temporal dependencies between training samples, they do not solve the problem entirely. Another approach is experience replay, originally proposed by Lin (1993), which caches a maximum of N (state, action, reward, next state) tuples at any point in time and trains the algorithm using a random minibatch of the N samples [3]. As new experiences arrive, the oldest experiences are discarded to keep the cache at size N. This is in contrast to online training. Random sampling from memory further eliminates temporal correlations between training samples.

3.4 Double Q-Learning

Van Hasselt (2010) observes that using the maximum operator over all possible actions from the next state when updating the Q value of a state-action pair can lead to overestimations, because the maximum value is used in place of maximum expected value [4]. To have an unbiased estimation of the Q-value at each step, he proposes the use of two separate Q functions. At every update step, one of the two functions is randomly chosen to be updated, but the other function is used in calculating the Q-value for the next state-action pair that gives V_{opt}. The resulting estimates may underestimate, but do not suffer from the overestimation bias of conventional single function Q-learning and attains better performance more quickly.

Double Q-Learning, when applied to deep learning and to the DQN algorithm, was shown by van Hasselt et al. (2015) to perform better than DQN [5].

4 Methods

4.1 Initial Approach

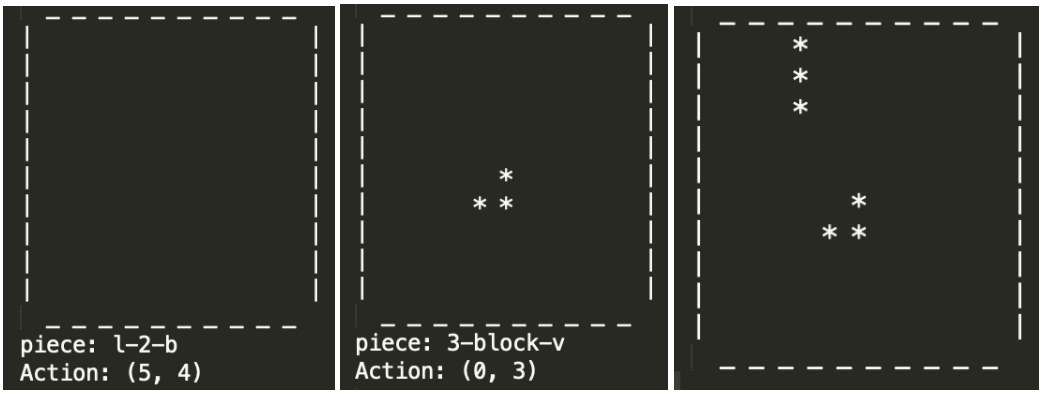
We implemented a random player that places a random piece on a random available spot on the board, and simulated 1000 games. The average score of this player was 75.2 with a standard deviation of 22. As an oracle for this problem, we use the leaderboard scores for the game, as described in the evaluation metric.

4.2 Defining the Problem as a Markov Decision Process

State: The board is represented as a grid of booleans. A cell is true if a piece has been placed (and not cleared) there. Otherwise it is false. A piece is represented as a grid of booleans with the smallest rectangular grid encapsulating the piece, with the positions occupied by the piece having boolean value True.

We cannot use the board itself as the state, since the piece that needs to be placed will inform the action that needs to be taken. So the state is a board, piece pair. At any given moment of the game before we place a piece, we have a board which is a grid (a portion of which is filled), and a piece that needs to be placed. We reach an end state when there are no more spaces on the board to place the current piece that needs to be placed.

Action: An action is an (x,y) pair that represents the top-left coordinate of the placement of a single piece.



Reward: This is the change in score after taking the action, as defined by the original game. The exact formula is: R = [$5 + 5*$(number of rows to eliminate + number of cols to eliminate)] * (number of rows to eliminate + number of cols to eliminate) + size of piece placed

Transition Probabilities: The state of the board given a board and a piece placement is deterministic. But the new piece that we will get is randomized. So we have
successorState((board,piece)) = (state of board after piece is placed and rows are cleared, new_piece) where new_piece is one of the 19 possible pieces, with 1/19 probability each.

