Getting Started With Unity* Machine Learning Using Intel-optimized Python* (Part 1)

by Manisha Biswas

Abstract
This article will show game developers how to use reinforcement learning to create better artificial intelligence (AI) behavior. Using Intel-optimized Python*—an improved version of the popular object-oriented, high-level programming language—readers will glean how to train pre-existing machine-language (ML) agents to learn and adapt. In this scenario, we will use Intel-optimized TensorFlow* to run Unity* ML-Agents in the localized environments.

Introduction
Unity ML-Agents are a good way for game developers to learn how to apply concepts of reinforcement learning while creating a scene in the popular Unity engine. We used the ML-Agents plugin to create a simulated environment. We then configure rigorous training to generate an output file from TensorFlow that can be consumed by the created scene in Unity and improve the simulation.

The basic steps are as follows:

1. Start with an introduction to reinforcement learning.
2. Perform the setup from the “requirements.txt” file that installs TensorFlow 1.4, and other dependencies.
3. Train a pre-existing ML-Agent.

System Configuration
The following configuration was used:

* Standard ASUS laptop
What is Reinforcement Learning?
Reinforcement learning is a method for “training” intelligent programs—known as agents—to constantly adapt and learn in a known or unknown environment. The system advances based on receiving points that might be positive (rewards) or negative (punishments). Based on the interaction between the agents and their environment, we then imply which action needs to be taken.

Some important points about reinforcement learning:

- It differs from normal machine learning, as we don’t look at a training dataset.
- It works not with data, but with environments, through which we depict real-world scenarios.
- It is based upon environments, so many parameters come into play as “RL” takes in lots of information to learn and act accordingly.
- It uses potentially large-scale environments that are real-world scenarios; they might be 2D or 3D environments, simulated worlds, or a game-based scenario.
- It relies on learning objectives to reach a goal.
- It obtains rewards from the available environment.

The reinforcement learning cycle is depicted below.

Figure 1: Reinforcement learning cycle.

How the Reward System Works
Rewards work by offering points when single or multiple agents transition from one state to another during interaction with their environment. These points are known as rewards. The more we train, the more rewards we receive, and thus the more accurate the system becomes. Environments can have many different features, as explained below.
Agents

Agents are software routines that make intelligent decisions. Agents should be able to perceive what is happening around them in the environment. The agent is able to perceive the environment based on making decisions that result in an action. The action that the agents perform must be the optimal one. Software agents might be autonomous, or might work together with other agents or people.

![Flow chart showing the environment workflow.](image)

Environments

Environments determine the parameters within which the agent interacts with its world. The agent must adapt to the environmental factors in order to learn. Environments may be a 2D or 3D world or grid.

Some important features of environments:

a) Deterministic
b) Observable
c) Discrete or continuous
d) Single or multiagent

Each of these features is explained below.

Deterministic

If we can logically infer and predict what will happen in an environment based on inputs and actions, the case is deterministic. Being deterministic, the changes that happen are very predictable for the AI, and the reinforcement learning problem becomes easier because everything is known.
Deterministic Finite Automata (DFA)
In automata theory, a system is described as “DFA” if each of its transitions is uniquely determined by its source state and input symbol. Reading an input symbol is required for each state-transition. Such systems work through a finite number of steps and can only perform one action for a state.

Non-Deterministic Finite Automata (NDFA)
If we are working in a scenario where it cannot be guaranteed which exact state the machine will move into, then it is described as “NDFA.” There is still a finite number of steps, but the transitions are not unique.

Observable
If we can say the environment around us is fully observable, that environment is suitable for implementing reinforcement learning. If you consider a chess game, the environment is predictable, with a finite number of potential moves. In contrast, a poker game is not fully observable, because the next card is unknown.

Discrete or continuous
Continuing with the chess/poker scenarios, when the next choice for a move or play is limited, it is in a discrete state. If there are multiple possible states, we call it continuous.

Single or multiagent
Solutions in reinforcement learning can use single agents or multiple agents. When we are dealing with non-deterministic problems, we use multiagent reinforcement learning. The key to understanding reinforcement learning is in how we use the learning techniques. In multiagent solutions, the agent interactions between different environments is enormous. The key is understanding what kind of information is generally available.

Single agents cannot tackle convergence, so when there is some portion of convergence in reinforcement learning it is handled by multiple agents in dynamic environments. In multiagent models, each agent’s goals and actions impact the environment.

