

DLP LAB 4 REINFORCEMENT LEARNING

NAME : QASIM HASAN

ROLL NO: 21K-3210

CLASS : BCS-6A

I HAVE MADE PDF IN THE FOLLOWING ORDER:

- 1.) SETUP FOR THE REINFORCEMENT LIBRARY AND AGENT(pg:1-3)
- 2.) IPYNB FILE FOR THE REINFORCEMENT CODE(pg:4-62)
- 3.) GOOGLE COLAB SCREENSHOTS FOR RUNTIME (pg:63-81)

Question 1

Attached all ss below

Question 2:

I've made 30 iterations.

Depending on the algorithm and problem you are working on, there can be a variation in the number of iterations required for Q-value estimation. Algorithms such as Q-learning in reinforcement learning usually iteratively update Q-values depending on the rewards that are obtained from the environment.

Question 3:

You can use methods such as preprocessing, data augmentation rotation, and choosing suitable feature extraction methods CNN to make sure that features are extracted from images correctly. Visualizing the features that have been extracted can also assist you in determining whether or not they have captured pertinent information from the images.

Question 4:

*In deep learning models, such as those used in reinforcement learning, the activation function is essential. It gives the model **non-linearity**, which enables it to recognize intricate patterns in the data.*

Rectified Linear Unit (ReLU), sigmoid, and tanh are popular activation functions in reinforcement learning. These functions aid the model's ability to estimate Q-values and make judgments using the features it has learned.

1.) REQUIRED SETUP FOR RUNNING THE LAB

```
Files
├── ..
├── .config
├── checkpoints_tutorial16
├── sample_data
└── reinforcement_learning.py

+ Code + Text

Imports

Qasim hasan

21k-3210

[1] 1 !pip install gym[atari]
    2 !pip install autorom[accept-rom-license]

Requirement already satisfied: gym[atari] in /usr/local/lib/python3.10/dist-packages (0.23.0)
Requirement already satisfied: numpy>=1.18.0 in /usr/local/lib/python3.10/dist-packages (from gym[atari]) (1.25.2)
Requirement already satisfied: cloudpickle>=1.2.0 in /usr/local/lib/python3.10/dist-packages (from gym[atari]) (2.2.1)
Requirement already satisfied: gym-notices>=0.0.4 in /usr/local/lib/python3.10/dist-packages (from gym[atari]) (0.0.8)
Requirement already satisfied: ale-py~=0.7.4 in /usr/local/lib/python3.10/dist-packages (from gym[atari]) (0.7.5)
Requirement already satisfied: importlib-resources in /usr/local/lib/python3.10/dist-packages (from ale-py~=0.7.4->gym[atari]) (6.4.0)
Requirement already satisfied: autorom[accept-rom-license] in /usr/local/lib/python3.10/dist-packages (0.6.1)
Requirement already satisfied: click in /usr/local/lib/python3.10/dist-packages (from autorom[accept-rom-license]) (8.1.7)
Requirement already satisfied: requests in /usr/local/lib/python3.10/dist-packages (from autorom[accept-rom-license]) (2.31.0)
Requirement already satisfied: AutoROM.accept-rom-license in /usr/local/lib/python3.10/dist-packages (from autorom[accept-rom-license]) (0.6.1)
Requirement already satisfied: charset-normalizer<4,>=2 in /usr/local/lib/python3.10/dist-packages (from requests->autorom[accept-rom-license]) (3.3.2)
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.10/dist-packages (from requests->autorom[accept-rom-license]) (3.7)
Requirement already satisfied: urllib3<3,>=1.21.1 in /usr/local/lib/python3.10/dist-packages (from requests->autorom[accept-rom-license]) (2.0.7)
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.10/dist-packages (from requests->autorom[accept-rom-license]) (2024.2.2)
```

imported this
Library to run the
depricated reinforcement-learning


Replacing the agent code

21K-3210

▼ Create Agent

The Agent-class implements the main loop for playing the game, recording data and optimizing the Neural Network. We create an object-instance and need to set `training=True` because we want to use the replay-memory to record states and Q-values for plotting further below. We disable logging so this does not corrupt the logs from the actual training that was done previously. We can also set `render=True` but it will have no effect as long as `training=True`.

```
[12] 1 np.float = float
      2 np.int = int
      3 np.object = object
      4 np.bool = bool
      5
      6 agent = rl.Agent(env_name=env_name,
      7                 training=True,
      8                 render=True,
      9                 use_logging=True)
```

 /usr/local/lib/python3.10/dist-packages/gym/envs/registration.py:505: UserWarning: WARN: The environment Breakout-v0 is out of date. You should consider upgrading to version `v5` with the environment ID `ALE/Breakout-v5`.
logger.warn(
/content/reinforcement_learning.py:1186: UserWarning: `tf.layers.conv2d` is deprecated and will be removed in a future version. Please Use `tf.keras.layers.Conv2D` instead.
net = tf.layers.conv2d(inputs=net, name='layer_conv1',
/content/reinforcement_learning.py:1192: UserWarning: `tf.layers.conv2d` is deprecated and will be removed in a future version. Please Use `tf.keras.layers.Conv2D` instead.
net = tf.layers.conv2d(inputs=net, name='layer_conv2',
/content/reinforcement_learning.py:1198: UserWarning: `tf.layers.conv2d` is deprecated and will be removed in a future version. Please Use `tf.keras.layers.Conv2D` instead.
net = tf.layers.conv2d(inputs=net, name='layer_conv3',
/content/reinforcement_learning.py:1205: UserWarning: `tf.layers.flatten` is deprecated and will be removed in a future version. Please use `tf.keras.layers.Flatten` instead.
net = tf.layers.flatten(net)
/content/reinforcement_learning.py:1208: UserWarning: `tf.layers.dense` is deprecated and will be removed in a future version. Please use `tf.keras.layers.Dense` instead.
net = tf.layers.dense(inputs=net, name='layer_fc1', units=1024,
/content/reinforcement_learning.py:1212: UserWarning: `tf.layers.dense` is deprecated and will be removed in a future version. Please use `tf.keras.layers.Dense` instead.
net = tf.layers.dense(inputs=net, name='layer_fc2', units=1024,
/content/reinforcement_learning.py:1216: UserWarning: `tf.layers.dense` is deprecated and will be removed in a future version. Please use `tf.keras.layers.Dense` instead.
net = tf.layers.dense(inputs=net, name='layer_fc3', units=1024,
/content/reinforcement_learning.py:1220: UserWarning: `tf.layers.dense` is deprecated and will be removed in a future version. Please use `tf.keras.layers.Dense` instead.
net = tf.layers.dense(inputs=net, name='layer_fc4', units=1024,
/content/reinforcement_learning.py:1224: UserWarning: `tf.layers.dense` is deprecated and will be removed in a future version. Please use `tf.keras.layers.Dense` instead.
net = tf.layers.dense(inputs=net, name='layer_fc_out', units=num_actions,
WARNING:tensorflow:From /usr/local/lib/python3.10/dist-packages/tensorflow/python/training/rmsprop.py:188: calling Ones.__init__ (from tensorflow.python.ops.init_ops) with dtype is deprecated and will be removed in a future version.
Instructions for updating:
Call initializer instance with the dtype argument instead of passing it to the constructor
Trying to restore last checkpoint ...
Failed to restore checkpoint from: checkpoints_tutorial16/Breakout-v0
Initializing variables instead.

2.) IPYNB FILE AS PDF FOR THE CODE EXECUTED: 21K-3210

Dlp lab 4 reinforcement learning

NAME: QASIM HASAN

ROLL NO: 21K-3210

BCS-6J

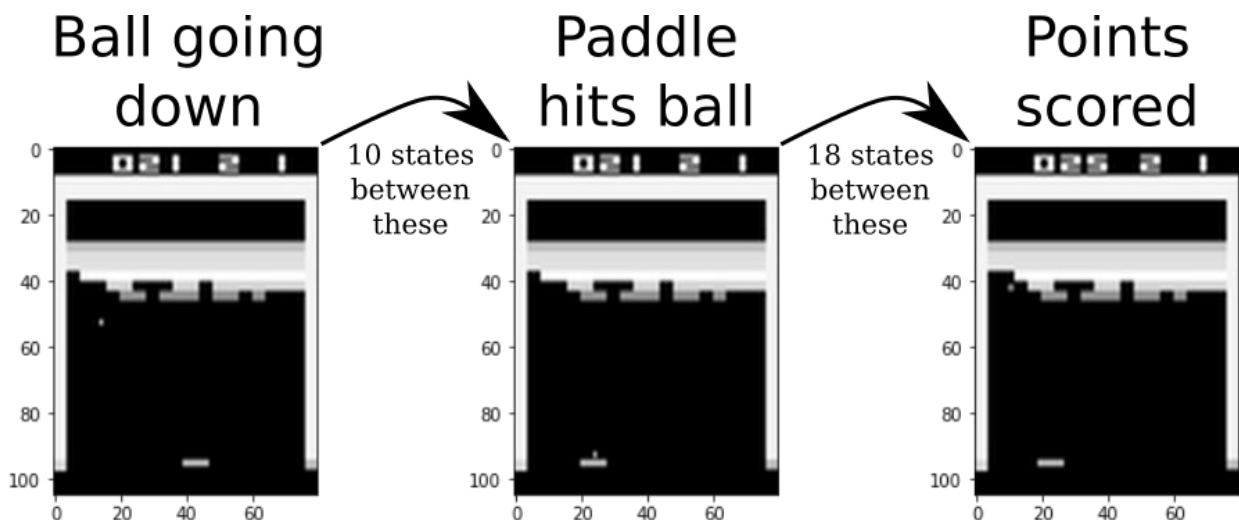
uploaded reinforcement_learning.py to run the library and change the user agent code and dependent libraries as well to make the code run

The Problem

This tutorial uses the Atari game Breakout, where the player or agent is supposed to hit a ball with a paddle, thus avoiding death while scoring points when the ball smashes pieces of a wall.

When a human learns to play a game like this, the first thing to figure out is what part of the game environment you are controlling - in this case the paddle at the bottom. If you move right on the joystick then the paddle moves right and vice versa. The next thing is to figure out what the goal of the game is - in this case to smash as many bricks in the wall as possible so as to maximize the score. Finally you need to learn what to avoid - in this case you must avoid dying by letting the ball pass beside the paddle.

Below are shown 3 images from the game that demonstrate what we need our agent to learn. In the image to the left, the ball is going downwards and the agent must learn to move the paddle so as to hit the ball and avoid death. The image in the middle shows the paddle hitting the ball, which eventually leads to the image on the right where the ball smashes some bricks and scores points. The ball then continues downwards and the process repeats.



The problem is that there are 10 states between the ball going downwards and the paddle hitting the ball, and there are an additional 18 states before the reward is obtained when the ball hits the wall and smashes some bricks. How can we teach an agent to connect these three situations and generalize to similar situations? The answer is to use so-called Reinforcement Learning with a Neural Network, as shown in this tutorial.

Q-Learning

One of the simplest ways of doing Reinforcement Learning is called Q-learning. Here we want to estimate so-called Q-values which are also called action-values, because they map a state of the game-environment to a numerical value for each possible action that the agent may take. The Q-values indicate which action is expected to result in the highest future reward, thus telling the agent which action to take.

Unfortunately we do not know what the Q-values are supposed to be, so we have to estimate them somehow. The Q-values are all initialized to zero and then updated repeatedly as new information is collected from the agent playing the game. When the agent scores a point then the Q-value must be updated with the new information.

There are different formulas for updating Q-values, but the simplest is to set the new Q-value to the reward that was observed, plus the maximum Q-value for the following state of the game. This gives the total reward that the agent can expect from the current game-state and onwards. Typically we also multiply the max Q-value for the following state by a so-called discount-factor slightly below 1. This causes more distant rewards to contribute less to the Q-value, thus making the agent favour rewards that are closer in time.

The formula for updating the Q-value is:

Q-value for state and action = reward + discount * max Q-value for next state

In academic papers, this is typically written with mathematical symbols like this:

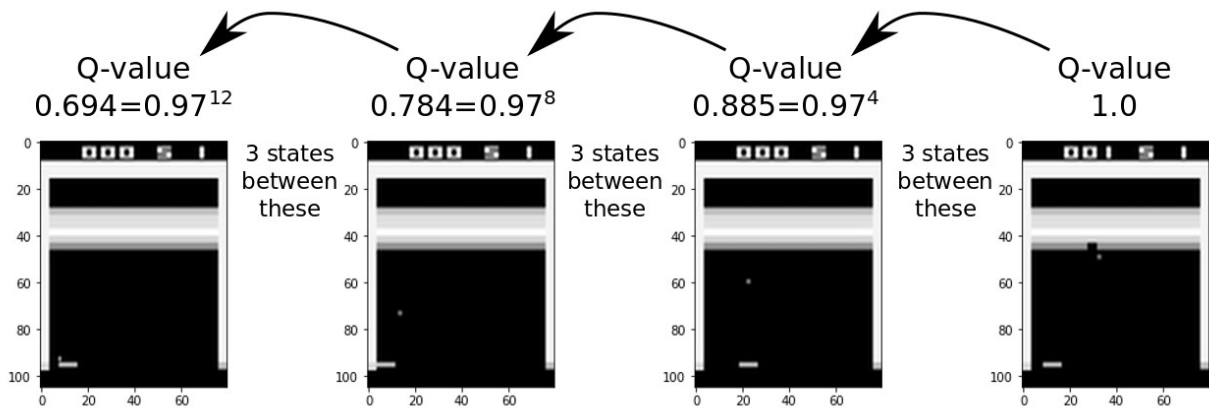
$$Q(s_t, a_t) \leftarrow \underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount}} \cdot \underbrace{\max_a Q(s_{t+1}, a)}_{\text{estimate of future rewards}}$$

Furthermore, when the agent loses a life, then we know that the future reward is zero because the agent is dead, so we set the Q-value for that state to zero.

Simple Example

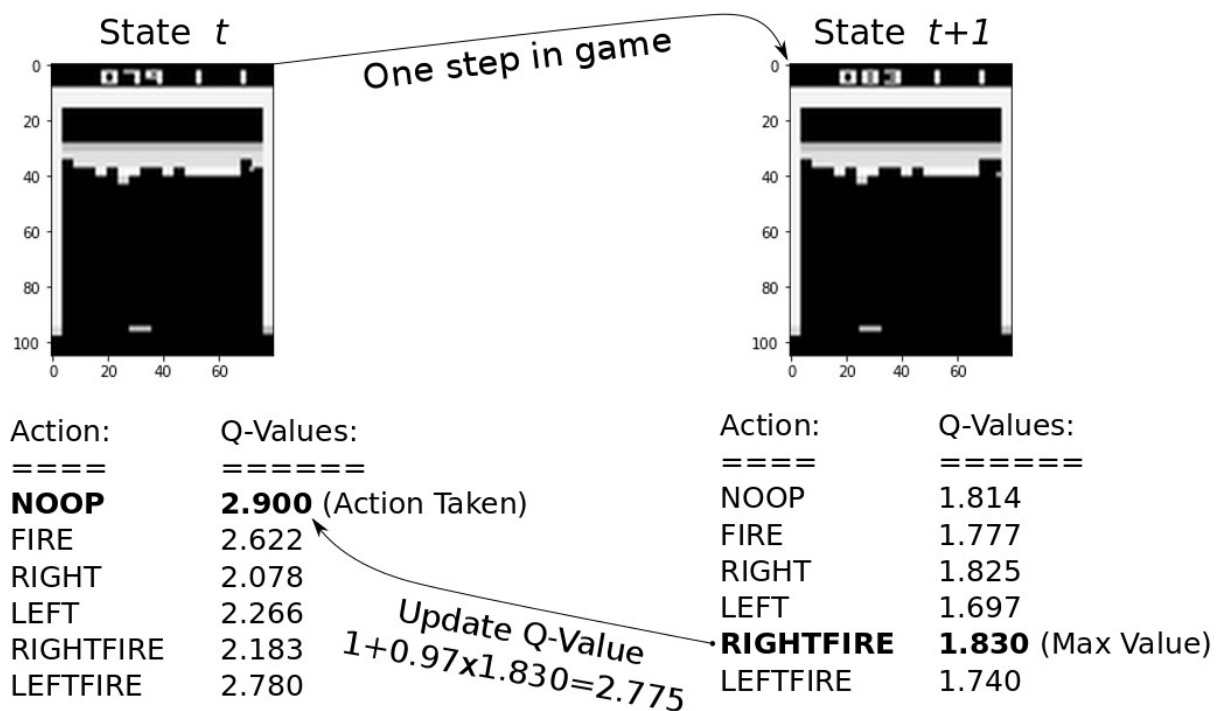
The images below demonstrate how Q-values are updated in a backwards sweep through the game-states that have previously been visited. In this simple example we assume all Q-values have been initialized to zero. The agent gets a reward of 1 point in the right-most image. This reward is then propagated backwards to the previous game-states, so when we see similar game-states in the future, we know that the given actions resulted in that reward.

The discounting is an exponentially decreasing function. This example uses a discount-factor of 0.97 so the Q-value for the 3rd image is about $0.885 \approx 0.97^4$ because it is 4 states prior to the state that actually received the reward. Similarly for the other states. This example only shows one Q-value per state, but in reality there is one Q-value for each possible action in the state, and the Q-values are updated in a backwards-sweep using the formula above. This is shown in the next section.



Detailed Example

This is a more detailed example showing the Q-values for two successive states of the game-environment and how to update them.



The Q-values for the possible actions have been estimated by a Neural Network. For the action NOOP in state t the Q-value is estimated to be 2.900, which is the highest Q-value for that state so the agent takes that action, i.e. the agent does not do anything between state t and $t+1$ because NOOP means "No Operation".

In state $t+1$ the agent scores 4 points, but this is limited to 1 point in this implementation so as to stabilize the training. The maximum Q-value for state $t+1$ is 1.830 for the action RIGHTFIRE. So if we select that action and continue to select the actions proposed by the Q-values

estimated by the Neural Network, then the discounted sum of all the future rewards is expected to be 1.830.

Now that we know the reward of taking the NOOP action from state t to $t + 1$, we can update the Q-value to incorporate this new information. This uses the formula above:

$$Q(\text{state}_t, \text{NOOP}) \leftarrow \underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount}} \cdot \underbrace{\max_a Q(\text{state}_{t+1}, a)}_{\text{estimate of future rewards}} = 1.0 + 0.97 \cdot 1.830 \simeq 2.775$$

The new Q-value is 2.775 which is slightly lower than the previous estimate of 2.900. This Neural Network has already been trained for 150 hours so it is quite good at estimating Q-values, but earlier during the training, the estimated Q-values would be more different.

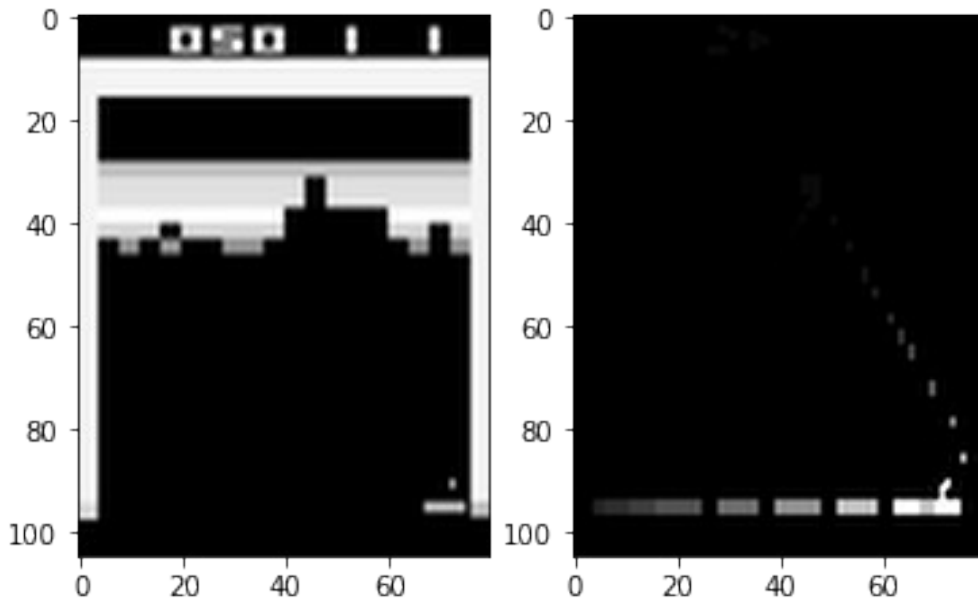
The idea is to have the agent play many, many games and repeatedly update the estimates of the Q-values as more information about rewards and penalties becomes available. This will eventually lead to good estimates of the Q-values, provided the training is numerically stable, as discussed further below. By doing this, we create a connection between rewards and prior actions.

Motion Trace

If we only use a single image from the game-environment then we cannot tell which direction the ball is moving. The typical solution is to use multiple consecutive images to represent the state of the game-environment.

This implementation uses another approach by processing the images from the game-environment in a motion-tracer that outputs two images as shown below. The left image is from the game-environment and the right image is the processed image, which shows traces of recent movements in the game-environment. In this case we can see that the ball is going downwards and has bounced off the right wall, and that the paddle has moved from the left to the right side of the screen.

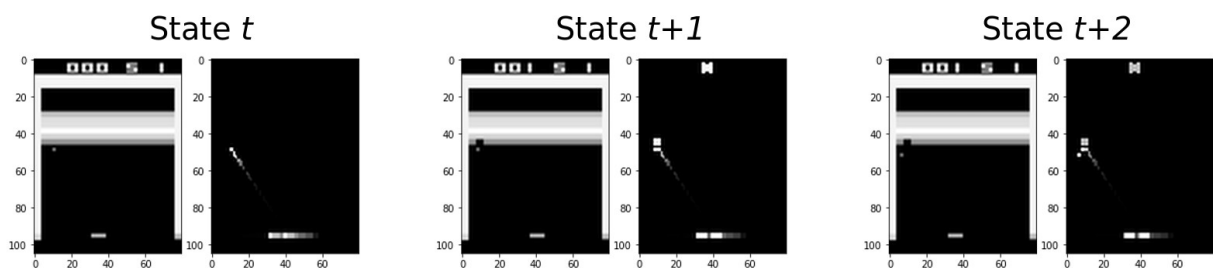
Note that the motion-tracer has only been tested for Breakout and partially tested for Space Invaders, so it may not work for games with more complicated graphics such as Doom.



Training Stability

We need a function approximator that can take a state of the game-environment as input and produce as output an estimate of the Q-values for that state. We will use a Convolutional Neural Network for this. Although they have achieved great fame in recent years, they are actually a quite old technologies with many problems - one of which is training stability. A significant part of the research for this tutorial was spent on tuning and stabilizing the training of the Neural Network.

To understand why training stability is a problem, consider the 3 images below which show the game-environment in 3 consecutive states. At state t the agent is about to score a point, which happens in the following state $t+1$. Assuming all Q-values were zero prior to this, we should now set the Q-value for state $t+1$ to be 1.0 and it should be 0.97 for state t if the discount-value is 0.97, according to the formula above for updating Q-values.



If we were to train a Neural Network to estimate the Q-values for the two states t and $t+1$ with Q-values 0.97 and 1.0, respectively, then the Neural Network will most likely be unable to distinguish properly between the images of these two states. As a result the Neural Network will also estimate a Q-value near 1.0 for state $t+2$ because the images are so similar. But this is clearly wrong because the Q-values for state $t+2$ should be zero as we do not know anything about future rewards at this point, and that is what the Q-values are supposed to estimate.

If this is continued and the Neural Network is trained after every new game-state is observed, then it will quickly cause the estimated Q-values to explode. This is an artifact of training Neural Networks which must have sufficiently large and diverse training-sets. For this reason we will use a so-called Replay Memory so we can gather a large number of game-states and shuffle them during training of the Neural Network.

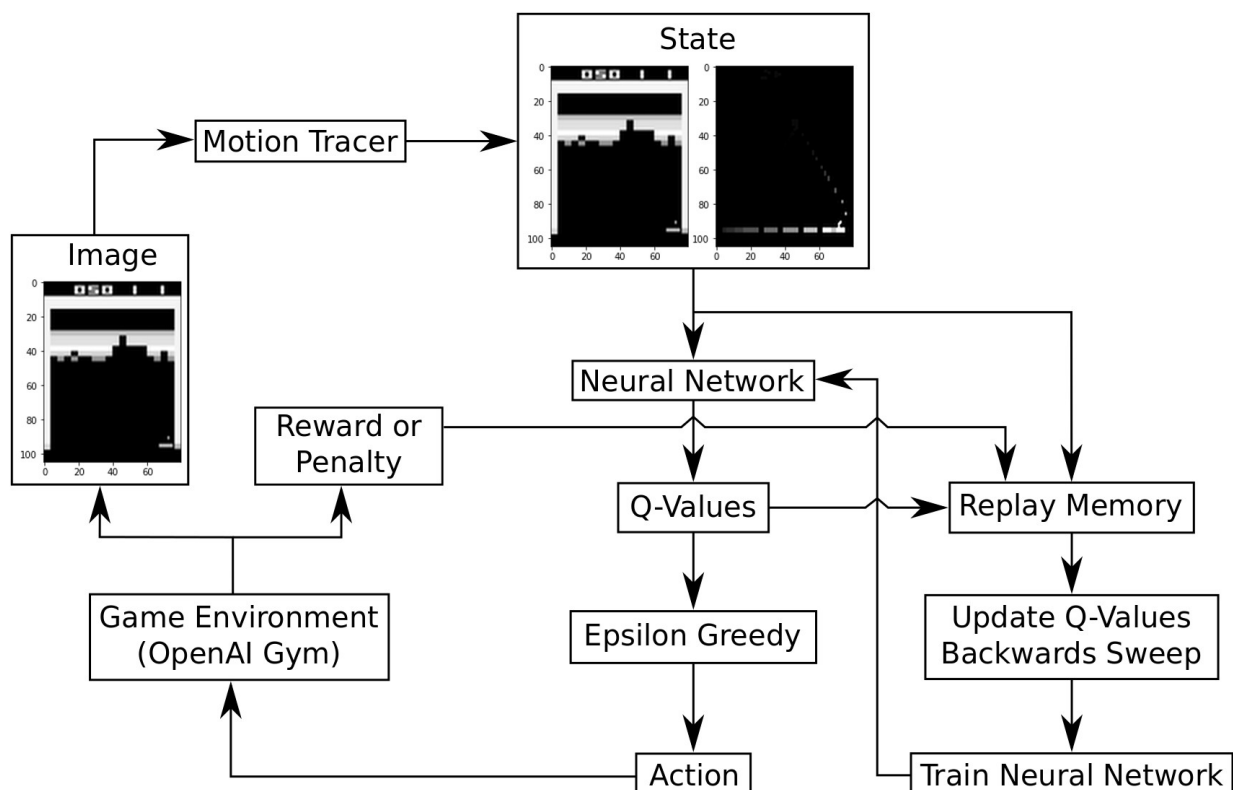
Flowchart

This flowchart shows roughly how Reinforcement Learning is implemented in this tutorial. There are two main loops which are run sequentially until the Neural Network is sufficiently accurate at estimating Q-values.

The first loop is for playing the game and recording data. This uses the Neural Network to estimate Q-values from a game-state. It then stores the game-state along with the corresponding Q-values and reward/penalty in the Replay Memory for later use.

The other loop is activated when the Replay Memory is sufficiently full. First it makes a full backwards sweep through the Replay Memory to update the Q-values with the new rewards and penalties that have been observed. Then it performs an optimization run so as to train the Neural Network to better estimate these updated Q-values.

There are many more details in the implementation, such as decreasing the learning-rate and increasing the fraction of the Replay Memory being used during training, but this flowchart shows the main ideas.



Neural Network Architecture

The Neural Network used in this implementation has 3 convolutional layers, all of which have filter-size 3x3. The layers have 16, 32, and 64 output channels, respectively. The stride is 2 in the first two convolutional layers and 1 in the last layer.

Following the 3 convolutional layers there are 4 fully-connected layers each with 1024 units and ReLU-activation. Then there is a single fully-connected layer with linear activation used as the output of the Neural Network.

This architecture is different from those typically used in research papers from DeepMind and others. They often have large convolutional filter-sizes of 8x8 and 4x4 with high stride-values. This causes more aggressive down-sampling of the game-state images. They also typically have only a single fully-connected layer with 256 or 512 ReLU units.

During the research for this tutorial, it was found that smaller filter-sizes and strides in the convolutional layers, combined with several fully-connected layers having more units, were necessary in order to have sufficiently accurate Q-values. The Neural Network architectures originally used by DeepMind appear to distort the Q-values quite significantly. A reason that their approach still worked, is possibly due to their use of a very large Replay Memory with 1 million states, and that the Neural Network did one mini-batch of training for each step of the game-environment, and some other tricks.

The architecture used here is probably excessive but it takes several days of training to test each architecture, so it is left as an exercise for the reader to try and find a smaller Neural Network architecture that still performs well.

Installation

The [documentation](#) for OpenAI Gym currently suggests that you need to build it in order to install it. But if you just want to install the Atari games, then you only need to install a single pip-package by typing the following commands in a terminal.

