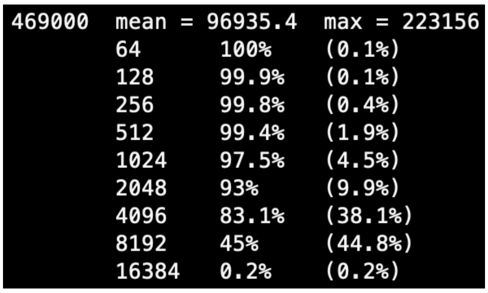
Deep Learning and Practice Report
Lab3: 2048-Temporal Difference Learning

1 A plot shows episode scores of at least 100,000 training episodes.

The overall results after training $469,000$ episodes are shown in figure 1. Figure 2 also shows the maximum score, mean score and average 2048 win rate in $1,000$ games during training process. The learning rate is set to 0.1 and the model is trained for a total of $500,000$ episodes.



```
469000  mean = 96935.4  max = 223156
        64      100%    (0.1%)
        128     99.9%   (0.1%)
        256     99.8%   (0.4%)
        512     99.4%   (1.9%)
        1024    97.5%   (4.5%)
        2048    93%     (9.9%)
        4096    83.1%   (38.1%)
        8192    45%     (44.8%)
        16384   0.2%    (0.2%)
```

Figure 1: Average scores in 1000 games after training $469,000$ episodes.

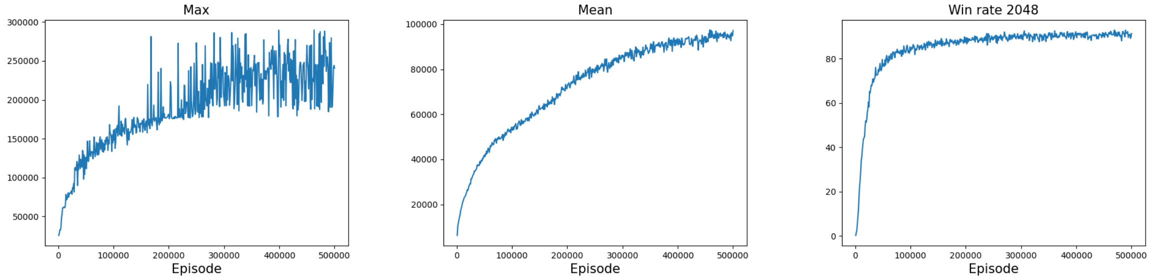


Figure 2: Learning curves of max, mean, and 2048 win rate.

2 Describe the implementation and the usage of n-tuple network.

N-tuple network is a commonly used value function approximator for training 2048 agents, it reduces the number of total states significantly while preserving huge amount of features. Instead of calculating the value of a given board directly, we sum up the values from different n-tuples to represent the board. Also, each n-tuple can provide a total of 8 features if we consider all isomorphisms (rotations, reflections). Therefore, the value for each board (state s) can be obtained as follows,

$$V(s) = \sum_{i=1}^{k} \sum_{j=1}^{8} f_{ij}(s)$$

where f_{ij} stands for the value function of the j-th isomorphism of the i-th n-tuple.

Group-ID	N-Tuple 1	N-Tuple 2	N-Tuple 3	N-Tuple 4
0	012345	456789	012456	45689a
1	156	0125	01259	04589a
2	01259	6$abde$	12569	0125

Table 1: Different groups of N-tuples.

In this lab, I've tried different combinations of n-tuples. Three groups of 4 N-tuples are list in table 1. Each tuple is represented by a string of numbers, where each number stands for the index of the game board. Group 0 is the default n-tuples from the sample code, which contains four 6-tuples, and group 1 consists of 3, 4, 5, and 6-tuple, while group 2 contains three 5-tuples and a 4-tuple.

3 Explain the mechanism of TD(0).

TD(0) is the simplest form of **t**emporal **d**ifference learning. Different from Monte-Carlo (MC) learning, who needs to wait until the entire episode terminates so as to update the value function of each state, TD learning updates the value function every step along with the estimated value of next state and the obtained reward. This is also called bootstrapping. On the other hand, TD learning exists larger bias (error) than MC learning since it always updates the value towards a guess, while MC learning updates the value by the actual one obtained in each episode. However, in games like 2048, same states must appear in different episodes and have quite different values. Therefore, TD learning enables the agent to learn in a stabler manner.

