MANIPAL INSTITUTE OF TECHNOLOGY
MANIPAL
(A constituent unit of MAHE, Manipal)

SEMINAR REPORT ON

APPLICATION OF DEEP LEARNING IN GAME BALANCE TESTING

SUBMITTED TO

Department of Computer Science & Engineering

by

Gurunandan
170905018
Roll Number: 04
Semester VII Section D

Name & Signature of Evaluator 1

Name & Signature of Evaluator 2

Table of Contents

(Aug 2019)

1. INTRODUCTION

1.1 Game Balancing

Games have shaped a good portion of pop culture especially that of 21st century. Gamming has turned into a multi-billion Dollar industry with millions of fans and thousands of developers. However, game development has been one the most challenging industries for developers given the deadlines associated with releases, consumer demands and elaborate testing required. It is easy for developers to lose focus while simultaneously trying to improve graphics, add functionality and balance difficulty or strategies of the game.

One can find all kinds of games, we have action, puzzle, story based, strategy to name some. We have single player and multiplayer games, competitive and co-operative and so on. Today we will focus on multiplayer and competitive games. Certain multiplayer games have a gameplay where the player can take up certain strategies in the action space. A developer building such a game would not only have to work on the usual graphics and mechanics of the game but also on balancing the strategies mentioned above to make up for an entertaining game where all of the strategies should have a use-case and none of the strategies should dominate over other strategies so that the game remains diverse and happening.

One of the most time-consuming and difficult processes in designing phase is game balancing. A developer especially one that of an indie game would not the resource to play test the game on a large enough much needed population. This could generally result in game releases which tend to have overpowered strategies which could be spammed and result in not so interesting gameplay. On the other hand, some strategies might be relatively un-advantageous result in underutilization. Generally, such problems might cause early players to lose interest and quit the game essentially preventing the game from gaining momentum.

In some cases, in a well-established game when adding a new strategy, say a weapon or a character or some abilities, this new strategy maybe by its own or by combining with other strategy might essentially prove overpowered thereby causing havoc in an otherwise balanced game. Players might lose their progress because of such incidents.

1.2 Machine Learning

Past couple of decades have seen major developments in the field of Artificial Intelligence. Most of which is thanks to rise of soft computing paradigms. Soft computing gives low cost approximate solutions to otherwise difficult problems. Machine Learning as in its name is about learning solutions rather than programming logic for the solution.

Machine Learning is broadly classified into 3 categories:

1. **Supervised Learning**

- We have training data which comprises of input data paired with its corresponding output. This data is used to approximate the function we are trying to implement.
- Our goal is to generalize the model from training data so that we have enough precession and accuracy for fresh data.

2. **Unsupervised Learning**

- As opposed to Supervised learning, we do not need the output information while training, instead it is used derive correlation or patterns from a given data set to perform various actions like grouping them and so on.
- Used for Clustering, in recommendation systems or dimensionality decreasing of a dataset.

3. **Reinforcement Learning**

- Model is made to act on its own, reward function is built to reward the model when it acts the way we want to, model is trained such that reward is maximized.
- Reinforcement Learning is core concept for many important implementations of machine learning such as self-driving cars, bots playing games and so on.

.

1.2.1 Deep Learning

Deep Learning is an Artificial Intelligence function that imitates the working of human brain in processing data and creating patterns for use in decision making. Human brains have fascinating structures and are capable of incredible achievements, to complete or help complete humanly tasks a humanly way of computing was necessary, Deep Learning tries to do just that by implementing mathematical models that neurons and networks in our brain works on.

Deep learning is a particular kind of machine learning that achieves great power and flexibility by learning to represent the world as a nested hierarchy of concepts, with each concept defined in relation to simpler concepts, and more abstract representations computed in terms of less abstract ones.

Deep Learning revolves in and around the idea of a Neural Network. A Neural Network is a mathematical function that could be tuned using its parameters such as weights and biases. Deep Learning and by extension Neural Networks have been used in all kinds of Supervised Learning and Reinforcement learning solutions. Mathematics and inner workings of Neural Networks are discussed further down.

As processing and power has increased exponentially in last 20 years or so, Deep Learning and Machine Learning have come to forefront. Today Deep Learning has become integral part of many workflows and opened up a whole new industry. Learning aspect of deep learning could help people simplify their work or might get rid of repetitive works. We can see this action in various industries, take automotive for example, Self-driving cars although haven’t replaced all driving however is already making driving far easier and less attention demanding. Greater part of the journey in a self-driving car is automatic. Computer vision for example has reached new heights thanks to Deep Learning. Face recognition systems, Better image processing and others have already come down to consumer level all thanks to Deep Learning. World of personal assistants and natural language processing has seen amazing progress, while at the same time applications of Deep learning in data science cannot be overlooked.

2. BACKGROUND THEORY

2.2 Neural Networks

A neural network comprises of several layers of neurons, first layer is called the input layer while the last layer is called the output layer, all other layers in between are hidden layers. To implement a neural network in programs we need to model it mathematically. For gaining mathematical intuition varying degrees of abstractions would come in handy.

Network as a whole could be thought of as a function which for a given input throws out an output on the other side. Translating this in terms of Linear algebra, neural network is a transformation which transforms a given in vector in m dimensional input space to n dimensional output space. Hence each of the hidden layer could be thought of as a linear transformation. These linear transformations are characterized by the "parameters" of the layers called weights and biases. To learn a function i.e. to learn the necessary transformation parameters to map input space to output space we use various algorithms depending upon the problem.

Inputs which we stated as being n dimensional vector might be more generally be considered a tensor. Each layer in the network is also a tensor and hence transformation is just a tensor product maybe coupled with addition with a bias factor. This would mean process of getting output from input would just be series of linear transformations (tensor products and addition).