4.3 Q-Learning

Having framed the problem as a Markov Decision Process, we employed Q-Learning as discussed above, with an ϵ-greedy policy. To increase performance of the algorithm after some amount of learning has taken place and favor exploitation over exploration, we used an exponentially decaying ϵ value dependent on the number of updates.

As mentioned by Sutton and Barto, the algorithm converges correctly if all state-action pairs are updated [6]. However, given that the state space is too large (no. of board configurations = 19 pieces $*$ 2^{100}, and no. of states as we have defined them = 19 pieces $*$ 2^{100} $*$ 100 actions states as we have defined them) Q scores for all state-action pairs cannot be stored.

4.4 Linear Regression Approximator

We initially used a linear regression approximator to evaluate state-action pairs such that $Q_{opt}(s,a) \approx w \cdot \phi(s,a)$. Our feature vector consisted of indicator variables for the board in the current state (0 if free, 1 if occupied, and flattened), along with one-hot representation of the piece to be placed. We also hand-crafted a few features, discussed in the next subsection. The weights for the regression are learned after each (state, action, reward, next state) tuple through gradient descent:

$$w \leftarrow w - \eta[\hat{Q}_{opt}(s,a,w) - (r + \gamma \hat{V}_{opt}(s'))]\phi(s,a)$$
$$\hat{Q}_{opt}(s,a,w) = w \cdot \phi(s,a)$$
$$\hat{V}_{opt}(s') = \max_{a'} \hat{Q}_{opt}(s',a',w)$$

4.5 Features

Alongside using the board itself, flattened into a vector, we experimented with using the following features in our feature vector for Q-value approximation:

- Number of occupied cells on the board
- Number of available spots for the 3 by 3 square block
- Indicator variable for whether there is an available spot for the 3 by 3 square block
- Number of occupied cells in each row (10 features)
- Indicator variable for whether a row is clean (10 features)
- Number of occupied cells in each column (10 features)
- Indicator variable for whether a column is clean (10 features)
- Weighted sum of the cells on the board, with higher weights associated with cells closer to the center of the board

The last item on the list tries to capture the idea that placing pieces in the middle of the board is less favorable, and placing pieces adjacent to corners leaves the board in a more convenient position. The weighted sum is a nonnegative integer, with lower scores corresponding to better states of the board.

4.6 Deep Q-Learning

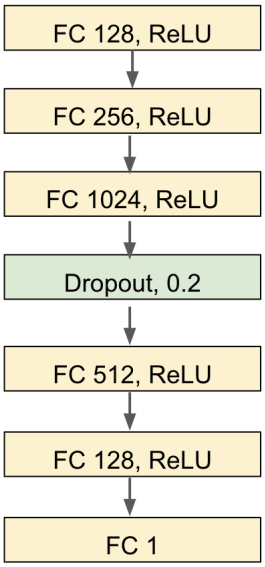
Seeing that linear features were not sufficient for successful approximation, we employed deep learning for Q-value approximation. We used the neural net to approximate $Q_{opt}(s,a)$ given a state s and action a, and used $r + \gamma V_{opt}(s')$ as the ground truth value. All our implementations were done in the Keras framework [7].

For both network architectures, we used the Adam optimizer as introduced by Kingma and Ba (2014) which is appropriate for our problem since it is a stochastic gradient descent optimizer that can work on non-stationary targets [8]. According to the implied loss function in the gradient descent equation mentioned above, we used Mean Square Error as the loss function. To tackle the problem of temporal correlation between training examples, we separated the valuator and target network as discussed earlier. As for the frequency of synchronisation between two networks, we used every 5000 updates.

Even though the state transition is nondeterministic (because of the randomness of the piece selection), the board configuration of the next state given a state and action is deterministic. Since the Q-value for a state and action pair should be an indicator of the quality of an action given a state, we can try to quantify the quality of the next board configuration. Therefore, for a state-action pair, the neural network input board state is the state of the board after the piece has been placed to the board and cells are cleared accordingly. Thus we abandoned our early input format to the approximator.

