https://blogs.unity3d.com/2017/12/11/using-machine-learning-agents-in-a-real-game-a-beginners-guide/

Using Machine Learning Agents Toolkit in a real game: a beginner's guide

Alessia Nigretti, December 11, 2017

Community Machine Learning Technology

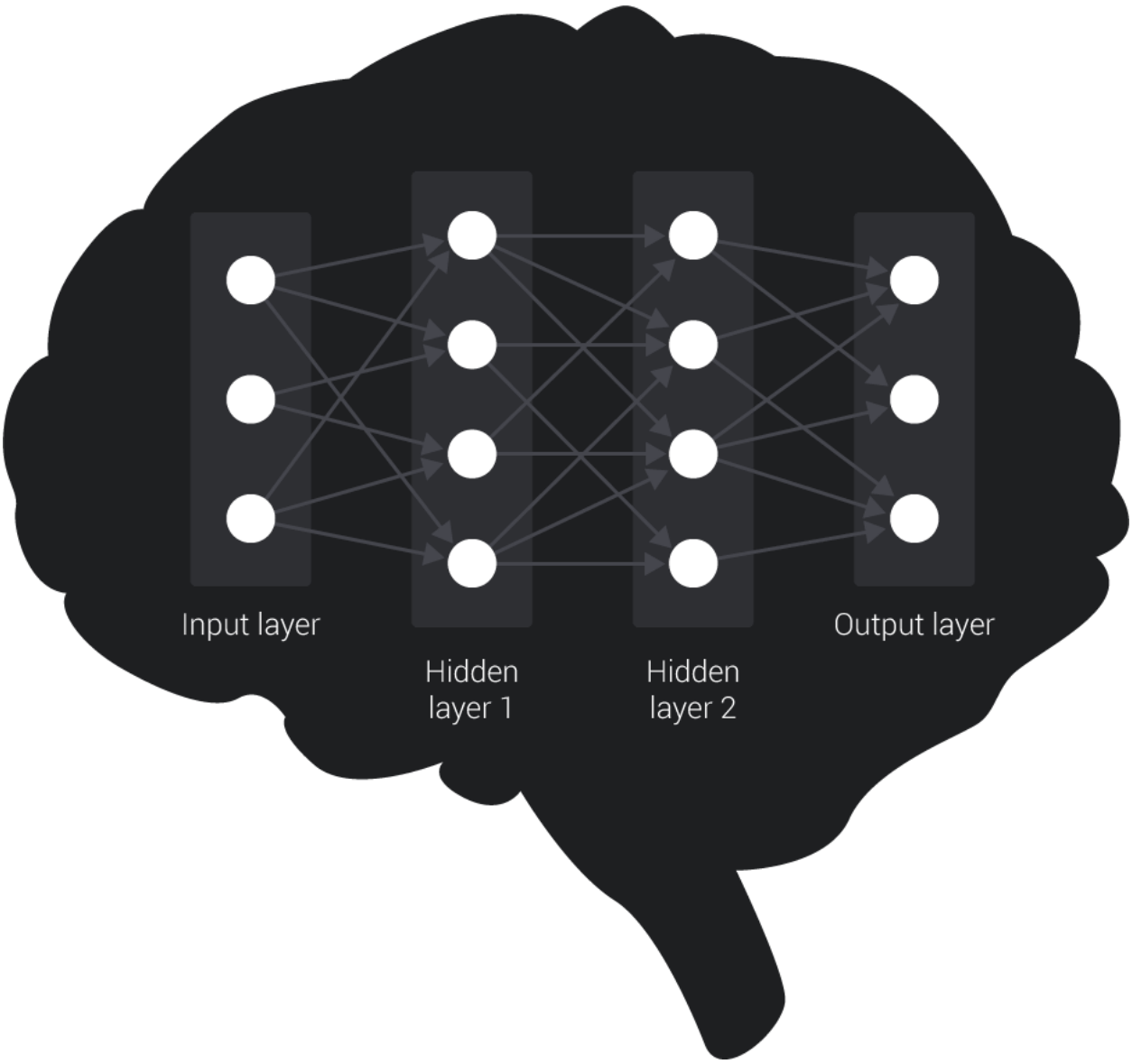
My name is Alessia Nigretti and I am a Technical Evangelist for Unity. My job is to introduce Unity's new features to developers. My fellow evangelist Ciro Continisio and I developed the first demo game that uses the new Unity Machine Learning Agents toolkit and showed it at DevGamm Minsk 2017. This post is based on our talk and explains what we learned making the demo. At the same time, we invite you to join the ML-Agents Challenge and show off your creative use-cases of the toolkit.