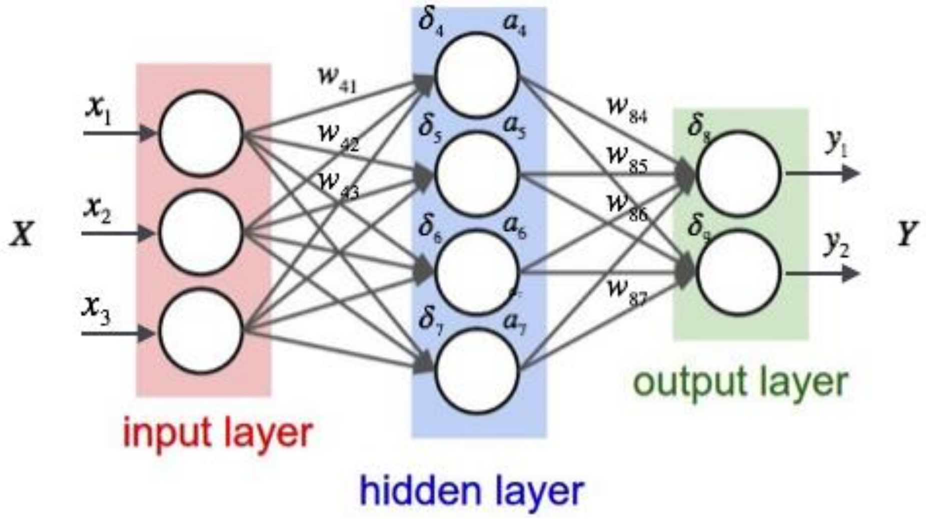


Fig. 1.1 A Neural network comprising of 3 dimensional input and 2 dimensional output with 1 hidden layer.

Why multiple layer? A linear transformation could only make up linear boundaries hence could solve linearly separable problems. Think of trying to draw a line to divide space for XOR operation with each basis indicating a bit. It is impossible to draw one line to separate true cases from false cases that means the problem itself is linearly inseparable. To solve such problems all we have to do is use multiple layers since each layer would transform the input space into preferably a higher dimensional space and at last to output space till when the problem is linearly separable. Using of multiple layers is exactly gives Deep Learning its name.

For an input vector X, in a layer the resulting vector A will be,

$$A = WX + b$$

Where, W is the weight tensor of the layer and b is the bias vector.

Note how depending on dimensionality of weight vector X would be transformed into higher or lower dimensions. Here, WX indicates tensor product (generally a matrix multiplication) and addition is a vector addition.

Often an Activation function is used as well, activation function may act as a normalizing function, a step function, as a tool to get rid of negative signs or as threshold function. Some commonly used activation functions are Sigmoid, RELU, Tanh and Softmax.

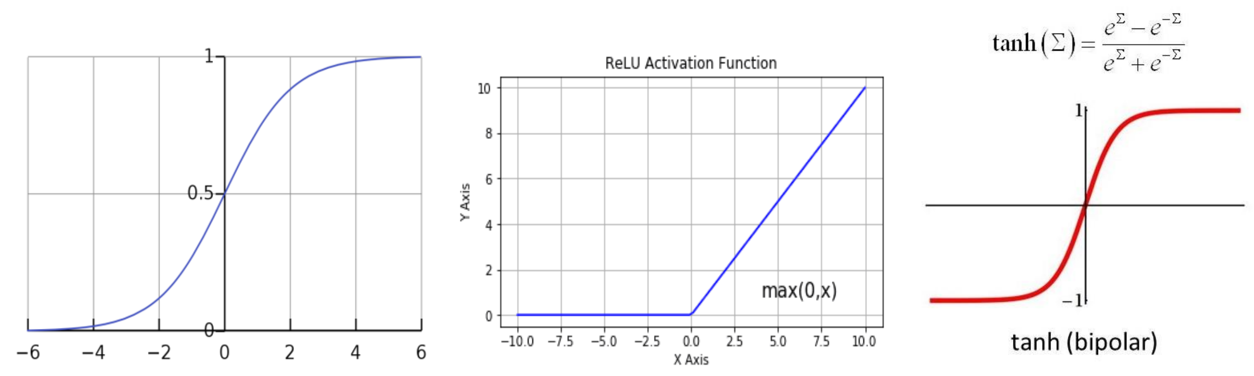


Fig. 1.2 Some Activation functions: sigmoid, RELU and Tanh in the same order

Sigmoid normalizes each value in the tensor between 0 and 1, RELU zeros negative values. Tanh normalizes value between -1 to 1. Softmax outputs index containing max value.

2.3 Learning Algorithms

A Neural Network is only a general function that could be tuned with the help of its parameters. But how to tune the parameters? How to make the network give out desired output rather than random outputs? We discuss some techniques to train the neural network in this section.

2.3.1 Supervised Learning Algorithm

- We first build a neural network with m inputs and n output by looking at the problem. We then randomly initialize the weights and biases of the network. We have also prepared a dataset containing input data and its corresponding output. We preprocess this data to clean it and make it usable. We try to remove unnecessary information out of this dataset.

- For a given input we pass it through the network and get the output. We call this forward pass. Here our input passes through the many layers and activation functions of our network

- We define something called Loss metric. Loss metric is metric corresponding to how wrong our prediction was. So the calculation of Loss involves predicted tensor and actual tensor (obtained from dataset).

- Next step is to backpropagate the loss calculated above. This involves calculating derivatives to find the direction and the amount to adjust the parameter. Hyper-parameters such as learning rate has to defined earlier. Intuition is to tune the parameters little bit such that the output would have been closer to the actual output. Hence loss is minimized over time.

- We then loop back to step 2 with next value in the dataset.

- After enough iterations and given a good and large enough labelled dataset we will end up with a sufficiently accurate model.

