Hsin-Lei Charlene Wang

Artificial Intelligence Capstone

Fall 2019

Tron Capstone

Overview

I will explain the **motivations** behind the overall architecture design, **model design and architecture**, and **performance gains** in the Research/Implementation section. The motivations behind how the bot works will be addressed in the README section.

Research/Implementation process

The problem is framed in two different approaches: reinforcement learning and supervised learning.

1. Reinforcement Learning: **DQN**
2. Model architecture and Performance
3. Competitive Analysis of All Supplied Bots
4. Supervised Learning: **Policy Network**
5. Final Model & Learned Weights

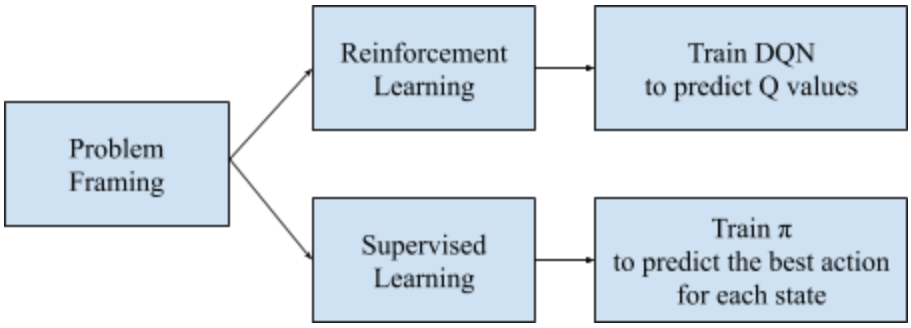


Image 1

Reinforcement Learning: DQN

Recent successes such as AlphaGo and AlphaGo Zero spike a trend of training deep neural networks on large amount of self-play data to learn better representations [1]. As I designed the strategy for Tron, I began to look at how reinforcement learning trains agents to play single, adversarial and multi-agent games effectively.

In the reinforcement learning homework, we implemented the SARSA and SARSA-lambda algorithms to store state-action values in the Q table. Those Q-learning algorithms work best when there are a few states. However, in the case of Tron, the problem space is too large for Tabular Q-learning algorithms because each game cell can have many cell_types and powerup_types. The combinations of all possible states are too big to store in the Q table. Therefore, I started to look for deep learning solutions that use neural networks to predict q-values without storing all the state-action pairs. Instead of looking up q-values in a Q table, the neural network would develop an estimation function that can predict the q-value of any state-action pair. Now the question is which framework to use.

After reading many popular papers on deep neural networks, I decided to go with a simple, elegant model called **Deep Q-learning** or **DQN**. DQN "connects a reinforcement learning algorithm to a deep neural network which operates directly on RGB images and efficiently process training data by using stochastic gradient updates" [2]. DQN stores the agent's experience in an N-size buffer, called **experience replay**, at each time-step [3]. Then, Q-learning is applied to some samples of experience from the replay buffer. Afterward, the agent selects an action that would maximize the reward. DQN trains samples from the replay buffer to fulfill the i.i.d. (independent and identically distributed) requirement.

With DQN, I built a large and a small neural network to predict q-value. Since the nature of the game board is similar to an image, I decided to go with the convolution neural network used by AlphaGo.

