Machine Learning in RTS game

A/Prof Richard Yi Da Xu, Chen Deng
Yida.Xu@uts.edu.au, Chen.Deng@student.uts.edu.au
notes can be downloaded: `richardxu.com`

University of Technology Sydney (UTS)

April 12, 2019

Contents

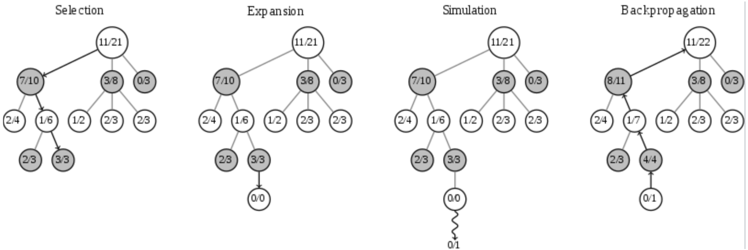
- our research expertise
- How does AlphaGo and AlphaGo Zero work?
- what may the challenge be to apply them directly to RTS game?
- some of our proposed research

- this work is carried out jointly with my team members:

  Dr Jason Traish, Mr (Dr-soon-to-be) Joshua Brown, Mr David Cotton

Monte Carlo Tree Search (MCTS)

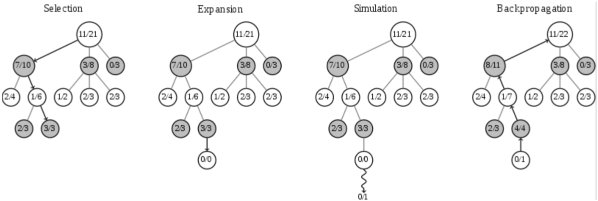
- first, how does a traditional MCTS work?, diagram from Wikipedia

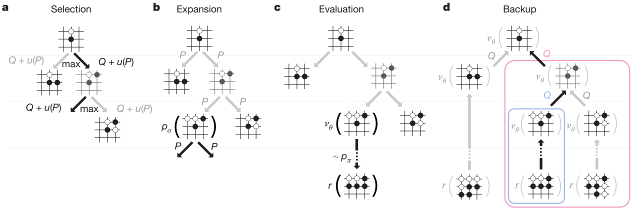


- notes, it's not the same as *back-propagation* in Neural Network training, I prefer to use "backup"
- "backup" simply changes probability of **a tree branch** involving the recently expanded leaf node, after the "simulation"/rollout
- it's useful in situations where its impossible to navigate to every states in the game
- after building the **partial tree** to some degree, one takes a move from $s \rightarrow s'$ using the **new** tree value
- then, discard everything except the sub-tree where s' is the new root

Traditional MCTS vs AlphaGo

- *D Silver et. al.,2016, Nature, Mastering the game of Go with deep neural networks and tree search*





- steps are almost identical, but many interesting innovations to reduce MCTS's search space
- P is a policy network, V is value network using neural networks
- in **selection**, it selects and action a by $\arg\max_a \left(\underbrace{Q(s,a)}_{\text{exploitation}} + \underbrace{U(s,a)}_{\text{exploration}} \right)$

$$\text{where} \quad U(s,a) = \underbrace{c_{puct}}_{\text{degree of exploration}} \quad \underbrace{p(s,a)}_{\text{prior prob: } p_\sigma(s)} \quad \frac{\sqrt{\sum_b N_r(s,b)}\ \sqrt{\text{sum of visit to siblings of } a}}{1 + N_r(s,a) \text{visits to node } a}$$

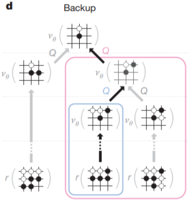
more about AlphaGo

- it uses p_σ for **expansion**, which is trained by Supervised Learning of human players
- board positions are treated like binary images stacks (black and white) to train Policy (P_{θ_σ}) networks and Value (V_θ) networks, and roll-out policy p_π
- at the end of simulation, leaf node is evaluated in two ways:
  1. using value network $v_\theta(.)$
  2. run a rollout to end with fast rollout policy p_π, then compute *the winner* with function r

     fast rollout policy p_π is trained using **linear-softmax**, i.e., no neural networks!

