Check for updates

# Nengo and Low-Power AI Hardware for Robust, Embedded Neurorobotics

*Travis DeWolf [1]\*, Pawel Jaworski [1] and Chris Eliasmith [1,2]*

[1] *Applied Brain Research, Waterloo, ON, Canada,* [2] *Centre for Theoretical Neuroscience, University of Waterloo, Waterloo, ON, Canada*

In this paper we demonstrate how the Nengo neural modeling and simulation libraries enable users to quickly develop robotic perception and action neural networks for simulation on neuromorphic hardware using tools they are already familiar with, such as Keras and Python. We identify four primary challenges in building robust, embedded neurorobotic systems, including: (1) developing infrastructure for interfacing with the environment and sensors; (2) processing task specific sensory signals; (3) generating robust, explainable control signals; and (4) compiling neural networks to run on target hardware. Nengo helps to address these challenges by: (1) providing the NengoInterfaces library, which defines a simple but powerful API for users to interact with simulations and hardware; (2) providing the NengoDL library, which lets users use the Keras and TensorFlow API to develop Nengo models; (3) implementing the Neural Engineering Framework, which provides white-box methods for implementing known functions and circuits; and (4) providing multiple backend libraries, such as NengoLoihi, that enable users to compile the same model to different hardware. We present two examples using Nengo to develop neural networks that run on CPUs and GPUs as well as Intel's neuromorphic chip, Loihi, to demonstrate two variations on this workflow. The first example is an implementation of an end-to-end spiking neural network in Nengo that controls a rover simulated in Mujoco. The network integrates a deep convolutional network that processes visual input from cameras mounted on the rover to track a target, and a control system implementing steering and drive functions in connection weights to guide the rover to the target. The second example uses Nengo as a smaller component in a system that has addressed some but not all of those challenges. Specifically it is used to augment a force-based operational space controller with neural adaptive control to improve performance during a reaching task using a real-world Kinova Jaco[2] robotic arm. The code and implementation details are provided[1], with the intent of enabling other researchers to build and run their own neurorobotic systems.

**Keywords: Nengo, neuromorphic, neurorobotic, spiking neural networks, robotic control, adaptive control, embedded robotics**

## 1. INTRODUCTION

Specialized AI hardware offers exciting potential for the development of low-power, highly responsive robotic systems with embedded control. Edge devices for accelerating neural networks are starting to become commercially available from companies, such as NVIDIA, BrainChip, GrAI Matter Labs, Google, Intel, and IBM. While the promise of low-power and low-latency embedded

---

[1] https://github.com/abr/neurorobotics-2020

processing and control is highly desirable, the process of implementing algorithms on the hardware generally remains a significant hurdle. Developing neural networks for processing sensory data and generating control signals is a difficult problem, and adding further constraints specific to a particular piece of hardware only increases the challenge. In this paper we focus on the development of spiking neural networks (SNNs) for the subset of devices known as neuromorphic hardware (Mead, 1990). Effectively using such hardware often requires additional expert knowledge outside of traditional machine learning and neural network methods to program effectively. In short, it is difficult to quickly and easily build robust, integrated neural models for controlling robots using neuromorphic hardware.

Building neurorobotic systems can be characterized as consisting of four tasks:

1. Developing infrastructure to send and receive signals from the environment. There are a multitude of different interface protocols for sensors, hardware, and simulators. To minimize development time, simple interfaces should be available and interchangeable with minimal changes to the model description.

2. Processing task specific sensory signals. Deep neural networks (DNNs) are the principle machine learning tool used for sensory processing, and it is important to take advantage of the extensive literature and solutions in this field. To that end, users need to be able to take DNNs, convert them to networks that can run on neuromorphic hardware, and integrate them into a neurorobotics control system. For systems using perception methods not rooted in neural networks, it is also important to be able to easily integrate their output with downstream networks.

3. Generating robust control signals with explainable neural networks. When generating control signals, having guarantees on performance is important, and often necessary. To accomplish this users needs to know exactly what operations are being implemented to guarantee stability. The Neural Engineering Framework (NEF; Eliasmith and Anderson, 2003) offers "white-box" neural network development methods that allow integration of these methods into neurorobotics control systems, making an API for building up such networks quickly desirable.

4. Compiling neural networks to run on multiple targeted hardware platforms. During the process of designing control and perception systems it is often desirable to develop neural network models on standard hardware with minimal compilation overhead. Once a prototype network is working, it should be straightforward to compile to targeted special purpose hardware. Being able to compile the same model to different hardware can greatly speed up the development of neurorobotics systems.

In this paper we present a neurorobotics development workflow for building neural networks that run on standard and neuromorphic hardware using the Nengo neural modeling platform (http://nengo.ai/; Bekolay et al., 2014). As part of this workflow, we take advantage of the NengoInterface package to streamline interfacing with the physics simulators, the NengoDL package for integrating Keras and TensorFlow models that process incoming sensory data, and the NengoLoihi package for compiling the model to run on Intel's Loihi neuromorphic chip (Davies et al., 2018).

We illustrate two variations on this workflow by describing two example neurorobotics applications in detail. The first example implements an end-to-end perception and action system in Nengo for tracking a target with a rover simulated in Mujoco (Todorov et al., 2012). The rover has four mounted cameras whose input is fed into a DNN built using Keras. The DNN estimates the distance to the target, and this estimate is sent to a control network which generates torques to apply to the steering wheel and drive wheels to move the rover to the target. This full system is then compiled onto Loihi. In the second example, we demonstrate how Nengo can be integrated with an existing system by augmenting a standard robotic arm force controller using a neural adaptive controller that learns online. We implement the adaptive component both on standard hardware and Loihi, where we take advantage of its on-chip learning. We compare implementations of the adaptive control system as it drives a physical Jaco[2] robot arm from Kinova to perform a reaching task while adapting to the unmodeled force of holding a two pound weight. We discuss the workflow bottlenecks and challenges that are encountered, addressed, and remaining.

## 2. BACKGROUND

### 2.1. Nengo and Supporting Development Packages

Nengo is a neural modeling development and simulation platform. Users specify the architecture of models using a Python-based API, referred to as the "front-end," and then compile their model for simulation on hardware using a "back-end." The API is designed such that the same model can be run on different hardware with few to no changes in the front-end script. Supported hardware includes CPUs, GPUs, FPGAs, and specialized neuromorphic hardware (such as Intel's Loihi chip).

Nengo users are able to quickly design and simulate neural networks, and use the NengoGUI package to visualize and interface with them during run-time. The NengoDL package extends Nengo's API to interface with and integrate deep and machine learning networks built in Keras or TensorFlow, as well as take advantage of TensorFlow's resource distribution manager for efficient simulation across multiple processors. The NengoInterfaces package provides easy interface access with the Mujoco simulator, abstracting out the setup and overhead involved in connecting, running, communicating, and restarting Mujoco simulations. The NengoLoihi package allows us to compile our models to run on the Loihi, and also handles communication to and from the chip. Additionally, the NengoLoihi package provides a Loihi emulator that allows users to run their models while simulating Loihi dynamics and computations on their computers, which aids efficient development. Nengo has more supporting packages, but in the interest of space we limit our review

to the above packages relevant to the work presented in this paper.