Last September, the Machine Learning team introduced the Machine Learning Agents toolkit with three demos and a blog post. The news ignited the interest of a huge number of developers who are either technical experts in the field of Artificial Intelligence or just interested in the ways Machine Learning is changing the world and the way we make and play games. As a consequence, the team has recently launched a new beta version (0.2) of the ML-Agents Toolkit and announced the very first community challenge dedicated to Unity Machine Learning Agents.

Ciro and I aren't experts in Machine Learning, and we both wanted to highlight this during our talk. Since one of our main values at Unity is to **democratise game development**, we wanted to make sure that this new feature was available to everyone who felt like diving into it. First, we engaged with the Machine Learning team and produced some small demos to understand how this was working from the bottom. We had a lot of questions, and we figured those would be the same questions that a developer that is dealing with Machine Learning Agents for the first time would ask. So we wrote them all down and went through them during our talk. If you are reading this and you have no idea about what Machine Learning is or how it works, you can still find your way through the world of the Unity Machine Learning Agents Toolkit!
Our demo **Machine Learning Roguelike** is a 2D action game where the player has to fight ferocious smart slimes. These Machine Learning-controlled enemies will attack without giving our hero a break, and will run away when they feel like their life is in danger.

What is Machine Learning?

Let's start with an introduction to Machine Learning. **Machine Learning** is an application of artificial intelligence that provides a system with the ability of learning from data autonomously, rather than in an engineered way. This works by feeding the system with information and observations that can be used to find patterns and make predictions about future outcomes. More broadly, the system has to learn the desired input-output mapping. This way, the system will be able to choose the best action to perform next in order to optimise its outcome.
There are different ways of doing this, that we can choose depending on what kind of observations we have available to feed our system. The one used in this context is **Reinforcement Learning**. Reinforcement Learning is a learning method based on not telling the system what to do, but only what is right and what is wrong. This means that we let the system perform random actions and we provide a reward or a punishment when we think that these actions are respectively right or wrong. Eventually, the system will understand that it has to perform certain actions in order to get a positive reward. We can think about it as teaching a dog how to sit: a dog will not understand what we mean, but if we feed it a treat when it does the right thing, it will eventually associate the action with the reward.



