Engineering Document Design for CSCI599

Zhihan Zhang, ChuanZhe Li, Jia Xu

April 2020

1 Project Goal

As the history passes, there are only two things stay in stale, survive and competition. The life in earth can be simplified to the maximum their ability to live, to become healthy, the simplest goal for nature beings may induce millions of undocumented strategies for every unique species to sustain themselves.
Here we want to mimic this status and find hidden strategies based on simulator, thus we want to create many agents fight for living everyone need to acquire daily necessities to survive and by adopting deep reinforcement we may mimic the process of each species can learn from the terrain and conditions provided by the infinite and may induce serendipitous results.

2 Background and Game Overview

2.1 The genre of the game

RPG(Role-playing games) is a popular genre with thousands of famous Ips such as Final Fantasy and pokemon, they will create many scenarios and players need to make curtain decisions in a given term for further steps.
Here we want to use Massively Multiplayer Role-playing game for this sand-box simulator which allows hundreds of players play on the same map to make their decisions and live as long as possible.

2.2 How the game played

The artificially intelligent agents compete to survive, and learn new skills in a long the way. Every agent has three parameters: food, water and health. At each server tick (time step), agents may move one tile and make an attack. The AI agents must acquire food and water to stay alive, and they move across the map to gain both. This brings them into conflict with other agents and requires the AI to move carefully in order to maximize the chance of finding resources as it explores. Agents forage for food and must refill their water supplies .[14]

We set environment with different terrains to increase the reality of the game. there are six types of terrain types and each of them have different properties.

terrain types	properties
Stone	inaccessible
Grass	Traversable
Forest	have food & traversable
Water	have water & inaccessible
Scrub	Traversable

Table 1: Terranin types

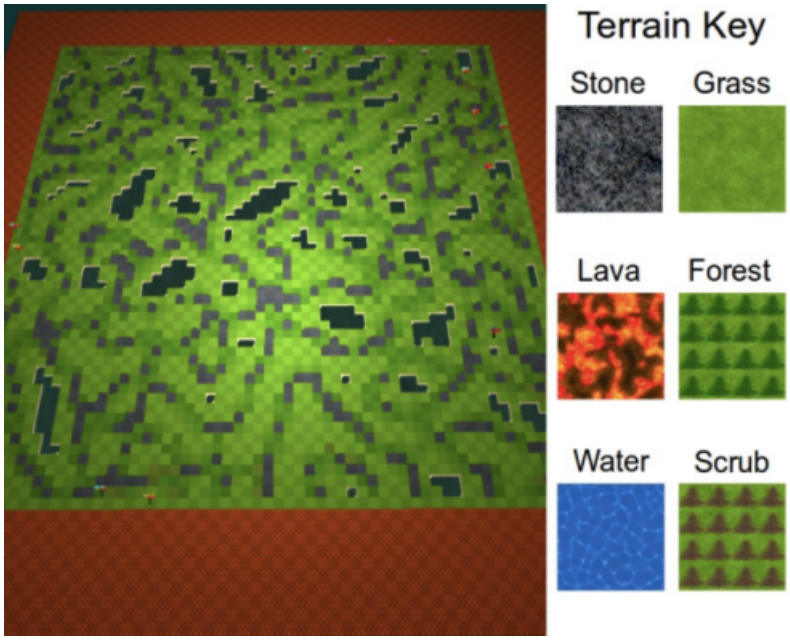


Figure 1: Terrain and Map Example

Also, we set three attack options for agents, each of them has different damage value and trade offs.
At each server tick (time step), agents may move one tile or just stay there. Stepping on a forest tile or next to a water tile refills a portion of the agent's food or water supply, respectively.
The training purpose is to make the agents stay alive as long as much. Each agent has an full observation. They make the best decision according to the current status and gain wise decisions on the next step.

Attack types	Description
Mage	Inflicts 10 damage at 1 range
Range	Inflicts 2 damage at 1-2 range
Melee	Inflicts 1 damage at 1-3 range and freezes the target in place for two ticks

Table 2: Attack types

3 Prior Researches

3.1 training methodology stage 1: monte-carlo tree search

At the very begining, we explored classic algorithms such as model-based learning like Monte Carlo Tree Search[9] and their application in tic-tac-toe[8] and harvesting game made by myself https://github.com/ZhiHanZ/monte_carlo_tree_search This strategy is perfect if we have small amount of agents and need less generalization but for our current problem, it cannot fulfill our requirement because the performance is bad in scale.

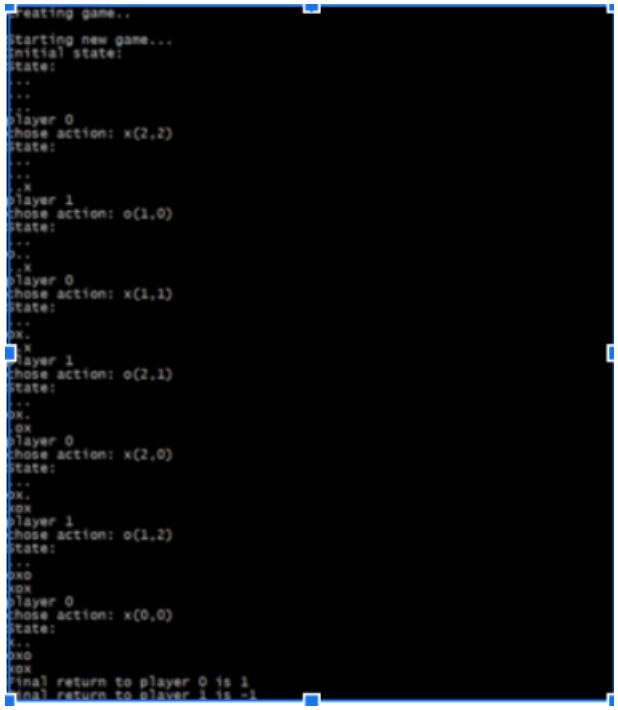


Figure 2: Tic Tac Toe Example

3.2 training methodology stage 2: unity turn-based AI, minmax pruning

After that we try to work on the turn-based strategy board game which the human player can fight against the AI player. In the mean time we learned the skills and knowledge by adopting Machine Learning-Agent which benefited a lot. We step by step deployed it on the unity environment but the problems are found that the turn-based game needs an on-demand decision making process which is still under development in Unity and we have no example as reference.[7]

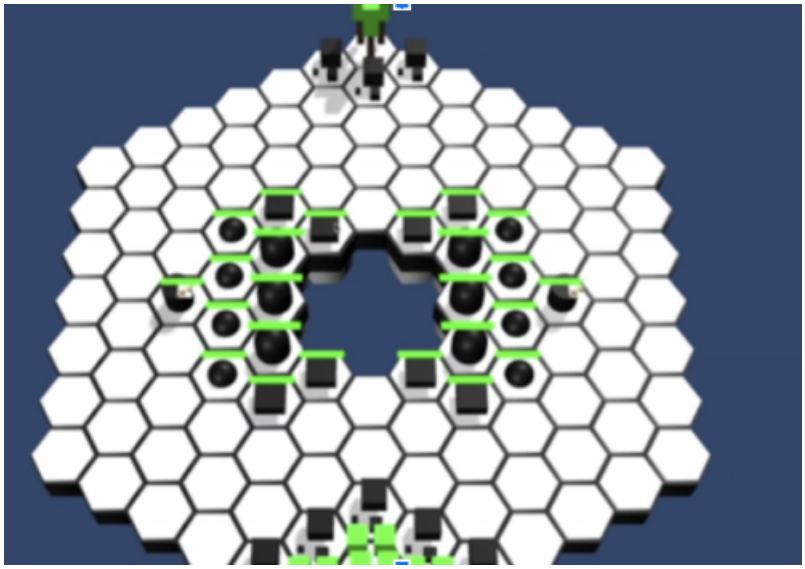


Figure 3: Unity Turn Based Game example

3.3 neural network optimization : dota2 design patterns

After building our basic training model, we tried to improve our training model by changing the neural network from one single-layer linear network to multiple-layer network, also, we tried to make the neural network has memory ability so more layers' trajectory information can be recorded and the neural network can learn from earlier data.
We investigated the algorithm of Deep RL optimization using LSTM. We found a very similar algorithm to our previous work – an AI system used for Dota2 named OpenAI Five.
Dota 2 is a multiplayer real-time strategy game, and the OpenAI Five is an AI system for Dota 2, which is developed based on large scale deep reinforcement learning and is trained with a distributed training system. Each timestep,

OpenAI Five receives an observation from the game engine encoding all the information a human player would see such as units' health, position,etc. OpenAI Five then returns a discrete action to the game engine, encoding a desired movement, attack, etc[5].
The basic idea of OpenAI Five is that it processes the complex multi-array observation space into a single vector, which is then passed through a 4096-unit LSTM. The LSTM state is projected to obtain the policy outputs (actions and value function). Each of the five heroes on the team is controlled by a replica of this network with nearly identical inputs, each with its own hidden state. The networks take different actions due to a part of the observation processing's output indicating which of the five heroes is being controlled.

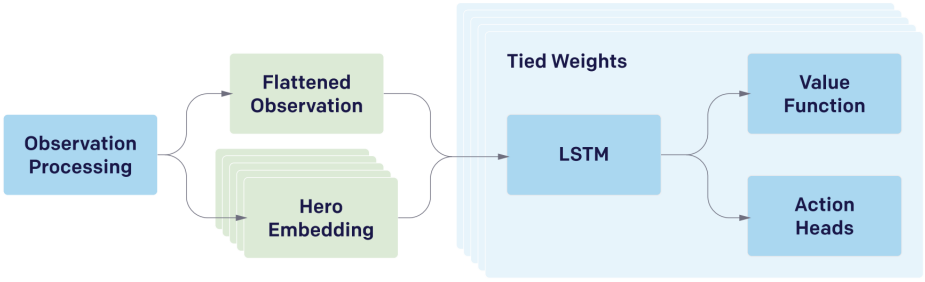


Figure 4: Simplified OpenAI Five Model Architecture

The advantage of this model is that each LSTM has a hero embedding, which allows the agent to memorize and make decisions in combination with the actions of other agents. Then we design a similar model like OpenAI Five. In this model, We embed the information from the observation of each agent, and train with LSTM, then we get an action.

