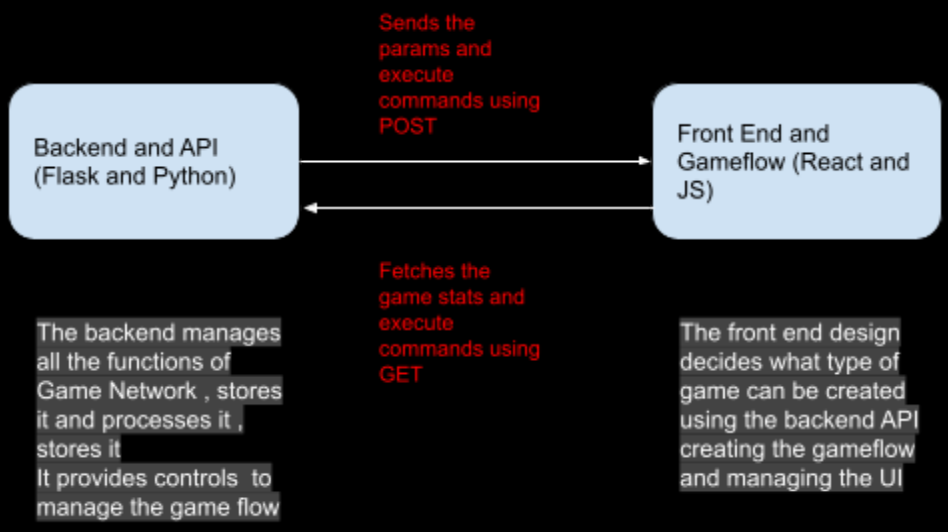
The Butterfly

Introduction

The Butterfly is a simulation model that is used to simulate the butterfly effect on human decision-making. The model uses Convolutional Neural Networks to create the character and player model. It also uses various types of learning methods to map the interactions between the characters.

The Simulation is divided into two parts:



The backend is made using Flask and Python to manage the multiple Neural networks and provide an interface for the Game model. It is the place where all the interactions take place and the character models, day state, saves are stored. It also provides the API calls to manage different types of Game flows

The Front End will be made using JS and possibly some framework that will control which type of game flow will be there, set rules for the simulation and most importantly provide a UI for the game.

Inspiration

The project is inspired from several well known titles like Until Dawn and Life is strange (series)

One of the few similar projects would be What If Elon Musk Didn’t Start Tesla by Jabrils

Tech used

- Flask (for the backend)
- Python 3.2.8
- Numpy
- Currently no ML library is used (will try to add it later)