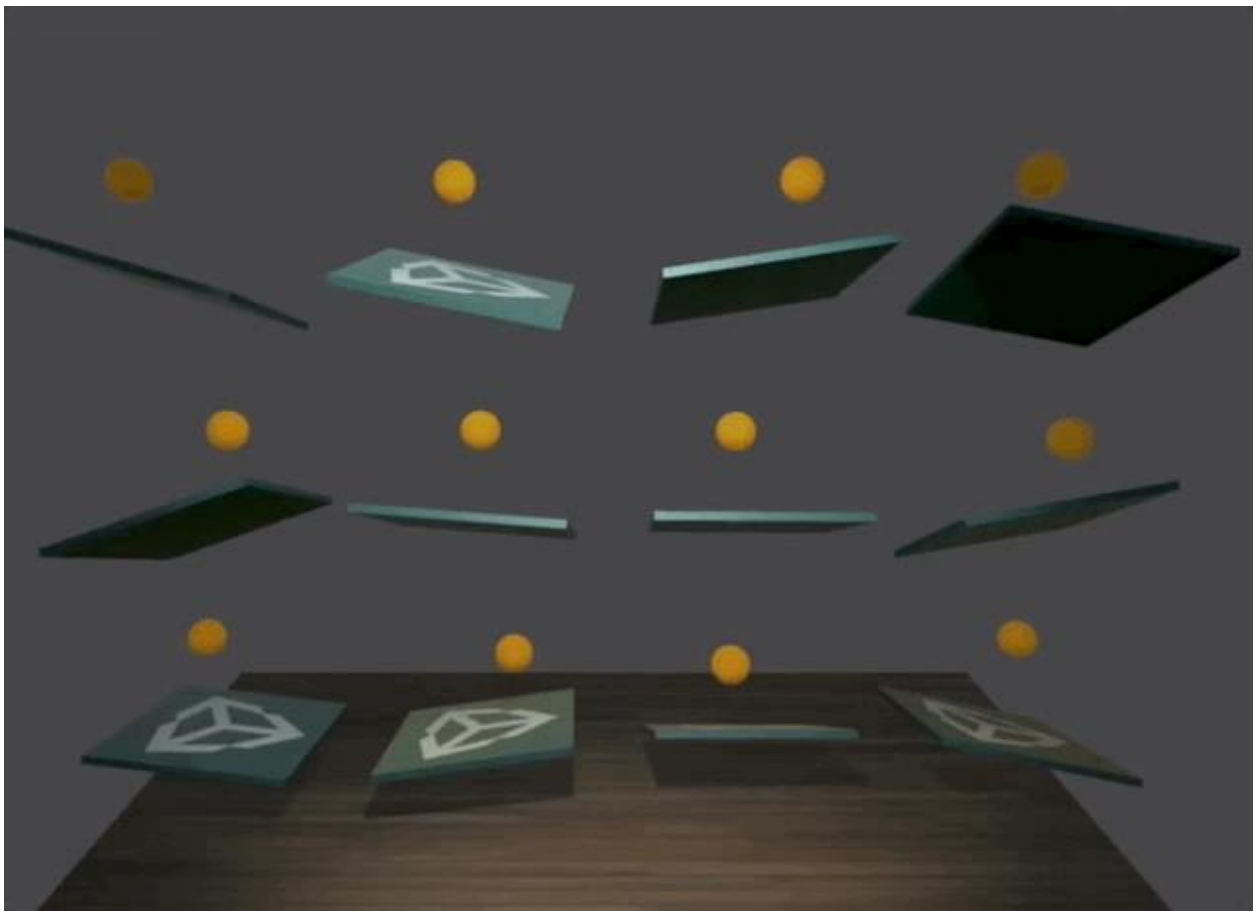
We use Reinforcement Learning to train a **Neural Network**, which is a computer system modeled on the nervous system. Neural Networks are structured in units, or “neurons”. Neurons are divided into layers. The layer that interacts with the external environment and gathers all input information is the Input layer. Opposite to that, the Output layer contains all neurons that store information about the outcome of a certain input in the network. In the middle, the Hidden layers contain the neurons that perform all the calculations. They learn complex abstracted representations of the input data, which allow their outputs to ultimately be “intelligent”. Most layers are “fully connected”, which means that neurons in that layer are connected to all neurons in the previous layer. Each connection is defined by a “weight”, which is a numerical value that can either strengthen or weaken the link between the neurons.

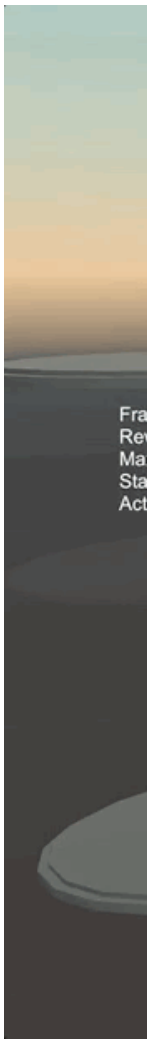
In Unity, our “agents” (the entities who perform the learning) use the Reinforcement Learning model. The agents perform actions on an environment. The actions cause a change in the environment, which in turn is fed back to the agent, together with some reward or punishment. The action happens in what we call the Learning Environment, which in practical terms corresponds to a regular Unity scene.