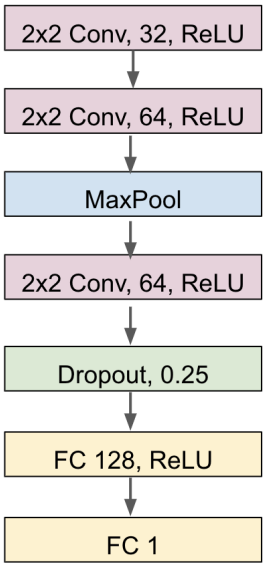
4.6.1 Fully Connected Neural Network

We implemented the architecture below, which has a linear regression unit at the end without an activation function. The input is the next board state (with indicator variables for each cell and flattened), and a subset of the features discussed above.



4.6.2 Convolutional Neural Network

Convolutional neural networks take advantage of local dependencies in multidimensional inputs and require significantly fewer parameters due to the convolution process. CNNs are appropriate for 1010! because a piece placement and board quality depend on local features of the board (ie. a piece placement is usually better if the piece is placed adjacent to other pieces). We implemented the following architecture, with the board (with indicator variables for each cell) as input resulting in a (10,10,1) input size, and a linear regression unit without an activation function as output.



4.7 Evaluation Metric

There are multiple collaborative, online sites such as the official 1010! Leaderboard on Facebook which records the historical highest scores amongst all users during the play of the game. Currently, there have been approximately 28 million players of the game and the highest score on the leaderboard is 124,933. We will compare the performance (i.e. the overall earned score) of our agent against the historical human leaders of the game to evaluate our performance. In addition to evaluating scores,

we could also measure the efficiency (i.e. number of rounds played to reach high score) and speed (i.e. fastest time to reach k points) of our agent playing the game against human performers, for which the data exists online.

One side note is that the highest score in the leaderboard might also be a nonhuman agent. It might also be useful to note that an average human score in the game is 1,000-5,000.

5 Results and Analysis

5.1 Initial Results

5.1.1 Random Algorithm

Our random baseline algorithm did not perform well, as expected. The agent would end the game early as the scores averaged around 70, usually without being able to clear any rows or columns.

5.1.2 Linear Regression Q-Value Approximator

Even though our linear approximator more than doubled in score compared to our random baseline algorithm, averaging around 150, our overall performance was still not as high as desired.

5.1.3 Deep Q-Learning

While continuously tracking the performance of the deep Q-learning algorithm, we noticed that the agent did not seem to be learning and improving in performance. Rather, the scores inconsistently fluctuated from the low 80s to every so often reaching a high score in the 300s. Overall, the scores oscillated around the mean of 170.

In attempts to improve our score and the consistency of our performance, we attempted to incorporate a subset of the features from the linear model and change the discount factor and learning rate. However, none of these modifications to our features and parameters significantly improved the performance.

Resolution: After numerous attempts of modifying the parameters to improve our Q-Learning algorithm to no avail, we concluded that the state space of the 10x10 grid was too large. Therefore, to resolve this, we decided to conduct our future tests on a 6x6 grid instead. All the results reported in the rest of the report are for a game played on a 6x6 board, with pieces with size more than 2 in any dimension removed (except the 3x1 piece). This resulted in a 10-piece catalogue, and the state space thus reduced by a factor of $\frac{2^{100}*19*100}{2^{36}*10*36} \sim 10^{20}$.

