frontiers

Embodied Neuromorphic Benchmarks

Terrence C. Stewart [1,*], **Ashley Kleinhans** [2] **and Chris Eliasmith** [1]

[1] *Centre for Theoretical Neuroscience, University of Waterloo, Waterloo, ON, Canada*
[2] *Mobile Intelligent Autonomous Systems group, Council for Scientific and Industrial Research, Pretoria, South Africa*

Correspondence*:
Terrence C. Stewart
Centre for Theoretical Neuroscience, University of Waterloo, Waterloo, ON, Canada,
tcstewar@uwaterloo.ca

ABSTRACT

Evaluating the effectiveness and performance of neuromorphic hardware is difficult. It is even more difficult when the task of interest is an embodied task; that is, a task where the output from the neuromorphic hardware affects its future input. However, these embodied situations are one of the primary potential uses of neuromorphic hardware. To address this, we present a methodology for embodied benchmarking and a particular example of a benchmark for adaptive control of an arbitrary system. These benchmarks are flexible, in that they allow researchers to explicitly modify the benchmark to identify particular task domains where particular hardware excels. Furthermore, the benchmarks make use of a hybrid of real physical embodiment and a type of "minimal" simulation that has been shown to lead to robust real-world performance, allowing the same benchmark to be used by many researchers.

Keywords: neuromorphic hardware, benchmarking, minimal simulation, adaptive control, neural networks

1 INTRODUCTION

Neuromorphic hardware holds great promise for a wide variety of applications. The combination of massively parallel computation and low power consumption means that there is the potential to have complex algorithms running in embedded processing situations, without being a significant drain on battery life. The question, of course, is to identify what sort of always-on or interactive functionality is feasible with these devices.

To evaluate applications of this hardware, we need benchmark tasks. These tasks must allow us to compare across different instances of neuromorphic hardware (and potentially across different algorithms implemented in said hardware). This allows us to quantitatively compare systems, letting researchers both measure the progress in the field and also directly compare competing approaches.

In this paper, we focus on the development of *embodied* benchmarks. These are tasks where the output of the neuromorphic hardware *influences its own future input*. This is in contrast to standard categorization or pattern identification tasks, where the input is some fixed sequence, and the main question is whether the hardware produces the correct output for each input (or input pattern).

We believe embodied benchmarks should be of particular interest to neuromorphic research, given that many of the applications of neuromorphic hardware are in exactly this domain of embedded and

interactive control of robotic or other physical systems. However, embodiment raises a number of issues that complicate the development of such benchmarks. Rather than simply providing a data file of inputs and desired outputs, the benchmark must either specify a full physical system for that embodiment, or it must provide software for a simulation of that system. As we discuss below, either approach is problematic, and addressing this difficulty is the primary goal of this paper.

2 EMBODIED BENCHMARKS

An embodied benchmark task is one where the system we are studying has a two-way interaction with some sort of environment. That is, the outputs from the neuromorphic hardware are sent to the environment where they cause some sort of effect, the results of which change the subsequent input. For example, the outputs might control the movement of a robot, which in turn affects the sensory data received by the robot.

2.1 Simulation versus Physical Instantiation

To define such a benchmark, we need to be explicit about the embodiment. If it is a situation where a robot is to be controlled, we need to be explicit about all the details of that robot. What motors are there? How are they configured? How strong are they? What sensors are there? Where are they placed? How accurate are they? However, even if these questions are answered, there is a fundamental problem in that *other researchers need access to that exact robot*. If a benchmark is to be widely used, other researchers developing their own hardware should be able to do their own testing.

Furthermore, using a physical robot also imposes significant practical difficulties when performing extensive benchmark testing. When testing, we often want to run the same task over and over again, both for robustness and to see the effects of varying parameters. With a physical robot, this means manually setting up the task, letting the test run, gathering the resulting data, and then resetting the robot back to the initial state. This means that issues like battery life become problematic, and not just because there is a limited amount of time available for testing. As the battery level changes, the performance of the robot itself can also change. Futhermore, for any rigorous testing of the benchmark, we will want to examine situations where the system fails. This means that some of the testing will involve parameter settings that lead to poor behaviour, which could involve physical damage to the robot itself.

