A REPORT
ON

‘Solving Cart Pole problem with imperfections’

BY

Name of the Students	ID Number
ANKIT AGARWAL	***2016B5A70468G***
ARCHIT JAIN	***2015B4A30369G***

In partial fulfilment of the course
BITS F464
Machine Learning

BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE, PILANI

April, 2020

ACKNOWLEDGEMENT

We would like to thank Prof. Ashwin Shrinivasan and the TAs of the course BITS F464 for this excellent opportunity to do this project and guide us through the difficulties. We would also like to thank the developers of Python API, openAI's gym environments which enabled us to do the project. We would also like to thank Google Colab for giving us the GPU required to run the code.

Without the help and contribution of the above-mentioned people, this project would not have been a success.

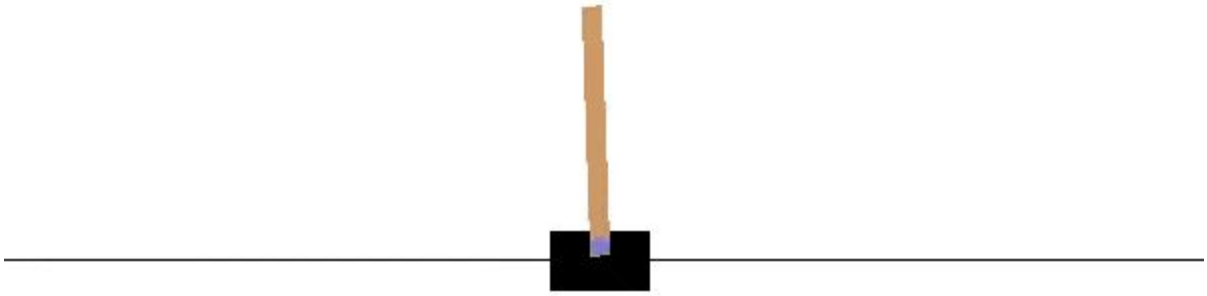
ABSTRACT

The objective of this project was to solve a cart-pole problem in different scenarios with additional constraints. The way deep-learning works, anything that can be simulated can be easily learned by AI. For this project we used openAI's "CartPole" environment. The idea of CartPole is that there is a pole standing up on top of a cart. The goal is to balance this pole by moving the cart from side to side so as to keep the pole balanced upright such that the pole doesn't fall down. We solved the problem by using Genetic Algorithms.

TABLE OF CONTENTS

1 INTRODUCTION

The Cart Pole problem is a task in which there is a cart unbalanced pole attached to it. It looks as shown below:-



The Task to be performed is to balance the rod. To balance the rod we can move the cart towards right or the left. The game ends if either cart moves more than 2.4 units in distance from the center or the angle this rod makes with the vertical is more than 15 degrees.

1.1 AIM

The aim of the project is to build a learning model that'll balance the pole on a cart which is running on a flat-terrain with a noisy set of parameters like action and sensors. Moreover the values for gravity and friction randomly vary at each step for the same episode.

2 Final algorithm used:- Genetic Algorithm

2.1 Intro to Genetic Algorithm

Genetic Algorithms are a family of computational models inspired by evolution. These algorithms encode a potential solution to a specific problem on a simple chromosome like data structure and apply recombination operators to these structures so as to preserve critical information. Genetic algorithms are often viewed as function optimizers although the range of problems to which genetic algorithms have been applied is quite broad.

An implementation of a genetic algorithm begins with a population of typically random chromosomes. One then evaluates these structures and allocates reproductive opportunities in such a way that those chromosomes which represent a better solution to the target problem are given more chances to reproduce than those chromosomes which are worse solutions. The goodness of a solution is typically defined with respect to the current population.

https://citeseerx.ist.psu.edu/viewdoc/download;jsessionid=507F123B14A421F5EDB3FD29B50D87DC?doi=10.1.1.184.3999&rep=rep1&type=pdf

2.2 Our Implementation of the Genetic Algorithm

We begin with initially playing a 1000 games (Initial Population) with random choices. In this initial population some games have higher scores than others(due to “luck” or statistical variance). These games with higher scores are more “fit than others”. We use the games with top 100 scores to train the neural network. Now we have a neural network trained using the top 100 games of the initial population. We use this neural network to create a new population(play 1000 games). Now we do the same thing with the new population and train a new neural network. This process is repeated until required fitness is achieved. Each new population can be called a generation.

