Reinforcment Learning

Preparing enviroment

In [17]:

```
#Prepare the environment
!apt-get install -y xvfb python-opengl > /dev/null 2>&1
!pip install gym pyvirtualdisplay scikit-video > /dev/null 2>&1

import tensorflow as tf
tf.enable_eager_execution()


import gym
import numpy as np
import matplotlib.pyplot as plt
from IPython import display as ipythondisplay
import time

# Download the class repository
! git clone https://github.com/aamini/introtodeeplearning_labs.git  > /dev/null 2>&1
% cd introtodeeplearning_labs
! git pull
% cd ..

!cp -r /content/introtodeeplearning_labs/lab1 lab1
!cp -r /content/introtodeeplearning_labs/lab1 lab2
!cp -r /content/introtodeeplearning_labs/lab1 lab3
!cp -r /content/lab1/util.py util.py

import introtodeeplearning_labs as util

env = gym.make("CartPole-v0")
env.seed(1) # reproducible, since RL has high variance
env._max_episode_steps = 1000
print ("Enviornment has observation space = {}".format(env.observation_space))

n_actions = env.action_space.n
print ("Number of possible actions that the agent can choose from = {}".format(n_actions))
```

```
/content/introtodeeplearning_labs
Already up to date.
/content
Enviornment has observation space = Box(4,)
Number of possible actions that the agent can choose from = 2
```

Create model

In [0]:

```
def create_cartpole_model():
  model = tf.keras.models.Sequential([
      tf.keras.layers.Dense(units=32, activation='relu'),
      # TODO: define the output dimension of the last Dense layer
      # Hint: think about that the space the agent needs to act in
      tf.keras.layers.Dense(units=n_actions)
  ])
  return model
def create_cartpole_model_2dropout():
  model = tf.keras.models.Sequential([
      tf.keras.layers.Dense(units=32, activation='relu'),
      tf.keras.layers.Dropout(0.5),
      tf.keras.layers.Dense(units=64, activation='relu'),
      tf.keras.layers.Dropout(0.5),
      # TODO: define the output dimension of the last Dense layer
      # Hint: think about that the space the agent needs to act in
      tf.keras.layers.Dense(units=n_actions)
  ])
  return model
def create_cartpole_model_2():
  model = tf.keras.models.Sequential([
      tf.keras.layers.Dense(units=32, activation='relu'),
      tf.keras.layers.Dense(units=64, activation='relu'),
      # TODO: define the output dimension of the last Dense layer
      # Hint: think about that the space the agent needs to act in
      tf.keras.layers.Dense(units=n_actions)
  ])
  return model
def create_cartpole_model_dropout():
  model = tf.keras.models.Sequential([
      tf.keras.layers.Dense(units=32, activation='relu'),
      tf.keras.layers.Dropout(0.5),
      # TODO: define the output dimension of the last Dense layer
      # Hint: think about that the space the agent needs to act in
      tf.keras.layers.Dense(units=n_actions)
  ])
  return model
def create_cartpole_model_3dropout():
  model = tf.keras.models.Sequential([
      tf.keras.layers.Dense(units=32, activation='relu'),
      tf.keras.layers.Dropout(0.3),
      tf.keras.layers.Dense(units=64, activation='relu'),
      tf.keras.layers.Dropout(0.3),
      tf.keras.layers.Dense(units=128, activation='relu'),
      tf.keras.layers.Dropout(0.3),
      # TODO: define the output dimension of the last Dense layer
      # Hint: think about that the space the agent needs to act in
      tf.keras.layers.Dense(units=n_actions)
  ])
  return model
```

Choose Action

In [0]:

```
def choose_action(model, observation):

  observation = observation.reshape([1, -1])
  '''TODO: feed the observations through the model to predict the log probabilities of each possible action.
'''
  logits = model.predict(observation);
  # pass the log probabilities through a softmax to compute true probabilities
  prob_weights = tf.nn.softmax(logits).numpy()

  '''TODO: randomly sample from the prob_weights to pick an action.
  Hint: carefully consider the dimensionality of the input probabilities (vector) and the output action (sca
lar)'''
  action = np.random.choice(n_actions, size=1, replace = False, p = prob_weights[0])[0]
  return action
```

Memory module

In [0]:

```
class Memory:
  def __init__(self):
      self.clear()

  def clear(self):
      self.observations = []
      self.actions = []
      self.rewards = []

  def add_to_memory(self, new_observation, new_action, new_reward):
      self.observations.append(new_observation)
      '''TODO: update the list of actions with new action'''
      self.actions.append(new_action)
      '''TODO: update the list of rewards with new reward'''
      self.rewards.append(new_reward)
```