therefore, during backup, $Q(s, a)$ must come from both v_θ and r

for every visited edge (s, a), update using:



$$N_r(s, a) \leftarrow N_r(s, a) + 1$$

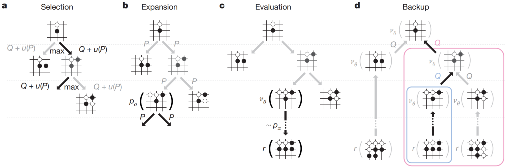
$$W_r(s, a) \leftarrow W_r(s, a) + r(s_T) \text{ roll-out win}$$

$$N_v(s, a) \leftarrow N_v(s, a) + 1$$

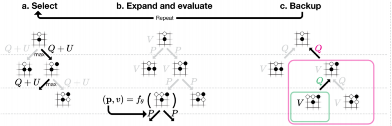
$$W_v(s, a) \leftarrow W)v(s, a) + v_\theta(s') \text{ “simulated” win}$$

$$Q(s, a) = (1 - \lambda)\frac{W_v(s, a)}{N_v(s, a)} + \lambda\frac{W_r(s, a)}{N_r(s, a)}$$

- **AlphaGo**



- **AlphaGo Zero** combines **expansion** (need $P_\theta(s)$) and **evaluation** (estimate $V_\theta(s)$) together

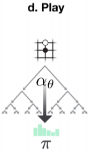


- In **AlphaGo**, evaluation can be done by play-till-end using fast roll-out policy
- in **AlphaGo Zero**, When new node is encountered, instead of performing a rollout, value of the new node is obtained from the neural network itself by evaluation $v \sim f_\theta(s)$
- for this reason, in **AlphaGo Zero**, $Q(s, a)$ is coming from only $V(s)$:

$$Q(s,a) = \frac{1}{N(s,a)} \sum_{s' | s,a \rightarrow s'} V(s') \qquad \text{mean of value function of all visited children states}$$

D Silver, et., al, 2017, Mastering the game of go without human knowledge, Nature

- **selection** step uses $\arg\max(Q(s,a) + U(s,a))$, same as **AlphaGo**
  $U(s,a) = c_{puct} \cdot P(s,a) \cdot \frac{\sqrt{\Sigma_b N(s,b)}}{1+N(s,a)}$
- **expansion** movements use $p_\theta(s)$ with θ updated
- after partial tree has completed, it moves using the partial tree



$$\pi(s) = \frac{N(s,\cdot)^{1/\tau}}{\sum_b (N(s,b)^{1/\tau})}$$

smooth the probabilities, since $N(s,\cdot) \geq 1$

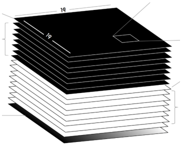
at the end of each game of self-play, neural network is provided training examples of the form, $(s_t, \vec{\pi}_t, z_t)$

- $\vec{\pi}_t$ is an estimate of the policy from state s_t
- $z_t \in \{-1, 1\}$ is final outcome of the game from the perspective of the player at s_t
- neural network is trained to minimise the following loss function (excluding regularisation terms):

$$l = \sum_t \underbrace{(v_\theta(s_t) - z_t)^2}_{\text{square loss}} \underbrace{-\vec{\pi}_t \cdot \log(\vec{p}_\theta(s_t))}_{\text{cross entropy loss}}$$

- over time, network will learn policy would give a good estimate of what the best action is from a given state

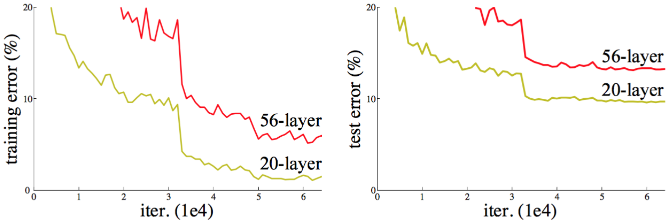
- game state s_t is $19 \times 19 \times 17$ image input stack