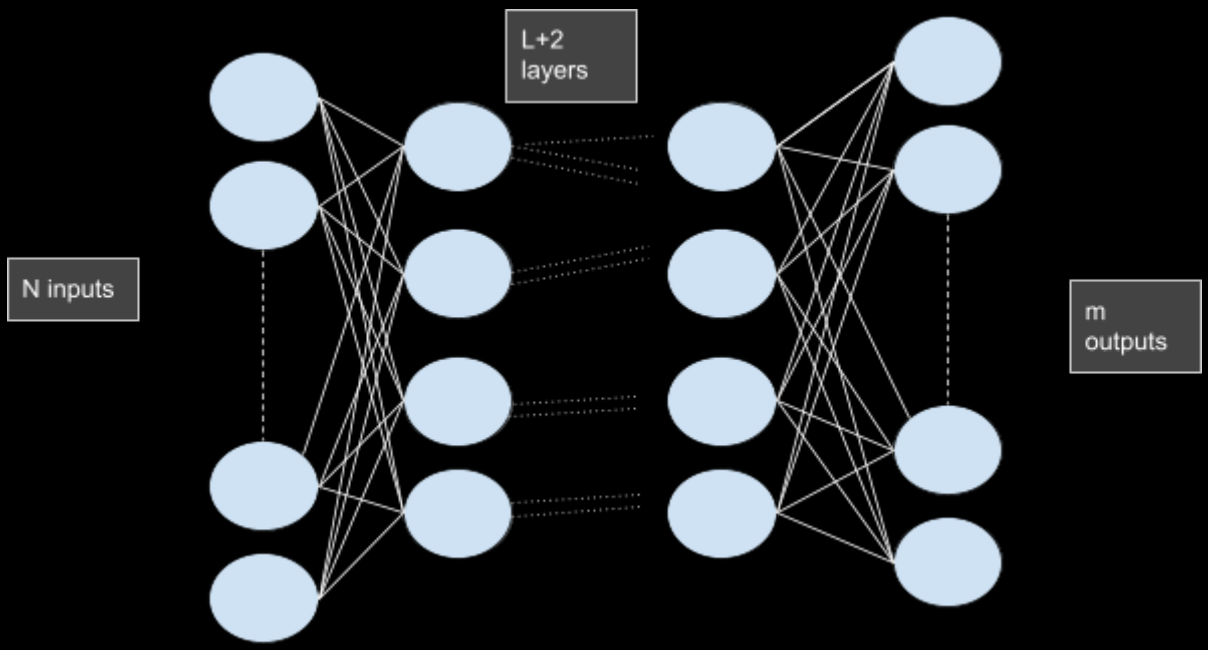
Backend

The Backend is built componentially and could be broken down into the following components

1) Neural Network Model(NN.py): The base of the backend is a NN model that provides different types of learning models.
2) Kar Models(kar.py): The character models uses the NNs created before to craft a character that has different types of interactions with other characters. Currently, there are two types of Characters models and their interactions
    a) The player model: The player model is the more complex model that takes the influences as input and gives the choices as output
    b) The Other model: The other model is a much simpler model that takes the choices as an input and provides the conformity as output and could also be run in the reverse to give the choices as output given the conformity
3) Game Model(Game.py): This file consists of the game initialization model and different flows of the game, the day cycle, saving and retrieving a day
4) API(main.py): This file consists of the API that is built using flask and supports calls for most of the Game.py ‘s functionalities and additional basic landing page and error handlers

Neural Network Models (NN.py):

The neural network is a simple CNN network with different types of learning



Learning Models:

a)Backpropagation(backpropogation.py): The simple backpropagation learning model for the CNN is implemented that is primarily used in discussion mode in the further outputs

```
def backprop(net, x, y):
       nabla_b = [np.zeros(b.shape) for b in net.biases]
       nabla_w = [np.zeros(w.shape) for w in net.weights]
       activation = x
       activations = [x]
       zs = []
       for b, w in zip(net.biases, net.weights):
           z = np.dot(w, activation)+b
           zs.append(z)
           activation = sigmoid(z)
           activations.append(activation)
       delta = net.cost_derivative(activations[-1], y) * \
```

```
            sigmoid_prime(zs[-1])
        nabla_b[-1] = delta
        nabla_w[-1] = np.dot(delta, activations[-2].transpose())
        for l in range(2, net.num_layers):
            z = zs[-l]
            sp = sigmoid_prime(z)
            delta = np.dot(net.weights[-l+1].transpose(), delta) * sp
            nabla_b[-l] = delta
            nabla_w[-l] = np.dot(delta, activations[-l-1].transpose())
        return (net,nabla_b, nabla_w)
```

b) MultiLayer Hebbian[link]: A Hebbian styled multilayer learning model is used by the models for primarily the influence phase:

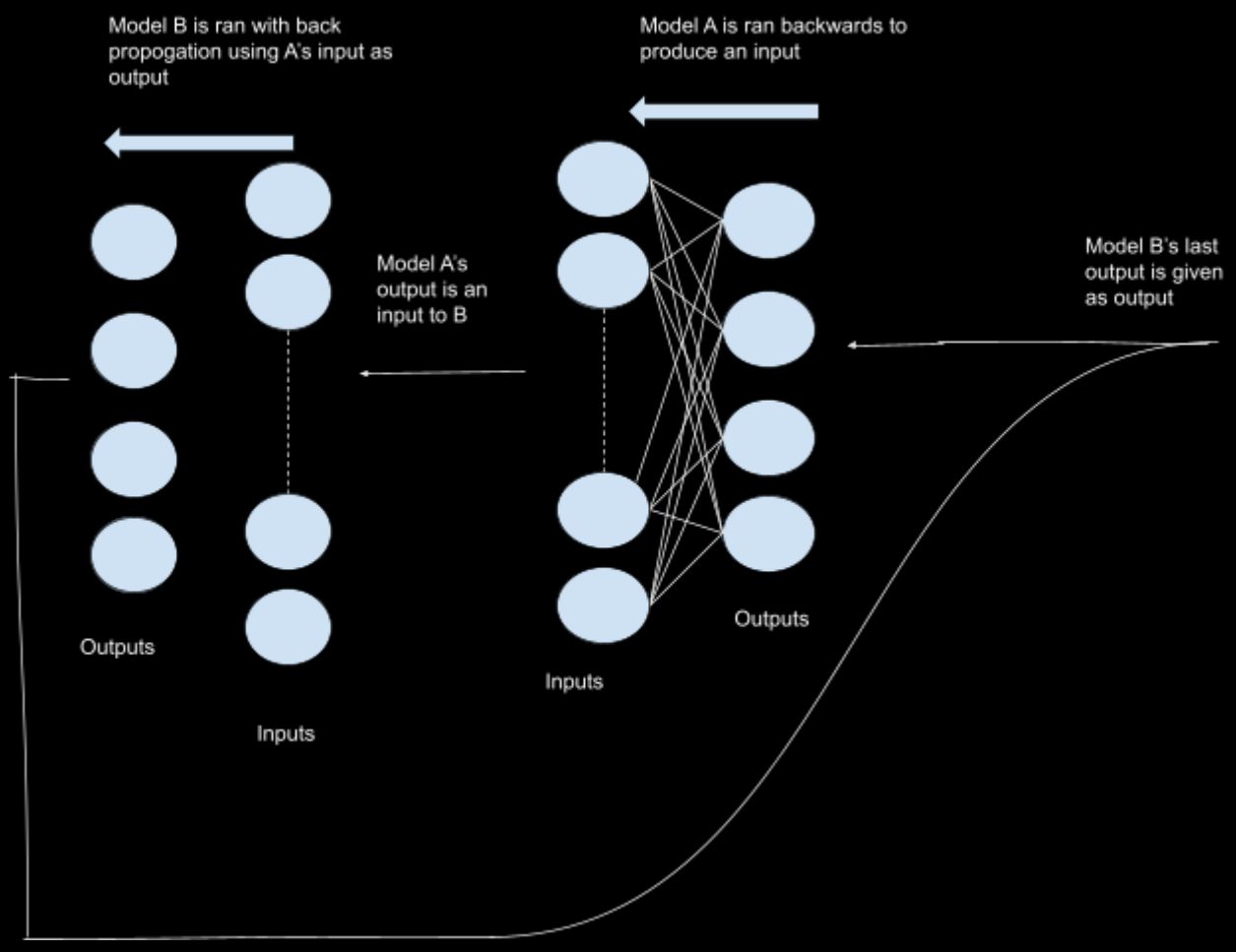
```
def hebbian(self,input,eta,forget):
        out=[]
        a =input

        out.append(a)
        for b, w in zip(self.biases, self.weights):
            a = sigmoid(np.dot(w, a)+b)
            out.append(a)
        nabw=[]
        for j,i in zip(out[1:],out[:-1]):
          nabw.append(np.dot(i,np.transpose(j)).transpose())

        #implement forgetting factor
        self.weights=[w+eta*nw  for w,nw in zip(self.weights,nabw)]
        #check the bias update
        self.biases=[b+eta*nw for b,nw in zip(self.biases,out[1:])]
```

c)RBM styled Learning (Inspiration): A RBM styled learning that will be used by the other kar models for the discussion interaction.

In this, the model first runs in reverse to calculate the output, and then that output is fed to another model as its input while using backpropagation



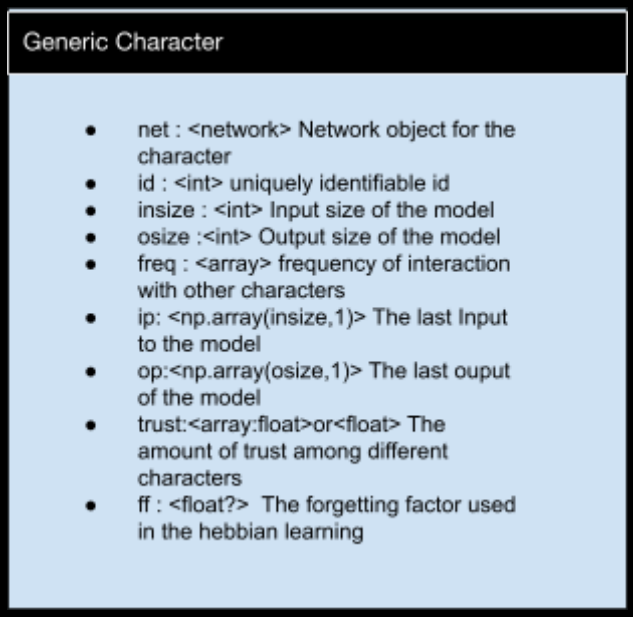
Save and Load: The NN.py also contains a feature to save and load the data for the timeline management

```
def save(self):
       data = {"sizes": self.sizes,
               "weights": [w.tolist() for w in self.weights],
               "biases": [b.tolist() for b in self.biases]}
       return data

   def load(self,data):
     self.weights = [np.array(w) for w in data["weights"]]
```

```
    self.biases = [np.array(b) for b in data["biases"]]
```