Utils for the process

In [0]:

```
def normalize(x):
  x -= np.mean(x)
  x /= np.std(x)
  return x

def discount_rewards(rewards, gamma=0.95):
  discounted_rewards = np.zeros_like(rewards)
  R = 0
  for t in reversed(range(0, len(rewards))):
      # update the total discounted reward
      R = R * gamma + rewards[t]
      discounted_rewards[t] = R

  return normalize(discounted_rewards)
def save_video_of_model(model, env_name, filename='agent.mp4'):
  import skvideo.io
  from pyvirtualdisplay import Display
  display = Display(visible=0, size=(40, 30))
  display.start()

  env = gym.make(env_name)
  obs = env.reset()
  shape = env.render(mode='rgb_array').shape[0:2]

  out = skvideo.io.FFmpegWriter(filename)

  done = False
  while not done:
      frame = env.render(mode='rgb_array')
      out.writeFrame(frame)

      action = model(tf.convert_to_tensor(obs.reshape((1,-1)), tf.float32)).numpy().argmax()
      obs, reward, done, info = env.step(action)
  out.close()
  print ("Successfully saved into {}!".format(filename))
```

Loss function

In [0]:

```
def compute_loss(logits, actions, rewards):
  '''TODO: complete the call to compute the loss'''
  loss = tf.nn.sparse_softmax_cross_entropy_with_logits(logits=logits, labels = actions)
  '''TODO: scale the negative log probability by the rewards'''
  loss = loss * rewards
  loss = tf.reduce_mean(loss)
  return loss
```

Training Steps

In [0]:

```
def train_step(model, optimizer, observations, actions, discounted_rewards):
  with tf.GradientTape() as tape:
      # Forward propogate through the agent
      observations = tf.convert_to_tensor(observations, dtype=tf.float32)
      logits = model(observations)

      '''TODO: call the compute_loss function to compute the loss'''
      loss = compute_loss(logits, actions, discounted_rewards)
  '''TODO: run backpropagation using the tape.gradient method'''
  grads = tape.gradient(loss, model.variables)
  optimizer.apply_gradients(zip(grads, model.variables), global_step=tf.train.get_or_create_global_step())
```

Training model module

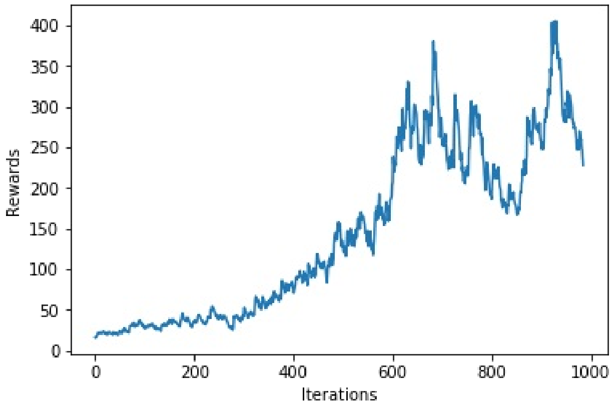
In [0]:

```
def test_model(cartpole_model):
  memory = Memory()
  learning_rate = 1e-3
  optimizer = tf.train.AdamOptimizer(learning_rate)
  smoothed_reward = util.LossHistory(smoothing_factor=0.9)
  plotter = util.PeriodicPlotter(sec=5, xlabel='Iterations', ylabel='Rewards')


  for i_episode in range(1000):

    plotter.plot(smoothed_reward.get())

    # Restart the environment
    observation = env.reset()

    while True:
        # using our observation, take an action
        action = choose_action(cartpole_model, observation)
        next_observation, reward, done, info = env.step(action)
        # add to memory
        memory.add_to_memory(observation, action, reward)

        # is the episode over? did you crash or do so well that you're done?
        if done:
            # determine total reward and keep a record of this
            total_reward = sum(memory.rewards)
            smoothed_reward.append( total_reward )

            # initiate training - remember we don't know anything about how the agent is doing until it's cr
ashed!
            train_step(cartpole_model,
                       optimizer,
                       observations = np.vstack(memory.observations),
                       actions = np.array(memory.actions),
                       discounted_rewards = discount_rewards(memory.rewards)
                      )

            memory.clear()
            break
        # update our observatons
        observation = next_observation



  save_video_of_model(cartpole_model, "CartPole-v0")
```

