Optimized Space Invaders using Deep Q-learning: An Implementation in Tensorflow 2.0.

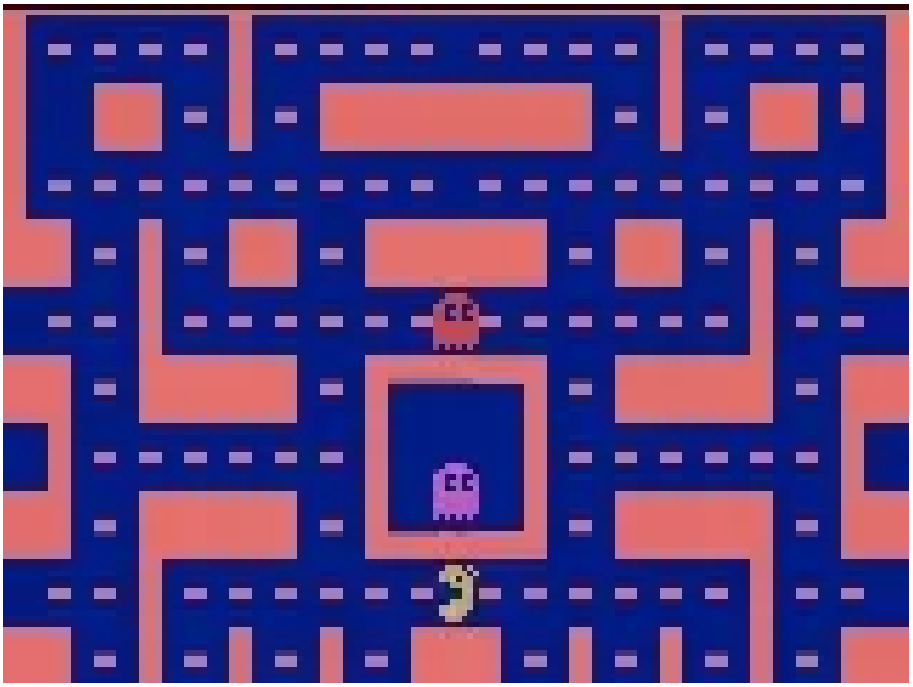
Exploring the Effects of Data Preprocessing

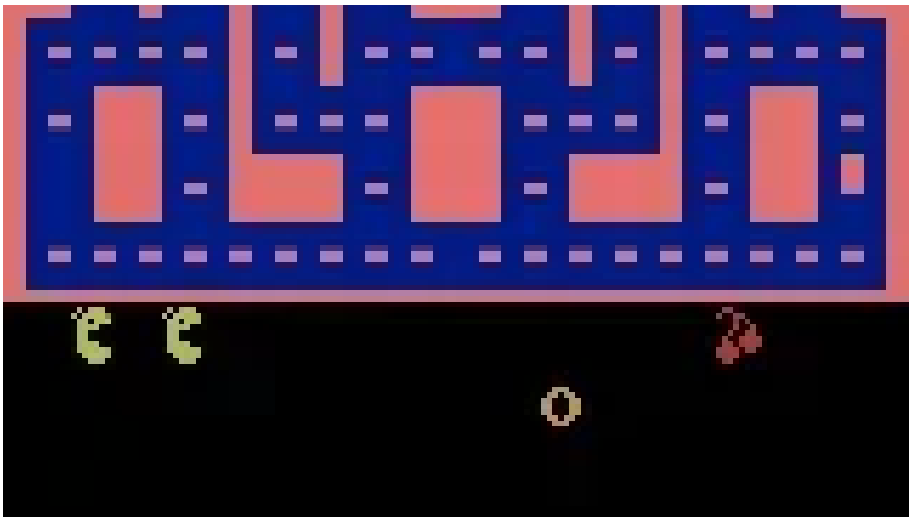
Adrian Yijie Xu Dec 27, 2019 · 13 min read ★

Introduction

Over the past few articles on GradientCrescent, we've spent a significant period of time exploring the field of online learning, a highly reactive family of reinforcement learning algorithms behind many of the latest achievements in general AI. Online learning belongs to the sample-based learning class of approaches, reliant allow for the determination of state values simply through repeated observations, eliminating the need for transition dynamics. Unlike their offline counterparts, **online learning approaches allow for the incremental updates of the values of states and actions during an environmental episode, allowing for constant, incremental performance improvements to be observed.**

Beyond Temporal Difference learning (TD), we've discussed the theory and practical implementations of Q-learning, an evolution of TD designed to allow for incremental estimations and improvement of state-action values. Q-learning has been made famous as becoming the backbone of reinforcement learning approaches to simulated game environments, such as those observed in OpenAI's gyms. As we've already covered theoretical aspects of Q-learning in past articles, they will not be repeated here.