Character Models (kar.py):



Some terms :

i) Chaure[ ]: Chaure consists of all the Character objects and the player object at the last

ii) trust: Trust is defined both as the learning rate pair between two characters and it can be changed only for the player character using the artificialtrusts( )

iii) freq: Frequency is the number of times two characters meet on an average in a day and further by it influence each other

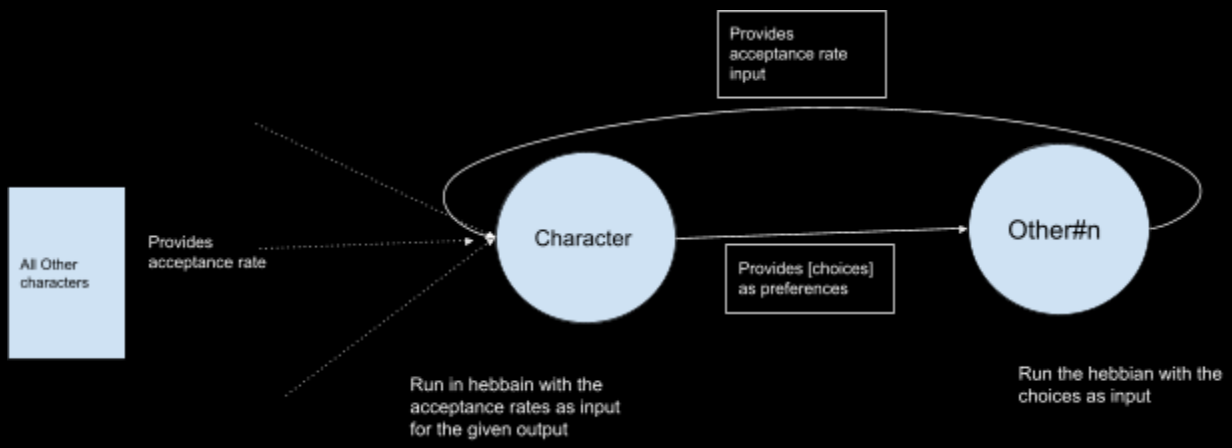
iv) forget_factor: Since the Hebbian learning can go so strong that the output is just influenced by one character the game model using the forgetting factor to manage using a forget factor determined by the number of total meetings vs the characters meeting

Code

```
def forget_factor(self,id):
  meets=0
  for i in Chaure:
    meets=meets+Chaure.freq
```

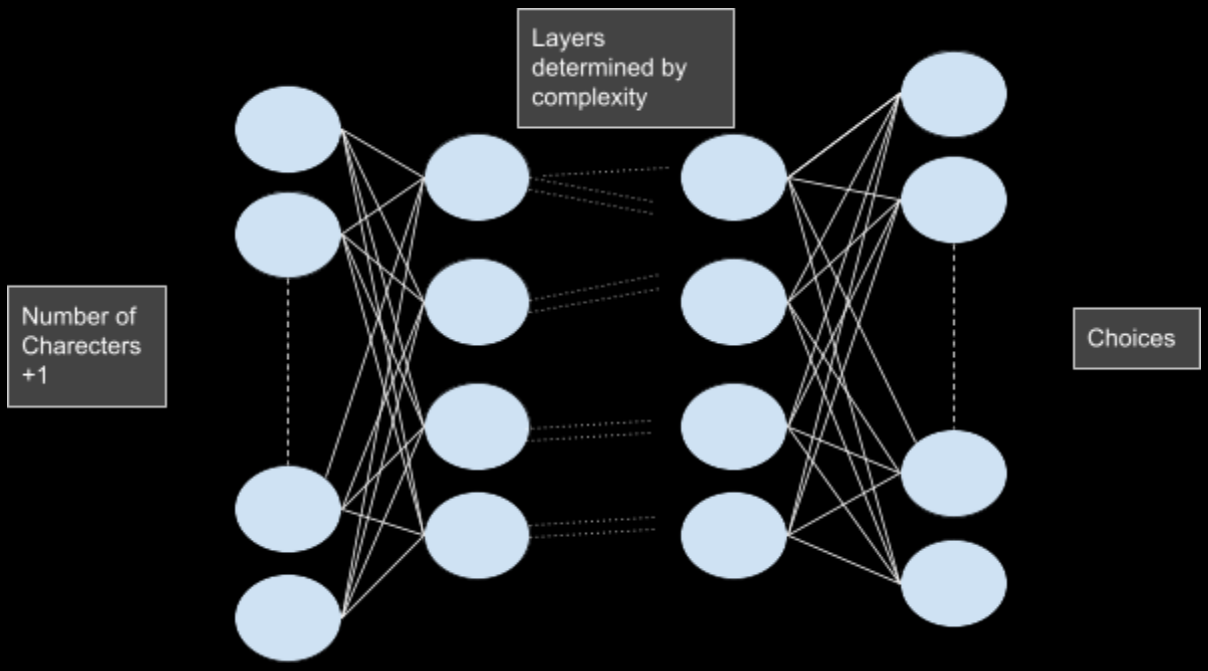
```
ff=1-(Chaure[id].freq/meets)
Chaure.ff=ff
return ff
```