The following figures depict the differences between single-agent and multiagent models.
Figure 3: Single-agent system.
Getting Started
We will be using the Unity Integrated Development Engine (IDE) to demonstrate reinforcement learning in game-based simulations. After creating the simulation from scratch, we will use Unity ML-Agents to showcase how reinforcement learning is implemented in the created project and observe how accurate the results are.

Step 1: Create the environment
To start, we will create an environment for Intel-optimized Python.

Prerequisites
Make sure you have the Anaconda* IDE installed. Anaconda is a free and open-source distribution of the Python programming language for data science and machine learning-related applications. Through Anaconda, we can install different Python libraries which are useful for machine learning.

The download link is here: https://www.anaconda.com/download/.
The command to create a new environment with an Intel build is shown below.

```
conda create -n idp intelpython3_core python=3
```

After all dependencies are installed we proceed to step 2.

**Step 2: Activate the environment**

Now we will activate the environment. The command is shown below.

```
source activate idp
```

**Step 3: Inspect the environment**

As we have activated the environment, let us check the Python version. (It should reflect the Intel one.)

```
(idp) C:\Users\abhic>python
Python 3.6.3 |Intel Corporation| (default, Oct 17 2017, 23:26:12) [MSC v.1900 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
Intel® Distribution for Python is brought to you by Intel Corporation.
Please see: https://software.intel.com/en-us/python-distribution
```

**Step 4: Clone the GitHub repository**

We need to clone or copy the Unity ML repo from the GitHub link while inside the activated Intel-optimized environment (i.e., named idp). To clone the repo, we use the following command:

```
(idp) C:\Users\abhic\Desktop>git clone https://github.com/Unity-Technologies/ml-agents.git
```

**Step 5: Install requirements**

As cloning is proceeding, we need to install certain requirements. The requirements.txt is found in the Python subdirectory.

```
(idp) C:\Users\abhic\Desktop\ml-agents\python>pip install -r requirements.txt
```

This will install the mandatory dependencies.

**Step 6: Create the build**

The build is created inside the Unity IDE and the executable is generated. The crawler executable is shown below.
Step 7: Optimize the build

To make the training go faster with Intel-optimized Python, issue the following command from the Python subdirectory:

(idp) C:\Users\abhic\Desktop\ml-agents\python>python learn.py manisha.exe --run-id=manisha --train

Once the training has completed a full run, we get the byte file needed to use inside the brain, within the child object of Academy:

INFO: unityagents: Saved Model
INFO: unityagents: Ball3DBrain: Step: 50000. Mean Reward: 100.000. STD of Reward: 0.000.
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/manisha/model-50000.cptk
INFO: tensorflow: Restoring parameters from ./models/manisha/model-50000.cptk
INFO: tensorflow: Froze 15 variables.
INFO: tensorflow:Froze 15 variables.
Converted 15 variables to const ops.
The byte file we generated is used for making the simulation work with machine learning.

**Advantages of Using Intel-optimized Python**

Python was not designed for multithreading. It runs on one thread, and while developers can run code on other threads, those threads cannot easily access any Python objects. Intel-optimized Python features thread-enabled libraries, so consider Intel® Threading Building Blocks as a potential tool for multithreading.

The advantages of using Intel-optimized Python with Unity ML-Agents are as follows:

- The training process is much faster.
- The CPU version of TensorFlow involves less overhead.
- Handling multiple agents using the Intel-optimized pipeline is easier and faster.

**Unity ML-Agents v 0.3**

Unity ML-Agents are constantly evolving, with updates responding to community feedback. ML-Agents are based on imitation learning, which is different from reinforcement learning. The most common imitation learning method is “behavioral cloning.” Behavioral cloning is a method that we apply to neural networks (specifically Convolutional Neural Networks, or CNNs) to replicate a behavior such as a self-driving car environment, where the goal is for the system is to drive the car as humans do.

**Imitation Learning**

Generally, when we are talking about “imitation learning,” we refer to learning by demonstration. The demonstration is based upon the learning behavior patterns we get while analyzing and generating learning signals to the agent. In the table below, you can see the differences between imitation learning and reinforcement learning.

