Chrome Dino Run game using

Deep Q network and Reinforcement Learning

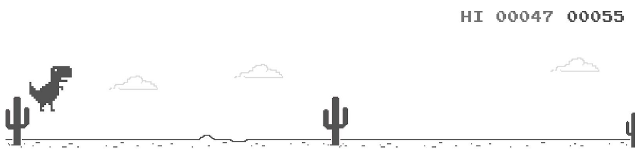
Sameer Shinde, Yogesh Jadhav, Pooja Thakoor
19644693, 47969389, 94203377
Group #6
University of California, Irvine

Abstract
In this project, we implement random exploration with random sampling limited experience replay and an end-to-end experience replay, deep reinforcement learning method to learn to control Chrome offline dinosaur game directly from high-dimensional game screen input. Results show that compared with the random sampling limited experience replay, end-to-end experience replay is more powerful and effective. It leverages all the experiences generated to be trained with equal probability and learn as proceed. Finally, we propose run-time parameter tuning for the size of experience buffer and random batch size. Even with a limited amount of training, our model is able to achieve an acceptable score of 712, which is way beyond most humans can do.

Keywords
Deep Q-Learning, Experience Replay, Random Exploration and Random Sampling.

1 Introduction

Dino Run or T-Rex run is an endless runner game in Chrome browser which is available to play when you're offline aka 'the game you don't usually like to see.



2 Related works

2.1 Deepmind Q learning

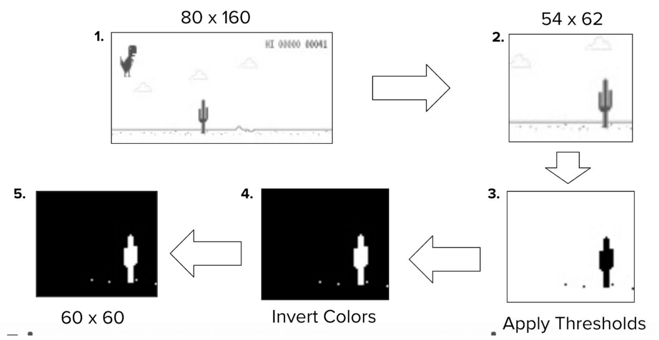
Deepmind at NIPC 2013 presented the first deep learning model to successfully learn control policies directly from high-dimensional sensory input using reinforcement learning. The model is a convolutional neural network, trained with a variant of Q-learning, whose input is raw pixels and whose output is a value function estimating future rewards. They applied this method to seven Atari 2600 games from the Arcade Learning Environment, with no adjustment of the architecture or learning algorithm. The resulting model outperformed all previous approaches on six of the games and surpasses a human expert on three of them

2.2 Experience Replay

To perform experience replay we store the agent's experiences et5 (st,at,rt,st 1 1) at each time-step t in a data set Dt 5 {e1,…,et}. During learning, we apply Q-learning updates, on samples (or mini batches) of experience(s,a,r,s') , U(D), drawn uniformly at random from the pool of stored samples. The Q-learning update at iteration i uses the following loss function:
in which c is the discount factor determining the agent's horizon (Theta), hi are the parameters of the Q-network at iteration i and hi are the network parameters used to compute the target at iteration i. The target network parameters hi are only updated with the Q-network parameters(hi) every C steps and are held fixed between individual updates. Please refer deepmind paper for detailed and correct paragraph

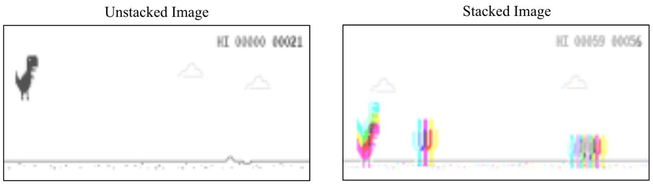
3 Dataset and Preprocessing

Our game is implemented using Open AI gym environment, developed by Elvis Yu-Jing Lin which allows us to extract game screenshots at each frame. Without touching the underlying dynamics of the T-Rex game, the state-space would be a set of screenshots in the form of an 80x160 grid of RGB pixels. However, modelling this large state-space directly is difficult and computationally expensive. Thus, we apply pre-processing to raw pixels for data filtering and feature extraction.



3.1 Image capture and stacking

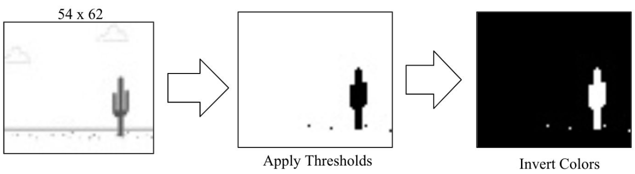
The image processed by the gym environment has a resolution of around 80x160 with 3 (RGB) channels. We intend to use 4 consecutive screenshots as a single input to the model. That makes our single input of dimensions 80x160x3x4. This is computationally expensive and not all the features are useful for playing the game. So we use the OpenCV library to resize, crop and process the image. The final processed input is of just 54x62 pixels and single channelled (grey scale).