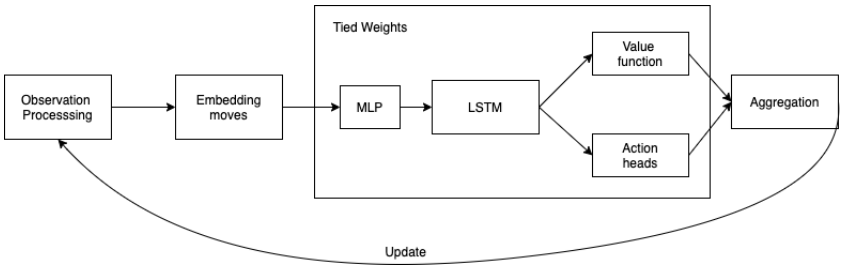
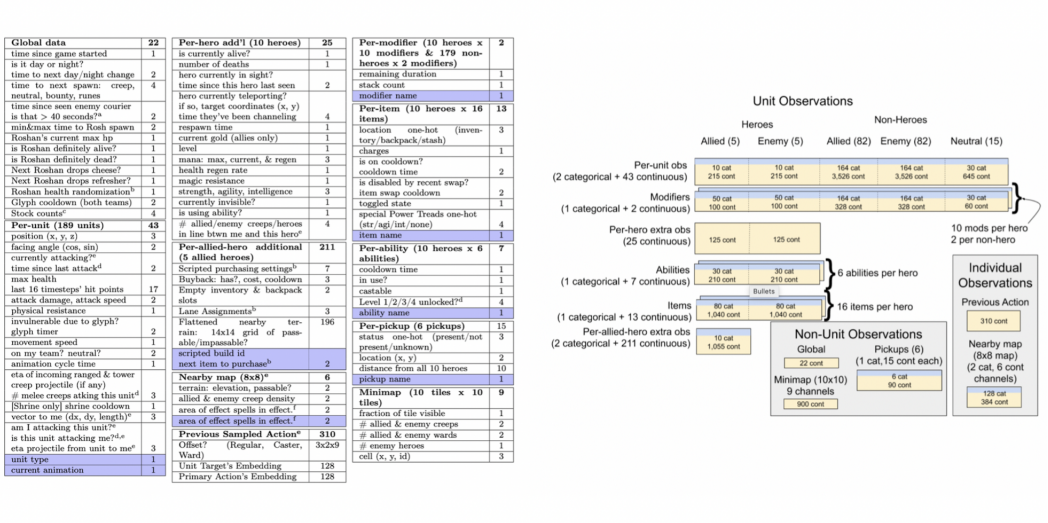


Figure 5: Our model with LSTM

However, soon we found that there exists some problems in this model. First, the usage scenario is not consistent with ours. Our observation and action space

are very small, but the space for observation and action of OpenAI Five is pretty large. For example, just for actions like updating skills, buying, selling items, there are 8-9 states. Second, LSTM is actually not suitable for massive amount



Global data	22
time since game started	1
is it day or night?	
time to next day/night change	2
time to next spawn: creep, neutral, bounty, runes	4
time since seen enemy courier	
is that > 40 seconds?[n]	2
min&max time to Rosh spawn	2
Roshan's current max hp	1
is Roshan definitely alive?	1
is Roshan definitely dead?	1
Next Roshan drops cheese?	1
Next Roshan drops refresher?	1
Roshan health randomization[b]	1
Glyph cooldown (both teams)	2
Stock counts[c]	4
Per-unit (189 units)	**43**
position (x, y, z)	3
facing angle (cos, sin)	2
currently attacking?[o]	
time since last attack[d]	2
max health	
last 16 timesteps' hit points	17
attack damage, attack speed	2
physical resistance	1
invulnerable due to glyph?	
glyph timer	2
movement speed	1
on my team? neutral?	2
animation cycle time	1
eta of incoming ranged & tower creep projectile (if any)	
# melee creeps atking this unit[d]	3
[Shrine only] shrine cooldown	1
vector to me (dx, dy, length)[e]	3
am I attacking this unit?[o]	
is this unit attacking me?[d,e]	
eta projectile from unit to me[e]	3
unit type	1
current animation	1

Per-hero add'l (10 heroes)	25
is currently alive?	1
number of deaths	1
hero currently in sight?	
time since this hero last seen	2
hero currently teleporting?	
if so, target coordinates (x, y)	
time they've been channeling	4
respawn time	1
current gold (allies only)	1
level	1
mana: max, current, & regen	3
health regen rate	1
magic resistance	1
strength, agility, intelligence	3
currently invisible?	1
is using ability?	1
# allied/enemy creeps/heroes in line btwn me and this hero[o]	4
Per-allied-hero additional (5 allied heroes)	**211**
Scripted purchasing settings[b]	7
Buyback: has?, cost, cooldown	3
Empty inventory & backpack slots	2
Lane Assignments[b]	3
Flattened nearby terrain: 14x14 grid of passable/impassable?	196
scripted build id	
next item to purchase[b]	2
Nearby map (8x8)[g]	**6**
terrain: elevation, passable?	2
allied & enemy creep density	2
area of effect spells in effect.[f]	2
area of effect spells in effect.[f]	2
Previous Sampled Action[a]	**310**
Offset? (Regular, Caster, Ward)	3x2x9
Unit Target's Embedding	128
Primary Action's Embedding	128

Per-modifier (10 heroes x 10 modifiers & 179 non-heroes x 2 modifiers)	2
remaining duration	1
stack count	1
modifier name	1
Per-item (10 heroes x 16 items)	**13**
location one-hot (inventory/backpack/stash)	3
charges	1
is on cooldown? cooldown time	2
is disabled by recent swap? item swap cooldown	2
toggled state	1
special Power Treads one-hot (str/agi/int/none)	4
item name	1
Per-ability (10 heroes x 6 abilities)	**7**
cooldown time	1
in use?	1
castable	1
Level 1/2/3/4 unlocked?[d]	4
ability name	1
Per-pickup (6 pickups)	15
status one-hot (present/not present/unknown)	3
location (x, y)	2
distance from all 10 heroes	10
pickup name	1
Minimap (10 tiles x 10 tiles)	**9**
fraction of tile visible	1
# allied & enemy creeps	2
# allied & enemy wards	2
# enemy heroes	1
cell (x, y, id)	3

Figure 6: Overview of the action space and observation space of OpenAI Five

of agents. It requires that all agents use one LSTM cell to compute the result. It is feasible for a game with 5 agents but once the number of agents become more than 1000 or 1 Million, the computation cost will not be accessible. For example, just for 1v1, the parameters amount can be pretty large: the combined policy and value network uses 158,502,815 parameters, utilizing thousands of GPUs over multiple months.

As a result, we decide not to use LSTM for our project. However, what we got inspirations from OpenAI Five is about the distributed system. The training system of OpenAI Five consists of 4 primary types of machines. Rollouts run the Dota 2 game on 51200 CPUs, and they communicate in a tight loop with Forward Pass on 512 GPUs to sample actions from the policy given the current observation. From this we found that our original training is single core, which results in low training efficiency. If more agents are to be supported, a network platform similar to distributed training must be designed, like the distributed system built in the OpenAI Five.

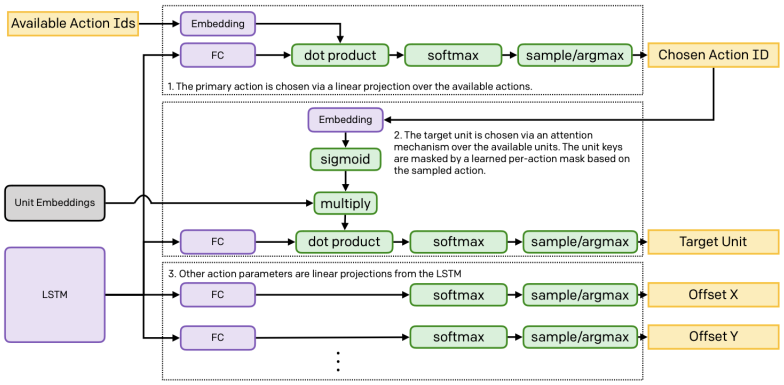


Figure 7: The hidden state of the LSTM and unit embeddings are used to parameterize the actions

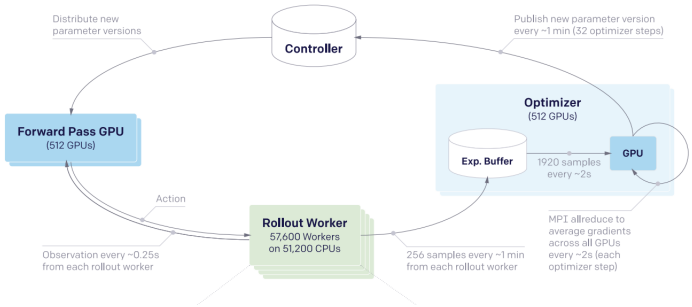


Figure 8: The System Overview of OpenAI Five

4 What we use?

4.1 Tools utilized and application fields

4.1.1 Computing and neural network

We adopted several neural network and scientific computing frameworks to accelerate the development of deep reinforcement learning neural network applications:

pytorch[10]: pytorch is a widely used neural network computing framework help us to build neural networks in a dynamic way and support distributed training and GPU, JIT optimizations which help us to provide high quality, serializable, optimized codes.

tensorflow[3]: tensorflow is one of the most famous neural network library and also have many supports to our neural network building process. using both py-

torch and tensorflow is to accelerate the model implementation speed because those two models can be deployed to our server and some of our team members only proficient in tensorflow
numpy : numpy is a fast matrix computing framework and compatible to tensorflow and pytorch, plus it supports serialization and deserialization which enabled high performance transaction between clients and server

4.1.2 Server application architecture

twisted server: Twisted is an event-based framework for internet applications, supporting Python 2.7 and Python 3.5+. we used twisted mainly for client and server communication because it supports asynchronous communication which can reduce the latency between clients and servers during game playing

4.1.3 Distributed training and distributed computing

hadoop[12]: hadoop is a well known distributed computing framework with huge big data ecosystem such as HDFS, HBase and so on. the main purpose of using hadoop is to implement some Map reduce distributed computing applications for the investigation on distributed training and inference.
ray

4.1.4 User Interface and rendering supports

Unity:[7] Unity client is used for GUI in sandbox

5 What we built? Methodology and Implementations

5.1 Unity Client game UI, and map generations

5.1.1 Unity client

Unity client UI is implemented by unity hub and we used the original UI assets to get avoid of rebuilding wheels. here we optimized the network connection setting and changed UI map drastically to support different size of maps with support to much more clients live in the same map, as figure 9, 10 shown below