- include all 7 previous positions

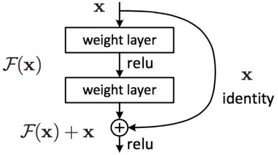
- the simulation loop (**selection**, **expansion + evaluation**, **backup**), runs for 1600 times
- training is performed on mini-batch of 2048 positions from last 500,000 self-played games

Other **AlphaGo Zero** Armory: Deep Residual Learning (1)

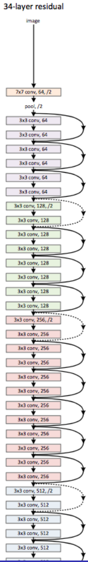


- when network depth increasing, accuracy gets saturated and then degrades rapidly.
- authors claim such degradation is **not** caused by overfitting

He., et al, (2016), Deep Residual Learning for Image Recognition, CVPR, pp. 770-778



- instead of few stacked layers directly fit a desired underlying mapping $H(x)$
- explicitly let these layers fit a residual mapping $F(x)$: $H(x) = F(x) + x$
- **hypothesize**: easier to optimize residual mapping than optimize original unreferenced mapping.
- if an identity mapping were optimal, it would be easier to push the residual to zero than to fit an identity mapping by a stack of nonlinear layers.



Our research question

- can we apply **AlphaGo Zero** methodologies to RTS game?
- RTS game state space is bigger than Go

  game state abstraction may help

- RTS game has a lot more units to move instead of a single Go piece, i.e,**larger action space**

  unit grouping learning may help

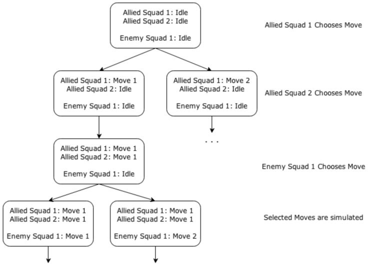
- in RTS games, some unit's move may only depends/follows a particular unit *instead* of all other units

  unit dependence learning may help

all these efforts will cut down the complexities of MCTS in RTS

Some exiting work on **game state abstraction**

- *D Soemers, 2014, "Tactical planning using MCTS in the game of StarCraft"*

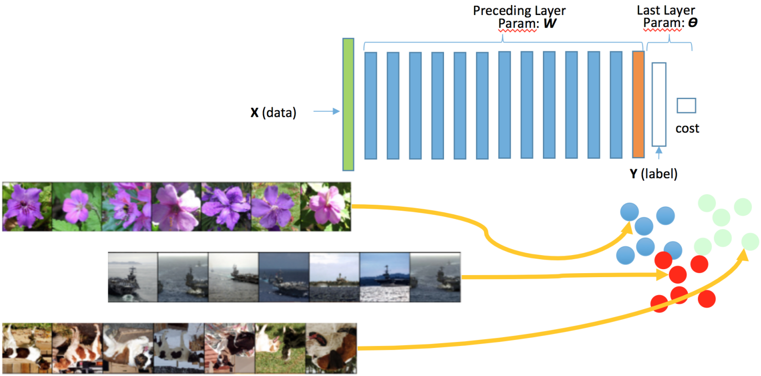


question 1: hand-crafted abstraction strategy may not be the best, can it be learned also?

- **rationale**: learn to group "context-similar" game states together
- **labels**: may need to use manually classify states in the beginning, or
- **labels**: may also used in combination of other clever tricks, such as **next action groups**

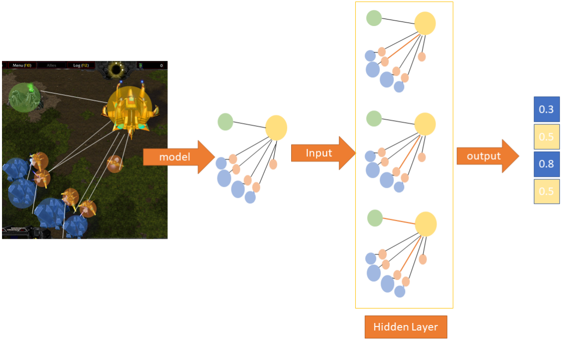
Research Idea 1: Learning **game state abstraction** (1)

- consider a normal Convolution Neural Networks:



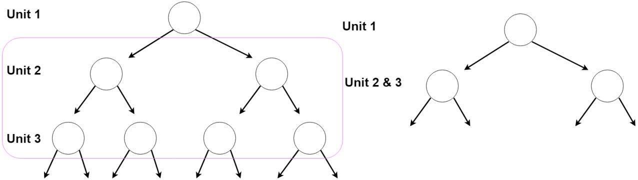
- the layers **preceding networks of Softmax** plays the role of the **feature embedding**, which aims to transform data of the same label to be closer together in this new representation.