3.2 Background Filtering

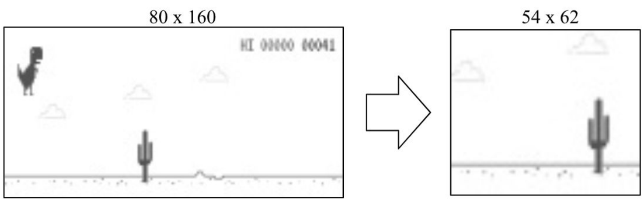
Getting rid of the noise in the image is very important. For example, the background clouds and terrain add a lot of noise in the feature space.

Therefore, We apply thresholds to the input image of size 54x62, such that the pixels with value > 180 turn to all white and pixels with value < 180 turn to black. Then, we invert this image to make important features non-zero.



3.3 Cropping Region of Interest

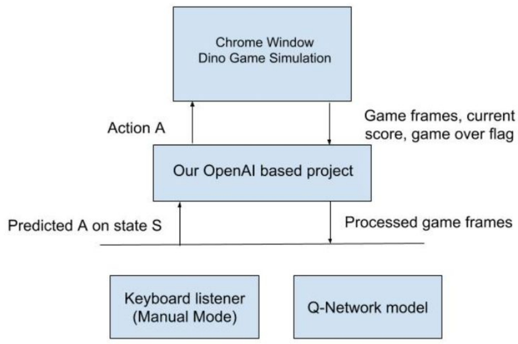
Here we have cropped the agent out because we don't need to learn the agent's features but only the obstacles and the distance from edge. Similarly, score appearing at the upper right corner does not contribute to our decision to jump or not. Therefore we have cropped 20 rows from top and 20 columns from left to form our final input image to be operated.



4. Model

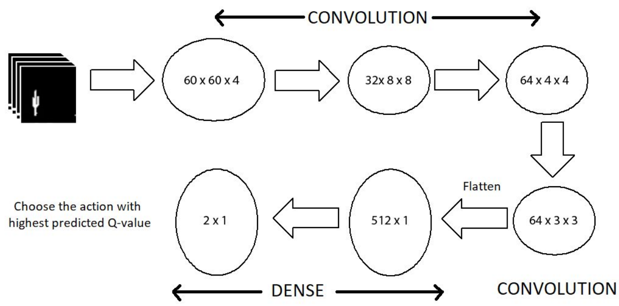
"A child learning to walk"

This might be a new word for many but each and every one of us has learned to walk using the concept of Reinforcement Learning(RL) and this is how our brain still works. A reward system is a basis for any RL algorithm. If we go back to the analogy of a child's walk, a positive reward would be a clap from parents or the ability to reach a candy and a negative reward would say no candy. The child then first learns to stand up before starting to walk. In terms of Artificial Intelligence, the main aim for an agent, in our case the Dino, maximizes a certain numeric reward by performing a particular sequence of actions in the environment. The biggest challenge in RL is the absence of supervision (labelled data) to guide the agent. It must explore and learn on its own. The agent starts by randomly performing actions and observing the rewards each action brings and learns to predict the best possible action when faced with a similar state of the environment.

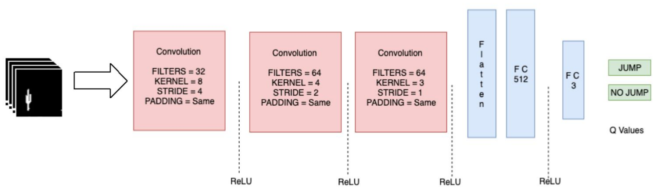


4.1 Deep Q Network

We use a series of three Convolutional layers which are flattened onto a Dense layer of 512 neurons. Don't be surprised by missing pooling layers. They are really useful in image classification problems like ImageNet where we need the network to be insensitive to the location of the object. In our case, however, we care about the position of obstacles.



Our output has a shape equal to the number of possible actions. The model predicts a Q-value, also known as a discounted future reward, for both the actions and we choose the one with the highest value. The method below returns a model built using Keras with tensorflow as back-end.



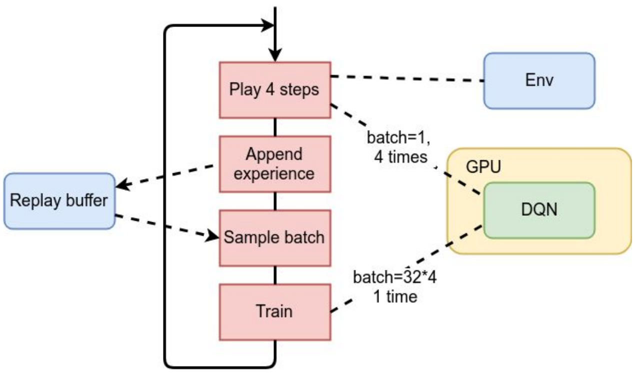
4.2 Experience Replay

Experience replay is an important step in learning the gameplay for our agent. Deepmind had suggested in their paper about a random sampling of ‘n’ batch size experiences to be replayed (trained) on DQN. However, by randomly sampling ‘n’ experiences can lead us to replay unimportant experiences before important ones. Also, if not handled carefully, it can also lead to duplicate the replay for the same. As a result of this, it will take the number of episodes to be played for converging towards goal.