However, *not* using a real physical embodiment for testing is also problematic. First and foremost, without an actual real-world task, why should anyone have any confidence that the performance on the benchmark is reflective of the actual usefulness of the neuromorphic hardware? It is very widely known that *simulations of robots* (or other physical systems) are often much easier to control and better-behaved than the real thing. The field of robotics is filled with algorithms that work well "in theory," but fail when run on actual hardware. We do not want a benchmark that falls into this trap of failing to generalize to real situations.

Furthermore, neuromorphic hardware has another constraint that severely limits the possibilities of simulation. The normal approach when a simulation is too simple to reflect reality is to add details to the simulation itself. Incredibly finely detailed simulations can be created, filling in all of the details needed. However, those resulting simulations *cannot be run in real-time*. This is a fundamental problem, in that most neuromorphic hardware is about real-time interactions, and there is no way to slow down the hardware to match the simulated environment. This means that even if we spent the considerable amount of research effort needed to define a simulated environment for an embodied benchmark, that simulation could not be run fast enough to interact with most of the hardware that we would want to test.

2.2 Minimal Simulation

The above considerations seem to indicate that even though using real-world phyiscal hardware for benchmarking is problematic, it is still better than using simplistic simulations which may not generalize to real tasks. However, we believe there is an alternate approach known as *Minimal Simulation* (Jakobi 1997).

First, we note that the problem faced here is remarkably similar to a problem faced by the evolutionary robotics community. In evolutionary robotics, the goal is to use genetic algorithms to *evolve* systems that can control robots to perform various tasks. These tasks can be as simple as navigation and obstacle avoidance, but have also included walking, object identification, and visual tracking [refs???].

However, performing this evolution on real physical robots is problematic for the same reasons that benchmarks on physical robots are problematic. The robots must be reset to the same state each time; they often involve behaviour that can physically damage the robots; and they take a very long time to run. For this reason, attempts were made to evolve using simulated robots. However, the general finding was that algorithms that worked on the simulated robots would not work when run on the real physical robots. If the simulations were improved, adding complex physical detail, then it was possible to generalize to real behaviour; unfortunately, such complex simulations would run slower than real-time [refs].

To address this problem, Nicholas Jacobi proposed the creation of "minimal" simulations. These are simulations where there is variability *within the simulation itself*. In other words, we make *poor* simulations, but ensure that the way in which it is poor is itself variable. We then make sure that the controllers work across that whole range of variability. "Instead of trying to eliminate the differences between simulation and reality, they are acknowledged, and mechanisms are put in place to prevent evolving controllers from relying on them." [ref: jacobi thesis].

With this approach, it was possible to build minimal simulations that would run faster than real-time and yet also be complex enough that if a system could successfully control the simulation, it was also likely to successfully control a real robot. To achieve this, the simulations are made to be unreliable in almost every respect. For example, for a simulation of a simple motor it would still be the case that if power is applied it would generally try to spin, but the exact amount of torque, the amount of sensory noise, the amount of time needed, the amount of static and dynamic friction, and so on would all be randomly chosen. A successful controller would have to deal with this wide range of variability, and if it could handle that variability then we would have reason to believe it could also handle the real system.

It is also worth noting that a minimal simulation *only has to be a good simulation for successful behaviour*. That is, "if we are evolving corridor following behaviour, the dynamics of the simulation might differ wildly from those of reality if the controller hits a wall or goes round in circles, but this does not matter, since the controllers we are interested in transferring across the reality gap will neither hit walls nor go round in circles." If the controller is poor, we do not need the simulation to be at all accurate in exactly *how* that poor behaviour is manifest. We do not need an exact detailed physics model of the collision between a robot and a wall, or a detailed model of what happens to a robot arm when it starts oscillating wildly due to a poor control signal. All we need is for the simulation to be just good enough to indicate that things have gone wrong, and thus give a low score to that controller. This means that, for example, in a minimal simulation of an eight-legged walking robot, it is not necessary to have a physics simulation that correctly models what happens when two legs collide with each other. Rather, if legs collide with each other, that is an indication that the walking behaviour is very poor, so as long as that result is indicated we can greatly simplify the simulation by not including all the details necessary to deal with these sorts of physical interactions.

Given the success of this approach for evolutionary robotics, we propose that it can be directly used for embodied neuromorphic benchmarks. To do this, we create software simulations for each benchmark task. These simulations must be fast enough to run in realtime (so that they can be controlled by real neuromorphic hardware), and they must be extremely variable. Each time the simulation is run, different parameters will be chosed for this variability (so one run might have a large degree of sensor noise while the next run has none at all; one run might have more delay in the motor response and another might have less power available). Being successful at the benchmark means being successful across all this variability.