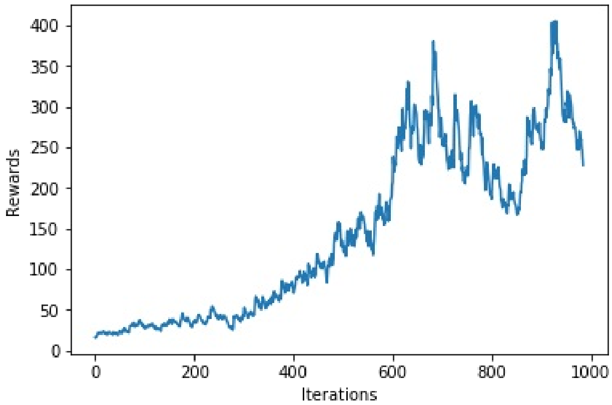
Basic 1 hidden layer module

In [9]:

```
cartpole_model = create_cartpole_model()
test_model(cartpole_model)
```



Successfully saved into agent.mp4!



In [10]:

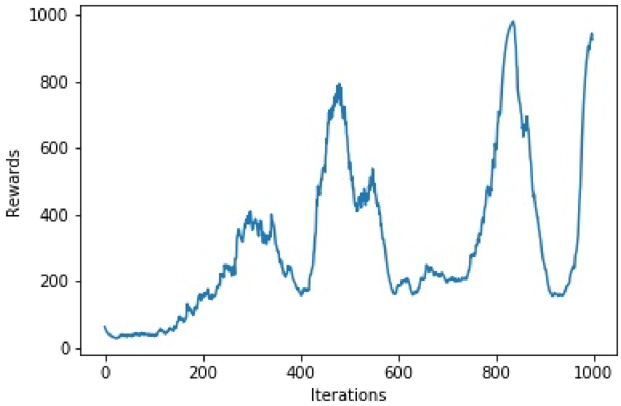
```
from IPython.display import HTML
import io, base64
video = io.open('./agent.mp4', 'r+b').read()
encoded = base64.b64encode(video)
HTML(data='''
<video controls>
    <source src="data:video/mp4;base64,{0}" type="video/mp4" />
</video>'''.format(encoded.decode('ascii')))
```

Out[10]:

Two hidden layers with dropout 0.5

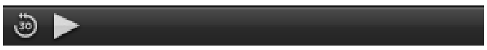
In [11]:

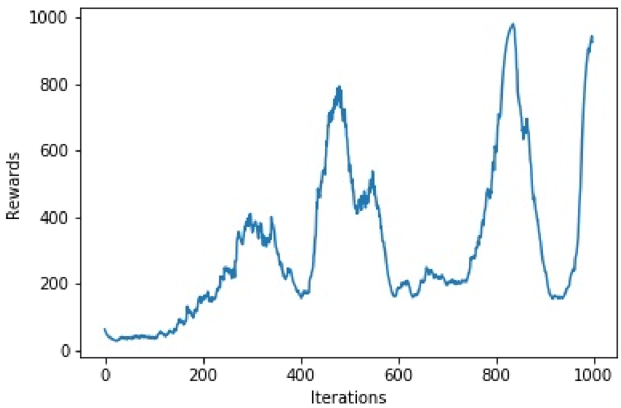
```
model = create_cartpole_model_2dropout()
test_model(model)
from IPython.display import HTML
import io, base64
video = io.open('./agent.mp4', 'r+b').read()
encoded = base64.b64encode(video)
HTML(data='''
<video controls>
    <source src="data:video/mp4;base64,{0}" type="video/mp4" />
</video>'''.format(encoded.decode('ascii')))
```



Successfully saved into agent.mp4!