5.1.2 Map Generations

We are writing several scripts to build map in a systematic way to reduce the effort to draw maps from scratch, at first it will generate forged binary file and then specialized rendering algorithms are used to show different kinds of map. below is the generation scripts

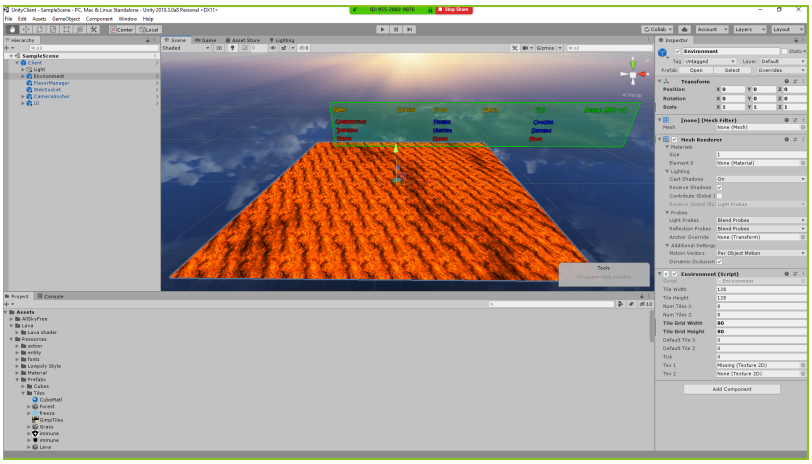


Figure 9: Unity client side Unity programming page, with dashboard and lava in here

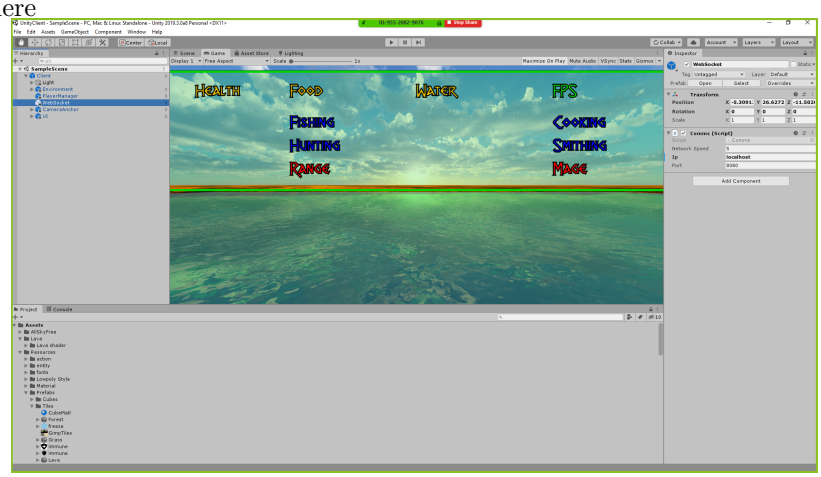


Figure 10: Unity client side with port and game UI settings

```
from pdb import set_trace as T
from opensimplex import OpenSimplex
gen = OpenSimplex()

from forge.blade.lib import enums
from matplotlib import pyplot as plt

import sys

try:
```

```
    from imageio import imread, imsave
except ImportError:
    try:
        from scipy.misc import imread, imsave
    except ImportError:
        print(sys.exc_info())

from shutil import copyfile
from copy import deepcopy
import numpy as np
import os

template_tiled = """<?xml version="1.0" encoding="UTF-8"?>
<map version="1.0" tiledversion="1.1.5" orientation="orthogonal" renderorder="right-down" wi
 <tileset firstgid="0" source="../../tiles.tsx"/>
 <layer name="Tile Layer 1" width="{0}" height="{1}">
  <data encoding="csv">
  {2}
</data>
 </layer>
</map>"""

def saveTiled(dat, path):
   """"Saved a map into into a tiled compatiable file given a save_path,
       width and hieght of the map, and 2D numpy array specifiying enums for the array"""
   height, width = dat.shape
   dat = str(dat.ravel().tolist())
   dat = dat.strip('[').strip(']')
   with open(path + 'map.tmx', "w") as f:
      f.write(template_tiled.format(width, height, dat))
def readTiled(path):
    import xml.etree.ElementTree as ET
    tree  = ET.parse(path)
    root = tree.getroot()
    serial = root[1][0].text
    val = np.fromstring(serial, dtype=int, sep=",")
    print(root.attrib['height'], root.attrib['width'])
    return val.reshape(int(root.attrib['height']), int(root.attrib['width']))/np.max(val)

#Bounds checker
def inBounds(r, c, shape, border=0):
   R, C = shape
   return (
         r > border and
         c > border and
         r < R - border - 1 and
```

```
        c < C - border - 1
        )

def noise(nx, ny):
    # Rescale from -1.0:+1.0 to 0.0:1.0
    return gen.noise2d(nx, ny) / 2.0 + 0.5

def sharp(nx, ny):
    return 2 * (0.5 - abs(0.5 - noise(nx, ny)));

def perlin(nx, ny, octaves, scale=1):
   val = 0
   for mag, freq in octaves:
      val += mag * noise(scale*freq*nx, scale*freq*ny)
   return

def ridge(nx, ny, octaves, scale=1):
   val = []
   for idx, octave in enumerate(octaves):
      mag, freq = octave
      v = mag*sharp(scale*freq*nx, scale*freq*ny)
      if idx > 0:
         v *= sum(val)
      val.append(v)
   return sum(val)

def expParams(n):
   return [(0.5**i, 2**i) for i in range(n)]

def norm(x, X):
   return x/X - 0.5

def grid(X, Y, n=8, scale=1, seed=0):
   terrain = np.zeros((Y, X))
   for y in range(Y):
       for x in range(X):
           octaves = expParams(n)
           nx, ny = norm(x, X), norm(y, Y)
           terrain[x, y] = ridge(seed+nx, seed+ny, octaves, scale)
   return terrain / np.max(terrain)

def textures():
   lookup = {}
   for mat in enums.Material:
      mat = mat.value
      tex = imread(
```

```
            'resource/assets/tiles/' + mat.tex + '.png')
        key = mat.tex
        mat.tex = tex[:, :, :3][::4, ::4]
        lookup[key] = mat
    return lookup

def tile(val, offset):
    if val == 0:
        return 'lava'
    elif val < 0.3:
        return 'forest'
    elif val < 0.675 - offset:
        return 'grass'
    elif val < 0.75 - offset/2:
        return 'stone'
    else:
        return 'water'

def material(terrain, tex, X, Y, border=9):
    terrain = deepcopy(terrain).astype(object)
    for y in range(Y):
        for x in range(X):
            #Center coords
            xRel = x - sz/2
            yRel = y - sz/2
            #mag  = np.sqrt(xRel**2 + yRel**2)
            mag = max(abs(xRel), abs(yRel))

            if not inBounds(y, x, (Y, X), border-1):
                mat = 'lava'
            elif not inBounds(y, x, (Y, X), border):
                mat = 'grass'
            elif np.sqrt(xRel**2 + yRel**2) < 2.5:
                mat = 'water'
            elif mag < 6:
                mat = 'grass'
            else:
                val = float(terrain[y, x])
                norm = mag / (sz / 2)
                offset = norm
                #curve = val + 0.1*mag/sz + 0.9*(mag/sz)**3
                mat = tile(val, 0.325*offset)

            terrain[y, x] = tex[mat]

    return terrain
```

```
def render(mats, path):
    images = [[e.tex for e in l] for l in mats]
    image = np.vstack([np.hstack(e) for e in images])
    imsave(path, image)

def index(mats, path):
    inds = np.array([[e.index+1 for e in l] for l in mats])
    saveTiled(inds, path)

def fractal(terrain, path):
    frac = (256*terrain).astype(np.uint8)
    imsave(path, terrain)

nMaps, sz = 1,128 + 16
#nMaps, sz = 1, 512 + 16
seeds = np.linspace(2**6, 2**32, nMaps)
scale = int(sz / 5)
root = 'resource/maps/'
tex = textures()
np.set_printoptions(threshold=sys.maxsize)
print('Generating {} game maps. This may take a moment'.format(nMaps))
for i, seed in enumerate(seeds):
    print('Generating map ' + str(i))
    path = root + 'procedural/map' + str(i) + '/'
    readTiled(path + "map.tmx")
    try:
        os.mkdir(path)
    except:
        pass
    #terrain = grid(sz, sz, scale=scale, seed=seed)
    terrain = readTiled(root + "v1full.tmx")
    print(terrain)
    tiles = material(terrain - 0.02 , tex, terrain.shape[1] , terrain.shape[0])
    fractal(terrain, path+'fractal.png')
    render(tiles, path+'map.png')
    index(tiles, path)
```

5.2 deep reinforcement learning

5.2.1 Why using Deep Reinforcement Learning

After many trials, we finally decided to use Deep Reinforcement Learning as our final methodology for life simulator. Traditional Reinforcement Learning needs to design complex rules for each agents to maximize their rewards during gaming play, That is not make sense because the intelligence made by beings

come from learning.
There are something like walking or sitting come in nature and everybody can do it.
But something people can just learn from practice such as driving or dance. We can learn a huge variety of things even very complex one such as Go game or StarCraft II [5]
There are many contemporary works made by Google or DeepMind which train their agents to achieve high degree of proficiency in domain governed by simple known rules such as The Game of Go [13].