## 2.2. The Loihi Chip

We use Intel's Loihi chip (Davies et al., 2018) for demonstration in the examples below. The Loihi chip is a many-core mesh with 128 neuromorphic cores, 3 embedded × 86 processing cores, and off-chip communication interfaces. An asynchronous network-on-chip communicates packetized messages between cores, allowing write, read request and response, spike messages, and barrier messages for time synchronization to be sent between cores. Each neuromorphic core has 1,024 neural "compartments," where a compartment can be allocated to simulate a neuron or dendrite.

The NengoLoihi package allows users to compile their front-end script to the Loihi chip, handling all of the low-level mapping and communication. Some front-end scripts will require modification specifying low-level details, such as how to allocate neural populations across Loihi cores, but largely the details of this low-level mapping are abstracted out and handled automatically.

## 2.3. Other Neurorobotic Workflows and Toolkits

Most commonly, building neurorobotic applications involves hand-crafting and tuning SNNs for the task of interest. In Gutierrez-Galan et al. (2020) the authors build an SNN inspired by biology to implement a central-pattern generator that runs on the SpiNNaker neuromorphic board (Furber et al., 2014) and drives a hexapod robot to walk, trot, or run. In Kreiser et al. (2019), the authors hand craft an SNN to run on the Loihi to steer a small rover. In Stagsted et al. (2020), a PID controller is implemented in an SNN running on Loihi to steer an unmanned aerial vehicle. The authors accomplish this using one-hot encoding, such that only one neuron in a population is able to spike at a time, and each neuron represents a different possible variable value, to build up networks implementing addition and subtraction, at which point a PID controller can be built. Implementation of non-linear functions is listed as future work. We note that these models can be built using the Nengo API, and the NEF API makes non-linear function implementation straight-forward.

The authors of Taunyazov et al. (2020) implement an SNN visual tactile system that runs on Loihi and performs container classification and rotational slip detection. The network is a deep net trained with the SLAYER (Bam Shrestha and Orchard, 2018) method, which uses stochastic spiking neurons to overcome the undefined derivative in spiking neurons that prevents backpropagation from working. In Hwu et al. (2017), the authors train an Energy-Efficient Deep Neural Network (EEDN), a deep network designed specifically to run on IBM's TrueNorth neuromorphic chip (Sawada et al., 2016), on trail photos to train up a network that attaches to a real-world rover and guides it along a path. The authors mention that the trained weights work well in a standard convolutional neural network or in the EEDN, which can transfer its weights directly to the TrueNorth. As we detail in our examples below, the NengoDL package allows users

to take advantage of similar deep learning methods for training SNNs to run on neuromorphic hardware.

Another way to program neuromorphic hardware is using the Python Neural Networks (PyNN) interface (Davison et al., 2009). PyNN is a front-end API that shares Nengo's goal of creating a high-level front-end API that specifies neural network architecture without being tied to the low-level implementations specifics. PyNN was developed to standardize scripting neural networks across several different low-level neural simulators, including Brian (Goodman and Brette, 2008), NEURON (Hines and Carnevale, 1997), and Nest (Gewaltig and Diesmann, 2007). Since its development, others have extended PyNN to include backends for other simulators and neuromorphic hardware. While Nengo allows the same low-level specificity of PyNN, Nengo also allows many of these details to be easily abstracted, and has more focus on high-level objects that speed the development of large or complex neural systems. For example, to implement a communication channel between two populations of neurons takes ~12 lines in Nengo, and over 80 lines in PyNN (Bekolay, 2011). Importantly, Nengo also supports methods for generating connection weight matrices, including the NEF as well as Keras/TensorFlow techniques, which PyNN does not.

The Neurorobotics Platform (NRP; Falotico et al., 2017) is a web-based simulation environment for running SNNs hooked up to virtual robots. Developed as part of the Human Brain Project (Markram, 2012), the goal of the NRP is to streamline the process of running experiments with SNNs and robots. The NRP lets users quickly select a virtual environment, robot, and SNNs to run an experiment. The NRP has a broad scope, offering tools, such as the web-based robot designer, experimental workflow editor, and Gazebo simulation environment editor. In contrast, Nengo focuses primarily on the development, integration, and simulation of neural networks (both spiking and non-spiking), support of different neural network programming paradigms like the NEF and Keras/TensorFlow, compiling to different hardware backends, and systems interfacing through Python. There is potential for collaboration between Nengo and the NRP, expanding the neural network development and simulator interfacing of the NRP, and the experiment design and web interface of Nengo.

## 3. NEUROROBOTIC ROVER SYSTEM

In this example we develop an end-to-end perception and action system for tracking a target with a rover in Mujoco (Todorov et al., 2012). The simulated rover we use is a four wheeled vehicle, built using Mujoco's XML modeling language, with Ackerman steering and rear differential drive in a boundless environment with no obstacles. The rover has 4 RGB cameras mounted on its back, each with a 90° field-of-vision, that provide a full 360° view of the environment, and a sensor on the front wheels that provide steering angle information. The rover accepts two torque input signals, one to control the acceleration of the rear wheels, and one to turn the front wheels right or left. The target is a red sphere that floats in the air and warps to a new location (generated from
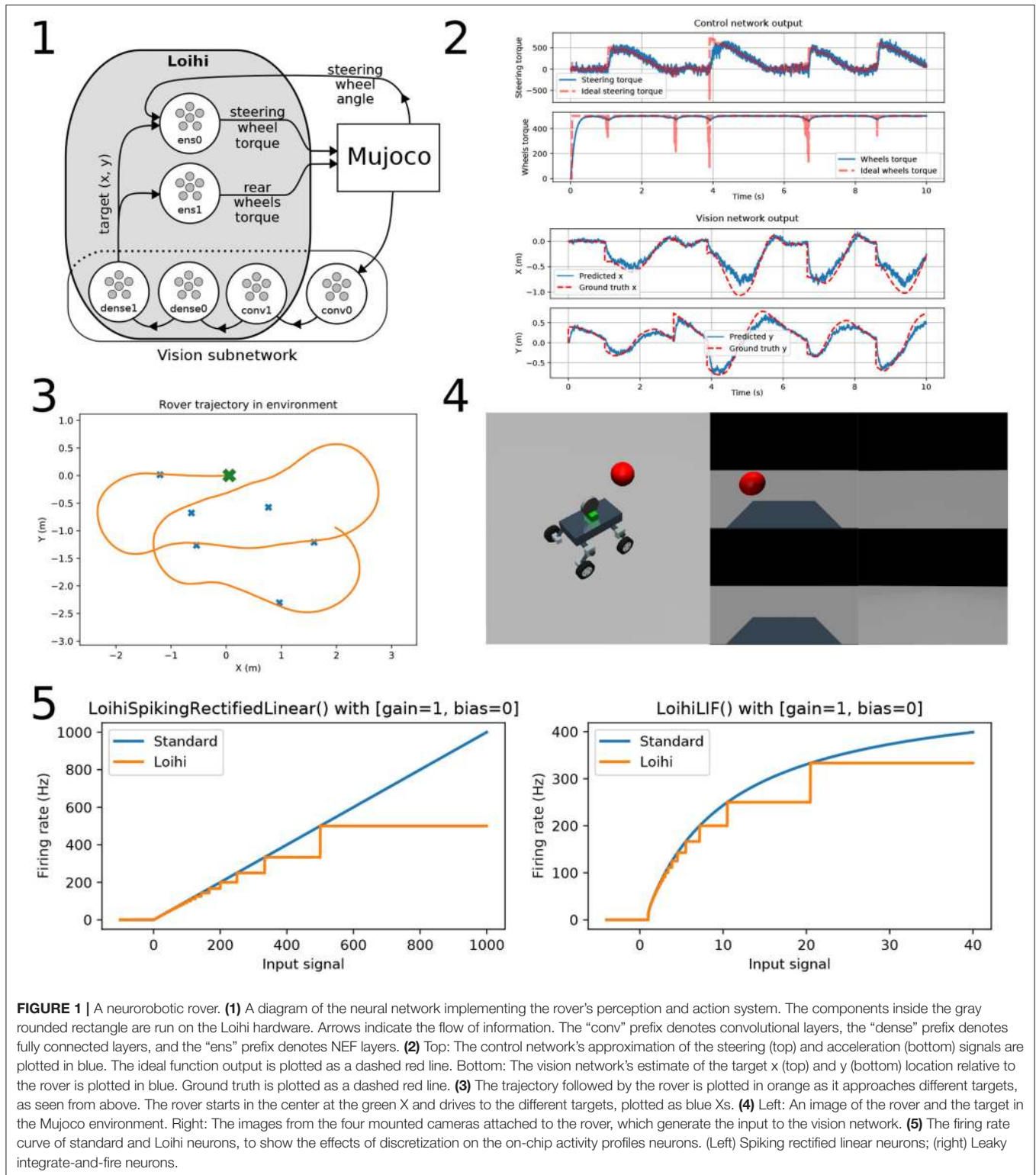
**FIGURE 1** | A neurorobotic rover. **(1)** A diagram of the neural network implementing the rover's perception and action system. The components inside the gray rounded rectangle are run on the Loihi hardware. Arrows indicate the flow of information. The "conv" prefix denotes convolutional layers, the "dense" prefix denotes fully connected layers, and the "ens" prefix denotes NEF layers. **(2)** Top: The control network's approximation of the steering (top) and acceleration (bottom) signals are plotted in blue. The ideal function output is plotted as a dashed red line. Bottom: The vision network's estimate of the target x (top) and y (bottom) location relative to the rover is plotted in blue. Ground truth is plotted as a dashed red line. **(3)** The trajectory followed by the rover is plotted in orange as it approaches different targets, as seen from above. The rover starts in the center at the green X and drives to the different targets, plotted as blue Xs. **(4)** Left: An image of the rover and the target in the Mujoco environment. Right: The images from the four mounted cameras attached to the rover, which generate the input to the vision network. **(5)** The firing rate curve of standard and Loihi neurons, to show the effects of discretization on the on-chip activity profiles neurons. (Left) Spiking rectified linear neurons; (right) Leaky integrate-and-fire neurons.

a random distribution within 3 m of the origin) when the center-of-mass of the rover is within 50 cm of the center of the target. **Figure 1**–4 shows the rover in the world on the left, and the view from each of the 4 cameras on the right.

To begin developing our perception and action systems, we first build out the controller without using neural networks. We use the exact ground-truth information provided by Mujoco to identify the target location relative to the rover, and calculate

the torques for acceleration and steering in Python. Next, we address sensory perception by building a DNN that can identify target $(x, y)$ location relative to the rover based on input from the four mounted cameras. This requires building a dataset using the simulator to train the DNN. We initially train the system using standard artificial rate neurons, to confirm that the desired level of performance can be achieved with our network architecture. We then us NengoDL to convert the network to spiking neurons and tune the parameters to optimize performance. Next, we replace each function in the control system with spiking analogs, testing each in isolation before finally integrating the entire spiking network. Once performance is achieved in a fully spiking neural network, we move the network simulation from Nengo into the NengoLoihi emulator, and tune the parameters again to optimize under the constraints of the Loihi. Finally we compile the network to the Loihi hardware, again using NengoLoihi. In the next sections we describe each of these steps in more detail.

## 3.1. Interfacing With Mujoco

Interfacing to Mujoco is done through NengoInterfaces, which uses the `mujoco-py` (Ray et al., 2020) library for Python bindings to the Mujoco C API. The interface accepts force signals from the neural network, applies them inside Mujoco and moves the simulation forward one time step, and then returns feedback from the rover. Environment information can be accessed directly from the NengoInterfaces API, and less common functions are available through the `mujoco-py` simulation and environment model parameters.

## 3.2. Processing Visual Input Using a Keras DNN Converted to a Nengo SNN

The network used for tracking the target location consists of two convolutional layers and three dense layers, and is shown in the bottom block of **Figure 1**–1. The first convolutional layer uses $1 \times 1$ kernels with a stride of 1, and a filter size of 3; its purpose is to convert the image signal into spikes to be sent to the layers running on Loihi[2]. To generate the input image, we take a $32 \times 32$ pixels resolution snapshot from each camera, and concatenate them horizontally to create a $32 \times 128$ pixels input to the network. This resolution was chosen as the smallest network size that could still identify targets at a distance of 3 m. To retrieve the signal from the last layer running on-chip we use neural probes, which monitor spiking activity.

The dataset used for training the model was generated by recording both input from the mounted cameras and the relative distance to the target. The data was collected while our non-spiking control system drove the rover to the targets, recording every 10th frame. The final dataset used consists of roughly 40,000 images and target (x, y) locations (the height of the target is constant and is not relevant to control so we ignore it).

Training the network with non-spiking ReLU activation functions using standard DNN tools (i.e., Keras) was the first step. This allows us to validate the network architecture. For all of our training we use the RMSprop optimizer from TensorFlow

and the mean squared error loss function on network output to learn to output the target $(x, y)$ locations associated with an image. When converting the network into spiking neurons take into account both the desired firing rates and the activation function of neurons running on the Loihi. We have found that if the average firing rates are $<$50 Hz, spiking neurons are not driven strongly enough to generate any activity. We target the 175 Hz range for firing rates, because it is large enough to ensure spiking, but still inside the range where the Loihi neurons approximate standard neurons well (discussed below). To achieve this, we initialize the weights of our network by setting the `scale_firing_rates` parameter of the NengoDL built-in Keras converter to `400`. This parameter encourages the optimizer to converge to firing rates that are higher or lower, based on the scaling. An alternative, and more fine-grained and reliable method, is to add a firing rate regularization term to the cost function that penalizes neurons firing outside of the desired range.

The second factor we need to account for is the activation function of neurons on the target neuromorphic hardware. Neurons on the Loihi have a unique activation profile because of discretization that occurs on-chip, as shown in **Figure 1**–5. We use NengoLoihi's model of the Loihi rectified linear neuron during training to ensure that the network is trained on the same kind of activation functions used during inference.