2.3.2 Reinforcement Learning

Supervised Learning has proved to be very effective and useful. However, in many cases, we won't have a dataset mapping input to outputs. Reinforcement Learning aims to train a network by allowing it to make decisions and then correcting it. We define a reward function which gives out a reward for any event that occurs given the state of the environment which is by extension a function of the agents actions. Rewards are negative when we want to punish the network, for example if it loses health or loses a game. Our goal is to maximize the net reward over time.

Many reinforcement learning algorithms exists. Couple of more popular ones are:

1. **Evolutionary Algorithm:**

   Here we start with a population or randomly acting agents, for each iteration we let them interact with the environment. We pick agents whose actions game them the max reward. We send these agents into next iteration, we also mutate copies of these agents, breed some of the agents and send them to next iteration. We also add some random agents to next iteration as well.

   Given enough iterations highly performing agents are left out giving us a fairly good solution. This algorithm doesn't need to calculate any gradients and is effectively inspired by and based on natural biological evolution.

2. **Deep Q-Learning (DQN)**:

   Classical Q-learning is about creating a Q-table, which is a table with the size actions X states where each entry is called a Q value. Higher Q value corresponds to better decision. Hence once the Q-table is built we can choose the actions to take depending on the state we are in by choose the action with the highest Q value. However, real world problems tend to have a massive state space as well as a fairly large action space. Hence creating a Q-table becomes impractical.

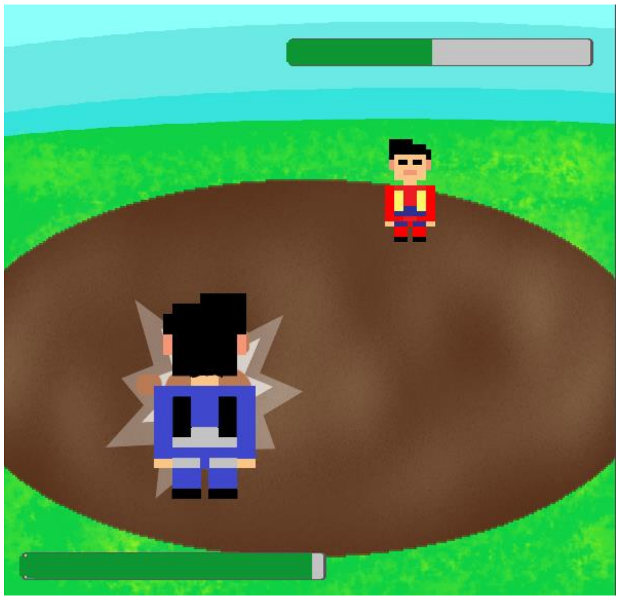
   Deep Q-learning aims at building a neural network that can predict the Q values by learning it over time. Many versions of Deep Q-Learning exist for various use cases and are widely used in reinforcement Learning.

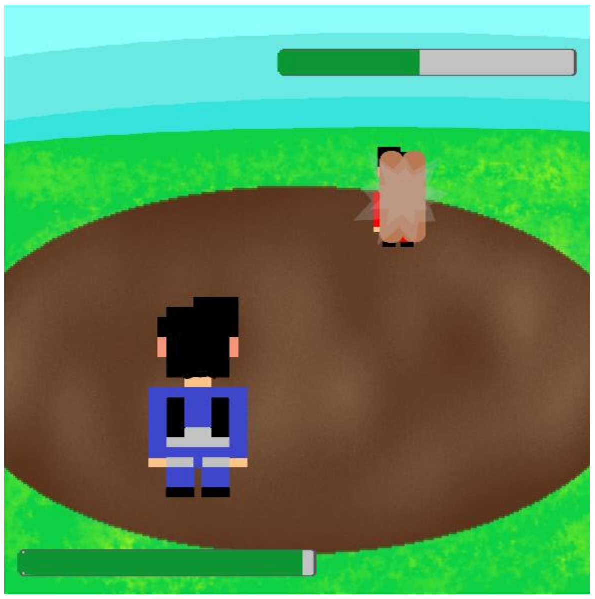
3. METHODOLOGY AND IMPLEMENTATION

We will try to build a game and train an agent to play the game optimally and then get the statistics related to strategy usage of the bot and try to figure out which strategies are overpowered and which ones are underpowered and then try to balance out the game so that every strategy has a use case and so that none of the strategies could be blindly spammed.

3.1 Building a Game

We use python pygame module to build a fairly simple strategic multiplayer game heavily inspired by the classic game Pokémon. At a given time, player has multiple strategies to choose from, in our case multiple moves to choose from where each move changes the game state in unique ways. Moves will also have some element of randomness built into them so that the game isn't completely deterministic.