In our experiment, we found, experiences are duplicated in each episode and therefore, it would be beneficial to train all experiences gathered in every episode after the end of that episode and start a new experience buffer for next episode. For our reference, we are naming this method “End to end experience replay”.

This will help the agent to learn what is good and bad, immediately after every episode. The basic intuition behind this idea is to learn more as you progress more.

In our observation, we found this process takes time but converges faster and strives to get global maximum. This method can be utilized with the help GPU.



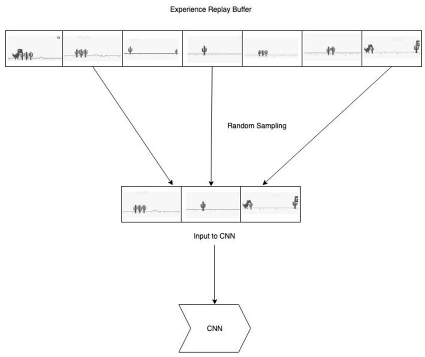
5. Experiment and Discussion

5.1 Random Exploration

Exploration vs Exploitation problem arises when our model tends to stick to same actions while learning, in our case the model might learn that jumping gives better reward rather than doing nothing and in turn apply an always jump policy. However, we would like our model to try out random actions while learning which can give a better reward. We introduce ε, which decides the randomness of actions. We gradually decay its value to reduce the randomness as we progress and then exploit rewarding actions.

5.2 Random Sampling Experience Replay

We performed an experiment with the way we were sampling experiences from Experience replay buffer. By randomly sampling experiences from ERB we tried to break the temporal relationship between experiences.



5.3 Limited Frame Training

The basic reasoning behind this experiment was to not look further than the current jump location. This helped in focusing on only features be captured without looking any further. We cropped the input image (originally 80 X 160) into 80 X 100 so that we only focus on “region of interest” which will be the only next location of cactus.

This process helped in learning faster for gaining increasing score. However this process tends to overfit the model. For instance, if there is a need to have two consecutive jumps. To successfully achieve that, we will have to trigger first jump little earlier than for single cactus case, in order to land on the ground for the next jump. But since we are overlooking the important information about the next consecutive cactus(es), our agent is not able to prepone the jump. As shown in this example, we can see, at score 40 agent is consistently predicting high accurate q values, which signifies the local maximum.

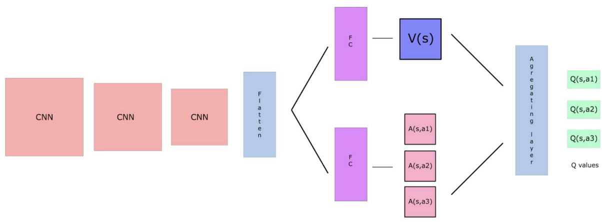
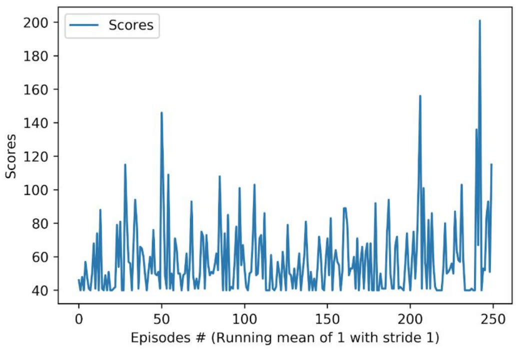


Figure. Deep Q Learning.

6 Results

6.1 Experiment 1:

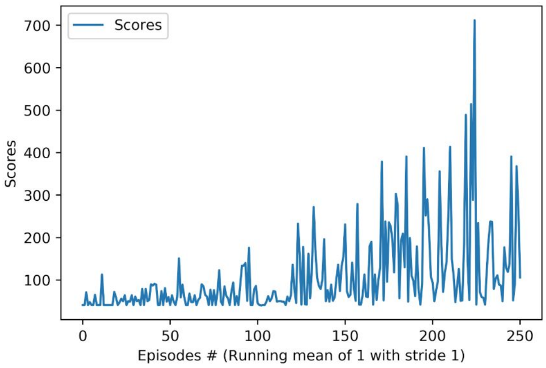
In this experiment, we have a stacked image as a input to the model and observed the score of the game as the number of episodes increased.



