The Thirty-Second International Florida
Artificial Intelligence Research Society Conference (FLAIRS-32)

Dynamic Action Selection in OpenAI Using Spiking Neural Networks

Chad Peters,[1] Terrance Stewart,[2] Robert West,[3] Babak Esfandiari[4]
1,3,4 Carleton University, Ottawa ON, Canada
2 University of Waterloo, Waterloo ON, Canada

Abstract

Modelling biologically-plausible neural structures for intelligent agents presents a unique challenge when operating in real-time domains. Neurons in our brains have different response properties, firing rates, and propagation lengths, creating noise that cannot be reliably decoded. This research explores the strengths and limitations of LIF spiking neuron ensembles for application in OpenAI virtual environments. Topics discussed include how we represent arbitrary environmental signals from multiple senses, choosing between equally viable actions in a given scenario, and how one can create a generic model that can learn and operate in a verity of situations.

Introduction

This paper describes research on modeling biologically-plausible neural ensembles for action selection and initiation in virtual environments, and viable approaches to the suppression of competing impulses as one might find in our own basal ganglia nuclei (Stewart, Choo, & Eliasmith, 2010) by constructing a biologically plausible agent to play virtual games, such as Lunar Lander.

The main contributions of this research include the creation of a novel interface between the CTN[1] Nengo simulation environment and the OpenAI Gym API, an implementation of simulated annealing (a form of offline reinforcement learning) to capture and encode each control/decision system, and a world-first application of the Neural Engineering Framework to the OpenAI research environment.

The first section discusses the background necessary to understand and appreciate the Neural Engineering Framework, as well as an overview of the OpenAI framework that we used to test the neural models. Next, we present the research questions that defined the test methodology, followed by an overview of the test environment, approach to measurement, and the results of each. Finally, we present our observations of both the successes and failures of this approach, lessons learned through the coupling of two very different machine learning frameworks, and future research that may add value to this domain.

Background

The research presented in this paper is an exploration of the possibilities and challenges associated with coupling biologically-inspired neural architectures with real-time simulation environments.

We realized that even though this approach was somewhat new for the Nengo community, development of an experimental approach and proper structuring of evaluation should keep these potential challenges and limitations in mind for future evaluation. After all, we wanted to know how future agents built upon a similar foundation as our minds may or may not perform as well as their optimized counterparts (Jordan, Weidel, & Morrison, 2017), and if we are truly fortunate, understand why. The following two sections provides a cursory review of the Nengo simulation environment, as well as an introduction to the OpenAI test framework used by the Nengo simulator.

Neural Engineering Framework

The Neural Engineering Framework (NEF) originally proposed by Eliasmith and Anderson (2003) provides a mechanism for transforming high level functions (written in Python) into biologically-plausible spiking networks, based on simulated Leaky Integrate-and-Fire (LIF) neurons. The NEF provides a close approximation of the functions encoded in the neural connections between neural clusters. This can be thought of as analogous to how a compiler translates human-readable code into machine code. This approach is useful to implement biologically plausible functions in an intelligent agent. However, it does not provide an explanation of how these neural functions were generated in the first place.

Modeling biologically-plausible neural networks creates a number of challenges, including heterogeneity, reliability, non-linearity, and scaling (Eliasmith, 2007). First, the

[1] Centre for Theoretical Neuroscience, University of Waterloo, Canada

neurons in our brains have different response properties and many neurotransmitters, therefore we cannot use homogeneous neurons in our models; this may be contrasted with networks using identical activation functions (such as ReLU or Sigmoid functions) as one might find in a system such as a Convolution Neural Network (CNN). Second, our brains have a variety of synaptic responses, different firing rates, and different propagation lengths. Third, spiking neurons have different spikes (over time/voltage), with a Gaussian distribution of synaptic firing based on Brownian Motion[2]; thus, functions have non-linear properties. Last, our brains have interconnected systems, with millions of neurons in ensemble networks that are hard to model without reducing entire clusters to a single function.

The NEF deals with these challenges through three fundamental design principles of representation, computation/transformation, and dynamical assumptions (Eliasmith, 2003). The first principle of *Representation* deals with how information is represented in a neural network. For example, neural spike trains are nonlinear encodings of vector spaces that can be linearly decoded. The second principle of *Transformation* allows for the possible alternate linear decodings of those encodings such that they can compute arbitrary vector functions. This is the same as first principle, except instead of getting a specific function back (like a velocity estimate), we can ask for a different computation (like the square of the value provided). The third principle of *Dynamics* combines elements of the first two, and assumes the neural representations (from 1) are control theoretic state variables in a nonlinear dynamical system (from 2); this is how the NEF represents time-varying phenomenon.

OpenAI

The OpenAI research company[3] has developed the open-source Gym (Brockman et al., 2016) toolkit for community-driven research in Reinforcement Learning, and provides a standard interface for researchers to measure an agent's ability to learn how to navigate a variety of environments. Gym environments[4] range from the extremely simple to highly complex with graduating degrees of difficulty, and supports both local and remote training and testing configurations. Environments are also divided into a number of standard classes depending on problem type, such as *Algorithmic* (text processing problems), *Classic Control* (of agents in a one-dimension plane), and *Box2D* (control of agents in a two-dimension plane), and every environment provides a standard interface for observation and action by the agent. The Gym API supports Python 2.7 or 3.5, and allows researchers to record and upload results to compare agent performance.

Many virtual environments designed for testing intelligent agents make certain assumptions around how the agent will interface with that environment. The OpenAI framework, in comparison, makes no assumptions about the agent interface, and instead dictates a standard API by which agents can interrogate, observe, and act on the environment; the actual definition of the environment is left to each designer, and does not assume the agent will read sprites from a region of the screen.

Research Questions

The types of environments and tasks that we selected were constrained by a set of relevant questions that may be answered using the tools at hand. This exploration can be factored into the following questions:

1. *How do we represent arbitrary environmental signals from multiple senses?*
2. *How do we choose between multiple and equally-viable actions in a given scenario?*
3. *Can we create a generic architecture that can learn and operate in a variety of situations?*

The first question regarding arbitrary environment signals is an important one; as real agents explore a new and sometimes unique environments, sensory data is received with no prior information on how it should be represented.

Our second question regarding action selection presents a unique challenge; as we will see, using spiking-neurons, and more specifically the Leaky Integrate and Fire (LIF) variety coupled with signal propagation delays as evidenced in the human brain (Abbot, 1999), forces some measure of consideration when training and testing such an agent.

The last question on generic architectures, is a reflection of our desire to take a model that may work in a single environment or test suite, and judge a model's generalizability to previously unknown domains. We explored these questions through a coupling of Nengo neural ensembles capable of observation and action, with a variety of games in Gym, to see if we could find a reliable way of generating agents capable of successfully playing different types of games.

Experimental Design and Development

This section discusses the practical application and integration between the Nengo framework and OpenAI. The novelty of the application suggested an exploratory approach to model design. In this section we review the lab environment used for development, how a simulation can interface with

[2] The seemingly random drift of individual molecules over time.
[3] https://openai.com/about/
[4] https://gym.openai.com/envs

Nengo, followed by ensemble learning, and finally an evaluation of the selected environments.

Lab Environment

Our test environment was developed in a Xubuntu 17.04 virtual machine, running on a Windows 10 host with an Intel Core i7 processor and 16GB of memory. This setup was chosen since OpenAI is not officially supported in Windows, and many of the online environments require different Python packages best handled in a separate docker[5] images in Linux. We used Anaconda for Python package management (v 3.5.3), and installed Nengo through the Python Package Index and pip installer.

The Nengo Simulator

The simulation environment can be run from command line or run locally inside of a browser-based GUI to provide direct and dynamic access to various node/ensemble probes and visualization tools for testing and debugging. This approach worked well, and only encountered performance issues when dealing with more complicated environments.