v) id : id is the kar id that can be used to identify the character from the Chaure array C



Using the base NN we construct two types of characters and their interactions

1)Player: The player model is one of the more complex models in the simulation with More interactions and layers and much more control



There are Number of Charecters+1 input to the player model where each input is the amount of acceptance by the other character for our character's current choices + The last one being the characters self-acceptance (which is generally 1)

Interactions for the player :

1) Influenced :
   Influenced determines how a player will be influenced by other character s by taking their acceptance output as input and applying a hebbian learning to intensify the output of the model for the input

   Code

```
#influenced defines how a player is influneced by someone
#ips is the input
#trust is the trust trust_level
#forget is the forgetting factor
 def influenced(self,id):
   #inputs to be gathered
     global Chaure
     ips=Chaure[id].op# needs to be fixed
     trust =self.trusts(id)
     forget=self.forget_factor(id)
     self.net.hebbian(ips,trust,forget)
     self.op=self.net.feedforward(ips)
```

2) Meing: Meing is just like influencing the self character and can be using as configuring the player for set choices

   Code

```
#MEing ie turning a player into me (One time only function assumes
no other influence )
#ops defines the outputs of other players
#iter defines the number of iterations
```

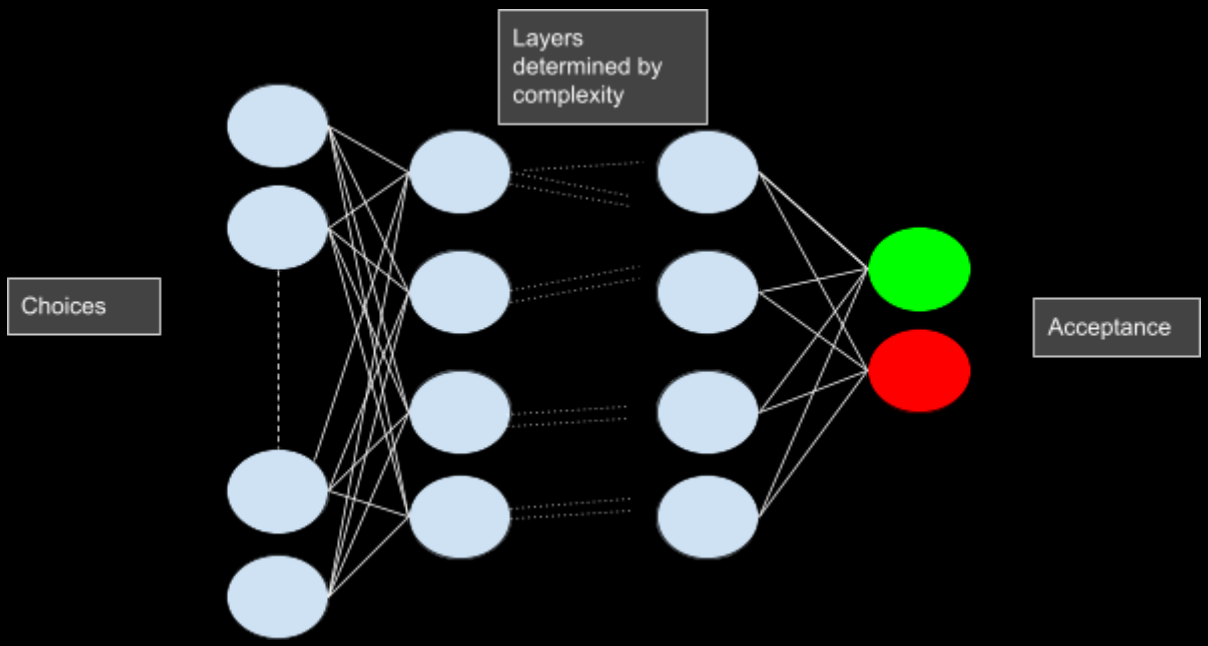
```
#selfcon is the level of confidence ie the learning rate
#ops are my choices
 def meing(self,ops,selfcon,iter):
   ips=np.zeros((self.net.insize,1))
   ips[-1]=[1]
   data = list(zip([ips],[ops]))
   self.net.QS(data,iter,selfcon)
   self.op=self.net.feedforward(ips)
```