5.2.2 Define given terms used in our model

s_t : the current state of given agents given time tick t, state contains position information, current food, water, health and so on
o_t : the current observation based on one agent.
a_t : the action made by given agent which have five types, North, West, East, South, Pass
$\pi_\theta(a_t|s_t)$: Policy(fully observed)
τ: the trajectory of given agents which is a sequence of observation, action and reward. $\tau = (o_t, a_t, r_t, \ldots, o_T, a_T, r_T)$
$\mathbf{R}(\tau)$: reward given trajectory which is the sum of each survival one in every time titch with discount γ which by default is 0.99 $\mathbf{R}(\tau) = \sum_t^T \gamma^t r_t$

5.2.3 Training steps

The purpose is to maximize the reward in given trajectory

$$\underbrace{p_\theta\left(\mathbf{s}_1, \mathbf{a}_1, \ldots, s_T, \mathbf{a}_T\right)}_{p_\theta(\tau)} = p\left(\mathbf{s}_1\right) \prod_{t=1}^{T} \pi_\theta\left(\mathbf{a}_t|\mathbf{s}_t\right) p\left(\mathbf{s}_{t+1}|\mathbf{s}_t, \mathbf{a}_t\right) \tag{1}$$

$$\theta^\star = \arg\max_\theta E_{\tau\sim p_\theta(\tau)}\left[\sum_t r\left(\mathbf{s}_t, \mathbf{a}_t\right)\right] \tag{2}$$

Here we want to optimize the learned parameters maximize the expected survival rewards by adopting **policy gradient method**

$$J(\theta) = E_{\tau\sim p_\theta(\tau)}\left[\sum_t r\left(\mathbf{s}_t, \mathbf{a}_t\right)\right] \approx \frac{1}{N}\sum_i \sum_t r\left(\mathbf{s}_{i,t}, \mathbf{a}_{i,t}\right) \tag{3}$$

$$\nabla_\theta J(\theta) = E_{\tau\sim\pi_\theta(\tau)}\left[\left(\sum_{t=1}^{T} \nabla_\theta \log \pi_\theta\left(\mathbf{a}_t|\mathbf{s}_t\right)\right)\left(\sum_{t=1}^{T} r\left(\mathbf{s}_t, \mathbf{a}_t\right)\right)\right] \tag{4}$$

We will update based on epoch

$$\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta) \tag{5}$$

5.2.4 Training Model

Here the model is just a linear model, we use embedding and flatten a agents' reward to a one-dimensional encoded vector and fit it into a linear layer of with trainable parameters.

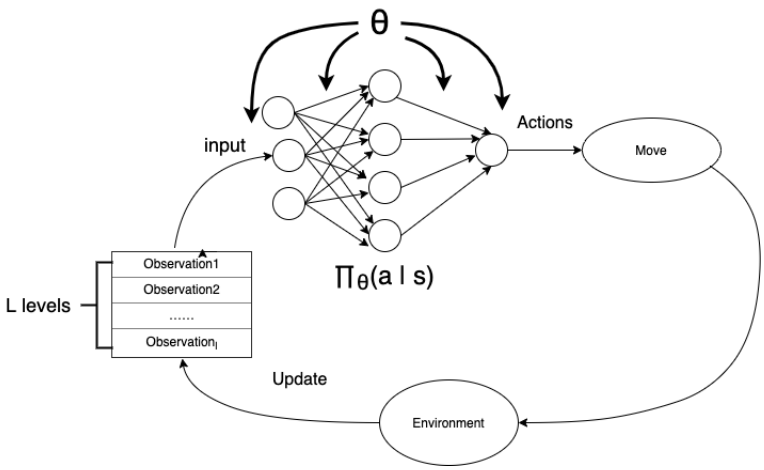


Figure 11: current Net word model

5.3 deep learning layer implementation

Here it is a typical classification scenario, each agents need to make several kinds of decisions based on its trajectory and environment, here our team developed several kinds of neural network architecture layers to help agents fulfill the goal

5.3.1 attention layer[15]

attention layers are going to resolve the the problem that LSTM is judged by the hidden layer which depends more on the last trajectory input which if we adopting attention mechanism, it may lead us to achieve better decision making by adopting global level understanding to the whole trajectory information and pick up the best one for decision/ action making. here is our implementation

```
""" Global attention modules  """
import torch
import torch.nn as nn
import torch.nn.functional as F

from onmt.modules.sparse_activations import sparsemax
from onmt.utils.misc import aeq, sequence_mask
```

```
# This class is mainly used by decoder.py for RNNs but also
# by the CNN / transformer decoder when copy attention is used
# CNN has its own attention mechanism ConvMultiStepAttention
# Transformer has its own MultiHeadedAttention


class GlobalAttention(nn.Module):
    """
    Global attention takes a matrix and a query vector. It
    then computes a parameterized convex combination of the matrix
    based on the input query.

    Constructs a unit mapping a query `q` of size `dim`
    and a source matrix `H` of size `n x dim`, to an output
    of size `dim`.


    .. mermaid::

       graph BT
          A[Query]
          subgraph RNN
            C[H 1]
            D[H 2]
            E[H N]
          end
          F[Attn]
          G[Output]
          A --> F
          C --> F
          D --> F
          E --> F
          C -.-> G
          D -.-> G
          E -.-> G
          F --> G

    All models compute the output as
    :math:`c = sum_{j=1}^{SeqLength} a_j H_j` where
    :math:`a_j` is the softmax of a score function.
    Then then apply a projection layer to [q, c].

    However they
    differ on how they compute the attention score.

    * Luong Attention (dot, general):
```

```
   * dot: :math:`score(H_j,q) = H_j^T q`
   * general: :math:`score(H_j, q) = H_j^T W_a q`


* Bahdanau Attention (mlp):
   * :math:`score(H_j, q) = v_a^T tanh(W_a q + U_a h_j)`


Args:
   dim (int): dimensionality of query and key
   coverage (bool): use coverage term
   attn_type (str): type of attention to use, options [dot,general,mlp]

"""

def __init__(self, dim, coverage=False, attn_type="dot",
             attn_func="softmax"):
    super(GlobalAttention, self).__init__()

    self.dim = dim
    assert attn_type in ["dot", "general", "mlp"], (
        "Please select a valid attention type.")
    self.attn_type = attn_type
    assert attn_func in ["softmax", "sparsemax"], (
        "Please select a valid attention function.")
    self.attn_func = attn_func

    if self.attn_type == "general":
        self.linear_in = nn.Linear(dim, dim, bias=False)
    elif self.attn_type == "mlp":
        self.linear_context = nn.Linear(dim, dim, bias=False)
        self.linear_query = nn.Linear(dim, dim, bias=True)
        self.v = nn.Linear(dim, 1, bias=False)
    # mlp wants it with bias
    out_bias = self.attn_type == "mlp"
    self.linear_out = nn.Linear(dim * 2, dim, bias=out_bias)

    if coverage:
        self.linear_cover = nn.Linear(1, dim, bias=False)

def score(self, h_t, h_s):
    """
    Args:
      h_t (`FloatTensor`): sequence of queries `[batch x tgt_len x dim]`
      h_s (`FloatTensor`): sequence of sources `[batch x src_len x dim]`
```

```
        Returns:
          :obj:`FloatTensor`:
           raw attention scores (unnormalized) for each src index
          `[batch x tgt_len x src_len]`

        """

        # Check input sizes
        src_batch, src_len, src_dim = h_s.size()
        tgt_batch, tgt_len, tgt_dim = h_t.size()
        aeq(src_batch, tgt_batch)
        aeq(src_dim, tgt_dim)
        aeq(self.dim, src_dim)

        if self.attn_type in ["general", "dot"]:
            if self.attn_type == "general":
                h_t_ = h_t.view(tgt_batch * tgt_len, tgt_dim)
                h_t_ = self.linear_in(h_t_)
                h_t = h_t_.view(tgt_batch, tgt_len, tgt_dim)
            h_s_ = h_s.transpose(1, 2)
            # (batch, t_len, d) x (batch, d, s_len) --> (batch, t_len, s_len)
            return torch.bmm(h_t, h_s_)
        else:
            dim = self.dim
            wq = self.linear_query(h_t.view(-1, dim))
            wq = wq.view(tgt_batch, tgt_len, 1, dim)
            wq = wq.expand(tgt_batch, tgt_len, src_len, dim)

            uh = self.linear_context(h_s.contiguous().view(-1, dim))
            uh = uh.view(src_batch, 1, src_len, dim)
            uh = uh.expand(src_batch, tgt_len, src_len, dim)

            # (batch, t_len, s_len, d)
            wquh = torch.tanh(wq + uh)

            return self.v(wquh.view(-1, dim)).view(tgt_batch, tgt_len, src_len)

    def forward(self, source, memory_bank, memory_lengths=None, coverage=None):
        """

        Args:
          source (`FloatTensor`): query vectors `[batch x tgt_len x dim]`
          memory_bank (`FloatTensor`): source vectors `[batch x src_len x dim]`
          memory_lengths (`LongTensor`): the source context lengths `[batch]`
          coverage (`FloatTensor`): None (not supported yet)
```

```
Returns:
  (`FloatTensor`, `FloatTensor`):

  * Computed vector `[tgt_len x batch x dim]`
  * Attention distribtutions for each query
     `[tgt_len x batch x src_len]`
"""

# one step input
if source.dim() == 2:
    one_step = True
    source = source.unsqueeze(1)
else:
    one_step = False

batch, source_l, dim = memory_bank.size()
batch_, target_l, dim_ = source.size()
aeq(batch, batch_)
aeq(dim, dim_)
aeq(self.dim, dim)
if coverage is not None:
    batch_, source_l_ = coverage.size()
    aeq(batch, batch_)
    aeq(source_l, source_l_)

if coverage is not None:
    cover = coverage.view(-1).unsqueeze(1)
    memory_bank += self.linear_cover(cover).view_as(memory_bank)
    memory_bank = torch.tanh(memory_bank)

# compute attention scores, as in Luong et al.
align = self.score(source, memory_bank)

if memory_lengths is not None:
    mask = sequence_mask(memory_lengths, max_len=align.size(-1))
    mask = mask.unsqueeze(1)  # Make it broadcastable.
    align.masked_fill_(1 - mask, -float('inf'))

# Softmax or sparsemax to normalize attention weights
if self.attn_func == "softmax":
    align_vectors = F.softmax(align.view(batch*target_l, source_l), -1)
else:
    align_vectors = sparsemax(align.view(batch*target_l, source_l), -1)
align_vectors = align_vectors.view(batch, target_l, source_l)

# each context vector c_t is the weighted average
```

```
        # over all the source hidden states
        c = torch.bmm(align_vectors, memory_bank)

        # concatenate
        concat_c = torch.cat([c, source], 2).view(batch*target_l, dim*2)
        attn_h = self.linear_out(concat_c).view(batch, target_l, dim)
        if self.attn_type in ["general", "dot"]:
            attn_h = torch.tanh(attn_h)

        if one_step:
            attn_h = attn_h.squeeze(1)
            align_vectors = align_vectors.squeeze(1)

            # Check output sizes
            batch_, dim_ = attn_h.size()
            aeq(batch, batch_)
            aeq(dim, dim_)
            batch_, source_l_ = align_vectors.size()
            aeq(batch, batch_)
            aeq(source_l, source_l_)

        else:
            attn_h = attn_h.transpose(0, 1).contiguous()
            align_vectors = align_vectors.transpose(0, 1).contiguous()
            # Check output sizes
            target_l_, batch_, dim_ = attn_h.size()
            aeq(target_l, target_l_)
            aeq(batch, batch_)
            aeq(dim, dim_)
            target_l_, batch_, source_l_ = align_vectors.size()
            aeq(target_l, target_l_)
            aeq(batch, batch_)
            aeq(source_l, source_l_)

        return attn_h, align_vectors
```