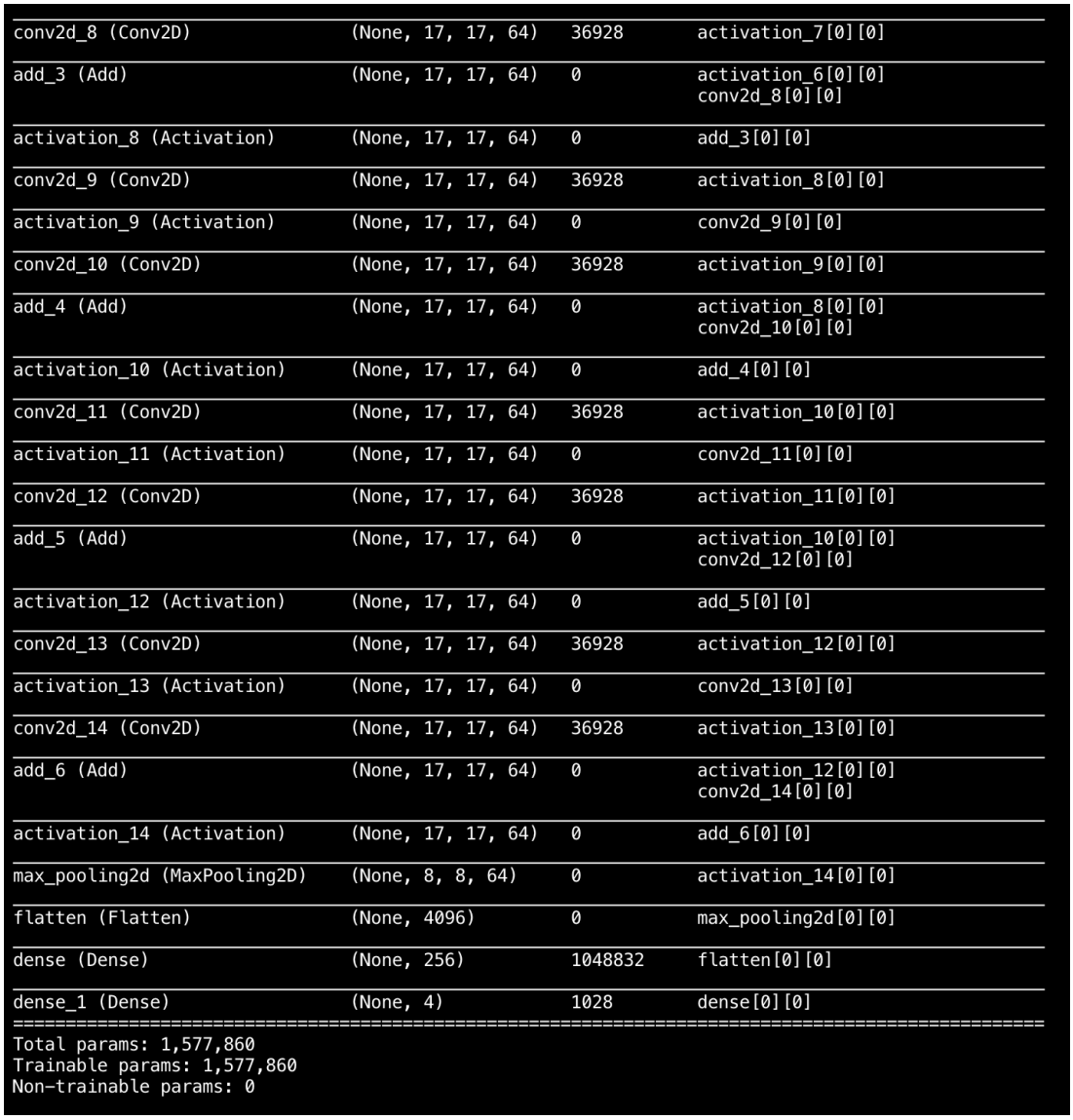
Model architecture and Performance

AlphaGo uses 192 filters for each conv2d layer, which would be extremely slow to train. Hence, I reduced it to 64 filters in each conv2d layer. Here is the large model architecture similar to AlphaGo's CNN:

```
training mode
Model: "model"
```

Layer (type)	Output Shape	Param #	Connected to
input_1 (InputLayer)	[(None, 17, 17, 19)]	0	
conv2d (Conv2D)	(None, 17, 17, 64)	11008	input_1[0][0]
activation (Activation)	(None, 17, 17, 64)	0	conv2d[0][0]
conv2d_1 (Conv2D)	(None, 17, 17, 64)	36928	activation[0][0]
activation_1 (Activation)	(None, 17, 17, 64)	0	conv2d_1[0][0]
conv2d_2 (Conv2D)	(None, 17, 17, 64)	36928	activation_1[0][0]
add (Add)	(None, 17, 17, 64)	0	activation[0][0] conv2d_2[0][0]
activation_2 (Activation)	(None, 17, 17, 64)	0	add[0][0]
conv2d_3 (Conv2D)	(None, 17, 17, 64)	36928	activation_2[0][0]
activation_3 (Activation)	(None, 17, 17, 64)	0	conv2d_3[0][0]
conv2d_4 (Conv2D)	(None, 17, 17, 64)	36928	activation_3[0][0]
add_1 (Add)	(None, 17, 17, 64)	0	activation_2[0][0] conv2d_4[0][0]
activation_4 (Activation)	(None, 17, 17, 64)	0	add_1[0][0]
conv2d_5 (Conv2D)	(None, 17, 17, 64)	36928	activation_4[0][0]
activation_5 (Activation)	(None, 17, 17, 64)	0	conv2d_5[0][0]
conv2d_6 (Conv2D)	(None, 17, 17, 64)	36928	activation_5[0][0]
add_2 (Add)	(None, 17, 17, 64)	0	activation_4[0][0] conv2d_6[0][0]
activation_6 (Activation)	(None, 17, 17, 64)	0	add_2[0][0]
conv2d_7 (Conv2D)	(None, 17, 17, 64)	36928	activation_6[0][0]
activation_7 (Activation)	(None, 17, 17, 64)	0	conv2d_7[0][0]