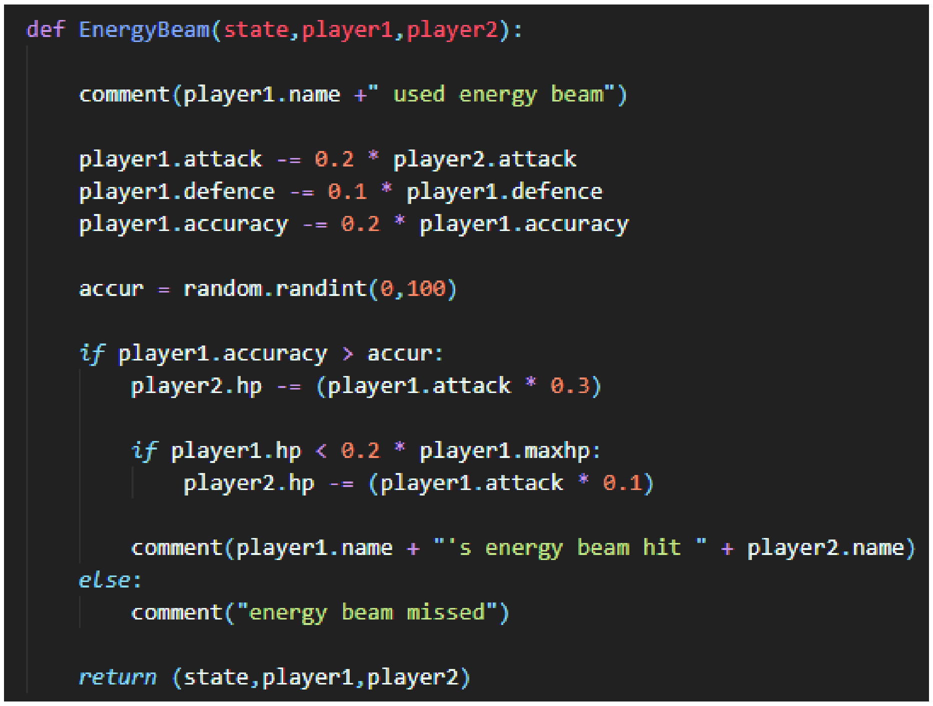


Mechanics of the game:

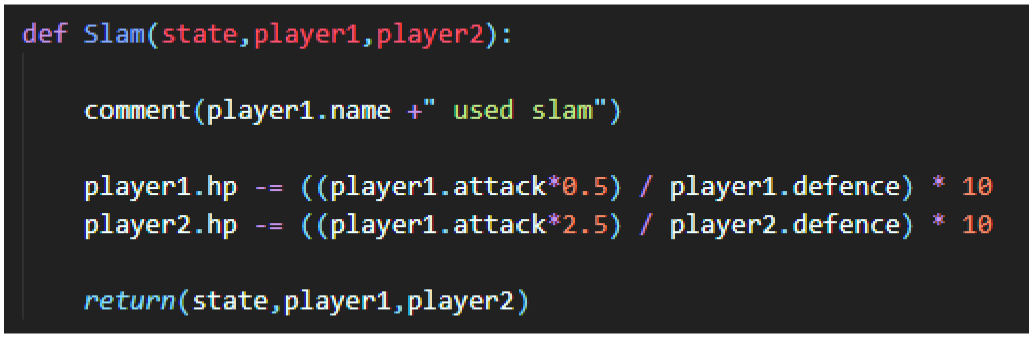
- It's a 2 player game where each player has a set of moves to choose from (set of moves would form the action space for an agent).
- Each player has stats corresponding to oneself initialized. Stats include player's max health, current health, attack stat, defense stat, agility, accuracy. New stats could be added anytime further along the development process. Both player's stats together define the state of the game at a given point in time.

- At each step, both players get a chance to choose a move, depending on one's agility stats player gets to execute his move first, other player plays his move after that.

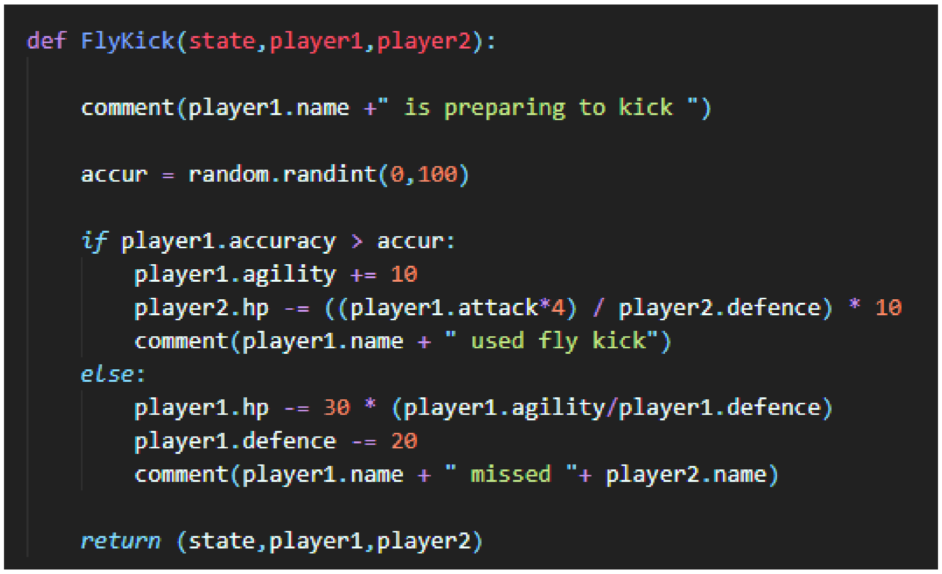
- Player to reach 0 hp first loses the game.



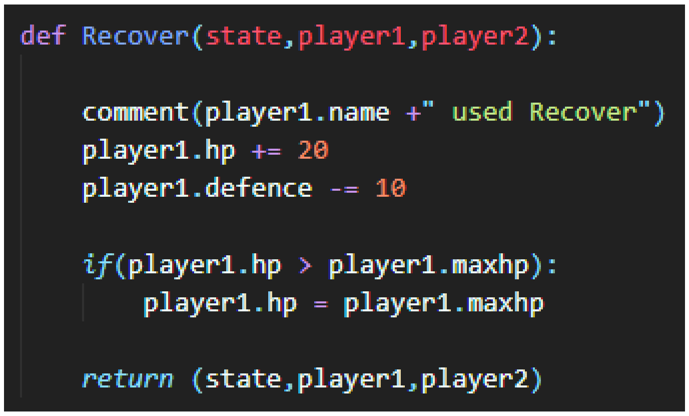
```
def EnergyBeam(state,player1,player2):

    comment(player1.name +" used energy beam")

    player1.attack -= 0.2 * player2.attack
    player1.defence -= 0.1 * player1.defence
    player1.accuracy -= 0.2 * player1.accuracy

    accur = random.randint(0,100)

    if player1.accuracy > accur:
        player2.hp -= (player1.attack * 0.3)

        if player1.hp < 0.2 * player1.maxhp:
            player2.hp -= (player1.attack * 0.1)

        comment(player1.name + "'s energy beam hit " + player2.name)
    else:
        comment("energy beam missed")

    return (state,player1,player2)
```



