Neuroinformatics (2019) 17:611–628
https://doi.org/10.1007/s12021-019-09424-z

SOFTWARE ORIGINAL ARTICLE

NengoDL: Combining Deep Learning and Neuromorphic Modelling Methods

Daniel Rasmussen[1]

Published online: 10 April 2018

Abstract

NengoDL is a software framework designed to combine the strengths of neuromorphic modelling and deep learning. NengoDL allows users to construct biologically detailed neural models, intermix those models with deep learning elements (such as convolutional networks), and then efficiently simulate those models in an easy-to-use, unified framework. In addition, NengoDL allows users to apply deep learning training methods to optimize the parameters of biological neural models. In this paper we present basic usage examples, benchmarking, and details on the key implementation elements of NengoDL. More details can be found at https://www.nengo.ai/nengo-dl.

Keywords Nengo · TensorFlow · Deep learning · Computational neuroscience

Introduction

Deep learning and neuromorphic modelling share many methodological similarities: at their core, they are concerned with how groups of neurons, communicating via connection weights, can carry out some function of interest. By "neuromorphic modelling" we mean the construction of models that include increased levels of biological detail, in an effort to understand or recreate the functionality of the brain (this is on a continuum with deep learning, rather than a sharp distinction). Despite these computational similarities, researchers in the respective fields tend to be isolated from one another. We usually think of deep learning in terms of abstract nonlinear optimization problems, and practitioners are rarely concerned with applying those methods to the study of the brain (with exceptions, e.g. Kriegeskorte 2015; Yamins and DiCarlo 2016). Correspondingly, there is a perception among neural modellers that deep learning methods are limited to abstract applications of artificial neural networks, and not of great help to those interested in studying how the brain works (Kay 2017).

One significant outcome of this divide is that the tools of the two fields have become quite isolated. Deep learning researchers use, e.g., TensorFlow (Abadi et al. 2016), Theano (Team 2016), Caffe (Jia et al. 2014), or Torch (Collobert et al. 2011), while neuromorphic modellers use, e.g., Nengo (Bekolay et al. 2014), Brian (Stimberg et al. 2013), NEST (Gewaltig and Diesmann 2007), NEURON (Hines and Carnevale 1997), or PyNN (Davison et al. 2009). There is very little overlap between the two groups of users.

Our aim with NengoDL is to provide a tool that brings these two worlds together. We want to combine the robust neuromorphic modelling API of Nengo with the power of deep learning frameworks (specifically TensorFlow). In particular, there are three key design goals of NengoDL:

- Allow users to construct neuromorphic models using Nengo and then optimize model parameters using deep learning methods
- Improve the simulation speed of neuromorphic models
- Make it easy to construct hybrid models (e.g., inserting convolutional layers into a neuromorphic model, or converting a deep learning network to use spiking neurons)

In Section "Background" we discuss the two tools that form the basis of NengoDL (i.e., Nengo and TensorFlow). Section "Using NengoDL" explains the basic features of NengoDL, with usage examples. Section "Implementation" dives into the underlying implementation of NengoDL. Finally, Section "Results" presents some benchmarking data, as well as results from some more complicated examples.

All of the source code for NengoDL can be found at https://github.com/nengo/nengo-dl, and installation instructions, more examples, and API descriptions can be found in the documentation at https://www.nengo.ai/nengo-dl.

✉ Daniel Rasmussen
daniel.rasmussen@appliedbrainresearch.com

[1] Applied Brain Research Inc., Waterloo, ON, Canada

Related Work

As mentioned, NengoDL sits at the intersection between deep learning and neuromorphic modelling tools, combining Nengo (Bekolay et al. 2014) and TensorFlow (Abadi et al. 2016). The relationship of those tools to their respective fields is better described in their own papers, so we do not attempt to reiterate that here. With regards to NengoDL itself, we are not aware of any similar tools that have attempted to combine deep learning and neuromorphic modelling methods. There has been work at the intersection of deep learning and neuromorphic modelling (e.g., Esser et al. 2015; Hunsberger and Eliasmith 2016; Kriegeskorte 2015; Yamins and DiCarlo 2016; Lee et al. 2016), which often involves developing specific software or hardware implementations that combine methods from the two fields. However, none of these efforts have developed a general modelling tool for others to use.