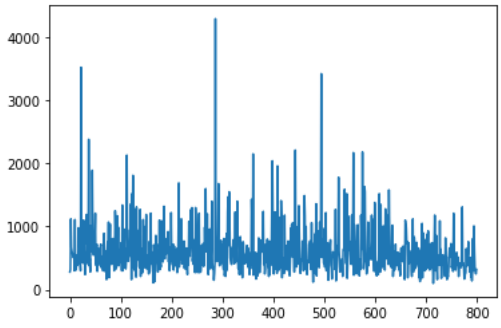


Q-learning powered Miss Pacman, a implemented in our previous article.

In our previous implementation of OpenAI's Miss Pacman gym environment, we relied on a set of observation instances (states) of individual game frames as the inputs for our training process. However, this method is partially flawed, as it fails to take into account many of the idiosyncrasies of the Atari game environment, including:

- The frame-skipping rendering of the game environment that's observed in classic Atari games.
- The presence of multiple fast-moving actors within the environment.
- Frame-specific blinking observed in the agent and environment.

Together, these problems can serve to drastically reduce agent performance, as some instances of data essentially become out-of-domain, or completely irrelevant to the actual game environment. Furthermore, these problems would only become compounded with more complex game environments and real-world applications, such as in autonomous driving. This was observed in our previous implementation as a high level of variance and flat-lining in performance during training.



Reward versus training episodes for our Q-learning trained Miss Pacman agent, trained over 600+800 cycles.

To overcome these problems, we can utilize a couple of techniques first introduced by the Deepmind team in 2015.

- **Frame stacking**: the joining of several game frames together to provide a temporal reference of our game environment.
- **Frame composition:** the element-wise maximization of two game frames together to provide a motion reference that also overcomes the issue of partial rendering.

Let's implement examine the effects of these techniques on the more complex Atari Space Invader's environment.

Implementation

Our Google Colaboratory implementation is written in Python utilizing Tensorflow Core, and can be found on the GradientCrescent Github. We've converted our code to be TF2 compliant, using the new *compat* package. To start with, let's briefly go over the actions required for our Q-learning implementation.

1. **We define our Deep Q-learning neural network**. This is a CNN that takes in-game screen images and outputs the probabilities of each of the actions, or Q-values, in the Ms-Pacman gamespace. To acquire a tensor of probabilities, we do not include any activation function in our final layer.

2. As Q-learning requires us to have knowledge of both the current and next states, we need to **start with data generation**. We feed preprocessed input images of the game space, representing initial states *s*, into the network, and acquire the initial probability distribution of actions, or Q-values. Before training, these values will appear random and sub-optimal. Note that our preprocessing now includes stacking and composition as well.

3. With our tensor of probabilities, we then **select the action with the current highest probability** using the argmax() function and use it to build an epsilon greedy policy.

4. Using our policy, we'll then select the action *a*, and evaluate our decision in the gym environment to **receive information on the new state *s'*, the reward *r***, and whether the episode has been finished.

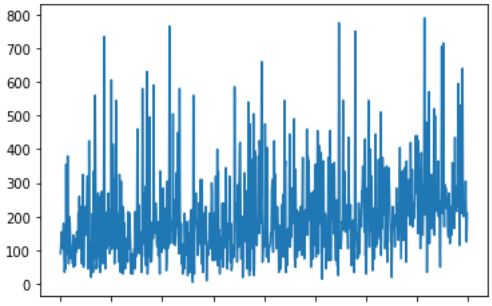
5. We store this combination of information in a buffer in the list form <s,a,r,s',d>, and repeat steps 2–4 for a preset number of times to build up a large enough buffer dataset.

6. Once step 5 has finished, we move to **generate our target *y*-values, *R'* and *A'***, that are required for the loss calculation. While the former is simply discounted from *R*, we obtain the A' by feeding *S'* into our network.