4 Explain the TD-backup diagram of V(after-state).

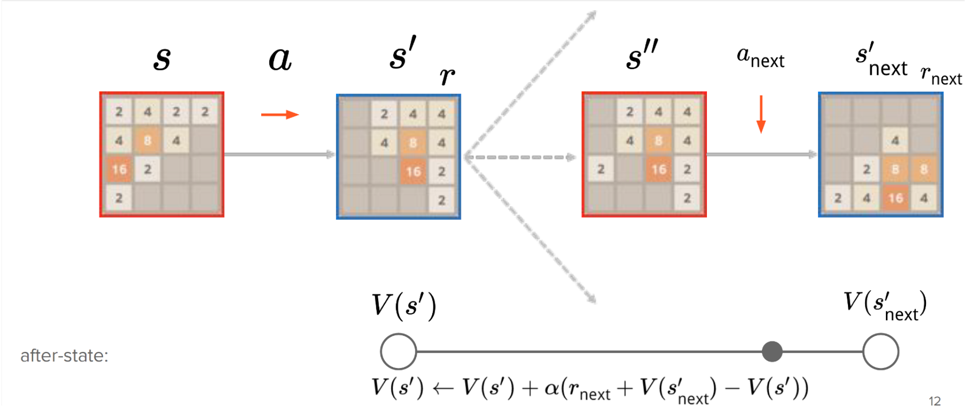


Figure 3: TD-backup diagram of V(after-state).

Figure 3 illustrates the TD-backup diagram for after-state. The states with red bounding box represent the before-states, while those with blue bounding box represent the after-states. Suppose we begin the game in the before-state s, then we can obtain the after-state s' and its corresponding reward r after an action a. After a new tile (either 2 or 4) is popped up on the board, it then transfer to a new state s''. Again, we can get a new after-state s'_{next} and its reward r_{next} after an action a_{next} is performed. The value of each state is denoted as $V(s)$. In the backup step of TD Learning, we update an episode by updating the value of each state backwards. Take state s' as an example, its TD target is the sum of the reward after next action and the estimated value of next after-state. Hence, the value of s' is updated by the error between TD target and its current value, multiplied by the learning rate α.

5 Explain the action selection of V(after-state) in a diagram.

For training after-state agents, we only need to consider the reward and the value of next state after each possible action, which is same as the TD target mentioned in section 4. Therefore, we select the action which maximizes $r_{next} + V(s'_{next})$.

6 Explain the TD-backup diagram of V(state).

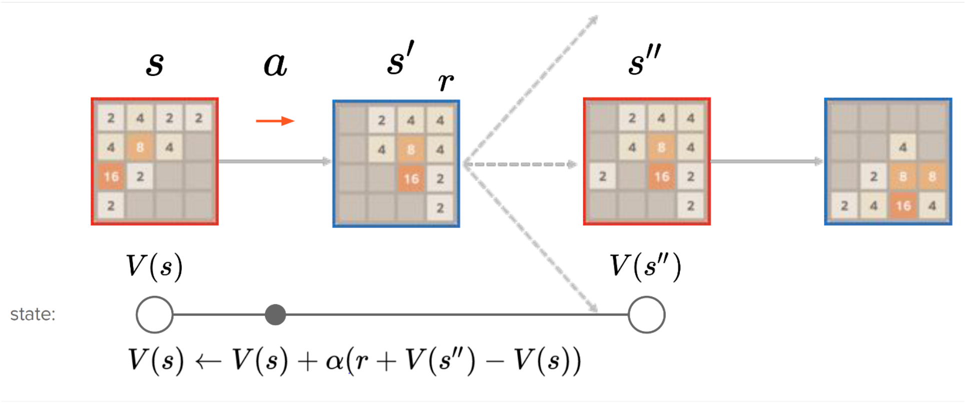


Figure 4: TD-backup diagram of V(state).

Figure 4 illustrates the TD-backup diagram for before-state. Basically, all notations have the same definition as in figure 3. In this diagram, we update the value for each (before-)state. Different from after-states, the TD target here becomes the sum of the reward after an action and the estimated value of next before-state. Hence, the value for state s is updated by the error between TD target and and its current value, multiplied by the learning rate α.