- in RTS game, we potentially can apply Graph CNNs for this task
- there is many Graph CNN works around, for example (*Defferrard et al., NIPS 2016; Kipf & Welling, ICLR 2017*)
- specifically, in our work, we plan to run a supervised learning, to "categorize" semantically similar games states together by first represent the game states using Graph:



- intra-class variance may be huge! a lot of research and innovation needed

Research Idea 2: Learning **unit grouping**



- different to Go, in RTS, multiple units need to be moved
- increase the depth (or width) of the search tree
- intuitively, we may group units together, to reduce search time
- grouping heuristics can be used
- **an idea**, can we train a neural network for grouping $g_\theta(s)$? even more so, can we combine this with some prior partition model?

Research Idea 2: Learning **unit grouping** example starcraft

Very exploratory idea: Neural network with Context-driven Partition labels

- **problem** when there are N "marines" in starcraft, one may potentially assign them to $k \in \{1, \ldots, N\}$ groups; what is probability of a partition having $\{n_1, n_2, \ldots, n_K\}$ "marines"
- let $K \equiv n(\Pi)$, number of groupings for a particular partition:
- one may have the following two"partitions":

$$
\begin{aligned}
\{3, 1, 2, 3, 2, 3, 2, 3\} &\implies \{n_1 = 1, n_2 = 3, n_3 = 4\} \\
\{3, 3, 3, 2, 1, 1, 1, 1\} &\implies \{n_1 = 4, n_2 = 1, n_3 = 3\}
\end{aligned}
$$

  they are **equivalent**:
- **in words**: for all partitions of 8 "marines" having:

  "4 *"marines" in one of group,* 3 *in one of group and* 1 *"marine" in one group*"

  then these partitions should be treated **equivalently**, i.e., it does **not** matter which particular group has 4 "marines"
- obviously, different **process** in generate "marines" grouping result in different probabilities of partitions

For example: Partition Model using CRP

- in scenarios where we'd like to assign "marines" to uneven groupings, so more elements in a cluster, the more they will joint:
- we may use conditional density $\Pr(z_i = m | z_{i-1}, \dots z_1) \equiv \Pr(z_i = m | \mathbf{z}_{-i}, \alpha)$

$$\Pr(z_i = m | \mathbf{z}_{-i}, \alpha) \propto \begin{cases} \frac{n_{m,-i}}{N+\alpha-1} & \text{for existing cluster } m \\ \frac{\alpha}{N+\alpha-1} & \text{for new cluster} \end{cases}$$

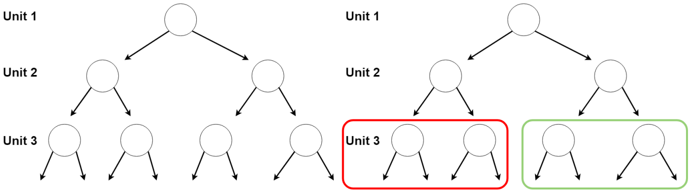
- using DP, the probability on a partition is:

$$\pi(\Pi_N) = \frac{\alpha^k \prod_{l=1}^{k} \Gamma(n_l)}{\prod_{i=1}^{n} (\alpha + i - 1)}$$

- $k \equiv n(\Pi)$: number of clusters
- n_l: number of "marines" in a group l

Research Idea 3: Learning **unit dependence**

- in RTS, some unit may only depending on one particular unit
- think about soldiers moving in a line formation, he follows the person in front only
- intuitively, we may learn such dependencies to reduce search time:
- , say $\Pr(u_3 | u_2, u_1) = \Pr(u_3 | u_2)$, making red and green part of the tree identical:



- of course, we needed to propose dependence at every state s: