Abstract

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 Jaco² robotic arm. The code and implementation details are provided with the intent of enabling other researchers to build and run their own neurorobotic systems.

Keywords

1 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 processing and control is highly desirable, the process of implementing algorithms...
Nengo and low-power AI hardware for robust, embedded neurorobotics

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 for the subset of devices known as neuromorphic hardware \[1\]. 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 special purpose low-power hardware, and integrate them into a neurorobotics control system.

3. Generating robust control signals with explainable neural networks. When generating control signals, having guarantees on performance is important, and often necessary. To this end, users needs to know exactly what operations are being implemented to guarantee stability. The Neural Engineering Framework (NEF; \[2\]) offers white-box neural network development methods, so it can be useful to integrate these methods into neurorobotics control systems.

4. Compiling neural networks to run on multiple targeted hardware platforms. During the process of designing the control or perception systems it is often desirable to develop neural network models on a standard hardware with minimal compilation overhead. Once a prototype network is working, it should be straightforward to compile to targeted special purpose hardware. To this end, 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/ \[3\]). As part of this workflow, we take advantage of the NengoInterface package to streamline interfacing with different hardware and 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 \[4\]. We also show how Nengo can either serve as the main development environment, or be used to augment a pre-existing robotics application.

To illustrate two variations on this workflow, we describe 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. The rover has 4 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 added adaptive component on standard hardware, as well as Loihi, where we take advantage of its on-chip learning. We compare various hardware 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 unmodelled force of holding a two pound weight. We discuss the workflow bottlenecks and challenges that are encountered, addressed, and remaining.
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 [5]. The simulated rover we use is a 4 wheeled vehicle built using Mujoco’s XML modeling language with Ackerman steering and rear differential drive. 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 when the centre-of-mass of the robot is within 50 centimetres 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 mounted 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 rate neurons, to confirm that the desired level of performance can be achieved with our network architecture. We then use 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 analogues, 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 [6] 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 relative to the rover consists of 2 convolutional layers and 3 dense layers, and is shown in the bottom block of Figure 1. The first convolutional layer uses 1x1 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 over to the network layers running on the Loihi (as the Loihi can only accept spikes as input). The input image is 32*128 pixels with 3 channels, chosen based on communication bandwidth limitations of the Loihi. Each of the mounted cameras renders a 32*32 image and they are concatenated horizontally. Of the three dense layers, the first two run on the chip, and the third decodes the target (x, y) estimate using a linear activation function. As a result it is optimized out during the mapping onto the chip. To communicate the signal from the last layer running on-chip, neural probes are created to report the activity of the neurons in that layer.

The dataset used for training the model was generated by recording input from the mounted cameras and the relative distance to the target, based on absolute location information provided by Mujoco. The data was collected while our non-spiking control system drove the rover to track targets, recording every 10th frame. The final data set used consists of roughly 40,000 images and target (x, y) locations.

Training the network with non-spiking ReLU activation functions using standard DNN tools (i.e., Keras) was the first step. This allows easy validation of the network architecture as being able to perform this task. To convert the network into spiking neurons, we had to take two factors into account: neuron firing rates; and the activation function of neurons running on the Loihi. For conversion into a spiking neural network targeted for Loihi, we have found that it helps to ensure that neurons in the network should, on average, fire with a frequency of about 50Hz. To achieve this, we initialize the weights of our network using the scale_firing_rates parameter of the NengoDL built-in Keras converter. Specifically, we initialize the weights of the first layer to large values, in our case we used scale_firing_rates=400. This parameter enables the user to encourage the firing rates converged upon during training to be higher or lower, based on the provided scaling. An alternative, and more fine-grained and reliable method, is to add a firing rate regularization term to the cost function that will penalize 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. 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 where they are required for filtering noise. For this network, when converted to spiking neurons, we don’t require any synapses between the layers. We did find empirically, however, that applying a filter to
the output provided a benefit to downstream control generating components. As a result, in the final implementation we use a low-pass filter with a 0.05s time constant to smooth output predictions of the target location from this network.

3.3 Generating robust, explainable control signals using the NEF

The motor system for the rover is implemented using the white box neural circuit design methods of the NEF, which represents vectors in neural activities and determines connection weights to apply functions to represented vectors in the connections between ensembles of neurons. Because we know the desired vectors and functions at each step in a controller, the NEF provides an effective method for conversion from a control description to a spiking neural network. The first step in this implementation of the control system is deriving the functions for vehicle acceleration and steering wheel control.

We designed the acceleration to be calculated as the distance to target multiplied by a gain term and clipped at a maximum value to prevent the rover from flipping when travelling at top speed and turning sharply. Formally,

\[ u_{\text{acceleration}} = k_a \min(|\text{target}|, 1) \]  

where \( u \) is the control signal sent to the rover, \( k_a \) is the acceleration gain term, and \(|\text{target}|\) is the 2-norm of the target relative to the rover.