7. With all of our components in place, we can then **calculate the loss to train our network**.

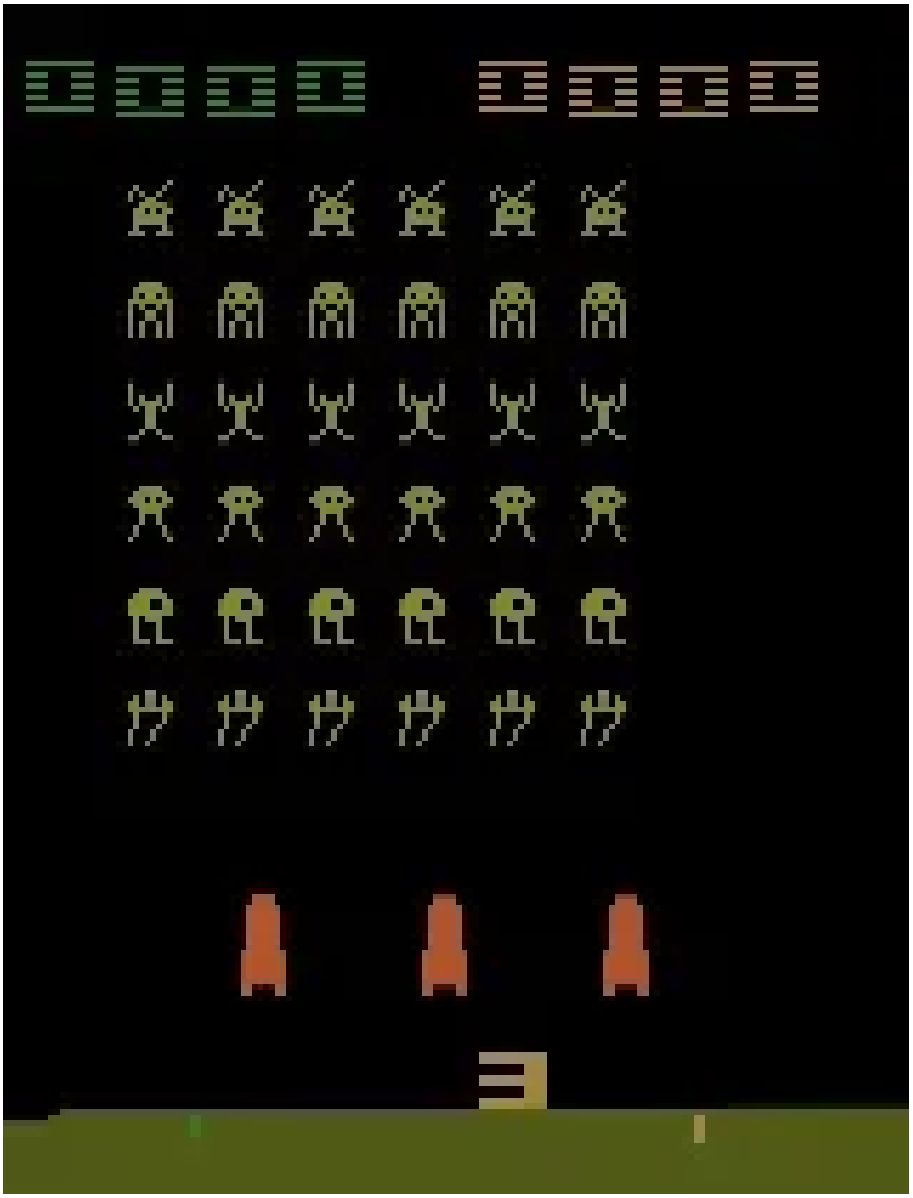
8. Once training has finished, we'll evaluate the performance of our agent graphically and through a demonstration.

For reference, let's first demonstrate the results using the vanilla data input approach, which is essentially the same as that observed in our previous implementation for Miss Pacman. After training our agent for 800 episodes, we observe the following reward distribution.



Reward distribution for the vanilla data input approach for the Space Invaders environment.

Note how the variation in performance exhibits high variation, with a very limited amount of improvement observed after 650 episodes.



Likewise, the performance of our agent is less than stellar, with almost no evasion behavior being detected. If you look closely, you'll notice the blinking in both the outgoing and incoming laser trails — this is an intentional part of the game environment, and results in certain frames having no projectiles in them at all, or only one set of projectiles visible. **This means that elements of our input data are highly misleading, and negatively affect agent performance.**

Let's go over our improved implementation.

We start by importing all of the necessary packages, including the OpenAI gym environments and Tensorflow core.

```
import numpy as np
```

```
import gym

import tensorflow as tf

from tensorflow.contrib.layers import flatten, conv2d,
fully_connected

from collections import deque, Counter

import random

from datetime import datetime
```

Next, we define a preprocessing function to crop the images from our gym environment and convert them into one-dimensional tensors. We've seen this before in our Pong automation implementation.