Our test files relied on imported libraries from NumPy[6] to perform matrix multiplication between state-action maps, and imported the supplied Gym wrappers so the entire Gym environment passed to the Simulator could run inside a Node object; this design permitted direct access for further signal analysis and ensemble tuning without connecting extra Probe objects.

The Gym Environment

The Gym environment is supported in Python[7] 2 and 3, works in both Linux and Windows (the first author has successfully used Gym on both platforms), and provides a standardized interface for each environment. A simple environment is instantiated in a python script (as shown[8] in Figure 1), and runs on the local machine of the Gym host.

```
import gym
env = gym.make('CartPole-v0')
env.reset()
for _ in range(1000):
    env.render()
    env.step(env.action_space.sample())
```

Figure 1: Example Gym Script

The Gym framework uses a standard agent-environment loop that steps through a new frame whenever the environment's *step* function is called, and returns a vector of four values:

Observation *(object):* the state of the environment, represented as an array of double values. These values can represent any number of features, from the position or angle of an object, to a pixel on the screen. This representation is left up to the environment creator.

Reward (float): the reinforcement value used for to learn in order to maximize the utility of each action. This value can take different ranges for the completion of each environment, as well as signal major events, such as entering a failed state, or achieving a checkpoint required for later success.

Done (Boolean): returns true if the environment has finished one round, otherwise always false.

Info (dictionary): a key-value collection that provides additional information about the state of the game. The OpenAI Gym standards do not allow agents to use this information in order to gain an advantage; rather it can be used by the researcher for development and debugging.

Problem Classes

The Gym platform divides the environments into problems subtypes, depending on a number of factors such as the complexity of the representation, the possible feature-action space, and the increasing degree of overall difficulty to put the agent into a "solved" state. Example problem classes include *Search & Optimization* for text-based board games, *Classic Control* using a joystick or control pad, and *Box2D* environments for more complex physical representations.

Action and Observation Spaces

The Gym API also defines the concept of a *space*[9], that allows the calling agent to briefly interrogate the allowable actions for that environment, as well as the expected range for each feature in an observation. For example, the *Lunar Lander* environment will report a total of four allowable actions for each thruster, and the expected number range for features that describe the position, angle, and velocity of various dimensions. Feature spaces can be a standard unit vector represented as [*-1, +1]*, or an infinite boundary represented as [*-inf, +inf*].

The Gym test cases included problems from the Classic Control and Box2D categories as previously described. The Classical Control environments were Cartpole-v010, and MountainCar-v011. The Box2D environment was LunarLander-v212.

[5] Docker images are application images that include all required dependencies independent of the host Operating System, and tend to run faster than a separate virtual machine.
[6] http://www.numpy.org/
[7] At the time of this writing, Python 2.7 and 3.6 was used.
[8] From https://gym.openai.com/docs/
[9] https://github.com/openai/gym/blob/master/gym/core.py
[10] https://gym.openai.com/envs/CartPole-v0
[11] https://gym.openai.com/envs/MountainCar-v0
[12] https://gym.openai.com/envs/LunarLander-v2

Learning Algorithm

Our first approach to function encoding was based on expert knowledge for each domain; for each test case, we provided a general heuristic to deal with typical PID[13] control problems such as angle, velocity, momentum, and thrust. This approach worked okay for the Classical Control domain, however, was not a scalable solution for high-dimensional vector spaces such as 2D Lunar Lander environment. We tried to overcome this limitation using a PID controller supplied by the OpenAI codebase, however the given mappings did not translate well to signals based on spiking neurons.