Finally, we set network synapses throughout the network as required to smooth out the signal and filter noise. In Nengo, you can set synapses to 0 to implement no filtering, or to None to collapse the computations of two connected layers into a single layer. We set each of the synapses on the connections between layers to None. This speeds up the propagation of spikes through the network, but also has the potential to decrease performance due increased noise in the signal. We found empirically, however, that applying a 0.05 s time constant low-pass filter on the vision network output smoothed the target location estimate and improved the control signal generated downstream.

## 3.3. Generating Robust, Explainable Control Signals Using the NEF

We described the NEF as a "white-box" approach to building neural networks because of its mechanistic approach. Briefly, the NEF uses populations of neurons to represent vectors, feed-forward connections between populations to implement functions on those vectors, and recurrent connections to implement differential equations. Rather than specifying a task-level cost function, as in standard machine learning methods, the user must first design a circuit that solves the problem, including specifying a state space representation, set of computations, and flow of information. The user then uses Nengo's NEF API to implement this circuit in neurons. These added top-down constraints give clear network structure that allows users to identify points of error and apply specific changes to debug and improve network performance. This is in contrast to "black-box" deep learning methods, which use training algorithms to find a network configuration that solves the problem, without knowledge of the function implemented. In situations, such

---

[2]It is also possible to send information to Loihi by setting the bias and current for neurons, but we have found this method to be slower for a dynamic input signal.

as complex visual analysis where the algorithmic solution is unknown this is very desirable, but in cases where proven solutions are available and we would like performance guarantees the methods of the NEF are preferable.

The motor system for the rover is implemented using the NEF. The first step in this implementation is deriving the functions for vehicle acceleration and steering wheel control. We calculate acceleration as distance to target multiplied by a gain term, clipped to a maximum magnitude empirically chosen to prevent the rover from flipping when traveling at top speed and turning sharply. Formally,

$$u_{acceleration} = k_a \min(\|(x^*, y^*)\|, 1) \tag{1}$$

where $u$ is the control signal sent to the rover, $k_a$ is the acceleration gain term, $x^*$ and $y^*$ are the target location relative to the rover, and $\| \cdot \|$ denotes the 2-norm.

The torque applied to the steering wheel is calculated with a simple proportional controller using the difference between the current and desired angle of the wheels multiplied by a gain term. The desired angle is calculated as the angle to the target using $\arctan2(-x^*, y^*)$, where order of the arguments and the negative sign in front of the $x$ term account for the orientation of the rover relative to the environment. Formally,

$$u_{steer} = k_p(\arctan2(-x^*, y^*) - q) \tag{2}$$

where $k_p$ is the proportional gain term, and $q$ is the current angle of the steering wheel.

The neural circuit implementation of this controller is done in two parallel ensembles, with the connections weights on the outbound connections calculated to approximate the above equations, as shown in the top portion of **Figure 1**–1. In Nengo, we project the relevant variables into each population and specify the functions to be computed on their outbound connections using Python code. Nengo then solves for connection weights using the principles of the NEF. In this particular case, we compute these functions using separate ensembles, rather than having a single ensemble with two separate outbound connections, because the functions they are calculating depend on different sets of variables. The acceleration function only requires the estimated target (x, y) values for calculation, while the steering function requires the estimated target (x, y) and the current angle of the steering wheel. While the variables required by the acceleration function are a subset of the variables used in the steering function, we can achieve greater precision by using an ensemble that only encodes the target (x, y).

In the hardware implementation, each ensemble consists of 4,096 Loihi leaky integrate-and-fire neurons, spread across four cores. This specific number of neurons is chosen to satisfy hardware constraints on the number of inbound connections a population of neurons can receive. The neurons in the ensemble are set up to have maximum firing rates between 175 and 220 Hz, chosen because in general higher firing rates provide for more accurate function approximation.

## 3.4. Integration and Compiling to Hardware

Putting the vision and control networks together is a simple matter of connecting the output of one to the input of the other in Nengo. As both networks were built using Loihi-type neurons, they are also prepared to be mapped to neuromorphic hardware. During the initial building and debugging process running on a CPU backend, which is the Nengo default, greatly expedited development. The NengoLoihi backend can then be used to compile the network to run on the Loihi (we could also use NengoOCL or NengoDL to compile to GPU and run directly in Nengo). We used a workstation with and Intel Core i7-6700K CPU @ 4.00 GHz × 8 with 32 GB RAM and GeForce GTX 1070/PCIe/SSE2 running Ubuntu 18.04, and the Intel Nahuku board with 32 Loihi chips (Davies et al., 2018) running NxSDK 0.9. In this example, we are only using one of the Loihi chips on the board. When the system is running, Nengo provides the interface between the simulator and the hardware, but all computations are run on the Loihi chip.

## 3.5. Performance

**Figure 1**–3 shows the (x, y) trajectory of the rover moving throughout the environment to six different targets in the environment, starting from the green "x." **Figure 1**–2 shows the perception and action signals from the network, with the target (x, y) estimated in the top figures in blue and the ground truth shown in red. The lower figure shows the steering and acceleration control signals generated by the network in blue with the ideal values in orange. As can be seen, the rover drives accurately (to within 50 cm) over the course of the trial. We have not performed extensive testing of the accuracy, and do not provide quantitative results as our purpose here is to focus on the methods used to develop the system. There is clearly significant room for improvement and extension to this work. Nevertheless, this simple example demonstrates the implementation of an end-to-end perception and action spiking neural network running on neuromorphic hardware. All code is available online at https://github.com/abr/neurorobotics-2020. We have provided full code to serve as a starting point for those interested in exploring neurorobotic solutions that can leverage embedded neuromorphic hardware for next generation systems.

## 4. NEUROROBOTIC ADAPTIVE ARM CONTROL

In this second example we augment an existing force controller with an adaptive neural network implemented on Loihi, using on-chip learning to control a Kinova Jaco[2] physical robot in a reaching task while it holds an unexpected weight. The existing control system generates joint torques using a standard proportional derivative (PD) operational space controller (OSC; Slotine et al., 1988), designed to move the hand along a target path. The adaptive controller adds an adaptive signal trained online to account for any unexpected forces affecting movement, which is tuned online. We compare performance of the adaptive control system to a non-adaptive PD OSC, and an industry standard proportional integrated-error derivative (PID) OSC.

We also compare the neuromorphic implementation with CPU and GPU implementations of the adaptive control system. We compare all systems in terms of accuracy, power use, and control loop update latency.

Operational space control relies on an accurate model of the arm dynamics to generate torques that will move the arm as desired. If the arm picks up an object, is subjected to external forces or perturbances, or wears down over time, the dynamics have changed and OSC performance will degrade unless the changes can be accurately modeled. In general, these perturbations can not be predicted in advance, so updating the controller on-the-fly is desirable. This is the purpose of the adaptive controller we use here, implemented as a neural network. Intuitively, the adaptive controller acts as a context sensitive integrated error term. Where standard integrated error terms (such as the I in PID control) apply the same learned error regardless of the current joint angles or velocities, the adaptive controller learns to account for errors specific to different arm states. The difference becomes significant in the control of highly non-linear systems, where the error that needs to be compensated for changes significantly with system state.

## 4.1. Interfacing With the Jaco$^2$ Robotic Arm

In this example Nengo is called as a sub-function of the PD OSC. The PD OSC itself is implemented in Python and runs on a workstation that interfaces with a physical Kinova Jaco$^2$ 6 degrees-of-freedom (DOF) arm. The interface implemented is through the ABR Jaco$^2$ repository, and includes no neural network infrastructure. To integrate neural computation with this standard Python code, at each time step in the control model, Nengo is called to run the neural network for a single step. The control code sends feedback from the arm to Nengo and receives back an adaptive control signal to add into the outbound set of joint torques sent to the Jaco$^2$. Nengo takes care of running the neural network on a CPU, GPU, or the Loihi neuromorphic hardware.

## 4.2. Processing Sensory Feedback Using the NEF

In this application we are augmenting an existing control system with an adaptive control signal generated by a neural ensemble. The ensemble requires sensory feedback related to joint positions and velocities as input in order to compute the necessary correction to the control signal. Because the sensory feedback is a relatively low-dimensional signal (e.g., compared to image input), it does not need to be processed by a deep neural network before we can use it to generate a corrective control signal. The adaptive controller requires that the effects of the unexpected force are predictable given the input provided to the ensemble to be able to learn to compensate for unexpected forces affecting the arm. If, for example, the force affecting the arm was a function of joint angle, and the ensemble only had inputs related to joint velocity, then the network would not be able to adapt to the force.

Given that we have appropriate inputs, we further need to make sure that the neurons are sufficiently sensitive to different states of the arm relevant for compensating for the unexpected force. In the NEF, the neural tuning properties are determined by a combination of the neuron encoders (or "preferred" direction vectors), gains, and biases. This tuning determines which parts of the input state space are represented by the neural ensemble. In this section, we present considerations that determine how to appropriately pick these tuning curves for adaptive control in a highly non-linear state space.
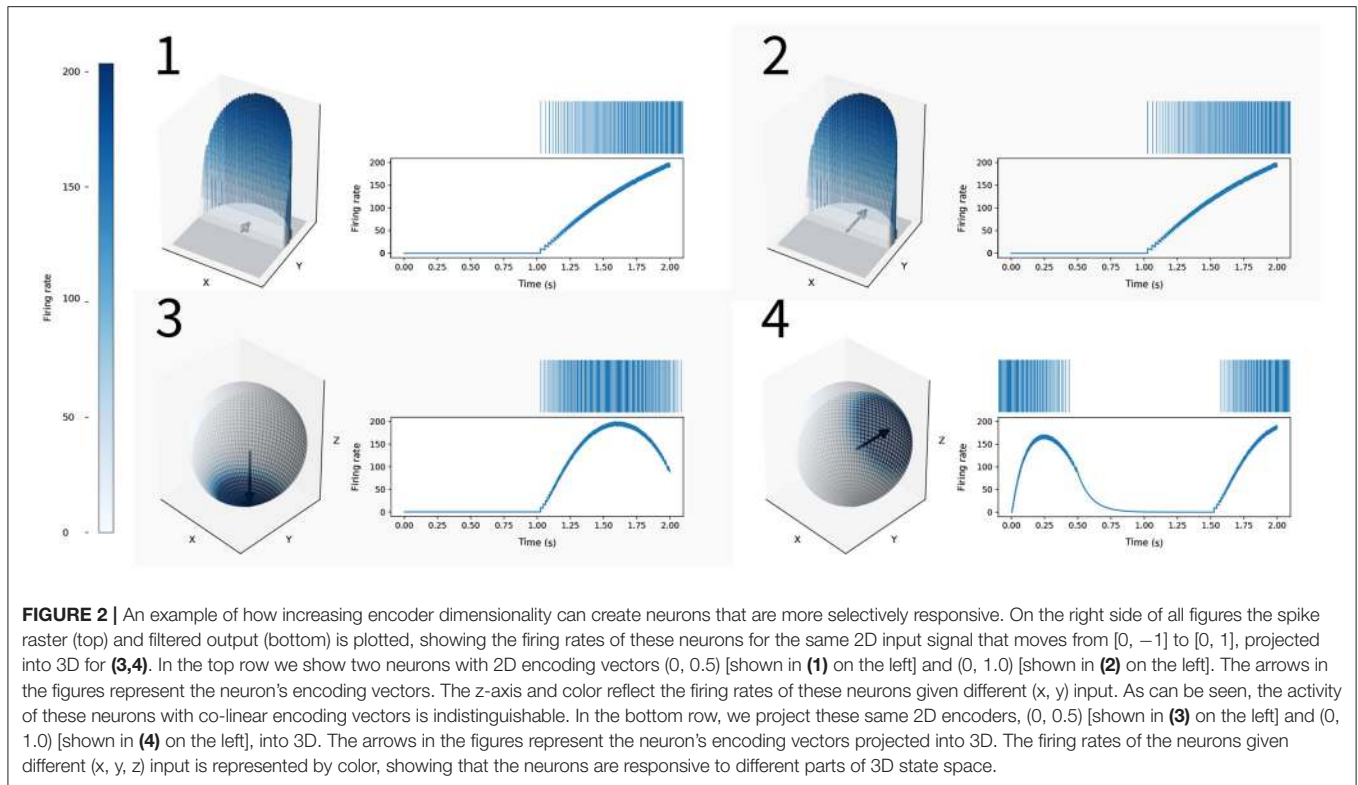
In particular, we need to ensure that neurons are not active over a large part of state space. If this is the case, the compensatory signal they learn in one part of state space may incorrectly generalize to other parts. In contrast, if neurons are active over a small part of state space, then the compensatory signal they learn in one area will not affect what learning occurs in other parts of state space. Unsurprisingly, there is a trade-off between the specificity of neural responses and their generalization abilities. In arm control, because the dynamics are highly non-linear, we generally want to ensure that neurons in our ensemble are sensitive to localized parts of state space. We also do not want to waste neural resources. Consequently, neurons should only be sensitive to parts of state space that are actually explored by the arm. In other words, we do not want our population to include neurons that never become active.

To handle both of these issues, we need to carefully choose neural tuning curves, and hence NEF encoders. To optimize neurons for the relevant parts of the state space, we begin by subtracting the mean and then normalizing the input signals for each dimension given their joint limits. Next, we project into the D + 1 unit hypersphere, where D is the number of dimensions represented by the ensemble. By doing this it becomes possible to carefully control the range of values that cause neurons in the ensemble to respond. Because we have 6 joints and two input signals from each (i.e., position and velocity), we project the normalized signals onto the 13-dimensional (i.e., 12 + 1) unit hypersphere and use that as input to the ensemble.

**Figure 2** illustrates how this allows us to control neural responses by considering the simpler case of projecting 2D into 3D. Assuming we have inputs in the −1 to 1 range along each dimension, the inputs are going to lie somewhere in the unit square. In the NEF, by default, each of our neurons will initially have an encoder that is a vector pointing from the origin to somewhere inside unit circle. Neurons with encoding vectors that point in the same direction will have similar firing rates responses to the same input, as shown in the top half of **Figure 2**. By projecting our encoding vectors and input signal into the 2 + 1 dimensional unit hypersphere, our neurons are still sensitive to all parts of the original 2D input signal, but co-linear encoding vectors can also generate distinct activity, as shown in the bottom half of **Figure 2**. For example, we can have neurons sensitive to an input signal of (0, 0.5) but not (0, 1), which was not possible with our 2D preferred direction vectors. Essentially, this method allows us to have neurons sensitive to more specific parts of state space.