<table>
<thead>
<tr>
<th>Imitation Learning</th>
<th>Reinforcement Learning</th>
</tr>
</thead>
<tbody>
<tr>
<td>The process of learning happens through demonstration.</td>
<td>Involves learning from rewards and punishments.</td>
</tr>
<tr>
<td>No such mechanism for rewards and punishments are required. Rewards are not necessary.</td>
<td>Based on trial-and-error methods.</td>
</tr>
<tr>
<td>Generally evolved for real-time interaction.</td>
<td>Specifically meant for speedy simulation methods.</td>
</tr>
<tr>
<td>After training, the agent becomes “human-like” at performing a task.</td>
<td>After training, the agent becomes “optimal” at performing a task.</td>
</tr>
</tbody>
</table>

*Table 1: Imitation learning versus reinforcement learning.*

**TensorFlowSharp**

In this section, we will cover some basics of TensorFlowSharp. First released in 2015, TensorFlow is Google’s open-source library for dataflow programming and framework for deep learning. TensorFlow doesn’t provide a C# native API, and the internal brain as it is written in C# is not natively supported. To enable the internal brain for machine learning, we need to utilize TensorFlowSharp, a third-party library which has the specific purpose of binding the .NET framework to TensorFlow.

**Running the Examples**

We will now go through an example of a Unity ML-Agents project to implement imitation learning. The process will involve the following steps.
1. Include the TensorFlowSharp Unity Plugin.
2. Launch Unity IDE.
3. Find the example folder which is inside Assets. There is a subset folder within the ML-Agents project folder named “Examples.” We will work with the example named Crawler. Every change will occur inside this folder.

As we are working to create an environment for training, we will have to set the brain used by the agents to External. Doing this will allow the agents to communicate with the external training process when they are trying to make decisions.

We are exploring the example project “Crawler.” The setup is a creature with four limbs, from which extend a shorter limb, or forearm (see Figure 5 above). For this scenario to be successful, we have the following goal: The agent must move its body along the x axis without falling.

We need to set some parameters to fine-tune the example. The environment contains three agents linked to a single brain. Inside Academy, locate the child object “CrawlerBrain” within the Inspector window. Set the Brain type to External.

Next, open Player Settings:

1. Go to the menu Edit > ProjectSetting> Player.
2. Go to the options Resolution and Presentation.
   Check that “Run in Background” is selected. Then check that the resolution dialog is set to “disabled.” Finally, click “Build.” Save within the Python folder. Name it “Manisha1” and then save it.

![Figure 6: Saving the build within the Python* folder.](image)

Train the Brain
Now we will work on training the brain. To open the Anaconda prompt, use the search option in Windows and type in “Anaconda.” The Anaconda prompt will open. Once inside the Anaconda prompt, we need to find out the environments available.
We will pass the following command:

(base) C:\Users\abhic>conda --envs

We are displaying the contents of the folder to make it easier to identify the files that reside inside in the Python subfolder. As major changes or the supportive code all resides within this subfolder it is easier and efficient to make changes to the way we are going to train the Machine learning agents within the subfolder. The python subfolder is important because the default code and other supportive library reside within this subfolder. As we have created the build for the Unity scene, we see that one auto-executable file is generated, along with data folders named "manisha1.exe" and "manisha1_Data."

Directory of C:\Users\abhic\Desktop\ml-agents\python

28-05-2018 06:28   <DIR>          ..
21-05-2018 11:34   6,635 Basics.ipynb
21-05-2018 11:34   <DIR>          curricula
21-05-2018 11:34   2,685 learn.py
29-01-2018 13:48   650,752 manisha.exe
29-01-2018 13:24   650,752 manisha1.exe
28-05-2018 06:28   <DIR>          manisha1_Data
21-05-2018 11:58   <DIR>          manisha_Data

We will pass the following command:

(base) C:\Users\abhic>activate idp

Intel-optimized Python and Intel-optimized TensorFlow are installed in the environment idp. Next, we will activate the idp by opening the cloned folder in the desktop.

(idp) C:\Users\abhic\Desktop\ml-agents>

As we have saved the .exe file in the Python subdirectory, we will locate it there.

(idp) C:\Users\abhic\Desktop\ml-agents>cd python

Using the directory command “dir” we can list the items in the Python subfolder:

28-05-2018 06:28   <DIR>          ..
21-05-2018 11:34   6,635 Basics.ipynb
21-05-2018 11:34   <DIR>          curricula
21-05-2018 11:34   2,685 learn.py
29-01-2018 13:48   650,752 manisha.exe
29-01-2018 13:24   650,752 manisha1.exe
28-05-2018 06:28   <DIR>          manisha1_Data
21-05-2018 11:58   <DIR>          manisha_Data
Look inside the subdirectory to locate the executable “manisha1.” We are now ready to use Intel-optimized Python and Intel-optimized TensorFlow to train the model. For training, we will use the learn.py file. The command for using Intel-optimized Python is shown below.