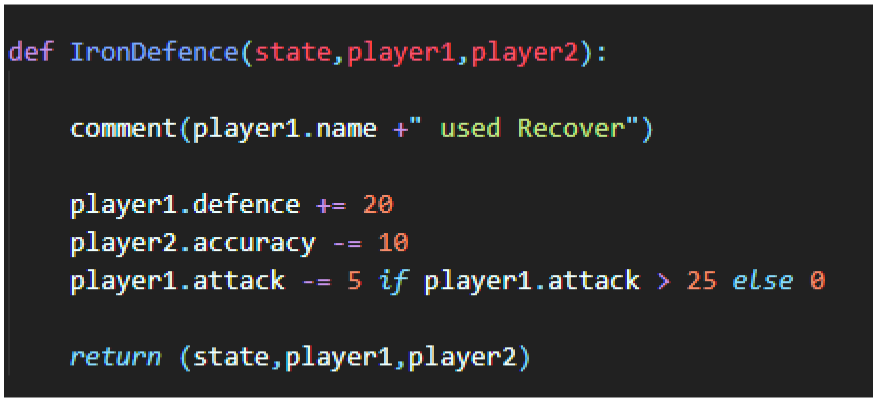
```
def Slam(state,player1,player2):

    comment(player1.name +" used slam")

    player1.hp -= ((player1.attack*0.5) / player1.defence) * 10
    player2.hp -= ((player1.attack*2.5) / player2.defence) * 10

    return(state,player1,player2)
```



```python
def FlyKick(state,player1,player2):

    comment(player1.name +" is preparing to kick ")

    accur = random.randint(0,100)

    if player1.accuracy > accur:
        player1.agility += 10
        player2.hp -= ((player1.attack*4) / player2.defence) * 10
        comment(player1.name + " used fly kick")
    else:
        player1.hp -= 30 * (player1.agility/player1.defence)
        player1.defence -= 20
        comment(player1.name + " missed "+ player2.name)

    return (state,player1,player2)
```



```python
def Recover(state,player1,player2):

    comment(player1.name +" used Recover")
    player1.hp += 20
    player1.defence -= 10

    if(player1.hp > player1.maxhp):
        player1.hp = player1.maxhp

    return (state,player1,player2)
```



```python
def IronDefence(state,player1,player2):

    comment(player1.name +" used Recover")

    player1.defence += 20
    player2.accuracy -= 10
    player1.attack -= 5 if player1.attack > 25 else 0

    return (state,player1,player2)
```

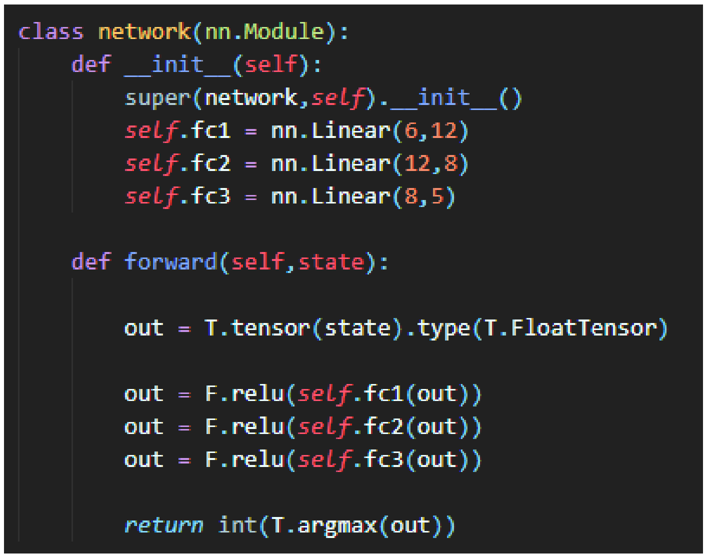
Although the rules of the game itself are simple, balancing of the moves becomes exponentially hard as the number of moves increases. Developer might have scenario for a strategy in mind but the strategy might be overpowered in other scenarios or when paired with other strategies. Testing the game might prove tiresome and time consuming, for a large enough game, expensive.

3.2 Training an Agent

1. Neural Network

Our Agent will have a neural network which would be fed states of the game and we would get back the action to take (move to use). State information to send into a neural network as input is choice of the programmer and would depend on game. Here, we will send only 6 values. An agent will get its hp (in percentage), opponents hp (also in percentage), its attack, defense, accuracy and agility at point in game. Hence the input dimensions will be 6. Output will be an integer representing the id of the move to use.

Our Network is comprised of 3 fully connected layers. Also we will use RELU activation function after each layer for each forward pass. We would take argmax of the output which would essentially give us the index of the maximum value in the output vector.



```
class network(nn.Module):
    def __init__(self):
        super(network,self).__init__()
        self.fc1 = nn.Linear(6,12)
        self.fc2 = nn.Linear(12,8)
        self.fc3 = nn.Linear(8,5)

    def forward(self,state):

        out = T.tensor(state).type(T.FloatTensor)

        out = F.relu(self.fc1(out))
        out = F.relu(self.fc2(out))
        out = F.relu(self.fc3(out))

        return int(T.argmax(out))
```