The most closely related work is the "SNN Toolbox" (Rueckauer et al. 2017). The goal of that tool is to take a network constructed and trained using one of the deep learning packages (e.g., Theano or Caffe), and convert it into a special form of spiking neural network that will be able to match the performance of the source network as closely as possible. Although this is something a user can do in NengoDL (e.g., see Section "Spiking MNIST"), the scope of NengoDL is more general. In addition to supporting deep learning style networks, with NengoDL we are able to construct neuromorphic models (including spiking neural models), and support the simulation and optimization of both types of model (or mixtures of the two) in a unified framework.

Background

In this section we give a brief introduction to the two tools we bring together in this work, Nengo (Bekolay et al. 2014) and TensorFlow (Abadi et al. 2016). Our goal is not to give a comprehensive or representative introduction to their features, but rather to focus on those elements that are most relevant to the upcoming discussion of NengoDL. The interested reader can find more information at

- **Nengo**: https://www.nengo.ai, (Bekolay et al. 2014)
- **TensorFlow**: https://www.tensorflow.org, (Abadi et al. 2016)

Nengo

Nengo is a software framework designed to enable the construction and simulation of large-scale neural models. It has been used to develop, e.g., models of human motor control (DeWolf et al. 2016), visual attention (Bobier et al. 2014), inductive reasoning (Rasmussen and Eliasmith 2014), working memory (Choo and Eliasmith 2010), reinforcement learning (Stewart et al. 2012; Rasmussen et al. 2017), as well as large integrative models that combine these systems to produce end-to-end functional models of the brain (Eliasmith et al. 2012).

There are a number of common characteristics of these kinds of models, which Nengo is designed to support:

- **Temporal dynamics**: Nengo models are essentially temporal; they are simulated over time, and even a constant input (although inputs are rarely constant) will result in complex internal dynamics (the accumulation of neuron voltages, post-synaptic filtering, communication delays, online learning rules, etc.).
- **Complex neurons**: Nengo models typically use more complex neuron models than standard deep learning nonlinearities. In particular, these models are often simulated using spiking neurons. Managing the parameters of these neuron models as well as their internal simulation is an important element of Nengo's design.
- **Complex network structure**: Neural models are often highly interconnected (e.g., containing large numbers of lateral and feedback connections), without the regular, feedforward, layer-based structure common in many deep learning models. The Nengo model construction API has been designed to support this style of network.
- **Neuromorphic hardware**: There are a number of interesting neuromorphic hardware platforms that are in development or have been recently released (e.g., Khan et al. 2008; Benjamin et al. 2014; Davies et al. 2018). Nengo's architecture has been designed to allow the same model to run across diverse hardware backends, with minimal cognitive load on the user.

Note that none of these issues are exclusive to neuromorphic modelling. However, they are common or prominent concerns in that field, which has shaped the design emphasis of tools such as Nengo. This is why it is important to combine the strengths of Nengo with the strengths of TensorFlow, rather than simply choosing one over the other.

Architecture

As mentioned, one of the important design goals of Nengo is to allow the same model to run transparently (from the user's perspective) across a wide range of hardware platforms. Thus Nengo's architecture has been designed to provide a clean separation between the front-end code (which modellers use to define the structure/parameters of their network) and back-end implementation (which handles the simulation of that network on some underlying computational platform).