5.3.2 multi-head attention layer[4]

multi-head attention improved the robustness of attention layer by adopting several heads instead of one head into consideration, here is the keras version of our code, integration needs much more time

```
from keras.engine.topology import Layer
import tensorflow as tf
import keras
from keras import backend as K
```

```
class MultiHeadPixelAttention(Layer):

    def __init__(self,kernel_range=[3,3],numHead =  2, shift=1,ff_kernel=[3,3],useBN=False,
        self.kernel_range=kernel_range # should be a list
        self.shift=shift
        self.numHead = numHead
        self.ff_kernel=ff_kernel
        self.useBN=useBN
        self.useRes=useRes
        super(MultiHeadPixelAttention,self).__init__(**kwargs)

    def build(self,input_shape):
        D=input_shape[-1]
        n_p=self.kernel_range[0]*self.kernel_range[1]
        self.K_P=self.add_weight(name='K_P',shape=(1,1,D,D*n_p),
                                 initializer='glorot_uniform',
                                 trainable=True)
        self.V_P=self.add_weight(name='V_P',shape=(1,1,D,D*n_p),
                                 initializer='glorot_uniform',
                                 trainable=True)
        self.Q_P=self.add_weight(name='Q_P',shape=(1,1,D,D),
                                 initializer='glorot_uniform',
                                 trainable=True)


        self.ff1_kernel=self.add_weight(name='ff1_kernel',
                                        shape=(1,1,D,D),
                                        initializer='glorot_uniform',trainable=True)
        self.ff1_bais=self.add_weight(name='ff1_bias',
                                        shape=(D,),initializer='glorot_uniform',trainable=Tr
        self.ff2_kernel=self.add_weight(name='ff2_kernel',
                                        shape=(3,3,D,2*D),
                                        initializer='glorot_uniform',trainable=True)
        self.ff2_bais=self.add_weight(name='ff2_bias',
                                        shape=(2*D,),initializer='glorot_uniform',trainable=

        self.ff3_kernel=self.add_weight(name='ff3_kernel',
                                        shape=(3,3,2*D,D),
                                        initializer='glorot_uniform',trainable=True)
        self.ff3_bais=self.add_weight(name='ff3_bias',
                                        shape=(D,),initializer='glorot_uniform',trainable=Tr

        super( MultiHeadPixelAttention,self).build(input_shape)

    def call(self,x):
        h_half=self.kernel_range[0]//2
```

```
w_half=self.kernel_range[1]//2
_,_,_,D=x.shape
s=K.shape(x)
x_k=tf.nn.conv2d(input=x,filter=self.K_P,padding="SAME",strides=[1,1,1,1])
x_v=tf.nn.conv2d(input=x,filter=self.V_P,padding="SAME",strides=[1,1,1,1])
x_q=tf.nn.conv2d(input=x,filter=self.Q_P,padding="SAME",strides=[1,1,1,1])
paddings=tf.constant([[0,0],[h_half*self.shift,h_half*self.shift],[w_half*self.shift
x_k=tf.pad(x_k,paddings,"CONSTANT")
x_v=tf.pad(x_v,paddings,"CONSTANT")
mask_x=tf.ones(shape=(s[0],s[1],s[2],self.numHead))
mask_pad=tf.pad(mask_x,paddings,"CONSTANT")

k_ls=list()
v_ls=list()
masks=list()

c_x,c_y=h_half*self.shift,w_half*self.shift
layer=0
for i in range(-h_half,h_half+1):
    for j in range(-w_half,w_half+1):
        k_t=x_k[:,c_x+i*self.shift:c_x+i*self.shift+s[1],c_y+j*self.shift:c_y+j*self
        k_ls.append(k_t)

        v_t=x_v[:,c_x+i*self.shift:c_x+i*self.shift+s[1],c_y+j*self.shift:c_y+j*self
        v_ls.append(v_t)

        _m=mask_pad[:,c_x+i*self.shift:c_x+i*self.shift+s[1],c_y+j*self.shift:c_y+j*
        masks.append(_m)
        layer+=1
m_stack=tf.stack(masks,axis=3,name="mask") # B H M 9 4
m_stack = tf.transpose(m_stack, perm=[0,1,2,4,3]) # B H M 4 9
m_vec=tf.reshape(m_stack,shape=[self.numHead*s[0]*s[1]*s[2],self.kernel_range[0]*self

#m_list = [m_vec]*self.numHead
#m_stack = tf.stack(m_list, axis=1, name="mask1")
#m_vec=tf.reshape(m_stack,shape=[self.numHead*s[0]*s[1]*s[2],self.kernel_range[0]*s


k_stack=tf.stack(k_ls,axis=3,name="k_stack") # B H M 9, 32
v_stack=tf.stack(v_ls,axis=3,name="v_stack") # B H M 9 32
headDim = int(D//self.numHead)
multiHeadK = tf.reshape(k_stack, (s[0],s[1],s[2], self.kernel_range[0]*self.kernel_r
multiHeadK = tf.transpose(multiHeadK, perm=[0,1,2,4,3,5]) # B * H * W * numHead * 9
multiHeadV = tf.reshape(v_stack ,(s[0],s[1],s[2], self.kernel_range[0]*self.kernel_r
multiHeadV = tf.transpose(multiHeadV, perm=[0,1,2,4,3,5]) # B * H * W * numHead * 9
multiHeadQ = tf.reshape(x_q, (s[0], s[1], s[2], 1, self.numHead, headDim))
```

```
        multiHeadQ = tf.transpose(multiHeadQ, perm=[0,1,2,4,3,5]) # B * H * W * numHead * 1
        k=tf.reshape(multiHeadK,shape=[self.numHead*s[0]*s[1]*s[2],self.kernel_range[0]*self
        v=tf.reshape(multiHeadV,shape=[self.numHead*s[0]*s[1]*s[2],self.kernel_range[0]*self
        q=tf.reshape(multiHeadQ,shape=[self.numHead*s[0]*s[1]*s[2],1,headDim])
        alpha=tf.nn.softmax(tf.matmul(k,q,transpose_b=True)*m_vec/3,axis=1)
        __res=tf.matmul(alpha,v,transpose_a=True)
        ___res = tf.reshape(__res, shape=[s[0],s[1],s[2], self.numHead, headDim, 1])
        _res=tf.reshape(___res,shape=[s[0],s[1],s[2],D])
        if self.useRes:
            t=x+_res
        else:
            t=_res
        if self.useBN:
            t=keras.layers.BatchNormalization(axis=-1)(t)
        _t=t
        t=tf.nn.relu(tf.nn.conv2d(input=t,filter=self.ff1_kernel,padding='SAME',strides=[1,1
        t=tf.nn.relu(tf.nn.conv2d(input=t,filter=self.ff2_kernel,padding='SAME',strides=[1,1
        t=tf.nn.relu(tf.nn.conv2d(input=t,filter=self.ff3_kernel,padding='SAME',strides=[1,1
        if self.useRes:
            t=_t+t
        if self.useBN:
            res=keras.layers.BatchNormalization(axis=-1)(t)
        else:
            res=t
        return res



    def compute_output_shape(self,input_shape):
        return input_shape
```

5.3.3 transformer[15]

neural machine translation may be applicable to our neural network because trajectory information may be viewed as a foreign language sentence and if we can translate them into one action, it may become a valid action, here we have adopted transformer model because of its success in google translation competition to realize our design

```
# encoder code
#!/usr/bin/env python
# coding: utf-8

# In[3]:
```

```
from MyLayers import *
import json


# In[4]:


class Encoder:
    setting=None
    n_heads=None
    n_encoder=None
    useBN=None
    ksize=None
    def __init__(self,config_fileName):
        json_str=None
        with open(config_fileName) as obj:
            json_str=obj.read()
        self.setting=json.loads(json_str)["Encoder_stack"]
        self.n_heads=self.setting["n_heads"]
        self.n_encoder=self.setting["n_encoder"]
        self.useBN=self.setting["useBN"] # not use
        self.ksize=self.setting["ksize"]

    def one_encoder(self,inpt,ind):
        selfAtt=Multi_SelfAttention2D(qkv=inpt,ksize=self.ksize,n_heads=self.n_heads,name="e
        norm1=Add_Normalize(selfAtt,inpt,name="encoder"+str(ind)+"_step2")
        ff=Feed_Forward(hidden_layer_dims=[64,32],name="encoder"+str(ind)+"_step3")(norm1)
        norm2=Add_Normalize(ff,norm1,name="encoder"+str(ind)+"_step4")
        return norm2

    def encoders(self,x):
        for i in range(self.n_encoder):
            x=self.one_encoder(x,ind=i)
        return x


# In[6]:


if __name__=="__main__":
    t=Encoder("config.json")

#decoder code
#!/usr/bin/env python
# coding: utf-8
```

```
from MyLayers import *
import json

class Decoder:
    setting=None
    n_heads=None
    n_decoder=None
    useBN=None
    ksize=None
    def __init__(self,config_fileName):
        json_str=None
        with open(config_fileName) as obj:
            json_str=obj.read()
        self.setting=json.loads(json_str)["Decoder_stack"]
        self.n_heads=self.setting["n_heads"]
        self.n_decoder=self.setting["n_decoder"]
        self.useBN=self.setting["useBN"] # not use
        self.ksize=self.setting["ksize"]

    def one_decoder(self,q,kv,ind):
        selfAtt=Multi_SelfAttention2D(qkv=q,ksize=self.ksize,n_heads=self.n_heads,name="deco
        norm1=Add_Normalize(selfAtt,q,name="decoder"+str(ind)+"_step2")
        att=Multi_Attention2D(n_heads=self.n_heads,ksize=self.ksize,name="decoder"+str(ind)+
        norm2=Add_Normalize(att,norm1,name="decoder"+str(ind)+"_step4")
        ff=Feed_Forward(hidden_layer_dims=[64,32],name="decoder"+str(ind)+"_step5")(norm2)
        norm3=Add_Normalize(ff,norm2,name="decoder"+str(ind)+"_step6")
        return norm3

    def decoders(self,x,kv):
        for i in range(self.n_decoder):
            x=self.one_decoder(x,kv,ind=i)
        return x


# In[3]:


if __name__=="__main__":
    t=Decoder("config.json")
```