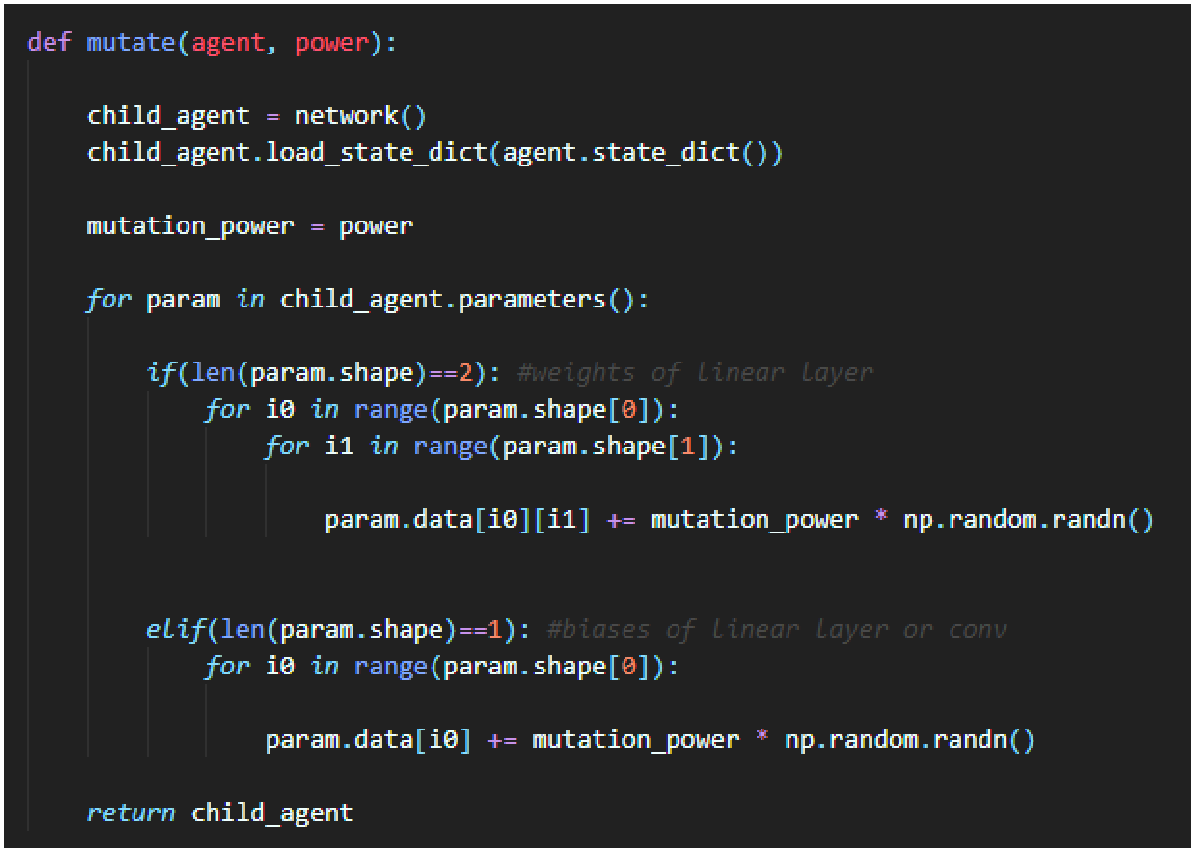
2. Training algorithm

Picking a training algorithm is one of the most important part of building a deep learning solution. Since our environment at hand is multi-agent, we cannot use the regular DQN algorithm, this is because for any agent in the game its environment is game mechanics + other agent. When one agent tries to learn its environment other agent is also learning which implies the environment for a given agent seems to be evolving (rather changing). All the old Q-values learnt may not be valid now and hence making it extremely difficult to learn.

There are some other solutions which could help agent learn this game, one of the easier options is the evolutionary algorithm mentioned before. Evolutionary algorithm is based on idea of maximizing reward function by creating a lot of networks, mutating and breeding them until minimum reward requirements are met. For every iteration, population gets a little better, and over time we have a population of well-trained networks that have learnt the function to gain optimal rewards.

Mutation

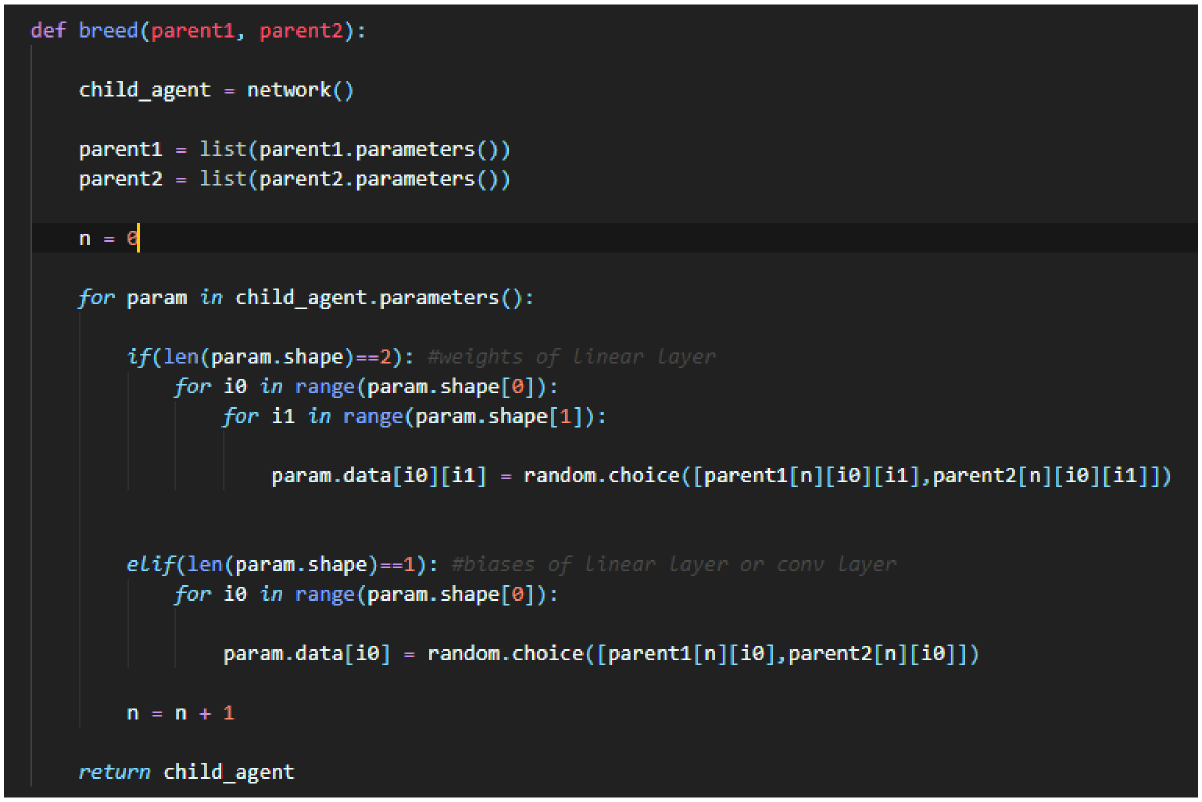
We can mutate a given network by taking its parameter and adding a random value which is multiplied by a parameter called mutation power. Higher value of mutation power would have greater degree of mutation.



```
def mutate(agent, power):

    child_agent = network()
    child_agent.load_state_dict(agent.state_dict())

    mutation_power = power

    for param in child_agent.parameters():

        if(len(param.shape)==2): #weights of linear layer
            for i0 in range(param.shape[0]):
                for i1 in range(param.shape[1]):

                    param.data[i0][i1] += mutation_power * np.random.randn()


        elif(len(param.shape)==1): #biases of linear layer or conv
            for i0 in range(param.shape[0]):

                param.data[i0] += mutation_power * np.random.randn()

    return child_agent
```