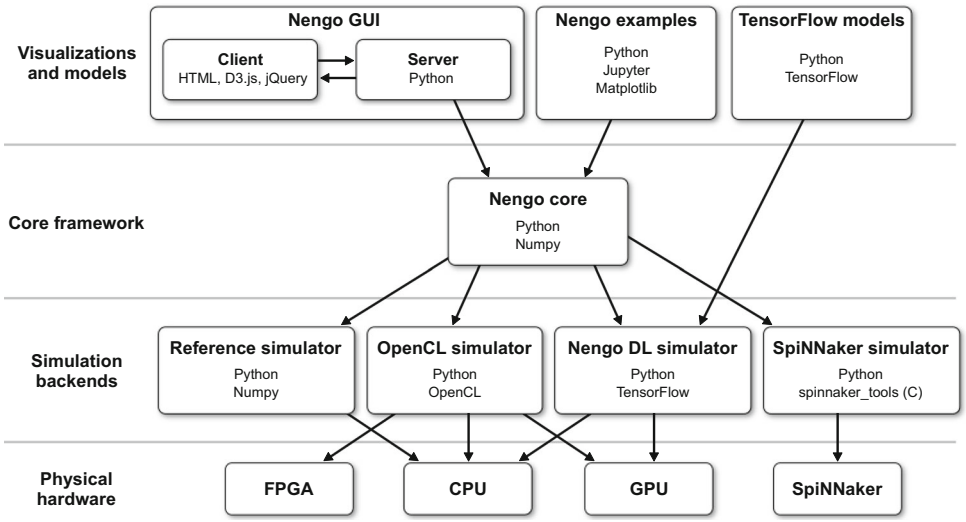

Fig. 1 Architecture of the Nengo ecosystem. The primary role of NengoDL is as a Nengo simulator, meaning that it interfaces between the core Nengo modelling API and the underlying hardware. However, NengoDL spans multiple levels, as it allows TensorFlow models to integrate in the same simulation environment, and also provides new user-facing functionality (augmenting the core framework)

Users begin by constructing a `Network`, and populating it with the objects that define their model. This is then passed to a `Simulator`, which encapsulates the back-end logic required to simulate that `Network`. For example, the default `nengo.Simulator` will simulate a network on a conventional CPU, or the user can simply replace `nengo.Simulator` with `nengo_ocl.Simulator` or `nengo_spinnaker.Simulator` to run that same model on a GPU (using OpenCL) or SpiNNaker (custom neuromorphic hardware; Khan et al. 2008), respectively.

In the case of NengoDL, we provide a `nengo_dl.Simulator` that will simulate a Nengo network via TensorFlow (Fig. 1). Thus NengoDL resides primarily on the back-end side of the Nengo architecture (although it does provide some new user-facing features, which we discuss later). In other words, the model construction process is largely unchanged from the user's perspective when switching from Nengo to NengoDL; any model constructed for the default `nengo.Simulator` will also run on `nengo_dl.Simulator` and produce the same output (within the bounds of floating point precision). Thus we give a brief introduction to the front-end side of Nengo here, but focus primarily on the back-end. More details on the front-end can be found at https://www.nengo.ai or Bekolay et al. (2014).

Front-End Objects

A Nengo model is composed of 5 basic objects:

- **Network**: Acts as a container for other Nengo objects
- **Ensemble**: A group of neurons
- **Node**: Used to insert signals into a model (e.g., sensory input)
- **Connection**: Used to connect Nodes or Ensembles, allowing objects to pass information to one another
- **Probe**: Extracts data from the model for analysis