Cont.



conv2d_8 (Conv2D)	(None, 17, 17, 64)	36928	activation_7[0][0]
add_3 (Add)	(None, 17, 17, 64)	0	activation_6[0][0] conv2d_8[0][0]
activation_8 (Activation)	(None, 17, 17, 64)	0	add_3[0][0]
conv2d_9 (Conv2D)	(None, 17, 17, 64)	36928	activation_8[0][0]
activation_9 (Activation)	(None, 17, 17, 64)	0	conv2d_9[0][0]
conv2d_10 (Conv2D)	(None, 17, 17, 64)	36928	activation_9[0][0]
add_4 (Add)	(None, 17, 17, 64)	0	activation_8[0][0] conv2d_10[0][0]
activation_10 (Activation)	(None, 17, 17, 64)	0	add_4[0][0]
conv2d_11 (Conv2D)	(None, 17, 17, 64)	36928	activation_10[0][0]
activation_11 (Activation)	(None, 17, 17, 64)	0	conv2d_11[0][0]
conv2d_12 (Conv2D)	(None, 17, 17, 64)	36928	activation_11[0][0]
add_5 (Add)	(None, 17, 17, 64)	0	activation_10[0][0] conv2d_12[0][0]
activation_12 (Activation)	(None, 17, 17, 64)	0	add_5[0][0]
conv2d_13 (Conv2D)	(None, 17, 17, 64)	36928	activation_12[0][0]
activation_13 (Activation)	(None, 17, 17, 64)	0	conv2d_13[0][0]
conv2d_14 (Conv2D)	(None, 17, 17, 64)	36928	activation_13[0][0]
add_6 (Add)	(None, 17, 17, 64)	0	activation_12[0][0] conv2d_14[0][0]
activation_14 (Activation)	(None, 17, 17, 64)	0	add_6[0][0]
max_pooling2d (MaxPooling2D)	(None, 8, 8, 64)	0	activation_14[0][0]
flatten (Flatten)	(None, 4096)	0	max_pooling2d[0][0]
dense (Dense)	(None, 256)	1048832	flatten[0][0]
dense_1 (Dense)	(None, 4)	1028	dense[0][0]

Total params: 1,577,860
Trainable params: 1,577,860
Non-trainable params: 0

Image 2

However, this giant architecture is slow to train and doesn't reduce the loss function efficiently. Therefore, I **reduced the large model into a smaller one**. Specifically, I took out 7 residual blocks, simplified 14 convolution layers into 2, and excluded skipped connections. Here is a summary of the small CNN:

```
Model: "model"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         [(None, 17, 17, 19)]      0
_________________________________________________________________
conv2d (Conv2D)              (None, 17, 17, 16)        24640
_________________________________________________________________
activation (Activation)      (None, 17, 17, 16)        0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 17, 17, 32)        4640
_________________________________________________________________
activation_1 (Activation)    (None, 17, 17, 32)        0
_________________________________________________________________
flatten (Flatten)            (None, 9248)              0
_________________________________________________________________
dense (Dense)                (None, 256)               2367744
_________________________________________________________________
dense_1 (Dense)              (None, 4)                 1028
=================================================================
Total params: 2,398,052
Trainable params: 2,398,052
Non-trainable params: 0
_________________________________________________________________
```

Image 3

My **bot performances** trained on each CNN are shown below.