The result should be a benchmark that can be run by any researcher. The fact that it is a simulation means that source code can be shared, and that no specialized hardware is needed. Furthermore, the variability in the simulation itself can be controlled, and this can help give a rich characterization of the benchmarked hardware. For example, some hardware might only work with small amounts of sensor noise, or other hardware might only work when there is significant delay in the motor response. This flexibility in parameters in the benchmark allows researchers to explicitly characterize that particular siuations where their hardware excells.

2.3 Cheap Robotics

All of that said, we also need real physical embodiment as part of any benchmark of this kind. While we claim that reliable minimal simulations are relatively easy to create, they are not a widely-used technique, and the real proof of behaviour is always in the real world. For this reason, we also argue that every benchmark based on a minimal simulation should also have an easy-to-construct physical analog. However, this physical version is not to be used as the primary benchmark. Rather, this is meant as a double-check that the system does actually behave as expected.

For this physical aspect of the benchmark, we recommend cheap, widely-available components. This allows a greater chance for other researchers to have access to the same (or similar) hardware. For the particular example benchmark described in the next section, we use the Lego Mindstorms EV3 kit, a simple robotics platform available at most toy stores.

It is important to note that there is actually a theoretical advantage to using cheap robotics hardware for benchmarking, in addition to the practical advantages. In particular, we *don't want benchmarks that rely on particular high-speed, high-accuracy devices*. The purpose of the benchmark is not to indicate how well this neuromorphic hardware works to control this one particular robot in this task. Rather, the purpose of a benchmark is to characterise how well this neuromorphic hardware works on this task *in general*. The variability in the minimal simulation means that it should be able to function across a wide variety of physical embodiments, and so if we are to choose one particular physical embodiment to test in the real world, then we should choose one that is not extremely high-precision. For this reason, we believe using cheap Lego robot is actually more useful for benchmarking than an expensive high-precision robot.[1]

3 A BENCHMARK: ADAPTIVE MOTOR CONTROL

To demonstrate this approach to creating embodied neuromorphic benchmarks, we now consider a basic control task. Suppose we have a system with a number of joints q and we want to send an output u to those motors such that the joints move to a particular desired position q_d. Our only output is the signal u (one for each motor) and our only input is the current position of each motor q.

[1] Of course, for more complex benchmark tasks we may need sensory and motor capabiities that are beyond that of a simple Lego robot.

The simplest controller for such a situation is a P (proportional) controller, where $u = K_p(q_d - q)$. This is often supplemented with a D (derivative) term, which helps to slow the system down as it approaches the desired position, thus avoiding overshooting and oscillation ($u = K_p(q_d - q) + K_d(\dot{q}_d - \dot{q})$), leading to the standard PD controller. Both K_p and K_d are constants that can be tuned to particular situations.

However, this controller has difficulty in the presence of significant external forces. For example, consider a single motor controlling the angle of a single arm. If the arm is held out to the side, gravity acting on the mass of the arm itself will pull the arm downward. Thus to hold the arm still at a particular q_d will require the controller to apply a force to counteract gravity. Since the PD controller always produces an output $u = 0$ when $q = q_d$, it cannot compensate for this, and so the arm will stay stationary at some angle below the desired angle. [TODO: add diagram]

The standard solution to this problem is to add an I (integral) term ($K_i \int (q_d - q)dt$) to the controller, making it a PID controller. The idea here is that as the difference between where it is and where we want it to be accumulates over time, the K_i term gradually increases how much extra force is being applied until it is large enough to counteract the external force of gravity (or whatever other external forces are present). However, this approach has great difficulty when q_d changes, since the external force due to gravity changes depending on the position of the arm q. The controller ends up having to use the accumulated integral to "relearn" the correct amount of extra force needed every time q_d changes.

In some robotics applications, the solution to this problem is to mathematically analyze the geometry and mass of the system and compute exactly how much extra force is needed. In this particular case, the answer is straight-forward, in that the extra torque due to gravity is $\tau = mg\frac{l}{2}sin(q)$, where m is the mass of the arm, l is the length, and g is $9.8m/s^2$. If the force applied by the motor is linear in u, then we could simply compute this value and add it to our controller's output. However, this assumes a perfectly even distribution of weight in the arm, ignores momentum and other forces, and gets much more complex as more joints are added. Furthermore, if this initial computation is slightly off, or if details of the system change, there is no way to adjust this compensation.