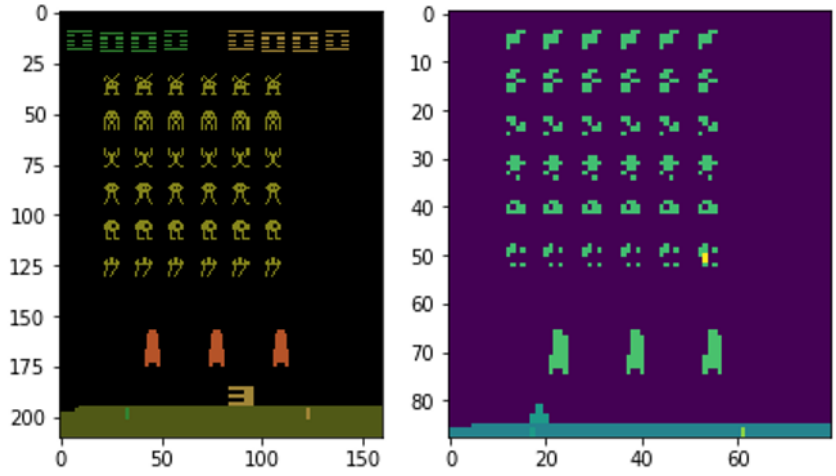
```
def preprocess_observation(obs):

  # Crop and resize the image

  img = obs[25:201:2, ::2]

  # Convert the image to greyscale

  img = img.mean(axis=2)

  # Improve image contrast

  img[img==color] = 0

  # Next we normalize the image from -1 to +1

  img = (img - 128) / 128 - 1

  return img.reshape(88,80)
```

Next, let's initialize the gym environment, and inspect a few screens of gameplay, and also understand the 9 actions available within the gamespace. Naturally, this information is not available to our agent.

```
env = gym.make("SpaceInvaders-v0")

n_outputs = env.action_space.n

print(n_outputs)

print(env.env.get_action_meanings())

observation = env.reset()

import tensorflow as tf

import matplotlib.pyplot as plt

for i in range(22):

if i > 20:

plt.imshow(observation)

plt.show()

observation, _, _, _ = env.step(1)
```

You should observe the following:

We can take this chance to compare our original and preprocessed input images:



Next, we introduce input stacking and input composition into our preprocessing pipeline. Upon a new episode, we start off by taking two of our input frames, and returning an element-wise maximum summation *maxframe* of the two (note that technically this is not necessary, as the two frames are the same, but serves for good practice). The stacked frames are stored in a deque, which automatically removes older entries as new entries are introduced. Initially, we copy the preprocessed maxframe to fill out our deque. As the episode progresses, we create new *maxframes* by taking the new frame, element-wise maximum summing it with the most recent entry in our deque, and then appending the new maxframe to our deque. We then stack the frames at the very end of the process.

```
stack_size = 4 # We stack 4 composite frames in total

# Initialize deque with zero-images one array for each image. Deque
is a special kind of queue that deletes last entry when new entry
comes in

stacked_frames = deque([np.zeros((88,80), dtype=np.int) for i in
range(stack_size)], maxlen=4)

def stack_frames(stacked_frames, state, is_new_episode):

# Preprocess frame
```

```
frame = preprocess_observation(state)

  if is_new_episode:

    # Clear our stacked_frames

    stacked_frames = deque([np.zeros((88,80), dtype=np.int) for i in
range(stack_size)], maxlen=4)

    # Because we're in a new episode, copy the same frame 4x, apply
elementwise maxima

    maxframe = np.maximum(frame,frame)

    stacked_frames.append(maxframe)

    stacked_frames.append(maxframe)

    stacked_frames.append(maxframe)

    stacked_frames.append(maxframe)

  # Stack the frames

     stacked_state = np.stack(stacked_frames, axis=2)

  else:

  #Since deque append adds t right, we can fetch rightmost element

     maxframe=np.maximum(stacked_frames[-1],frame)

  # Append frame to deque, automatically removes the oldest frame

     stacked_frames.append(maxframe)

  # Build the stacked state (first dimension specifies different
frames)

     stacked_state = np.stack(stacked_frames, axis=2)

  return stacked_state, stacked_frames
```

Next, let's define our model, a deep Q-network. This is essentially a three layer convolutional network that takes preprocessed input images, flattens and feeds them to a fully-connected layer, and outputs the probabilities of taking each action in the game space. As previously mentioned, there's no activation layer here, as the presence of one would result in a binary output distribution.

```
def q_network(X, name_scope):

# Initialize layers

initializer =
tf.compat.v1.keras.initializers.VarianceScaling(scale=2.0)

with tf.compat.v1.variable_scope(name_scope) as scope:

# initialize the convolutional layers

layer_1 = conv2d(X, num_outputs=32, kernel_size=(8,8), stride=4,
padding='SAME', weights_initializer=initializer)

tf.compat.v1.summary.histogram('layer_1',layer_1)

layer_2 = conv2d(layer_1, num_outputs=64, kernel_size=(4,4),
stride=2, padding='SAME', weights_initializer=initializer)

tf.compat.v1.summary.histogram('layer_2',layer_2)
```

```
    layer_3 = conv2d(layer_2, num_outputs=64, kernel_size=(3,3),
    stride=1, padding='SAME', weights_initializer=initializer)

    tf.compat.v1.summary.histogram('layer_3',layer_3)

    flat = flatten(layer_3)

    fc = fully_connected(flat, num_outputs=128,
    weights_initializer=initializer)

    tf.compat.v1.summary.histogram('fc',fc)

    #Add final output layer

    output = fully_connected(fc, num_outputs=n_outputs,
    activation_fn=None, weights_initializer=initializer)

    tf.compat.v1.summary.histogram('output',output)

    vars = {v.name[len(scope.name):]: v for v in
    tf.compat.v1.get_collection(key=tf.compat.v1.GraphKeys.TRAINABLE_VAR
    IABLES, scope=scope.name)}

    #Return both variables and outputs together

    return vars, output
```