- `conda create --name tf-gym --clone tf`
- `source activate tf-gym`
- `pip install gym[atari]`

This assumes you already have an Anaconda environment named `tf` which has TensorFlow installed, it will then be cloned to another environment named `tf-gym` where OpenAI Gym is also installed. This allows you to easily switch between your normal TensorFlow environment and another one which also contains OpenAI Gym.

You can also have two environments named `tf-gpu` and `tf-gpu-gym` for the GPU versions of TensorFlow.

TensorFlow 2

This tutorial was developed using TensorFlow v.1 back in the year 2016-2017. There have been significant API changes in TensorFlow v.2. This tutorial uses TF2 in "v.1 compatibility mode". It

would be too big a job for me to keep updating these tutorials every time Google's engineers update the TensorFlow API, so this tutorial may eventually stop working.

Imports

```
!pip install gym[atari]
!pip install autorom[accept-rom-license]

Requirement already satisfied: gym[atari] in
/usr/local/lib/python3.10/dist-packages (0.23.0)
Requirement already satisfied: numpy>=1.18.0 in
/usr/local/lib/python3.10/dist-packages (from gym[atari]) (1.25.2)
Requirement already satisfied: cloudpickle>=1.2.0 in
/usr/local/lib/python3.10/dist-packages (from gym[atari]) (2.2.1)
Requirement already satisfied: gym-notices>=0.0.4 in
/usr/local/lib/python3.10/dist-packages (from gym[atari]) (0.0.8)
Requirement already satisfied: ale-py~=0.7.4 in
/usr/local/lib/python3.10/dist-packages (from gym[atari]) (0.7.5)
Requirement already satisfied: importlib-resources in
/usr/local/lib/python3.10/dist-packages (from ale-py~=0.7.4-
>gym[atari]) (6.4.0)
Requirement already satisfied: autorom[accept-rom-license] in
/usr/local/lib/python3.10/dist-packages (0.6.1)
Requirement already satisfied: click in
/usr/local/lib/python3.10/dist-packages (from autorom[accept-rom-
license]) (8.1.7)
Requirement already satisfied: requests in
/usr/local/lib/python3.10/dist-packages (from autorom[accept-rom-
license]) (2.31.0)
Requirement already satisfied: AutoROM.accept-rom-license in
/usr/local/lib/python3.10/dist-packages (from autorom[accept-rom-
license]) (0.6.1)
Requirement already satisfied: charset-normalizer<4,>=2 in
/usr/local/lib/python3.10/dist-packages (from requests-
>autorom[accept-rom-license]) (3.3.2)
Requirement already satisfied: idna<4,>=2.5 in
/usr/local/lib/python3.10/dist-packages (from requests-
>autorom[accept-rom-license]) (3.7)
Requirement already satisfied: urllib3<3,>=1.21.1 in
/usr/local/lib/python3.10/dist-packages (from requests-
>autorom[accept-rom-license]) (2.0.7)
Requirement already satisfied: certifi>=2017.4.17 in
/usr/local/lib/python3.10/dist-packages (from requests-
>autorom[accept-rom-license]) (2024.2.2)

%matplotlib inline
import matplotlib.pyplot as plt
import gym
import numpy as np
import math
```

```
# Use TensorFlow v.2 with this old v.1 code.  
# E.g. placeholder variables and sessions have changed in TF2.  
import tensorflow.compat.v1 as tf  
tf.disable_v2_behavior()
```

```
WARNING:tensorflow:From  
/usr/local/lib/python3.10/dist-packages/tensorflow/python/compat/v2_co  
mpat.py:108: disable_resource_variables (from  
tensorflow.python.ops.variable_scope) is deprecated and will be  
removed in a future version.  
Instructions for updating:  
non-resource variables are not supported in the long term
```

```
!pip install keras_rl2
```

```
Requirement already satisfied: keras_rl2 in  
/usr/local/lib/python3.10/dist-packages (1.0.5)  
Requirement already satisfied: tensorflow in  
/usr/local/lib/python3.10/dist-packages (from keras_rl2) (2.15.0)  
Requirement already satisfied: absl-py>=1.0.0 in  
/usr/local/lib/python3.10/dist-packages (from tensorflow->keras_rl2)  
(1.4.0)  
Requirement already satisfied: astunparse>=1.6.0 in  
/usr/local/lib/python3.10/dist-packages (from tensorflow->keras_rl2)  
(1.6.3)  
Requirement already satisfied: flatbuffers>=23.5.26 in  
/usr/local/lib/python3.10/dist-packages (from tensorflow->keras_rl2)  
(24.3.25)  
Requirement already satisfied: gast!=0.5.0,!0.5.1,!0.5.2,>=0.2.1  
in /usr/local/lib/python3.10/dist-packages (from tensorflow->  
keras_rl2) (0.5.4)  
Requirement already satisfied: google-pasta>=0.1.1 in  
/usr/local/lib/python3.10/dist-packages (from tensorflow->keras_rl2)  
(0.2.0)  
Requirement already satisfied: h5py>=2.9.0 in  
/usr/local/lib/python3.10/dist-packages (from tensorflow->keras_rl2)  
(3.9.0)  
Requirement already satisfied: libclang>=13.0.0 in  
/usr/local/lib/python3.10/dist-packages (from tensorflow->keras_rl2)  
(18.1.1)  
Requirement already satisfied: ml-dtypes~=0.2.0 in  
/usr/local/lib/python3.10/dist-packages (from tensorflow->keras_rl2)  
(0.2.0)  
Requirement already satisfied: numpy<2.0.0,>=1.23.5 in  
/usr/local/lib/python3.10/dist-packages (from tensorflow->keras_rl2)  
(1.25.2)  
Requirement already satisfied: opt-einsum>=2.3.2 in  
/usr/local/lib/python3.10/dist-packages (from tensorflow->keras_rl2)  
(3.3.0)  
Requirement already satisfied: packaging in
```

```
/usr/local/lib/python3.10/dist-packages (from tensorflow->keras_rl2)
(24.0)
Requirement already satisfied: protobuf!=4.21.0,!4.21.1,!4.21.2,!
=4.21.3,!4.21.4,!4.21.5,<5.0.0dev,>=3.20.3 in
/usr/local/lib/python3.10/dist-packages (from tensorflow->keras_rl2)
(3.20.3)
Requirement already satisfied: setuptools in
/usr/local/lib/python3.10/dist-packages (from tensorflow->keras_rl2)
(67.7.2)
Requirement already satisfied: six>=1.12.0 in
/usr/local/lib/python3.10/dist-packages (from tensorflow->keras_rl2)
(1.16.0)
Requirement already satisfied: termcolor>=1.1.0 in
/usr/local/lib/python3.10/dist-packages (from tensorflow->keras_rl2)
(2.4.0)
Requirement already satisfied: typing-extensions>=3.6.6 in
/usr/local/lib/python3.10/dist-packages (from tensorflow->keras_rl2)
(4.5.0)
Requirement already satisfied: wrapt<1.15,>=1.11.0 in
/usr/local/lib/python3.10/dist-packages (from tensorflow->keras_rl2)
(1.14.1)
Requirement already satisfied: tensorflow-io-gcs-filesystem>=0.23.1 in
/usr/local/lib/python3.10/dist-packages (from tensorflow->keras_rl2)
(0.37.0)
Requirement already satisfied: grpcio<2.0,>=1.24.3 in
/usr/local/lib/python3.10/dist-packages (from tensorflow->keras_rl2)
(1.63.0)
Requirement already satisfied: tensorboard<2.16,>=2.15 in
/usr/local/lib/python3.10/dist-packages (from tensorflow->keras_rl2)
(2.15.2)
Requirement already satisfied: tensorflow-estimator<2.16,>=2.15.0
in /usr/local/lib/python3.10/dist-packages (from tensorflow-
>keras_rl2) (2.15.0)
Requirement already satisfied: keras<2.16,>=2.15.0 in
/usr/local/lib/python3.10/dist-packages (from tensorflow->keras_rl2)
(2.15.0)
Requirement already satisfied: wheel<1.0,>=0.23.0 in
/usr/local/lib/python3.10/dist-packages (from astunparse>=1.6.0-
>tensorflow->keras_rl2) (0.43.0)
Requirement already satisfied: google-auth<3,>=1.6.3 in
/usr/local/lib/python3.10/dist-packages (from tensorboard<2.16,>=2.15-
>tensorflow->keras_rl2) (2.27.0)
Requirement already satisfied: google-auth-oauthlib<2,>=0.5 in
/usr/local/lib/python3.10/dist-packages (from tensorboard<2.16,>=2.15-
>tensorflow->keras_rl2) (1.2.0)
Requirement already satisfied: markdown>=2.6.8 in
/usr/local/lib/python3.10/dist-packages (from tensorboard<2.16,>=2.15-
>tensorflow->keras_rl2) (3.6)
Requirement already satisfied: requests<3,>=2.21.0 in
```

```

/usr/local/lib/python3.10/dist-packages (from tensorboard<2.16,>=2.15-
>tensorflow->keras_rl2) (2.31.0)
Requirement already satisfied: tensorboard-data-server<0.8.0,>=0.7.0
in /usr/local/lib/python3.10/dist-packages (from
tensorboard<2.16,>=2.15->tensorflow->keras_rl2) (0.7.2)
Requirement already satisfied: werkzeug>=1.0.1 in
/usr/local/lib/python3.10/dist-packages (from tensorboard<2.16,>=2.15-
>tensorflow->keras_rl2) (3.0.3)
Requirement already satisfied: cachetools<6.0,>=2.0.0 in
/usr/local/lib/python3.10/dist-packages (from google-auth<3,>=1.6.3-
>tensorboard<2.16,>=2.15->tensorflow->keras_rl2) (5.3.3)
Requirement already satisfied: pyasn1-modules>=0.2.1 in
/usr/local/lib/python3.10/dist-packages (from google-auth<3,>=1.6.3-
>tensorboard<2.16,>=2.15->tensorflow->keras_rl2) (0.4.0)
Requirement already satisfied: rsa<5,>=3.1.4 in
/usr/local/lib/python3.10/dist-packages (from google-auth<3,>=1.6.3-
>tensorboard<2.16,>=2.15->tensorflow->keras_rl2) (4.9)
Requirement already satisfied: requests-oauthlib>=0.7.0 in
/usr/local/lib/python3.10/dist-packages (from google-auth-
oauthlib<2,>=0.5->tensorboard<2.16,>=2.15->tensorflow->keras_rl2)
(1.3.1)
Requirement already satisfied: charset-normalizer<4,>=2 in
/usr/local/lib/python3.10/dist-packages (from requests<3,>=2.21.0-
>tensorboard<2.16,>=2.15->tensorflow->keras_rl2) (3.3.2)
Requirement already satisfied: idna<4,>=2.5 in
/usr/local/lib/python3.10/dist-packages (from requests<3,>=2.21.0-
>tensorboard<2.16,>=2.15->tensorflow->keras_rl2) (3.7)
Requirement already satisfied: urllib3<3,>=1.21.1 in
/usr/local/lib/python3.10/dist-packages (from requests<3,>=2.21.0-
>tensorboard<2.16,>=2.15->tensorflow->keras_rl2) (2.0.7)
Requirement already satisfied: certifi>=2017.4.17 in
/usr/local/lib/python3.10/dist-packages (from requests<3,>=2.21.0-
>tensorboard<2.16,>=2.15->tensorflow->keras_rl2) (2024.2.2)
Requirement already satisfied: MarkupSafe>=2.1.1 in
/usr/local/lib/python3.10/dist-packages (from werkzeug>=1.0.1-
>tensorboard<2.16,>=2.15->tensorflow->keras_rl2) (2.1.5)
Requirement already satisfied: pyasn1<0.7.0,>=0.4.6 in
/usr/local/lib/python3.10/dist-packages (from pyasn1-modules>=0.2.1-
>google-auth<3,>=1.6.3->tensorboard<2.16,>=2.15->tensorflow-
>keras_rl2) (0.6.0)
Requirement already satisfied: oauthlib>=3.0.0 in
/usr/local/lib/python3.10/dist-packages (from requests-
oauthlib>=0.7.0->google-auth-oauthlib<2,>=0.5-
>tensorboard<2.16,>=2.15->tensorflow->keras_rl2) (3.2.2)

!pip install tf-agents

Requirement already satisfied: tf-agents in
/usr/local/lib/python3.10/dist-packages (0.19.0)
Requirement already satisfied: absl-py>=0.6.1 in

```

```

/usr/local/lib/python3.10/dist-packages (from tf-agents) (1.4.0)
Requirement already satisfied: cloudpickle>=1.3 in
/usr/local/lib/python3.10/dist-packages (from tf-agents) (2.2.1)
Requirement already satisfied: gin-config>=0.4.0 in
/usr/local/lib/python3.10/dist-packages (from tf-agents) (0.5.0)
Requirement already satisfied: gym<=0.23.0,>=0.17.0 in
/usr/local/lib/python3.10/dist-packages (from tf-agents) (0.23.0)
Requirement already satisfied: numpy>=1.19.0 in
/usr/local/lib/python3.10/dist-packages (from tf-agents) (1.25.2)
Requirement already satisfied: pillow in
/usr/local/lib/python3.10/dist-packages (from tf-agents) (9.4.0)
Requirement already satisfied: six>=1.10.0 in
/usr/local/lib/python3.10/dist-packages (from tf-agents) (1.16.0)
Requirement already satisfied: protobuf>=3.11.3 in
/usr/local/lib/python3.10/dist-packages (from tf-agents) (3.20.3)
Requirement already satisfied: wrapt>=1.11.1 in
/usr/local/lib/python3.10/dist-packages (from tf-agents) (1.14.1)
Requirement already satisfied: typing-extensions==4.5.0 in
/usr/local/lib/python3.10/dist-packages (from tf-agents) (4.5.0)
Requirement already satisfied: pygame==2.1.3 in
/usr/local/lib/python3.10/dist-packages (from tf-agents) (2.1.3)
Requirement already satisfied: tensorflow-probability~=0.23.0 in
/usr/local/lib/python3.10/dist-packages (from tf-agents) (0.23.0)
Requirement already satisfied: gym-notices>=0.0.4 in
/usr/local/lib/python3.10/dist-packages (from gym<=0.23.0,>=0.17.0->tf-agents) (0.0.8)
Requirement already satisfied: decorator in
/usr/local/lib/python3.10/dist-packages (from tensorflow-
probability~=0.23.0->tf-agents) (4.4.2)
Requirement already satisfied: gast>=0.3.2 in
/usr/local/lib/python3.10/dist-packages (from tensorflow-
probability~=0.23.0->tf-agents) (0.5.4)
Requirement already satisfied: dm-tree in
/usr/local/lib/python3.10/dist-packages (from tensorflow-
probability~=0.23.0->tf-agents) (0.1.8)

```

The main source-code for Reinforcement Learning is located in the following module:

```
import reinforcement_learning as rl
```

This was developed using Python 3.6.0 (Anaconda) with package versions:

```

# TensorFlow
tf.__version__

{"type": "string"}

# OpenAI Gym
gym.__version__

```

```
{"type": "string"}
```

Game Environment

This is the name of the game-environment that we want to use in OpenAI Gym.

```
env_name = 'Breakout-v0'  
# env_name = 'SpaceInvaders-v0'
```

This is the base-directory for the TensorFlow checkpoints as well as various log-files.

```
rl.checkpoint_base_dir = 'checkpoints_tutorial16/'
```

Once the base-dir has been set, you need to call this function to set all the paths that will be used. This will also create the checkpoint-dir if it does not already exist.

```
rl.update_paths(env_name=env_name)
```

Download Pre-Trained Model

The original version of this tutorial provided some TensorFlow checkpoints with pre-trained models for download. But due to changes in both TensorFlow and OpenAI Gym, these pre-trained models cannot be loaded anymore so they have been deleted from the web-server. You will therefore have to train your own model further below.

Create Agent

The Agent-class implements the main loop for playing the game, recording data and optimizing the Neural Network. We create an object-instance and need to set `training=True` because we want to use the replay-memory to record states and Q-values for plotting further below. We disable logging so this does not corrupt the logs from the actual training that was done previously. We can also set `render=True` but it will have no effect as long as `training==True`.

```
np.float = float  
np.int = int  
np.object = object  
np.bool = bool  
  
agent = rl.Agent(env_name=env_name,  
                 training=True,  
                 render=True,  
                 use_logging=True)  
  
/usr/local/lib/python3.10/dist-packages/gym/envs/registration.py:505:  
UserWarning: WARN: The environment Breakout-v0 is out of date. You  
should consider upgrading to version `v5` with the environment ID
```



```

`ALE/Breakout-v5`.
logger.warn(
/content/reinforcement_learning.py:1186: UserWarning:
`tf.layers.conv2d` is deprecated and will be removed in a future
version. Please Use `tf.keras.layers.Conv2D` instead.
net = tf.layers.conv2d(inputs=net, name='layer_conv1',
/content/reinforcement_learning.py:1192: UserWarning:
`tf.layers.conv2d` is deprecated and will be removed in a future
version. Please Use `tf.keras.layers.Conv2D` instead.
net = tf.layers.conv2d(inputs=net, name='layer_conv2',
/content/reinforcement_learning.py:1198: UserWarning:
`tf.layers.conv2d` is deprecated and will be removed in a future
version. Please Use `tf.keras.layers.Conv2D` instead.
net = tf.layers.conv2d(inputs=net, name='layer_conv3',
/content/reinforcement_learning.py:1205: UserWarning:
`tf.layers.flatten` is deprecated and will be removed in a future
version. Please use `tf.keras.layers.Flatten` instead.
net = tf.layers.flatten(net)
/content/reinforcement_learning.py:1208: UserWarning:
`tf.layers.dense` is deprecated and will be removed in a future
version. Please use `tf.keras.layers.Dense` instead.
net = tf.layers.dense(inputs=net, name='layer_fc1', units=1024,
/content/reinforcement_learning.py:1212: UserWarning:
`tf.layers.dense` is deprecated and will be removed in a future
version. Please use `tf.keras.layers.Dense` instead.
net = tf.layers.dense(inputs=net, name='layer_fc2', units=1024,
/content/reinforcement_learning.py:1216: UserWarning:
`tf.layers.dense` is deprecated and will be removed in a future
version. Please use `tf.keras.layers.Dense` instead.
net = tf.layers.dense(inputs=net, name='layer_fc3', units=1024,
/content/reinforcement_learning.py:1220: UserWarning:
`tf.layers.dense` is deprecated and will be removed in a future
version. Please use `tf.keras.layers.Dense` instead.
net = tf.layers.dense(inputs=net, name='layer_fc4', units=1024,
/content/reinforcement_learning.py:1224: UserWarning:
`tf.layers.dense` is deprecated and will be removed in a future
version. Please use `tf.keras.layers.Dense` instead.
net = tf.layers.dense(inputs=net, name='layer_fc_out',
units=num_actions,
WARNING:tensorflow:From
/usr/local/lib/python3.10/dist-packages/tensorflow/python/training/
rmsprop.py:188: calling Ones.__init__ (from
tensorflow.python.ops.init_ops) with dtype is deprecated and will be
removed in a future version.
Instructions for updating:
Call initializer instance with the dtype argument instead of passing
it to the constructor

```

```
Trying to restore last checkpoint ...  
Failed to restore checkpoint from: checkpoints_tutorial16/Breakout-v0  
Initializing variables instead.
```

The Neural Network is automatically instantiated by the Agent-class. We will create a direct reference for convenience.

```
model = agent.model
```

Similarly, the Agent-class also allocates the replay-memory when `training==True`. The replay-memory will require more than 3 GB of RAM, so it should only be allocated when needed. We will need the replay-memory in this Notebook to record the states and Q-values we observe, so they can be plotted further below.

```
replay_memory = agent.replay_memory
```

Training

The agent's `run()` function is used to play the game. This uses the Neural Network to estimate Q-values and hence determine the agent's actions. If `training==True` then it will also gather states and Q-values in the replay-memory and train the Neural Network when the replay-memory is sufficiently full. You can set `num_episodes=None` if you want an infinite loop that you would stop manually with `ctrl-c`. In this case we just set `num_episodes=1` because we are not actually interested in training the Neural Network any further, we merely want to collect some states and Q-values in the replay-memory so we can plot them below.

```
agent.run(num_episodes=30)
```

1:271	Epsilon: 1.00	Reward: 2.0	Episode Mean: 2.0
2:488	Epsilon: 1.00	Reward: 1.0	Episode Mean: 1.5
3:894	Epsilon: 1.00	Reward: 4.0	Episode Mean: 2.3
4:1165	Epsilon: 1.00	Reward: 2.0	Episode Mean: 2.2
5:1344	Epsilon: 1.00	Reward: 0.0	Episode Mean: 1.8
6:1526	Epsilon: 1.00	Reward: 0.0	Episode Mean: 1.5
7:1776	Epsilon: 1.00	Reward: 1.0	Episode Mean: 1.4
8:2050	Epsilon: 1.00	Reward: 2.0	Episode Mean: 1.5
9:2363	Epsilon: 1.00	Reward: 2.0	Episode Mean: 1.6
10:2667	Epsilon: 1.00	Reward: 2.0	Episode Mean: 1.6
11:2892	Epsilon: 1.00	Reward: 1.0	Episode Mean: 1.5
12:3121	Epsilon: 1.00	Reward: 1.0	Episode Mean: 1.5
13:3335	Epsilon: 1.00	Reward: 1.0	Episode Mean: 1.5
14:3614	Epsilon: 1.00	Reward: 2.0	Episode Mean: 1.5
15:3799	Epsilon: 1.00	Reward: 0.0	Episode Mean: 1.4
16:4012	Epsilon: 1.00	Reward: 0.0	Episode Mean: 1.3
17:4314	Epsilon: 1.00	Reward: 2.0	Episode Mean: 1.4
18:4522	Epsilon: 1.00	Reward: 1.0	Episode Mean: 1.3
19:4997	Epsilon: 1.00	Reward: 5.0	Episode Mean: 1.5
20:5166	Epsilon: 1.00	Reward: 0.0	Episode Mean: 1.4

21:5337	Epsilon: 1.00	Reward: 0.0	Episode Mean: 1.4
22:5766	Epsilon: 0.99	Reward: 4.0	Episode Mean: 1.5
23:5961	Epsilon: 0.99	Reward: 0.0	Episode Mean: 1.4
24:6237	Epsilon: 0.99	Reward: 2.0	Episode Mean: 1.5
25:6478	Epsilon: 0.99	Reward: 1.0	Episode Mean: 1.4
26:6716	Epsilon: 0.99	Reward: 1.0	Episode Mean: 1.4
27:6902	Epsilon: 0.99	Reward: 0.0	Episode Mean: 1.4
28:7174	Epsilon: 0.99	Reward: 2.0	Episode Mean: 1.4
29:7422	Epsilon: 0.99	Reward: 2.0	Episode Mean: 1.4
30:7655	Epsilon: 0.99	Reward: 1.0	Episode Mean: 1.4

In training-mode, this function will output a line for each episode. The first counter is for the number of episodes that have been processed. The second counter is for the number of states that have been processed. These two counters are stored in the TensorFlow checkpoint along with the weights of the Neural Network, so you can restart the training e.g. if you only have one computer and need to train during the night.

Note that the number of episodes is almost 90k. It is impractical to print that many lines in this Notebook, so the training is better done in a terminal window by running the following commands:

```
source activate tf-gpu-gym # Activate your Python environment with TF
and Gym.
python reinforcement_learning.py --env Breakout-v0 --training
```

Training Progress

Data is being logged during training so we can plot the progress afterwards. The reward for each episode and a running mean of the last 30 episodes are logged to file. Basic statistics for the Q-values in the replay-memory are also logged to file before each optimization run.

This could be logged using TensorFlow and TensorBoard, but they were designed for logging variables of the TensorFlow graph and data that flows through the graph. In this case the data we want logged does not reside in the graph, so it becomes a bit awkward to use TensorFlow to log this data.

We have therefore implemented a few small classes that can write and read these logs.

```
log_q_values = rl.LogQValues()
log_reward = rl.LogReward()
```

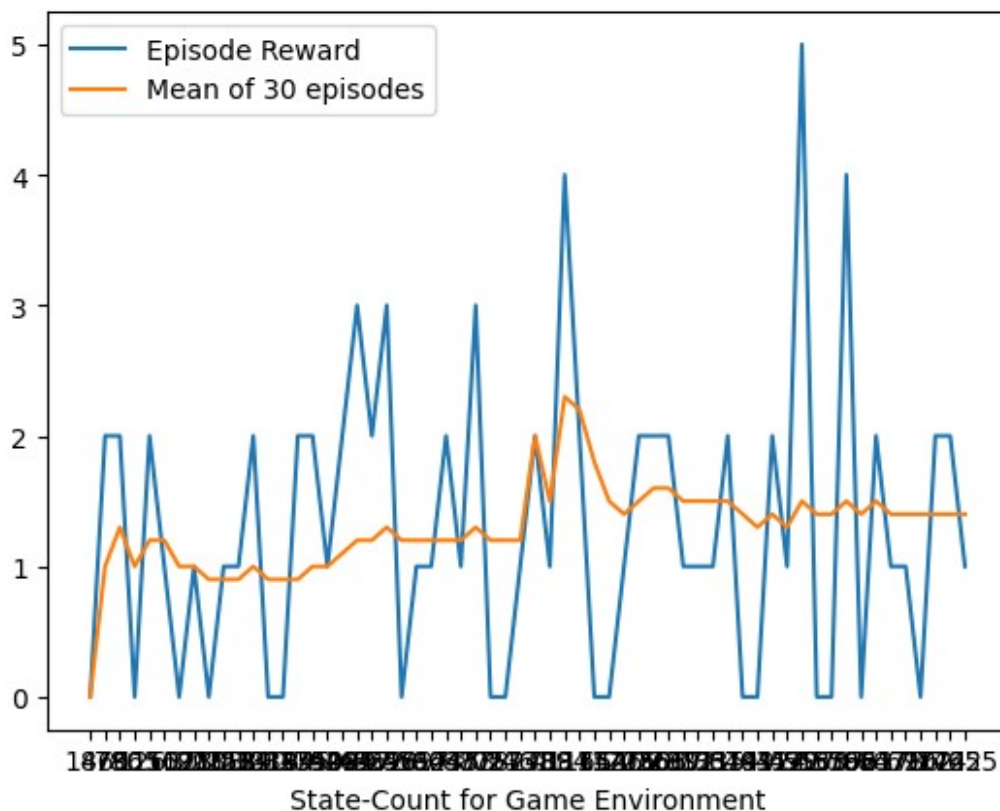
We can now read the logs from file:

```
log_reward.read()
```

Training Progress: Reward

This plot shows the reward for each episode during training, as well as the running mean of the last 30 episodes. Note how the reward varies greatly from one episode to the next, so it is difficult to say from this plot alone whether the agent is really improving during the training, although the running mean does appear to trend upwards slightly.

```
plt.plot(log_reward.count_states, log_reward.episode, label='Episode  
Reward')  
plt.plot(log_reward.count_states, log_reward.mean, label='Mean of 30  
episodes')  
plt.xlabel('State-Count for Game Environment')  
plt.legend()  
plt.show()
```



Training Progress: Q-Values

The following plot shows the mean Q-values from the replay-memory prior to each run of the optimizer for the Neural Network. Note how the mean Q-values increase rapidly in the beginning and then they increase fairly steadily for 40 million states, after which they still trend upwards but somewhat more irregularly.

The fast improvement in the beginning is probably due to (1) the use of a smaller replay-memory early in training so the Neural Network is optimized more often and the new information is used

faster, (2) the backwards-sweeping of the replay-memory so the rewards are used to update the Q-values for many of the states, instead of just updating the Q-values for a single state, and (3) the replay-memory is balanced so at least half of each mini-batch contains states whose Q-values have high estimation-errors for the Neural Network.

The [original paper from DeepMind](#) showed much slower progress in the first phase of training, see Figure 2 in that paper but note that the Q-values are not directly comparable, possibly because they used a higher discount factor of 0.99 while we only used 0.97 here.

Testing

When the agent and Neural Network is being trained, the so-called epsilon-probability is typically decreased from 1.0 to 0.1 over a large number of steps, after which the probability is held fixed at 0.1. This means the probability is 0.1 or 10% that the agent will select a random action in each step, otherwise it will select the action that has the highest Q-value. This is known as the epsilon-greedy policy. The choice of 0.1 for the epsilon-probability is a compromise between taking the actions that are already known to be good, versus exploring new actions that might lead to even higher rewards or might lead to death of the agent.

During testing it is common to lower the epsilon-probability even further. We have set it to 0.01 as shown here:

```
agent.epsilon_greedy.epsilon_testing  
0.01
```

We will now instruct the agent that it should no longer perform training by setting this boolean:

```
agent.training = False
```

We also reset the previous episode rewards.

```
agent.reset_episode_rewards()
```

We can render the game-environment to screen so we can see the agent playing the game, by setting this boolean:

```
agent.render = True
```

We can now run a single episode by calling the `run()` function again. This should open a new window that shows the game being played by the agent. At the time of this writing, it was not possible to resize this tiny window, and the developers at OpenAI did not seem to care about this feature which should obviously be there.

Mean Reward

The game-play is slightly random, both with regard to selecting actions using the epsilon-greedy policy, but also because the OpenAI Gym environment will repeat any action between 2-

4 times, with the number chosen at random. So the reward of one episode is not an accurate estimate of the reward that can be expected in general from this agent.