5.4 communication network architecture design

5.4.1 Network Architecture

Network here is simple and easy to configure, we used Twisted Network Library to communicate packet information between client and server, which supports

asynchronous communications here is the source code

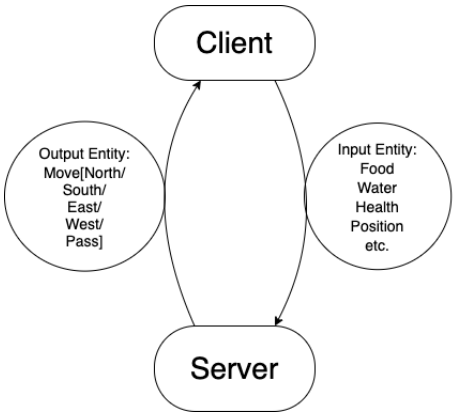


Figure 12: current Net word model

```
from pdb import set_trace as T
import numpy as np

from signal import signal, SIGINT
import sys, os, json, pickle, time
import ray

from twisted.internet import reactor
from twisted.internet.task import LoopingCall
from twisted.python import log
from twisted.web.server import Site
from twisted.web.static import File

from autobahn.twisted.websocket import WebSocketServerFactory, \
    WebSocketServerProtocol
from autobahn.twisted.resource import WebSocketResource

def sign(x):
    return int(np.sign(x))

def move(orig, targ):
    ro, co = orig
    rt, ct = targ
    dr = rt - ro
    dc = ct - co
    if abs(dr) > abs(dc):
        return ro + sign(dr), co
    elif abs(dc) > abs(dr):
```

```
            return ro, co + sign(dc)
        else:
            return ro + sign(dr), co + sign(dc)

class GodswordServerProtocol(WebSocketServerProtocol):
    def __init__(self):
        super().__init__()
        print("Created a server")
        self.frame = 0
        self.packet = {}

    def onOpen(self):
        print("Opened connection to server")

    def onClose(self, wasClean, code=None, reason=None):
        print('Connection closed')

    def connectionMade(self):
        super().connectionMade()
        self.factory.clientConnectionMade(self)

    def connectionLost(self, reason):
        super().connectionLost(reason)
        self.factory.clientConnectionLost(self)

    #Not used without player interaction
    def onMessage(self, packet, isBinary):
        print("Message", packet)

    def onConnect(self, request):
        print("WebSocket connection request: {}".format(request))
        realm = self.factory.realm
        self.realm = realm
        self.frame += 1

        data = self.serverPacket()
        sz = data['environment'].shape[0]

        self.vals = None
        if data['values'] is not None:
           self.vals = self.visVals(data['values'], sz)

        self.sendUpdate()

    def serverPacket(self):
        data = ray.get(self.realm).clientData()
```

```
        return data

    def sendUpdate(self):
        ent = {}
        data = self.serverPacket()
        entities = data['entities']
        environment = data['environment']
        self.packet['ent'] = entities

        gameMap = environment.np().tolist()
        self.packet['map'] = gameMap

        tiles = []
        for tileList in environment.tiles:
           tl = []
           for tile in tileList:
              tl.append(tile.counts.tolist())
           tiles.append(tl)
        self.packet['counts'] = tiles

        self.packet['values'] = self.vals
        if self.vals is not None:
           self.packet['values'] = self.vals.tolist()

        packet = json.dumps(self.packet).encode('utf8')
        self.sendMessage(packet, False)

    #Todo: would be nicer to move this into the javascript,
    #But it would possibly have to go straight into the shader
    def visVals(self, vals, sz):
      ary = np.zeros((sz, sz, 3))
      vMean = np.mean([e[1] for e in vals])
      vStd  = np.std([e[1] for e in vals])
      nStd, nTol = 4.0, 0.5
      grayVal = int(255 / nStd * nTol)
      for v in vals:
         pos, mat = v
         r, c = pos
         mat = (mat - vMean) / vStd
         color = np.clip(mat, -nStd, nStd)
         color = int(color * 255.0 / nStd)
         if color > 0:
             color = (0, color, 128)
         else:
             color = (-color, 0, 128)
         ary[r, c] = color
```

```
      return ary.astype(np.uint8)

class WSServerFactory(WebSocketServerFactory):
    def __init__(self, ip, realm, step):
        super().__init__(ip)
        self.realm, self.step = realm, step
        self.clients = []

        self.tickRate = 0.6
        self.tick = 0

        lc = LoopingCall(self.announce)
        lc.start(self.tickRate)

    def announce(self):
        self.tick += 1
        uptime = np.round(self.tickRate*self.tick, 1)
        print('Uptime: ', uptime, ', Tick: ', self.tick)

        if self.tick == 5:
           pass
           #time.sleep(20)
        self.step()
        for client in self.clients:
            client.sendUpdate()

    def clientConnectionMade(self, client):
        self.clients.append(client)

    def clientConnectionLost(self, client):
        self.clients.remove(client)

class Application:
   def __init__(self, realm, step):
      signal(SIGINT, self.kill)
      self.realm = realm
      log.startLogging(sys.stdout)
      port = 8080

      factory = WSServerFactory(u'ws://localhost:' + str(port), realm, step)
      factory.protocol = GodswordServerProtocol
      resource         = WebSocketResource(factory)

      # We server static files under "/" and our WebSocket
      # server under "/ws" (note that Twisted uses bytes
      # for URIs) under one Twisted Web Site
```

```
    root = File(".")
    root.putChild(b"ws", resource)
    site = Site(root)

    reactor.listenTCP(port, site)
    reactor.run()

def kill(*args):
    print("Killed by user")
    reactor.stop()
    os._exit(0)
```

5.5 scaling network architecture

5.5.1 MapReduce paradigms[6]

Currently our machine's architecture is a simple single machine and server. We have a single process and cannot utilize the parallelization of agents to deploy the massive agents on different machines, so we want to change the architecture in order to increase the number of agents we can train simultaneously.

First it reminds us Map Reduce, which is a common big data analytics technique. The basic method includes two steps:
(1) Map: distribute massive data to different machines for the data process.
(2) Reduce: summarize all the distributed data from different machines to get the final result.

This gives us the basic idea of how to train in distributed systems: we can distribute our data to different worker processes or machines and train agents on different machines and update the parameters for every worker agent.

Here a large number of records are broken into segments and **Map** will extracts something of interesting from each segment, **Group** operation sorts the intermediate results from each segment (sometimes called shuffle), **Reduce** will aggregates intermediate results

5.5.2 MapReduce demo code

Here is one of the sample code implementing the inverted index algorithm for search engine write by one of our team member, it will map the word to their corresponding documents and then shuffle them and sort them based on the key, then scheduler can allocate reducer to get the inverted index of every word

```
import java.io.IOException;
import java.util.HashMap;
import java.util.Iterator;
import java.util.Map;
import java.util.StringTokenizer;
```

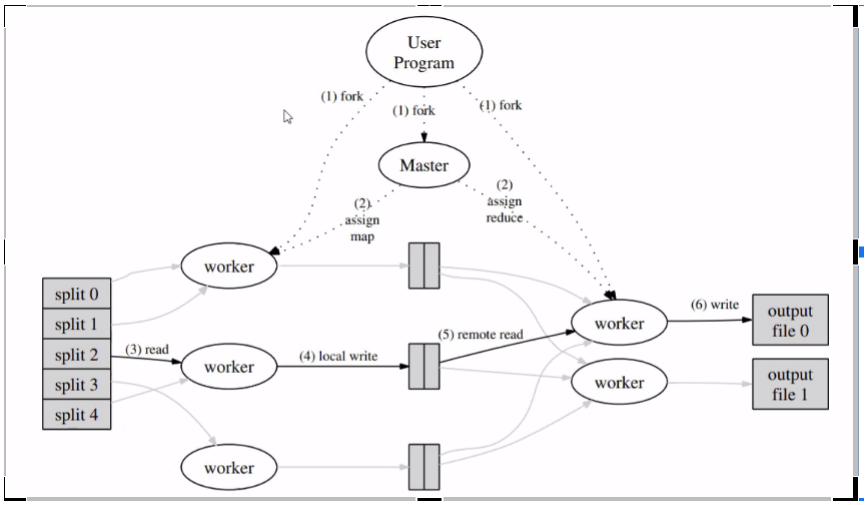


Figure 13: mapreduce working pipeline

```
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class InvertedIndexJob {

 public static class TokenizerMapper
 extends Mapper < LongWritable, Text, Text, Text > {


  private Text word = new Text();
  Text docid = new Text();

  public void map(LongWritable key, Text value, Context context) throws IOException,
  InterruptedException {

   String line = value.toString();
   StringTokenizer tokens = new StringTokenizer(line);
   String docidstr = tokens.nextToken();
   docid = new Text(docidstr);
```

```
  while (tokens.hasMoreTokens()) {
   word.set(tokens.nextToken());
   context.write(word, docid);
  }
 }
}

public static class IntSumReducer
extends Reducer < Text, Text, Text, Text > {



 public void reduce(Text key, Iterable < Text > values,
  Context context
 ) throws IOException,
 InterruptedException {

  HashMap < String, Integer > hm = new HashMap < String, Integer > ();
  Iterator < Text > itr = values.iterator();
  int freq = 0;
  String val;
  while (itr.hasNext()) {
   val = itr.next().toString();
   if (hm.containsKey(val)) {
    freq = (hm.get(val));
    freq += 1;
    hm.put(val, freq);

   } else {
    hm.put(val, 1);
   }

  }
  StringBuffer sb = new StringBuffer("");
  for (Map.Entry < String, Integer > map: hm.entrySet()) {
   sb.append(map.getKey() + ":" + map.getValue() + "\t");

  }
  context.write(key, new Text(sb.toString()));


 }
}

public static void main(String[] args) throws Exception {
 Configuration conf = new Configuration();
```

```
  Job job = Job.getInstance(conf, "inverted index");
  job.setJarByClass(InvertedIndexJob.class);
  job.setMapperClass(TokenizerMapper.class);
  //job.setCombinerClass(IntSumReducer.class); //try removing
  job.setReducerClass(IntSumReducer.class);
  job.setOutputKeyClass(Text.class);
  job.setOutputValueClass(Text.class);
  FileInputFormat.addInputPath(job, new Path(args[0]));
  FileOutputFormat.setOutputPath(job, new Path(args[1]));
  System.exit(job.waitForCompletion(true) ? 0 : 1);
 }
}
```