Let's also take this chance to define our hyperparameters for our model and training process. Note that the X_shape is now *(None, 88, 80, 4)*, on account of our stacked frames.

```
num_episodes = 800

batch_size = 48

input_shape = (None, 88, 80, 1)

learning_rate = 0.001

X_shape = (None, 88, 80, 4)

discount_factor = 0.97

global_step = 0

copy_steps = 100

steps_train = 4

start_steps = 2000
```

Recall, that Q-learning requires us to select actions with the highest action values. To ensure that we still visit every single possible state-action combination, we'll have our agent follow an epsilon-greedy policy, with an exploration rate of 5%. We'll set this exploration rate to decay with time, as we eventually assume all combinations have already been explored — any exploration after that point would simply result in the forced selection of sub-optimal actions.

```
epsilon = 0.5

eps_min = 0.05

eps_max = 1.0

eps_decay_steps = 500000
```

```
#

def epsilon_greedy(action, step):

  p = np.random.random(1).squeeze() #1D entries returned using
squeeze

  epsilon = max(eps_min, eps_max - (eps_max-eps_min) *
step/eps_decay_steps) #Decaying policy with more steps

  if np.random.rand() < epsilon:

    return np.random.randint(n_outputs)

  else:

    return action
```

Recall from the equations above, that the update function for Q-learning requires the following:

- The current state *s*
- The current action *a*
- The reward following the current action *r*
- The next state *s'*
- The next action *a'*

To supply these parameters in meaningful quantities, we need to evaluate our current policy following a set of parameters and store all of the variables in a buffer, from which we'll draw data in minibatches during training. Let's go ahead and create our buffer and a simple sampling function:

```
buffer_len = 20000

#Buffer is made from a deque - double ended queue

exp_buffer = deque(maxlen=buffer_len)

def sample_memories(batch_size):

  perm_batch = np.random.permutation(len(exp_buffer))[:batch_size]

  mem = np.array(exp_buffer)[perm_batch]

  return mem[:,0], mem[:,1], mem[:,2], mem[:,3], mem[:,4]
```

Next, let's copy the weight parameters of our original network into a target network. This dual-network approach allows us to generate data during the training process using an existing policy while still optimizing our parameters for the next policy iteration.

```
# we build our Q network, which takes the input X and generates Q
values for all the actions in the state

mainQ, mainQ_outputs = q_network(X, 'mainQ')

# similarly we build our target Q network, for policy evaluation

targetQ, targetQ_outputs = q_network(X, 'targetQ')
```

```
copy_op = [tf.compat.v1.assign(main_name, targetQ[var_name]) for
var_name, main_name in mainQ.items()]

copy_target_to_main = tf.group(*copy_op)
```

Finally, we'll also define our loss. This is simply the squared difference of our target action (with the highest action value) and our predicted action. We'll use an ADAM optimizer to minimize our loss during training.

```
# define a placeholder for our output i.e action

y = tf.compat.v1.placeholder(tf.float32, shape=(None,1))

# now we calculate the loss which is the difference between actual
value and predicted value

loss = tf.reduce_mean(input_tensor=tf.square(y - Q_action))

# we use adam optimizer for minimizing the loss

optimizer = tf.compat.v1.train.AdamOptimizer(learning_rate)

training_op = optimizer.minimize(loss)

init = tf.compat.v1.global_variables_initializer()

loss_summary = tf.compat.v1.summary.scalar('LOSS', loss)

merge_summary = tf.compat.v1.summary.merge_all()

file_writer = tf.compat.v1.summary.FileWriter(logdir,
tf.compat.v1.get_default_graph())
```

With all of our code defined, let's run our network and go over the training process. We've defined most of this in the initial summary, but let's recall for posterity.