We observed a maximum score of 200

6.2 Experiment 2:

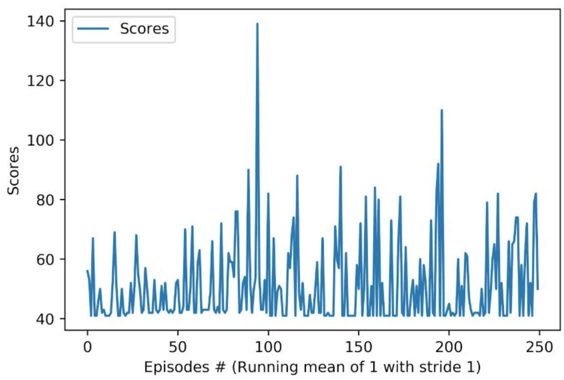
In this experiment, we have a stacked image as an input to the model as the previous experiment but we performed guided training. In this one game was played manually by a user and those experiences were later used for the training in the next episodes. This helped us get better experiences in the buffer and then we observed the score of the game as the number of episodes increased.



We observed maximum score of 712

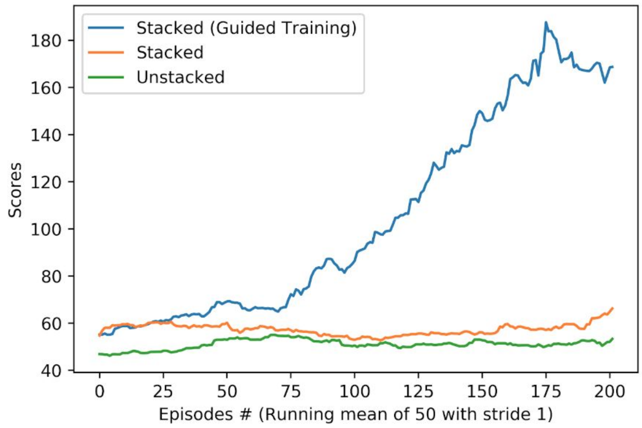
6.3 Experiment 3:

In this experiment, we gave an unstacked image as an input to the model and observed the score of the game as the number of episodes increased.



We observed maximum score of 140

6.4 Observation



We observed that Stacked version learned better as compared to unstacked version as we train more.

7. Conclusion

Thus, we successfully implemented DQN for simulating and learning to ace the gameplay of Chrome-Dino game with many different approaches and compared their results.

8. Future Work

8.1 Long sighted model

We plan to use a wider view of the gameplay for state by cropping less

8.2 Additional Training

We plan to increase the score of the game by performing more training on more episodes.

9 References

9.1 Research papers

[1] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, Martin A. Riedmiller: Playing Atari with Deep Reinforcement Learning at CoRR December 2013

[2] Nelson, Marie, 1988, Bitter bread. The famine in Norrbotten 1867–1868. Studia historica Upsaliensia, Uppsala universitet.

9.2 Development Links

[1] https://pypi.org/project/gym-chrome-dino/

[2] https://medium.com/acing-ai/how-i-build-an-ai-to-play-dino-run-e37f37bdf153

[3] https://blog.paperspace.com/dino-run/

[4]https://github.com/elvisyjlin/gym-chrome-dino/tree/master/gym_chrome_dino

[5] https://www.cs.toronto.edu/~vmnih/docs/dqn.pdf

[6]https://www.intel.ai/demystifying-deep-reinforcement-learning/

10 Appendix

10.1 Division Of Labour:

Sameer:

Q-Table implementation

To get familiar with Reinforcement Learning, we decided to implement a basic Reinforcement Learning program using Q Learning. I took this responsibility for that. I studied Reinforcement learning from Deep reinforcement learning course from here. This helped us get basic understanding and insights about what needs to be implemented.

Deep Q-network model architecture

I worked with Yogesh to build our basic Deep Q network model which would extend our logic from basic reinforcement learning to Deep Q learning.

Yogesh:

Environment Setup and problem formulation

Because of my experience in a similar project, I set up the project environment required for the development. There were minor challenges, but we overcome that with the help of the gym environment available. After carefully analyzing the game world, I defined the scope of development.

Research and Proof of Concept of DQN for Chrome-Dino

Developed multiple versions of DQN to test the result on proof of concepts decided. During this process, I identified an area of improvement with a technique of guided start. This improvement proved to be a major boost to the learning process of the environment.

Pooja:

Image Capturing and Preprocessing:

I worked on the image capturing and preprocessing part of this project. I performed various trials for capturing the frames in real time from chrome browser by using selenium and chrome driver. Then I found a gym environment which was very well suited for our project requirement. After this image was captured, I performed various trials for processing this image. First we tried to give the captured frame directly as an input to the model. But the results were not satisfactory. So, I tried different trials for image cropping and resizing. Also, I tried different methods for background filtering in order to remove the noise. And also performed various trials for improving the score of the Dino run game.