7 Explain the action selection of V(state) in a diagram.

The action selection for before-state agents is relatively complex. Since the agent is ignorant of the new tile's pop-up location and its actual value, we need to consider all possible cases for the next before-state (s''). By listing all s'' cases after different actions, we can estimate their corresponding values and the probability of each case appearing. The total expected value of each action is then obtained by summing the weighted values of all possible cases. Finally, we select the action which maximizes $r + \mathbb{E}[V]$.

$$\mathbb{E}[V] = \sum_{i=1}^{n} p_i \times \mathbb{E}[V(s''_i)]$$

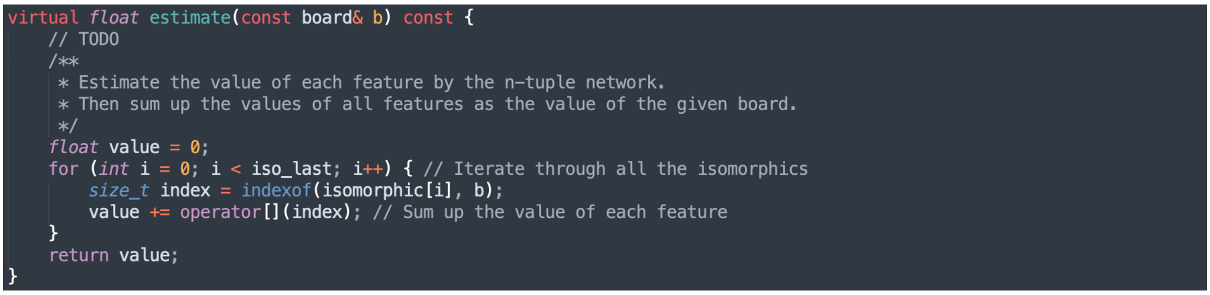
where p_i and $\mathbb{E}[V(s''_i)]$ denotes the probability and the estimated value of the i-th case, respectively.

8 Describe your implementation in detail.

There are five TODOs in the sample code. Three of them are functions of the class `pattern`, which is a subclass of `feature`, and the others are functions of the class `learning`.

8.1 Functions for pattern

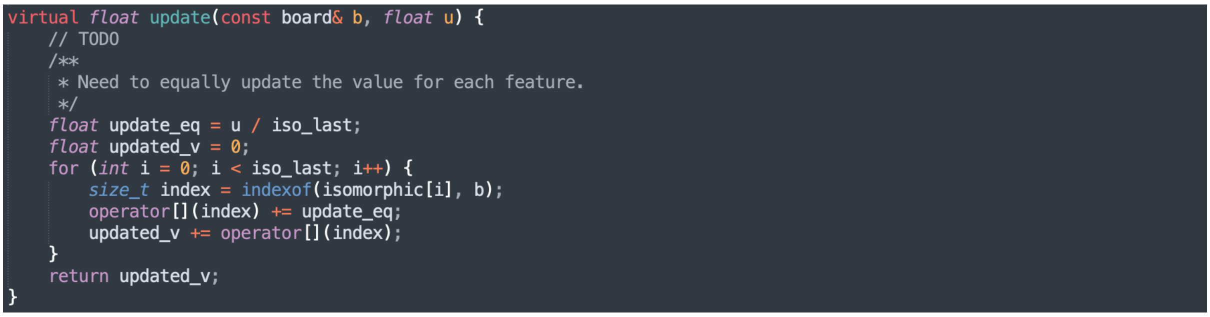
Figure 5 shows the code of the function `estimate`. This function is used to estimate the value of a given board by summing up all isomorphisms' values of an n-tuple.



```
virtual float estimate(const board& b) const {
    // TODO
    /**
     * Estimate the value of each feature by the n-tuple network.
     * Then sum up the values of all features as the value of the given board.
     */
    float value = 0;
    for (int i = 0; i < iso_last; i++) { // Iterate through all the isomorphics
        size_t index = indexof(isomorphic[i], b);
        value += operator[](index); // Sum up the value of each feature
    }
    return value;
}
```

Figure 5: Code of the function `estimate`.

