Identifying AI’s Strategy in µRTS using CNN over Game States

Students

- Guy Danielli
- Nitzan Farhi

Abstract

In the course “Artificial Intelligence in Game Theory” given by Dr. Kobi Gal at Ben Gurion University of the Negev we chose our project topic to be in the field of RTS games. The problem we chose is to be able to detect an AI’s strategy during a game, out of a pool of known strategies using a Convolutional Neural Network, regarding the SCV agent. We chose to focus on identifying the player’s strategy rather than the map, although, our method can be easily extended to also detect the played map.

Background

In the original paper: “Strategy Generation for Multi-Unit Real-Time Games via Voting”, player strategy identification was done by counting the number of units of each type owned by the player and feeding them into a naïve Bayes algorithm model. The number of resources owned by the player was also considered as a feature but was removed in the feature selection process by using information gain. We find this method to be a naïve one, since it only counts units and does not consider the strategic view of the map.

Methodology

1. Classification

For the classification task we used a CNN[1]. The CNN consists of the following layers:

1. Convolutional layer for local feature extraction
   a. Size: 16x16 with 10 filters (i.e. depth of 10)
   b. Relu activation
2. Max Pooling layer to reduce the dimensionality of the input with pooling size 4x4
3. Fully connected layer with 128 neurons for higher level features
4. Fully connected layer with Softmax activation for classification

The loss function we used is Categorical Crossentropy. We explain the architecture below.

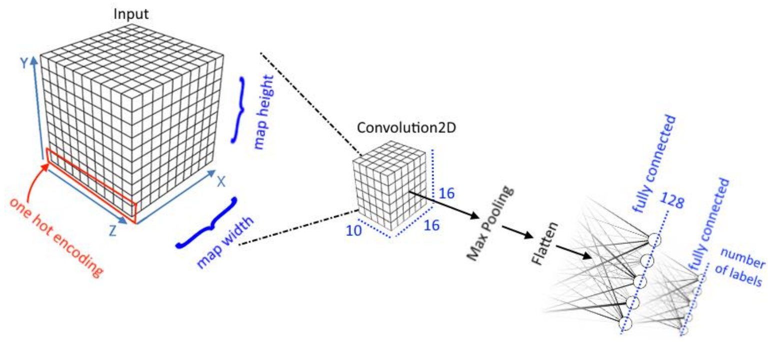

Figure 1: CNN and Input architecture

In neural networks, a convolutional layer is a type of layer where each neuron is connected to a local group of neurons in the previous layer. Convolutional layers are often used in tasks concerning images and have proven to be very efficient. A max-pooling layer is a functional layer (i.e. it applies a hard-coded function instead of using neurons) which outputs the maximal activation value of a local group of neurons. Max pooling layers are often found together with convolutional layers, and they have proven efficient in reducing dimensionality.

In classification tasks, it is customary to apply Softmax activation function on the output, as it normalizes the output vector so that the sum of all values is 1. This way, the ith entry in the output vector can be considered the probability given by the model for the input being in the ith class.

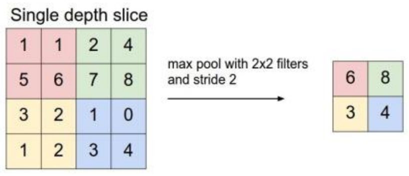

Figure 2: Max-Pooling layer with 2x2 pooling size

[1] Convolutional Neural Network

2. State Encoding

As mentioned above, we used a CNN to classify opponents based on game state snapshots of games where they participated in. A snapshot of the game contains the terrain of the map (which indicates for each position on the map if it is a wall or an empty square), the neutral units (which can only be Resources in the case of Micro-RTS) and the players' units. As input to the CNN we encoded the snapshot into a 3-dimensional array, where the array's first two dimensions mirror the map 's X and Y axes, and the 3rd dimension is used for units' encoding. As we will describe below, some units were ignored in some cases. We also ignored the terrain of the map (in reality, we only used maps which didn't contain walls). We define two different encoding methods which we will compare in the results and discussion sections: ***Ignore*** and ***Indicate***. Both methods are described below.

1. Ignore

The intuition behind this method is that a game state contains considerable amounts of noise regarding one's strategy: resources, walls, other players' units – all of those might be statistically independent of the player's own units and their scatter on the map at least in some cases. Our method when using ***Ignore*** is to only encode the player's friendly units into the snapshot to feed into the neural network, everything else is ignored (that includes both other players' units and neutral units). The encoding in this case is a plain onehot encoding of the unit type. The size of the 3rd dimension in the snapshot is then 6 (there are 6 types of units which are not neutral).

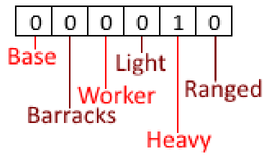

*Figure 3: OneHot encoding of a heavy unit in **Ignore***

2. Indicate

Using ***Indicate*** we encode everything into the snapshot. We also add an extra bit in the encoding which indicates the owner of a unit: 0 for a resource, 1 for a friendly unit, -1 for an opponent unit (a friendly unit belongs to the player we are trying to classify – ultimately, this is actually an opponent unit of the SCV agent). The size of the encoding in this case is 8: +1 for the resource type which we didn't have in ***Ignore*** and +1 for the "owner bit".