We need to run 30 or even 50 episodes to get a more accurate estimate of the reward that can be expected.

We will first reset the previous episode rewards.

```
agent.reset_episode_rewards()
```

We disable the screen-rendering so the game-environment runs much faster.

```
agent.render = False
```

We can now run 5 episodes. This records the rewards for each episode. It might have been a good idea to disable the output so it does not print all these lines - you can do this as an exercise.

```
agent.run(num_episodes=5)
```

32:8032	Q-min: 0.001	Q-max: 0.009	Lives: 5	Reward: 1.0
Episode Mean: 0.0				
32:8081	Q-min: 0.003	Q-max: 0.009	Lives: 5	Reward: 2.0
Episode Mean: 0.0				
32:8108	Q-min: 0.002	Q-max: 0.010	Lives: 4	Reward: 2.0
Episode Mean: 0.0				
32:9217	Q-min: 0.002	Q-max: 0.010	Lives: 3	Reward: 2.0
Episode Mean: 0.0				
32:9722	Q-min: 0.003	Q-max: 0.011	Lives: 3	Reward: 3.0
Episode Mean: 0.0				
32:9764	Q-min: 0.002	Q-max: 0.010	Lives: 2	Reward: 3.0
Episode Mean: 0.0				
32:9986	Q-min: 0.002	Q-max: 0.010	Lives: 1	Reward: 3.0
Episode Mean: 0.0				
32:10997	Q-min: 0.002	Q-max: 0.009	Lives: 0	Reward: 3.0
Episode Mean: 0.0				
32:10998	Q-min: 0.002	Q-max: 0.010	Lives: 0	Reward: 3.0
Episode Mean: 3.0				
33:11079	Q-min: 0.001	Q-max: 0.008	Lives: 5	Reward: 1.0
Episode Mean: 3.0				
33:11128	Q-min: 0.002	Q-max: 0.008	Lives: 5	Reward: 2.0
Episode Mean: 3.0				
33:11154	Q-min: 0.002	Q-max: 0.010	Lives: 4	Reward: 2.0
Episode Mean: 3.0				
33:11551	Q-min: 0.002	Q-max: 0.010	Lives: 4	Reward: 3.0
Episode Mean: 3.0				
33:11593	Q-min: 0.002	Q-max: 0.010	Lives: 3	Reward: 3.0
Episode Mean: 3.0				
33:11720	Q-min: 0.002	Q-max: 0.010	Lives: 2	Reward: 3.0
Episode Mean: 3.0				
33:12233	Q-min: 0.003	Q-max: 0.010	Lives: 2	Reward: 4.0

Episode Mean: 3.0				
33:12276	Q-min: 0.002	Q-max: 0.010	Lives: 1	Reward: 4.0
Episode Mean: 3.0				
33:12581	Q-min: 0.002	Q-max: 0.010	Lives: 0	Reward: 4.0
Episode Mean: 3.0				
33:12582	Q-min: 0.002	Q-max: 0.010	Lives: 0	Reward: 4.0
Episode Mean: 3.5				
34:12795	Q-min: 0.001	Q-max: 0.009	Lives: 5	Reward: 1.0
Episode Mean: 3.5				
34:12845	Q-min: 0.002	Q-max: 0.010	Lives: 5	Reward: 2.0
Episode Mean: 3.5				
34:12871	Q-min: 0.002	Q-max: 0.010	Lives: 4	Reward: 2.0
Episode Mean: 3.5				
34:12920	Q-min: 0.002	Q-max: 0.010	Lives: 3	Reward: 2.0
Episode Mean: 3.5				
34:13239	Q-min: 0.002	Q-max: 0.010	Lives: 2	Reward: 2.0
Episode Mean: 3.5				
34:14654	Q-min: 0.002	Q-max: 0.010	Lives: 2	Reward: 3.0
Episode Mean: 3.5				
34:14692	Q-min: 0.002	Q-max: 0.010	Lives: 1	Reward: 3.0
Episode Mean: 3.5				
34:15621	Q-min: 0.002	Q-max: 0.010	Lives: 0	Reward: 3.0
Episode Mean: 3.5				
34:15622	Q-min: 0.002	Q-max: 0.010	Lives: 0	Reward: 3.0
Episode Mean: 3.3				
35:16132	Q-min: 0.001	Q-max: 0.009	Lives: 4	Reward: 0.0
Episode Mean: 3.3				
35:16457	Q-min: 0.002	Q-max: 0.009	Lives: 3	Reward: 0.0
Episode Mean: 3.3				
35:16529	Q-min: 0.001	Q-max: 0.009	Lives: 2	Reward: 0.0
Episode Mean: 3.3				
35:17058	Q-min: 0.001	Q-max: 0.010	Lives: 1	Reward: 0.0
Episode Mean: 3.3				
35:17108	Q-min: 0.001	Q-max: 0.010	Lives: 0	Reward: 0.0
Episode Mean: 3.3				
35:17109	Q-min: 0.001	Q-max: 0.010	Lives: 0	Reward: 0.0
Episode Mean: 2.5				
36:17654	Q-min: 0.001	Q-max: 0.009	Lives: 4	Reward: 0.0
Episode Mean: 2.5				
36:17784	Q-min: 0.002	Q-max: 0.009	Lives: 3	Reward: 0.0
Episode Mean: 2.5				
36:18080	Q-min: 0.001	Q-max: 0.010	Lives: 2	Reward: 0.0
Episode Mean: 2.5				
36:18555	Q-min: 0.001	Q-max: 0.010	Lives: 1	Reward: 0.0
Episode Mean: 2.5				
36:18723	Q-min: 0.001	Q-max: 0.009	Lives: 0	Reward: 0.0
Episode Mean: 2.5				
36:18724	Q-min: 0.001	Q-max: 0.009	Lives: 0	Reward: 0.0
Episode Mean: 2.0				

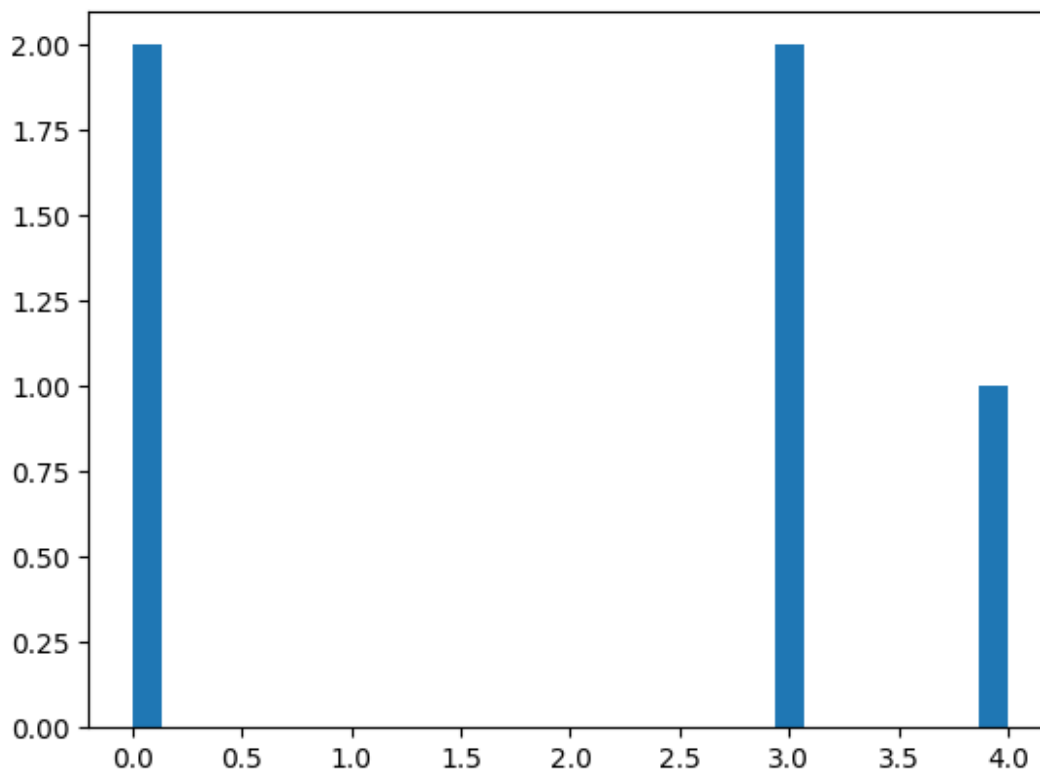
We can now print some statistics for the episode rewards, which vary greatly from one episode to the next.

```
rewards = agent.episode_rewards
print("Rewards for {0} episodes:".format(len(rewards)))
print("- Min:    ", np.min(rewards))
print("- Mean:    ", np.mean(rewards))
print("- Max:     ", np.max(rewards))
print("- Stdev:   ", np.std(rewards))
```

```
Rewards for 5 episodes:
- Min:    0.0
- Mean:   2.0
- Max:    4.0
- Stdev:  1.6733200530681511
```

We can also plot a histogram with the episode rewards.

```
_ = plt.hist(rewards, bins=30)
```



Example States

We can plot examples of states from the game-environment and the Q-values that are estimated by the Neural Network.

This helper-function prints the Q-values for a given index in the replay-memory.

```
def print_q_values(idx):
    """Print Q-values and actions from the replay-memory at the given
    index."""

    # Get the Q-values and action from the replay-memory.
    q_values = replay_memory.q_values[idx]
    action = replay_memory.actions[idx]

    print("Action:      Q-Value:")
    print("=====")

    # Print all the actions and their Q-values.
    for i, q_value in enumerate(q_values):
        # Used to display which action was taken.
        if i == action:
            action_taken = "(Action Taken)"
        else:
            action_taken = ""

        # Text-name of the action.
        action_name = agent.get_action_name(i)

        print("{0:12}{1:.3f} {2}".format(action_name, q_value,
                                          action_taken))

    # Newline.
    print()
```

This helper-function plots a state from the replay-memory and optionally prints the Q-values.

```
def plot_state(idx, print_q=True):
    """Plot the state in the replay-memory with the given index."""

    # Get the state from the replay-memory.
    state = replay_memory.states[idx]

    # Create figure with a grid of sub-plots.
    fig, axes = plt.subplots(1, 2)

    # Plot the image from the game-environment.
    ax = axes.flat[0]
    ax.imshow(state[:, :, 0], vmin=0, vmax=255,
              interpolation='lanczos', cmap='gray')

    # Plot the motion-trace.
    ax = axes.flat[1]
    ax.imshow(state[:, :, 1], vmin=0, vmax=255,
              interpolation='lanczos', cmap='gray')
```

```

    # This is necessary if we show more than one plot in a single
    # Notebook cell.
    plt.show()

    # Print the Q-values.
    if print_q:
        print_q_values(idx=idx)

```

The replay-memory has room for 200k states but it is only partially full from the above call to `agent.run(num_episodes=1)`. This is how many states are actually used.

```

num_used = replay_memory.num_used
num_used

```

7656

Get the Q-values from the replay-memory that are actually used.

```

q_values = replay_memory.q_values[0:num_used, :]

```

For each state, calculate the min / max Q-values and their difference. This will be used to lookup interesting states in the following sections.

```

q_values_min = q_values.min(axis=1)
q_values_max = q_values.max(axis=1)
q_values_dif = q_values_max - q_values_min

```

Example States: Highest Reward

This example shows the states surrounding the state with the highest reward.

During the training we limit the rewards to the range $[-1, 1]$ so this basically just gets the first state that has a reward of 1.

```

idx = np.argmax(replay_memory.rewards)
idx

```

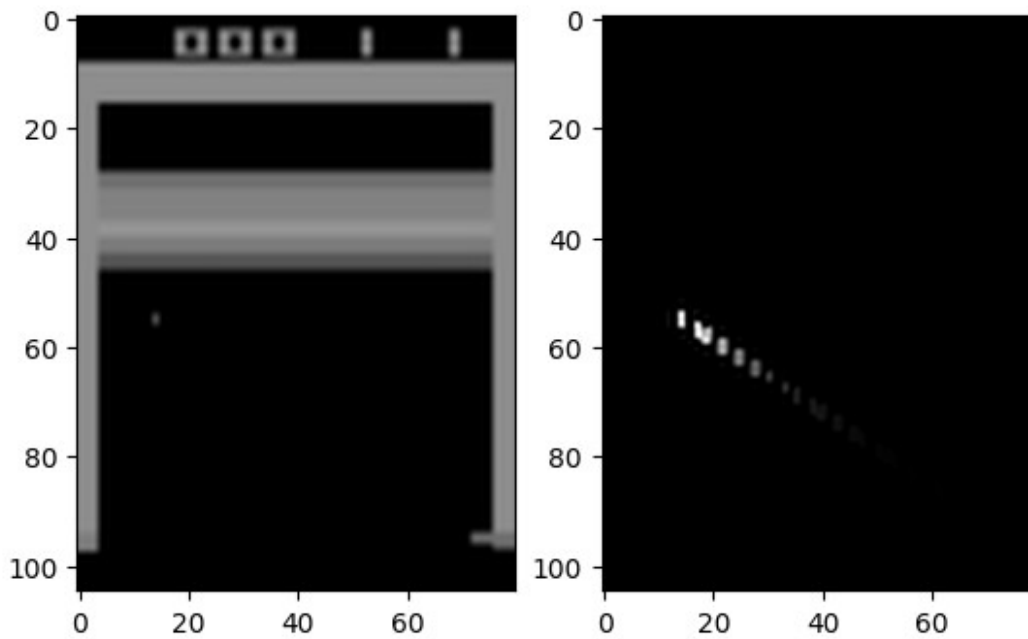
186

This state is where the ball hits the wall so the agent scores a point.

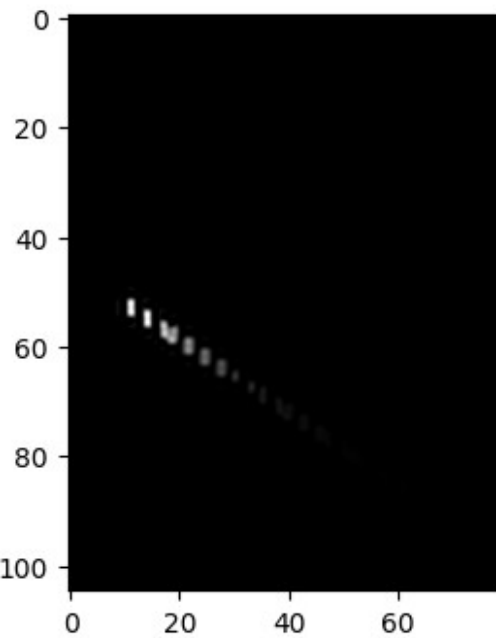
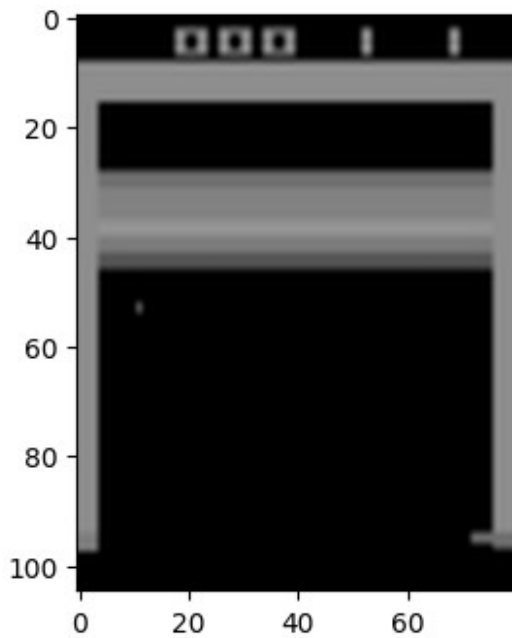
We can show the surrounding states leading up to and following this state. Note how the Q-values are very close for the different actions, because at this point it really does not matter what the agent does as the reward is already guaranteed. But note how the Q-values decrease significantly after the ball has hit the wall and a point has been scored.

Also note that the agent uses the Epsilon-greedy policy for taking actions, so there is a small probability that a random action is taken instead of the action with the highest Q-value.

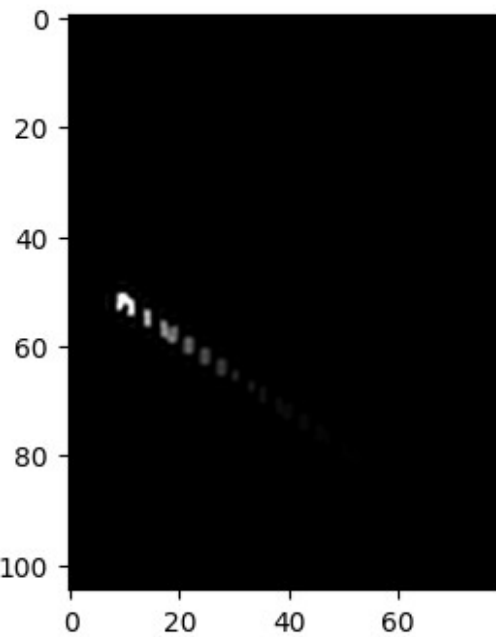
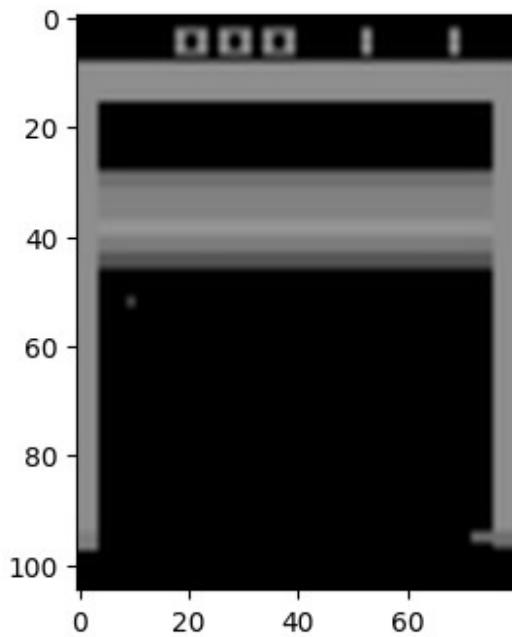
```
for i in range(-5, 3):
    plot_state(idx=idx+i)
```



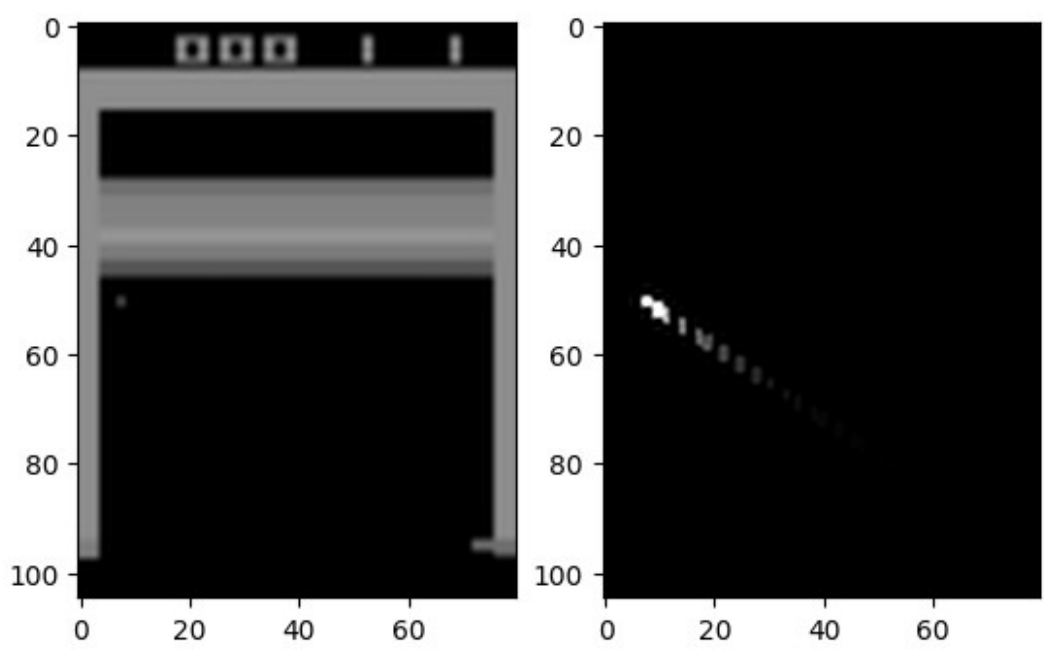
Action:	Q-Value:
NOOP	0.003
FIRE	0.000
RIGHT	0.010 (Action Taken)
LEFT	0.002



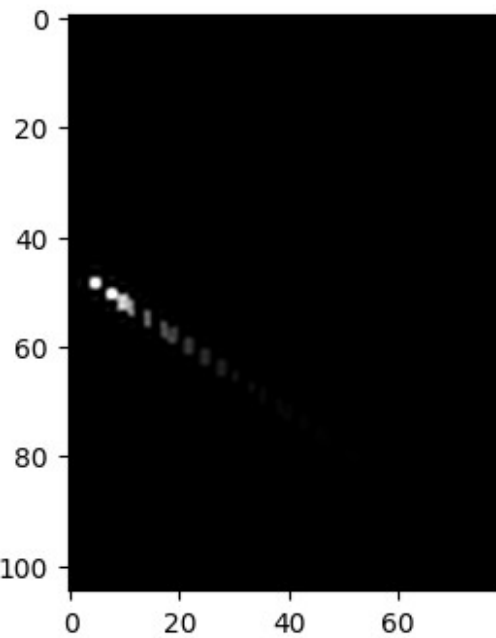
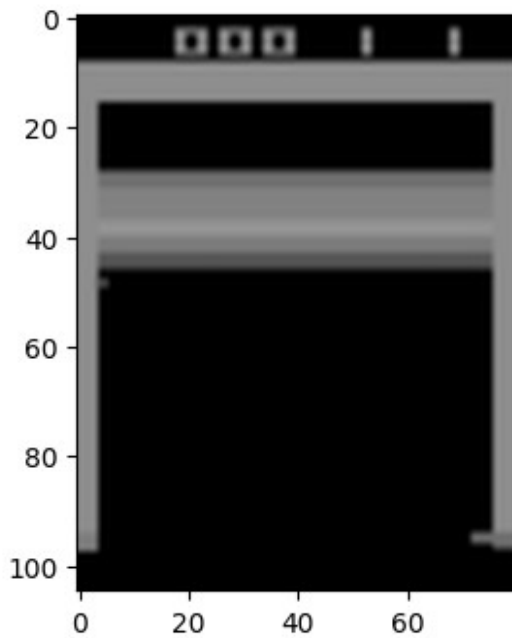
Action:	Q-Value:
=====	
NOOP	0.004 (Action Taken)
FIRE	0.001
RIGHT	0.010
LEFT	0.002



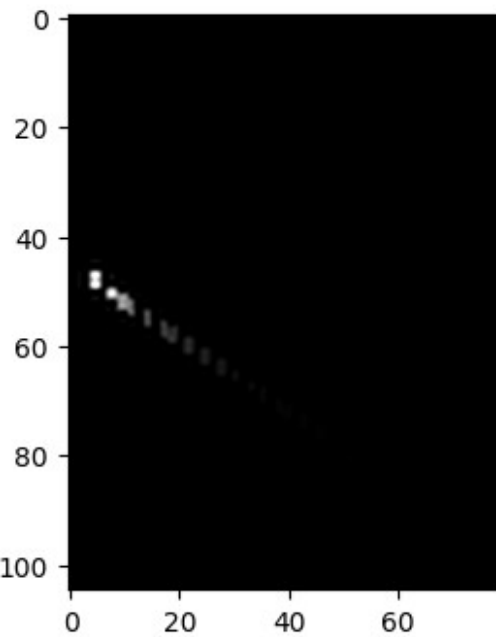
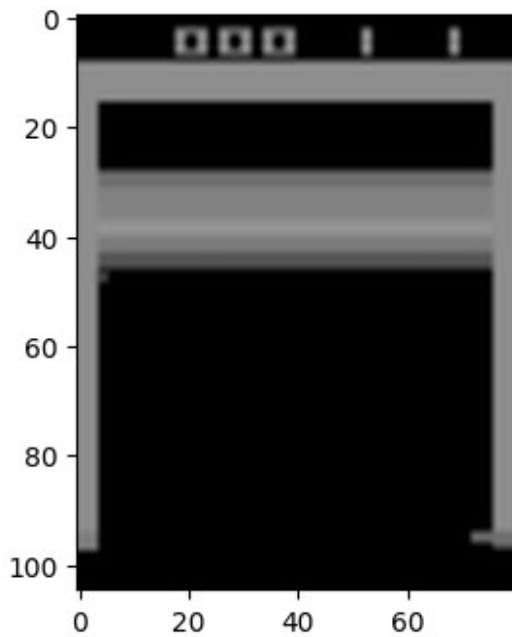
Action:	Q-Value:
=====	
N00P	0.004 (Action Taken)
FIRE	0.002
RIGHT	0.009
LEFT	0.002



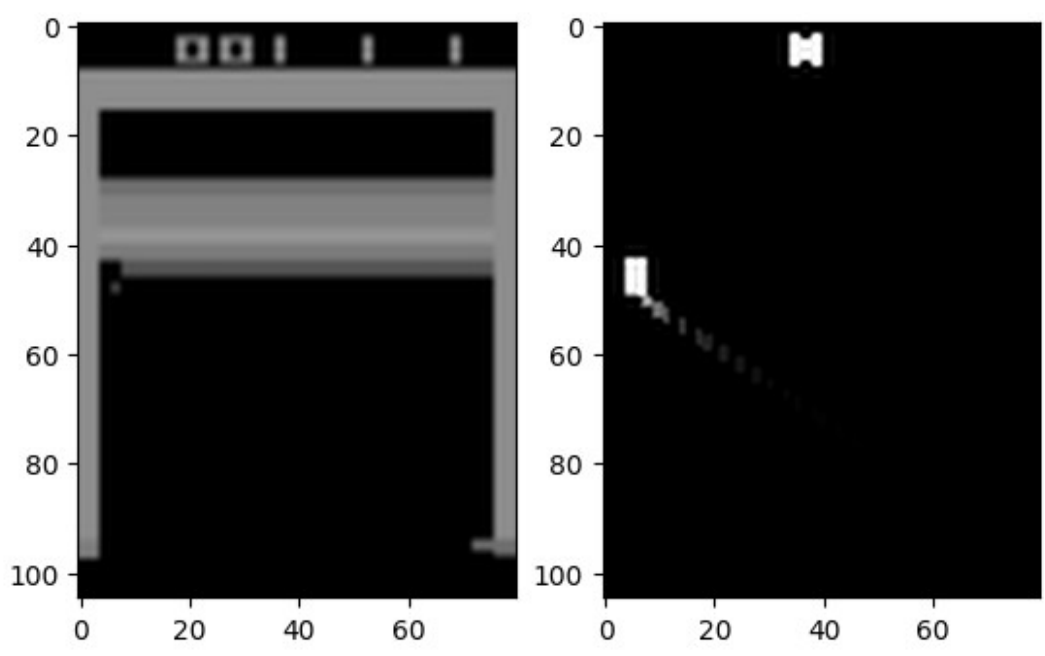
Action:	Q-Value:
=====	
N00P	0.005
FIRE	0.003 (Action Taken)
RIGHT	0.009
LEFT	0.000



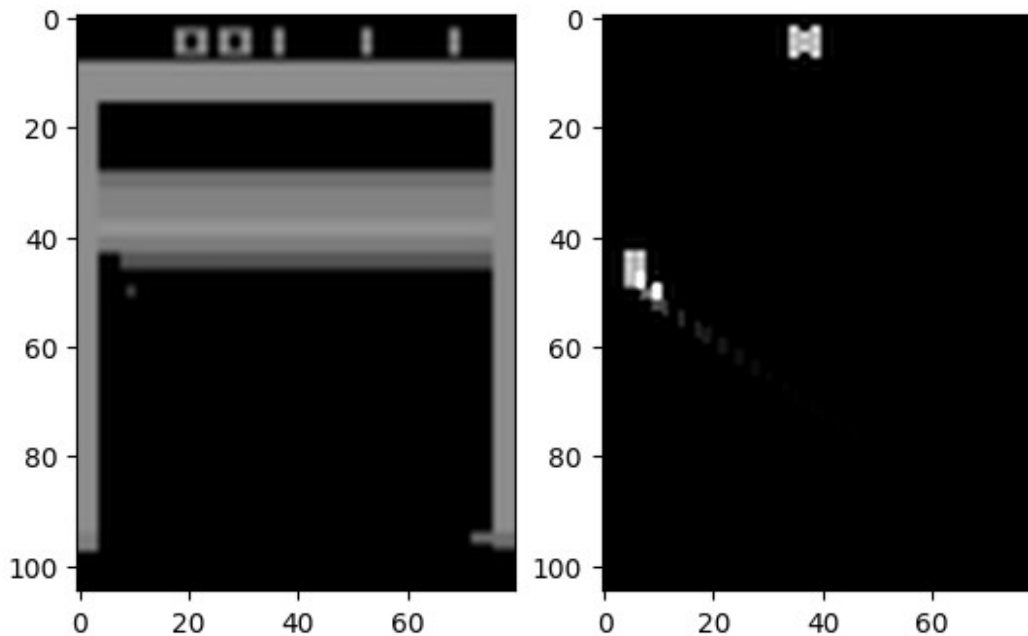
Action:	Q-Value:
NOOP	0.004
FIRE	0.003 (Action Taken)
RIGHT	0.010
LEFT	0.001