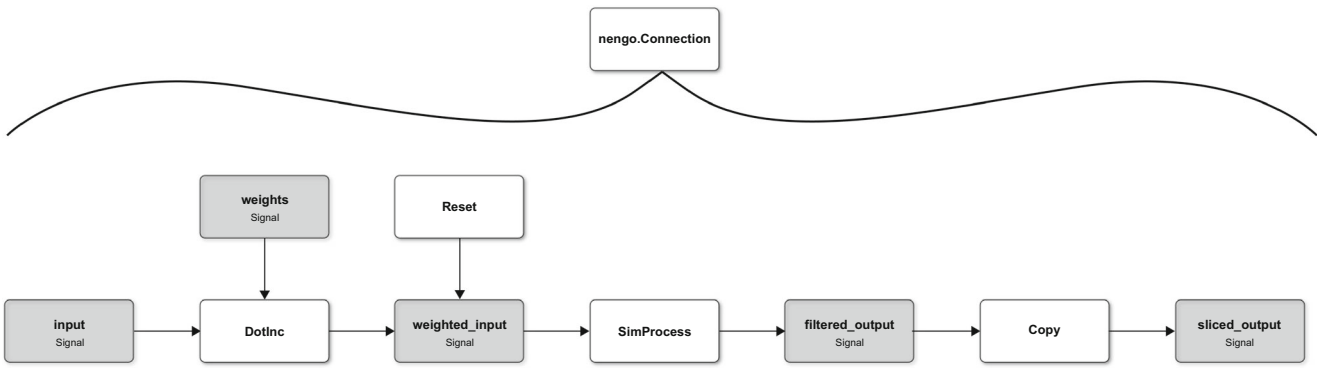

Fig. 2 An example of how a `Connection` is translated into lower level `Operations` and `Signals`. The `DotInc` op multiplies the input signal (the source of the `Connection`) by the connection weights, and adds the result to another signal (which is `Reset` to zero at the beginning of each timestep). Next a `SimProcess` op implements the synaptic filtering. Finally, a `Copy` op copies the filtered input signal to the appropriate indices of the output signal (the destination of the `Connection`)

We can think of these objects as defining a graph, where Nodes and Ensembles are the vertices and Connections are the edges. Each of these objects, in addition to defining the structure of the graph, stores information about the parameters of that object (e.g., neuron types and bias values for the `Ensemble`, or synaptic filters and connection weights for the `Connection`). Thus in the end we can think of a front-end Nengo network as a large data structure defining the structure and parameters of a model. It is then the task of the back-end to take the information encoded in that data structure and construct an implementation that matches that specification.

Back-End Objects

In general, a Nengo back-end is free to process the input `Network` however it wants. However, Nengo provides a builder that translates the high-level Nengo objects described above into a collection of lower-level operations (Fig. 2). This intermediate representation is often useful when designing a `Simulator`, as it is closer to the underlying computational operations that need to be executed on the back-end platform. This intermediate representation consists of two basic objects:

- **Signals**: A `Signal` is a generic tensor array that stores internal simulation values
- **Operators**: An `Operator` reads data from some number of input `Signals`, performs some computation, and writes the result to some output `Signals`

There are a number of basic `Operator` types, such as `Copy` (to copy a value from one signal to another), `ElementwiseInc` (computes the element-wise multiplication of two input signals and adds the result to some output signal), or `DotInc` (computes a matrix-vector multiplication). There are also Operators for the different neuron types (e.g., `SimLIF`, which reads signals containing the input currents and internal state values of a group of leaky-integrate-and-fire neurons and computes output spikes) or online learning rules.

The task of the back-end is then to provide a concrete implementation for these operations. For example, the default `nengo.Simulator` uses the Python `numpy` library, where `numpy.ndarray` is used to represent `Signals` and, e.g., `numpy.dot` is used to implement `DotInc`. The first challenge for NengoDL is to do the same, but using TensorFlow to implement these basic operations.

TensorFlow

TensorFlow is a software framework developed by Google (Abadi et al. 2016). Its primary use case is deep learning, but we can think of it more generally as a numeric computation library that we want to use to run a neural simulation.