Out[11]:

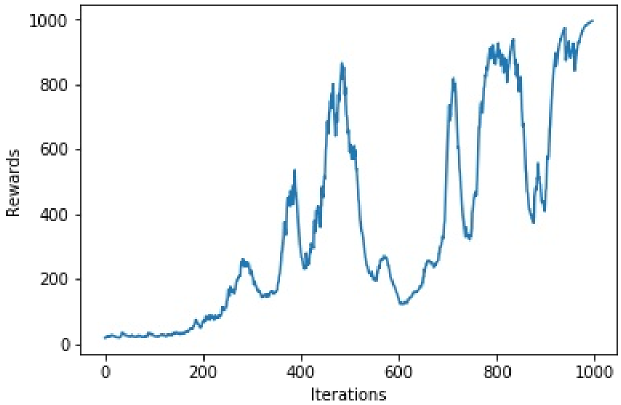




Two hidden layer

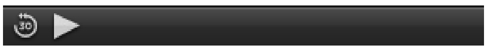
In [12]:

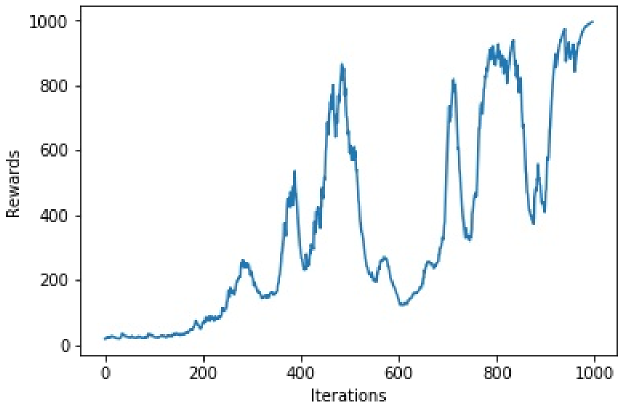
```
model = create_cartpole_model_2()
test_model(model)
from IPython.display import HTML
import io, base64
video = io.open('./agent.mp4', 'r+b').read()
encoded = base64.b64encode(video)
HTML(data='''
<video controls>
    <source src="data:video/mp4;base64,{0}" type="video/mp4" />
</video>'''.format(encoded.decode('ascii')))
```



Successfully saved into agent.mp4!

Out[12]:

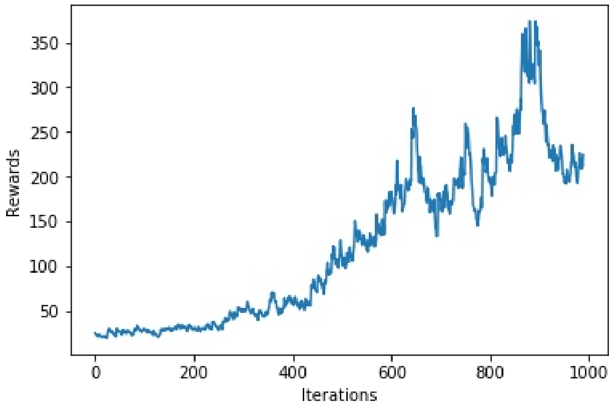




One hidden layer with dropout

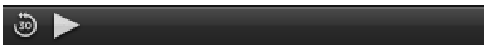
In [13]:

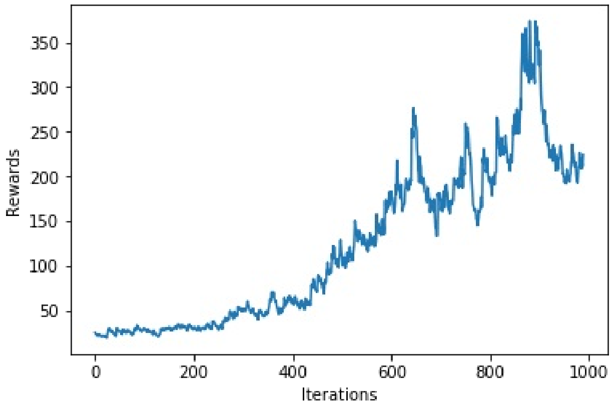
```
model = create_cartpole_model_dropout()
test_model(model)
from IPython.display import HTML
import io, base64
video = io.open('./agent.mp4', 'r+b').read()
encoded = base64.b64encode(video)
HTML(data='''
<video controls>
    <source src="data:video/mp4;base64,{0}" type="video/mp4" />
</video>'''.format(encoded.decode('ascii')))
```



Successfully saved into agent.mp4!