Action:	Q-Value:
=====	
NOOP	0.004 (Action Taken)
FIRE	0.004
RIGHT	0.010
LEFT	0.001



Action:	Q-Value:
=====	
NOOP	0.004
FIRE	0.004
RIGHT	0.010
LEFT	0.002 (Action Taken)

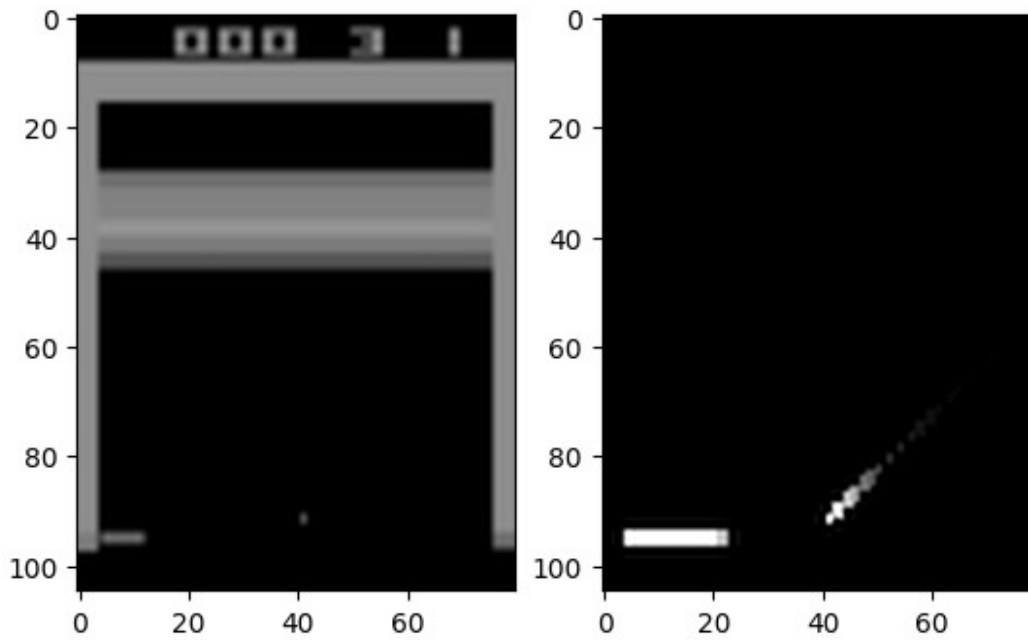


Action:	Q-Value:
=====	
N00P	0.005 (Action Taken)
FIRE	0.003
RIGHT	0.010
LEFT	0.004

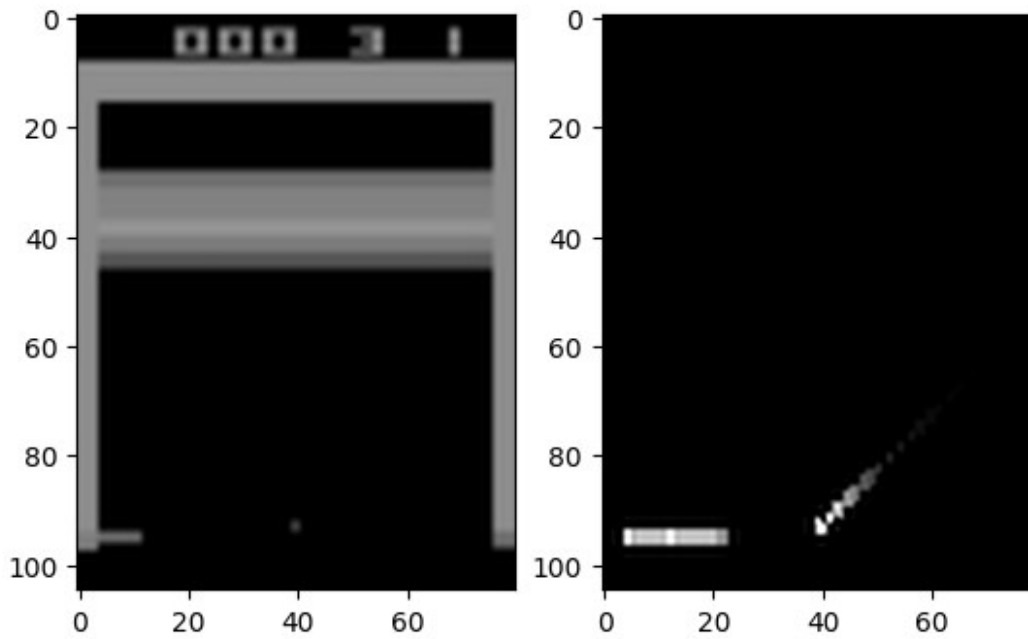
Example: Highest Q-Value

This example shows the states surrounding the one with the highest Q-values. This means that the agent has high expectation that several points will be scored in the following steps. Note that the Q-values decrease significantly after the points have been scored.

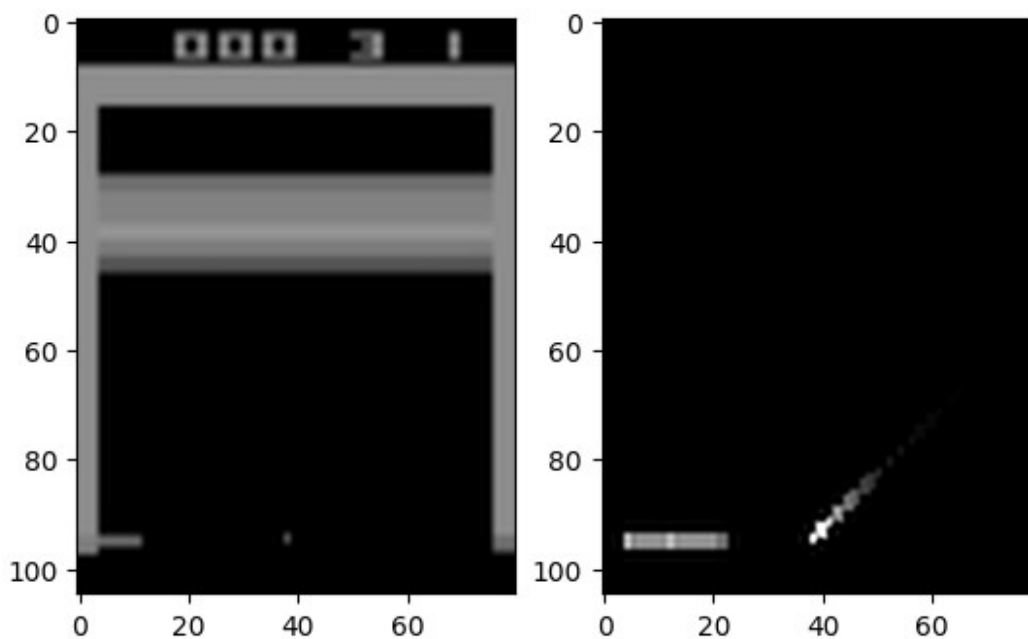
```
idx = np.argmax(q_values_max)
idx
7505
for i in range(0, 5):
    plot_state(idx=idx+i)
```

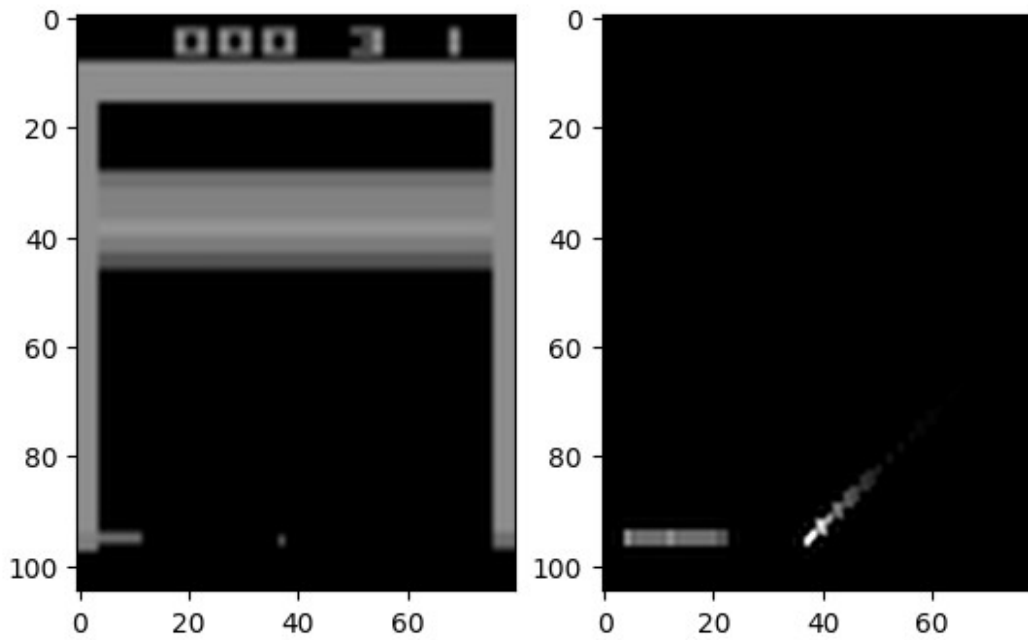
Action:	Q-Value:
=====	
NOOP	0.006 (Action Taken)
FIRE	0.004
RIGHT	0.019
LEFT	0.001



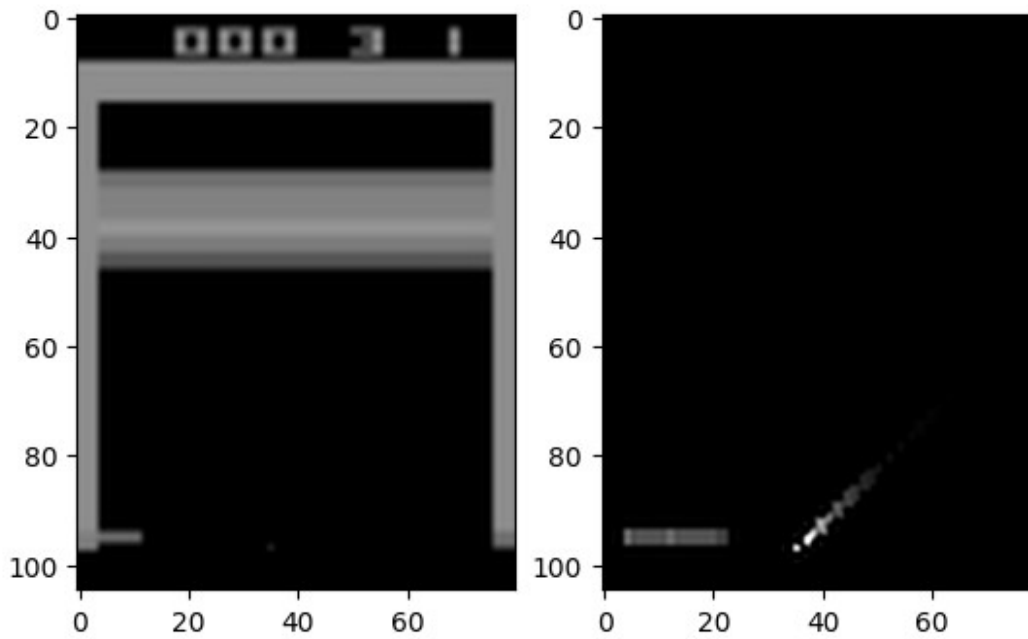
Action:	Q-Value:
=====	=====
NOOP	0.007
FIRE	0.004
RIGHT	0.017
LEFT	0.001 (Action Taken)



Action:	Q-Value:
=====	=====
NOOP	0.006
FIRE	0.003 (Action Taken)
RIGHT	0.015
LEFT	0.001



Action:	Q-Value:
NOOP	0.006
FIRE	0.003
RIGHT	0.014
LEFT	0.001 (Action Taken)

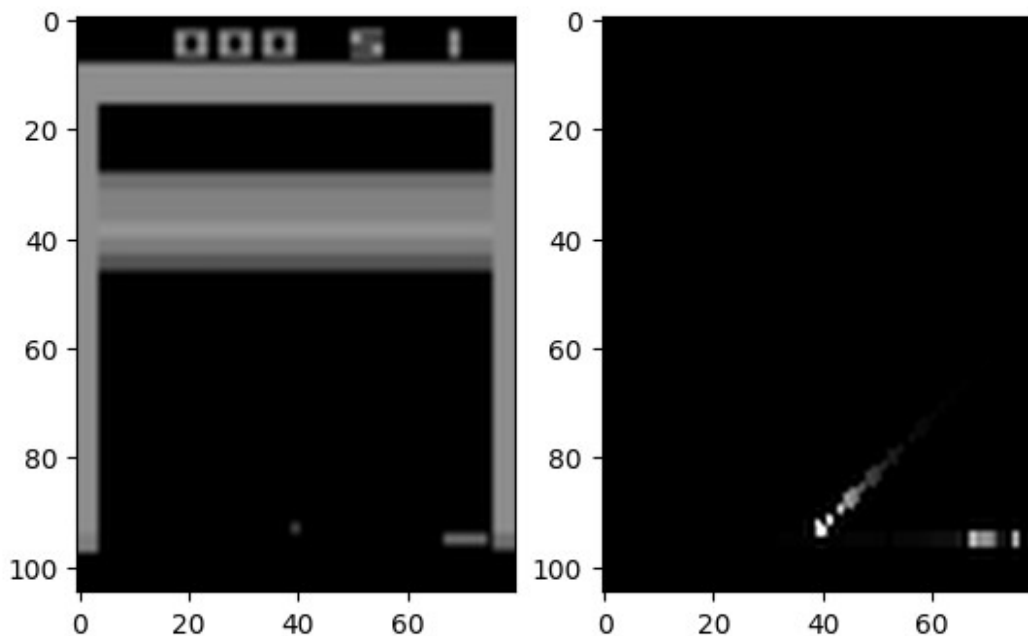


Action:	Q-Value:
=====	=====
N00P	0.006
FIRE	0.002
RIGHT	0.012
LEFT	0.001 (Action Taken)

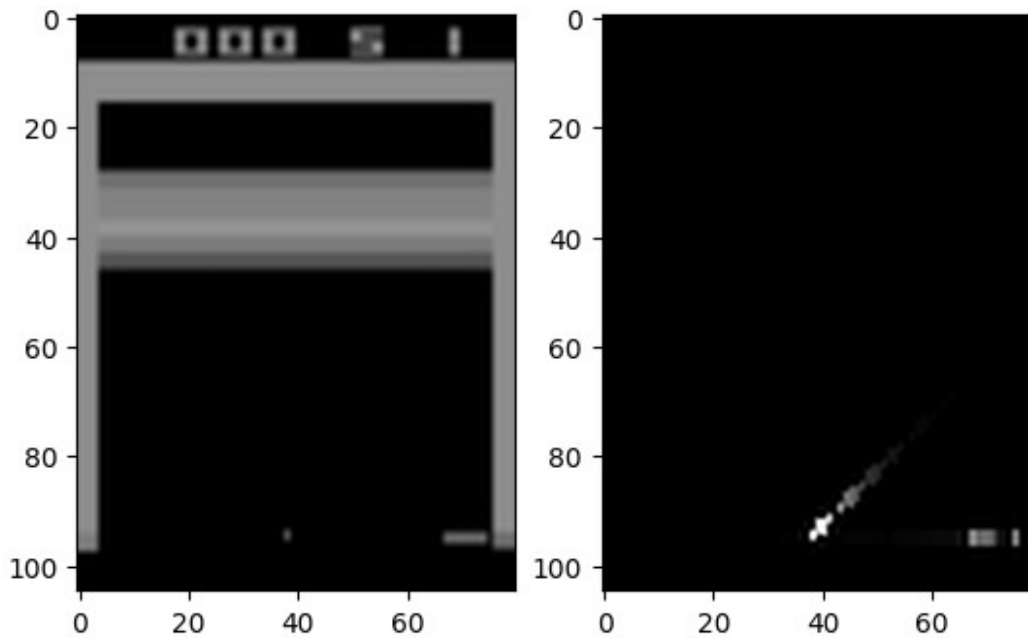
Example: Loss of Life

This example shows the states leading up to a loss of life for the agent.

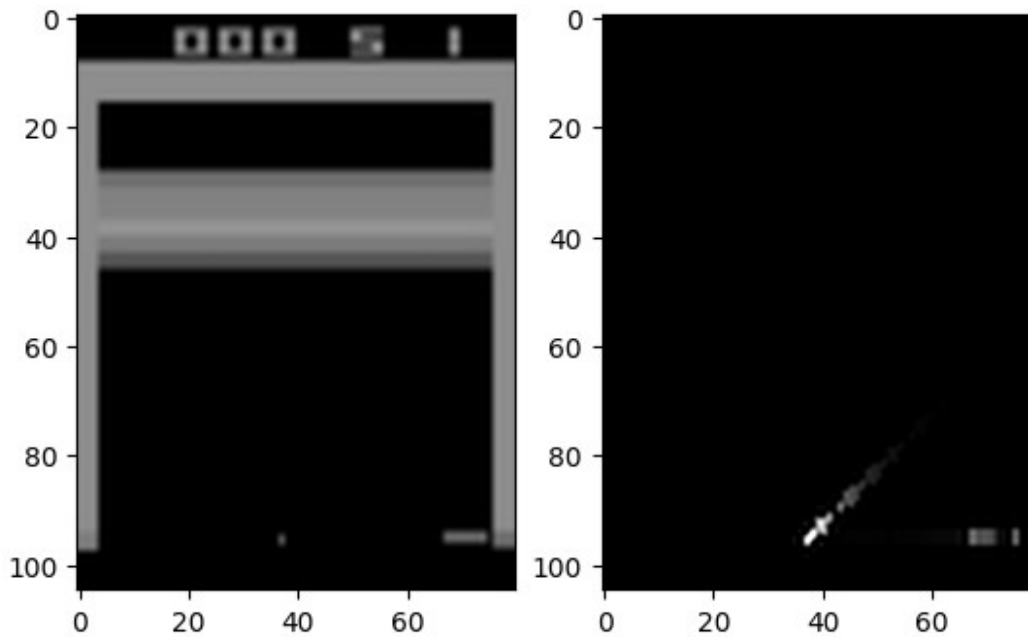
```
idx = np.argmax(replay_memory.end_life)
idx
34
for i in range(-10, 0):
    plot_state(idx=idx+i)
```



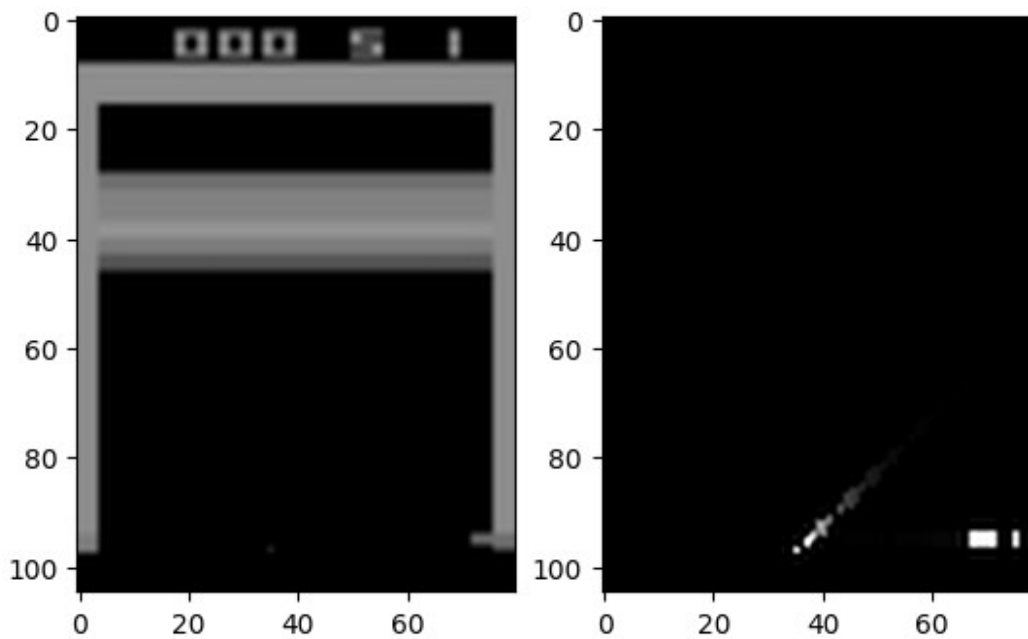
Action:	Q-Value:
=====	=====
N00P	0.006
FIRE	0.002 (Action Taken)
RIGHT	0.012
LEFT	0.001



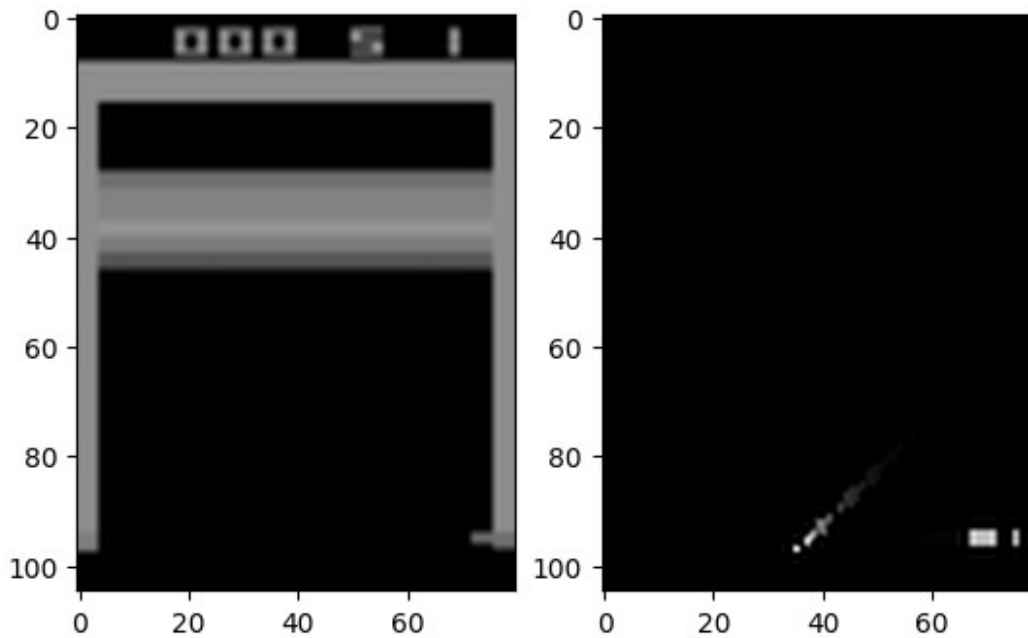
Action:	Q-Value:
=====	
NOOP	0.006 (Action Taken)
FIRE	0.002
RIGHT	0.012
LEFT	0.001



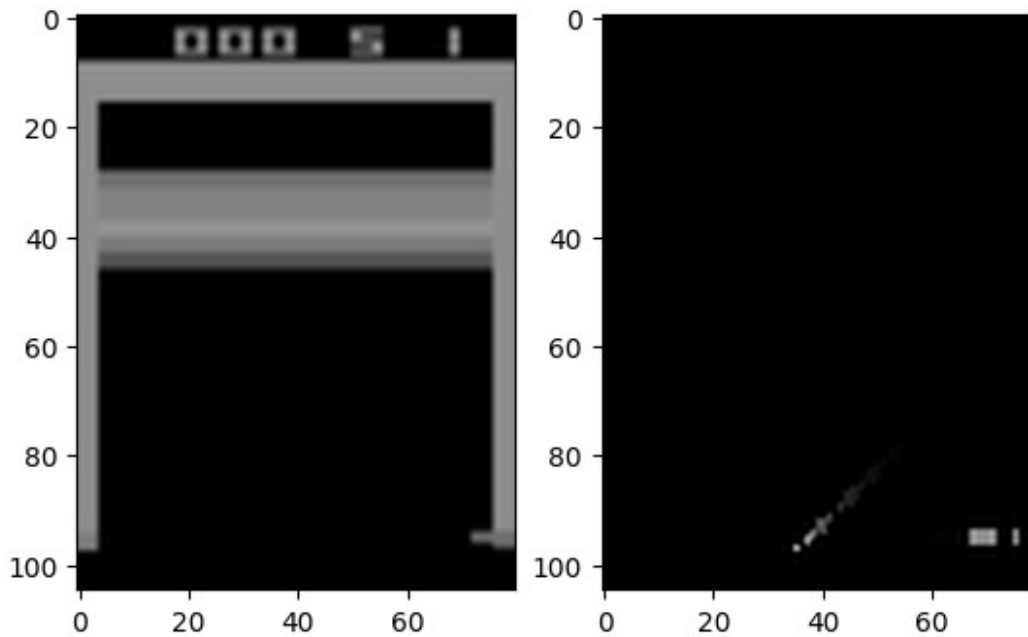
Action:	Q-Value:
=====	=====
NOOP	0.006
FIRE	0.002
RIGHT	0.011 (Action Taken)
LEFT	0.001



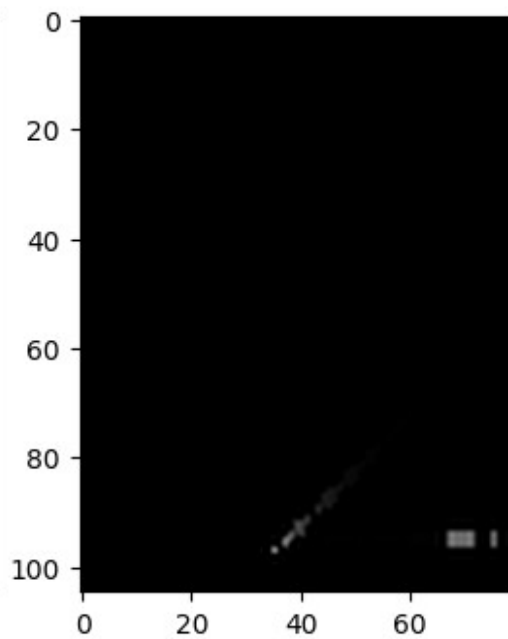
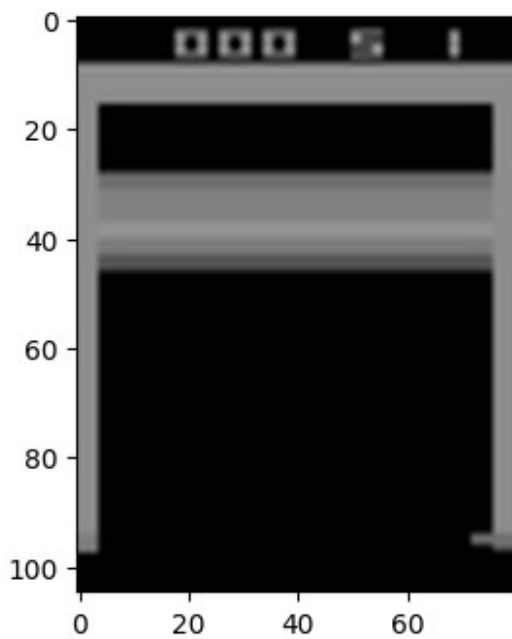
Action:	Q-Value:
=====	=====
NOOP	0.007
FIRE	0.002
RIGHT	0.010 (Action Taken)
LEFT	0.002



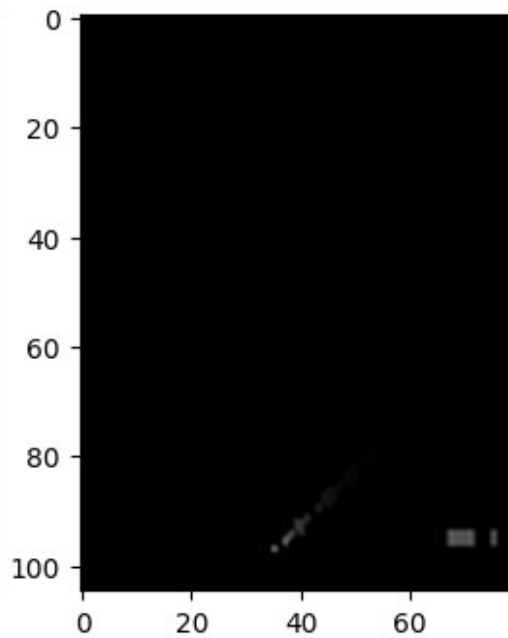
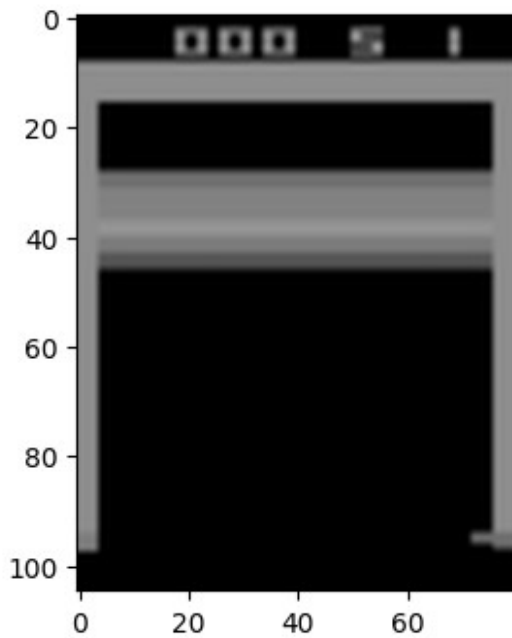
Action:	Q-Value:
NOOP	0.006
FIRE	0.001 (Action Taken)
RIGHT	0.010
LEFT	0.002