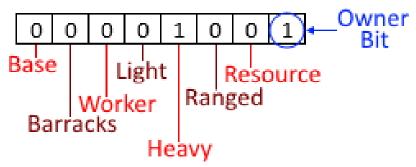

*Figure 4: OneHot encoding of a friendly heavy unit in **Indicate***

Empirical Methodology

1. Datasets

To evaluate our model, we generated several datasets. The datasets were created by running Fixed Tournaments in Micro-RTS. Custom tournaments in Micro-RTS can be managed by using changing several parameters. The more interesting ones are as follows:

- List of competing AIs (player 0, i.e. left player).
- List of opponents (player 1, i.e. right player).
- Number of iterations – each tournament can run each game several times. Each independent run of all the games in the tournament is called an *Iteration*.
- List of maps to play each game.

Each dataset we created consists of a structured subset and a random subset, generated from two different tournaments. In the random subset, each AI plays against the AIs “RandomAI” and “RandomBiasedAI”. In the structured subset, each AI plays against each of the other AIs and against itself. We used the *Iterations* parameter to increase the size of the random subsets by running more games. In the structured subsets, on the other hand, we used a single iteration. The parameters which differ between our datasets are as follows:

- AIs - the list of AIs (labels).
- Random Iterations - the number of iterations used to create the random subset.
- Sampling Frequency – the frequency in which we sampled game states, e.g. every 10 decision points.

In our experiments, the profiled player is always player 0, i.e. the left player in the map. When using the model in a real game, if the opponent is not the left player, the input of the model can be mirrored or rotated before being classified.

1. RUSH_RND2

This is a simple dataset. The AIs in this dataset are: WorkerRush, LightRush, HeavyRush and RangedRush. The number of random iterations is 2 and we sampled a game state every 5 decisions.

2. RUSH_RND5

This is a larger version of RUSH_RND2. It is identical to RUSH_RND2 in terms of the set of AIs and sampling frequency. The number of random iterations in this dataset is 5.

3. RUSH_CRUSH_PO_RND5

The AIs in this dataset are: WorkerRush, LightRush, HeavyRush, RangedRush, POWorkerRush, POLightRush, POHeavyRush, PORangedRush and CRush_V1. The number of random iterations is 5, and a game state was sampled every 15 decision points.

4. MIXED

The AIs in this dataset are: WorkerRush, LightRush, HeavyRush, RangedRush, CRush_V1, PGSAI, IDRTMinimax, MonteCarlo, PortfolioAI and PVAIML_ED. Here a state was sampled every 15 decision points.

2. Experiments

For each dataset, we used the random subset as training set and the structured subset as testing set. That way we can have a more diverse training set, which we can increase in size with possibly a fewer duplicates than by using a structured dataset (a game of AI1 vs. AI2 is not random in most cases). The fact that we played different games helped us avoid overfitting. This also usually resulted in a test set larger or equally large to the train set.

One interesting property this classification problem has is that at the start of the game, and possibly at the end of the game, all the AIs have similar units. Therefore, it is hard to discriminate between strategies at certain points in the game. Our way to cope with this problem is to constraint the minimum number of friendly units (i.e. units owned by the classified player) in snapshots we use for training/testing. If the number of friendly units in a snapshot is smaller than the define minimum, we ignore it during the experiment. Experiments differ from each other in:

- Unit encoding style: ***Ignore*** vs. ***Indicate***
- Minimum number of friendly units (=min_units)

Results

Figures 5-7 describe the results of our most recent experiments. We can clearly see that the RUSH_RND datasets were the easiest to classify. In both RUSH_RND2 and RUSH_RND5 our CNN reached a 100% accuracy when the ***Ignore*** method was used. We believe the reason for this is that WorkerRush, LightRush, HeavyRush and RangedRush are very distinct from each other in that each strategy has a unique unit that only that strategy chooses to create, with the exception of workers, which WorkerRush create much more of than the other strategies.

