Game Dev with Unity* ML-Agents and Intel-optimized Python* (Part Two)

Abstract
In the final part of this two-part series on machine learning with Unity* ML-Agents, we will dig deeper into the architecture and create an ML-Agent from scratch. Before training, we will inspect the files that require parameters for machine learning to proceed. Finally, we will train the agent using Intel® optimized Python* and show how the completed system works.
Architecture of Unity* ML-Agents

Figure 1 shows the architecture of Unity ML-Agents:

![Architecture of Unity* ML-Agents](image)

Figure 1: Unity* ML-Agents architecture.

At first glance, it might seem that the external communicator and Intel-optimized Python can only be used by the external brain, but this is not the case. The external brain can be accessed by other training modes, too.

Every scene will have two entities:

1. An “Academy,” using an “Academy Script” that will be added later.
2. “Brains,” which are the logic inside Unity ML-Agents where the main connection lies. Agents share the same brain; each agent has an agent script on it which links back to the brain. The brain itself has a brain script on it. It may or may not have a decision script.
Changes in V3 with Respect to V2
Unity ML-Agents have seen several changes, many based on community feedback. Some of the changes are described below:

- The ML-Agents reward system changed to “AddReward()” or “SetReward().”
- When we are working with an Agent and it has worked in its entirety or performed its function, we now use the “Done()” method.
- The concept of state has been changed to observations, so “CollectStates()” have been replaced by “CollectObservations()”.
- When we collect Observations, we have to call “AddVectorObs()” with floats, integers, lists, and an array of floats, vectors, and quaternions. (Quaternions represent the orientation of every object in Unity.) The names of the inputs in the Internal Brain have been changed accordingly.
- We must replace State with “Vector_Observation” and observation with “Visual_Observation.”

The table below summarizes the key changes in V3:

<table>
<thead>
<tr>
<th>Old (V2)</th>
<th>New (V3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>State</td>
<td>Vector Observation</td>
</tr>
<tr>
<td>Observation</td>
<td>Visual Observation</td>
</tr>
<tr>
<td></td>
<td>(New) Text Observation</td>
</tr>
<tr>
<td>Action</td>
<td>Vector Action</td>
</tr>
<tr>
<td></td>
<td>(New) Text Action</td>
</tr>
</tbody>
</table>

Table 1. Changes in Unity* ML-Agents from v2 to v3.

Let’s Start with an Example
Use the following steps to start creating your own example of machine learning using Unity ML-Agents and Intel-optimized Python:

1. Open up the Unity ML cloned project. Everything we do will be kept inside the Examples folder.

   ![Image of Unity project view]

   The cloned project is opened in Unity.

2. Create a new subfolder named “MyBall” within the Examples folder. We will keep all of our resources within this folder.
The Examples folder is where we are keeping all the content and the resources.

3. Create a new scene using the suggested name “MyBall(scene).”

Next, we will create a new scene.
To start setting up machine learning inside the scene, we will have to create 3D objects, using the following steps:

1. Create a 3D object cube.
2. Add “rigid body” and make it “kinematic.”
3. Change the color of the cube. For adding colors to our object, we need to create a new material and name it “Blue.” We will change the color content to blue. (We can also change the color of the background.)
4. Create a 3D object sphere and add a rigid body to it.

We will now organize the scene and add an event system from the UI.

5. Right-click on “Hierarchy” then select “Event System.”

To follow the procedure for Unity ML-Agents, we need to separately create an Academy object and a brain object, and then associate the scripts properly. We will create an Academy object, then have a child object created from Academy named “Brain.” Within the brain, we will add the brain script; but when we do, we will notice an error in the inspector window, which we can quickly resolve.

**Adding Functionality to the Academy and the Brain Object**

When we add functionality to the Academy and Brain object by adding a C# script in it, we remove the error condition. The script follows a basic flow with some override methods. As we have created the ball Academy object, we can now create a C# script named “MyBallAcademy” and attach the script to the Academy in the hierarchy.

Before editing, the script looks like this:

```csharp
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class MyBallAcademy : MonoBehaviour {
    // Use this for initialization
    void Start () {
    }

    // Update is called once per frame
    void Update () {
    }
}
```
We will not inherit from monobehaviour, as we are not deriving any characteristics from it. After we change the script, everything will be derived from Academy and we don’t need “void Start()” and “void Update().”