3 Implementation details

3.1 Generating Population

To create a population of any generation, we have written the population function. This function takes in the model and the number of games to play (initial_games) and returns the mean of the top 100 games and the training data of the top 100 games. The training data contains observation of the step and the move made with that observation. If the model is not provided the function generates models with random action, otherwise the function generates population using the action predicted by the neural network model passed.

```
env = gym.make("CartPole-v1")
env.reset()
goal_steps = 500

def population(model=False,initial_games=1000):
    training_data = []
    scores = []
    accepted_scores = []
    all_game_memory=[]
    for _ in range(initial_games):
        score = 0
        game_memory = []
        prev_observation = []
        for _ in range(goal_steps):
            if model==False or len(prev_observation)==0:
                action = random.randrange(0,2)
            else:
                action =
np.argmax(model.predict(prev_observation.reshape(-1,len(prev_observation)))
[0])
            observation, reward, done, info = env.step(action)
            if len(prev_observation) > 0 :
                game_memory.append([prev_observation, action])
            prev_observation = observation
            score+=reward
            if done:
              all_game_memory.append(game_memory)
```

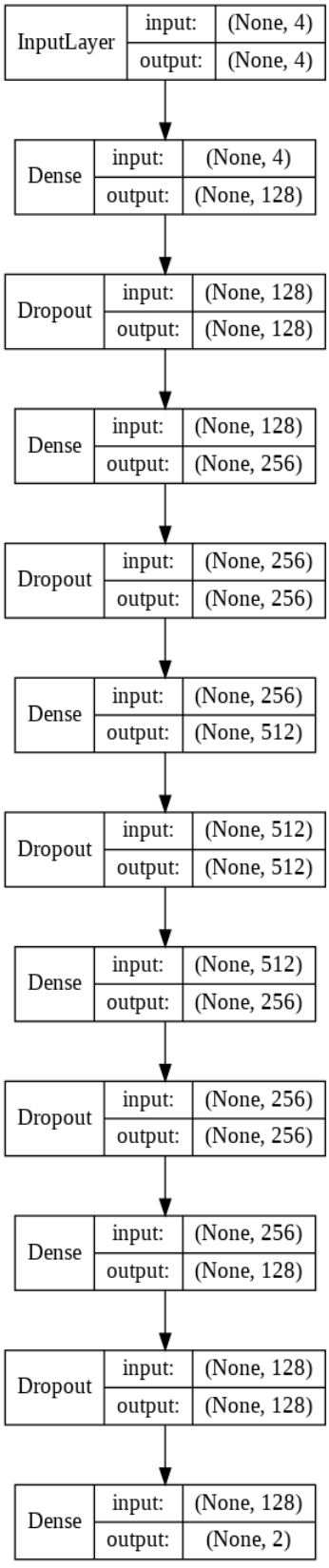
```
            scores.append(score)
            break
        env.reset()
    top100=sorted(range(len(scores)),key=lambda i:scores[i])[-100:]
    for ind in top100:
        accepted_scores.append(scores[ind])
        for data in all_game_memory[ind]:
            if data[1] == 1:
                output = [0,1]
            elif data[1] == 0:
                output = [1,0]
            training_data.append([data[0], output])
    # some stats here, to further illustrate the neural network magic!
    # print('Average accepted score:',mean(accepted_scores))
    print('Median score for accepted scores:',median(accepted_scores))
    print(Counter(accepted_scores))
    del all_game_memory
    return mean(accepted_scores),training_data
```

3.2 The Neural Network Architecture

The neural network Architecture we are using is a typical neural network architecture. The model is generated using this function below which takes input_size as the input and gives out the compiled model.

```
from keras.layers import Dense, Embedding, Dropout, Input
import keras
def neural_network_model(input_size):
    input1 = keras.layers.Input(shape=(input_size,))
    x = keras.layers.Dense(128, activation='relu')(input1)
    x=Dropout(rate=0.2)(x)
    x = keras.layers.Dense(256, activation='relu')(x)
    x=Dropout(rate=0.2)(x)
    x = keras.layers.Dense(512, activation='relu')(x)
    x=Dropout(rate=0.2)(x)
    x = keras.layers.Dense(256, activation='relu')(x)
    x=Dropout(rate=0.2)(x)
    x = keras.layers.Dense(128, activation='relu')(x)
    x=Dropout(rate=0.2)(x)
    out = keras.layers.Dense(2,activation='softmax')(x)
    model = keras.models.Model(inputs=[input1], outputs=out)
```

```
    model.compile(loss = 'categorical_crossentropy', optimizer='adam',metrics =
['categorical_crossentropy'])
    return model
```


InputLayer
input: (None, 4)
output: (None, 4)
Dense
input: (None, 4)
output: (None, 128)
Dropout
input: (None, 128)
output: (None, 128)
Dense
input: (None, 128)
output: (None, 256)
Dropout
input: (None, 256)
output: (None, 256)
Dense
input: (None, 256)
output: (None, 512)
Dropout
input: (None, 512)
output: (None, 512)
Dense
input: (None, 512)
output: (None, 256)
Dropout
input: (None, 256)
output: (None, 256)
Dense
input: (None, 256)
output: (None, 128)
Dropout
input: (None, 128)
output: (None, 128)
Dense
input: (None, 128)
output: (None, 2)