$SmallNetwork$	$(student, random)$	$(student, wall)$
$empty\_room$	$(60, 40)$	$(5, 95)$
$joust$	$(22, 78)$	$(56, 44)$
$divider$	$(31, 69)$	$(52, 48)$
$hunger\_games$	$(69, 31)$	$(77, 23)$

$LargeNetwork$	$(student, random)$	$(student, wall)$
$empty\_room$	$(64, 36)$	$(31, 69)$
$joust$	$(51, 49)$	$(28, 72)$
$divider$	$(60, 40)$	$(17, 83)$
$hunger\_games$	$(66, 34)$	$(38, 62)$

Image 4

When trained on the *small network*, my bot struggles on joust and divider when competed with the random bot. When competed with the wall bot, my bot only struggles at the empty_room.

When trained on the *large network*, my bot **consistently beats the random bot and consistently loses to the wall bot**. While the performance versus wall bot is worse than the small network, the large network has more consistency across. Therefore, I decided to let the bot trained on a large network compete with ta bots.

$LargeNetwork$	$(student, ta1)$	$(student, ta2)$
$empty\_room$	$(3, 97)$	$(1, 99)$
$joust$	$(26, 74)$	$(7, 93)$
$divider$	$(38, 62)$	$(19, 81)$
$hunger\_games$	$(18, 82)$	$(11, 89)$

Image 5

Sadly, my bot not only loses to beat wall bot but also loses to ta1 and ta2 entirely.

Given the superior performance of the supplied bots, I first experimented with different reward structures. For example, I compared only rewarding the last step versus rewarding every step. While rewarding the last step is the most common approach in reinforcement learning games, I want to know if there are any merits in keeping the bot on the Tron game for long. This novel approach did not improve the performance much either.

Therefore, I began to analyze their performances and research other ways to learn from the bots.

Competitive Analysis of Supplied Bots

I conducted the following study to see which bot has the best move. From the table below, *ta2* outperforms ta1, random, and wall bots in most cases. There is one scenario (divider) where the wall bot slightly outperforms *ta2*, so I plan to learn from ta2 bot and pick up some strategy from the wall bot.

The following tables show the number of wins out of 100 games.

(a) $ta1$ vs $ta2$: $ta2$ consistently outperform $ta1$ on all maps.

Map	$ta1$	$ta2$
$empty_room$	6	94
$joust$	39	61
$divider$	48	52
$hunger\_games$	30	70

(b) $ta1$ vs $random$ and $wall$: While $ta1$ consistently outperform $random$ and $wall$, $ta1$ takes the most hit on the divider map.

Map	$(ta1, random)$	$(ta1, wall)$
$empty_room$	$(98, 2)$	$(85, 15)$
$joust$	$(88, 12)$	$(71, 29)$
$divider$	$(70, 30)$	$(67, 33)$
$hunger\_games$	$(92, 8)$	$(73, 27)$

(c) $ta2$ vs $random$ and $wall$: While $ta1$ consistently outperform $random$ bots, $ta2$ lost $wall$ on the divider map.

Map	$(ta2, random)$	$(ta2, wall)$
$empty_room$	$(100, 0)$	$(99, 1)$
$joust$	$(90, 10)$	$(74, 26)$
$divider$	$(75, 25)$	$(47, 53)$
$hunger\_games$	$(91, 9)$	$(89, 11)$

Image 6

Supervised Learning: Policy Network

AlphaGo trains a “supervised learning policy network directly from expert human moves in the Go game” [4].

Extended Data Table 2 | Input features for neural networks

Feature	# of planes	Description
Stone colour	3	Player stone / opponent stone / empty
Ones	1	A constant plane filled with 1
Turns since	8	How many turns since a move was played
Liberties	8	Number of liberties (empty adjacent points)
Capture size	8	How many opponent stones would be captured
Self-atari size	8	How many of own stones would be captured
Liberties after move	8	Number of liberties after this move is played
Ladder capture	1	Whether a move at this point is a successful ladder capture
Ladder escape	1	Whether a move at this point is a successful ladder escape
Sensibleness	1	Whether a move is legal and does not fill its own eyes
Zeros	1	A constant plane filled with 0
Player color	1	Whether current player is black

Input features to AlphaGo’s Policy Network [4]

In a similar spirit, I am going to learn from the best player (ta2) in the Tron game with the feature described below. To mimic the behavior from all supplied bots, I started to train the policy π, which predicts an action for each state ($\pi: s \rightarrow a$). The goal is to let π learn from the master moves and best players, so it can become the best player too.

