Generalised reinforcement learning with self play

INF552 Project

Kaladhar
Abhinav
Pallavi

INTRODUCTION

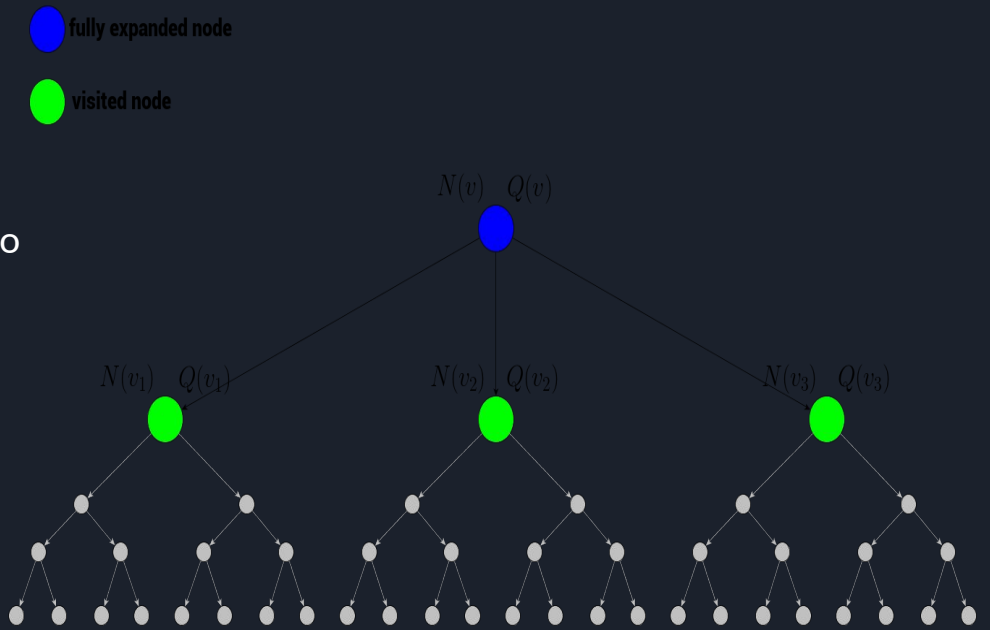
- Reinforcement learning- software agents ought to take actions in an environment to maximize some notion of reward.
- Alpha zero algorithm
- Neural networks
- Self play

ALPHAGO COMPONENTS

- MCTS
- Policy network
- Value Network
- Reinforcement learning used for training the network via self-plays

MCTS

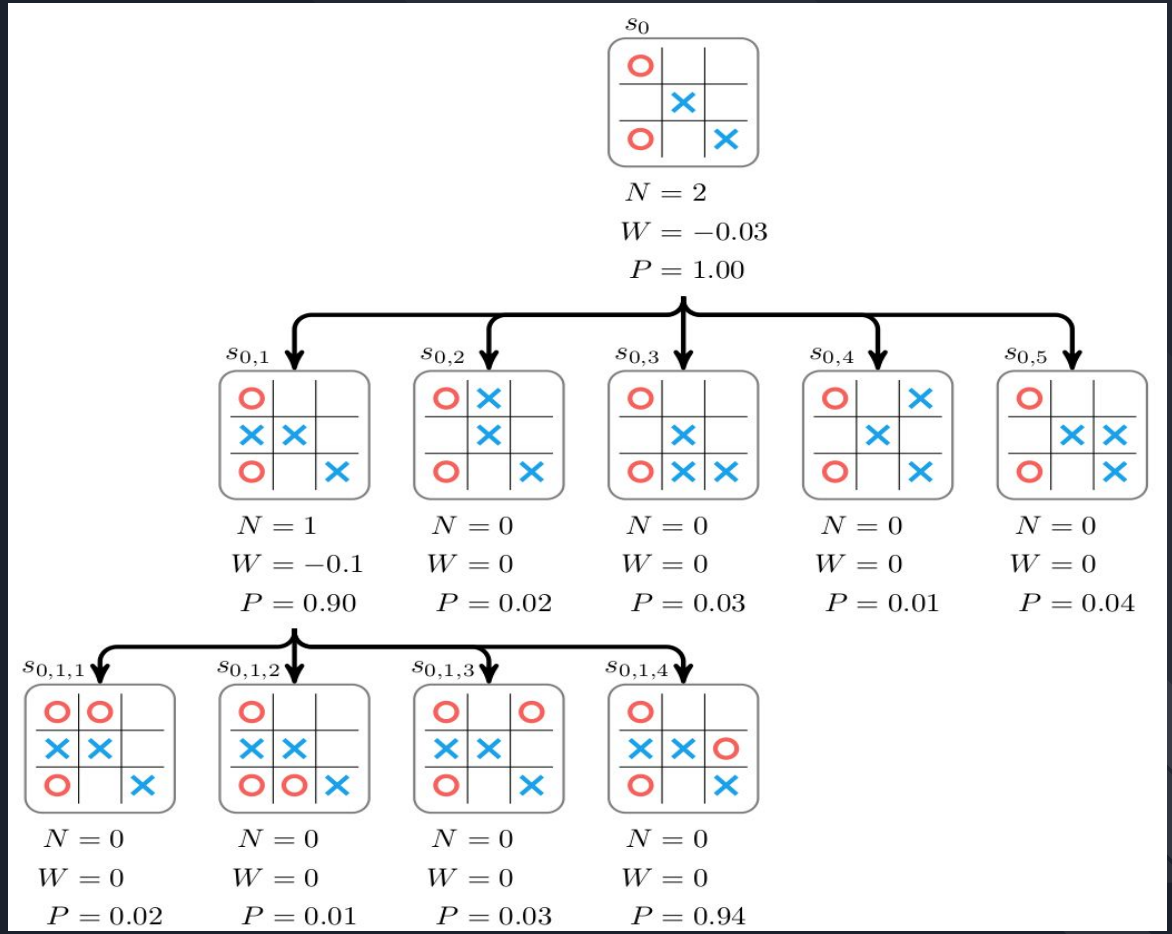
- Simulates the game many times
- Computes the most promising move
- Moves are chosen with respect to policy function
- Result is back propagated to the current game tree node.
- Q(v) total simulation reward
- N(v) total number of visits