## 4.3. Online Learning for Adaptive Control

The neural adaptive controller presented here is a neuromorphic implementation of the control system presented in DeWolf et al. (2016), where it is proven that this adaptive controller performs as well or better than a PID controller. The neural adaptive

**FIGURE 2 |** An example of how increasing encoder dimensionality can create neurons that are more selectively responsive. On the right side of all figures the spike raster (top) and filtered output (bottom) is plotted, showing the firing rates of these neurons for the same 2D input signal that moves from [0, −1] to [0, 1], projected into 3D for **(3,4)**. In the top row we show two neurons with 2D encoding vectors (0, 0.5) [shown in **(1)** on the left] and (0, 1.0) [shown in **(2)** on the left]. The arrows in the figures represent the neuron's encoding vectors. The z-axis and color reflect the firing rates of these neurons given different (x, y) input. As can be seen, the activity of these neurons with co-linear encoding vectors is indistinguishable. In the bottom row, we project these same 2D encoders, (0, 0.5) [shown in **(3)** on the left] and (0, 1.0) [shown in **(4)** on the left], into 3D. The arrows in the figures represent the neuron's encoding vectors projected into 3D. The firing rates of the neurons given different (x, y, z) input is represented by color, showing that the neurons are responsive to different parts of 3D state space.

controller uses the Prescribed Error Sensitivity (PES; MacNeil and Eliasmith, 2011) learning rule, which is a local, spiking or non-spiking, error-driven Hebbian rule. The Loihi chip supports several different kinds of online learning, providing the ability to use microcode to define different kinds of rules. NengoLoihi implements the PES learning rule using this feature of the chip, which allows weight updates to be calculated on-chip. For the other hardware, core Nengo includes a definition of the PES rule.

The adaptive control signal is calculated via

$$\mathbf{u}_{\text{adapt}} = \mathbf{a}\,\mathbf{d}, \qquad (3)$$

where $\mathbf{a}$ is the vector of neural activities (i.e., filtered neural spike trains) and $\mathbf{d}$ denotes a vector of "decoders" which are the output weights from the ensemble. The resulting $\mathbf{u}_{adapt}$ is a vector of the same dimensionality as the OSC control signal. We initialize $\mathbf{d}$ to a vector of zeros, and use the learning rule

$$\Delta\mathbf{d} = -\kappa\;\mathbf{a} \otimes \mathbf{u}, \qquad (4)$$

to update the decoder weights, where $\kappa$ is a learning rate, $\mathbf{u}$ is the OSC's outbound control signal (acting as the error in the PES rule), and $\otimes$ denotes the outer product. This training signal, $\mathbf{u}$, was chosen based on Lyapunov stability analysis. Details, derivation, and proof of stability of this adaptive neural controller are provided in DeWolf et al. (2016).
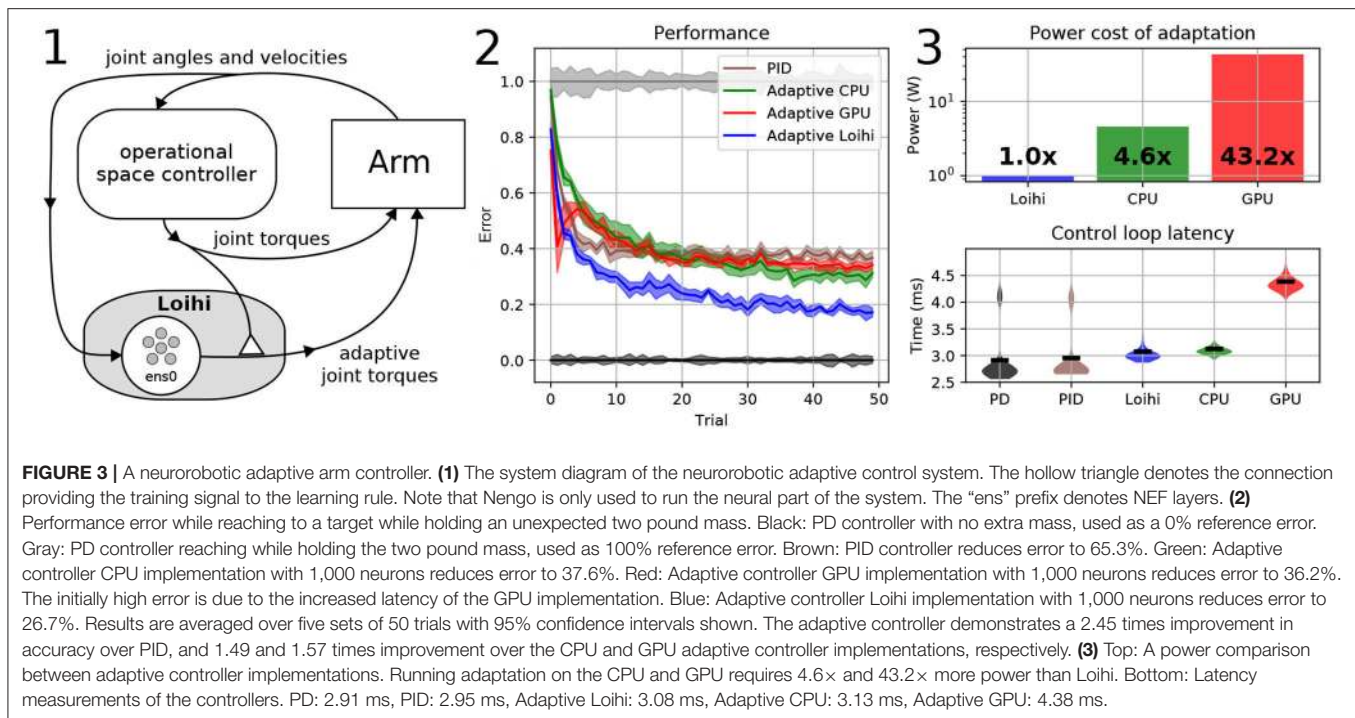
In the system diagram shown in **Figure 3**–1, the hollow triangle denotes the connection providing the training signal, $\mathbf{u}$, for the learning rule.

## 4.4. Compiling to Neuromorphic Hardware

The NengoLoihi backend is used to instantiate the neural ensemble on the Loihi with the parameters discussed above, and implement the PES learning rule on-chip. Core Nengo is used for the CPU implementation and the NengoOCL backend package is used for the GPU implementation. In this example we used a workstation with and Intel Core i7-6700K CPU @ 4.00 GHz × 8 with 32 GB RAM and GeForce GTX 1070/PCIe/SSE2 running Ubuntu 18.04 and the Intel Kapoho Bay board with 8 Loihi chips running NxSDK 0.9 In this example, we are only using one of the Loihi chips on the board.