10.2 Source Code:

```
import numpy as np
import cv2

import scipy
import random

import matplotlib
matplotlib.use('Agg')
import matplotlib.pyplot as plt

from keras.models import load_model, clone_model
import keras
from keras.models import Sequential
from keras.layers import Dense, Dropout, Flatten, Conv2D, MaxPooling2D, BatchNormalization, Activation
from  keras.optimizers import Adam
from sklearn.utils.extmath import softmax

import gym
import gym_chrome_dino
from gym_chrome_dino.utils.wrappers import make_dino

from collections import deque
import gc
import time
import pickle
import os
import pandas as pd

from IPython.core.debugger import set_trace

ENV = "master"

MODEL_NAME = ENV + "_model.h5"
MODEL_WEIGHTS_NAME = ENV + "_model_weights.h5"

MODEL_SAVE_PATH = "saved_models/" + MODEL_NAME
MODEL_WEIGHTS_SAVE_PATH = "saved_models/" + MODEL_WEIGHTS_NAME


INPUT_DIMENSIONS = [80, 80, 4]
N_OUTPUT = 2
OUTPUT_LABELS = ["STAY", "UP", "DOWN"]
```

```
# plots configs
PLOTS_DIR_PATH = "outputs/"
SCORE_PER_EPISODE_PLOT_NAME = ENV + "_score_per_episode_plot.pdf"
PLOTS_WINDOW_SIZE = 20
PLOTS_STRIDE_LENGTH = PLOTS_WINDOW_SIZE


# training configs
TRAINING_FLAG = True
SAVE_MODEL_AFTER_N_EPISODES = 10


# Generic configs
GAME_ACCELERATION = True
QUIET = False

# CNN hyperparameters
TRAINING_LEARNING_RATE = 1e-4
TRAINING_EPOCHS = 1
TRAINING_BATCH_SIZE = 16

USE_DROPOUT = False
DROPOUT = 0.25

# Q learning related hyperparameters
REWARD_DECAY_RATE = 0.99
Q_LEARNING_RATE = 0.8
RANDOM_EXPLORARION_INITIAL = 0.9
RANDOM_EXPLORARION_MAX = 1.0
EXPERIENCE_REPLAY_BUFFER_SIZE_MAX = 50000


'''
Random sample size from experience replay buffer with which DQN will be trained on
'''
EXPERIENCE_REPLAY_SAMPLE_SIZE = TRAINING_BATCH_SIZE


# Generic hyperparameters
T = 10 # After T episodes TN will borrow weights from DQN
TRANSFER_MODEL_FLAG = True

def init_cache():
  """initial variable caching, done only once"""
  save_obj(INITIAL_EPSILON,"epsilon")
  t = 0
  save_obj(t,"time")
  D = deque()
  save_obj(D,"D")

'''Call only once to init file structure
'''
#init_cache()

loss_file_path = "./objects/loss_df.csv"
actions_file_path = "./objects/actions_df.csv"
q_value_file_path = "./objects/q_values.csv"
scores_file_path = "./objects/scores_df.csv"


loss_df = pd.read_csv(loss_file_path) if os.path.isfile(loss_file_path) else pd.DataFrame(columns =['loss'])
scores_df = pd.read_csv(scores_file_path) if os.path.isfile(loss_file_path) else pd.DataFrame(columns = ['scores'])


def save_obj(obj, name ):
   with open('objects/'+ name + '.pkl', 'wb') as f: #dump files into objects folder
     pickle.dump(obj, f, pickle.HIGHEST_PROTOCOL)
def load_obj(name ):
  with open('objects/' + name + '.pkl', 'rb') as f:
     return pickle.load(f)


class UTILS:
  def __init__(
       self,
  ):

     self.quiet = QUIET

  def process_observation(self, observation):
     image = np.array(observation)
     x,image = cv2.threshold(image,180,255,cv2.THRESH_BINARY)
     #print(np.array(image).shape)
     image = image[20:, 20:] #Crop Region of Interest(ROI)
     image = cv2.resize(image, (80,80))
     image = 255.0 - image
     return np.array(image)


  '''
     converts the raw numpy observation array os shape (80, 160, 4) to (1, 80, 160, 4)
  '''
  def observation_to_state(self, observation):


     observation = self.process_observation(observation)
            reshaped_observation = np.reshape(observation, (1, INPUT_DIMENSIONS[0],
                                                            INPUT_DIMENSIONS[1], INPUT_DIMENSIONS[2]))
```

```
    return reshaped_observation

  '''
    prints logs
  '''
  def print_log(self, message):
    if not self.quiet: print(message)
    return


class ERB:
  def __init__(
      self,
  ):

    # experience replay buffer
    self.experience_replay_buffer = deque()



  '''
    get the count of samples to extract from the buffer
  '''
  def get_samples_count(self):
    '''
    size = len(self.experience_replay_buffer)
    sampling_ratio = 1.0 * TRAINING_BATCH_SIZE / EXPERIENCE_REPLAY_BUFFER_SIZE_MAX
    sampling_size = int(size * sampling_ratio)
    return min(sampling_size, TRAINING_BATCH_SIZE)
    '''
    return TRAINING_BATCH_SIZE

  '''
    randomly samples from the stored experience replay buffer
  '''
  def sample_from_experience(self):
    experience_size = len(self.experience_replay_buffer)
    sample = []

    # get the count of the samples to be taken from the buffer
    n_sample = self.get_samples_count()
    if n_sample == 0: return sample

    # randomly generate the indices of experiences to be taken
    indices = np.sort(np.random.choice(experience_size, min(experience_size, 10 * n_sample), replace=False))