3) Discussion: Disc with other characters is a backpropagation styled interaction that tries to set the output for the input of another character

   Code

```
 def dicu(self,id):
   global Chaure
   ops=Chaure[id].rev()
   trust=self.trusts(id)
   freq=self.freq[id]
   ips=self.ips()
   ips[id]=[1]
   data=zip([ips],[ops])
   self.net.QS(data,freq,trust)
   self.op=self.ops(ips)
```

2)Other: The other model is one of the simpler models in the simulation with much more automated flow



The other character model takes the choices of the player and the other character as input and returns an acceptance rate.

Interactions:

1) Player Influence : In this the Other model is influenced by the player characters using the hebbian learning and also includes the forgetting factor

```
def pinfluence(self):
  global Chaure
  pay_ips=Chaure[-1].op
  trust=self.tmatrix[-1]
  self.net.hebbian(pay_ips,trust,self.ff)
  self.op=self.net.feedforward(pay_ips)
```

2) Character Influence : Influencing the other characters by taking the player input we get running the model in reverse as input and then perform a MLH

```
def cinfluence(self,id):
  global Chaure
  ips=Chaure[id].ktb()
  influ=self.trusts(id)
  forg=Chaure[id].ff # needs a bit of work
```

```
    self.net.hebbian(ips,influ,forg)
```

3) Know The Character : Returns the current character choices for 100% acceptance rate

```
def ktb(self):
    ip=self.net.rev(self.op)
    return ip
    # For a bipolar output
```

4) Discussion : Discussion are only done with the player model as discussed in the player model

   Code:

```
 def disc(self):
    global Chaure
    ips=Chaure[-1].op
    ops=[[1],[0]]
    data = list(zip([ips],[ops]))
    trust=self.tmatrix[-1]
    iter=self.f[id]
    self.net.QS(data,iter,trust)
    self.op=self.net.feedforward(ips)
```

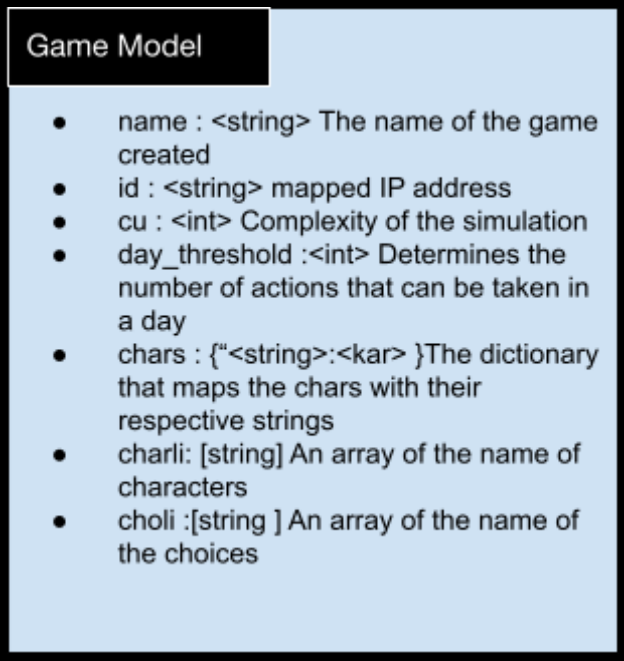
5) I Know the Character : It is a one time only function that is used to initialize the other character based on the presumption of the player at the start of the game

   Code:

```
def iktb(self,ips,iteration,selfcon=np.random.randn()):
    ops=np.zeros((2,1))
    ops[0]=[1]
    ops=[ops]
    ips=[ips]
    data=list(zip(ips,ops))
    self.net.QS(self.net,data,iteration,selfcon)
    self.op=self.net.feedforward(ips)
```

Game Models (Game.py):