## 4.5. Performance

**Figure 3**–2 shows the arm performing a reaching task while holding an unexpected two pound mass. In this task the arm repeatedly starts from the same position and reaches to the same target 50 times, with continuous learning between reaches. We perform the 50 reaches with each controller five times, using different randomly generated neuron ensemble parameters (such as bias and maximum firing rates), and calculate the mean error and 95% confidence intervals. The adaptive controller is run while simulating the neurons on the Loihi, and the CPU and GPU of our workstation (specifications are in section 4.4), and compared against a standard PID controller running on the same workstation. We normalize our performance results using the performance of a PD operational space controller. We consider that controller reaching under normal conditions with nothing in the hand as 0% error, and the results of that controller reaching while holding the unaccounted-for two pound weight as 100%

**FIGURE 3 |** A neurorobotic adaptive arm controller. **(1)** The system diagram of the neurorobotic adaptive control system. The hollow triangle denotes the connection providing the training signal to the learning rule. Note that Nengo is only used to run the neural part of the system. The "ens" prefix denotes NEF layers. **(2)** Performance error while reaching to a target while holding an unexpected two pound mass. Black: PD controller with no extra mass, used as a 0% reference error. Gray: PD controller reaching while holding the two pound mass, used as 100% reference error. Brown: PID controller reduces error to 65.3%. Green: Adaptive controller CPU implementation with 1,000 neurons reduces error to 37.6%. Red: Adaptive controller GPU implementation with 1,000 neurons reduces error to 36.2%. The initially high error is due to the increased latency of the GPU implementation. Blue: Adaptive controller Loihi implementation with 1,000 neurons reduces error to 26.7%. Results are averaged over five sets of 50 trials with 95% confidence intervals shown. The adaptive controller demonstrates a 2.45 times improvement in accuracy over PID, and 1.49 and 1.57 times improvement over the CPU and GPU adaptive controller implementations, respectively. **(3)** Top: A power comparison between adaptive controller implementations. Running adaptation on the CPU and GPU requires 4.6× and 43.2× more power than Loihi. Bottom: Latency measurements of the controllers. PD: 2.91 ms, PID: 2.95 ms, Adaptive Loihi: 3.08 ms, Adaptive CPU: 3.13 ms, Adaptive GPU: 4.38 ms.

error. As can be seen, the Loihi system outperforms all other controllers after 50 trials of training. Unsurprisingly, the CPU and GPU perform similarly, and better than the PID controller.

**Figure 3**–3 shows the power and latency measurements of the controllers during the task, where the neuromorphic implementation consumes the least energy of the adaptive controllers, with a minimal increase in latency compared to the non-adaptive controllers. Specifically, the CPU uses 4.6× more power, and the GPU 43.2× more. As well, the CPU is a similar latency (i.e., 2% slower), while the GPU is 42% slower than the Loihi.

To measure the power use of the CPU, we used the software package s-tui, available online at https://github.com/amanusk/s-tui. To measure the power use of the GPU, we used the nvidia-smi software. To measure the Loihi power use, we used the Linear Tech DC1613A dongle and LTpowerPlay software, which provides current and voltage measurements for the chip's two power supplies, from which we calculated the total power use.

## 5. DISCUSSION

We have demonstrated how to take advantage of neuromorphic technology to fully implement or augment existing robotic control systems. In particular, we showed how a set of tools in the Nengo ecosystem allows efficient execution of four central tasks for building neurorobotic systems.

1. The NengoInterfaces library provides an easy API for interfacing with the Mujoco simulation, used in the first example, both for sending in control signals and receiving feedback. While the second example is not directly providing simple API access, it illustrates the flexibility of Nengo to be incorporated into already developed Python programs and interfaces.

2. In the rover example, we show how NengoDL provides a natural way to integrate Keras and TensorFlow models. For non-neural perception algorithms, Python code can be directly executed from Nengo or the code can be run outside Nengo and sent into a Nengo model, as is done in the adaptive arm example.

3. We illustrated how a circuit design that solves the problem of interest can be implemented in a neural network using Nengo, providing white-box neural network systems. In the first example this was shown with the rover control network that steered and drove the rover to the target, and the second example showed the use of a vector-space training signal with stability guarantees to implement non-linear adaptive control.

4. The Nengo development toolkit allows users to compile their model to run on multiple different hardware platforms, including CPU (Bekolay et al., 2014), GPU (Rasmussen, 2019), FPGA (Morcos, 2019), Intel's Loihi (Hunsberger et al., 2018), and SpiNNaker (Mundy et al., 2015).

We have made all of the code used in these examples publicly available, to provide practical, reproducible examples for the community. We believe this set of tools and examples helps address the core challenge of making neurorobotic systems easier to build, and a wide variety of architectures easier to explore.

In the first example, while we did not benchmark or quantitatively characterize the result, it provides a demonstration of how our chosen tools allow the development of complete perception-action systems for neurorobotics. In particular,

it demonstrated how to couple white box (i.e., NEF) and black box (i.e., DNN) techniques and implement them on a single underlying spiking neuromorphic hardware platform (i.e., Loihi).

Nengo is unique in its ability to make such integration easily accessible. Nengo has several advantages in this context: (a) it is not vendor specific, and supports hardware from several sources; (b) it allows a high-level model specification for easy portability, while also allowing hardware specific details to be incorporated (e.g., via its configuration system); and (c) Nengo removes the need to have detailed knowledge about SNNs, neuromorphic hardware, simulator interfaces, embedded programming, and so on, while allowing those with such knowledge to leverage it (e.g., by easily defining new neuron models, using the configuration system, building new hardware specific backends, and so on).

Our second example provided a more quantitative characterization of the advantages of neurorobotics. From a tools perspective, it demonstrated how Nengo can be integrated into existing systems and run the same neural model across multiple kinds of hardware. But, more importantly, this example demonstrates the kinds of advantages we expect from neuromorphics: an increase in speed and accuracy, and several-fold decrease in power compared to traditional hardware. As is well-established, low latency and energy efficiency are critical for many mobile robotics applications.

Possible extensions to the examples provided here are many and varied. Perhaps one of the more obvious ones is to implement the entire adaptive controller on neuromorphic hardware. To the best of our knowledge this has only been done with a 3-link planar arm (DeWolf et al., 2016). Since the adaptive controller has performance guarantees, and the white box methods of the NEF allow us to implement it on neuromorphic hardware while preserving those guarantees, a full implementation would be a rare example of an adaptive, fully neurorobotic controller with clear performance guarantees.

While we believe the Nengo ecosystem is useful for the development of neurorobotic systems, there remain a variety of challenges and directions for future development that stand to improve it. For instance, Nengo backends that target non-spiking AI acceleration hardware, such as Google's Coral chip, would expand the community able to use the methods we have discussed because spiking neuromorphic hardware, such as Intel's Loihi chip, is not commercially or otherwise widely available. Extending the interfaces offered by the NengoInterfaces package to improve accessibility, as well as offering the same automatic conversion from DNNs to SNNs for PyTorch users also remains important future work. Perhaps most importantly, continuing to increase the number of available tutorials, ready-to-use models, and online examples is critical to reducing the startup overhead for new users and better supporting the neurorobotics, neural networks, and edge AI communities.