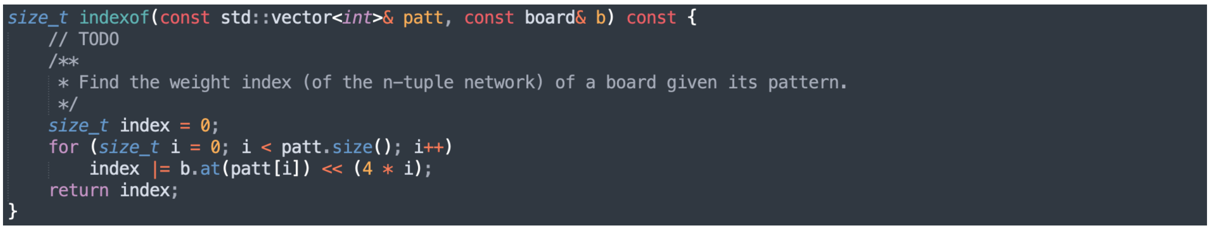
The code of the function `update` is shown in figure 6. The function updates the value of a given board and return its updated value. Note that the weight for each isomorphism should be updated equally, which is why the update value is split into 8 parts in the beginning of this function.



```
virtual float update(const board& b, float u) {
    // TODO
    /**
     * Need to equally update the value for each feature.
     */
    float update_eq = u / iso_last;
    float updated_v = 0;
    for (int i = 0; i < iso_last; i++) {
        size_t index = indexof(isomorphic[i], b);
        operator[](index) += update_eq;
        updated_v += operator[](index);
    }
    return updated_v;
}
```

Figure 6: Code of the function `update`.

Figure 7 shows the code of the function `indexof`. The function returns the weight index of the isomorphic pattern given a board. The index can be used to access the corresponding weight value.

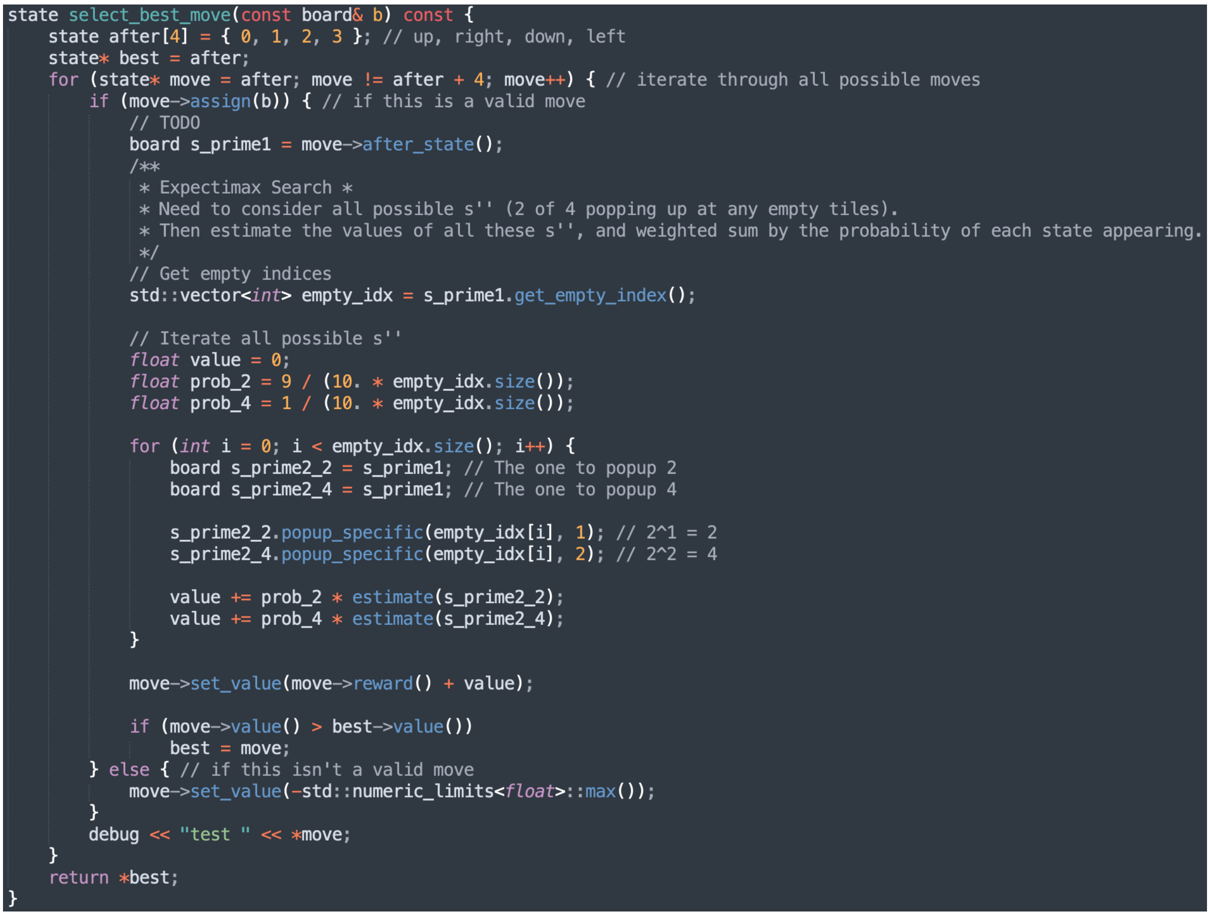


```
size_t indexof(const std::vector<int>& patt, const board& b) const {
    // TODO
    /**
     * Find the weight index (of the n-tuple network) of a board given its pattern.
     */
    size_t index = 0;
    for (size_t i = 0; i < patt.size(); i++)
        index |= b.at(patt[i]) << (4 * i);
    return index;
}
```