Within the Learning Environment, we have an Academy, which is a script that defines the properties of the training (framerate, timescale…). The Academy passes these parameters to the Brains, which are entities that contain the model to train. Finally, the Agents are connected to the Brains, from which they get the actions and feed information back to facilitate the learning process. To perform the training, the system uses an extra module that allows the Brains to communicate with an external Python environment using the TensorFlow Machine Learning library. Once the training is complete, this environment will distill the learning process into a “model”, a binary file that gets reimported into Unity to be used as a trained Brain.

Applying Machine Learning to a real game

After playing around with simple examples like 3D Ball and other small projects that we created in a few hours (see gifs), we were fascinated by the potential of this technology.





We found a great application for it in a 2D Roguelike game. To build the scene, we also used new features such as 2D Tilemaps and Cinemachine 2D.

The idea behind our Machine Learning Roguelike demo was to have a simple action game where all entities are Machine Learning agents. This way, we established a common interaction language that can be used both by the players and the enemies that they encounter. The goal is very simple: moving around and surviving enemy encounters.

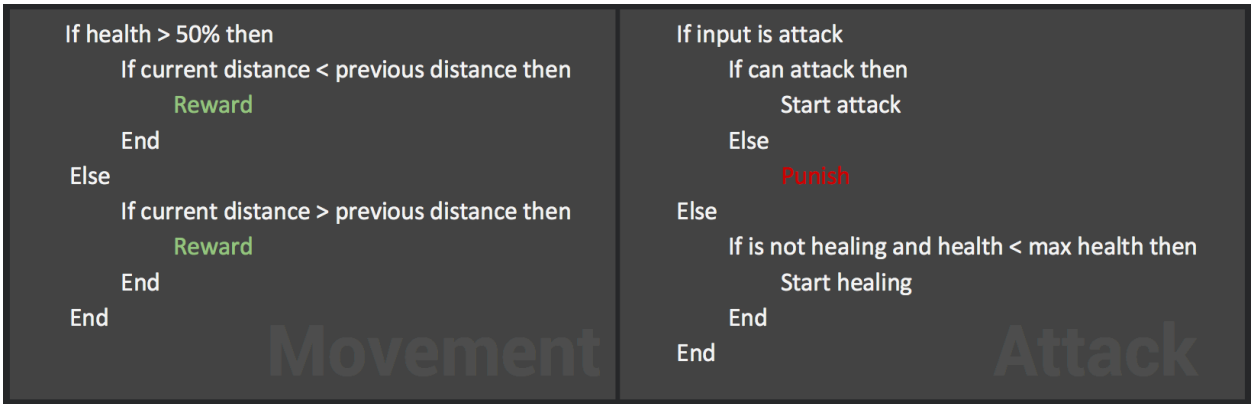
Setting up the training

Every good training starts with a high-level brainstorming session on the training algorithm. This is the code that runs every frame into the agent's class, determining the effect of inputs on the agent and the rewards they generate. We started off by defining what the base actions of the game are (move, attack, heal…), how they connect with each other, what we wanted the agent to learn in respect to them, what we were going to reward or punish.

One of the first decisions to make is whether to use Discrete or Continuous space for States and Actions. Discrete means that the states and actions can only have one value at a time – it is a simplified version of a real environment, therefore the agents can associate actions with rewards more easily. In our example, we use Discrete actions, which we allow to have 6 values: 0: stay still / 1-4: move in one direction / 5: attack. Continuous means that there can be multiple states or actions, and they all have float values. However, they are harder to use as their variability can confuse the agent. In Roguelike, we use Continuous states to check for health, target, distance from target etc.

The reward function

The initial algorithm we came up with was allowing the agent to earn reward if, when not in a situation of danger, its distance from the target was decreasing. Similarly, when in a situation of danger, it would earn reward if its distance from the target was increasing (the agent was "running away"). Additionally, the algorithm was including a punishment given to the agent in case it was attacking when not allowed to.