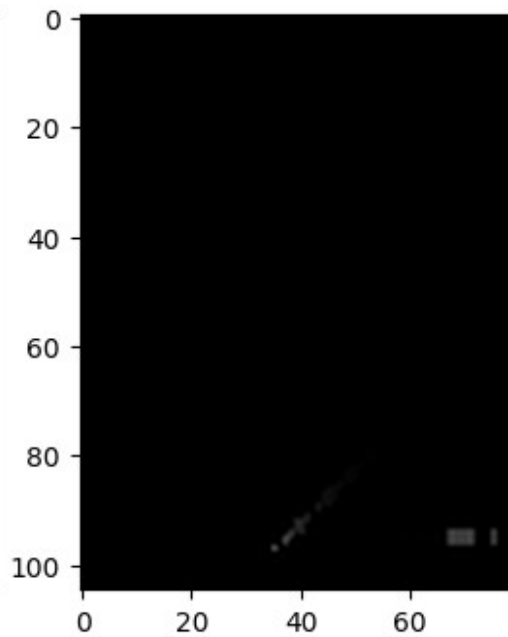
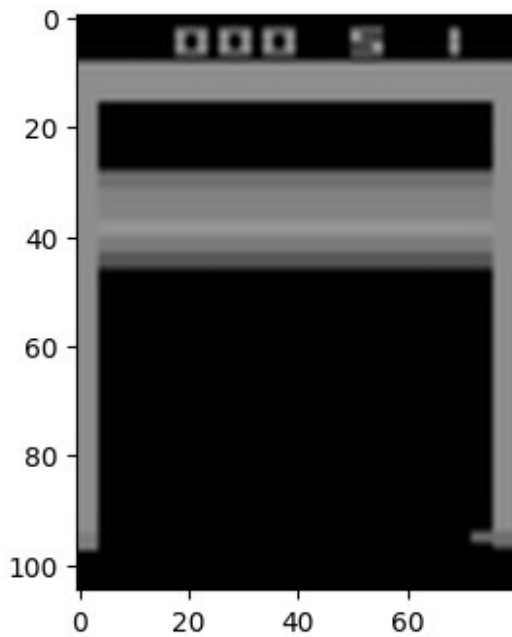
Action:	Q-Value:
=====	=====
NOOP	0.006
FIRE	0.001
RIGHT	0.010 (Action Taken)
LEFT	0.002



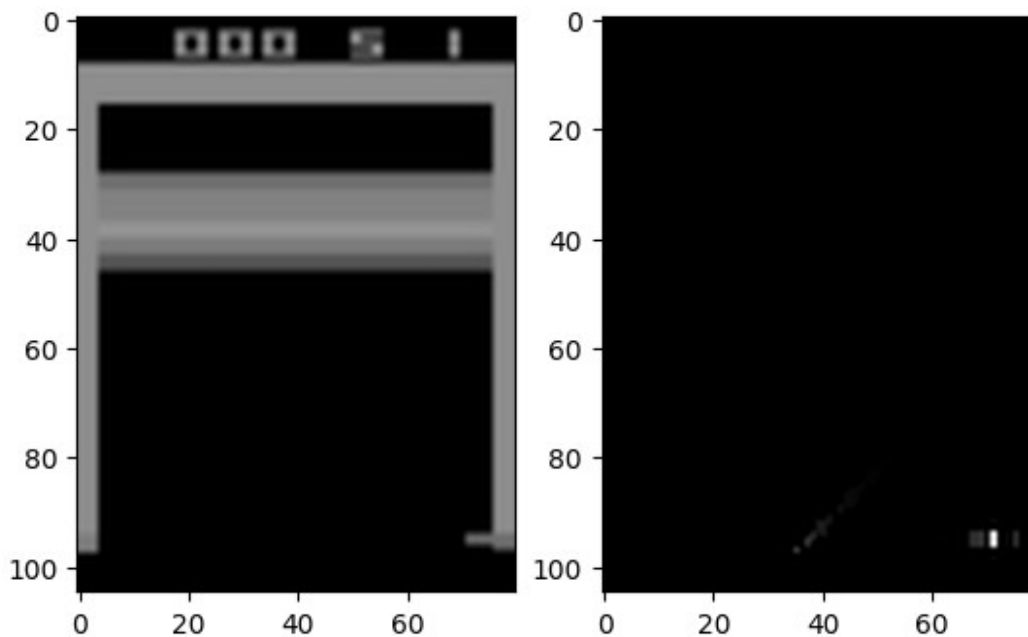
Action:	Q-Value:
=====	=====
NOOP	0.005
FIRE	0.001
RIGHT	0.010
LEFT	0.002 (Action Taken)



Action:	Q-Value:
=====	
NOOP	0.005
FIRE	0.001
RIGHT	0.010
LEFT	0.002 (Action Taken)



Action:	Q-Value:
=====	=====
NOOP	0.005
FIRE	0.001
RIGHT	0.010
LEFT	0.002 (Action Taken)



Action:	Q-Value:
=====	=====
NOOP	0.005
FIRE	0.002 (Action Taken)
RIGHT	0.010
LEFT	0.002

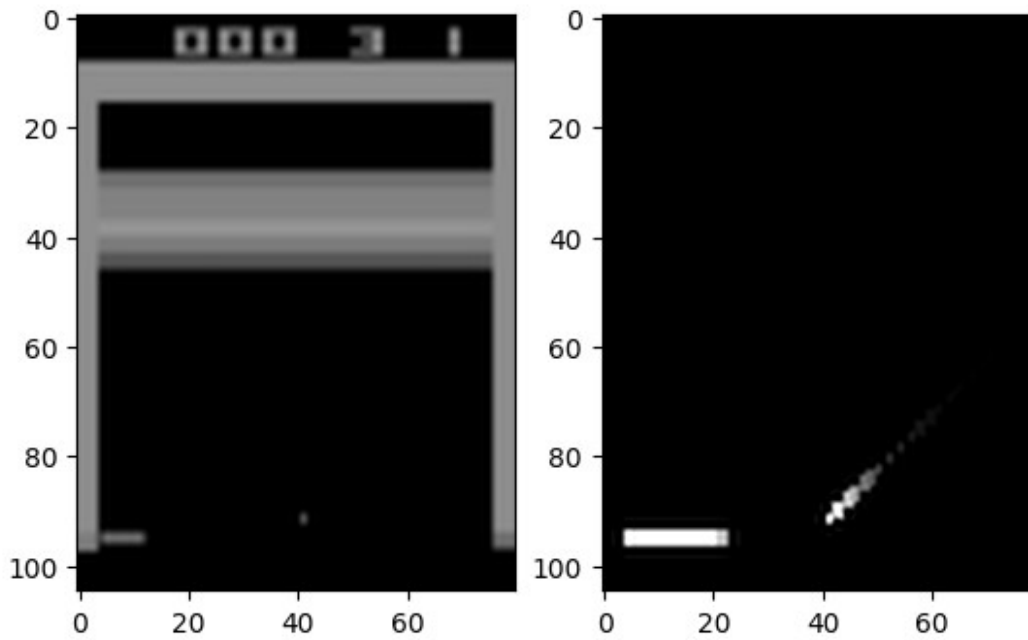
Example: Greatest Difference in Q-Values

This example shows the state where there is the greatest difference in Q-values, which means that the agent believes one action will be much more beneficial than another. But because the agent uses the Epsilon-greedy policy, it sometimes selects a random action instead.

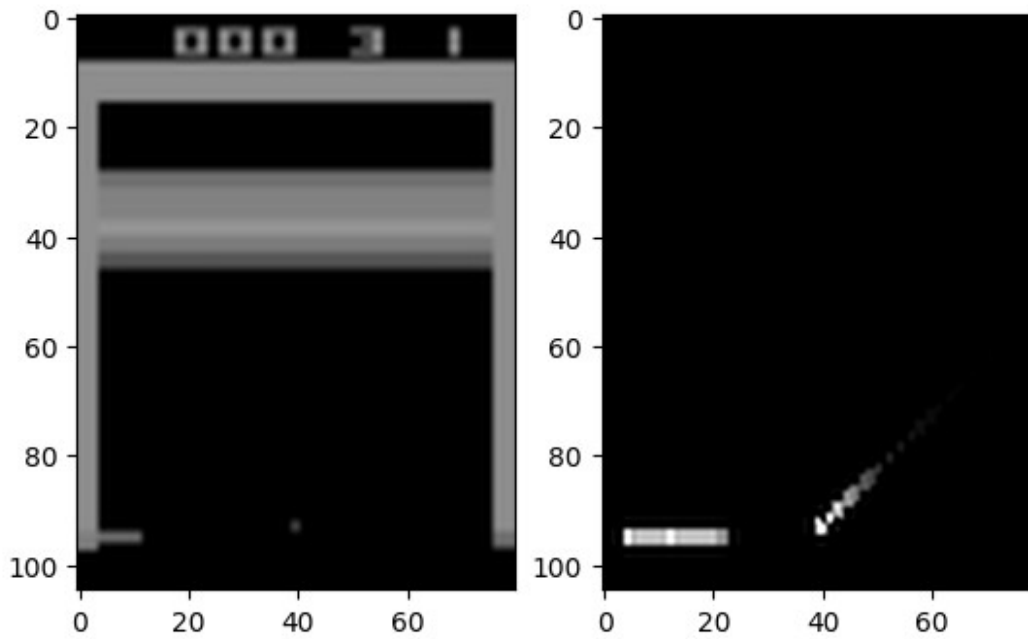
```
idx = np.argmax(q_values_dif)
idx

7505

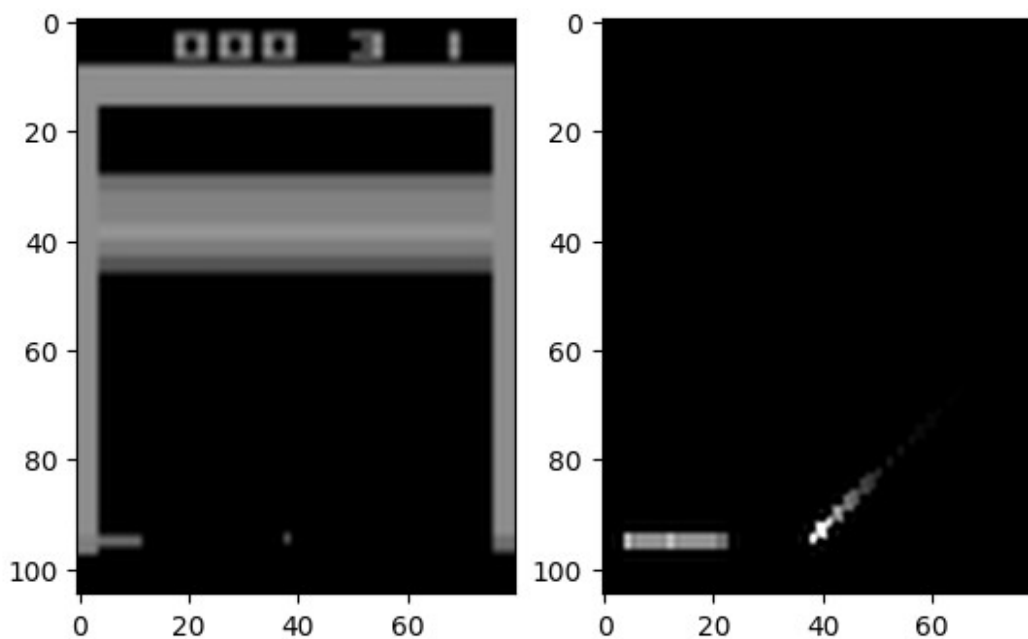
for i in range(0, 5):
    plot_state(idx=idx+i)
```



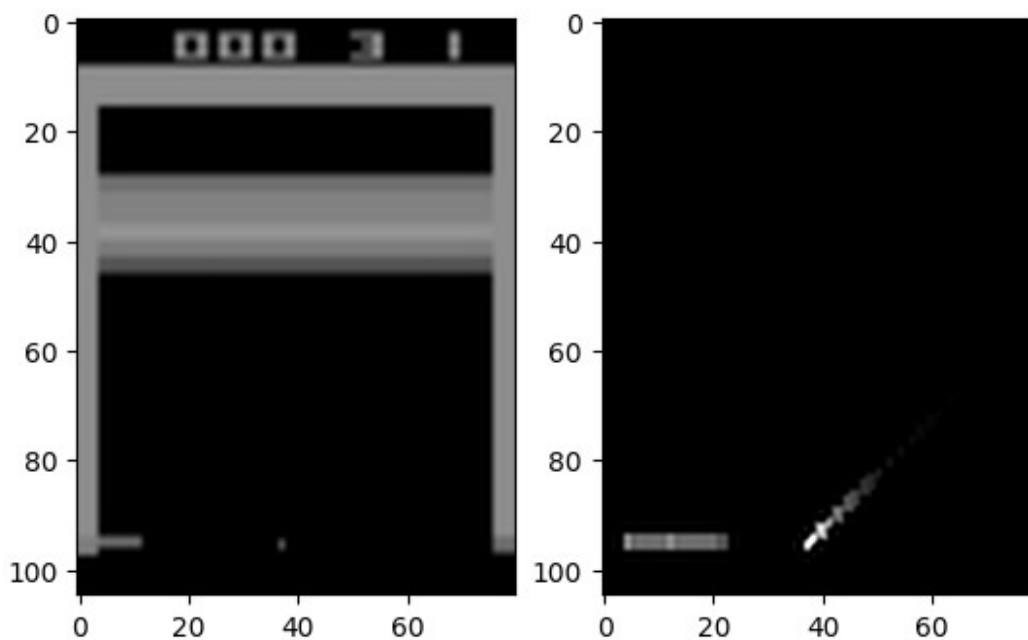
Action:	Q-Value:
=====	
NOOP	0.006 (Action Taken)
FIRE	0.004
RIGHT	0.019
LEFT	0.001



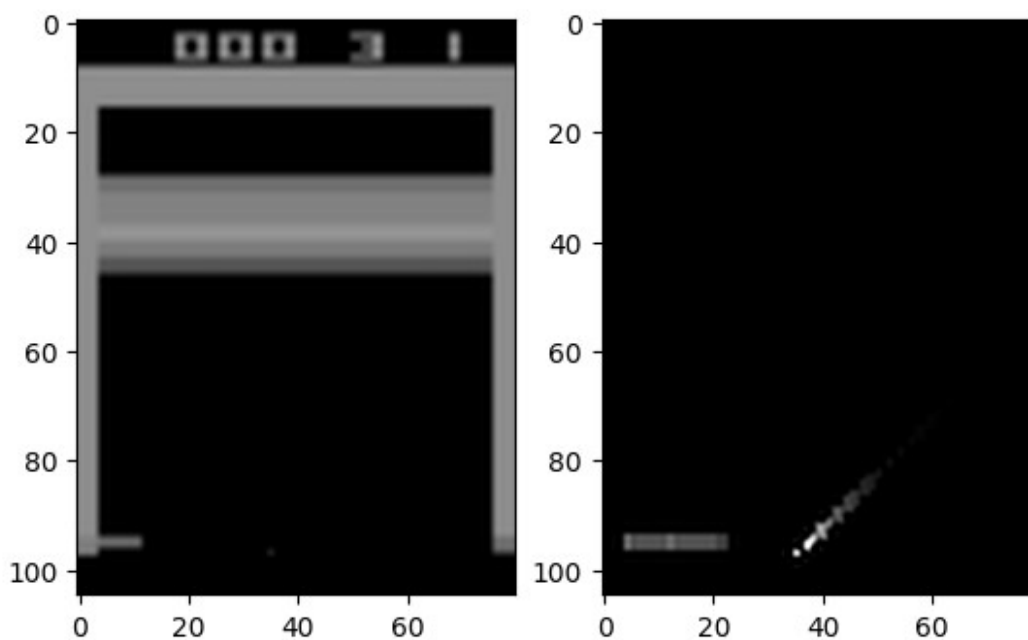
Action:	Q-Value:
=====	=====
NOOP	0.007
FIRE	0.004
RIGHT	0.017
LEFT	0.001 (Action Taken)



Action:	Q-Value:
=====	=====
NOOP	0.006
FIRE	0.003 (Action Taken)
RIGHT	0.015
LEFT	0.001



Action:	Q-Value:
NOOP	0.006
FIRE	0.003
RIGHT	0.014
LEFT	0.001 (Action Taken)



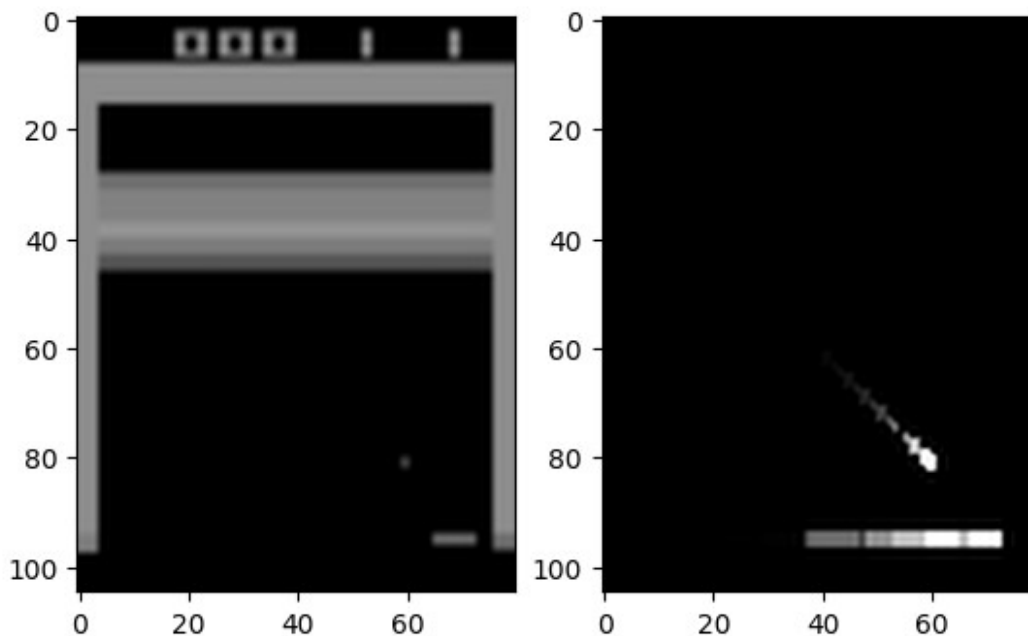
Action:	Q-Value:
=====	=====
NOOP	0.006
FIRE	0.002
RIGHT	0.012
LEFT	0.001 (Action Taken)

Example: Smallest Difference in Q-Values

This example shows the state where there is the smallest difference in Q-values, which means that the agent believes it does not really matter which action it selects, as they all have roughly the same expectations for future rewards.

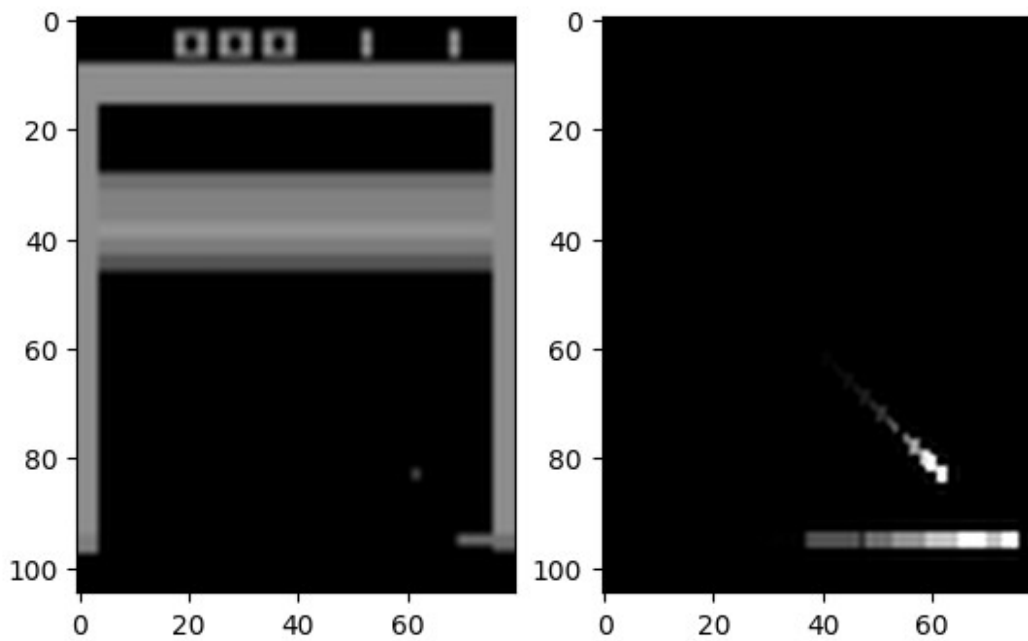
The Neural Network estimates these Q-values and they are not precise. The differences in Q-values may be so small that they fall within the error-range of the estimates.

```
idx = np.argmin(q_values_dif)
idx
3779
for i in range(0, 5):
    plot_state(idx=idx+i)
```

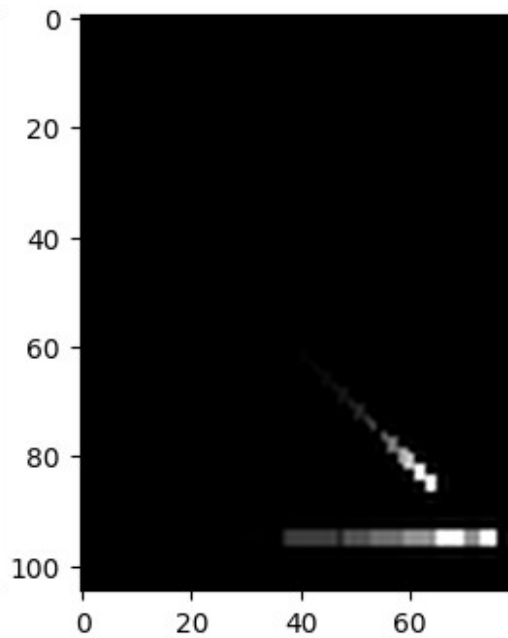
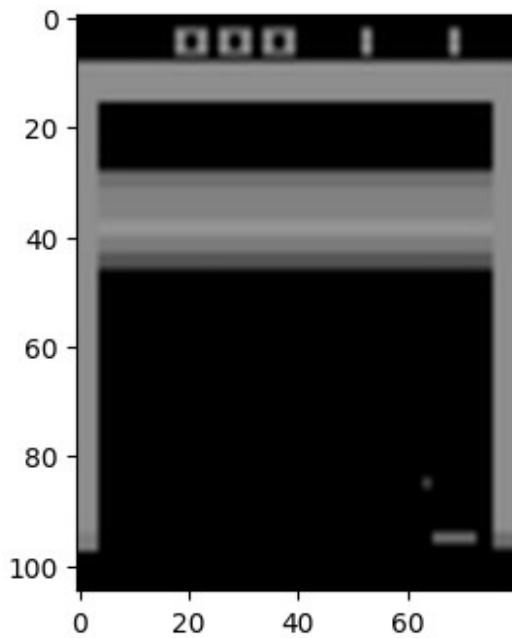


Action:	Q-Value:
=====	=====
NOOP	0.004
FIRE	0.005

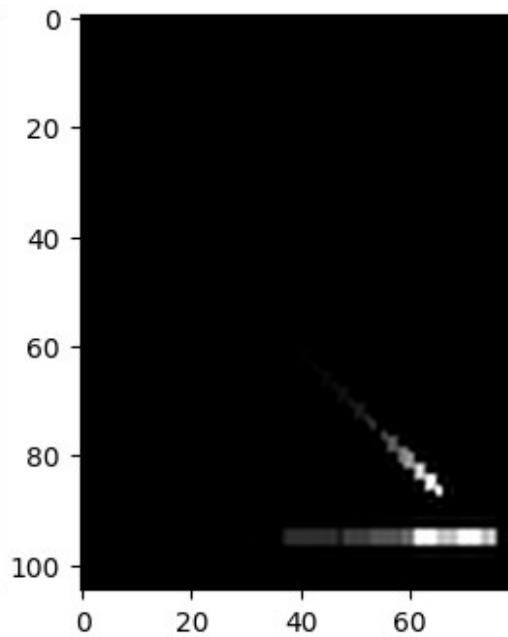
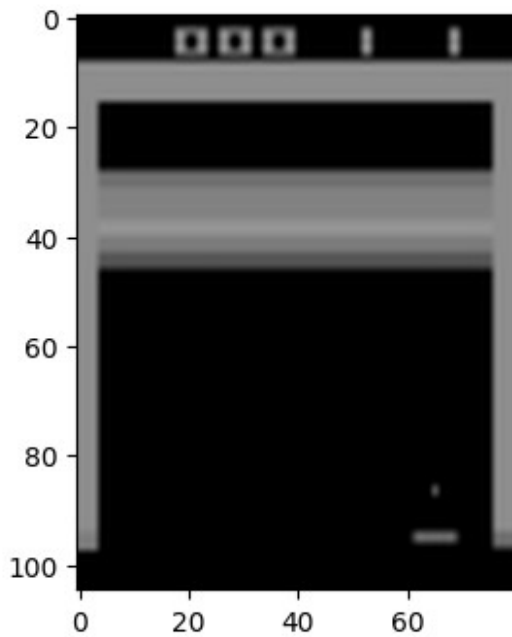
RIGHT	0.004
LEFT	0.004 (Action Taken)



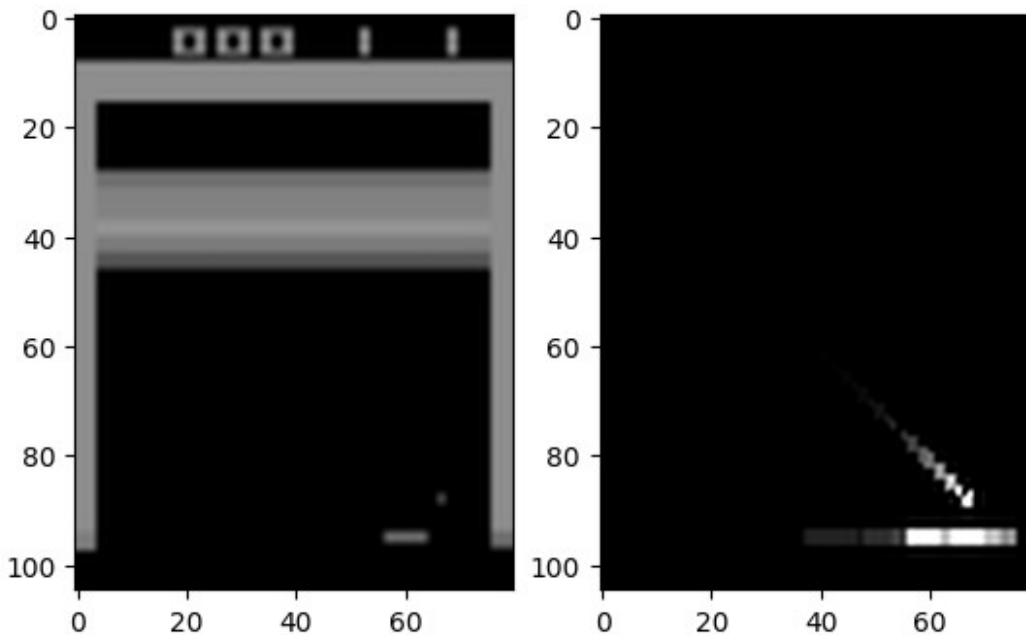
Action:	Q-Value:
=====	
NOOP	0.004
FIRE	0.003 (Action Taken)
RIGHT	0.006
LEFT	0.004



Action:	Q-Value:
NOOP	0.003
FIRE	0.004
RIGHT	0.005
LEFT	0.004 (Action Taken)



Action:	Q-Value:
=====	=====
NOOP	0.003
FIRE	0.006
RIGHT	0.008 (Action Taken)
LEFT	0.004



Action:	Q-Value:
=====	=====
NOOP	0.002
FIRE	0.006
RIGHT	0.007
LEFT	0.003 (Action Taken)

Output of Convolutional Layers

The outputs of the convolutional layers can be plotted so we can see how the images from the game-environment are being processed by the Neural Network.