    print("Indices:", len(indices))
    # collect experiences in the sample
    for indx in indices:
      if n_sample == 0: break
      exp = self.experience_replay_buffer[indx]
      #if exp == None: continue
      #self.experience_replay_buffer[indx] = None
      sample.append(exp)
      #self.experience_replay_buffer.append(exp)
      n_sample = n_sample - 1

    self._remove_experience()
    return sample


  def _remove_experience(self, index=0):
    diff = len(self.experience_replay_buffer) - EXPERIENCE_REPLAY_BUFFER_SIZE_MAX
    temp_buf = []
    while diff > 0 and len(self.experience_replay_buffer) > 0:
      exp = self.experience_replay_buffer.pop()
      if exp == None: diff = diff - 1
      else: temp_buf.append(exp)

    #while len(temp_buf) > 0 and len(self.experience_replay_buffer) < EXPERIENCE_REPLAY_BUFFER_SIZE_MAX:
    #  self.experience_replay_buffer.append(temp_buf.pop())



  '''
    saves the experience in the buffer
    state - current state
    action - action taken on state
    reward - reward for taking action on state
    state_ - new state
  '''
  def save_experience(self, state, action, reward, state_):
    self.experience_replay_buffer.appendleft((state, action, reward, state_))


class AGENT:
  def __init__(self,total_episodes=10):

    self.total_episodes = total_episodes
    self.curr_episode = 0

    # episodic scores
    self.episodic_scores = []

    self.erb = ERB()
    self.utils = UTILS()
    self.dqn = DQN()


  def choose_action(self, observation):

    # convert raw observation to state
    state = self.utils.observation_to_state(observation=observation)
```

```
        # predict action over the state using target network
        preds, preds_classes, preds_probs = self.dqn.predict(states=state)

        s = len(preds_probs.ravel())

        # true action is the action predicted by TN
        true_action = preds_classes[0]

        # random action is action decided randomly
        random_action = np.random.randint(0, s - 1)

        actions = [true_action, random_action]
        curr_random_exploration = self.get_current_random_exploration()

        action_index = np.random.choice(range(2), p=[curr_random_exploration, 1 - curr_random_exploration])
        return actions[action_index]


    '''
        current random exploration is calculated on top of initial random exploration using the episodic progress
    '''
    def get_current_random_exploration(self):
        if not TRAINING_FLAG: return 1.0

        diff = RANDOM_EXPLORARION_MAX - RANDOM_EXPLORARION_INITIAL

        # episodic factor is the progress made by simulation till now
        episodic_factor = self.curr_episode / self.total_episodes

        adjusted_random_exploration = RANDOM_EXPLORARION_INITIAL \
                                      + (episodic_factor * (1 - RANDOM_EXPLORARION_INITIAL))

        return min(adjusted_random_exploration, RANDOM_EXPLORARION_MAX)


    '''
        saves the experience in experience replay buffer
    '''
    def save_this_experience(self, observation, action, reward, observation_):
        state = self.utils.observation_to_state(observation=observation)
        state_ = self.utils.observation_to_state(observation=observation_)
        self.erb.save_experience(state=state, action=action, reward=reward, state_=state_)


    '''
        captures score and appends to the list of scores.
    '''
    def capture_episodic_score(self, score):
        self.episodic_scores.append(score)


    '''
        add the tasks here which the agent has to do at the end of each episode
    '''
    def post_episode_task(self, episode_score):

        force = self.curr_episode == self.total_episodes

        # capture the episodic score
        self.capture_episodic_score(score=episode_score)

        # plot the progress done by agent
        if self.curr_episode % PLOTS_WINDOW_SIZE == 0 or force:
            self.plot_progress(y_data=self.episodic_scores, y_label="Scores")

        if TRAINING_FLAG:
            if self.curr_episode % SAVE_MODEL_AFTER_N_EPISODES == 0:
                gc.collect()
                time.sleep(5)
                self.dqn.save_dqn_model()

#        # do transfer of DQN model to TN model every  T episodes
#        if self.curr_episode %  T == 0 or force:
#            self.dqn.transfer_model()
#        else:
#            # save model after every SAVE_MODEL_AFTER_N_EPISODES
#            if self.curr_episode % SAVE_MODEL_AFTER_N_EPISODES == 0:
#                self.dqn.save_dqn_model()

        return


    '''
        Creates a graph pdf.
        Consumes y_data and y_label and computes a running average of "window" size shifting by "stride" length
    '''