5.5.3 How could it affect our finally distributed training design

Our purpose is to expand model to realize the dream that massive amount of agents training and running in a single platform here our training architecture is simply using a single process to train the model and the speed is extremelt slow, because it cannot utilize the parallelization of agents to deploy the massive agents to different machines so we want to change the architecture in order to increase the number of agents we can train simultaneously

5.6 scaling training with parameter server

The initial training model with a single-layer linear neural network took us several days to train our data in CPU xeon, thus we consider methods to optimize our algorithm by distribute training to different process and utilize the computing resources completely

5.6.1 Distributed training with parameter server

The main idea of distributed training is to involve two or more machines collaborating on a single task by communicating over a network. There are a few basic patterns of communication that are used by distributed programs:
(1) Push: Machine A sends some data to machine B.
(2) Pull: Machine B requests some data from machine A.
(3) Reduce: Compute some reduction of data on multiple machines and materialize the result on one machine B.
(4) All-reduce: Compute some reduction of data on multiple machines and materialize the result on all those machines.
(5) Wait: One machine pauses its computation and waits for data to be received from another machine.
(6) Broadcast: Machine A sends some data to many machines.[1]

Communicating over the network can have high latency, so an important principle of parallel computing is to overlap computation and communication.

For the best performance, we want our workers to still be doing useful work while communication is going on (rather than having to stop and wait for the communication to finish).

The first solution comes to us is that we can use all-reduce to run learning algorithm SGD in a distributed fashion. To be simple, the idea is to just parallelize the minibatch. We start with an identical copy of the parameter wt on each worker. If the SGD update step is

$$\omega_{t+1} = \omega_t - \alpha_t \frac{1}{B} \sum_{b=1}^{B} \bigtriangledown f_{i_{b,t}}(\omega_t) \tag{6}$$

and there are M worker machines such that

$$B = M \times B' \tag{7}$$

, then we can re-write this update step as

$$\omega_{t+1} = \omega_t - \alpha_t \frac{1}{M} \sum_{m=1}^{M} \frac{1}{B'} \sum_{b=1}^{B'} \bigtriangledown f_{i_{m,b,t}}(\omega_t) \tag{8}$$

Here M stands for the number of workers, B'stands for the batch number of every worker. We assign the computation of the sum when m = 1 to worker 1, the computation of the sum when m = 2 to worker 2, etc. After all the gradients are computed, we can perform the outer sum with an all-reduce operation, after which the full sum

$$\sum_{m=1}^{M} \frac{1}{B'} \sum_{b=1}^{B'} \bigtriangledown f_{i_{m,b,t}}(\omega_t) \tag{9}$$

will be present on all the worker machines. From here, each worker can individually compute the new value of w_{t+1} and update its own parameter vector. After this update, the values of the parameters on each worker will be the same.

Although it's easy to reason about and to implement, the workers are idle when the communication for all-reduce is happening – we are not overlapping computation and communication. Thus, we need to do more optimization for the algorithm.

Using a parameter server helps us to solve this problem. In order to get w_t, which means the parameter value after t timesteps of algorithm SGD, we used to use SGD with all-reduce to make the value of the parameters are identical on all workers at any given time. However, if we use a single machine, the parameter server, to maintain the up-to-date parameter value which is computed by all worker machines, the overlapping of computation and communication in every worker will be reduced, which increases our computing efficiency.

Periodically, the parameter server get the parameters that is computed and pushed by workers, then broadcasts its updated parameters to all the other worker machines, so that they can use the updated parameters to compute gradients.

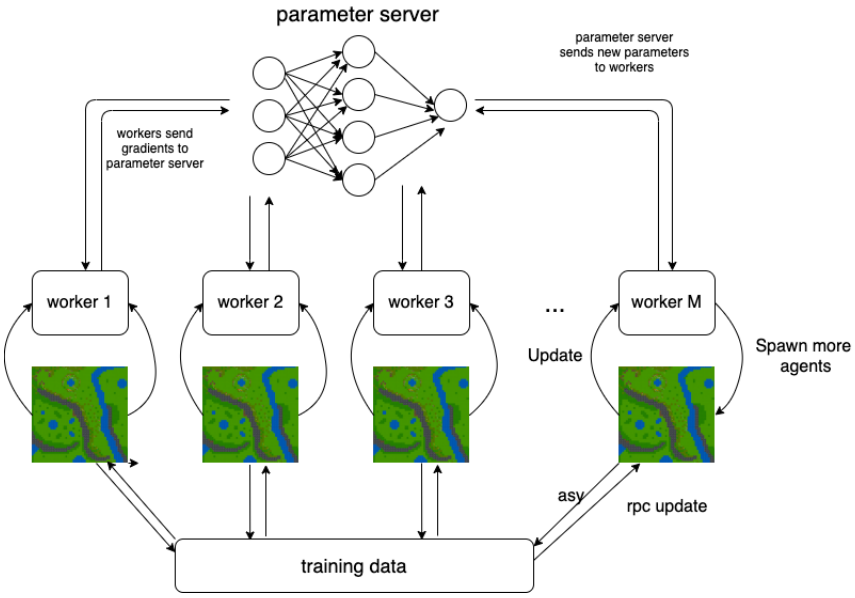


Figure 14: The parameter server architecture

We use the parameter server model to realize asynchronous distributed training, the algorithm is as follows:

---

input: loss function examples f1, f2,..., number of worker machines M, per-machine minibatch size B'
input: learning rate a, initial param w_0, number of iterations per worker T
for m = 1 **to** M **run in parallel on machine** m
  load $w_{m,0}$ from the parameter server
  for t = 1 **to** T **do**
    select a minibatch $i_{m,1,t}$,$i_{m,2,t}$,$i_{m,B',t}$ of size B'
    compute $g_{m,t} \leftarrow \frac{1}{B'} \sum_{b=1}^{B} \bigtriangledown f_{i_{m,b,t}}(\omega_{m,t-1})$
    push gradient $g_{m,t}$ to the parameter server
    receive new model $w_{m,t}$ from the parameter server
  end for
end parallel for
run in parallel on param server
  initialize model $w \leftarrow w_0$
  loop

reveive a gradient g from a worker
update model $w \leftarrow w - ag$
send w back to the worker
end loop
end run on param server
return w_t (from any machine)

From here, each worker can individually compute the new value of w_{t+1} and update its own parameter vector; after this update, the values of the parameters on each worker will be the same.

5.6.2 Implement parameter server with Ray

Now we decide to implement the parameter server model using a framework from UC Berkeley's riselab, Ray, which is a high-performance distributed execution framework targeted at large-scale machine learning and reinforcement learning applications.

Ray takes the existing concepts of functions and classes and translates them to the distributed setting as tasks and actors. A task represents the execution of a remote function on a stateless worker. An actor represents a stateful computation[11].
The basic idea in a Ray job is a dynamic task graph. A computation graph represents a neural network and is executed many times in a single application, in Ray, the task graph represents the entire application and is only executed a single time. It is constructed dynamically as an application runs, and the execution of one task may trigger the creation of more tasks.[2]

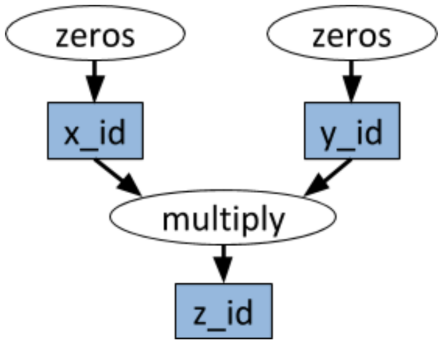


Figure 15: An example computation graph of Ray. The white ovals refer to tasks, and the blue boxes refer to objects. Arrows indicate that a task depends on an object or that a task creates an object.

Arbitrary Python functions can be executed as tasks, and they can depend arbitrarily on the outputs of other tasks. This is illustrated in the example below.

```
# Define two remote functions. Invocations of these functions create tasks
# that are executed remotely.

@ray.remote
def multiply(x, y):
    return np.dot(x, y)

@ray.remote
def zeros(size):
    return np.zeros(size)

# Start two tasks in parallel. These immediately return futures and the
# tasks are executed in the background.
x_id = zeros.remote((100, 100))
y_id = zeros.remote((100, 100))

# Start a third task. This will not be scheduled until the first two
# tasks have completed.
z_id = multiply.remote(x_id, y_id)

# Get the result. This will block until the third task completes.
z = ray.get(z_id)
```

One thing that can't be done with just the remote functions and tasks described above is to have multiple tasks operating on the same shared mutable

state. This comes up in multiple contexts in machine learning where the shared state may be the weights of a neural network. Ray uses an actor abstraction to encapsulate mutable state shared between multiple tasks. Thus, an actor can be used in very flexible ways, like, be used for distributed training as with a parameter server.

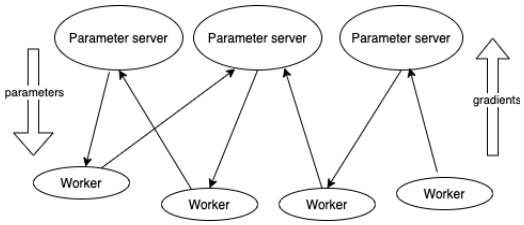


Figure 16: Multiple parameter server actors performing distributed training with multiple worker processes.

A parameter server can be implemented as a Ray actor as follows:

```
@ray.remote
class ParameterServer(object):
    def __init__(self, keys, values):
        # These values will be mutated, so we must create a local copy.
        values = [value.copy() for value in values]
        self.parameters = dict(zip(keys, values))

    def get(self, keys):
        return [self.parameters[key] for key in keys]

    def update(self, keys, values):
        # This update function adds to the existing values, but the update
        # function can be defined arbitrarily.
        for key, value in zip(keys, values):
            self.parameters[key] += value
```

To create four long-running workers that continuously retrieve and update the parameters, do the following:

```
@ray.remote
def worker_task(parameter_server):
    while True:
        keys = ['key1', 'key2', 'key3']
        # Get the latest parameters.
        values = ray.get(parameter_server.get.remote(keys))
        # Compute some parameter updates.
        updates = time.sleep(1)
```

```
        # Update the parameters.
        parameter_server.update.remote(keys, updates)

# Start 4 long-running tasks.
for _ in range(4):
    worker_task.remote(parameter_server)
```

With the Ray framework, we implement the parameter server pythonically in only a few lines of code, also, it allows us to scale our algorithm that runs on a laptop into a la high-performance distributed application that runs efficiently on a cluster with lightweight APIs.