We calculate the torque to apply to the steering wheels with a simple proportional controller that calculates the difference between the current angle of the wheels and the desired angle of the wheels and applies a gain term. The desired angle of the wheels 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,

\[ u_{\text{steer}} = k_p (\arctan2(-x, y) - q) \]  

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 results in two parallel ensembles, with the connections weights on the outbound connections calculated to approximate the above equations using the second principle of the NEF, as shown in top portion of Figure 1. In Nengo, we assign the relevant variables to each population, and then specify the function to be computed on particular connections as 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 acceleration function requires a subset of the variables used in the steering function, by using an ensemble that only encodes the target \((x, y)\) estimates we can achieve greater precision.

In the hardware implementation, each ensemble consists of 4096 Loihi leaky integrate-and-fire neurons, spread across 4 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 220Hz, chosen because in general higher firing rates provide for more accurate function approximation. However, there are limits on firing rates of Loihi ReLU neurons before discretization causes the behaviour of the neuron to depart substantially from that of a standard ReLU.

3.4 Integration and compiling to hardware

Putting the vision and control networks together is a simple matter of connecting the output of one subnetwork 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. However, they can be run using a CPU backend, which is the Nengo default, or a GPU
Figure 2: 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. 2) Reaching to a target while holding an unexpected two pound mass. Results are averaged over five sets of 50 trials with 95% confidence intervals shown. Black: PD controller with no extra mass, used as a 0% reference error. Grey: 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%. 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 the adaptive controller implementations relative to the Loihi implementations. Running adaptation on the CPU and GPU requires 4.6x and 43.2x more power than Loihi. Bottom: Latency measurements of the controllers. PD: 2.91ms, PID: 2.95ms, Adaptive Loihi: 3.08ms, Adaptive CPU: 3.13ms, Adaptive GPU: 4.38ms

backend using NengoOCL or NengoDL directly in Nengo. The NengoLoihi backend can then be used to compile the network to run on the Loihi. During the compiling stage the backend will provide alters regarding any memory constraint violations. We used a workstation with and Intel Core i7-6700K CPU @ 4.00 GHz x 8 with 32 GB RAM and GeForce GTX 1070/PCIe/SSE2 running Ubuntu 18.04, and the Intel Nahuku board with 32 Loihi chips 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 3 shows the (x, y) trajectory of the rover moving throughout the environment to 6 different targets in the environment, starting from the center. Figure 2 shows the perception and action signals from the network, with the target (x, y) predicted 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 50cm) 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](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; \[7\]), designed to move the hand along a target path. The adaptive controller adds an adaptive signal designed 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 dynamics of the arm to generate torques that will move the arm as desired. Whenever the arm dynamics change, for instance if it picks up an object, is subjected to external forces or perturbances, or wears down over time, performance will degrade unless the change is accurately modeled by the OSC system. In general these perturbations can be predicted in advance, so updating the controller on-the-fly is useful. This is the purpose of the adaptive controller we use here, which is implemented as a neural network. Intuitively, the adaptive controller acts as a context sensitive integrated error term. Where standard integrated error terms (such as in PID control) apply the same learned error regardless of the current joint angles or velocities, the adaptive controller learns to account for reaching errors specific to different movements in different parts of state space. The difference becomes significant in the control of highly nonlinear 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. While the adaptive controller can learn to compensate for the unexpected forces affecting the arm, it requires that the effects of the unexpected force are predictable given the input provided to the ensemble. It then learns the relationship between that input and the perturbations. 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 adaptive 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 of an arm.
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 nonlinear, we generally want to ensure that neurons in our ensemble are sensitive to localized parts of state space. Additionally, we 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. That is, 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 pick encoders that ensure neurons represent relevant parts of the state space, we begin by subtracting the mean and then normalizing the input signals for each dimension to ensure they are in the range \([0 - 1]\) given joint limits. 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. By projecting to the D+1 unit hypersphere, it becomes possible to carefully control the range of values that cause neurons in the ensemble to respond.

To provide an intuition for how this allows us to control neural responses, consider the 2D case. Assuming we have inputs in the 0-1 range along each dimension, the inputs are going to lie somewhere in the unit square. In the NEF, each of our neurons will initially have an encoder that is a unit vector that points from the origin towards a point on the unit circle, as this provides an unbiased representation of the space. If that distribution is evenly spread around the circle, only about a quarter of the neurons will be sensitive to the inputs on the unit square, as the inputs are only ever in the first quadrant of the cartesian plane. By projecting the square to the 2+1 dimensional unit hypersphere and defining our preferred directions in this 3D space, we have neurons that are sensitive to all parts of the original 2D input signal (the additional dimension allows our encoders to remain unit vectors). As well, neurons can be sensitive to some points and not others. For example, we can have neurons sensitive to an input signal of \((.5, 0)\) but not \((1, 0)\), which was not possible with our 2D preferred direction vectors, as those points are co-linear. 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 [8], where it is proven that this adaptive controller will perform as well or better than a PID controller. The neural adaptive controller uses the Prescribed Error Sensitivity (PES; [9]) 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

\[
\mathbf{u}_{\text{adapt}} = \mathbf{a} \mathbf{d},
\]