Upper confidence bound

- Function that chooses next node among visited nodes to traverse through-
- Exploration component
- Exploitation component

MCTS


s_0
$N = 2$
$W = -0.03$
$P = 1.00$
$s_{0,1}$
$N = 1$
$W = -0.1$
$P = 0.90$
$s_{0,2}$
$N = 0$
$W = 0$
$P = 0.02$
$s_{0,3}$
$N = 0$
$W = 0$
$P = 0.03$
$s_{0,4}$
$N = 0$
$W = 0$
$P = 0.01$
$s_{0,5}$
$N = 0$
$W = 0$
$P = 0.04$
$s_{0,1,1}$
$N = 0$
$W = 0$
$P = 0.02$
$s_{0,1,2}$
$N = 0$
$W = 0$
$P = 0.01$
$s_{0,1,3}$
$N = 0$
$W = 0$
$P = 0.03$
$s_{0,1,4}$
$N = 0$
$W = 0$
$P = 0.94$

Policy and Value Network

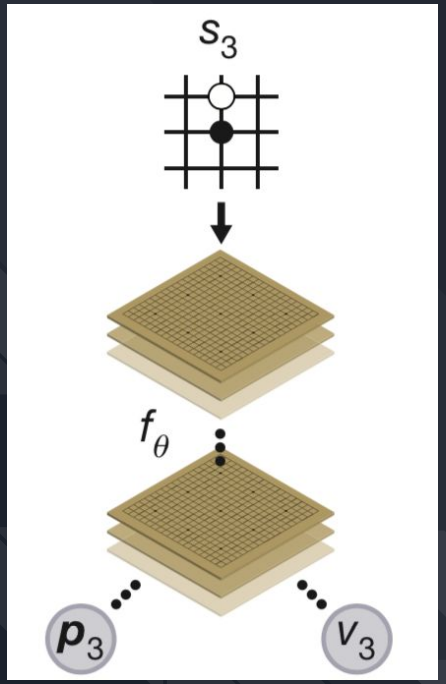
Both Policy and Value are determined by the same network.

The heads (output layer) are different for the Network. The value function is a continuous-valued function between [-1,1]. Also, Policy network provides the Probability for the next best move.

This makes the network robust as the loss function is applied to the ‘common network’ and the respective heads.