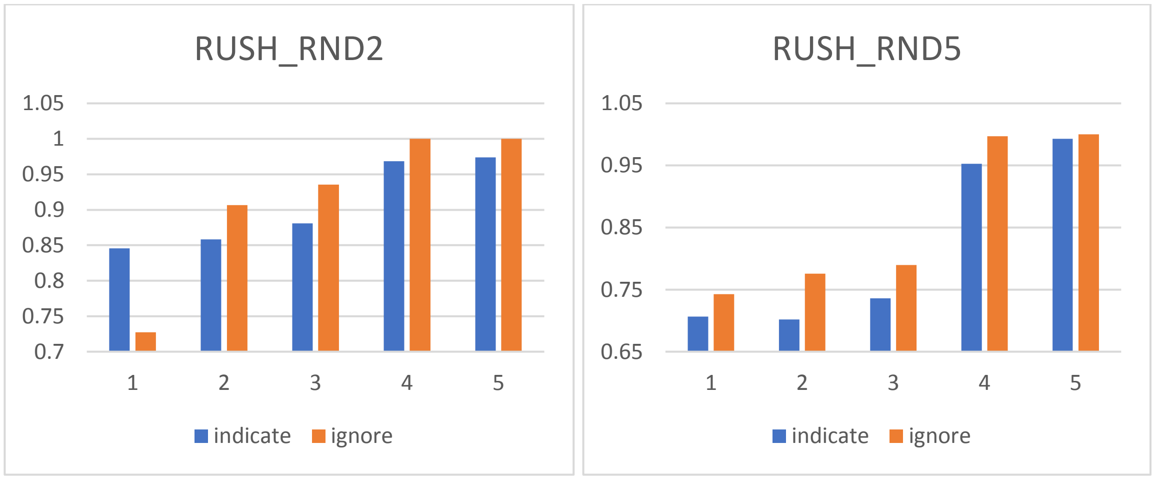

Figure 5: RUSH_RND results

The RUSH_CRUSH_PO_RND5 dataset was the hardest to classify. As can be seen in the figure below, the highest accuracy we reached is slightly above 56%. After looking into the problem a bit we found out that Rush strategies and their consequent PORush strategies are very similar. For example, a game of WorkerRush versus POWorkerRush results in a totally symmetrical board from game start to end. In this case there is no way to discriminate between the two strategies. In this case, always identifying strategies as PORush, or alternatively, normal Rush, is similar to reducing our problem to the RUSH_RND datasets while limiting ourselves to 50% accuracy. That together with the ability to at least partially being able to identify CRush_V1 lets us to reach above about 60% max. That explains why we could reach 56% accuracy at best, although we didn't inspect misclassifications.

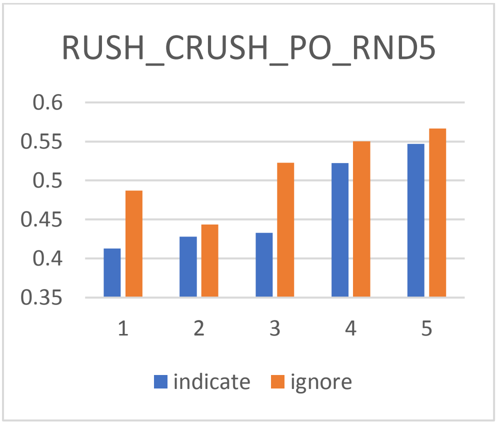

Figure 6

Generating the MIXED dataset was the most computationally expensive task in our project. Some of the agents take much time to play, and one of the agents (LSI) didn't make it into the dataset because they passed their rationed computation time in every turn. Ultimately, we had to stop this dataset's generation before it finished. With this setup, min_units=4 was enough to reach 87% accuracy, although, we believe that generating more samples would be enough to greatly improve the model, as we saw a considerable improvement to our results as we ran intermediate experiments while the dataset was generated and have been increasing in size.

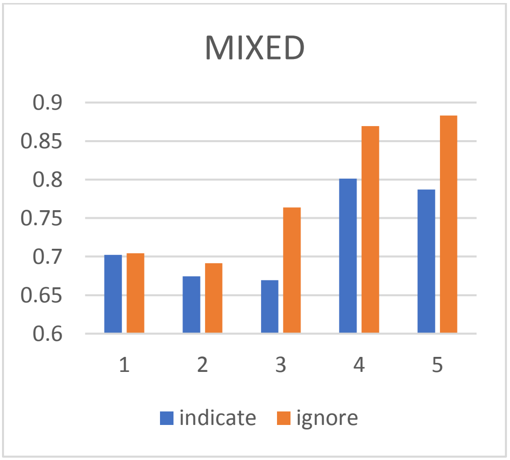

Figure 7

Another result we can clearly see is that the ***Ignore*** method had better results than ***Indicate***, sometimes with significant difference. This can be a result of lack of data, or the fact that in the training set the right player played random moves, and therefore didn't represent well the snapshots in test set. Perhaps using cross-validation is more fitting in this case. We can also conclude that the more units a player has, the easier it is to figure out their strategy.

Discussion and Future Work

Speed

Using our method in games is definitely an option: the classification speed of the neural network ranges from 4000 to 8000 samples per second on Nvidia Quadro K1100M GPU, or 2000 samples per second on CPU (much faster than needed). Although we reached low accuracy in some cases, in those cases the misclassified strategies practically played the same moves (although we need to further check that). Therefore, misclassification won't result in a game's loss.

Future Work

We suggest that in the future, our methods will be tested with larger datasets, perhaps by using cross-validation. The sizes of our current datasets indicate that 2-fold cross validation might already be enough to achieve decent results. Our method can also be extended to identify maps, and we suggest that it will be tested in tournaments. We also suggest combining it with temporal features regarding a player's actions to identify their strategy.

Another idea is to use a CNN with an encoder-decoder architecture to create a game state encoder, then use it to create a map (i.e. graph) of states and transition between them. This map can then be used to search using DFS for the best strategy to play during a game.