Breeding

While breeding 2 networks, we will loop through parameters of a random network while randomly setting them to be one among the parameter at same position of its parents. Breeding would give us a network that is in between the parents. We hope that the breeding gives us a network which has the best strategies of its parents.



```
def breed(parent1, parent2):

    child_agent = network()

    parent1 = list(parent1.parameters())
    parent2 = list(parent2.parameters())

    n = 0

    for param in child_agent.parameters():

        if(len(param.shape)==2): #weights of linear layer
            for i0 in range(param.shape[0]):
                for i1 in range(param.shape[1]):

                    param.data[i0][i1] = random.choice([parent1[n][i0][i1],parent2[n][i0][i1]])


        elif(len(param.shape)==1): #biases of linear layer or conv layer
            for i0 in range(param.shape[0]):

                param.data[i0] = random.choice([parent1[n][i0],parent2[n][i0]])

        n = n + 1

    return child_agent
```

3. Training method

Instead of defining a reward metric, we could build up a tournament style learning mechanism. In a training iteration each network would be pit another network until we get a winner. Selection criterion to go into next iteration would be to be in top 1/8th, we would also mutate some of the top performing networks to put them in next level, we would breed top performers as well to get networks with strategies of both the parent networks.

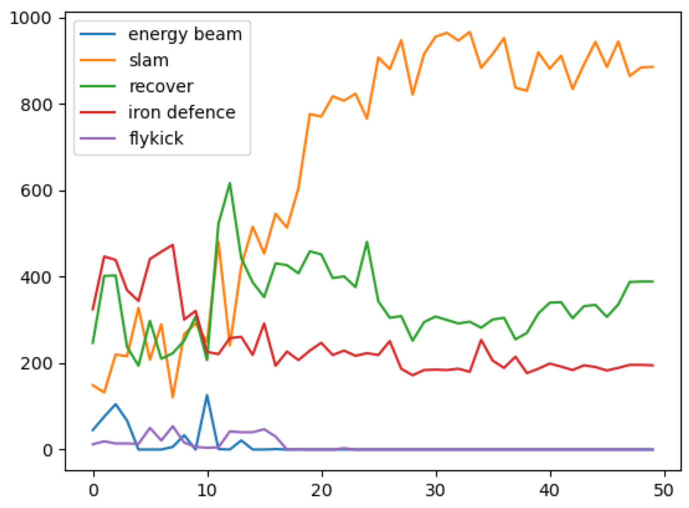
We define fight function which takes 2 agents and returns the winner of the fight. We also define fightHalf function which takes an array of networks, matches us first half of the network with the next half and returns the array containing the winners of the fight. We can now use these functions to set up our tournament style training loop.

Given enough training iteration, population would have developed the optimal strategy and we could get a lot of necessary data to comment and act on the state of the balance of the game.

4. RESULTS AND DISCUSSION

Once we have trained out network we get the results of the training. We have saved our bots during training as well so that we can see how they play.

First Time training:

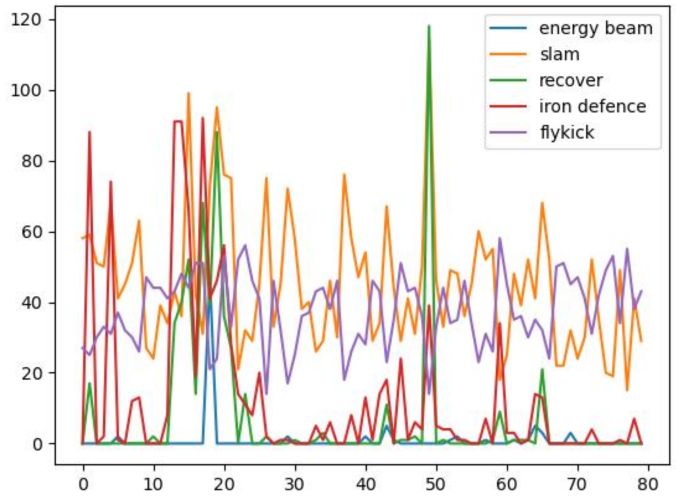


In the above graph, X-axis represents the iteration of the training loop, while Y-axis represents the number of times that particular move was used in quarter finals (top 16).