Preprocessing

- **Data generation**: I created *selfplay.py* which would generate the following training datasets:
    - ta2 vs (rand, wall, ta1, ta2) on all maps as well as the reverse order
    - ta1 vs (rand, wall, ta1, ta2) on all maps as well as the reverse order
- **Training labels**:
    - X=19 planes (see below)
    - Y=numpy array((4,)) with one-hot encoding for the 4 classes
- **Training/Validation/Testing Split:** 70/20/10
- **Augmentation**: I augmented the training data by apply coordinate transformations, i.e. "flipping" the board along various axes. The final augmented list of (state, action) is flipped in 8 different ways.
- **Data size:** 200 (times) x 4 (maps) x 8 (augmentation) x 60 (moves/game) = ~400k training samples for each round.

Model Training

1. **Feature Selection**: 19 planes (number of channels in the convolutional layers). The planes represent the following information:
    1.1. walls = 1
    1.2. barriers = 1
    1.3. current_player_position = 1 (only one)
    1.4. opponent_position = 1 (only one)
    1.5. speed_locations = 1
    1.6. bomb_locations = 1
    1.7. trap_loc = 1
    1.8. armor_loc = 1
    1.9. current_player_has_armor = ones
    1.10. opponent_has_armor = ones
    1.11. current_player_has_speed_with_four_steps
    1.12. current_player_has_speed_with_three_steps
    1.13. current_player_has_speed_with_second_steps
    1.14. current_player_has_speed_with_one_steps
    1.15. opp_player_remaining_speed == 4

1.16. opp_player_remaining_speed == 3
1.17. opp_player_remaining_speed == 2
1.18. opp_player_remaining_speed == 1
1.19. is_current_player_one

2. **Learning rate:** Use the default setting of the Keras.Adam optimizer, which would be automatically adapted as epoch number increases.
3. **Optimizer:** I chose Adam optimizer because of its popularity and ability to adapt the learning rate on its own.
4. To optimize the weights in the most efficient manner, I trained many versions of the models on each bot and map while varying the epoch (5, 50, 100, 500) and batch_size (32, 64, 128). The sweet spot happens when **epoch=50** and **batch_size = 64**.
5. To combat overfitting, I started with strong regularization (L2(1e-4) for kernel_regularizer and two Dropout layers), data augmentation, and simple, small architecture.

The small CNN's model architecture is the same as the small CNN (image 3) used by DQN; the only difference is that the final layer is replaced with softmax for multiclass classification (four actions).

The following performance (Image 7) is a result of three training stages: ta2 self-play, ta2 x ta1 battle, and a different ta2 self-play.

$student$	$(student, random)$	$(student, wall)$
$empty\_room$	$(94, 6)$	$(50, 50)$
$joust$	$(82, 18)$	$(45, 55)$
$divider$	$(73, 27)$	$(29, 71)$
$hunger\_games$	$(89, 11)$	$(27, 73)$

Image 7

After training for a few hours, the training and validation loss converged to 0.7 and the performance (Image 8) improved:

	Empty	Divider	Hunger	Joust
VS bots.RandBot'>:	86%	77%	88%	87%
VS bots.WallBot'>:	80%	44%	79%	61%
VS ta_bots.TABot1'>:	15%	46%	46%	47%
VS ta_bots.TABot2'>:	21%	45%	38%	25%

Image 8

Given that the loss score has converged to about 0.7, I decided to create a larger CNN to utilize the power of the model.

The model architecture is the same as the large CNN (image 2) used by DQN; the only difference is that the final layer is replaced with softmax for multiclass classification (four actions). Due to architectural differences, I trained the large CNN with ta2 self-play from scratch. While the large CNN takes longer to train, the performance (Image 9) after a few epochs is promising:

$student$	$(student, random)$	$(student, wall)$
$empty\_room$	$(92, 8)$	$(76, 24)$
$joust$	$(82, 18)$	$(61, 39)$
$divider$	$(85, 15)$	$(59, 41)$
$hunger\_games$	$(87, 13)$	$(76, 24)$