TensorFlow uses a declarative programming approach, meaning that rather than directly specifying the steps of the program (imperative programming) the user specifies at a more abstract level the computations they want to perform. This declarative programming looks a lot like the Nengo back-end framework described above; at its core it consists of `Tensors`, which represent values, and `Ops`, which perform computations on some input `Tensors` to produce output `Tensors`. The programmer begins with some input `Tensors`, and then builds up a computation graph by applying different `Ops` that represent various transformations. For example, `y = tf.matmul(a, b)` adds a Tensor `y` to the graph that represents the matrix multiplication of two other tensors `a` and `b`. The user can then ask TensorFlow to compute the value of `y` (or any other `Tensor` in the graph), and TensorFlow will translate that declarative specification into actual steps that are executed on the CPU or GPU to compute the value.

There are two key features of TensorFlow that we take advantage of in NengoDL:

- **Automatic differentiation**: Specifying our programs via this declarative graph enables various automated manipulations of that graph. For example, we can add an element to the graph that represents the derivative $\frac{\partial y}{\partial a}$, and TensorFlow will automatically add all the elements to the graph required to compute that derivative. This makes it easy (from a user perspective) to apply gradient descent-based optimization methods; once we have specified the structure of our network in TensorFlow, we get the gradient calculations essentially for free.
- **Accelerator abstraction**: The term "accelerator" refers to any under-the-hood tool that allows a TensorFlow program to run faster. The most common example is a GPU, but this can also include custom hardware or software optimizations. The important feature from our perspective is that with the declarative programming style we do not need to worry about *how* our program will run; once we have defined the structure of the computation, we can leave it up to TensorFlow to figure out how to best take advantage of the available accelerators to run that program.

To summarize, once we are able to accurately translate a Nengo model into a TensorFlow computation graph, we are able to automatically differentiate our Nengo models and get significant improvements in simulation speed.

Using NengoDL

We begin by describing the features and usage of NengoDL from a user perspective; Section "Implementation" goes into more detail on how NengoDL is implemented. Our goal

here is not to provide a manual for NengoDL; that purpose is better served by the documentation, available at https://www.nengo.ai/nengo-dl. Instead we focus, at a relatively high level, on what users can do with NengoDL, in order to frame the upcoming implementation description.

Running a Model

The primary interface for NengoDL is through the `nengo_dl.Simulator` class. At its simplest, this is a drop-in replacement for the default `nengo.Simulator`. A very simple model would look something like

```
1  import nengo
2  import nengo_dl
3
4  with nengo.Network() as net:
5      a = nengo.Node(output=[1])
6      b = nengo.Ensemble(n_neurons=100, dimensions=1)
7      nengo.Connection(a, b)
8      p = nengo.Probe(b)
9
10 with nengo_dl.Simulator(net) as sim:
11     sim.run(1.0)
12     print(sim.data[p])
```

This creates a `Network` to hold our model (line 4), adds a `Node` that simply outputs a constant value of 1 (line 5), creates an `Ensemble` with 100 neurons and 1 represented dimension (line 6), connects the input `Node` to the `Ensemble` (line 7), and adds a probe to the output of the `Ensemble` (line 8). Note that this is all front-end code, which is completely independent of the back-end being used. We will not go into any detail on how to construct a Nengo model here; see Bekolay et al. (2014) or the documentation at https://www.nengo.ai/nengo for more information in that regard.

We specify the back-end by creating a `nengo_dl.Simulator` (line 10). We then run the simulation for one second (line 11) and print the data collected by the probe (line 12). Although we are using NengoDL here, we could switch line 10 to `nengo.Simulator` and everything else would continue to function in the same way.

However, the Nengo DL simulator also adds some new options for the user. We can use `nengo_dl.Simulator(net,device='/cpu:0')` or `nengo_dl.Simulator(net,device='/gpu:0')` to run the model on the CPU or GPU, respectively. Or we could use the `dtype=tf.float32/tf.float64` argument to control the floating point precision of the simulation.