This is the helper-function for plotting the output of the convolutional layer with the given name, when inputting the given state from the replay-memory.

```
def plot_layer_output(model, layer_name, state_index,
inverse_cmap=False):
    """
    Plot the output of a convolutional layer.
```

```

:param model: An instance of the NeuralNetwork-class.
:param layer_name: Name of the convolutional layer.
:param state_index: Index into the replay-memory for a state that
                    will be input to the Neural Network.
:param inverse_cmap: Boolean whether to inverse the color-map.
"""

# Get the given state-array from the replay-memory.
state = replay_memory.states[state_index]

# Get the output tensor for the given layer inside the TensorFlow
graph.
# This is not the value-contents but merely a reference to the
tensor.
layer_tensor = model.get_layer_tensor(layer_name=layer_name)

# Get the actual value of the tensor by feeding the state-data
# to the TensorFlow graph and calculating the value of the tensor.
values = model.get_tensor_value(tensor=layer_tensor, state=state)

# Number of image channels output by the convolutional layer.
num_images = values.shape[3]

# Number of grid-cells to plot.
# Rounded-up, square-root of the number of filters.
num_grids = math.ceil(math.sqrt(num_images))

# Create figure with a grid of sub-plots.
fig, axes = plt.subplots(num_grids, num_grids, figsize=(10, 10))

print("Dim. of each image:", values.shape)

if inverse_cmap:
    cmap = 'gray_r'
else:
    cmap = 'gray'

# Plot the outputs of all the channels in the conv-layer.
for i, ax in enumerate(axes.flat):
    # Only plot the valid image-channels.
    if i < num_images:
        # Get the image for the i'th output channel.
        img = values[0, :, :, i]

        # Plot image.
        ax.imshow(img, interpolation='nearest', cmap=cmap)

    # Remove ticks from the plot.
    ax.set_xticks([])

```

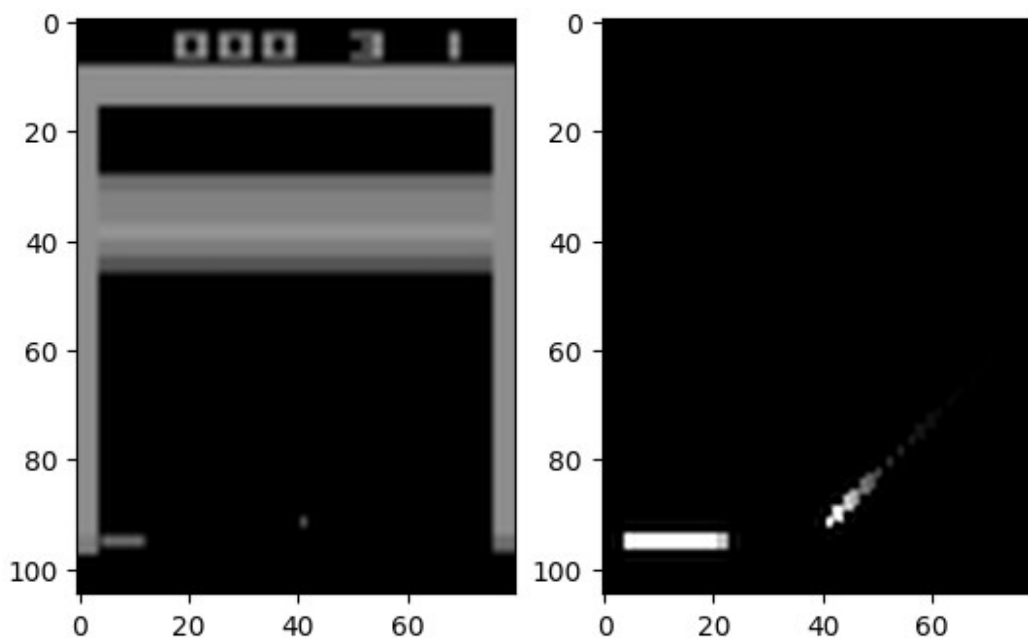
```
ax.set_yticks([])

# Ensure the plot is shown correctly with multiple plots
# in a single Notebook cell.
plt.show()
```

Game State

This is the state that is being input to the Neural Network. The image on the left is the last image from the game-environment. The image on the right is the processed motion-trace that shows the trajectories of objects in the game-environment.

```
idx = np.argmax(q_values_max)
plot_state(idx=idx, print_q=False)
```



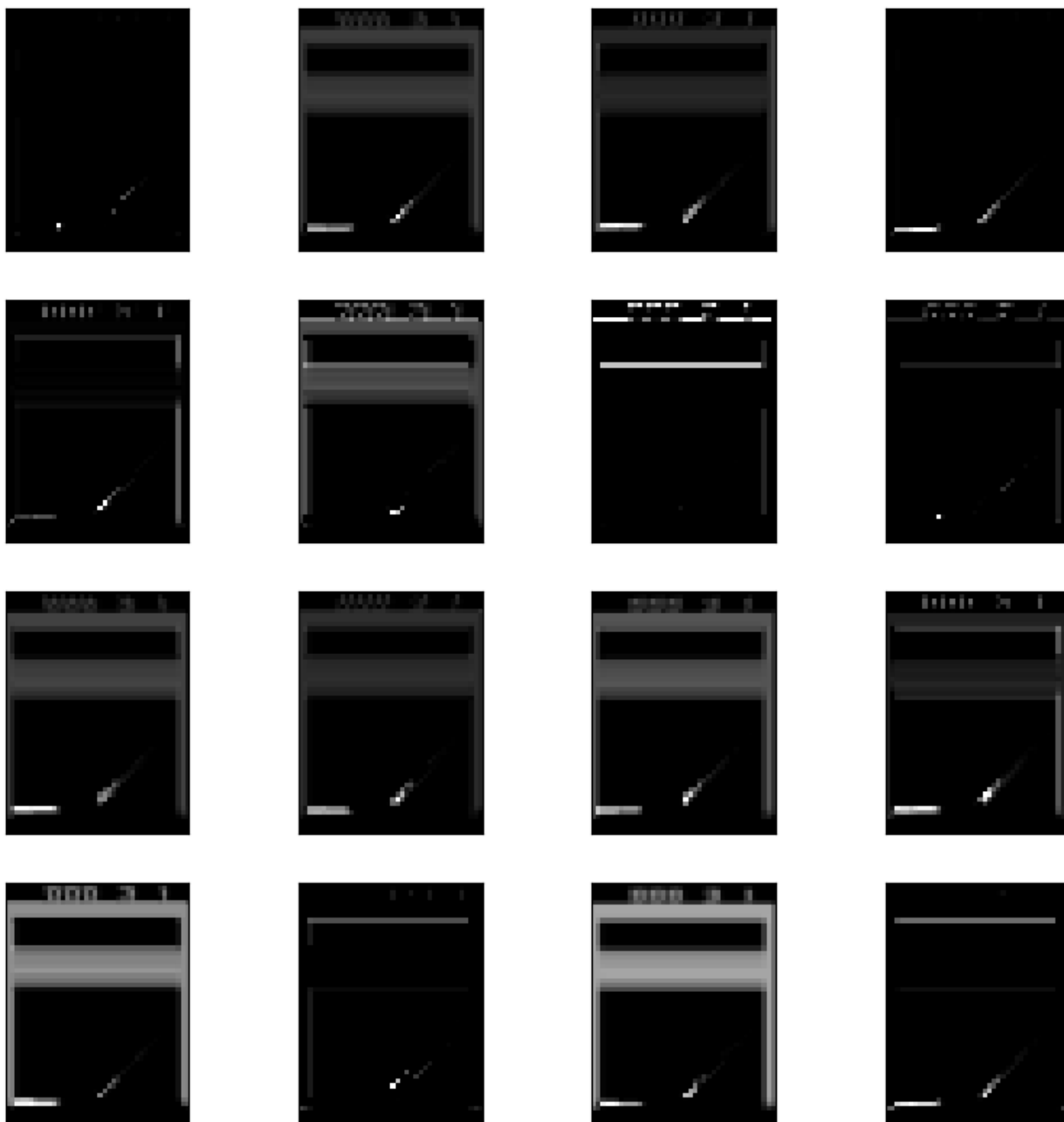
Output of Convolutional Layer 1

This shows the images that are output by the 1st convolutional layer, when inputting the above state to the Neural Network. There are 16 output channels of this convolutional layer.

Note that you can invert the colors by setting `inverse_cmap=True` in the parameters to this function.

```
plot_layer_output(model=model, layer_name='layer_conv1',
state_index=idx, inverse_cmap=False)
```

Dim. of each image: (1, 53, 40, 16)

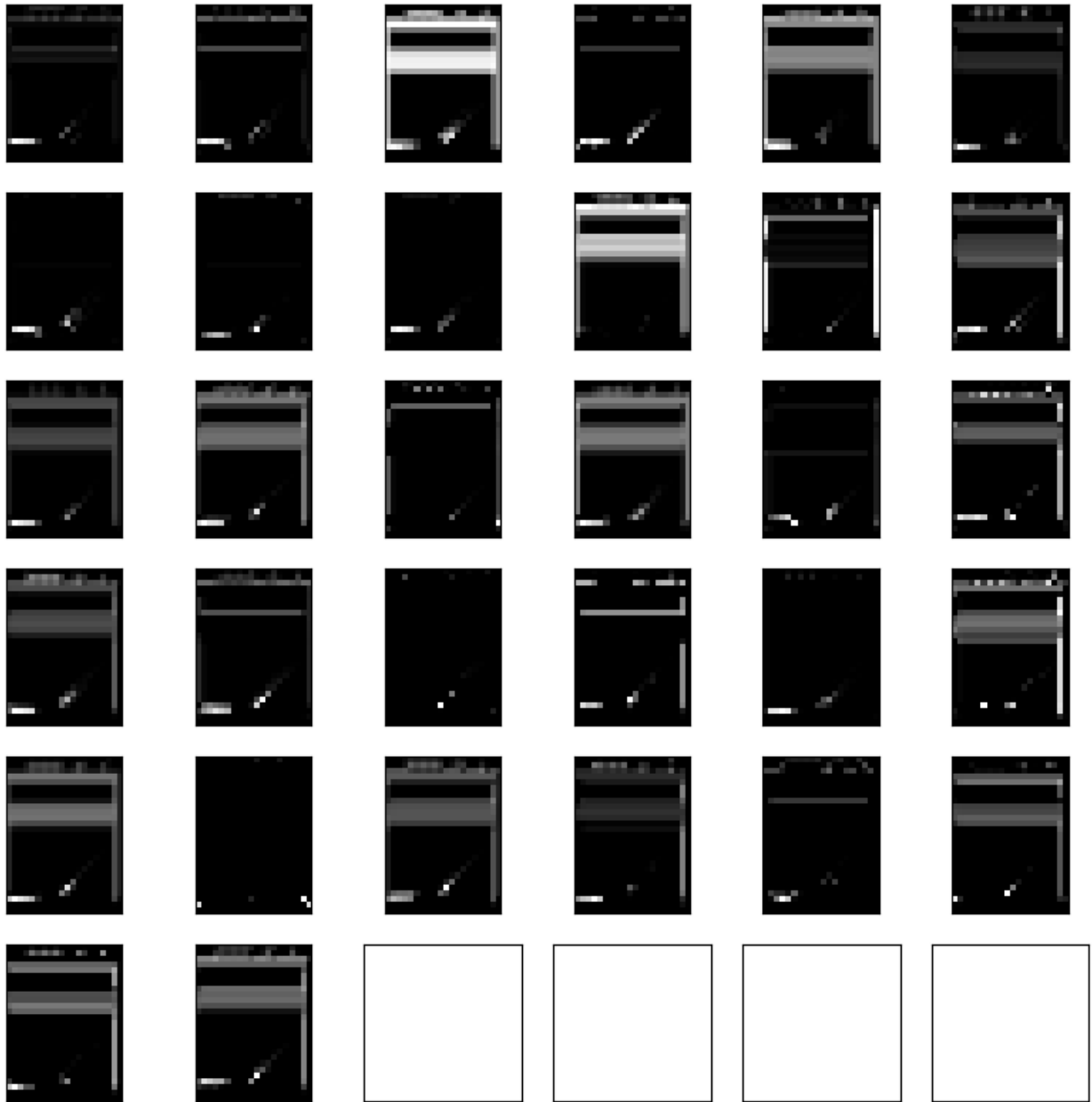


Output of Convolutional Layer 2

These are the images output by the 2nd convolutional layer, when inputting the above state to the Neural Network. There are 32 output channels of this convolutional layer.

```
plot_layer_output(model=model, layer_name='layer_conv2',
state_index=idx, inverse_cmap=False)
```

Dim. of each image: (1, 27, 20, 32)



Output of Convolutional Layer 3

These are the images output by the 3rd convolutional layer, when inputting the above state to the Neural Network. There are 64 output channels of this convolutional layer.

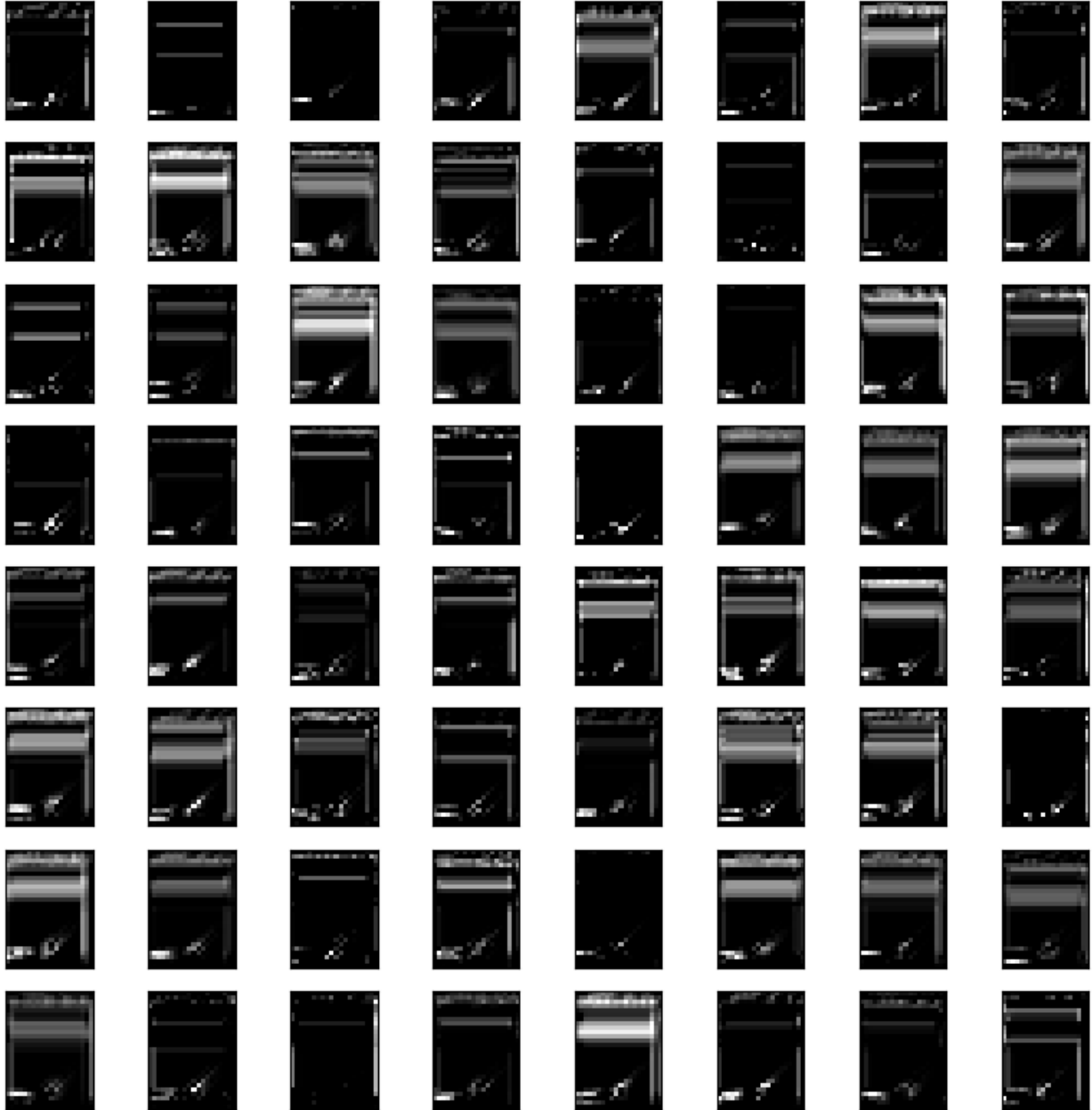
All these images are flattened to a one-dimensional array (or tensor) which is then used as the input to a fully-connected layer in the Neural Network.

During the training-process, the Neural Network has learnt what convolutional filters to apply to the images from the game-environment so as to produce these images, because they have proven to be useful when estimating Q-values.

Can you see what it is that the Neural Network has learned to detect in these images?

```
plot_layer_output(model=model, layer_name='layer_conv3',  
state_index=idx, inverse_cmap=False)
```

Dim. of each image: (1, 27, 20, 64)



Weights for Convolutional Layers

We can also plot the weights of the convolutional layers in the Neural Network. These are the weights that are being optimized so as to improve the ability of the Neural Network to estimate Q-values. Tutorial #02 explains in greater detail what convolutional weights are. There are also weights for the fully-connected layers but they are not shown here.

This is the helper-function for plotting the weights of a convolutional layer.

```
def plot_conv_weights(model, layer_name, input_channel=0):  
    """  
    Plot the weights for a convolutional layer.  
  
    :param model: An instance of the NeuralNetwork-class.  
    :param layer_name: Name of the convolutional layer.  
    :param input_channel: Plot the weights for this input-channel.  
    """  
  
    # Get the variable for the weights of the given layer.  
    # This is a reference to the variable inside TensorFlow,  
    # not its actual value.  
    weights_variable =  
model.get_weights_variable(layer_name=layer_name)  
  
    # Retrieve the values of the weight-variable from TensorFlow.  
    # The format of this 4-dim tensor is determined by the  
    # TensorFlow API. See Tutorial #02 for more details.  
    w = model.get_variable_value(variable=weights_variable)  
  
    # Get the weights for the given input-channel.  
    w_channel = w[:, :, input_channel, :]  
  
    # Number of output-channels for the conv. layer.  
    num_output_channels = w_channel.shape[2]  
  
    # Get the lowest and highest values for the weights.  
    # This is used to correct the colour intensity across  
    # the images so they can be compared with each other.  
    w_min = np.min(w_channel)  
    w_max = np.max(w_channel)  
  
    # This is used to center the colour intensity at zero.  
    abs_max = max(abs(w_min), abs(w_max))  
  
    # Print statistics for the weights.  
    print("Min: {0:.5f}, Max: {1:.5f}".format(w_min, w_max))  
    print("Mean: {0:.5f}, Stdev: {1:.5f}".format(w_channel.mean(),  
                                                w_channel.std()))  
  
    # Number of grids to plot.  
    # Rounded-up, square-root of the number of output-channels.  
    num_grids = math.ceil(math.sqrt(num_output_channels))  
  
    # Create figure with a grid of sub-plots.  
    fig, axes = plt.subplots(num_grids, num_grids)  
  
    # Plot all the filter-weights.  
    for i, ax in enumerate(axes.flat):
```

```

    # Only plot the valid filter-weights.
    if i < num_output_channels:
        # Get the weights for the i'th filter of this input-
channel.
        img = w_channel[:, :, i]

        # Plot image.
        ax.imshow(img, vmin=-abs_max, vmax=abs_max,
                  interpolation='nearest', cmap='seismic')

        # Remove ticks from the plot.
        ax.set_xticks([])
        ax.set_yticks([])

# Ensure the plot is shown correctly with multiple plots
# in a single Notebook cell.
plt.show()

```

Weights for Convolutional Layer 1

These are the weights of the first convolutional layer of the Neural Network, with respect to the first input channel of the state. That is, these are the weights that are used on the image from the game-environment. Some basic statistics are also shown.

Note how the weights are more negative (blue) than positive (red). It is unclear why this happens as these weights are found through optimization. It is apparently beneficial for the following layers to have this processing with more negative weights in the first convolutional layer.

```

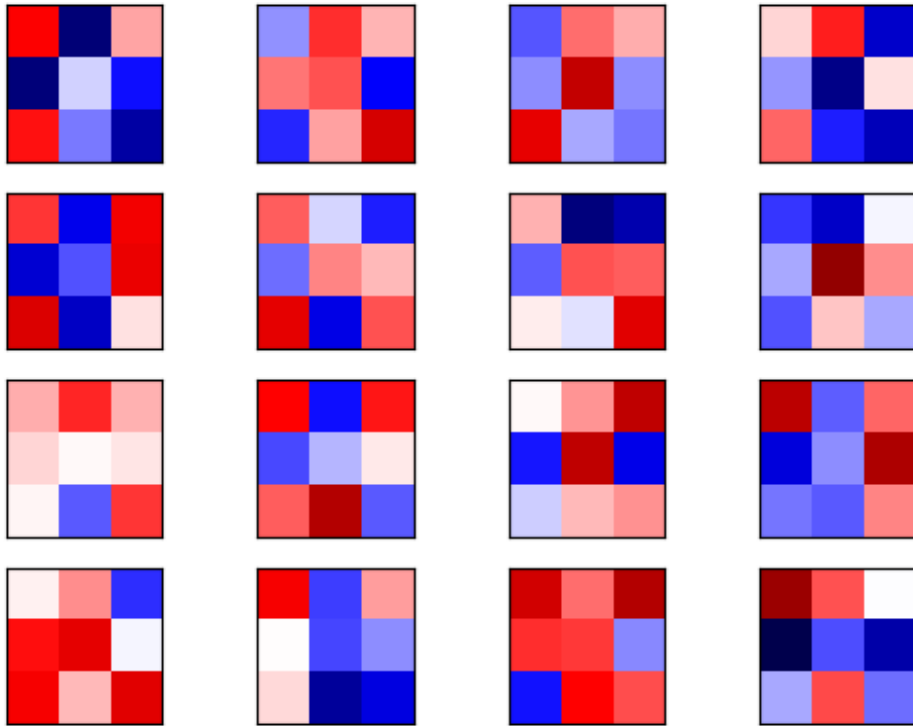
plot_conv_weights(model=model, layer_name='layer_conv1',
input_channel=0)

```

```

Min:  -0.03786, Max:   0.03483
Mean:  0.00054, Stdev: 0.01697

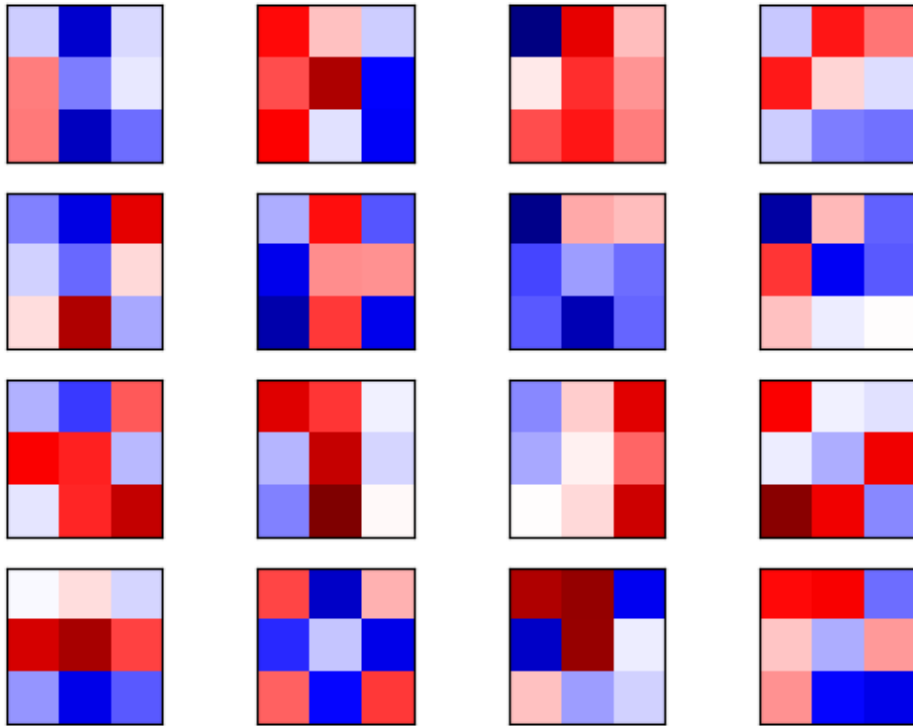
```

We can also plot the convolutional weights for the second input channel, that is, the motion-trace of the game-environment. Once again we see that the negative weights (blue) have a much greater magnitude than the positive weights (red).

```
plot_conv_weights(model=model, layer_name='layer_conv1',
input_channel=1)
```

```
Min: -0.03322, Max: 0.03933
Mean: 0.00143, Stdev: 0.01653
```



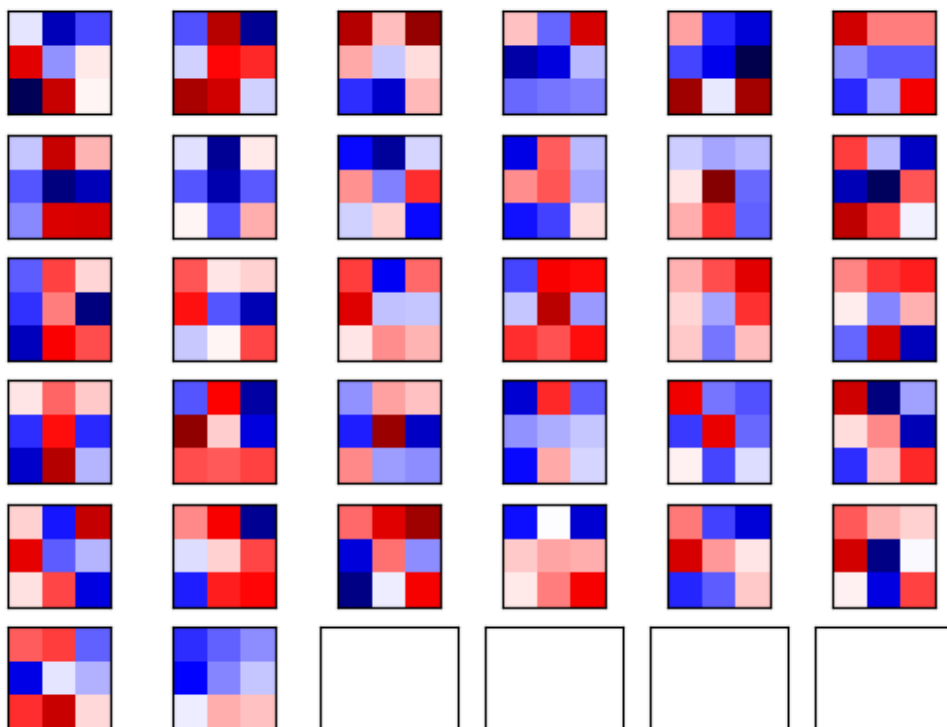
Weights for Convolutional Layer 2

These are the weights of the 2nd convolutional layer in the Neural Network. There are 16 input channels and 32 output channels of this layer. You can change the number for the input-channel to see the associated weights.

Note how the weights are more balanced between positive (red) and negative (blue) compared to the weights for the 1st convolutional layer above.

```
plot_conv_weights(model=model, layer_name='layer_conv2',  
input_channel=0)
```

```
Min:  -0.03990, Max:   0.03888  
Mean: -0.00016, Stdev: 0.01752
```



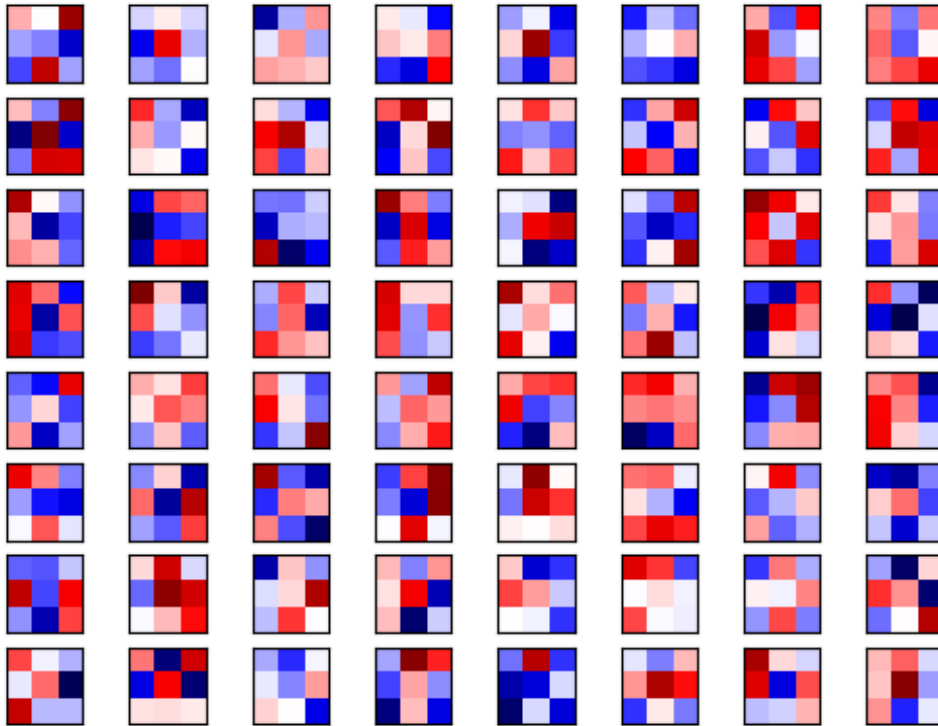
Weights for Convolutional Layer 3

These are the weights of the 3rd convolutional layer in the Neural Network. There are 32 input channels and 64 output channels of this layer. You can change the number for the input-channel to see the associated weights.