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class MyBallAcademy : Academy {
    // Use this for initialization
    public override void AcademyReset()
    {
    }

    public override void AcademyStep()
    {
    }
}
```

We have inherited from Academy and have declared two empty override methods as “AcademyReset()” and “AcademyStep().” We cannot change these methods, as this is the structure for any Academy script that you want to derive from. With both of these methods we have made the generalized script that can be used within the scene.

With the changes made to the script, we have a basic, bare-bones structure for linking Academy and the brain.

**Basic Setup for the Scene**

In this scene we will be creating a cube, which we will refer to as the “platform.” Within that platform, we will place a sphere, which will act like a ball. With movements, we can adjust the ball in order to prevent it from falling off the platform. If the ball falls off, the scene will reset, and we will restart the balancing act.

We now have our platform and the ball, but to demonstrate machine learning, we need to configure a brain to control the action. Once the system is under the control of the brain, it will drive the platform and then fire off an agent script. Our next job is to write the agent script.

**Programming and Scene Setup Logic**

We will now create an agent script and name it as MyBallAgent. We will inherit from the Agent. Once we add the MyBallAgent script to the system, we will immediately see what inherited values we need to put in. We will drag and drop Brain to the required inherited values.
First, we will drag and drop the MyBallAgent script created to the cube as shown below.

Then we drag and drop the child we created for Academy as brain to the Brain option, which showed none (shown below).

In the Agent code itself, we will write all the controlling parameters we intend to use. We will declare a GameObject “ball,” which we will include from the inspector that is ball.
public GameObject ball;

Now the flow of the agent is controlled by the Unity ML-Agents plugin. (We will not need Unity’s default update method.)

**Override**

Overriding common methods.

We need to override common methods because the type of environment we created might require changes and more training. For that we need to change the values of the parameters and override the common values present.

First, we have to find out where we are going to have the transformations and other declarations for the game object. In version 0.3, game object changes have been shifted to “AddVectorObs,” which are now known as “vector observations.”

For object transformation, positions, and rigid body, we are declaring eight AddVectorObs (also known as “vector objects”).

The method is called CollectObservations.

```csharp
AddVectorObs(gameObject.transform.rotation.z);
AddVectorObs(gameObject.transform.rotation.x);
AddVectorObs((ball.transform.position.x - gameObject.transform.position.x));
AddVectorObs((ball.transform.position.y - gameObject.transform.position.y));
AddVectorObs((ball.transform.position.z - gameObject.transform.position.z));
AddVectorObs(ball.transform.GetComponent<Rigidbody>().velocity.x);
AddVectorObs(ball.transform.GetComponent<Rigidbody>().velocity.y);
AddVectorObs(ball.transform.GetComponent<Rigidbody>().velocity.z);

SetTextObs("Testing " + gameObject.GetInstanceID());
```

Here is what the above code does:
1. We get the $x$ and $z$ rotation; the game object will rotate in two directions.

AddVectorObs(gameObject.transform.rotation.x);
AddVectorObs(gameObject.transform.rotation.z);

2. We get the difference between the ball’s $x$ position and the game object’s $x$ position.
3. We get where the ball is respective to the platform.
4. We get the ball’s velocity in $x$, $y$ and $z$ directions.

AddVectorObs(ball.transform.GetComponent<Rigidbody>().velocity.x);
AddVectorObs(ball.transform.GetComponent<Rigidbody>().velocity.y);
AddVectorObs(ball.transform.GetComponent<Rigidbody>().velocity.z);

**When the Game Resets, What Method Will We Override?**

The override method that we will be using for when the game resets is AgentReset(), which initiates when the ball is dropped onto the platform. Here are some of the key instructions:

1. Reset everything back to zero:

   gameObject.transform.rotation = new Quaternion(0f, 0f, 0f, 0f);

2. Change the velocity of the ball back to 0:

   ball.GetComponent<Rigidbody>().velocity = new Vector3(0f, 0f, 0f);

3. Set the position of the ball back to StartPos:

   ball.transform.position = ballStartPos;

4. Create “vector 3” to store the vector’s start position:

   Vector3 ballStartPos;

5. Configure the starting position by working inside “Void Start()” and declaring the following:

   ballStartPos = ball.transform.position;

We have now defined the starting environment when we hold the ball for the very first time, and when the system resets.

**Controlling the Platform**

Once we shift to the “Player” option, we must enable certain keys on the keyboard to control movement. We accomplish this by creating a way to physically control the platform. This is where all the actions get converted, and for any desired change for the scene that we have created the response that we do by giving the keyboard movements should produce the results in the scene for the movement of
the ball. We need to check as we map the keyboard keys to ensure that it is reflecting the same way that it is supposed to be. The entire updated code for MyBallAgent is shown below:

```csharp
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class MyBallAgent : Agent {

    public GameObject ball;
    Vector3 ballStartPos;

    void Start()
    {
        ballStartPos = ball.transform.position;
    }

    public override void AgentAction(float[] vectorAction, string textAction)
    {
        if (brain.brainParameters.vectorActionSpaceType == SpaceType.continuous)
        {
            float action_z = 2f * Mathf.Clamp(vectorAction[0], -1f, 1f);
            if ((gameObject.transform.rotation.z < 0.25f && action_z > 0f) ||
                (gameObject.transform.rotation.z > -0.25f && action_z < 0f))
                gameObject.transform.Rotate(new Vector3(0, 0, 1), action_z);
            float action_x = 2f * Mathf.Clamp(vectorAction[1], -1f, 1f);
            if ((gameObject.transform.rotation.x < 0.25f && action_x > 0f) ||
                (gameObject.transform.rotation.x > -0.25f && action_x < 0f))
                gameObject.transform.Rotate(new Vector3(1, 0, 0), action_x);
        }
        SetReward(0.1f);
    }

    if ((ball.transform.position.y - gameObject.transform.position.y) < -2f ||
        Mathf.Abs(ball.transform.position.x - gameObject.transform.position.x) > 3f)
    
        ||
        Mathf.Abs(ball.transform.position.z - gameObject.transform.position.z) > 3f)
    {
        Done();
        SetReward(-1f);
    }
}

public override void CollectObservations()
{
}
```
AddVectorObs(gameObject.transform.rotation.z);
AddVectorObs(gameObject.transform.rotation.x);
AddVectorObs((ball.transform.position.x - gameObject.transform.position.x));
AddVectorObs((ball.transform.position.y - gameObject.transform.position.y));
AddVectorObs((ball.transform.position.z - gameObject.transform.position.z));
AddVectorObs(ball.transform.GetComponent<Rigidbody>().velocity.x);
AddVectorObs(ball.transform.GetComponent<Rigidbody>().velocity.y);
AddVectorObs(ball.transform.GetComponent<Rigidbody>().velocity.z);
SetTextObs("Testing " + gameObject.GetInstanceID());
}

public override void AgentReset()
{
    gameObject.transform.rotation = new Quaternion(0f, 0f, 0f, 0f);
    ball.GetComponent<Rigidbody>().velocity = new Vector3(0f, 0f, 0f);
    ball.transform.position = ballStartPos;
}

Simulation Using Keyboard Inputs

For a simulation using keyboard inputs with the brain type set as “Player,” we will need to configure the brain script. Because there are eight AddVectorObs, the parameter for Vector Observation space size would be eight, and space type is “continuous.” Make the changes in the Inspector window, shown below:

![Inspector Window Configuration](image)

Figure 3: Configuring the brain script in the Inspector window.

Now we can add continuous player actions to control keyboard inputs. There are four keys to map, so there are four continuous player elements: up-arrow, down-arrow, right-arrow, and left-arrow. The parameter values are the following:

Element 0
Key -> Up Arrow  
Index->1  
Value->1

Element 1  
Key->Down Arrow  
Index->1  
Value->-1

Element 2  
Key->Right Arrow  
Index->0  
Value->-1

Element 3  
Key->Left Arrow  
Index->0  
Value->1

The keyboard mapping is shown in the figure below:

![Keyboard Mapping](image.png)

**Figure 4:** Keyboard mapping for elements 0-3.

Now we will click on “Play” to test the scene under player settings and try to keep the ball in the platform using the keyboard arrows up, down, left, and right.

For training the model using Intel-optimized TensorFlow*, we need to keep the brain type set to “external” for the build.
Figure 5: Play starts with the ball at the center of the platform.

As we have done before, we need to create the build for the project.
We have added the scene; now we will create the build and name it.

Now that the executable has been created, we must train it using our Intel-optimized Python module. However, before training can start, there are some things to know about the “learn.py” file and the
“trainer_config.yaml” file. The “learn.py” file contains certain details for running the training. The key parameters are declared in the config file. The main work of the “learn.py” file is to initialize general parameters such as run_id, fast_simulation, etc. and trigger the “trainer_config.yaml” file. We don’t have to make changes to the “learn.py” file; it has the format as shown below:

```python
# # Unity ML Agents
# ## ML-Agent Learning

import logging
import os
from docopt import docopt
from unitytrainers.trainer_controller import TrainerController

if __name__ == '__main__':
    logger = logging.getLogger("unityagents")
    USAGE = 'Usage: learn (<env>) [options]
    learn --help

    Options:
    --curriculum=<file>              Curriculum json file for environment [default: None].
    --keep-checkpoints=<n>           How many model checkpoints to keep [default: 5].
    --lesson=<n>                    Start learning from this lesson [default: 0].
    --load [default: False].
    --run-id=<path>                 The sub-directory name for model and summary statistics [default: ppo].
    --save-freq=<n>                  Frequency at which to save model [default: 50000].
    --seed=<n>                      Random seed used for training [default: -1].
    --slow [default: False].
    --train [default: False].
    --worker-id=<n>                 Number to add to communication port (5005). Used for multi-environment [default: 0].
    --docker-target-name=<dt>      Docker Volume to store curriculum, executable and model files [default: Empty].

    options = docopt(USAGE)
    logger.info(options)
    # Docker Parameters
    if options['--docker-target-name'] == 'Empty':
        docker_target_name = ''
    else:
        docker_target_name = options['--docker-target-name']

    # General parameters
    run_id = options['--run-id']
    seed = int(options['--seed'])
    load_model = options['--load']
    train_model = options['--train']
    save_freq = int(options['--save-freq'])
```

---

The trainer_config.yaml file contains certain details for running the training. The key parameters are declared in the config file. The main work of the “learn.py” file is to initialize general parameters such as run_id, fast_simulation, etc. and trigger the “trainer_config.yaml” file. We don’t have to make changes to the “learn.py” file; it has the format as shown above.
env_path = options['<env>']
keep_checkpoints = int(options['--keep-checkpoints'])
worker_id = int(options['--worker-id'])
curriculum_file = str(options['--curriculum'])
if curriculum_file == "None":
    curriculum_file = None
lesson = int(options['--lesson'])
fast_simulation = not bool(options['--slow'])

# Constants
# Assumption that this yaml is present in same dir as this file
base_path = os.path.dirname(__file__)
TRAINER_CONFIG_PATH = os.path.abspath(os.path.join(base_path,
        "trainer_config.yaml"))

tc = TrainerController(env_path, run_id, save_freq, curriculum_file,
        fast_simulation, load_model, train_model,
        worker_id, keep_checkpoints, lesson, seed,
docker_target_name, TRAINER_CONFIG_PATH)
tc.start_learning()

The “trainer_config.yaml” file contains more important information. Some default parameters are
already declared. The important ones are max_steps :5.0e4. (The max steps are how many times we
loop around and train the entire thing. For this scene it is 50,000 and is written as 5.0e4, which is 5 *
10^4. The value is default.) We can alter the value so that we can train the model more. The number of
times the model is trained is known as “epochs.” Generally, one epoch cycle is known as one full training
cycle on the set or, in this case, is the scene.

α- value or learning rate 3.0e-4

We can also override some values. We can override the value if we need to change the training times
such that we can increase the number of max steps, so that the scene is trained more. This helps us for
better machine-learning results. Within the file there are examples where the default brain script values
have been overridden.

A small snippet of the “config.yaml” file is shown below:

default:
    trainer: ppo
    batch_size: 1024
    beta: 5.0e-3
    buffer_size: 10240
    epsilon: 0.2
    gamma: 0.99
    hidden_units: 128
    lambd: 0.95
    learning_rate: 3.0e-4
    max_steps: 5.0e4
    memory_size: 256
    normalize: false
    num_epoch: 3
    num_layers: 2
time_horizon: 64
sequence_length: 64
summary_freq: 1000
use_recurrent: false

BananaBrain:
normalize: false
batch_size: 1024
beta: 5.0e-3
buffer_size: 10240

PushBlockBrain:
max_steps: 5.0e4
batch_size: 128
buffer_size: 2048
beta: 1.0e-2
hidden_units: 256
summary_freq: 2000
time_horizon: 64
num_layers: 2

Now we can start the training process. The following is the command we will use:

python learn.py mball2.exe --run-id=mball2 -train

As the process runs, the following details are populated:

(idp) C:\Users\abhic\Desktop\ml-agents\python>python learn.py mball2.exe --run-id=mball2 --train
INFO:unityagents:{'--curriculum': 'None',
'--docker-target-name': 'Empty',
'--help': False,
'--keep-checkpoints': '5',
'--lesson': '0',
'--load': False,
'--run-id': 'mball2',
'--save-freq': '50000',