The NengoDL simulator also has a `minibatch_size` argument, which will configure the simulation to run multiple inputs in parallel. That is,

```
with nengo_dl.Simulator(net, minibatch_size=10) as sim:
    sim.run(1.0)
    print(sim.data[p])
```

is functionally equivalent to

```
with nengo_dl.Simulator(net) as sim:
    for i in range(10):
        sim.run(1.0)
        print(sim.data[p])
        sim.reset()
```

The former will be much faster, as it takes better advantage of parallelism in the computations (see Section "Simulation Speed").

However, the output is not particularly interesting in this case, since the input is the same in all 10 instances (the constant input of 1 we specified when creating the input `Node`). To take better advantage of batched simulations we need to use another new NengoDL feature, input feeds.

Input feeds allow the user to override the default value of any input `Node` in the model. This is specified via the `data` argument of `sim.run`. This takes a dictionary mapping `Nodes` to arrays, where each array contain the values we want that node to output at each time step. For example, we could have the input node output a random number on each timestep, with different random numbers in each batch element, via

```
import numpy as np

with nengo_dl.Simulator(net, minibatch_size=10) as sim:
    sim.run(1.0, data={a: np.random.randn(10, 1000, 1)})
    print(sim.data[p])
```

Note the shape of the input array; the first dimension is the batch size (10), the second is the number of timesteps (1000, since we are running for one second with the default timestep of 0.001s), and the third is the output dimensionality of the node (1).

Again, this is not an exhaustive description of the features of the NengoDL simulator, see the documentation at https://www.nengo.ai/nengo-dl/simulator for more details and examples. We hope here to convey the basic flavour of running models with NengoDL; that is, largely the same as working with the default Nengo simulator, but with a few extra bonuses.

Training a Model

An entirely new feature of NengoDL is the ability to optimize parameters of the model via deep learning training methods. The default Nengo simulator also optimizes model parameters, but via a least squares optimization method (Eliasmith and Anderson 2003). The advantage of this method is that it is fast and flexible (e.g., it does not require the model to be differentiable). However, it can only optimize with respect to the inputs and outputs of a single layer, and is only applied to the output connection weights.

Deep learning methods allow us to jointly optimize across all the parameters in a model, allowing for more detailed fine-tuning. Note that all the standard Nengo optimization methods are also available and used in NengoDL; we are simply adding a new set of optimization methods to our modelling tool set.

These methods are accessed via the new `sim.train` method. For example, we could train our example network from above to compute the square function:[1]

```
1  import tensorflow as tf
2
3  inputs = np.random.randn(50, 1000, 1)
4  targets = inputs**2
5
6  with nengo_dl.Simulator(net, minibatch_size=10) as sim:
7      sim.train(
8          data={a: inputs,
9                p: targets},
10         optimizer=tf.train.AdamOptimizer(),
11         n_epochs=2,
12         objective={p: "mse"})
```

When performing this style of optimization we need to specify the input and target values (for each entry in the input array, we want the network to output the corresponding value from the target array). In line 3 we create a random input array; this works much the same as the `data` example above, with axes corresponding to batch size, number of timesteps, and the dimensionality of the input node, respectively. Note that the batch size is the total number of elements in the training data set; these will be split into chunks of `minibatch_size` elements during training, and the training will run through all 50 items in the dataset `n_epochs` times (line 11). We pass the inputs to the `train` function as a dictionary that maps input nodes to input arrays (line 8), as we did with `data`.

Specifying targets works in much the same way, but with respect to output `Probes` instead of input `Nodes` (lines 4 and 9). Semantically, this specifies that when the input node outputs the values from the `inputs` array, we expect to see the corresponding `targets` values at the output probe. It is then the goal of the training process to optimize the parameters in between `a` and `p` so as to make that happen.

On line 10 we specify the optimization method that should be used during the training process. TensorFlow provides a number of standard optimization methods, any of which can be used with NengoDL (or any custom optimizer that conforms to TensorFlow's optimizer API).