Figure 7: Code of the function `indexof`.

8.2 Functions for learning

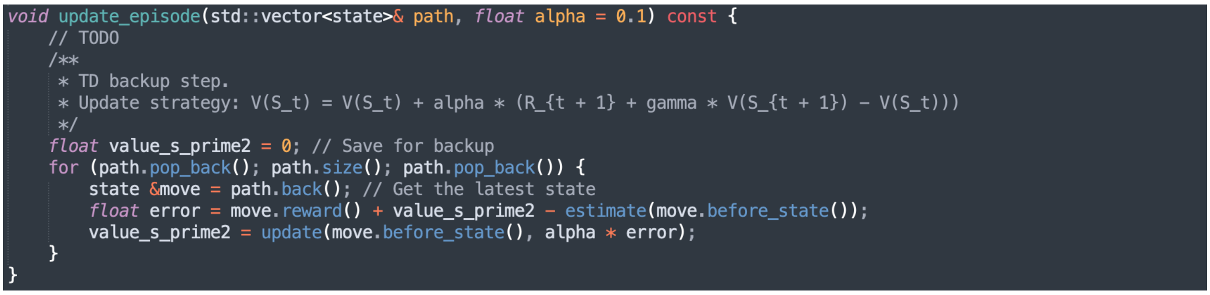
The function `select_best_move` (shown in figure 8) performs the action selection for before-state agent. As I described in section 7, we need to iterate through all possible after-states for each action. First, I obtain the empty tiles' indices by adding a function `get_empty_index` in the class `board`. Next, I iterate through all cases that 2 or 4 popping up at each empty index, and sum up their estimated values weighted by corresponding probability. Finally, the expected value of taking a specific action is the weighted-sum plus the reward, and we select the action maximizing this expectation.



```cpp
state select_best_move(const board& b) const {
    state after[4] = { 0, 1, 2, 3 }; // up, right, down, left
    state* best = after;
    for (state* move = after; move != after + 4; move++) { // iterate through all possible moves
        if (move->assign(b)) { // if this is a valid move
            // TODO
            board s_prime1 = move->after_state();
            /**
             * Expectimax Search *
             * Need to consider all possible s'' (2 of 4 popping up at any empty tiles).
             * Then estimate the values of all these s'', and weighted sum by the probability of each state appearing.
             */
            // Get empty indices
            std::vector<int> empty_idx = s_prime1.get_empty_index();

            // Iterate all possible s''
            float value = 0;
            float prob_2 = 9 / (10. * empty_idx.size());
            float prob_4 = 1 / (10. * empty_idx.size());

            for (int i = 0; i < empty_idx.size(); i++) {
                board s_prime2_2 = s_prime1; // The one to popup 2
                board s_prime2_4 = s_prime1; // The one to popup 4

                s_prime2_2.popup_specific(empty_idx[i], 1); // 2^1 = 2
                s_prime2_4.popup_specific(empty_idx[i], 2); // 2^2 = 4

                value += prob_2 * estimate(s_prime2_2);
                value += prob_4 * estimate(s_prime2_4);
            }

            move->set_value(move->reward() + value);

            if (move->value() > best->value())
                best = move;
        } else { // if this isn't a valid move
            move->set_value(-std::numeric_limits<float>::max());
        }
        debug << "test " << *move;
    }
    return *best;
}
```

Figure 8: Code of the function `select_best_move`.