Image 9

Policy Network with large CNN achieves the best performance so far, so I continue running it for hours. Here is a snapshot with 12 hours of training:

	Empty	Divider	Hunger	Joust
VS bots.RandBot'>:	97%	81%	92%	88%
VS bots.WallBot'>:	68%	65%	84%	67%
VS ta_bots.TABot1'>:	46%	61%	48%	50%
VS ta_bots.TABot2'>:	25%	51%	43%	39%

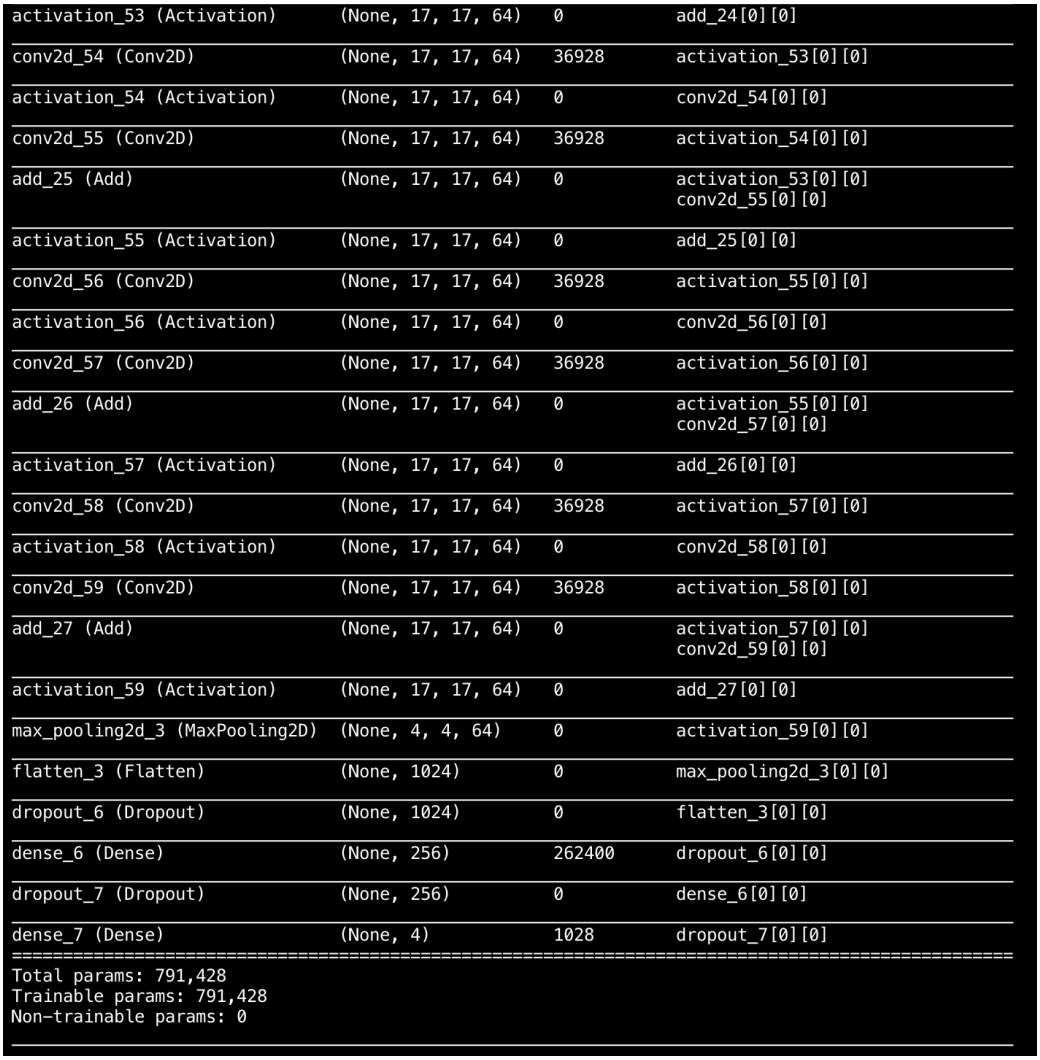
Image 10

Final Model & Learned Weights

With the outstanding performance of Policy Network trained on large CNN, I chose it (stored in `largenetwork_policy`) to be the final strategy. The large CNN architecture (Image 11) is **quite similar** to DQN's large CNN architecture (Image 2). The only difference is the final layer. DQN uses linear for regression while Policy Network uses **softmax for multiclass classification**. The multiclass classification shows the probability for all four directions, and the agent will select the one with the **highest probability**.