Note that most deep learning optimization methods rely on some version of gradient descent, which means that the network needs to be differentiable. In many neuromorphic models this is not the case (e.g., the spiking LIF neuron model is not differentiable), so applying these optimization methods restricts the kinds of models that can be studied. However, in many cases we can achieve good performance by training with a rate-based approximation of the spiking neuron model (which is differentiable), and then using those same trained parameters during inference with the spiking neuron model (Hunsberger and Eliasmith 2016). NengoDL will perform these transformations automatically (swapping between rate and spiking neurons for training and inference) for neuron types that have a differentiable rate-based approximation. This allows users to build a spiking neuron model and then optimize it with gradient-descent based training methods, with all the underlying details handled transparently by NengoDL. See Section "Spiking MNIST" for a demonstration of this idea in practice.

Finally, on line 12 we define the the objective function. This is the function applied to the output of the given probe in order to generate an error value, which the optimizer will then seek to minimize. Passing `'mse'` will use the common Mean Squared Error function, or the user can pass an arbitrary function that maps outputs and targets to an error value using TensorFlow operations.

More information on the features and usage of the `sim.train` function can be found at https://www.nengo.ai/nengo-dl/training.

Inserting TensorFlow Code

Another key feature of NengoDL is the ability to combine deep learning-style networks with Nengo neuromorphic-style networks. For example, we could use a state-of-the-art convolutional vision network to extract features from raw input images, and then connect the output of that network to a spiking neuromorphic model. This gives us the best of both worlds, allowing us to choose whichever paradigm is most appropriate for different aspects of a model.

NengoDL translates a Nengo model into a TensorFlow graph, so once that process is complete the user can, if they want, add whatever additional elements they want by working directly with that TensorFlow graph. However, the underlying TensorFlow graph can be quite complex, and it may not be obvious how to correctly insert code into that graph. In addition, such an approach splits the model definition across two qualitatively different frameworks. The key goal of NengoDL is to combine methodologies, so we would like a way to write TensorFlow code that smoothly integrates with the Nengo model definition.

This is accomplished through the `nengo_dl.TensorNode` class. `TensorNodes` are analogous to standard Nengo `Nodes`, except they integrate natively with TensorFlow code. The user writes some TensorFlow code (or reuses an existing network) that maps some input `Tensor` to an output `Tensor`. They then pass that as a

[1] Note that this code is only intended to introduce the syntax; it would not result in particularly effective training if we were to run it. Better performance would require a more complicated Nengo model, which we are trying to avoid in this description. Various full functional examples can be found at https://www.nengo.ai/nengo-dl/examples.

function to a TensorNode, which encapsulates that code as a Nengo object. The TensorNode can be added to a Nengo Network and connected to other parts of the model via Connections, the same as Ensembles and Nodes. Any values received from input Connections to the TensorNode will be passed as inputs to the TensorFlow function, and the output values of that function will be passed along any outgoing Connections. For example, we could add a TensorNode to our example network from Section "Running a Model" that applies a dense weight layer to the signal from the input node a, and sends the resulting value to the ensemble b:

```
1  with net:
2      def tensor_func(t, x):
3          return tf.layers.dense(x, 100, activation=tf.nn.relu)
4      t = nengo_dl.TensorNode(tensor_func, size_in=1)
5      nengo.Connection(a, t)
6      nengo.Connection(t, b.neurons)
```

First we define the TensorFlow function, which takes two input variables: the current simulation time, t, and the value from any incoming Connections to the TensorNode, x (line 2). Then we apply whatever TensorFlow ops we would like in order to compute the TensorNode output; in this case we are applying the tf.layers.dense function with 100 output nodes, which will create a dense weight matrix and apply the relu nonlinearity to the output (line 3). Next we create the TensorNode, passing it the function we defined and specifying the dimensionality of the function input x (line 4). Finally we connect up the inputs (from node a) and outputs (connecting directly to the neurons of ensemble b) (lines 5-6).