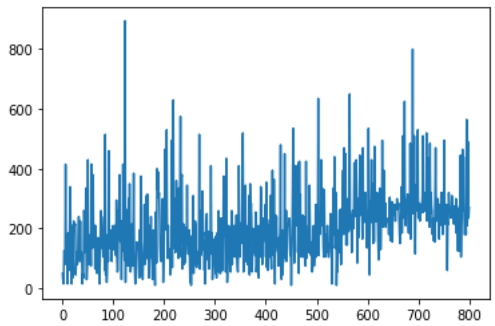
- For each epoch, we feed an input image stack into our network to generate a probability distribution of the available actions, before using an epsilon-greedy policy to select the next action
- We then input this into the network, and obtain information on the next state and accompanying rewards, and store this into our buffer. We update our stack and repeat this process over a number of pre-defined steps.
- After our buffer is large enough, we feed the next states into our network in order to obtain the next action. We also calculate the next reward by discounting the current one
- We generate our target y-values through the Q-learning update function, and train our network.
- By minimizing the training loss, we update the network weight parameters to output improved state-action values for the next policy.

```
with tf.compat.v1.Session() as sess:

  init.run()

  # for each episode
  history = []

  for i in range(num_episodes):
```

```
    done = False

    obs = env.reset()

    epoch = 0

    episodic_reward = 0

     actions_counter = Counter()

    episodic_loss = []

    #First step, preprocess + initialize stack

    obs,stacked_frames= stack_frames(stacked_frames,obs,True)

    # while the state is not the terminal state
    while not done:

    #Data generation using the untrained network

    # feed the game screen and get the Q values for each action

    actions = mainQ_outputs.eval(feed_dict={X:[obs],
in_training_mode:False})

    # get the action
    action = np.argmax(actions, axis=-1)

    actions_counter[str(action)] += 1

    # select the action using epsilon greedy policy

    action = epsilon_greedy(action, global_step)

    # now perform the action and move to the next state, next_obs,
receive reward

    next_obs, reward, done, _ = env.step(action)

    #Updated stacked frames with new episode

    next_obs, stacked_frames = stack_frames(stacked_frames,
next_obs, False)

    # Store this transition as an experience in the replay buffer!
Quite important

    exp_buffer.append([obs, action, next_obs, reward, done])

    # After certain steps, we train our Q network with samples from
the experience replay buffer

    if global_step % steps_train == 0 and global_step > start_steps:

      #Our buffer should already contain everything preprocessed and
stacked

      o_obs, o_act, o_next_obs, o_rew, o_done =
sample_memories(batch_size)

      # states

      o_obs = [x for x in o_obs]

      # next states

      o_next_obs = [x for x in o_next_obs]

      # next actions

      next_act = mainQ_outputs.eval(feed_dict={X:o_next_obs,
in_training_mode:False})

      # discounted reward: these are our Y-values
```

```
      y_batch = o_rew + discount_factor * np.max(next_act, axis=-1)
* (1-o_done)

      # merge all summaries and write to the file

      mrg_summary = merge_summary.eval(feed_dict={X:o_obs,
y:np.expand_dims(y_batch, axis=-1), X_action:o_act,
in_training_mode:False})

      file_writer.add_summary(mrg_summary, global_step)

      # To calculate the loss, we run the previously defined
functions mentioned while feeding inputs

      train_loss, _ = sess.run([loss, training_op], feed_dict=
{X:o_obs, y:np.expand_dims(y_batch, axis=-1), X_action:o_act,
in_training_mode:True})

      episodic_loss.append(train_loss)

      # after some interval we copy our main Q network weights to
target Q network

   if (global_step+1) % copy_steps == 0 and global_step >
start_steps:

      copy_target_to_main.run()

   obs = next_obs

   epoch += 1

   global_step += 1

   episodic_reward += reward

next_obs=np.zeros(obs.shape)

exp_buffer.append([obs, action, next_obs, reward, done])

obs= env.reset()

obs,stacked_frames= stack_frames(stacked_frames,obs,True)

history.append(episodic_reward)

print('Epochs per episode:', epoch, 'Episode Reward:',
episodic_reward,"Episode number:", len(history))
```

Once training is complete, we can plot the reward distribution against incremental episodes. The first 800 episodes are shown below:



Reward distribution for the stacked and composited approach in the Space Invaders environment.

Notice how the core variation in reward distribution has decreased significantly, allowing for a much more consistent inter-episode distribution to be observed, and increases in performance to become more statistically significant. **A visible increase in**

performance can be observed from 550 episodes onwards, a full 100 episodes earlier than that of the vanilla data approach, validating our hypothesis.