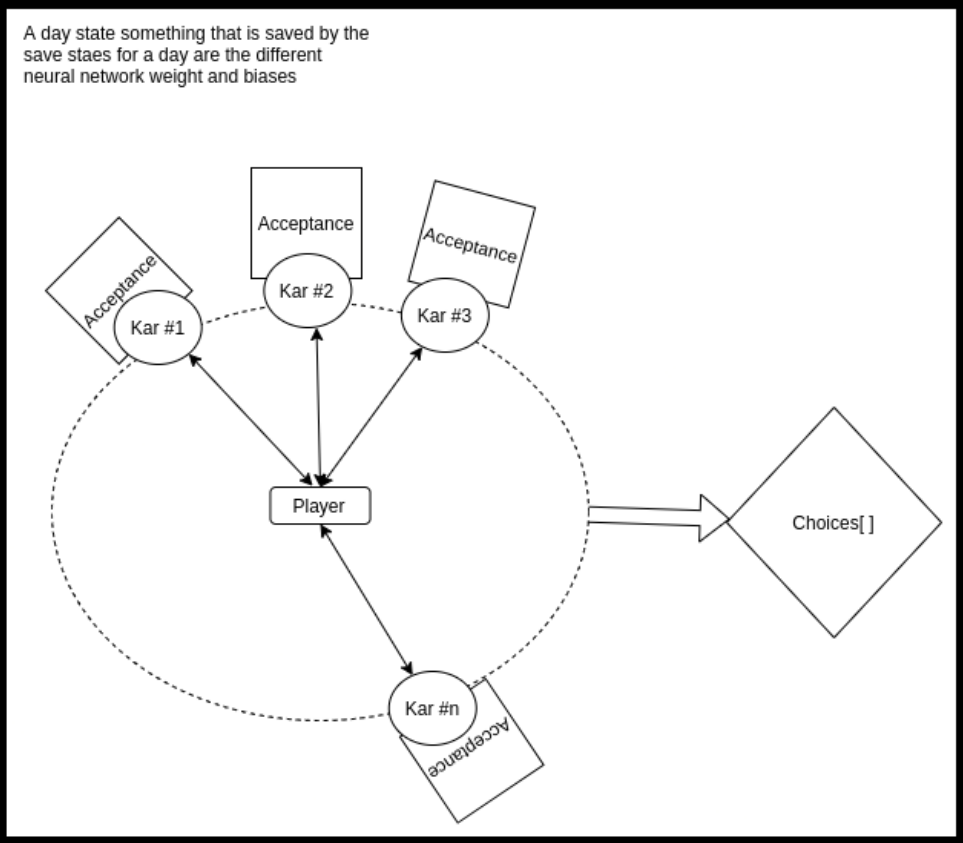
The game model helps us set up a simulation , provides us variety of flow controls and day cycle controls among other things



Static Day model

Game Model

This is a basic design of how a game would look on a static day



Influence And Discussions:

The player and character influence each other on the choices they've using their acceptance and trust values

The final choice of the player is a result of all the influences and discussions among the charecters

Some terms in the game models are

1) Name : This is the username of the person by which the save folder will be created
2) Id : This is be the Ip address logged for the username
3) charli : A named list for the characters which points to the respective character objects
4) choli: A named choice list that is mapped with the choices for better output

5) day: This is the day number that would be autoupdated after a number of action or manually when called upday( )
6) day_threshhold: Defines the number of actions that are allowed per day

Functions :

1)Start : This takes the name , choices and characters to start the game

```
def start(self,choices,char,level=[]):
   ca=len(char)
   co=len(choices)
   self.charli=char
   self.choli=choices
   if len(level)!=ca+1:
     level=[self.cu]*(ca+1)
   self.chars["You"]=Player(ca,co,level[-1])
   for i in range(ca):
     self.char[self.charli[i]]=Other(i,co,level[i])
   self.day=0
   self.save()
   return{"status":"Game created"}
```

2) pstat: Returns the current Player statistics

```
 def pstat(self):
   Me=self.chars["You"]
   choice=dict()
   for jane in range(self.choli):
     choice[self.choli[jane]]=Me.op[jane][0]
   return {"Playerop":choice}
```

3)cstat : Returns the character stats

```
 def cstat(self,name):
   Nm=self.chars[name]
   acc=self.ts(name)
   me=Nm.ip([[1],[0]])
   res=map(self.choli,me)
   return {"kar":name,"result":res,"acceptance":acc}
```

4)daystat : Returns the day stats

```
 def daystat(self):
   res={"day":self.day,"energy":self.day_threshhold}
   return res
```