    def plot_progress(self, y_data, y_label, x_label="Episodes #", window=PLOTS_WINDOW_SIZE,
                      stride=PLOTS_STRIDE_LENGTH, dir=PLOTS_DIR_PATH):
```

```
    filename = SCORE_PER_EPISODE_PLOT_NAME
                     if    not    TRAINING_FLAG:    filename    =
SCORE_PER_EPISODE_PLOT_NAME_SIMULATION

    filename = dir + filename
    y_data_mean = []
    index = window
    while True:
      if index > len(y_data): break
      fr = np.int(index - window)
      to = np.int(index)
      w = y_data[fr:to]
      y_data_mean.append(sum(w) * 1.0 / window)
      index = index + stride

    if len(y_data_mean) == 0: return
    x_data = [(x+1) for x in range(len(y_data_mean))]
    plt.plot(x_data, y_data_mean, linewidth=1)
    plt.xlabel(x_label)
    plt.ylabel(y_label)
    plt.savefig(filename)
    #plt.show()


  '''
    fetch samples from ERB randomly
    prepares samples from training
  '''
  def train_on_samples(self, iterations):
    #samples = self.erb.sample_from_experience()
    #print("len of erb:", len(self.erb.experience_replay_buffer))
      samples = random.sample(self.erb.experience_replay_buffer,
len(self.erb.experience_replay_buffer))
    sample_size = len(samples)
    #print("Sample Size:", sample_size)
    features = []
    labels = []

    for idx in range(sample_size):
      experience = samples[idx]
      state  =  experience[0]
      action = experience[1]
      reward = experience[2]
      state_ = experience[3]

      states = np.concatenate((state, state_), axis=0)
      preds, preds_classes, _ = self.dqn.predict(states=states)

      future_reward = 0
        if reward >= 0: future_reward =  REWARD_DECAY_RATE *
preds[1, preds_classes[1]]
      delta = reward + future_reward

      label = np.array([preds[0]])
      label[0, action] = delta

       #label[0, action] =((1 -  Q_LEARNING_RATE) * label[0, action])
\
      #              + ( Q_LEARNING_RATE * delta)

      if len(features) == 0: features = state
      else: features = np.concatenate((features, state), axis=0)

      if len(labels) == 0: labels = label
      else: labels = np.concatenate((labels, label), axis=0)

    #print("Training on ", len(features), " data points...", len(labels))
    l = 0.0
    a = 0.0
    for i in range(iterations):
      loss, acc = self.dqn.train(features=features, labels=labels)
      l += loss
      a += acc
    return l/iterations, a/iterations
class DQN:
  def __init__(
      self,
        restore_model=True # flag to prompt for loading previously
saved dqn model
  ):

    # models
   # self.tn_model = None
    self.dqn_model = None

    #flags
    self.restore_model = restore_model


                  self.decay_rate    =    TRAINING_LEARNING_RATE    /
TRAINING_EPOCHS # decay rate of aptimizer

    self.utils = UTILS()
    self.load_models_initially()

  '''
    trains the DQN model with passed features and labels
  '''
  def train(self, features, labels):

    if not TRAINING_FLAG: return
    if len(features) == 0: return
```

```
        features = np.reshape(features, (-1, INPUT_DIMENSIONS[0],
INPUT_DIMENSIONS[1], INPUT_DIMENSIONS[2]))
    verbose = 1
    if QUIET: verbose = 0
    loss, accuracy = self.dqn_model.train_on_batch(features, labels)
    '''
    self.dqn_model.fit(features, labels,
           batch_size=TRAINING_BATCH_SIZE,
           epochs=TRAINING_EPOCHS,
           verbose=verbose,
           validation_data=(features, labels))
    '''
           #_, accuracy = self.dqn_model.evaluate(features, labels,
verbose=verbose)
    return loss, accuracy

  '''
    predict the labels on passed state
    DEFAULT: model is Target Network
        returns raw pred values, pred classes and softmaxed pred
values
  '''
  def predict(self, states, model=None):
#       if model == None:
#          if TRANSFER_MODEL_FLAG:
#             model = self.tn_model
#          else:
#             model = self.dqn_model
    model = self.dqn_model
           states = np.reshape(states, (-1, INPUT_DIMENSIONS[0],
INPUT_DIMENSIONS[1], INPUT_DIMENSIONS[2]))
    preds = model.predict(states)
    preds_classes = np.argmax(preds, axis=1)
    preds_probs = softmax(preds)
    #set_trace()
    return preds, preds_classes, preds_probs



  '''
        if existing save model exists, loads that saved model and
returns true
    else returns false
  '''
  def load_dqn_model(self):
    status = True
    try:
      self.dqn_model = load_model(MODEL_SAVE_PATH)

self.dqn_model.load_weights(MODEL_WEIGHTS_SAVE_PATH)
    except:
      status = False
    return status


  '''
    saves the dqn model
  '''
  def save_dqn_model(self):
    self.dqn_model.save(MODEL_SAVE_PATH)
    self.dqn_model.save_weights(MODEL_WEIGHTS_SAVE_PATH)



  '''