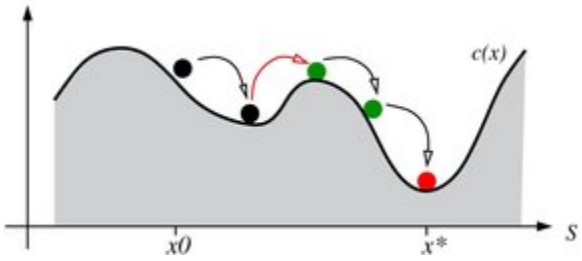

Figure 2. Simulated Annealing[14]

Given the reward value provided by each environment observation, we resorted to using a simulated annealing algorithm to try to find closer approximation to state-action mapping by both converging on the local minimum, as well as a random reset to look for the global minimum. For example (Figure 2), although the local optimal (left) has been found (as with Hill Climbing), a random reset (middle) may find an even better optimum (right). We ran each environment on a maximum of 10,000 epochs, settling the noise every 20 steps, and resetting the base parameters every 200 steps. This approach to learning within the standard Nengo simulation environment run inside the browser GUI would have been time prohibitive, so we ran it in an offline (CLI) simulation model without graphic rendering to find the desired weights.

Cartpole Game Simulation

The cartpole game involves moving a cart back and forth to prevent a pole from falling over. It is represented by a pole on a cart (Fig. 3) and it accepts two inputs (left, right). The game ends when the pole is greater than 15 degrees from the vertical. This game is incredibly difficult when played with manual controls since the player must always choose a direction (no input is not an option).

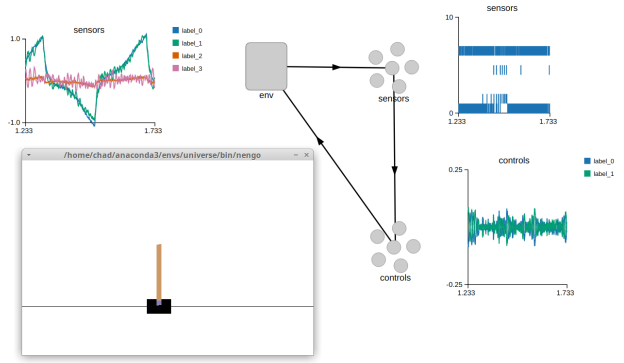

Figure 3. Nengo playing Cartpole

The Nengo model used two ensembles to represent the stimulus and action controls. The stimulus ensemble used 500 neurons, and four dimensions to represent the environment; notice the spiking pattern (top right) as represented by the sensor ensemble output. The controls ensemble used 500 Leaky Integrate-and-Fire (LIF) neurons and two dimensions to represent the action space, with a direct neuron interface to the control signal.

Mountain Car Game Simulation

The Mountain Car problem is represented by an underpowered car trying to get out of a valley. The environment accepts three inputs (left, right, brake), and provides a reward function based on the distance from the goal. This game is fairly easy when played with manual controls, however, is different from the previous Cartpole environment; it tests an agent's ability to learn that driving away from the goal is advantageous for future success (delayed gratification) and building the necessary momentum to escape before "running out of gas".

The Nengo model used two ensembles to represent the stimulus and action controls. Note the sensors here (Fig. 4, top right) can be represented as a two-dimensional plane. The stimulus ensemble used 500 LIF neurons, with two dimensions representing the momentum and distance from the goal.

[13] A *proportional-integral-derivative* control motor control loop provides self-correction by calculating the delta between current and desired angular velocity, and compensates accordingly.

[14] Example from http://www.mathematik.tu-clausthal.de

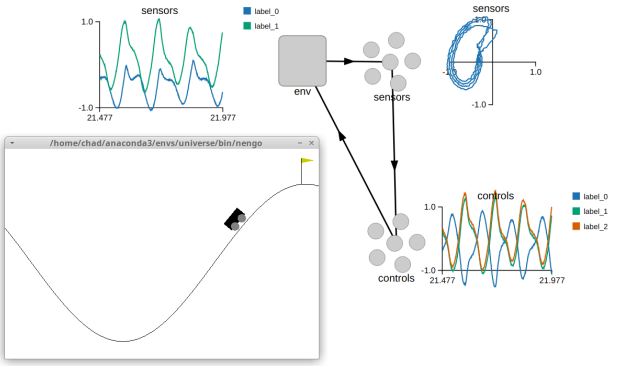

Figure 4. Nengo playing MountainCar

The controls ensemble used 500 LIF neurons, with three dimensions to represent the action space; output signals corresponded to left/right motor control, and braking to slow down.