Out[13]:

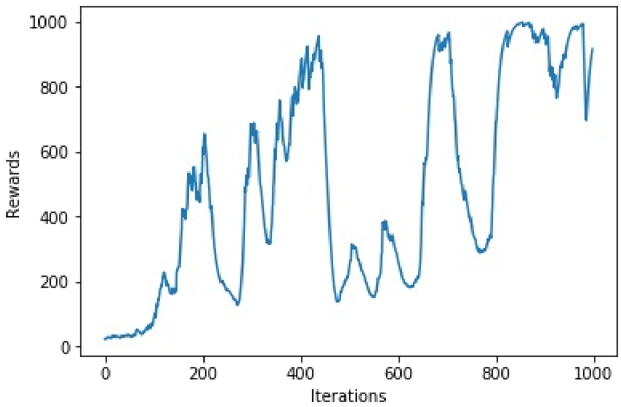




Three hidden layers with dropout

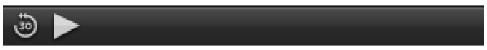
In [14]:

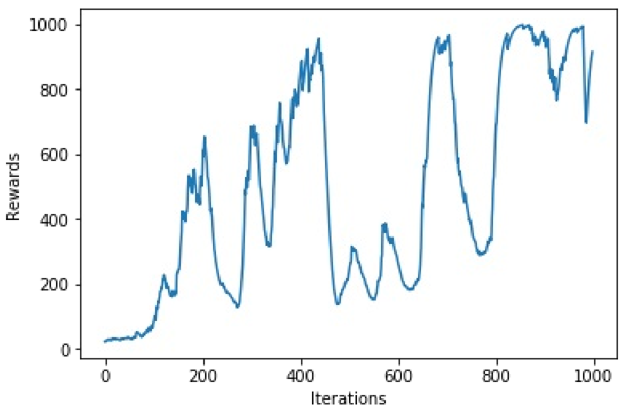
```
model = create_cartpole_model_3dropout()
test_model(model)
from IPython.display import HTML
import io, base64
video = io.open('./agent.mp4', 'r+b').read()
encoded = base64.b64encode(video)
HTML(data='''
<video controls>
    <source src="data:video/mp4;base64,{0}" type="video/mp4" />
</video>'''.format(encoded.decode('ascii')))
```




```
Successfully saved into agent.mp4!
```

Out[14]:





Pong game

In [25]:

```
#Start of the optional part

env = gym.make("Pong-v0")
env.seed(1) # reproducible, since RL has high variance

print "Enviornment has observation space = {}".format(env.observation_space)

n_actions = env.action_space.n
print "Number of possible actions that the agent can choose from = {}".format(n_actions)
```

```
Enviornment has observation space = Box(210, 160, 3)
Number of possible actions that the agent can choose from = 6
```

Create pong model

In [0]:

```
def create_pong_model():
  model = tf.keras.models.Sequential([
      # Define and reshape inputs
      tf.keras.layers.InputLayer(input_shape=(6400,), dtype=tf.float32),
      tf.keras.layers.Reshape((80, 80, 1)),

      # Convolutional layers
      tf.keras.layers.Conv2D(filters=16, kernel_size=(8,8), strides=(4,4), activation='relu', padding='same'
),
      # TODO: define a convolutional layer with 32 4x4 filters and 2x2 stride, ReLu activation
      tf.keras.layers.Conv2D(filters=32, kernel_size=(4,4), strides=(2,2), activation='relu', padding='same'
),
      tf.keras.layers.Flatten(),

      # Fully connected layer and output
      tf.keras.layers.Dense(units=256, activation='relu'),
      # TODO: define the output dimension of the last Dense layer
      # Hint: think about that the space the agent needs to act in
      tf.keras.layers.Dense(n_actions)

  ])
  return model
```

Utils for pong

In [0]:

```
def discount_rewards(rewards, gamma=0.99):
  discounted_rewards = np.zeros_like(rewards)
  R = 0
  for t in reversed(range(0, len(rewards))):
      # NEW: Reset sum
      if rewards[t] != 0:
        R = 0
      # update the total discounted reward as before
      R = R * gamma + rewards[t]
      discounted_rewards[t] = R

  return normalize(discounted_rewards)
```

In [0]:

```
def pre_process(image):
  I = image[35:195] # Crop
  I = I[::2, ::2, 0] # Downsample width and height by a factor of 2
  I[I == 144] = 0 # Remove background type 1
  I[I == 109] = 0 # Remove background type 2
  I[I != 0] = 1 # Set remaining elements (paddles, ball, etc.) to 1
  return I.astype(np.float).ravel()
```