    loads the target model from dqn model
        NOTE: Before calling this method always make sure to not
have self.dqn NONE and saved weights to be existed
  '''
  def load_tn_model(self):
    return
      # clones the architecture of dqn model with random weights to
create tn model
#       self.tn_model = clone_model(self.dqn_model)

#       self.tn_model.load_weights( MODEL_WEIGHTS_SAVE_PATH)



  '''
    initially loads the tn and dqn
     Call to this method is to be made in constructor at the start of
episodes
  '''
  def load_models_initially(self):
    if not TRAINING_FLAG:
      self.build_network()

self.dqn_model.load_weights(MODEL_WEIGHTS_SAVE_PATH)
      self.load_tn_model()
      return

    if not self.restore_model:
      self.build_network()
    else:
      load_status = self.load_dqn_model()
      if not load_status:
                    print("Cannot load the model at path " +
MODEL_SAVE_PATH + ". Creating a new model!")
        self.build_network()
      else:
                print("Successfully loaded the model at path " +
MODEL_SAVE_PATH + " !")

    self.transfer_model()
    return
```

```
'''
    transfer the weights from dqn to tn
'''
def transfer_model(self):
    self.save_dqn_model()
    if not TRANSFER_MODEL_FLAG: return
    self.load_tn_model()
    return

'''
    builds the architecture for dqn model
    initializes self.dqn_model with that architecture
'''
def build_network(self):
    self.dqn_model = None

    print("Now we build the model")
    self.dqn_model = Sequential()
    self.dqn_model.add(Conv2D(32, (8, 8), strides=(4, 4), padding='same',input_shape=(INPUT_DIMENSIONS[0], INPUT_DIMENSIONS[1], INPUT_DIMENSIONS[2]))) #20*40*4
    self.dqn_model.add(Activation('relu'))
    self.dqn_model.add(Conv2D(64, (4, 4), strides=(2, 2), padding='same'))
    self.dqn_model.add(Activation('relu'))
    self.dqn_model.add(Conv2D(64, (3, 3), strides=(1, 1), padding='same'))
    self.dqn_model.add(Activation('relu'))
    self.dqn_model.add(Flatten())
    self.dqn_model.add(Dense(512))
    self.dqn_model.add(Activation('relu'))
    self.dqn_model.add(Dense(N_OUTPUT))
    adam = Adam(lr=TRAINING_LEARNING_RATE,decay=self.decay_rate)

    self.dqn_model.compile(loss='mse',optimizer=adam,metrics=['accuracy'])
    print("We finish building the model")
    self.dqn_model.summary()


TOTAL_EPISODES = 100
FAIL_ATTEMPTS_ALLOWED = 10
agent = AGENT(total_episodes=TOTAL_EPISODES)

episode = 0
failed_attempts = 0
scores_df_idx = len(scores_df)
loss_df_idx = len(loss_df)

env = gym.make('ChromeDino-v0')
env = make_dino(env, timer=True, frame_stack=True)
while episode < TOTAL_EPISODES and failed_attempts < FAIL_ATTEMPTS_ALLOWED:
    try:
        #env = gym.make('ChromeDinoNoBrowser-v0')
        #env = make_dino(env, timer=True, frame_stack=True)
        observation = env.reset()
    except:
        gc.collect()
        time.sleep(5)
        continue

    agent.curr_episode = episode + 1
    done = False
    n_steps = 0

    while not done:
        action = agent.choose_action(observation=observation)
        observation_, reward, done, info = env.step(action)
        agent.save_this_experience(observation=observation, action=action, reward=reward, observation_=observation_)
        observation = observation_
        n_steps += 1

        '''
            # train DQN with random samples from experience replay buffer
        if len(agent.erb.experience_replay_buffer) >= 100:
            l, a = agent.train_on_samples()
        '''


    score = env.unwrapped.game.get_score()
    #env.close()
    #del env

    agent.post_episode_task(episode_score=score)

    # train DQN with random samples from experience replay buffer
    #if len(agent.erb.experience_replay_buffer) >= 100:
    #iterations = int(n_steps / 5)
    TRAINING_BATCH_SIZE = n_steps
    iterations = 2
    loss = 0.0
    acc = 0.0
    '''
    for idx in range(iterations):
        l, a = agent.train_on_samples(iterations)
        loss += l
        acc += a
    loss = loss/iterations;
    acc = acc/iterations
```

```
    '''
    l, a = agent.train_on_samples(iterations)

      print("Episode #", (episode + 1), " | Score: ", score, " | ERB: ",
len(agent.erb.experience_replay_buffer),
       " | N_STEPS: ", n_steps, "| Loss: ", l, "| Accuracy: " , a)
    agent.erb.experience_replay_buffer.clear()
    scores_df.loc[scores_df_idx] = score
    loss_df.loc[loss_df_idx] = l
    scores_df_idx = scores_df_idx + 1
    loss_df_idx = loss_df_idx + 1
    episode = episode + 1
    time.sleep(2)
    if(episode % 5 == 0): #write scores after every 5 episodes
      scores_df.to_csv("./objects/scores_df.csv",index=False)
      loss_df.to_csv("./objects/loss_df.csv",index=False)
```