(idp) C:\Users\abhic\Desktop\ml-agents\python>python learn.py manisha1.exe --run-id=manisha1 --train

(idp) C:\Users\abhic\Desktop\ml-agents\python>python learn.py manisha1.exe --run-id=manisha1 --train

INFO:unityagents: {'--curriculum': 'None',
    '--docker-target-name': 'Empty',
    '--help': False,
    '--keep-checkpoints': '5',
    '--lesson': '0',
    '--load': False,
    '--run-id': 'manisha1',
    '--save-freq': '50000',
    '--seed': '-1',
    '--slow': False,
    '--train': True,
    '--worker-id': '0',
    '<env>': 'manisha1.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: CrawlerBrain
- Number of Visual Observations (per agent): 0
- Vector Observation space type: continuous
- Vector Observation space size (per agent): 117
- Number of stacked Vector Observation: 1
- Vector Action space type: continuous
- Vector Action space size (per agent): 12
- Vector Action descriptions: , , , , , , , , , , ,

2018-05-28 06:57:56.872734: I C:\tf_jenkins\home\workspace\rel-win\M\windows\PY36\tensorflow\core\platform\cpu_feature_guard.cc:137] Your CPU supports instructions that this TensorFlow binary was not compiled to use: AVX AVX2

C:\Users\abhic\AppData\Local\conda\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 CrawlerBrain:
  - batch_size: 2024
  - beta: 0.005
  - buffer_size: 20240
  - epsilon: 0.2
  - gamma: 0.995
  - hidden_units: 128
  - lambda: 0.95
  - learning_rate: 0.0003
  - max_steps: 1e6
  - normalize: True
  - num_epoch: 3
  - num_layers: 2
  - time_horizon: 1000
The parameters above set up the training process. After the training process is complete (it can be lengthy) we get the following details in the console:

INFO: unityagents: Saved Model
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/manisha1/model-1000000.cptk
INFO:tensorflow: Restoring parameters from ./models/manisha1/model-1000000.cptk
INFO:tensorflow: Froze 15 variables.
INFO:tensorflow: Froze 15 variables.
Converted 15 variables to const ops.

**Integration of the Training Brain with the Unity Environment**

The idea behind using Intel-optimized Python is to make the training process more accurate. Some examples will require more time to complete the training process because of the large number of steps.

Since TensorFlowSharp is still in the experimental phase, it is disabled by default. To enable TensorFlow and make the internal brain available, follow these steps:

1. Make sure that the TensorFlow plugin is present in the Assets folder. Within the Project tab, navigate to this path: Assets->ML-Agents->Plugins->Computer.
2. Open the Edit->projectSettings->Player to enable TensorFlow and .NET support. Elect Scripting Runtime Version to Experimental(.net 4.6).
3. Open the Scripting-defined symbols and add the following text: ENABLE_TENSORFLOW.
4. Press the Enter key and save the project.

**Bringing the Trained Model to Unity**

After the training process is over, the TensorFlow framework creates a byte file for the project. Locate the model created during the training process under crawler/models/manisha1.

The executable file generated in the build for the Crawler scene will be the name we use for the next file to be generated. The file name will be the name of the environment with the extension of the bytes file when the training is complete.

If “manisha1.exe” is the executable file, then the byte file generated will be “manisha1.bytes,” which follows the convention <env-name>.bytes.

a. Copy the generated bytes file from the models folder to the TF Models subfolder.
b. Open up the Unity IDE and select the crawler scene.
c. Select the brain from the scene hierarchy.
d. Change the type of brain to internal.
e. Drag the .bytes file from the project folder to the graph model placeholder in the brain inspector, and hit play to run it.

The output should look similar to the screenshot below.
We then build the project with the internal brain. An executable is generated.

Figure 8: The executable after building the project with the internal brain.

Summary
Unity and Intel are lowering the entry barrier for game developers who seek more compelling AI behavior to boost immersion. Intelligent agents, each acting with dynamic and engaging behavior, offer promise for more realism and better user experiences. The use of reinforcement learning in game development is still in its early phase, but it has the potential to be a disruptive technology. Use the techniques and resources listed in this article to get started creating your own advanced game-play, and see what the excitement is all about.

Resources

Intel Distribution or 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, the Intel logo, and Core are trademarks of Intel Corporation in the U.S. and/or other countries.

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

Microsoft and Windows are trademarks or registered trademarks of Microsoft Corporation in the United States and/or other countries.

© 2018 Intel Corporation