'--seed': '-1',
'--slow': False,
'--train': True,
'--worker-id': '0',
'<env>': 'mball2.exe'}
INFO:unityagents:
'Academy' started successfully!
Unity Academy name: Academy
   Number of Brains: 1
   Number of External Brains : 1
   Lesson number : 0
   Reset Parameters :

Unity brain name: Brain
   Number of Visual Observations (per agent): 0
   Vector Observation space type: continuous
   Vector Observation space size (per agent): 8
   Number of stacked Vector Observation: 3
Vector Action space type: continuous
Vector Action space size (per agent): 2
Vector Action descriptions:
2018-06-04 05:28:49.992671: I k:\tf_jenkins_freddy\ cpu_feature_guard.cc:137] Your CPU supports instructions that this TensorFlow binary was not compiled to use: AVX AVX2
C:\<path>\conda\envs\idp\lib\site-packages\tensorflow\python\ops\gradients_impl.py:96: UserWarning: Converting sparse IndexedSlices to a dense Tensor of unknown shape. This may consume a large amount of memory.
"Converting sparse IndexedSlices to a dense Tensor of unknown shape. "
INFO:unityagents:Hyperparameters for the PPO Trainer of brain Brain:
batch_size: 1024
beta: 0.005
buffer_size: 10240
gamma: 0.99
epsilon: 0.2
hidden_units: 128
lambda: 0.95
learning_rate: 0.0003
max_steps: 5.0e4
normalize: False
num_epoch: 3
num_layers: 2
time_horizon: 64
sequence_length: 64
summary_freq: 1000
use_recurrent: False
graph_scope: summary_path: ./summaries/mball2
memory_size: 256
INFO:unityagents:Saved Model
INFO:unityagents:Saved Model
INFO:unityagents:Saved Model
INFO:unityagents:List of nodes to export :
INFO:unityagents:    action
INFO:unityagents:    value_estimate
INFO:unityagents:    action_probs
INFO:tensorflow:Restoring parameters from ./models/mball2\model-50000.cptk
INFO:tensorflow:Restoring parameters from ./models/mball2\model-50000.cptk
INFO:tensorflow:Froze 12 variables.
INFO:tensorflow:Froze 12 variables.
The bytes file is now generated in the /mball directory.

Figure 8: Directory contents after generating the bytes file.

In our project inside the folder, there is no TFModels directory, so we will have to create one and keep the bytes file there.
Figure 9: Create the TFModels directory to store the bytes file properly.

After creating the bytes file, copy it to the \TFModels folder. Once that step is complete, go back to the Unity project and move to the Inspector window. Change the brain type to “internal.” It will show an error.

Figure 10: After the bytes file is created, set the brain to “internal.”

We can now drag and drop the bytes file (inside the TFModels folder) corresponding to the Graph Model and resolve the error. The system is now ready to test to see how well the model has been trained.

Summary

Intelligent agents, each acting with dynamic and engaging behavior, offer promise for more realism and better user experiences. After completing the tasks described in part one and Part two of this series, you can now create a Unity ML-Agent from scratch, configure the key learning and training files, and understand the key parameters to set up in order to get started with machine learning. Based on what you learned in these articles, you should now be able to incorporate more compelling AI behavior in your own games to boost immersion and attract players.
Resources


Intel Distribution for Python: https://software.intel.com/en-us/distribution-for-python


Bringing gaming to life with AI and deep learning: https://www.oreilly.com/ideas/bringing-gaming-to-life-with-ai-and-deep-learning

Notices

No license (express or implied, by estoppel or otherwise) to any intellectual property rights is granted by this document.

Intel disclaims all express and implied warranties, including without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement, as well as any warranty arising from course of performance, course of dealing, or usage in trade.

This document contains information on products, services and/or processes in development. All information provided here is subject to change without notice. Contact your Intel representative to obtain the latest forecast, schedule, specifications and roadmaps.

The products and services described may contain defects or errors known as errata which may cause deviations from published specifications. Current characterized errata are available on request.

Copies of documents which have an order number and are referenced in this document may be obtained by calling 1-800-548-4725 or by visiting www.intel.com/design/literature.htm.

This sample source code is released under the Intel Sample Source Code License Agreement.

Intel and the Intel logo are trademarks of Intel Corporation or its subsidiaries in the U.S. and/or other countries.

*Other names and brands may be claimed as the property of others.

© 2018 Intel Corporation