The Loss function is for the network is given as follows :

SGD is used to reduce the loss for each batch.



$$l = \sum_t (v_\theta(s_t) - z_t)^2 - \vec{\pi}_t \cdot \log(\vec{p}_\theta(s_t))$$

$$U_i = \frac{W_i}{N_i} + cP_i\sqrt{\frac{\ln N_p}{1+N_i}}$$

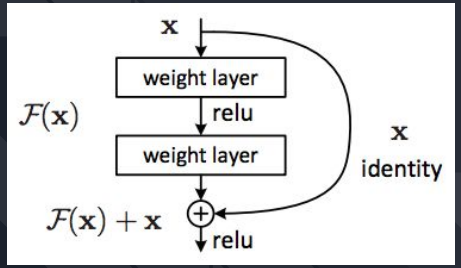
Neural Network Architecture

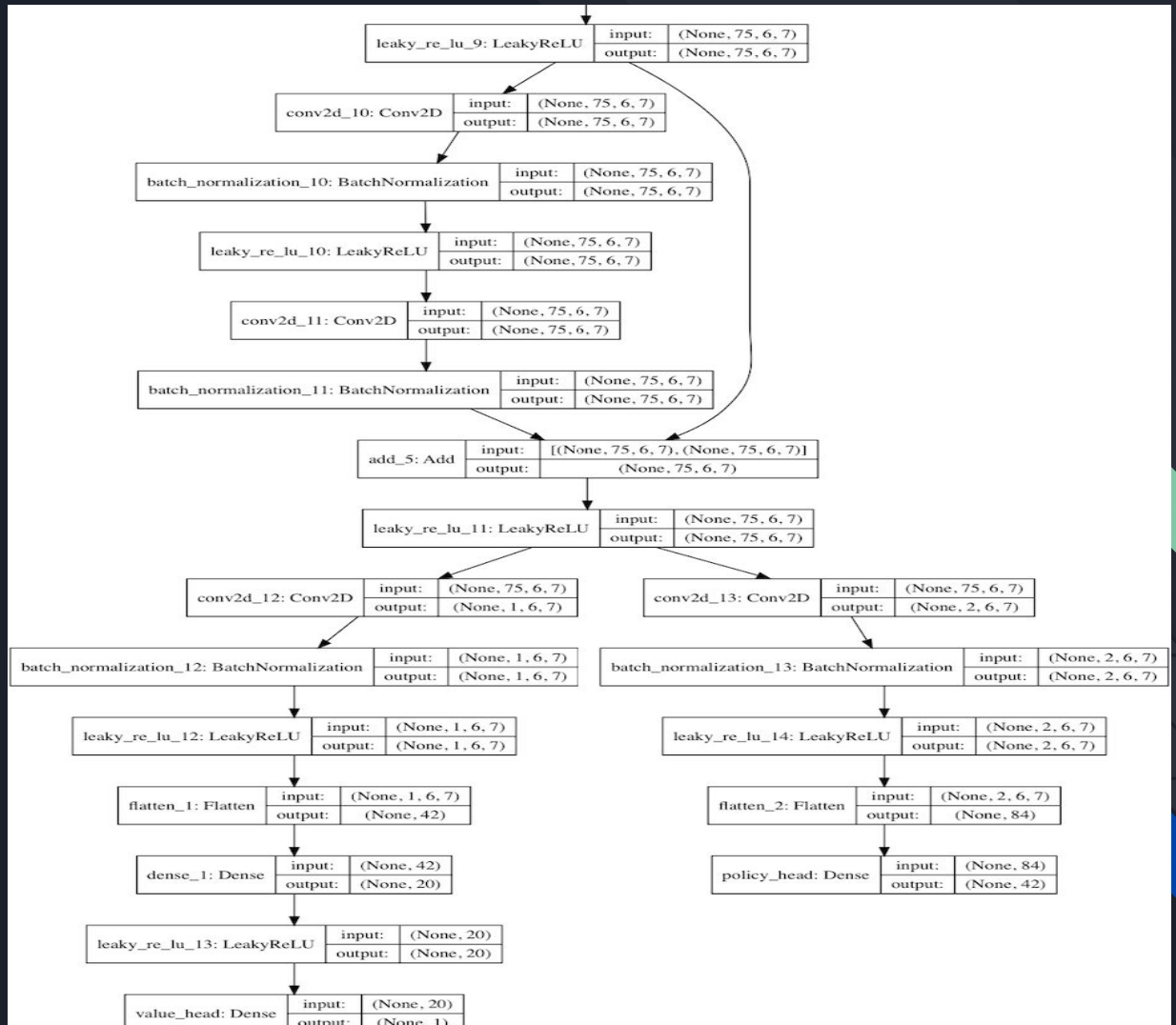
Residual Networks are used to achieve more depth in terms of layers.

The utilization of RN's allowed for 20 RN's.

Vanishing Gradients problem is reduced and training error does not increase as the input takes a shortcut with the help of an identity mapping.