The second function for learning is `update_episode`. As shown in figure 9, I update the value of each state encountered in the episode by looping all states backwards. The value of each state is updated as mentioned in section 6.



```cpp
void update_episode(std::vector<state>& path, float alpha = 0.1) const {
    // TODO
    /**
     * TD backup step.
     * Update strategy: V(S_t) = V(S_t) + alpha * (R_{t + 1} + gamma * V(S_{t + 1}) - V(S_t))
     */
    float value_s_prime2 = 0; // Save for backup
    for (path.pop_back(); path.size(); path.pop_back()) {
        state &move = path.back(); // Get the latest state
        float error = move.reward() + value_s_prime2 - estimate(move.before_state());
        value_s_prime2 = update(move.before_state(), alpha * error);
    }
}
```

Figure 9: Code of the function `update_episode`.

Learning Rate	**N-Tuples**	**Maximum**	**Mean**	**2048 Win Rate**
1.0×10^{-1}	Group 0	$242,412$	$95,550.3$	90.4%
5.0×10^{-2}	Group 0	$246,728$	$81,535.8$	90.2%
1.0×10^{-2}	Group 0	$103,156$	$41,503$	78.2%
2.5×10^{-3}	Group 0	$67,784$	$21,858.2$	37.4%
1.0×10^{-1}	Group 1	$139,112$	$56,391.8$	90.5%
1.0×10^{-1}	Group 2	$143,149$	$55,702.4$	88.9%
1.0×10^{-1}	Group $0 + 1$	$178,152$	$98,474.1$	94.3%

Table 2: Experiment results under different settings.

9 Other discussions or improvements.

In order to observe the model's behavior, I perform some experiments with different learning rates and n-tuple networks. Table 2 demonstrates the overall testing results within 1000 episodes. Also, the learning curves with different learning rates are shown in figure 10. At first, I thought that using smaller learning rates can improve the performance considerably. However, it is obvious that the 2048 win rate can at most achieve the same scale. All scores are even worse if the learning rate is too small.

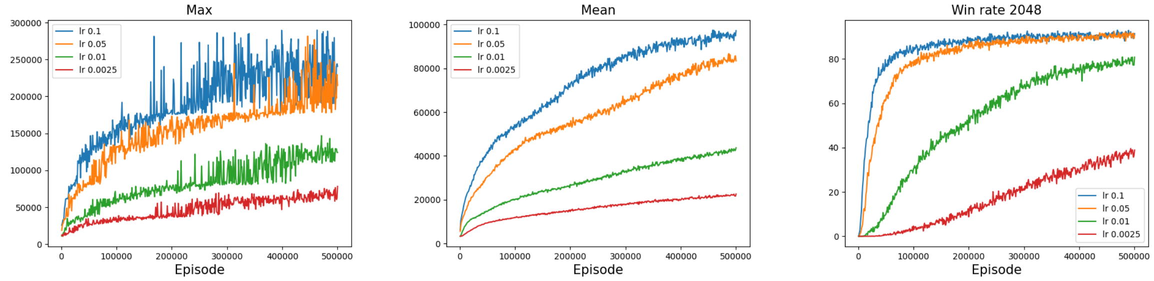


Figure 10: Learning curves with different learning rates.

I have also tried different types of n-tuples. From figure 11, we can see that using group 1 and 2 converges faster than group 0. Though they provide relatively few features and has smaller maximum and mean scores than group 0, their 2048 win rate can be achieved to roughly 90% after training $200,000$ episodes. Therefore, I also perform an experiment with group 0 and 1 together, and it achieves the highest among all scores.

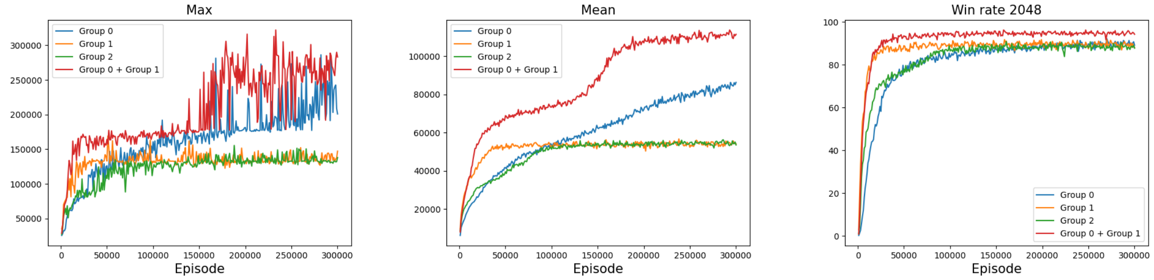


Figure 11: Learning curves with different n-tuples.