5.2 Deep Q-Learning on 6x6 Board

5.2.1 Fully Connected Neural Network Approach Results

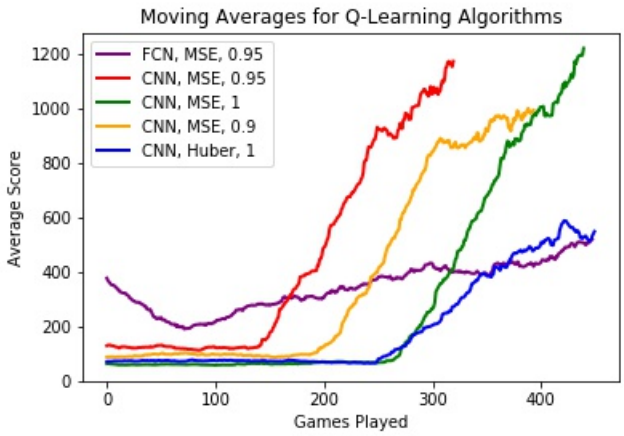
While experimenting with Fully Connected Neural Networks, we used a discount factor of 0.95 and Mean Squared Error as our loss function. Our results showed that this algorithm performed significantly better than our initial approaches, as expected. The highest score our model obtained was 2153.

5.2.2 Convolutional Neural Network Results

While experimenting with CNNs, we tried different structures, discount factors (0.9, 0.95, 1) and loss functions (Mean Squared Error, Huber). Huber loss is shown to improve performance in OpenAI's DQN (2015), which uses Deep Q-Learning on video game frames.

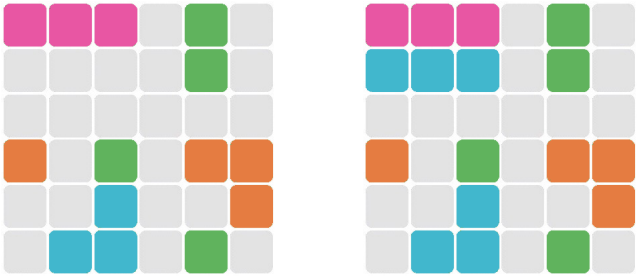
Initial results of our first CNN structure were not as high as our results with Fully Connected Neural Networks. Using our initial CNN structure, 0.95 discount factor, and mean squared error loss, the highest score we achieved was 1029. After changing our structure, we observed that the new architecture (included in Section 4.6.2) performed significantly better.

Loss Function	Discount Factor	Highest Score Achieved
Mean Squared Error Loss	0.9	3902
Mean Squared Error Loss	0.95	5302
Mean Squared Error Loss	1	5992
Huber Loss	1	3071



5.3 Error Analysis

After analyzing the games with significantly lower scores, we observed one pattern that caused these games to end early. One of the key strategies for this game is to leave large enough space for the largest piece to be placed after every action. Human players can discover this strategy after playing for a while. However, our algorithms failed to learn this. Here is a specific example from one of the games we observed:



The random piece given to be placed in the initial state was a 3x1 piece. In the next state we see that this piece was placed in (1,0). After this action, the game selected another random piece, which was a 2x2 square. Since the board did not have a large enough space for the 2x2 piece to be placed, the game ended. If our algorithm chose to place the 3x1 piece in (2,2) or (2,3) to leave a big enough space for a 2x2 piece, we would achieve higher scores. Certain intuitive approaches like these are not easy for neural networks to learn.

6 Future Work

We were limited in our experimentation by time and computing resources. Running the algorithm more will result in more games played, a larger dataset and better training. This would also allow us to see whether the learning curves for our models lose momentum over time. The use of Double

Q-Learning can further improve and speed up performance. Distributed training, to make the best use of resources, is also a possible improvement.

We can also experiment with different Convolutional Neural Network architectures. Even though there are inherent limitations due to input size, we can use more filters and less frequent pooling layers. We could also do analysis on layers to investigate whether there are neurons are activated by features human agents use when playing the game.

References

[1] Watkins, Christopher. Learning From Delayed Rewards, 1989.

[2] Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., et al. (2013). Playing Atari with deep reinforcement learning. Technical report. Deepmind Technologies, arXiv:1312.5602 [cs.LG]

[3] Jin, Long-Ji. Reinforcement learning for robots using neural networks, 1993.

[4] Hado van Hasselt. Double Q-Learning. Advances in Neural Information Processing Systems, 2010.

[5] Hado van Hasselt, Arthur Guez, David Silver. Deep Reinforcement Learning with Double Q-Learning. Association for the Advancement of Artificial Intelligence, 2015.

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

[7] Chollet, François. Keras, 2015.

[8] arXiv:1412.6980.