To evaluate our results within the confinement of the Colaboratory environment, we can record an entire episode and display it within a virtual display using a wrapped based on the IPython library:

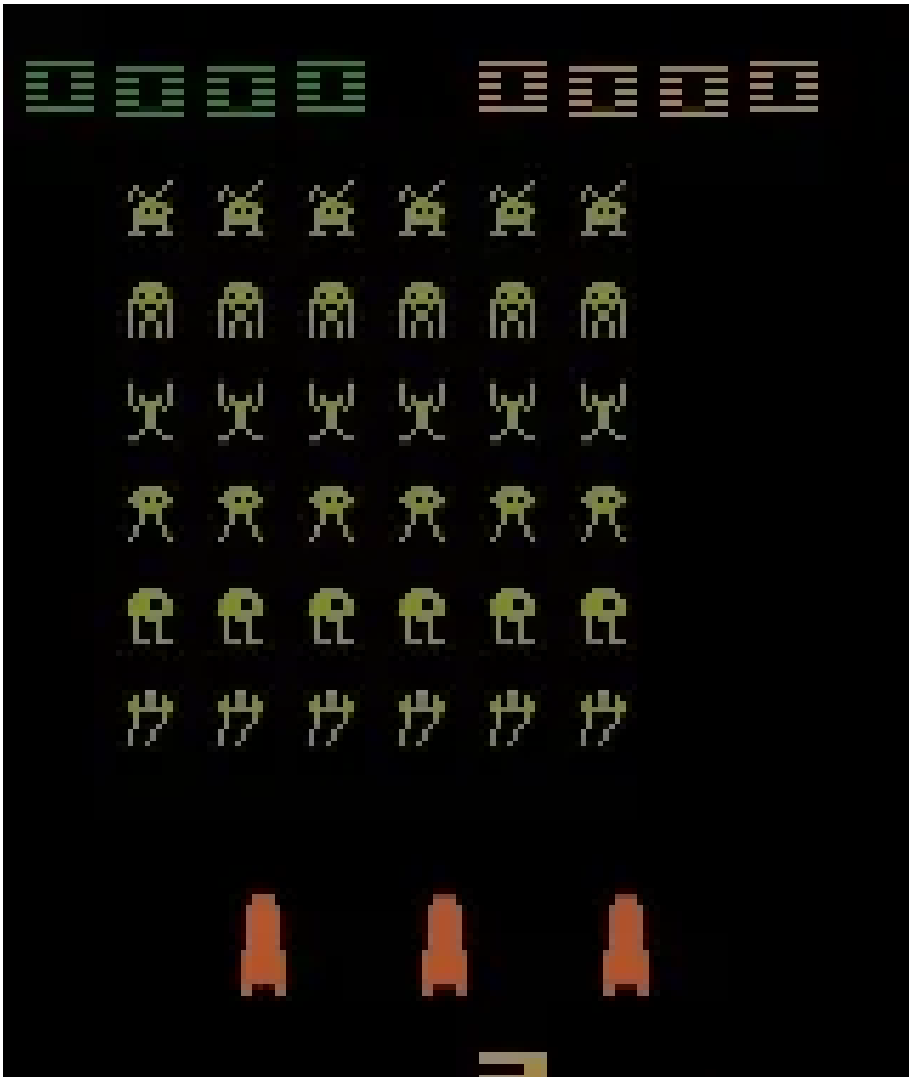
```
“””Utility functions to enable video recording of gym environment
and displaying it. To enable video, just do “env = wrap_env(env)”””

def show_video():

mp4list = glob.glob(‘video/*.mp4’)

if len(mp4list) > 0:

mp4 = mp4list[0]

video = io.open(mp4, ‘r+b’).read()

encoded = base64.b64encode(video)

ipythondisplay.display(HTML(data=’’’<video alt=”test” autoplay

loop controls style=”height: 400px;”>

<source src=”data:video/mp4;base64,{0}” type=”video/mp4" />

</video>’’’.format(encoded.decode(‘ascii’))))

else:

print(“Could not find video”)
def wrap_env(env):

env = Monitor(env, ‘./video’, force=True)

return env
```