Lunar Lander Game Simulation

The Lunar Lander game involves a space vessel falling towards a planetary surface, and requires the use of thruster engines to safely land within a designated area. The environment accepts four inputs (being left thruster, right thruster, bottom booster, or no action), which represents an observation through eight dimensions (vertical and horizontal position, angular velocity and trajectory, and touchdown) and provides a reward function based on how much fuel was consumed in the process. This game is somewhat challenging when played with manual controls, and provides a scale of difficulty since the starting situation is randomly generated, including the terrain profile, and lander trajectory. This environment (Fig. 5) allowed us to test the agent's ability to land from a spectrum of spawn points; we also found this aspect exposed a flaw in our training methodology as discussed later in this paper.

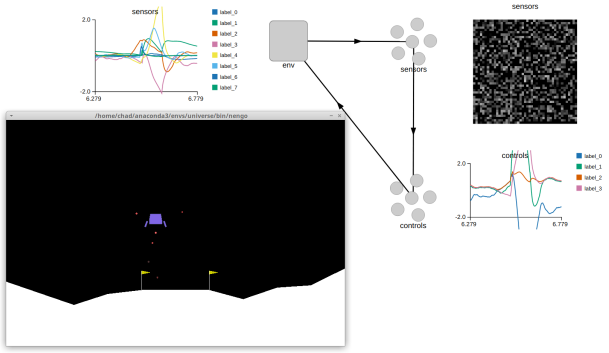

Figure 5. Nengo playing Lunar Lander

[15] This type adds signals to track the voltage, refractory time, and adaptation term for each neuron.

The stimulus ensemble used 2000 LIF neurons, with eight dimensions representing various aspects of position, angle, momentum, and distance from the point of origin. This ensemble was also modified to use the LIFRate[15] neuron type, an increased unit vector radius from one to two, and to reduce the default synapse delay from 50 milliseconds to zero. The controls ensemble used 500 LIF neurons, with four dimensions to represent the action space. The control signal used a direct neuron interface to the environment, and was modified to reduce the default synapse delay of 50 milliseconds to zero.

Observations and Results

Here we restate the original research questions, and attempt to answer them using the results of the experiments described, and observations around issues that could have been avoided.

Question 1: How do we represent arbitrary environmental signals from multiple senses?

Most of the OpenAI environments considered for this experiment provide a definition for each vector that is bounded by some upper and lower limit, however, this is not always the case. Some elements returned an INF (infinite) value, that makes normalization extremely difficult. In these cases, we had to run a series of trials with pseudorandom seeds to capture a sample distribution of each environment, and use each dimension's distribution to normalize the possible feature space into the standard unit vector. This approach may be considered similar to how we are able to acclimate and sensitize to a range of visual or haptic sensory input.

Assuming a model of the environment can be learned and evaluated within a reasonable amount of time, Nengo can instantiate and represent the observation and action space supplied by OpenAI at runtime; however, this approach requires bootstrapping. Learning a novel environment, on the other hand, would require an online approach to sampling the limits of each dimension such that the tuning curves used to approximate the learned function relies on the environment to be acted upon.

The question of representation can be answered by a proper sampling and understanding of the observation space, creation of an ensemble that subsumes (or perhaps grows) to handle the expected dimensionality, and regular online learning to handle changing conditions.

Question 2: How do we choose between multiple and equally-viable actions in a given scenario?