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 \(\mathbf{u}_{\text{adapt}}\) is a vector the same dimensionality as the OSC control signal. We initialize \(\mathbf{d}\) to a vector of zeros, and use the learning rule

\[
\Delta \mathbf{d} = -\kappa \mathbf{a} \otimes \mathbf{u},
\]

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. Details, derivation, and proof of stability of this adaptive neural controller are provided in [8].
In the system diagram shown in Figure 2.1, the hollow triangle denotes the connection providing the training signal, \( u \), 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 an Intel Core i7-6700K CPU @ 4.00 GHz x 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 2.2 shows the arm performing a reaching task while holding an unexpected two pound mass. The adaptive controller is run while simulating the neurons on the Loihi, 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 two pound weight as 100% 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 2.3 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 4.6x more power, and the GPU 43.2x 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](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 control systems. In particular, we showed how a set of tools in the Nengo ecosystem allow efficient execution of four central tasks for building neurorobotic systems. As well, we demonstrated how standard tools, like Keras and Python naturally integrate into this workflow. 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.

The first example demonstrated how to build an integrated system running on neuromorphic hardware and test it in simulation. 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.

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. Since that controller has performance guarantees, and the white box methods of the NEF allow us to implement it on neuromorphic hardware while preserving those guarantees, such an implementation would be a rare example of an adaptive, fully neurorobotic controller with clear performance guarantees.

While we believe that 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. In addition, the automatic conversion of Keras/TensorFlow models to Nengo models via NengoDL is effective for many popular DNN models, but should be extended to cover a wider variety network architectures. Similarly, offering the same ability to PyTorch users 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.

Conflict of Interest Statement

This work was funded by Applied Brain Research, Inc. Nengo and related packages belong to Applied Brain Research, and are freely available for non-commercial use.

Author Contributions

TD directed the project, TD, PJ, and CE implemented the examples, PJ and CE collected data for the arm example, TD wrote the manuscript, CE provided research guidance and edited the manuscript.

Funding

All funding was provided by Applied Brain Research, Inc.
Acknowledgments

The authors would like to thank Daniel Rasmussen, Eric Hunsberger, and Xuan Choo for their help in the development of this project.

Data Availability Statement

The code and datasets generated and analyzed for the neurorobotic adaptive arm control example can be found at https://www.github.com/abr/neurorobotics2020.

References

[1] Carver Mead. Neuromorphic electronic systems. *Proceedings of the IEEE*, 78(10):1629–1636, 1990.

[2] Chris Eliasmith and Charles H Anderson. *Neural engineering: Computation, representation, and dynamics in neurobiological systems*. MIT press, 2004.

[3] Trevor Bekolay, James Bergstra, Eric Hunsberger, Travis DeWolf, Terrence C Stewart, Daniel Rasmussen, Xuan Choo, Aaron Voelker, and Chris Eliasmith. Nengo: a python tool for building large-scale functional brain models. *Frontiers in neuroinformatics*, 7:48, 2014.

[4] Mike Davies, Narayan Srinivasa, Tsung-Han Lin, Gautham Chinya, Yongqiang Cao, Sri Harsha Choday, Georgios Dimou, Prasad Joshi, Nabil Imam, Shweta Jain, et al. Loihi: A neuromorphic manycore processor with on-chip learning. *IEEE Micro*, 38(1):82–99, 2018.

[5] Emanuel Todorov, Tom Erez, and Yuval Tassa. Mujoco: A physics engine for model-based control. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 5026–5033. IEEE, 2012.

[6] Alex Ray, Bob McGrew, Jonas Schneider, Jonathan Ho, Peter Welinder, Wojciech Zaremba, and Jerry Tworek. mujoco-py, https://github.com/openai/mujoco-py, 2020.

[7] Jean-Jacques E Slotine, Oussama Khatib, and D Ruth. Robust control in operational space for goal-positioned manipulator tasks. *International Journal of Robotics & Automation*, 3(1):28–34, 1988.

[8] Travis DeWolf, Terrence C Stewart, Jean-Jacques Slotine, and Chris Eliasmith. A spiking neural model of adaptive arm control. *Proceedings of the Royal Society B: Biological Sciences*, 283(1843):20162134, 2016.

[9] David MacNeil and Chris Eliasmith. Fine-tuning and the stability of recurrent neural networks. *PloS one*, 6(9), 2011.