The actual residual block consists of two CNN's instead of a Dense Layer.




leaky_re_lu_9: LeakyReLU | input: (None, 75, 6, 7) | output: (None, 75, 6, 7)
conv2d_10: Conv2D | input: (None, 75, 6, 7) | output: (None, 75, 6, 7)
batch_normalization_10: BatchNormalization | input: (None, 75, 6, 7) | output: (None, 75, 6, 7)
leaky_re_lu_10: LeakyReLU | input: (None, 75, 6, 7) | output: (None, 75, 6, 7)
conv2d_11: Conv2D | input: (None, 75, 6, 7) | output: (None, 75, 6, 7)
batch_normalization_11: BatchNormalization | input: (None, 75, 6, 7) | output: (None, 75, 6, 7)
add_5: Add | input: [(None, 75, 6, 7), (None, 75, 6, 7)] | output: (None, 75, 6, 7)
leaky_re_lu_11: LeakyReLU | input: (None, 75, 6, 7) | output: (None, 75, 6, 7)
conv2d_12: Conv2D | input: (None, 75, 6, 7) | output: (None, 1, 6, 7)
conv2d_13: Conv2D | input: (None, 75, 6, 7) | output: (None, 2, 6, 7)
batch_normalization_12: BatchNormalization | input: (None, 1, 6, 7) | output: (None, 1, 6, 7)
batch_normalization_13: BatchNormalization | input: (None, 2, 6, 7) | output: (None, 2, 6, 7)
leaky_re_lu_12: LeakyReLU | input: (None, 1, 6, 7) | output: (None, 1, 6, 7)
leaky_re_lu_14: LeakyReLU | input: (None, 2, 6, 7) | output: (None, 2, 6, 7)
flatten_1: Flatten | input: (None, 1, 6, 7) | output: (None, 42)
flatten_2: Flatten | input: (None, 2, 6, 7) | output: (None, 84)
dense_1: Dense | input: (None, 42) | output: (None, 20)
policy_head: Dense | input: (None, 84) | output: (None, 42)
leaky_re_lu_13: LeakyReLU | input: (None, 20) | output: (None, 20)
value_head: Dense | input: (None, 20) | output: (None, 1)

Policy Iteration through Self-play

In each simulation of our algorithm, the best_model play a number of episodes (10 in the current experiments) of self-play using MCTS. The resulting training examples is of the form (s(t), ~π (t), z(t)).

Then, the current best is updated neural network using the new training examples, to get a new neural network. We then play the old and new networks against each other for a number of games (40 in our experiments). If the new network wins more than a set threshold number of times, the network is updated and continue with the next iteration, resetting the MCTS tree. Else, we continue with the old network and the old MCTS tree, and conduct another iteration to augment our training examples further.

Algorithm 2 Policy Iteration through Self-Play

```
procedure PolicyIterationSP
    θ ←initNN()
    trainExamples ← []
    for i in [1, ..., numIters] do
        for e in [1, ..., numEpisodes] do
            ex ← executeEpisode(nn)
            trainExamples.append(ex)
        θ_new ← trainNN(trainExamples)
        if θ_new beats θ ≥ thresh then
            θ ← θ_new
    return θ
```

Algorithm 3 Execute Episode

```
procedure ExecuteEpisode(θ)
    examples ← []
    s ← gameStartState()
    while True do
        for i in [1, ..., numSims] do
            MCTS(s, θ)
        examples.add((s, π_s, _))
        a* ~ π_s
        s ← gameNextState(s, a*)
        if gameEnded(s) then
            //fill _ in examples with reward
            examples ←assignRewards(examples)
            return examples
```

Challenges faced..

DEMO

0_0	0_1	0_2	1_3	0_4	1_5	0_6	1_7
1_8	0_9	0_{10}	0_{11}	1_{12}	0_{13}	1_{14}	0_{15}
0_{16}	0_{17}	0_{18}	0_{19}	0_{20}	1_{21}	0_{22}	1_{23}
1_{24}	0_{25}	0_{26}	0_{27}	1_{28}	0_{29}	0_{30}	0_{31}
0_{32}	-1_{33}	0_{34}	1_{35}	0_{36}	0_{37}	0_{38}	0_{39}
0_{40}	0_{41}	0_{42}	0_{43}	0_{44}	0_{45}	-1_{46}	0_{47}
0_{48}	-1_{49}	0_{50}	-1_{51}	0_{52}	-1_{53}	0_{54}	-1_{55}
-1_{56}	0_{57}	-1_{58}	0_{59}	-1_{60}	0_{61}	-1_{62}	0_{63}