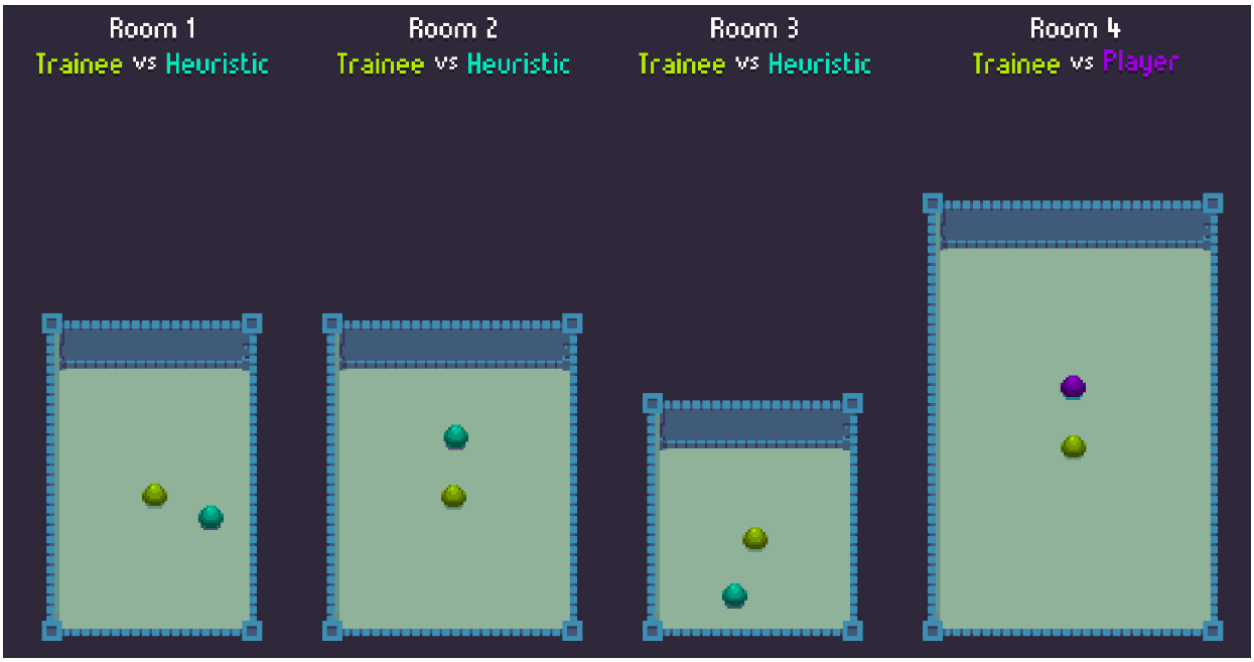


After days and days of testing, this initial algorithm changed a lot. What we learned is that the best way of reproducing the behaviour that we wanted to obtain, was to follow a simple pattern: try, fail, learn from your failure, loop. Like we do when we want the machine to learn from the observations we provide, we also want to learn from the observations that we make through trial and error. One of the things that we noticed, for example, is that the agent will always find a way to exploit the rewards: what happened in our case is that the agent started moving back and forth because for every time it would move forward again, it would get some reward. It basically found a way of taking advantage of our algorithm to get the most reward out of it!

What we learned

From our research for a suitable solution, we learned many different lessons that we managed to apply successfully to our final project. It is fundamental to keep in mind that rewards can be assigned at any moment of the gameplay, whenever we think that the agent is doing something right.

The training scene



After setting up the algorithm, the next step was to set-up the training scene. We chose to set-up a small scene to serve as the training environment, and then use the agents in gameplay in a different, bigger scene where we export the trained model. We decided to create four different rooms, each one with different parameters in terms of length and width, so that our agent doesn't get used to a specific type of room, but considers all the possible variables.

The training scene is responsible for configuring the position of the agents, their connections to the Brains and set-up of the academy and parameters.

What we learned

From setting up the training scene, we learned that parallelizing training instead of just repeating the same situation several times makes the training more solid, as the agents have more data to work with and learn from. To do this, it is useful to set-up a few Heuristic agents to perform simple actions that help the agents learn, without leading them into making wrong assumptions. For instance, we set-up a simple Heuristic script to make our agents attack randomly, to provide our training agents with an example of what happens when they get receive damage.