Layer (type)	Output Shape	Param #	Connected to
input_4 (InputLayer)	[(None, 17, 17, 19)]	0	
conv2d_45 (Conv2D)	(None, 17, 17, 64)	11008	input_4[0][0]
activation_45 (Activation)	(None, 17, 17, 64)	0	conv2d_45[0][0]
conv2d_46 (Conv2D)	(None, 17, 17, 64)	36928	activation_45[0][0]
activation_46 (Activation)	(None, 17, 17, 64)	0	conv2d_46[0][0]
conv2d_47 (Conv2D)	(None, 17, 17, 64)	36928	activation_46[0][0]
add_21 (Add)	(None, 17, 17, 64)	0	activation_45[0][0] conv2d_47[0][0]
activation_47 (Activation)	(None, 17, 17, 64)	0	add_21[0][0]
conv2d_48 (Conv2D)	(None, 17, 17, 64)	36928	activation_47[0][0]
activation_48 (Activation)	(None, 17, 17, 64)	0	conv2d_48[0][0]
conv2d_49 (Conv2D)	(None, 17, 17, 64)	36928	activation_48[0][0]
add_22 (Add)	(None, 17, 17, 64)	0	activation_47[0][0] conv2d_49[0][0]
activation_49 (Activation)	(None, 17, 17, 64)	0	add_22[0][0]
conv2d_50 (Conv2D)	(None, 17, 17, 64)	36928	activation_49[0][0]
activation_50 (Activation)	(None, 17, 17, 64)	0	conv2d_50[0][0]
conv2d_51 (Conv2D)	(None, 17, 17, 64)	36928	activation_50[0][0]
add_23 (Add)	(None, 17, 17, 64)	0	activation_49[0][0] conv2d_51[0][0]
activation_51 (Activation)	(None, 17, 17, 64)	0	add_23[0][0]
conv2d_52 (Conv2D)	(None, 17, 17, 64)	36928	activation_51[0][0]
activation_52 (Activation)	(None, 17, 17, 64)	0	conv2d_52[0][0]
conv2d_53 (Conv2D)	(None, 17, 17, 64)	36928	activation_52[0][0]
add_24 (Add)	(None, 17, 17, 64)	0	activation_51[0][0] conv2d_53[0][0]

Cont.



activation_53 (Activation)	(None, 17, 17, 64)	0	add_24[0][0]
conv2d_54 (Conv2D)	(None, 17, 17, 64)	36928	activation_53[0][0]
activation_54 (Activation)	(None, 17, 17, 64)	0	conv2d_54[0][0]
conv2d_55 (Conv2D)	(None, 17, 17, 64)	36928	activation_54[0][0]
add_25 (Add)	(None, 17, 17, 64)	0	activation_53[0][0] conv2d_55[0][0]
activation_55 (Activation)	(None, 17, 17, 64)	0	add_25[0][0]
conv2d_56 (Conv2D)	(None, 17, 17, 64)	36928	activation_55[0][0]
activation_56 (Activation)	(None, 17, 17, 64)	0	conv2d_56[0][0]
conv2d_57 (Conv2D)	(None, 17, 17, 64)	36928	activation_56[0][0]
add_26 (Add)	(None, 17, 17, 64)	0	activation_55[0][0] conv2d_57[0][0]
activation_57 (Activation)	(None, 17, 17, 64)	0	add_26[0][0]
conv2d_58 (Conv2D)	(None, 17, 17, 64)	36928	activation_57[0][0]
activation_58 (Activation)	(None, 17, 17, 64)	0	conv2d_58[0][0]
conv2d_59 (Conv2D)	(None, 17, 17, 64)	36928	activation_58[0][0]
add_27 (Add)	(None, 17, 17, 64)	0	activation_57[0][0] conv2d_59[0][0]
activation_59 (Activation)	(None, 17, 17, 64)	0	add_27[0][0]
max_pooling2d_3 (MaxPooling2D)	(None, 4, 4, 64)	0	activation_59[0][0]
flatten_3 (Flatten)	(None, 1024)	0	max_pooling2d_3[0][0]
dropout_6 (Dropout)	(None, 1024)	0	flatten_3[0][0]
dense_6 (Dense)	(None, 256)	262400	dropout_6[0][0]
dropout_7 (Dropout)	(None, 256)	0	dense_6[0][0]
dense_7 (Dense)	(None, 4)	1028	dropout_7[0][0]