Preprocessing of enviroment images

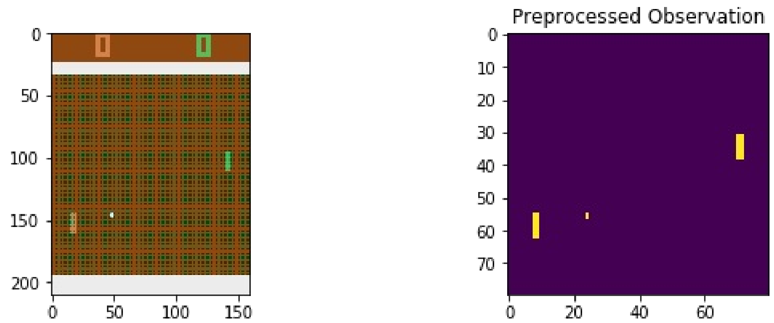
In [29]:

```
observation = env.reset()
for i in range(30):
  observation, _,_,_ = env.step(0)
observation_pp = pre_process(observation)

f = plt.figure(figsize=(10,3))
ax = f.add_subplot(121)
ax2 = f.add_subplot(122)
ax.imshow(observation); ax.grid('off');
ax2.imshow(observation_pp.reshape((80,80))); ax2.grid('off'); plt.title('Preprocessed Observation')
```

```
/usr/local/lib/python2.7/dist-packages/matplotlib/cbook/deprecation.py:107: MatplotlibDeprecati
onWarning: Passing one of 'on', 'true', 'off', 'false' as a boolean is deprecated; use an actua
l boolean (True/False) instead.
  warnings.warn(message, mplDeprecation, stacklevel=1)
```

Out[29]:

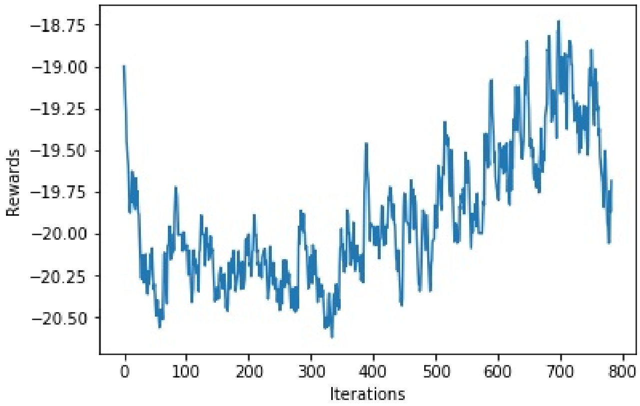
```
Text(0.5,1,'Preprocessed Observation')
```



Run model

In [0]:

```python
learning_rate=1e-4
optimizer = tf.train.AdamOptimizer(learning_rate)
pong_model = create_pong_model()
MAX_ITERS = 10000

smoothed_reward = util.LossHistory(smoothing_factor=0.9)
plotter = util.PeriodicPlotter(sec=5, xlabel='Iterations', ylabel='Rewards')
memory = Memory()

for i_episode in range(MAX_ITERS):

  plotter.plot(smoothed_reward.get())

  # Restart the environment
  observation = env.reset()
  previous_frame = pre_process(observation)


  while True:
      # Pre-process image
      current_frame = pre_process(observation)

      '''TODO: determine the observation change
      Hint: this is the difference between the past two frames'''
      difference = current_frame - previous_frame
      '''TODO: choose an action for the pong model, using the frame difference, and evaluate'''
      action = choose_action(pong_model, difference)

      # Take the chosen action
      next_observation, reward, done, info = env.step(action)
      '''TODO: save the observed frame difference, the action that was taken, and the resulting reward!'''
      memory.add_to_memory(difference, action, reward)

      # is the episode over? did you crash or do so well that you're done?
      if done:
          # determine total reward and keep a record of this
          total_reward = sum(memory.rewards)
          smoothed_reward.append( total_reward )

          # begin training
          train_step(pong_model,
                     optimizer,
                     observations = np.vstack(memory.observations),
                     actions = np.array(memory.actions),
                     discounted_rewards = discount_rewards(memory.rewards))

          memory.clear()
          break

      observation = next_observation
      previous_frame = current_frame

save_video_of_model(pong_model, "Pong-v0", filename='pong_agent.mp4')

from IPython.display import HTML
import io, base64
video = io.open('./pong_agent.mp4', 'r+b').read()
encoded = base64.b64encode(video)
HTML(data='''
<video controls>
    <source src="data:video/mp4;base64,{0}" type="video/mp4" />
</video>'''.format(encoded.decode('ascii')))
```



−18.75
−19.00
−19.25
−19.50
−19.75
−20.00
−20.25
−20.50
Rewards
0
100
200
300
400
500
600
700
800
Iterations