5)Untouched Day : Fast forwards a day by replaying the preset interactions:

```
def utd(self):
   Me=self.chars["You"]
   Nm=self.chars
   order=set(self.charli)
   inf=dict()
   cie=list(zip(random.shuffle(self.charli),random.shuffle(self.charli)))
   for i in order:
     for j in range(random.randint(2,12)):
       res= Nm[i].pinfluence()
       self.minchange(Nm[i].id)
     inf[res["kar"]]={"Choice":res["choice"],"Level":res["inf"]}
     for x,y in cie:
       if Nm[x].id != Nm[y].id:
         for bs in range(random.randint(1,12)):
           Nm[x].cinfluence(Nm[y].id)
           Nm[y].cinfluence(Nm[x].id)
   self.day+=1
   self.day_threshhold=randint(4,10)
   self.save()
   return inf
```

6)upday: Changes the day after set number of actions have been taken

```
def upday(self,v=1):
   self.utd()
   self.day+=v
   self.day_threshhold=randint(4,10)
   if self.day == int(self.day):
     self.save()
   return{"day":self.day}
```

7)save: [Can’t be called from the API] : Saves the current day game file

```
def save(self):
   data={}
   Nm=self.chars
```

```
  for jane in self.charli:
    data[Nm[jane].id]=Nm[jane].net.save()
    data[Nm[jane].id]={"op":Nm[jane].op}
  data[len(Nm)]=Nm["You"].net.save()
  data[len(Nm)]={"op":Nm["You"].op}
  name=str(self.day)
  f=open(self.name+'/'+name,'w')
  json.dump(data,f)
  f.close()
  print("Save performed")
```

8)rewind : Rewind the time to certain day's state

```
def rewind(self,day):
  f = open(self.name+'/'+str(day), "r")
  data = json.load(f)
  f.close()
  for id in data:
    Chaure[id].net.load(data[id])
    Chaure[id].op=data[id]["op"]#Needs testing
  self.day=day
  self.day_threshhold=randint(4,10)
  return {"Rewinded":True,"day":day}
```

9)discu : Provides a higher level interface for the discussion interaction by using names as references

```
def discu(self,name):
  self.limits()
  if name in self.charli and self.day!=0:
    Me=self.chars["You"]
    Op=self.chars[name]
    for i in range(random.randint(3,5)):
      Me.dicu(Op.id)
      Op.disc()
    return {"kar":name,"eft":max(Me.op)[0],"inf":max(Op.tb)[0]}
  else:
    return {"kar":name,"eft":0,"inf":0}
```

10)fight: Starts a series of cinfluence between the other characters

```
def fight(self):
   Nm=self.chars
```

```
    Ak=random.randint(4,25)# big numbers
    for i in range(Ak):

cie=list(zip(random.shuffle(self.charli[:-1]),random.shuffle(self.charli[:
-1])))
     for x,y in cie:
       if Nm[x].id != Nm[y].id:
         for bs in range(random.randint(1,12)):
           Nm[x].cinfluence(Nm[y].id)
           Nm[y].cinfluence(Nm[x].id)
    return{"fight":Ak}
```

11) Minchange :[Not callable from the API] : Changes the other characters randomly at some point

```
def minchange(self,id):
   iddt=self.chars[id]
   se=random.randint()
   if se>10:
     ip=random.rand(len(self.choli),1)
     iddt.iktb(ip,se,1)
```

12) Trust adjustment : We can adjust how much the player trust the other characters using the following functions

```
   #Trust adjust for players

 def trusty(self,name,trust):
   Me=self.chars["You"]
   Nm=self.chars[name]
   Me.trust[Nm.id]=trust
   return {"kar":name,"Action":"Trust Updated"}

   #Will never be used
 def fortrusty(self,id,trust):
   Me=self.chars["You"]
   Me.trust[id]+=random.rand(-trust,trust)

 #To implement the trust matrix could be dangerous need to add fuzziness
to it
```

```
def doawholetrusty(self,tmatrix):
  for jane in len(self.chars)-1:
    self.fortrusty(jane,tmatrix[jane])
  return {"Action":"Trusts Updated"}


def artificialtrusty(self,name):
  Me=self.chars["You"]
  Nm=self.chars[name]
  trust=Me.artificial_trust(Nm.id)
  return {"kar":name,"artrust":trust}
```

13) Distancing functions : For distancing players from a certain character we change the frequencies of the meet among the players

```
def distancedu(self,name):
  Me=self.chars["You"]
  Nm=self.chars[name]
  f=Me.freq[Nm.id]
  Me.freq[Nm.id]-=rand.randint(f/5,f/3)
  return {"kar":name ,"Freq":"reduced"}



def undistancedu(self,name):
  Me=self.chars["You"]
  Nm=self.chars[name]
  f=Me.freq[Nm.id]
  Me.freq[Nm.id]-=rand.randint(f,f*2)
  return {"kar":name ,"Freq":"increased"}
```