3.3 Training model

The training is done using the train_model function. This function takes training_data, model and number of epochs. It trains the a new model if model parameter is not passed, else it trains the existing model using the training data.

```
def train_model(training_data, model=False,epochs=3):

    X = np.array([i[0] for i in
training_data]).reshape(-1,len(training_data[0][0]))
    # print(X)
    y = np.array([i[1] for i in training_data])
    # print('y\n\n\n\n',y)
    if not model:
        model = neural_network_model(input_size = len(X[0]))
    # print('X shape',X.shape)
    model.fit(x=[X], y=y, epochs=epochs,batch_size=100,verbose=1)
    return model
```

3.4 Training multiple generations:-

This function is used to actually call all the other functions and run the whole algorithm. The input n is the number of generations you want to simulate. if the model is given, the model is used for the first generation (this parameter is used to continue the training with the previously used). The initial_games parameter is used to set the number of initial games needed.

```
def train_n_gens(n,model=False,initial_games=1000):
    for i in range(n):
        if i==0 and model==False:
            prev_mean,training_data = population()
        else:
            try:

prev_mean,training_data=population(model=model,initial_games=initial_games)
            except NameError:

prev_mean,training_data=population(model=model,initial_games=initial_games)

        model = train_model(training_data,epochs=10)
        # logging.info("%03d prev mean:- %f"%(i,prev_mean))
```

```
        model.save('/content/drive/My
Drive/Ml_para/ml_task1/model_%03d.h5'%(i))
    final_population(model)
```

3.5 Function for final test output:-

This function is used to get the average of 100 games for the given model.

```
def final_population(model):
    scores = []
    choices = []
    each_game in range(100):
        score = 0
        game_memory = []
        prev_obs = []
        env.reset()

        for _ in range(goal_steps):
            # env.render()

            if len(prev_obs)==0:
                action = random.randrange(0,2)
            else:
                action =
np.argmax(model.predict(prev_obs.reshape(-1,len(prev_obs)))[0])

            choices.append(action)

            new_observation, reward, done, info = env.step(action)
            prev_obs = new_observation
            # game_memory.append([new_observation, action])
            score+=reward
            if done:
              scores.append(score)
              break

        # scores.append(score)

    print('Average Score:',sum(scores)/len(scores))
    print(Counter(scores))
    print('choice 1:{}  choice
0:{}'.format(choices.count(1)/len(choices),choices.count(0)/len(choices)))
    return sum(scores)/len(scos)
```

3.6 Training Parameters:-

The Final Submitted model has been trained for 60 generations in cartpole task1 environment and 5 generations in cartpole task3 environment.

4 Scope for improvement

After training we realised instead of training for new models every generation, we should have used the old model and trained with the new weights, as that would make the training of the new models faster.

5 Results and conclusions

We were able to achieve a score of 495 ± 5 for all the three tasks. The model that we submitted was trained on task1 and task3, but it was working well for task2 as well. The score varies because of reproducibility issues with using OpenAI Gym.

https://harald.co/2019/07/30/reproducibility-issues-using-openai-gym/

6 The other Algorithms we tried

- First we began by trying the algorithm similar to what **sentdex** uses in his tutorial (which was specified in the resources for this project). We were able to achieve an average score of **175** using this approach.
- Next we changed the code we used in the above task to make it **similar to the genetic algorithm**. But, instead of taking the top n of the games we took all which satisfied a score requirement. This score requirement increased every generation. Through this algorithm we were able to achieve a score of **280**.
- After this we tried a basic code for **q learning**. This gave us an average score of **350**.
- Finally we upgraded the second algorithm to take the top 100 of every generation and were able to get an average score of 495 in every task. This algorithm also gave an **average score of 500** in some runs of 100 games. This cannot be reproduced using seed because of the reason mentioned above. This change was inspired by the AI Learns to run video of code bullet on youtube. (the link is in references.)

7 Individual Contributions

We divided our roles in the project such that Archit would try Q Learning models and Ankit would try Genetic Algorithms. The Report is a joint contribution. For the final model we trained on the saved weight of task 1 and the other person did the training for task 3.

REFERENCES

Keras-Documentation-https://keras.io/

numpy-https://numpy.org/

Open ai gym:-http://gym.openai.com/docs/

https://www.youtube.com/watch?v=62IheUGZQLU (AI Learns to run)

Papers:-

https://www.tandfonline.com/doi/abs/10.1080/00207729808929517

https://arxiv.org/abs/1712.06567

https://www.sciencedirect.com/science/S0020025502002232article/pii/

http://www.cs.utexas.edu/~ai-lab/pubs/stanley.gecco02_1.pdf

https://dl.acm.org/doi/abs/10.1145/1143997.1144202

http://www.jmlr.org/papers/v7/whiteson06a.html