Note again how the weights are more balanced between positive (red) and negative (blue) compared to the weights for the 1st convolutional layer above.

```
plot_conv_weights(model=model, layer_name='layer_conv3',
input_channel=0)
```

```
Min: -0.03941, Max: 0.03954
Mean: 0.00016, Stdev: 0.01732
```



Discussion

We trained an agent to play old Atari games quite well using Reinforcement Learning. Recent improvements to the training algorithm have improved the performance significantly. But is this true human-like intelligence? The answer is clearly NO!

Reinforcement Learning in its current form is a crude numerical algorithm for connecting visual images, actions, rewards and penalties when there is a time-lag between the signals. The learning is based on trial-and-error and cannot do logical reasoning like a human. The agent has no sense of "self" while a human has an understanding of what part of the game-environment it is controlling, so a human can reason logically like this: "(A) I control the paddle, and (B) I must avoid dying which happens when the ball flies past the paddle, so (C) I must move the paddle to hit the ball, and (D) this automatically scores points when the ball smashes bricks in the wall". A human would first learn these basic logical rules of the game - and then try and refine the eye-hand coordination to play the game better. Reinforcement Learning has no real comprehension of what is going on in the game and merely works on improving the eye-hand coordination until it gets lucky and does the right thing to score more points.

Furthermore, the training of the Reinforcement Learning algorithm required almost 150 hours of computation which played the game at high speeds. If the game was played at normal real-time speeds then it would have taken more than 1700 hours to train the agent, which is more than 70 days and nights.

Logical reasoning would allow for much faster learning than Reinforcement Learning, and it would be able to solve much more complicated problems than simple eye-hand coordination. I am skeptical if someone will be able to create true human-like intelligence from Reinforcement Learning algorithms.

Does that mean Reinforcement Learning is completely worthless? No, it has real-world applications that currently cannot be solved by other methods.

Another point of criticism is the use of Neural Networks. The majority of the research in Reinforcement Learning is actually spent on trying to stabilize the training of the Neural Network using various tricks. This is a waste of research time and strongly indicates that Neural Networks may not be a very good Machine Learning model compared to the human brain.

Exercises & Research Ideas

Below are suggestions for exercises and experiments that may help improve your skills with TensorFlow and Reinforcement Learning. Some of these ideas can easily be extended into full research problems that would help the community if you can solve them.

You should keep a log of your experiments, describing for each experiment the settings you tried and the results. You should also save the source-code and checkpoints / log-files.

It takes so much time to run these experiments, so please share your results with the rest of the community. Even if an experiment failed to produce anything useful, it will be helpful to others so they know not to redo the same experiment.

[Thread on GitHub for discussing these experiments](#)

You may want to backup this Notebook and the other files before making any changes.

You may find it helpful to add more command-line parameters to `reinforcement_learning.py` so you don't have to edit the source-code for testing other parameters.

- Change the epsilon-probability during testing to e.g. 0.001 or 0.05. Which gives the best results? Could you use this value during training? Why/not?
- Try and change the game-environment to Space Invaders and re-run this Notebook. The hyper-parameters such as the learning-rate were tuned for Breakout. Can you make some kind of adaptive learning-rate that would work better for both Breakout and Space Invaders? What about the other hyper-parameters? What about other games?
- Try different architectures for the Neural Network. You will need to restart the training because the checkpoints cannot be reused for other architectures. You will need to train the agent for several days with each new architecture so as to properly assess its performance.
- The replay-memory throws away all data after optimization of the Neural Network. Can you make it reuse the data somehow? The ReplayMemory-class has the function `estimate_all_q_values()` which may be helpful.
- The reward is limited to -1 and 1 in the function `ReplayMemory.add()` so as to stabilize the training. This means the agent cannot distinguish between small and large rewards. Can you use batch normalization to fix this problem, so you can use the actual reward values?
- Can you improve the training by adding L2-regularization or dropout?
- Try using other optimizers for the Neural Network. Does it help with the training speed or stability?

- Let the agent take up to 30 random actions at the beginning of each new episode. This is used in some research papers to further randomize the game-environment, so the agent cannot memorize the first sequence of actions.
- Try and save the game at regular intervals. If the agent dies, then you can reload the last saved game. Would this help training the agent faster and better, because it does not need to play the game from the beginning?
- There are some invalid actions available to the agent in OpenAI Gym. Does it improve the training if you only allow the valid actions from the game-environment?
- Does the MotionTracer work for other games? Can you improve on the MotionTracer?
- Try and use the last 4 image-frames from the game instead of the MotionTracer.
- Try larger and smaller sizes for the replay memory.
- Try larger and smaller discount rates for updating the Q-values.
- If you look closely in the states and actions that are display above, you will note that the agent has sometimes taken actions that do not correspond to the movement of the paddle. For example, the action might be LEFT but the paddle has either not moved at all, or it has moved right instead. Is this a bug in the source-code for this tutorial, or is it a bug in OpenAI Gym, or is it a bug in the underlying Atari Learning Environment? Does it matter?

License (MIT)

Copyright (c) 2017 by [Magnus Erik Hvass Pedersen](#)

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

3.) GOOGLE COLAB SCREENSHOT FOR RUNTIME SHOWING

The main source-code for Reinforcement Learning is located in the following module:

```
[6] 1 import reinforcement_learning as rl
```

This was developed using Python 3.6.0 (Anaconda) with package versions:

```
[7] 1 # TensorFlow
    2 tf.__version__
```

```
'2.15.0'
```

```
[8] 1 # OpenAI Gym
    2 gym.__version__
```

```
'0.23.0'
```

Game Environment

This is the name of the game-environment that we want to use in OpenAI Gym.

```
[9] 1 env_name = 'Breakout-v0'
    2 # env_name = 'SpaceInvaders-v0'
```

This is the base-directory for the TensorFlow checkpoints as well as various log-files.

```
[10] 1 rl.checkpoint_base_dir = 'checkpoints_tutorial16/'
```

Once the base-dir has been set, you need to call this function to set all the paths that will be used. This will also create the checkpoint-dir if it does not already exist.

```
[11] 1 rl.update_paths(env_name=env_name)
```

Download Pre-Trained Model

The original version of this tutorial provided some TensorFlow checkpoints with pre-trained models for download. But due to changes in both TensorFlow and OpenAI Gym, these pre-trained models cannot be loaded anymore so they have been deleted from the web-server. You will therefore have to train your own model further below.

```
[13] 1 model = agent.model
```

Similarly, the Agent-class also allocates the replay-memory when `training==True`. The replay-memory will require more than 3 GB of RAM, so it should only be allocated when needed. We will need the replay-memory in this Notebook to record the states and Q-values we observe, so they can be plotted further below.

```
[14] 1 replay_memory = agent.replay_memory
```

Training

The agent's `run()` function is used to play the game. This uses the Neural Network to estimate Q-values and hence determine the agent's actions. If `training==True` then it will also gather states and Q-values in the replay-memory and train the Neural Network when the replay-memory is sufficiently full. You can set `num_episodes=None` if you want an infinite loop that you would stop manually with `ctrl-c`. In this case we just set `num_episodes=1` because we are not actually interested in training the Neural Network any further, we merely want to collect some states and Q-values in the replay-memory so we can plot them below.

```
[15] 1 agent.run(num_episodes=30)
```

1:271	Epsilon: 1.00	Reward: 2.0	Episode Mean: 2.0
2:488	Epsilon: 1.00	Reward: 1.0	Episode Mean: 1.5
3:894	Epsilon: 1.00	Reward: 4.0	Episode Mean: 2.3
4:1165	Epsilon: 1.00	Reward: 2.0	Episode Mean: 2.2
5:1344	Epsilon: 1.00	Reward: 0.0	Episode Mean: 1.8
6:1526	Epsilon: 1.00	Reward: 0.0	Episode Mean: 1.5
7:1776	Epsilon: 1.00	Reward: 1.0	Episode Mean: 1.4
8:2050	Epsilon: 1.00	Reward: 2.0	Episode Mean: 1.5
9:2363	Epsilon: 1.00	Reward: 2.0	Episode Mean: 1.6
10:2667	Epsilon: 1.00	Reward: 2.0	Episode Mean: 1.6
11:2892	Epsilon: 1.00	Reward: 1.0	Episode Mean: 1.5
12:3121	Epsilon: 1.00	Reward: 1.0	Episode Mean: 1.5
13:3335	Epsilon: 1.00	Reward: 1.0	Episode Mean: 1.5
14:3614	Epsilon: 1.00	Reward: 2.0	Episode Mean: 1.5
15:3799	Epsilon: 1.00	Reward: 0.0	Episode Mean: 1.4
16:4012	Epsilon: 1.00	Reward: 0.0	Episode Mean: 1.3
17:4314	Epsilon: 1.00	Reward: 2.0	Episode Mean: 1.4
18:4522	Epsilon: 1.00	Reward: 1.0	Episode Mean: 1.3
19:4997	Epsilon: 1.00	Reward: 5.0	Episode Mean: 1.5
20:5166	Epsilon: 1.00	Reward: 0.0	Episode Mean: 1.4
21:5337	Epsilon: 1.00	Reward: 0.0	Episode Mean: 1.4
22:5766	Epsilon: 0.99	Reward: 4.0	Episode Mean: 1.5
23:5961	Epsilon: 0.99	Reward: 0.0	Episode Mean: 1.4
24:6237	Epsilon: 0.99	Reward: 2.0	Episode Mean: 1.5
25:6478	Epsilon: 0.99	Reward: 1.0	Episode Mean: 1.4
26:6716	Epsilon: 0.99	Reward: 1.0	Episode Mean: 1.4
27:6902	Epsilon: 0.99	Reward: 0.0	Episode Mean: 1.4
28:7174	Epsilon: 0.99	Reward: 2.0	Episode Mean: 1.4
29:7422	Epsilon: 0.99	Reward: 2.0	Episode Mean: 1.4
30:7655	Epsilon: 0.99	Reward: 1.0	Episode Mean: 1.4

In training-mode, this function will output a line for each episode. The first counter is for the number of episodes that have been processed. The second counter is for the number of states that have been processed. These two counters are stored in the TensorFlow checkpoint along with the weights of the Neural Network, so you can restart the training e.g. if you only have one computer and need to train during the night.

Note that the number of episodes is almost 90k. It is impractical to print that many lines in this Notebook, so the training is better done in a terminal window by running the following commands:

```
source activate tf-gpu-gym # Activate your Python environment with TF and Gym.
```

✓ 3s completed at 8:40 PM

log this data.

We have therefore implemented a few small classes that can write and read these logs.

```
[16] 1 log_q_values = rl.LogQValues()  
     2 log_reward = rl.LogReward()
```

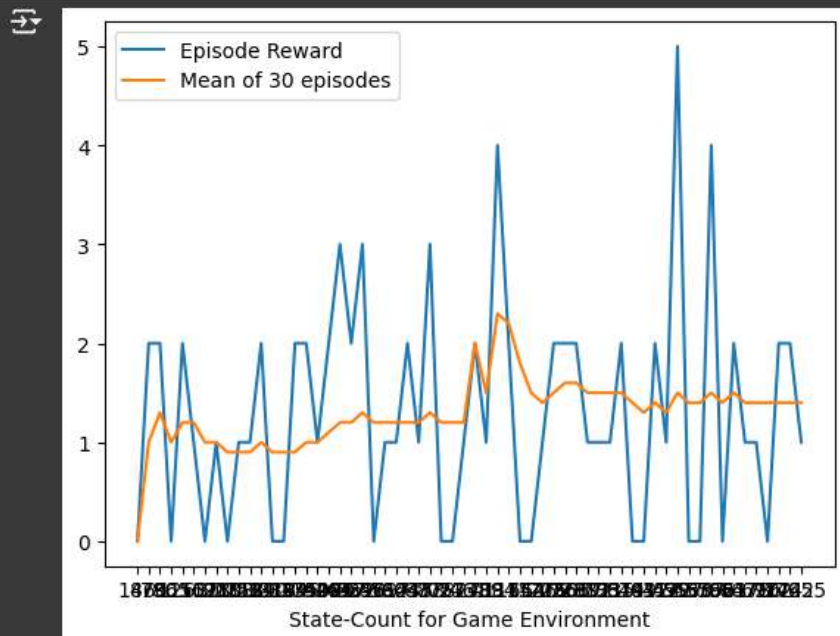
We can now read the logs from file:

```
[17] 1  
     2 log_reward.read()
```

✓ Training Progress: Reward

This plot shows the reward for each episode during training, as well as the running mean of the last 30 episodes. Note how the reward varies greatly from one episode to the next, so it is difficult to say from this plot alone whether the agent is really improving during the training, although the running mean does appear to trend upwards slightly.

```
[18] 1 plt.plot(log_reward.count_states, log_reward.episode, label='Episode Reward')  
     2 plt.plot(log_reward.count_states, log_reward.mean, label='Mean of 30 episodes')  
     3 plt.xlabel('State-Count for Game Environment')  
     4 plt.legend()  
     5 plt.show()
```



Testing

When the agent and Neural Network is being trained, the so-called epsilon-probability is typically decreased from 1.0 to 0.1 over a large number of steps, after which the probability is held fixed at 0.1. This means the probability is 0.1 or 10% that the agent will select a random action in each step, otherwise it will select the action that has the highest Q-value. This is known as the epsilon-greedy policy. The choice of 0.1 for the epsilon-probability is a compromise between taking the actions that are already known to be good, versus exploring new actions that might lead to even higher rewards or might lead to death of the agent.

During testing it is common to lower the epsilon-probability even further. We have set it to 0.01 as shown here:

```
[19] 1 agent.epsilon_greedy.epsilon_testing
0s 0.01
```

We will now instruct the agent that it should no longer perform training by setting this boolean:

```
[20] 1 agent.training = False
0s
```

We also reset the previous episode rewards.

```
[21] 1 agent.reset_episode_rewards()
0s
```

We can render the game-environment to screen so we can see the agent playing the game, by setting this boolean:

```
[22] 1 agent.render = True
0s
```

We can now run a single episode by calling the `run()` function again. This should open a new window that shows the game being played by the agent. At the time of this writing, it was not possible to resize this tiny window, and the developers at OpenAI did not seem to care about this feature which should obviously be there.

Mean Reward

The game-play is slightly random, both with regard to selecting actions using the epsilon-greedy policy, but also because the OpenAI Gym environment will repeat any action between 2-4 times, with the number chosen at random. So the reward of one episode is not an accurate estimate of the reward that can be expected in general from this agent.

We need to run 30 or even 50 episodes to get a more accurate estimate of the reward that can be expected.

We will first reset the previous episode rewards.

```
[23] 1 agent.reset_episode_rewards()
```

We disable the screen-rendering so the game-environment runs much faster.

```
[24] 1 agent.render = False
```

We can now run 5 episodes. This records the rewards for each episode. It might have been a good idea to disable the output so it does not print all these lines - you can do this as an exercise.

```
[25] 1 agent.run(num_episodes=5)
```

32:8032	Q-min: 0.001	Q-max: 0.009	Lives: 5	Reward: 1.0	Episode Mean: 0.0
32:8081	Q-min: 0.003	Q-max: 0.009	Lives: 5	Reward: 2.0	Episode Mean: 0.0
32:8108	Q-min: 0.002	Q-max: 0.010	Lives: 4	Reward: 2.0	Episode Mean: 0.0
32:9217	Q-min: 0.002	Q-max: 0.010	Lives: 3	Reward: 2.0	Episode Mean: 0.0
32:9722	Q-min: 0.003	Q-max: 0.011	Lives: 3	Reward: 3.0	Episode Mean: 0.0
32:9764	Q-min: 0.002	Q-max: 0.010	Lives: 2	Reward: 3.0	Episode Mean: 0.0
32:9986	Q-min: 0.002	Q-max: 0.010	Lives: 1	Reward: 3.0	Episode Mean: 0.0
32:10997	Q-min: 0.002	Q-max: 0.009	Lives: 0	Reward: 3.0	Episode Mean: 0.0
32:10998	Q-min: 0.002	Q-max: 0.010	Lives: 0	Reward: 3.0	Episode Mean: 3.0
33:11079	Q-min: 0.001	Q-max: 0.008	Lives: 5	Reward: 1.0	Episode Mean: 3.0
33:11128	Q-min: 0.002	Q-max: 0.008	Lives: 5	Reward: 2.0	Episode Mean: 3.0
33:11154	Q-min: 0.002	Q-max: 0.010	Lives: 4	Reward: 2.0	Episode Mean: 3.0
33:11551	Q-min: 0.002	Q-max: 0.010	Lives: 4	Reward: 3.0	Episode Mean: 3.0
33:11593	Q-min: 0.002	Q-max: 0.010	Lives: 3	Reward: 3.0	Episode Mean: 3.0
33:11720	Q-min: 0.002	Q-max: 0.010	Lives: 2	Reward: 3.0	Episode Mean: 3.0
33:12233	Q-min: 0.003	Q-max: 0.010	Lives: 2	Reward: 4.0	Episode Mean: 3.0
33:12276	Q-min: 0.002	Q-max: 0.010	Lives: 1	Reward: 4.0	Episode Mean: 3.0
33:12581	Q-min: 0.002	Q-max: 0.010	Lives: 0	Reward: 4.0	Episode Mean: 3.0
33:12582	Q-min: 0.002	Q-max: 0.010	Lives: 0	Reward: 4.0	Episode Mean: 3.5
34:12795	Q-min: 0.001	Q-max: 0.009	Lives: 5	Reward: 1.0	Episode Mean: 3.5
34:12845	Q-min: 0.002	Q-max: 0.010	Lives: 5	Reward: 2.0	Episode Mean: 3.5
34:12871	Q-min: 0.002	Q-max: 0.010	Lives: 4	Reward: 2.0	Episode Mean: 3.5
34:12920	Q-min: 0.002	Q-max: 0.010	Lives: 3	Reward: 2.0	Episode Mean: 3.5
34:13239	Q-min: 0.002	Q-max: 0.010	Lives: 2	Reward: 2.0	Episode Mean: 3.5
34:14654	Q-min: 0.002	Q-max: 0.010	Lives: 2	Reward: 3.0	Episode Mean: 3.5
34:14692	Q-min: 0.002	Q-max: 0.010	Lives: 1	Reward: 3.0	Episode Mean: 3.5
34:15621	Q-min: 0.002	Q-max: 0.010	Lives: 0	Reward: 3.0	Episode Mean: 3.5
34:15622	Q-min: 0.002	Q-max: 0.010	Lives: 0	Reward: 3.0	Episode Mean: 3.3
35:16132	Q-min: 0.001	Q-max: 0.009	Lives: 4	Reward: 0.0	Episode Mean: 3.3
35:16457	Q-min: 0.002	Q-max: 0.009	Lives: 3	Reward: 0.0	Episode Mean: 3.3
35:16529	Q-min: 0.001	Q-max: 0.009	Lives: 2	Reward: 0.0	Episode Mean: 3.3
35:17058	Q-min: 0.001	Q-max: 0.010	Lives: 1	Reward: 0.0	Episode Mean: 3.3
35:17108	Q-min: 0.001	Q-max: 0.010	Lives: 0	Reward: 0.0	Episode Mean: 3.3
35:17109	Q-min: 0.001	Q-max: 0.010	Lives: 0	Reward: 0.0	Episode Mean: 2.5
36:17654	Q-min: 0.001	Q-max: 0.009	Lives: 4	Reward: 0.0	Episode Mean: 2.5
36:17784	Q-min: 0.002	Q-max: 0.009	Lives: 3	Reward: 0.0	Episode Mean: 2.5
36:18080	Q-min: 0.001	Q-max: 0.010	Lives: 2	Reward: 0.0	Episode Mean: 2.5
36:18555	Q-min: 0.001	Q-max: 0.010	Lives: 1	Reward: 0.0	Episode Mean: 2.5
36:18723	Q-min: 0.001	Q-max: 0.009	Lives: 0	Reward: 0.0	Episode Mean: 2.5
36:18724	Q-min: 0.001	Q-max: 0.009	Lives: 0	Reward: 0.0	Episode Mean: 2.0

We can now print some statistics for the episode rewards, which vary greatly from one episode to the next.

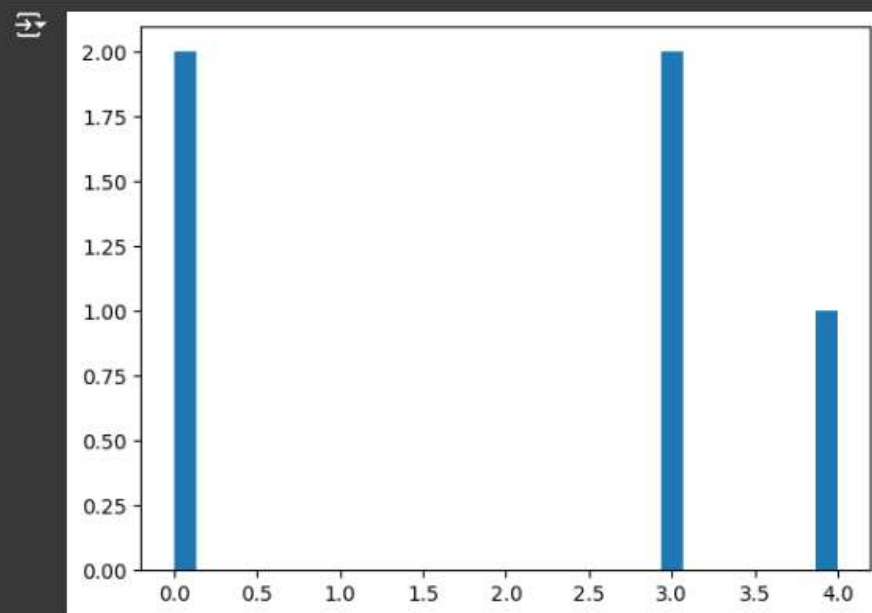
We can now print some statistics for the episode rewards, which vary greatly from one episode to the next.

```
[26] 1 rewards = agent.episode_rewards
      2 print("Rewards for {0} episodes:".format(len(rewards)))
      3 print("- Min: ", np.min(rewards))
      4 print("- Mean: ", np.mean(rewards))
      5 print("- Max: ", np.max(rewards))
      6 print("- Stdev: ", np.std(rewards))
```

```
↗ Rewards for 5 episodes:
- Min: 0.0
- Mean: 2.0
- Max: 4.0
- Stdev: 1.6733200530681511
```

We can also plot a histogram with the episode rewards.

```
[27] 1 _ = plt.hist(rewards, bins=30)
```



Example States

We can plot examples of states from the game-environment and the Q-values that are estimated by the Neural Network.

This helper-function prints the Q-values for a given index in the replay-memory.

```
[28] 1 def print_q_values(idx):
2     """Print Q-values and actions from the replay-memory at the given index."""
3
4     # Get the Q-values and action from the replay-memory.
5     q_values = replay_memory.q_values[idx]
6     action = replay_memory.actions[idx]
7
8     print("Action:      Q-Value:")
9     print("=====")
10
11    # Print all the actions and their Q-values.
12    for i, q_value in enumerate(q_values):
13        # Used to display which action was taken.
14        if i == action:
15            action_taken = "(Action Taken)"
16        else:
17            action_taken = ""
18
19        # Text-name of the action.
20        action_name = agent.get_action_name(i)
21
22        print("{0:12}{1:.3f} {2}".format(action_name, q_value,
23                                         action_taken))
24
25    # Newline.
26    print()
```

This helper-function plots a state from the replay-memory and optionally prints the Q-values.

```
[29] 1 def plot_state(idx, print_q=True):
2     """Plot the state in the replay-memory with the given index."""
3
4     # Get the state from the replay-memory.
5     state = replay_memory.states[idx]
6
7     # Create figure with a grid of sub-plots.
8     fig, axes = plt.subplots(1, 2)
9
10    # Plot the image from the game-environment.
11    ax = axes.flat[0]
12    ax.imshow(state[:, :, 0], vmin=0, vmax=255,
13              interpolation='lanczos', cmap='gray')
14
15    # Plot the motion-trace.
16    ax = axes.flat[1]
17    ax.imshow(state[:, :, 1], vmin=0, vmax=255,
18              interpolation='lanczos', cmap='gray')
19
20    # This is necessary if we show more than one plot in a single Notebook cell.
21    plt.show()
22
23    # Print the Q-values.
24    if print_q:
25        print_q_values(idx=idx)
```

The replay-memory has room for 200k states but it is only partially full from the above call to `agent.run(num_episodes=1)`. This is how many states are actually used.

```
[30] 1 num_used = replay_memory.num_used
2     num_used
```



7656


```
[31] 1 q_values = replay_memory.q_values[0:num_used, :]
```

For each state, calculate the min / max Q-values and their difference. This will be used to lookup interesting states in the following sections.

```
[32] 1 q_values_min = q_values.min(axis=1)
      2 q_values_max = q_values.max(axis=1)
      3 q_values_dif = q_values_max - q_values_min
```

Example States: Highest Reward

This example shows the states surrounding the state with the highest reward.

During the training we limit the rewards to the range [-1, 1] so this basically just gets the first state that has a reward of 1.

```
[33] 1 idx = np.argmax(replay_memory.rewards)
      2 idx
```

186

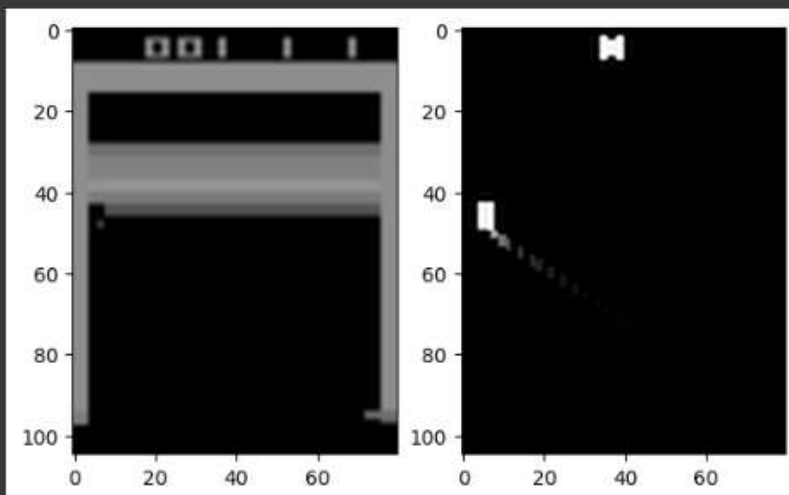
This state is where the ball hits the wall so the agent scores a point.

We can show the surrounding states leading up to and following this state. Note how the Q-values are very close for the different actions, because at this point it really does not matter what the agent does as the reward is already guaranteed. But note how the Q-values decrease significantly after the ball has hit the wall and a point has been scored.

Also note that the agent uses the Epsilon-greedy policy for taking actions, so there is a small probability that a random action is taken instead of the action with the highest Q-value.

```
[34] 1 for i in range(-5, 3):
      2     plot_state(idx=idx+i)
```

```
FIRE      0.004
RIGHT     0.010
LEFT      0.001
```



```
Action:    Q-Value:
=====
NOOP       0.004
FIRE       0.004
RIGHT      0.010
LEFT       0.002 (Action Taken)
```



Example: Highest Q-Value

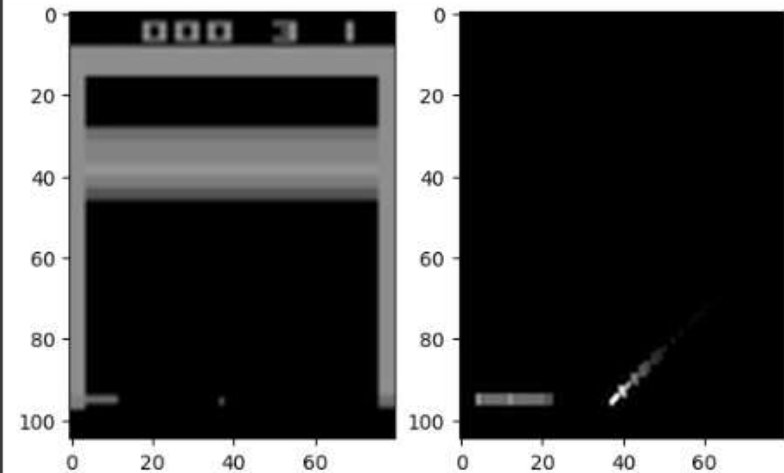
This example shows the states surrounding the one with the highest Q-values. This means that the agent has high expectation that several points will be scored in the following steps. Note that the Q-values decrease significantly after the points have been scored.

```
[35] 1 idx = np.argmax(q_values_max)
      2 idx
```

7505

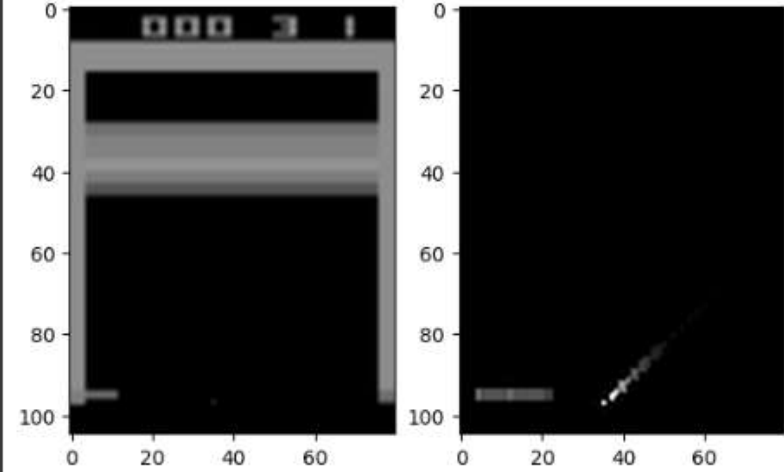
```
[36] 1 for i in range(0, 5):
      2     plot_state(idx=idx+i)
```

FIRE 0.003 (Action Taken)
RIGHT 0.015
LEFT 0.001



Action: Q-Value:
=====

NOOP	0.006
FIRE	0.003
RIGHT	0.014
LEFT	0.001 (Action Taken)



Action: Q-Value:
=====

NOOP	0.006
FIRE	0.002
RIGHT	0.012
LEFT	0.001 (Action Taken)

Example: Loss of Life

✓ Example: Loss of Life

This example shows the states leading up to a loss of life for the agent.