Total params: 791,428
Trainable params: 791,428
Non-trainable params: 0

Image 11

README for Tron Bot

Running Unit Tests

```
# run_tests.py runs tests on all combinations of bots and maps
python run_tests.py
```

Training Policy Network

- Collect moves for replay by running the games with specified bot pairs with `python selfplay.py`
- Once the raw training data is created, run `python trainingdatageneration.py` to take in the data from the replay history, augment it and convert it into training data (numpy arrays).
- `Largenetwork.py` trains the CNN and stores the best weights in checkpoint files.
- In `bots.py`, you can see how the trained Policy Network works. I first load the previously trained weights. Then, I use the trained policy network to decide the best action and conduct a safety check on the selected action. The safety check rules out suicidal actions.

Past Attempt

Note that this approach is not used in the final submission. The DQN training model can be found in `trainingdqn.py` with the following hyperparameters:

- `EPOCHS=100`
- `BATCH_SIZE = 32,`
- `REPLAY_BUFFER_SIZE = 20000,`
- `GAMMA = 0.999,`
- `EPSILON_DECAY = 0.99,`
- `EPSILON_MIN = 0.1,`
- `EPISODES = 500,`
- `TARGET_NETWORK_UPDATE_INTERVAL = 10,`
- `CHECK_PERFORMANCE_INTERVAL = 50`

Furthermore, I used an **epsilon-greedy** approach, starting with a high epsilon (1.0), which anneals to 0.01 over the episodes. This is to encourage exploration at the beginning and exploitation in the later stage.

Shortcomings

1. The model achieved a decent performance: validation loss=0.6547 and validation accuracy=0.6174. Therefore, it can't perfectly mimic TA2 bot's moves yet.
2. The current approach blindly mimics TA2 bot's moves, so at best it would beat TA2 bot for 50% of the time.

Potential areas of improvement

If I had more time, I would do the following:

- Decrease the training time with Episodic Backward Updates for DQN [5].
- Augment deep learning-based approach with searches and heuristics (e.g. A*) when there are limited steps.
- Monte Carlo Tree Search
- Ensemble several successful approaches I tried.
- Boost the model performance with feature engineering

If I had access to GPU, I would do the following:

- Train the large CNN from both approaches on more data
- Further increase the model power via architecture
- Try different sets of hyperparameter tuning to reach the sweet spot.
- Try strategies for adversarial and multi-agent DQN games, such as Opponent Modeling in Deep Reinforcement Learning [6].

References

[1] Alex Krizhevsky, Ilya Sutskever, and Geoff Hinton. Imagenet classification with deep convolutional neural networks. In Advances in Neural Information Processing Systems 25, pages 1106–1114, 2012.

[2] Mnih, Volodymyr et al. “Playing Atari with Deep Reinforcement Learning.” ArXiv abs/1312.5602 (2013): n. Pag.

[3] Long-Ji Lin. Reinforcement learning for robots using neural networks. Technical report, DTIC Document, 1993.

[4] Silver, David & Huang, Aja & Maddison, Christopher & Guez, Arthur & Sifre, Laurent & Driessche, George & Schrittwieser, Julian & Antonoglou, Ioannis & Panneershelvam, Veda & Lanctot, Marc & Dieleman, Sander & Grewe, Dominik & Nham, John & Kalchbrenner, Nal & Sutskever, Ilya & Lillicrap, Timothy & Leach, Madeleine & Kavukcuoglu, Koray & Graepel, Thore & Hassabis, Demis. (2016). Mastering the game of Go with deep neural networks and tree search. Nature. 529. 484-489. 10.1038/nature16961.

[5] Lee, Su & Sungik, Choi & Chung, Sae-Young. (2018). Sample-Efficient Deep Reinforcement Learning via Episodic Backward Update.

[6] He, He & Boyd-Graber, Jordan & Kwok, Kevin & Daumé, III. (2016). Opponent Modeling in Deep Reinforcement Learning.