Fortunately, there is an adaptive solution to this problem, and it is one that fits well with neuromorphic hardware. Slotine and Li (1987) shows that if you express the influence of these other external forces as $\tau = Y(q)\omega$ (where $Y(q)$ is a fixed set of functions of q, such as $sin(q)$, and ω is a vector of scalar weights, one for each function in Y), then you can learn to compensate for these external forces by using the learning rule $\Delta\omega = \alpha Y(q)u$, where u is the basic PD control signal.

Importantly, as pointed out by Sanner and Slotine (1992) and Lewis (1996), rather than making explicit assumptions about the exact functions that should be in $Y(q)$, we can use a neural network approach where each neuron is a different function of q. As long as there is enough hetereogenetity (i.e. as long as the neural activity forms a basis space that spans the desired space of functions), then the learning rule will continue to work. This approach has been used in the Recurrent Error-driven Adaptive Control Hierarchy model of human motor control (DeWolf 2014), and for quadcopter control [TODO: any citations for this?].

This then suggests an explicit neuromorphic benchmark. The input to the neuromorphic hardware is q, the system state. This input is fed to each neuron such that each neuron produces some output behaviour that is based on this input. Since q will be multi-dimensional (if there is more than one joint), we may give each neuron a random weighting of each q value ($J_i = e_i \cdot q$, where J_i is the input to neuron i, and e_i is a randomly chosen vector[2]). Given this input, the neurons will produce some output A. we now form a

[2] e could also be chosen so as to regularly span the space of possibilities

weighted sum of these outputs Ad, where d is a matrix (number of neurons by number of elements in q) that is initially all zeros.

To use this controller, we add its output to that of the standard PD controller. That is, the standard controller has $u = K_p(q_d - q) + K_d(\dot{q}_d - \dot{q})$, and so our actual output to the motor is $u + Ad$. We then apply a learning rule on d such that $\Delta d = \alpha A \times u$.

Notice that we can think of this system as a three-layer neural network, where the input and output layers are linear. The first layer is q, the input state, one value for each joint. The "hidden" layer is A, the activity of a large number of neurons. The output layer again has one value per joint, and is the extra added signal to apply to the motors, Ad. Given that this is such a canonical example of the use of neural networks, we hope that the majority of neuromorphic hardware is flexible enough to implement exactly this model.

3.1 Online and offline learning

The one major step here that does not exist in a lot of neuromorphic hardware is the ability to update the weights d. For hardware that does have a built-in learning rule, this rule is at least of a very common form, where the weight update from a neuron is proportional to the activity of that neuron and an external error signal. This makes it an instance of the ubiquitous delta rule, and hopefully supported by the hardware.

However, if the neuromorphic hardware being benchmarked does not have the ability to update weights online using a learning rule of this form, then there are two solutions. First, the multiplication by d could be done on the output from the neuromorphic hardware. There has to be some system to take the neural output from the hardware and send it to the motor (or the simulation of the motor). Instead of outputing the result of Ad, the hardware could output A (the activity of all the neurons), and the interface to the motor could be responsible for doing the multiplication by d and updating d according to the learning rule.

Alternatively, we can use offline learning. That is, rather than updating the weights d all the time, we simply record A and u, and then after a period of time stop the controller, compute the sum total of the changes to d, load the new value of d onto the neuromorphic hardware, and then start the controller again.

3.2 Minimal Simulation for Adaptive Control

Now that we have defined the task, we can use the principles of Minimal Simulation to construct a flexible and variable simulated environment for testing adaptive control. The idea is to make a bare-bones simulation of the system being controlled, with built-in variability. If the neuromorphic controller works well across this variability, then it is likely to work well outside of simulation as well.

The basic system variable is a vector of length N holding the joint angles q. Each joint has a velocity v. The force applied by each motor is related to the signal sent to the motor u, but will generally have some maximum value T, so we use $tanh(u)T$. For friction, we simply scale the velocity by some factor F every time step. This results in the simplistic simulation as follows:

$$\Delta v = -vF + tanh(u)T \tag{1}$$

$$\Delta q = v \tag{2}$$

On top of this, we need to add the external force for which the adaptive system is trying to compensate. Rather than deciding on what specific external force, we *randomly generate* an equation controlling this

force. We start with a small set of smooth functions on X which are often found in dynamics equations (x, x^2, $sin(x)$). We then generate an external force of $Q(\zeta X(\beta q + \gamma) + \eta)$ where ζ, β, γ, and η are all randomly chosen fron $N(0, 1)$ and Q is a scaling factor to control how strong this external force is. The result is added to Equation 1.