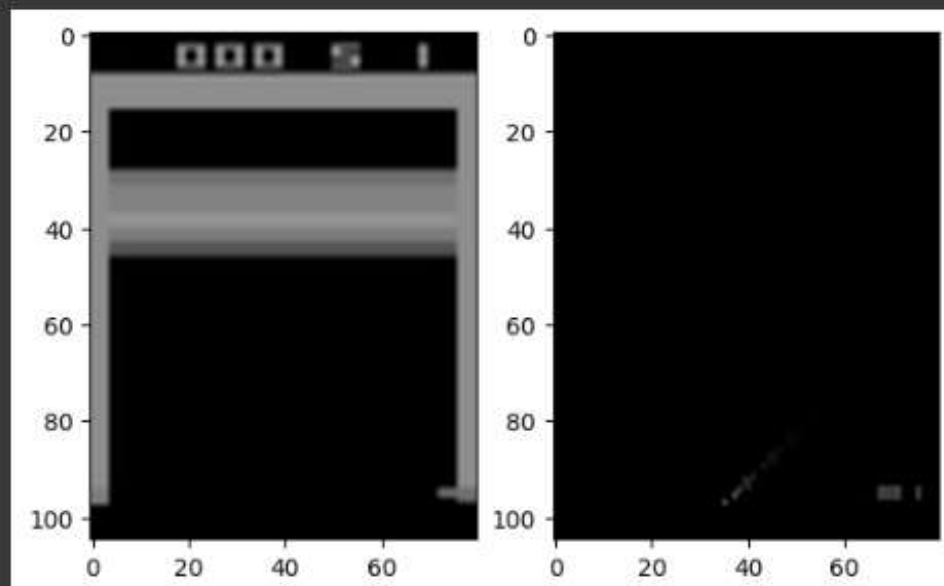
```
[37] 1 idx = np.argmax(replay_memory.end_life)
      2 idx
```

↔ 34

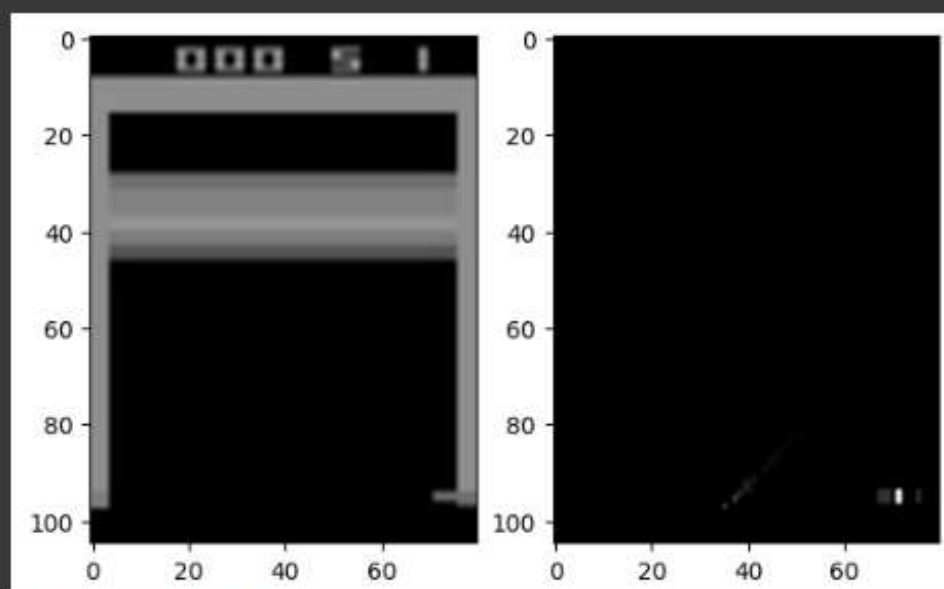
```
[38] 1 for i in range(-10, 0):
      2     plot_state(idx=idx+i)
```

↔

```
FIRE      0.001
RIGHT     0.010
LEFT      0.002 (Action Taken)
```



```
Action:      Q-Value:
-----
NOOP         0.005
FIRE         0.001
RIGHT        0.010
LEFT         0.002 (Action Taken)
```



```
Action:      Q-Value:
-----
NOOP         0.005
FIRE         0.002 (Action Taken)
RIGHT        0.010
LEFT         0.002
```


Example: Greatest Difference in Q-Values

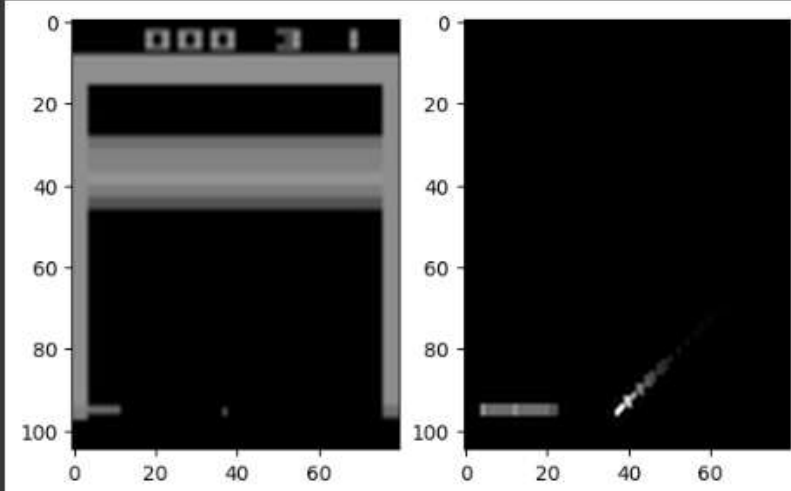
This example shows the state where there is the greatest difference in Q-values, which means that the agent believes one action will be much more beneficial than another. But because the agent uses the Epsilon-greedy policy, it sometimes selects a random action instead.

```
[39] 1 idx = np.argmax(q_values_dif)
      2 idx
```

7505

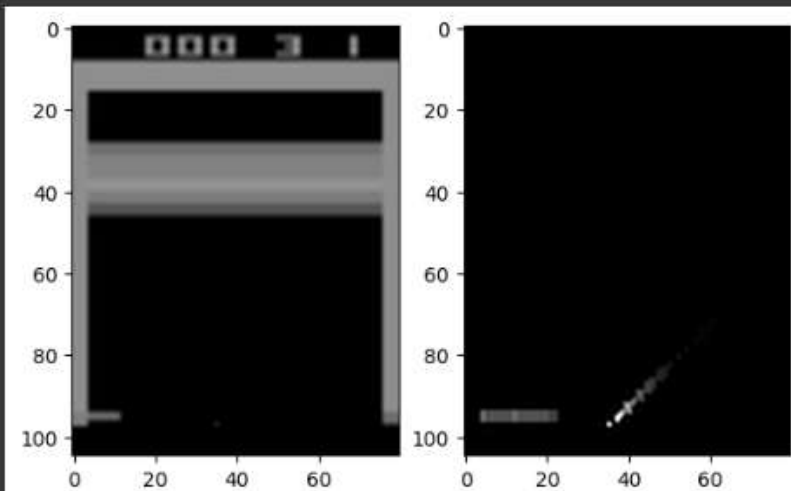
```
[40] 1 for i in range(0, 5):
      2     plot_state(idx=idx+i)

      FIRE 0.003 (ACTION TAKEN)
      RIGHT 0.015
      LEFT 0.001
```



Action: Q-Value:
=====

NOOP	0.006
FIRE	0.003
RIGHT	0.014
LEFT	0.001 (Action Taken)



Action: Q-Value:
=====

NOOP	0.006
FIRE	0.002
RIGHT	0.012
LEFT	0.001 (Action Taken)

Example: Smallest Difference in Q-Values

This example shows the state where there is the smallest difference in Q-values, which means that the agent believes it does not really matter which action it selects, as they all have roughly the same expectations for future rewards.

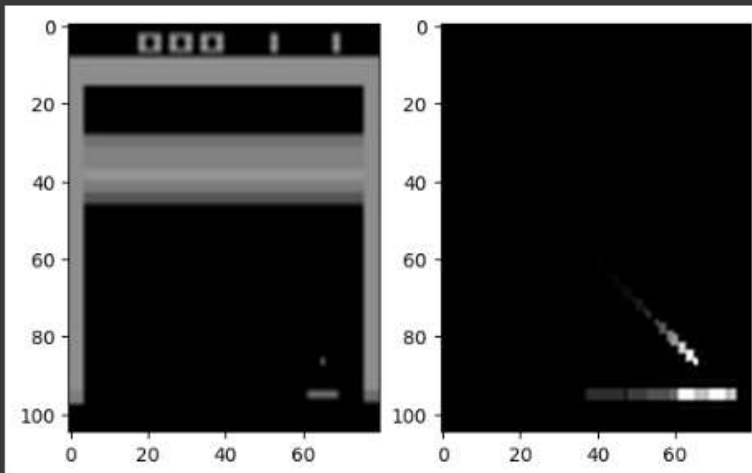
The Neural Network estimates these Q-values and they are not precise. The differences in Q-values may be so small that they fall within the error-range of the estimates.

```
[41] 1 idx = np.argmin(q_values_dif)
     2 idx
```

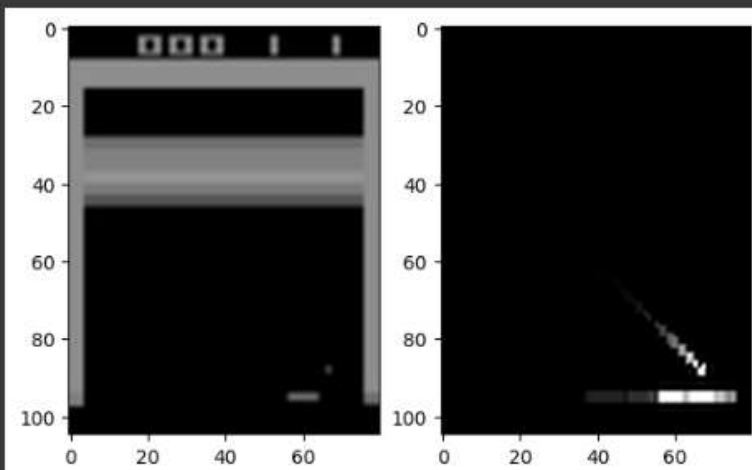
3779

```
[42] 1 for i in range(0, 5):
     2     plot_state(idx=idx+i)
```

```
=====
FIRE      0.004
RIGHT     0.005
LEFT      0.004 (Action Taken)
```



```
Action:  Q-Value:
=====
NOOP      0.003
FIRE      0.006
RIGHT     0.008 (Action Taken)
LEFT      0.004
```



```
Action:  Q-Value:
=====
NOOP      0.002
FIRE      0.006
RIGHT     0.007
LEFT      0.003 (Action Taken)
```

✓ Output of Convolutional Layers

The outputs of the convolutional layers can be plotted so we can see how the images from the game-environment are being processed by the Neural Network.

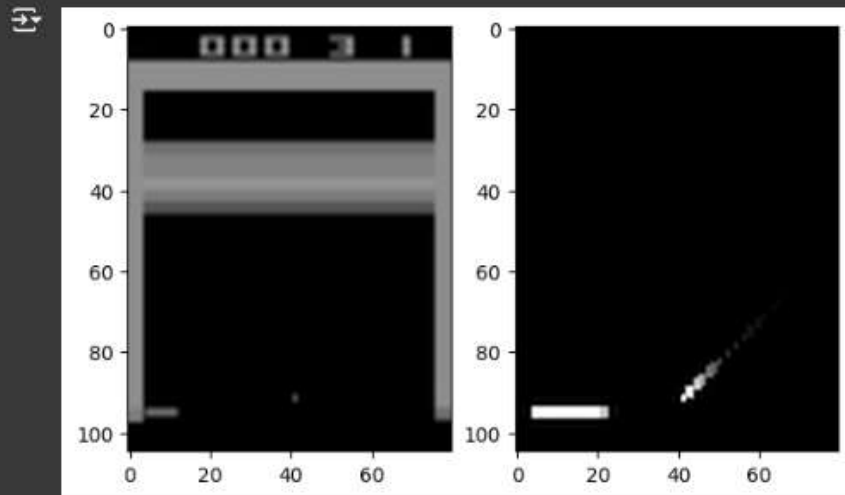
This is the helper-function for plotting the output of the convolutional layer with the given name, when inputting the given state from the replay-memory.

```
[43] 1 def plot_layer_output(model, layer_name, state_index, inverse_cmap=False):
2     """
3     Plot the output of a convolutional layer.
4
5     :param model: An instance of the NeuralNetwork-class.
6     :param layer_name: Name of the convolutional layer.
7     :param state_index: Index into the replay-memory for a state that
8     | | | | | | | | | | will be input to the Neural Network.
9     :param inverse_cmap: Boolean whether to inverse the color-map.
10    """
11
12    # Get the given state-array from the replay-memory.
13    state = replay_memory.states[state_index]
14
15    # Get the output tensor for the given layer inside the TensorFlow graph.
16    # This is not the value-contents but merely a reference to the tensor.
17    layer_tensor = model.get_layer_tensor(layer_name=layer_name)
18
19    # Get the actual value of the tensor by feeding the state-data
20    # to the TensorFlow graph and calculating the value of the tensor.
21    values = model.get_tensor_value(tensor=layer_tensor, state=state)
22
23    # Number of image channels output by the convolutional layer.
24    num_images = values.shape[3]
25
26    # Number of grid-cells to plot.
27    # Rounded-up, square-root of the number of filters.
28    num_grids = math.ceil(math.sqrt(num_images))
29
30    # Create figure with a grid of sub-plots.
31    fig, axes = plt.subplots(num_grids, num_grids, figsize=(10, 10))
32
33    print("Dim. of each image:", values.shape)
34
35    if inverse_cmap:
36        cmap = 'gray_r'
37    else:
38        cmap = 'gray'
39
40    # Plot the outputs of all the channels in the conv-layer.
41    for i, ax in enumerate(axes.flat):
42        # Only plot the valid image-channels.
43        if i < num_images:
44            # Get the image for the i'th output channel.
45            img = values[0, :, :, i]
46
47            # Plot image.
48            ax.imshow(img, interpolation='nearest', cmap=cmap)
49
50            # Remove ticks from the plot.
51            ax.set_xticks([])
52            ax.set_yticks([])
53
54    # Ensure the plot is shown correctly with multiple plots
55    # in a single Notebook cell.
56    plt.show()
57
```

Game State

This is the state that is being input to the Neural Network. The image on the left is the last image from the game-environment. The image on the right is the processed motion-trace that shows the trajectories of objects in the game-environment.

```
[44] 1 idx = np.argmax(q_values_max)
      2 plot_state(idx=idx, print_q=False)
```



Output of Convolutional Layer 1

This shows the images that are output by the 1st convolutional layer, when inputting the above state to the Neural Network. There are 16 output channels of this convolutional layer.

Note that you can invert the colors by setting `inverse_cmap=True` in the parameters to this function.

```
[45] 1 plot_layer_output(model=model, layer_name='layer_conv1', state_index=idx, inverse_cmap=False)
```

Dim. of each image: (1, 53, 40, 16)

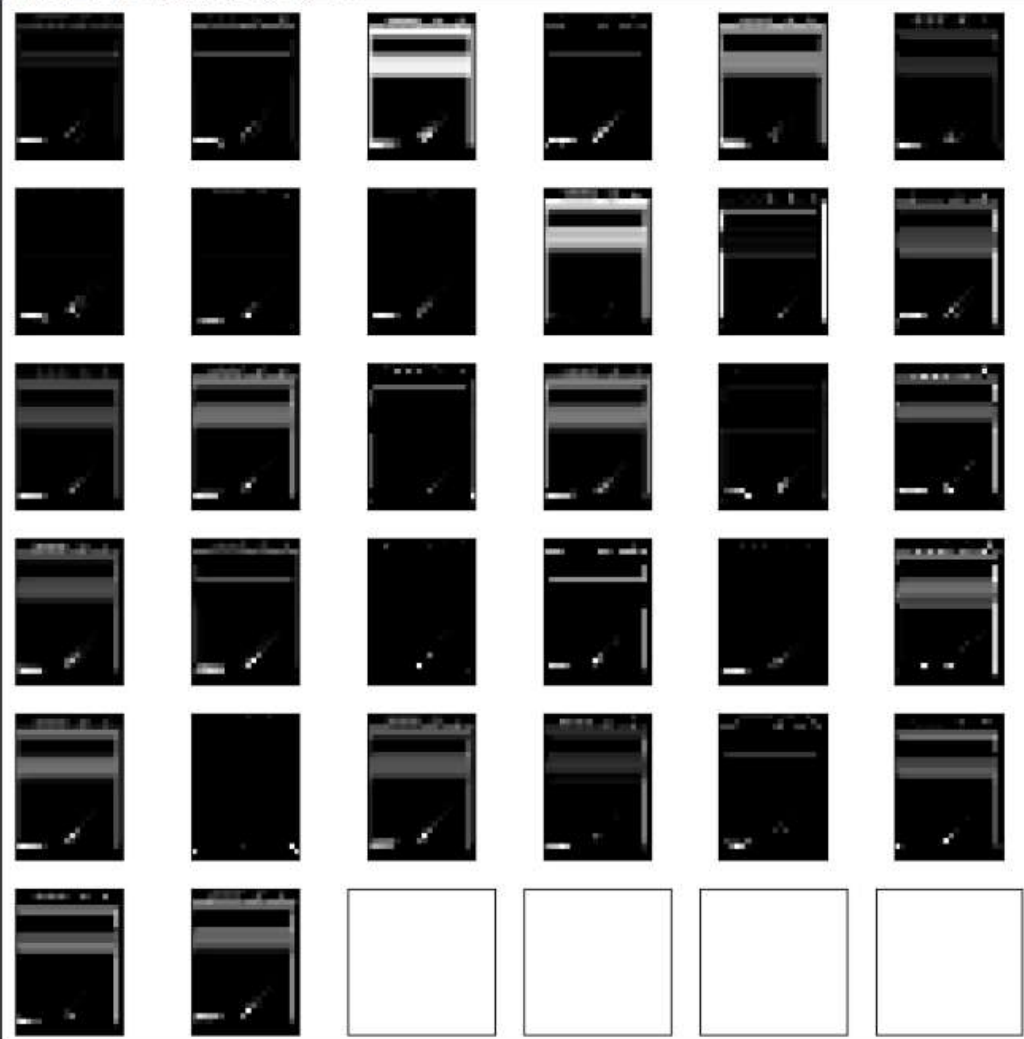


Output of Convolutional Layer 2

These are the images output by the 2nd convolutional layer, when inputting the above state to the Neural Network. There are 32 output channels of this convolutional layer.

```
[46] 1 plot_layer_output(model=model, layer_name='layer_conv2', state_index=idx, inverse_cmap=False)
```

Dim. of each image: (1, 27, 20, 32)



Output of Convolutional Layer 3

These are the images output by the 3rd convolutional layer, when inputting the above state to the Neural Network. There are 64 output channels of this convolutional layer.

All these images are flattened to a one-dimensional array (or tensor) which is then used as the input to a fully-connected layer in the Neural Network.

During the training-process, the Neural Network has learnt what convolutional filters to apply to the images from the game-environment so as to produce these images, because they have proven to be useful when estimating Q-values.

Can you see what it is that the Neural Network has learned to detect in these images?

```
[47] 1 plot_layer_output(model=model, layer_name='layer_conv3', state_index=idx, inverse_cmap=False)
```

Dim. of each image: (1, 27, 20, 64)



✓ Weights for Convolutional Layers

We can also plot the weights of the convolutional layers in the Neural Network. These are the weights that are being optimized so as to improve the ability of the Neural Network to estimate Q-values. Tutorial #02 explains in greater detail what convolutional weights are. There are also weights for the fully-connected layers but they are not shown here.

This is the helper-function for plotting the weights of a convolutional layer.

```
[48] 1 def plot_conv_weights(model, layer_name, input_channel=0):
2     """
3     Plot the weights for a convolutional layer.
4
5     :param model: An instance of the NeuralNetwork-class.
6     :param layer_name: Name of the convolutional layer.
7     :param input_channel: Plot the weights for this input-channel.
8     """
9
10    # Get the variable for the weights of the given layer.
11    # This is a reference to the variable inside TensorFlow,
12    # not its actual value.
13    weights_variable = model.get_weights_variable(layer_name=layer_name)
14
15    # Retrieve the values of the weight-variable from TensorFlow.
16    # The format of this 4-dim tensor is determined by the
17    # TensorFlow API. See Tutorial #02 for more details.
18    w = model.get_variable_value(variable=weights_variable)
19
20    # Get the weights for the given input-channel.
21    w_channel = w[:, :, input_channel, :]
22
23    # Number of output-channels for the conv. layer.
24    num_output_channels = w_channel.shape[2]
25
26    # Get the lowest and highest values for the weights.
27    # This is used to correct the colour intensity across
28    # the images so they can be compared with each other.
29    w_min = np.min(w_channel)
30    w_max = np.max(w_channel)
31
32    # This is used to center the colour intensity at zero.
33    abs_max = max(abs(w_min), abs(w_max))
34
35    # Print statistics for the weights.
36    print("Min: {0:.5f}, Max: {1:.5f}".format(w_min, w_max))
37    print("Mean: {0:.5f}, Stdev: {1:.5f}".format(w_channel.mean(),
38    |                                     w_channel.std()))
39
40    # Number of grids to plot.
41    # Rounded-up, square-root of the number of output-channels.
42    num_grids = math.ceil(math.sqrt(num_output_channels))
43
44    # Create figure with a grid of sub-plots.
45    fig, axes = plt.subplots(num_grids, num_grids)
46
47    # Plot all the filter-weights.
48    for i, ax in enumerate(axes.flat):
49        # Only plot the valid filter-weights.
50        if i < num_output_channels:
51            # Get the weights for the i'th filter of this input-channel.
52            img = w_channel[:, :, i]
53
54            # Plot image.
55            ax.imshow(img, vmin=-abs_max, vmax=abs_max,
56            |         interpolation='nearest', cmap='seismic')
57
58            # Remove ticks from the plot.
59            ax.set_xticks([])
60            ax.set_yticks([])
61
62    # Ensure the plot is shown correctly with multiple plots
63    # in a single Notebook cell.
64    plt.show()
```

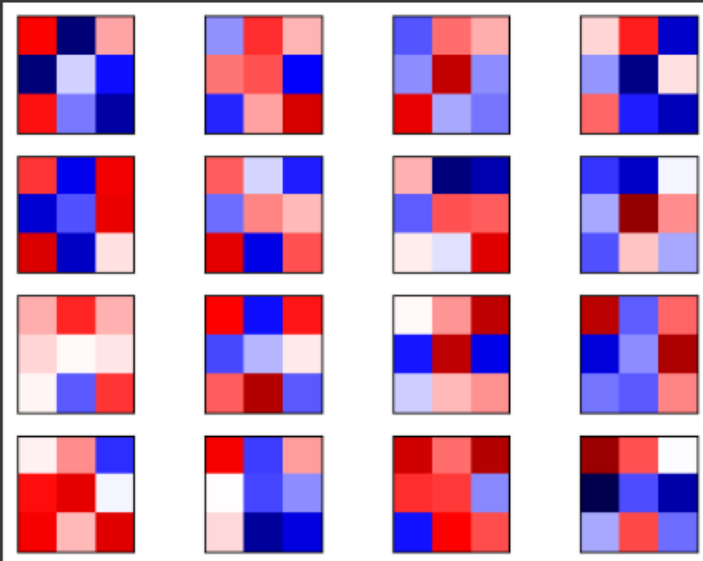
Weights for Convolutional Layer 1

These are the weights of the first convolutional layer of the Neural Network, with respect to the first input channel of the state. That is, these are the weights that are used on the image from the game-environment. Some basic statistics are also shown.

Note how the weights are more negative (blue) than positive (red). It is unclear why this happens as these weights are found through optimization. It is apparently beneficial for the following layers to have this processing with more negative weights in the first convolutional layer.

```
[49] 1 plot_conv_weights(model=model, layer_name='layer_conv1', input_channel=0)
```

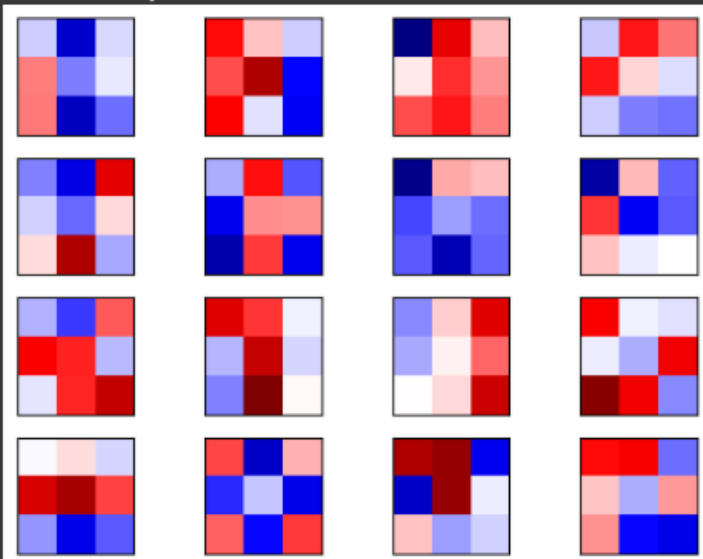
Min: -0.033786, Max: 0.03483
Mean: 0.00054, Stdev: 0.01697



We can also plot the convolutional weights for the second input channel, that is, the motion-trace of the game-environment. Once again we see that the negative weights (blue) have a much greater magnitude than the positive weights (red).

```
[50] 1 plot_conv_weights(model=model, layer_name='layer_conv1', input_channel=1)
```

Min: -0.03322, Max: 0.03933
Mean: 0.00143, Stdev: 0.01653



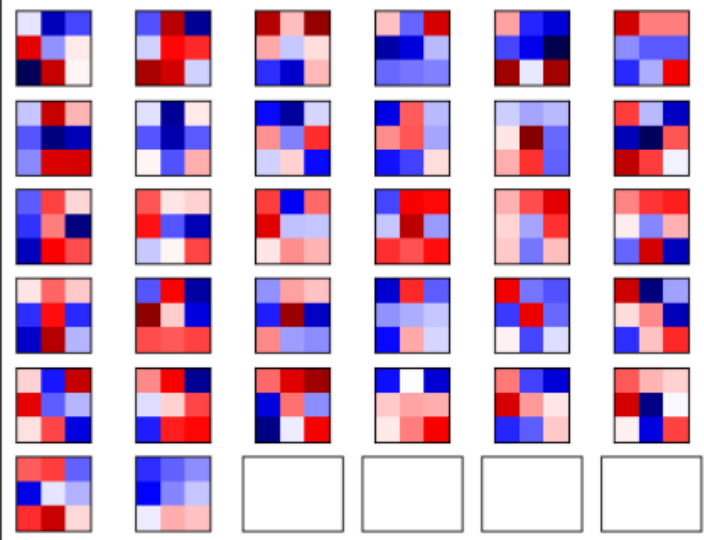
Weights for Convolutional Layer 2

These are the weights of the 2nd convolutional layer in the Neural Network. There are 16 input channels and 32 output channels of this layer. You can change the number for the input-channel to see the associated weights.

Note how the weights are more balanced between positive (red) and negative (blue) compared to the weights for the 1st convolutional layer above.

```
[51] 1 plot_conv_weights(model=model, layer_name='layer_conv2', input_channel=0)
```

Min: -0.03990, Max: 0.03888
Mean: -0.00016, Stdev: 0.01752



Weights for Convolutional Layer 3

These are the weights of the 3rd convolutional layer in the Neural Network. There are 32 input channels and 64 output channels of this layer. You can change the number for the input-channel to see the associated weights.

Note again how the weights are more balanced between positive (red) and negative (blue) compared to the weights for the 1st convolutional layer above.

```
[52] 1 plot_conv_weights(model=model, layer_name='layer_conv3', input_channel=0)
```

Min: -0.03941, Max: 0.03954
Mean: 0.00016, Stdev: 0.01732