It is very clear from the graph that slam was overused while other 2 offensive moves fly kick and energy beam were disadvantageous. Battling the saved bots shows that the strategy they developed was to use iron defense a bunch of times to better their defenses and then to keep slamming each other and use recover once hp reaches below a threshold. Although an interesting strategy, but since no one used remaining 2 moves signify that the strategy is overpowered and might not have any counter. Hence this tells us to add some kind of counter to this strategy or to buff up other moves or some other balancing strategy.

Second Time training:

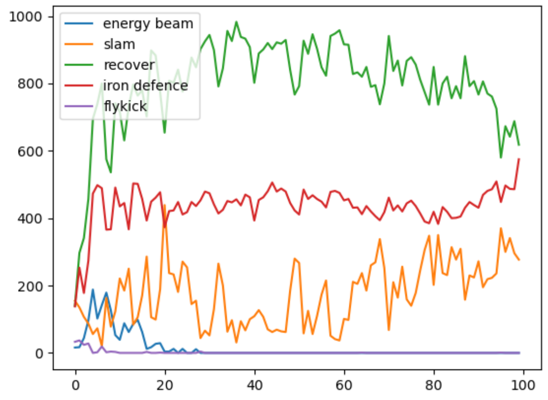
This time we nerfed down iron defense from giving +40 defense to just +20 defense and using iron defense also decreases agility.

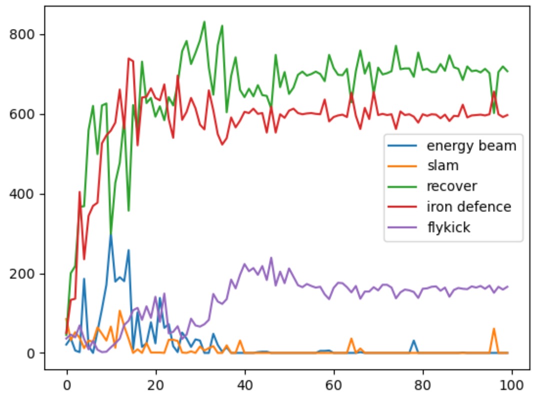


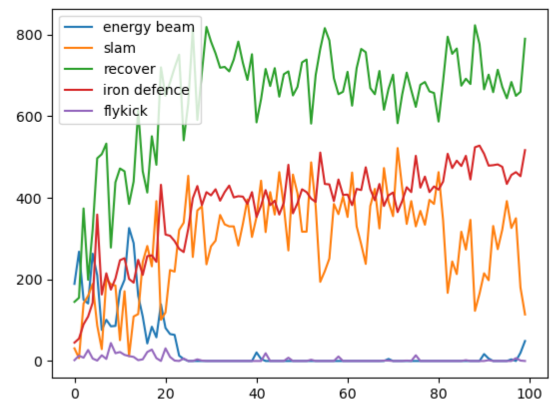
Surprisingly with just that, Iron defense is no longer a viable strategy. As we can see here, current dynamics of game are fairly more balanced than the previous. Fly kick and Slam still dominate at the end, however there were times when other strategies worked great as well. But still, usage of iron defense, energy beam and recover were almost zero. Hence further balancing is necessary.

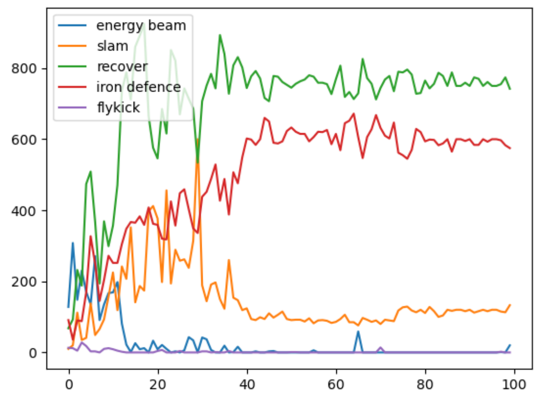
Further Balancing:

After few tweaks to game, we get the statistics as below.









5. CONCLUSIONS AND FUTURE ENHANCEMENTS

Using Deep learning agents to play game brings in a lot of valuable data to developers. Balancing a game is very hard and time consuming. Often the game is balanced only after the initial release. But learning agents can give us a quantitative and much more reliable approach to balance the game in alpha stages itself which would result in much better experience for early adopters of the game essentially kick-starting the game with initial momentum.

Using Learning agents also exposes some bug which the bots might be exploiting. Play testing with reinforcement learning bots is not very time consuming and is much cheaper and faster than conventional play testing. However conventional playtesting cannot be skipped since many bugs may not be identified by when testing with bots.

This method could also be used to develop large amounts of smart and challenging AI bots instead of hardcoded bots.

Limitations of agent based testing are:

1. Some games might depend on reflexes of the player and balancing such a game could prove to be difficult since bots might easily choose strategies which are very hard for human players to take up practically.
2. Game would still need to manually tweaked to get balanced game. This could still take some time and effort

REFERENCES

[1]. PyTorch documentation: https://pytorch.org/docs/stable

[2]. Evolutionary Learning: Advances in Theories and Algorithms, Zhi-Hua Zhou, Yang Yu, Chao Qian, doi :10.1007/978-981-13-5956-9

[3]. G. Skinner and T. Walmsley, "Artificial Intelligence and Deep Learning in Video Games a Brief Review," 2019 IEEE 4th International Conference on Computer and Communication Systems (ICCCS), Singapore, 2019, pp. 404-408, doi: 10.1109/CCOMS.2019.8821783.

[4] Playing Atari with Deep Reinforcement Learning, Mnih et al, 2013. **Algorithm: DQN**

[5] Evolution Strategies as a Scalable Alternative to Reinforcement Learning, Salimans et al, 2017. **Algorithm: ES.**