In conclusion, the Nengo ecosystem makes it possible for users to quickly develop applications for neuromorphic hardware, while taking advantage of already developed neural or non-neural machine learning solutions. We have shown two examples that demonstrate how the ecosystem can be used to address four core stages of the development workflow. We encourage interested researchers to use the code and tools that we have made available, and look forward to exploring the vast space of robust, embedded neurorobotics systems with the emerging research community.

## DATA AVAILABILITY STATEMENT

The datasets presented in this study can be found in online repositories. The names of the repository/repositories and accession number(s) can be found in the article/supplementary material.

## AUTHOR CONTRIBUTIONS

TD directed the project. TD, PJ, and CE implemented the examples. PJ and CE collected the data for the arm example. TD wrote the manuscript. CE provided research guidance and edited the manuscript. All authors contributed to the article and approved the submitted version.

## FUNDING

## ACKNOWLEDGMENTS

## REFERENCES

Bam Shrestha, S., and Orchard, G. (2018). Slayer: spike layer error reassignment in time. *arXiv* 1810.

Bekolay, T. (2011). *Learning in large-scale spiking neural networks* (Master's thesis), University of Waterloo, Waterloo, ON, Canada.

Bekolay, T., Bergstra, J., Hunsberger, E., DeWolf, T., Stewart, T. C., Rasmussen, D., et al. (2014). Nengo: a python tool for building large-scale functional brain models. *Front. Neuroinform.* 7:48. doi: 10.3389/fninf.2013.00048

Davies, M., Srinivasa, N., Lin, T.-H., Chinya, G., Cao, Y., Choday, S. H., et al. (2018). Loihi: a neuromorphic manycore processor with on-chip learning. *IEEE Micro* 38, 82–99. doi: 10.1109/MM.2018.112130359

Davison, A. P., Brüderle, D., Eppler, J. M., Kremkow, J., Muller, E., Pecevski, D., et al. (2009). Pynn: a common interface for neuronal network simulators. *Front. Neuroinform.* 2:11. doi: 10.3389/neuro.11.011.2008

DeWolf, T., Stewart, T. C., Slotine, J.-J., and Eliasmith, C. (2016). A spiking neural model of adaptive arm control. *Proc. R. Soc. B Biol. Sci.* 283:20162134. doi: 10.1098/rspb.2016.2134

Eliasmith, C., and Anderson, C. H. (2003). *Neural Engineering: Computation, Representation, and Dynamics in Neurobiological Systems*. Cambridge, MA: MIT Press.

Falotico, E., Vannucci, L., Ambrosano, A., Albanese, U., Ulbrich, S., Vasquez Tieck, J. C., et al. (2017). Connecting artificial brains to robots in a comprehensive simulation framework: the neurorobotics platform. *Front. Neurorobot.* 11:2. doi: 10.3389/fnbot.2017.00002

Furber, S. B., Galluppi, F., Temple, S., and Plana, L. A. (2014). The spinnaker project. *Proc. IEEE* 102, 652–665. doi: 10.1109/JPROC.2014.2304638

Gewaltig, M.-O., and Diesmann, M. (2007). Nest (neural simulation tool). *Scholarpedia* 2:1430. doi: 10.4249/scholarpedia.1430

Goodman, D. F., and Brette, R. (2008). Brian: a simulator for spiking neural networks in python. *Front. Neuroinform.* 2:5. doi: 10.3389/neuro.11.005.2008

Gutierrez-Galan, D., Dominguez-Morales, J. P., Perez-Peña, F., Jimenez-Fernandez, A., and Linares-Barranco, A. (2020). Neuropod: a real-time neuromorphic spiking CPG applied to robotics. *Neurocomputing* 381, 10–19. doi: 10.1016/j.neucom.2019.11.007

Hines, M. L., and Carnevale, N. T. (1997). The neuron simulation environment. *Neural Comput.* 9, 1179–1209. doi: 10.1162/neco.1997.9.6.1179

Hunsberger, E., Bekolay, T., Rasmussen, D., Voelker, A., Stewart, T., Patel, K., et al. (2018). *Nengoloihi*. Available online at: https://github.com/nengo/nengo-loihi

Hwu, T., Isbell, J., Oros, N., and Krichmar, J. (2017). "A self-driving robot using deep convolutional neural networks on neuromorphic hardware," in *2017 International Joint Conference on Neural Networks (IJCNN)* (Anchorage, AK: IEEE), 635–641. doi: 10.1109/IJCNN.2017.7965912

Kreiser, R., Waibel, G., Sandamirskaya, Y., and Renner, A. (2019). "Self-calibration and learning on chip: towards neuromorphic robots," in *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2019)* (Macao).

MacNeil, D., and Eliasmith, C. (2011). Fine-tuning and the stability of recurrent neural networks. *PLoS ONE* 6:e0022885. doi: 10.1371/journal.pone.0022885

Markram, H. (2012). The human brain project. *Sci. Am.* 306, 50–55. doi: 10.1038/scientificamerican0612-50

Mead, C. (1990). Neuromorphic electronic systems. *Proc. IEEE* 78, 1629–1636. doi: 10.1109/5.58356

Morcos, B. (2019). *NengoFPGA: an FPGA backend for the nengo neural simulator* (Master's thesis), University of Waterloo, Waterloo, ON, Canada.

Mundy, A., Knight, J., Stewart, T. C., and Furber, S. (2015). "An efficient spinnaker implementation of the neural engineering framework," in *2015 International Joint Conference on Neural Networks (IJCNN)* (Killarney: IEEE), 1–8. doi: 10.1109/IJCNN.2015.7280390

Rasmussen, D. (2019). NengoDL: combining deep learning and neuromorphic modelling methods. *Neuroinformatics* 17, 611–628. doi: 10.1007/s12021-019-09424-z

Ray, A., McGrew, B., Schneider, J., Ho, J., Welinder, P., Zaremba, W., et al. (2020). *mujoco-py*. Available online at: https://github.com/openai/mujoco-py

Sawada, J., Akopyan, F., Cassidy, A. S., Taba, B., Debole, M. V., Datta, P., et al. (2016). "Truenorth ecosystem for brain-inspired computing: scalable systems, software, and applications," in *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Salt Lake City, UT: IEEE), 130–141. doi: 10.1109/SC.2016.11

Slotine, J.-J. E., Khatib, O., and Ruth, D. (1988). Robust control in operational space for goal-positioned manipulator tasks. *Int. J. Robot. Autom.* 3, 28–34.

Stagsted, R. K., Vitale, A., Binz, J., Larsen, L. B., Sandarmirskaya, Y., and Renner, A. (2020). "Towards neuromorphic control: a spiking neural network based PID controller for UAV," in *Robotics: Science and Systems* (Corvalis, OR). doi: 10.15607/RSS.2020.XVI.074

Taunyazov, T., Sng, W., See, H. H., Lim, B., Kuan, J., Ansari, A. F., et al. (2020). Event-driven visual-tactile sensing and learning for robots. *Perception* 4:5. doi: 10.15607/RSS.2020.XVI.020

Todorov, E., Erez, T., and Tassa, Y. (2012). "Mujoco: a physics engine for model-based control," in *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems* (Vilamoura: IEEE), 5026–5033. doi: 10.1109/IROS.2012.6386109