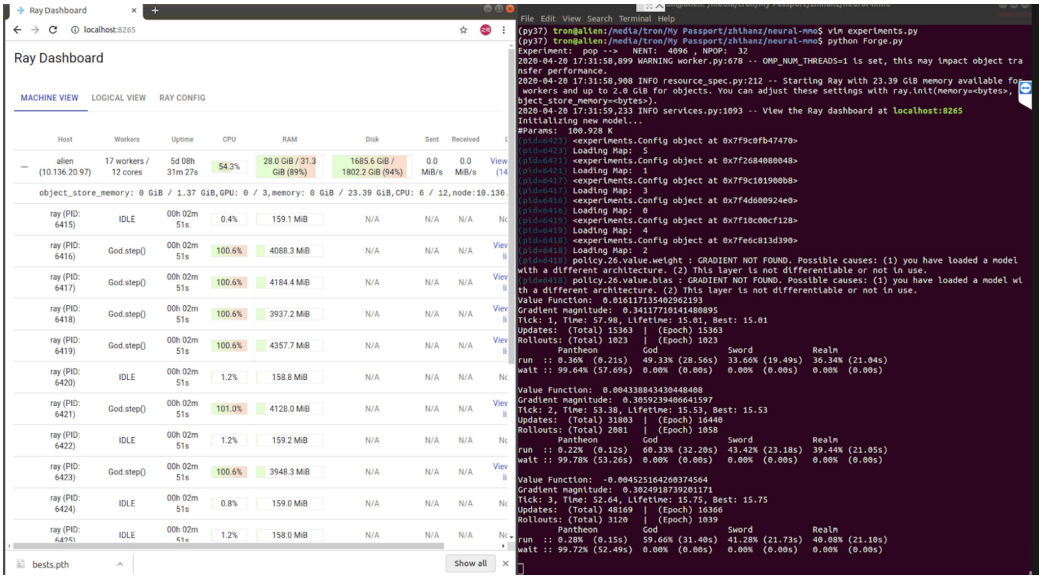


Figure 17: A training sample using ray

6 Training Result and Analysis

Trained on a 12 cores machine, we got the training curve.

The baseline model gets to on average more than 28 lifetime after training for several days on 12 cores. Agents do learnt something after training, like, as a sanity check, agents typically learn not to run into lava first, as indicated by the steep initial learning curve slope.

We have presented a neural MMO as a platform for multi-agent learning. Our environment supports a large number of concurrent agents, inbuilt map randomization, and detailed foraging and combat systems. The included baseline

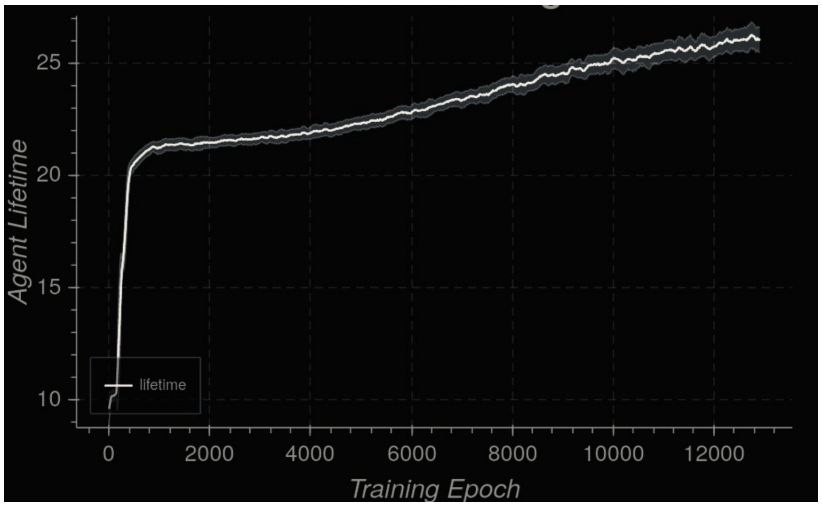


Figure 18: The training curve

experiments demonstrate our platform's capacity for research purposes. We find that population size magnifies exploration in our setting. It is our hope that our environment provides an effective venue for multi-agent experiments, including studies of niche formation, emergent cooperation, and co-evolution.

7 Limitation, Conclusion and Future Work

After designed the neural network and optimized the training structure, our project resolves several limitations along the development process based the environments. But there is a remaining problem for the project.

Given to the limited hardware capacity of ours, our current model's FPS drop drastically even lower than 9 if the map size become 500*500. So the map size rendering optimization can be the most rewarding future work. To achieve this goal, advanced disk capacity is required. In that case, more agents can be trained simultaneously.

Apart from the larger map rendering concern, there are more future work can be done.

Based on the current restricted map, we can add more terrain setups and explore of environment complexity on population behavior like will people are able to choose to live near river and generate collaboration instead of competition.

And after facing the hardware limitation and the remote computing restric-

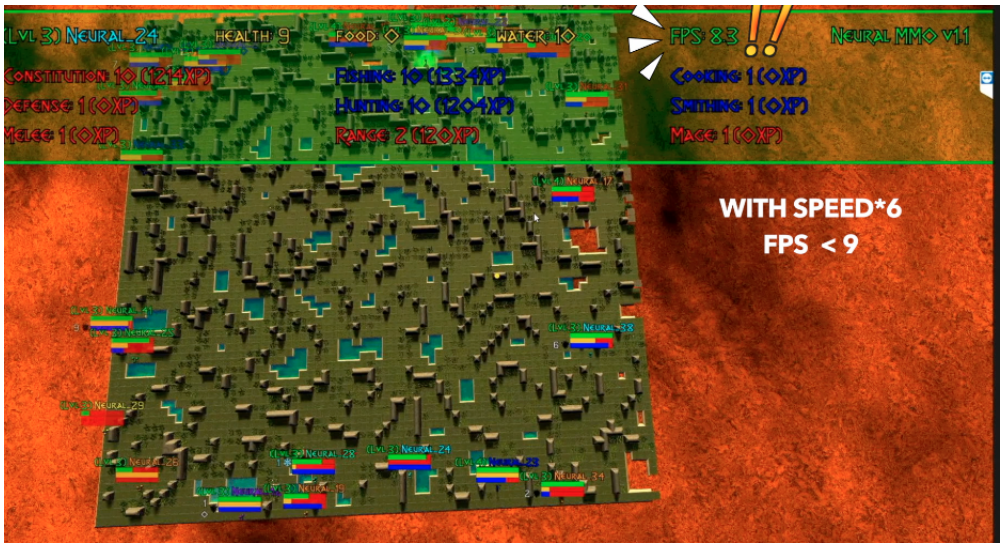


Figure 19: Advanced terrain map setup

tion under current COVID-19 situation. We move our concern to the cloud computing. Whether we can deploy AI training to cloud and realize cloud native AI training, deployment, visualization pipeline can be a interesting topic to try out.

Figure 20: manual map terrain

References

[1] Distributed Machine Learning and the Parameter Server. https://www.cs.cornell.edu/courses/cs4787/2019sp/notes/lecture22.pdf.

[2] Ray: A Distributed System for AI. https://bair.berkeley.edu/blog/2018/01/09/ray/.

[3] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[4] Joris Baan, Maartje ter Hoeve, Marlies van der Wees, Anne Schuth, and Maarten de Rijke. Understanding multi-head attention in abstractive summarization, 2019.

[5] Brooke Chan Vicki Cheung Przemysław Debiak Christy Dennison David Farhi Quirin Fischer Shariq Hashme Chris Hesse Rafal Józefowicz Scott Gray Catherine Olsson Jakub Pachocki Michael Petrov Henrique Pondé de Oliveira Pinto Jonathan Raiman Tim Salimans Jeremy Schlatter Jonas Schneider Szymon Sidor Ilya Sutskever Jie Tang Filip Wolski Susan Zhang Christopher Berner, Greg Brockman. Dota 2 with large scale deep reinforcement learning. abs/1912.06680, 2019.

[6] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.

[7] Arthur Juliani, Vincent-Pierre Berges, Esh Vckay, Yuan Gao, Hunter Henry, Marwan Mattar, and Danny Lange. Unity: A general platform for intelligent agents, 2018.

[8] Marc Lanctot, Edward Lockhart, Jean-Baptiste Lespiau, Vinicius Zambaldi, Satyaki Upadhyay, Julien Pérolat, Sriram Srinivasan, Finbarr Timbers, Karl Tuyls, Shayegan Omidshafiei, Daniel Hennes, Dustin Morrill, Paul Muller, Timo Ewalds, Ryan Faulkner, János Kramár, Bart De Vylder, Brennan Saeta, James Bradbury, David Ding, Sebastian Borgeaud, Matthew Lai, Julian Schrittwieser, Thomas Anthony, Edward Hughes, Ivo Danihelka, and Jonah Ryan-Davis. OpenSpiel: A framework for reinforcement learning in games. *CoRR*, abs/1908.09453, 2019.

[9] Marc Lanctot, Mark H. M. Winands, Tom Pepels, and Nathan R. Sturtevant. Monte carlo tree search with heuristic evaluations using implicit minimax backups, 2014.

[10] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. In *NIPS-W*, 2017.

[11] Stephanie Wang Alexey Tumanov Richard Liaw Eric Liang Melih Elibol Zongheng Yang William Paul Michael I. Jordan Philipp Moritz, Robert Nishihara and UC Berkeley Ion Stoica. Ray: A distributed framework for emerging ai applications. abs/1712.05889, 2018.

[12] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, MSST '10, page 1–10, USA, 2010. IEEE Computer Society.

[13] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. Mastering chess and shogi by self-play with a general reinforcement learning algorithm, 2017.

[14] Joseph Suarez, Yilun Du, Phillip Isola, and Igor Mordatch. Neural mmo: A massively multiagent game environment for training and evaluating intelligent agents, 2019.

[15] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *CoRR*, abs/1706.03762, 2017. cite arxiv:1706.03762Comment: 15 pages, 5 figures.