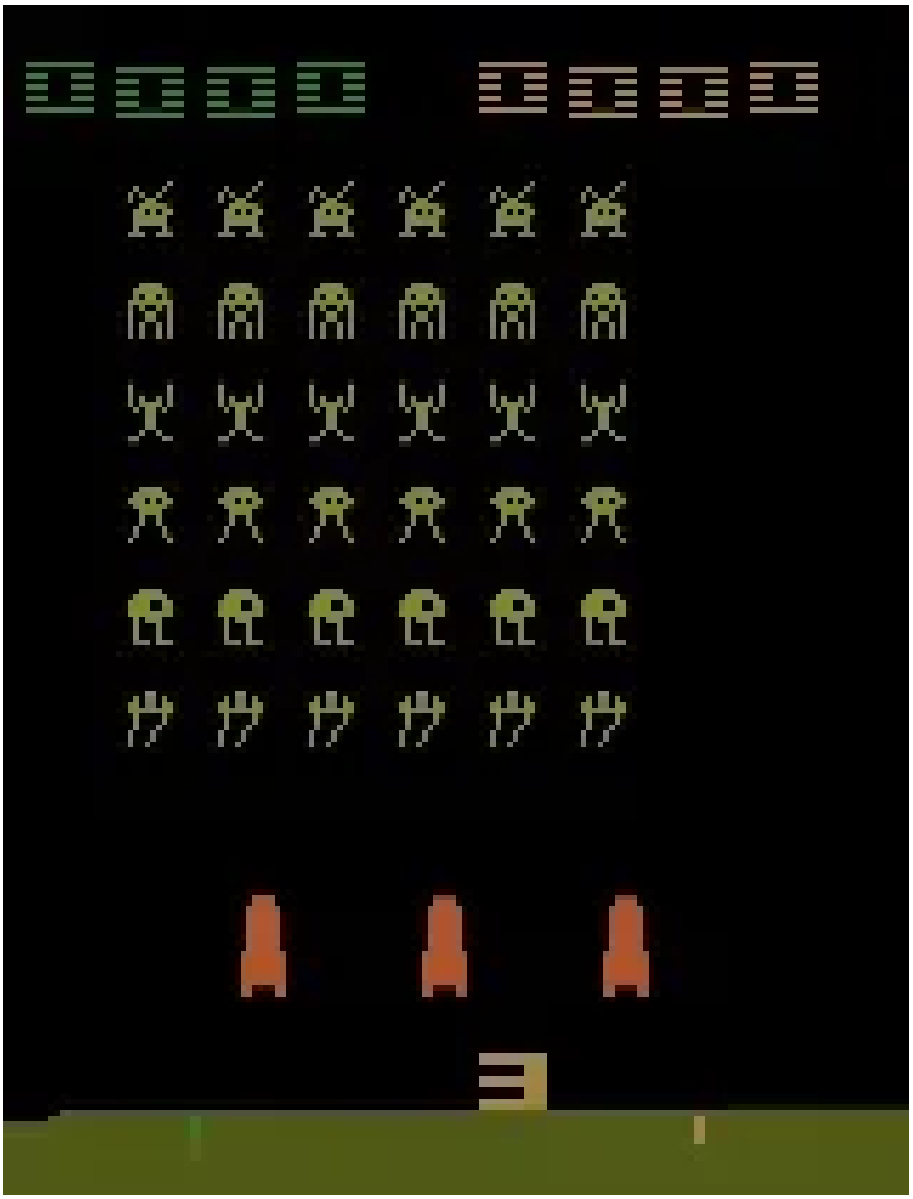
We then run a new session of our environment using our model and record it.

```
Evaluate model on openAi GYM

environment = wrap_env(gym.make('SpaceInvaders-v0'))

done = False

observation = environment.reset()

new_observation = observation

prev_input = None

with tf.compat.v1.Session() as sess:

  init.run()

  observation, stacked_frames = stack_frames(stacked_frames,
observation, True)

  while True:

    #set input to network to be difference image

    # feed the game screen and get the Q values for each action

    actions = mainQ_outputs.eval(feed_dict={X:[observation],
in_training_mode:False})
```

```
    # get the action

    action = np.argmax(actions, axis=-1)

    actions_counter[str(action)] += 1

    # select the action using epsilon greedy policy

    action = epsilon_greedy(action, global_step)

    environment.render()

    new_observation, stacked_frames = stack_frames(stacked_frames,
new_observation, False)

    observation = new_observation

    # now perform the action and move to the next state, next_obs,
receive reward

    new_observation, reward, done, _ = environment.step(action)

    if done:

      break

environment.close()

show_video()
```

Let's inspect a few rounds of gameplay.





Our agent has learned to behave both defensively and offensively, utilizing cover and evasion effectively. The drastic difference in behavior between the two episodes can be attributed to how Q-learning works —actions selected earlier on gain a Q-value, which tend to be favoured with an epsilon-greedy policy. With further training, we would expect these two behaviours to converge.

That wraps up this introduction to optimizing Q-learning. In our next article, we'll take all we've learned and move on from the world of Atari to tackling one of the most well known FPS games in the world.

We hope you enjoyed this article, and hope you check out the many other articles on GradientCrescent, covering applied and theoretical aspects of AI. To stay up to date with the latest updates on GradientCrescent, please consider following the publication and following our Github repository.

References

Sutton et. al, Reinforcement Learning

White et. al, Fundamentals of Reinforcement Learning, University of Alberta

Silva et. al, Reinforcement Learning, UCL

Ravichandiran et. al, Hands-On Reinforcement Learning with Python

Takeshi et. al, Github

Machine Learning Deep Learning Reinforcement Learning Atari Towards Data Science