Once our environment is ready, we can launch the training. If we're planning on launching a very long training, it is good to double-check that the logic of your algorithm makes sense. This isn't about compile-time errors or whether you are a good

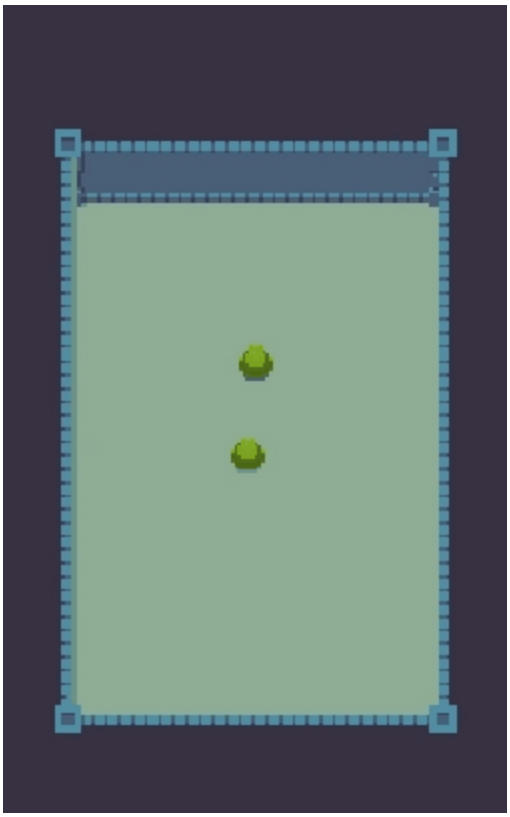
programmer or not – as I mentioned, the agents will find a way of exploiting your algorithm, so make sure that there is no logic gap. To be even more sure, launch a 1x speed training to see what happens at each frame. Observe it and see if your agent is acting like you were expecting it to.

Training

Once everything is ready for training, we need to create a build that will interact with the TensorFlow environment in Python.
First, we set the Brain that has to be trained externally as “External”, then we build the game. Once this is done, we open the Python environment, set the hyperparameters and launch the training. In the Machine Learning Readme page you can find information on how to build for training and how to choose your hyperparameters.

While we’re running the training, we can observe the mean reward that our agents are earning. This should be increasing slowly, until it stabilises – which should happen when the agents have stopped learning.

When we’re happy with our model, we import it back into Unity and see what happens.



The gif shows the result of ~1h training: as we wanted, our slime gets closer to the other agent to attack until it gets attacked back and loses most of its health. At that point, it runs away to give itself time to heal, then goes back to attack the opponent.

Testing

When we apply this model to our Roguelike game, we can see that it works the same way. Our slimes have learnt how to act intelligently and to adapt to scenarios that are potentially different from the ones where they have been trained in.

What now?

This article was meant to provide a little demonstration of how much you can do with the Unity’s Machine Learning Agents API without necessarily having any specific technical

knowledge about Machine Learning. If you want to take a look at this demo, it is available in the Evangelism Team repository.
There is a lot more that you can do to implement Machine Learning in a game that you have already started to work on: one example is to mix trained and heuristics, to hard-code behaviour that would be too complex to train, but also give some realism to your games using ML-Agents.

If you want to add more finesse to the training, you can also put together information about the game style of real players, their strategies and intentions through a closed beta or internal playtesting. Once you have this data, you can further refine the training of your agents to get the next level of AI complexity. And if you feel adventurous, you can try building your own Machine Learning models and algorithm to obtain even more flexibility!

Now that you learned how to use ML-Agents, come and join the ML-Agents Challenge! Look forward to seeing you there!

Recommended Readings:

- **Introducing ML-Agents Toolkit v0.2: Curriculum Learning, new environments, and more**
- **Introducing: Unity Machine Learning Agents Toolkit**
- **Unity AI – Reinforcement Learning with Q-Learning**
- **Unity AI-themed Blog Entries**