Finally, we add random noise, delay, and filtering to both the input and the output of the system. For noise, we add $N(0, \sigma_u)$ to the control signal u and $N(0, \sigma_q)$ to the q value reported back to the controller. We also use a low-pass filter to smooth both values (with time constants τ_u and τ_q) after this noise is added. Finally, both q and u are delayed by an amount of time t_q and t_u.

The result is not meant to be a simulation of a particular physical embodiment. Rather, the variability in this simulation is meant to be extremely fast to simulation, and to make it hard to "cheat" to control it. In other words, if a controller manages to be able to control the various randomly created minimal simulations of embodiment that are generated with this approach, then we have reason to believe that it will also be successful at controlling real embodiments. With this simulation and modern computers, we can easily simulate in real-time systems with dozens of joints and highly complex interactions between them, and measure how well an adaptive controller deals with these situations.

3.3 Cheap Robotics for Adaptive Control

Of course, we need physical confirmation that the minimal simulation defined in the previous section does a reasonable job. For this, we define a particular easy-to-build example of a system that can be usefully controlled by this adaptive method. For this, we use the Lego Mindstorms EV3 robot kit.

The physical body is depicted in [TODO: add figure]. It consists of a single motor, mounted in such a way that there is a significant amount of weight on the arm itself (the weight of the motor itself). Multiple motors can be added, and other configurations can be considered and should also be suitable for testing control, but here we just consider the basic case.

To interface to the physical hardware, we installed the `ev3dev` operating system (`http://ev3dev.org`), a Debian-based Linux system specifically developed for the EV3. We then installed and ran the `ev3_link` program from `ev3dev-c` (`https://github.com/in4lio/ev3dev-c`). This allows the EV3 to listen for UDP commands that tell it to set motor values and read sensor values. Communication with a PC was over a USB link (although the system also supports WiFi communication). With constant communication, the system is able to adjust the power sent to the motors u and give position feedback q from those motors at a rate of around 300Hz.

Figure [???] shows the system being controlled by a simple PD controller. The desired joint angle q_d is varied randomly over time. Notice that the system consistently undershoots and overshoots the desired angle, due to its inability to compensate for gravity.

In Figure [???], we show the result of using the adaptive controller algorithm. The system quickly improves on the standard PD controller, learning to adjust its control signal u to compensate for the unknown external forces.

4 BENCHMARK ANALYSIS

- what do we measure? performance: rmse and delay. Also, power consumption? cost?

- for given hardware, how many neurons can be done in realtime? that's how many should be used for comparison

- show how these things change as we change parameters in the simulation

- delays

- number of motors N

- magnitude of nonlinearity Q

CPU, GPU, and SpiNNaker?

5 OTHER BENCHMARKS

(sketch out how this approach could be applied to other task)

Adaptive Jacobian

Driving down a corridor, avoiding obstacles

Classical conditioning

Operant conditioning

DISCLOSURE/CONFLICT-OF-INTEREST STATEMENT

The authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

ACKNOWLEDGMENTS

Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text. Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text.

Funding: Text Text Text Text Text Text Text Text.

REFERENCES

DeWolf, T. (2014). *A neural model of the motor control system*. Phd thesis, University of Waterloo

Jakobi, N. (1997). Evolutionary robotics and the radical envelope-of-noise hypothesis. *Adaptive Behavior* 6, 325–368

Lewis, F. (1996). Neural network control of robot manipulators. *IEEE Expert: Intelligent Systems and Their Applications* 11, 64–75

Sanner, R. and Slotine, J.-J. (1992). Gaussian networks for direct adaptive control. *IEEE Transactions on Neural Networks* 3

Slotine, J.-J. E. and Li, W. (1987). On the adaptive control of robot manipulators. *Int. J. Robotics Research* 6, 49–59. doi:10.1177/027836498700600303

FIGURES

Figure 1. Enter the caption for your figure here. Repeat as necessary for each of your figures