The Lunar Lander configuration required several modifications away from the default synaptic delay settings in order to deal with issues of signal strength, and timing, or more precisely, the lack of delay presented during offline training. First, a default ensemble uses a standard unit vector to represent the non-linear function as a linearly activation. This approach works as long as the observation vector representation is sampled and normalized ahead of learning a conditioned response, and provides plausible support for our own pre-processing of sensor inputs. Second, the problems associated with time delay phenomenon can be caused from unrealistically instantaneous feedback provided during the offline learning process. The combination of integration and spike propagation delay during online simulation (being 50ms for humans) was enough to severely impact agent performance in higher order representations. We discovered that this can be mitigated by artificially introducing the same delay into offline simulation trials so the ensemble tuning curves account for the time delay. This approach to mapping the time-delay between offline learning and online simulation introduces an interesting assumption; the physical limitations imposed by synaptic delay is inherently overcome through learning to operate within an internal model that is, on average, 50ms behind reality!

The question of choice selection between multiple equally-viable actions based on unknown sensory phenomenon, then, can be confirmed through a proper representation of the normalized signal strength, normalized signal range, and accounting for time delay propagation during both training and testing phases of an ensemble that uses spiking neural networks.

Question 3: Can we create a generic architecture that can learn and operate in a variety of situations?

The NEF, as earlier described, assumes a human-level understanding of the problem domain prior to approximating the desired function. This can be achieved through either human-programmed heuristics, or through human-in-the-loop reinforcement learning; something the Nengo library also supports. Our first attempts at providing a transformation function did not scale, and we had to resort to machine learning to brute force a viable function. Our representation of a state-action space, albeit simplistic, worked well for small environments limited to 32 possible weights, but what about larger models?

Nengo provides a number of interesting learning methods, however, most of them rely on developing a heuristic before online/offline learning of the transformation function. Answering our questions about generalizing a learned Nengo ensemble to multiple OpenAI environments would require both a mechanism to poll the environment for this action space ahead of time, and a method to differentiate between goals, achievements, and rewards. Approaching an arbitrary (and costly) environment would likely require a construct to represent higher-order reasoning, and is beyond the scope of the current paper.

Conclusions

Our original research questions were positioned to deal with the common issues surrounding representation, goal-directed action selection, and the creation of a learned model without over- or under-fitting the observation, such that the model can properly generalize to new yet similar problem domains.

The Nengo suite is extremely powerful in customizability and affording the user the ability to fine-tune almost every aspect of large-scale models to fit many (sometimes opposing) neuro-theoretic assumptions.

Our results demonstrate that even biologically-plausible models must make some assumptions about the operating environment in order to normalize incoming signals and produce an ensemble tuning curve capable of optimal action selection. Smaller models are capable of replicating the requirements for signal processing and action selection in multi-dimensional environments, and point towards the necessity for higher-order representation to deal with the challenges of complexity imposed by operating in richer environments, as humans are apt to do.

References

Abbott, L. F. 1999. Lapicque's introduction of the integrate-and-fire model neuron (1907). *Brain Research Bulletin*, 50(5–6), 303–304.

Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., & Zaremba, W. 2016. OpenAI Gym. *arXiv*:1606.01540

Eliasmith, C. 2007. How to build a brain: From function to implementation. *Synthese*, 159(3), 373–388.

Eliasmith, C., & Anderson, C. H. 2003. Neural Engineering: Computation Representation and Dyamics in Neurobiological Systems. Cambridge, Mass. *MIT Press*.

Jordan, J., Weidel, P., & Morrison, A. 2017. Closing the loop between neural network simulators and the OpenAI Gym. a*rXiv*:1709.05650.

Stewart, T. C., Choo, X., & Eliasmith, C. 2010. Dynamic behaviour of a spiking model of action selection in the basal ganglia. *Proceedings of the 10th International Conference on Cognitive Modeling*, 235–240.

Stewart, T. C., & Eliasmith, C. 2014. Large-scale synthesis of functional spiking neural circuits. *Proceedings of the IEEE*.