NengoDL also provides the the nengo_dl.tensor_layer function, an alternate interface for creating TensorNodes designed to mimic the familiar layer-based syntax common to many deep learning packages. This is simply "syntactic sugar" that combines the creation of a TensorNode and the Connection from some input object to that TensorNode in one step. For example, we could redo the above example using tensor_layer:

```
1  with net:
2      t = nengo_dl.tensor_layer(a, tf.layers.dense, units=100,
3                                activation=tf.nn.relu)
4      nengo.Connection(t, b.neurons)
```

In these simple examples we could have easily achieved the same result using normal Nengo objects. However, more complicated deep learning network architectures may not be easily expressed through the Nengo API, which is where the value of TensorNodes becomes more apparent. See https://www.nengo.ai/nengo-dl/examples/pretrained-model for a more in-depth example. More details on the features of TensorNodes can be found at https://www.nengo.ai/nengo-dl/tensor-node.

Implementation

We now provide more detail on how NengoDL is implemented. Knowledge of this infrastructure is not required to use NengoDL, but is helpful for advanced users who want to do something like add new NengoDL neuron models. The implementation also showcases some somewhat esoteric uses of TensorFlow, which may be of interest to other TensorFlow users.

TensorFlow Mapping

As discussed in Section "Background", Nengo produces a back-end representation consisting of Signals and Operators, which we need to map into a TensorFlow computation graph.

Signals

A seemingly natural solution would be to map Signals to Tensors. However, in Nengo we often have multiple Operators that all want to write to the same Signal, or parts of a Signal, and then other Operators that want to read the final result of those combined writes (rather than the output from any individual Operator). Tensors do not naturally support this style of processing; once a Tensor has been created it cannot be modified, except by creating a new Tensor. That is, if three Operators all increment the same output Signal, that will actually result in a new output Tensor each time. Thus we need to think of Signals as a more abstract representation, where writing to the same Signal may represent writing to various different Tensors.

To manage this bookkeeping we use a data structure called SignalDict. This manages the mapping between Signals and the Tensor representing the current value of that Signal. For example, imagine we have a Signal s with current value x. Suppose an Operator wants to add 1 to the value of s. This will result in a new value $y = x + 1$, which will then be stored in the SignalDict as the current value of s. Then when another Operator wants to add 2 to the value of s we look up the current value y, create a new value $z = y + 2$, and store that again as the new value of s. Thus all the Operators have the illusion that they are reading and writing to the same signals, even though that Signal may actually be represented as an interconnected graph of Tensors.

A second issue alluded to above is that in Nengo we often want to write to some subset of the elements in a Signal array. Tensors are not designed to support this kind of operation; it is not possible to modify parts of a Tensor in-place, we can only create entirely new

Tensors. It is possible to achieve similar effects using conditional TensorFlow operations, but this is slow and inefficient (for example, if we wanted to increment just one element in a 1000-dimensional vector x, we would have to create a new 1000-dimensional vector y that is just a copy of x in 999 of its elements).

Fortunately there is another TensorFlow data structure that does support in-place modification of elements: Variables. Variables are usually used to represent things like connection weights, and because we want optimizers to be able to iteratively update those weights during the training process (without generating a new copy of all the model's parameters each time) they are designed to support in-place modification. However, more generally we can just think of Variables as stateful Tensors, which is exactly what we want for our Signal values. So in practice the SignalDict will actually maintain a mapping from Signals to Variables, so every time an Operator reads or writes to (part of) a Signal, the SignalDict will direct that information to the appropriate Variable.[2] We still need the SignalDict bookkeeping because we need to make sure that reads and writes to the Variable happen in the right order. So, for example, when an Operator reads from a Variable v it reads from the version of that variable *after* any other Operators have made their updates. The SignalDict keeps track of those versions, and directs the reads and writes to the appropriate place.

Operators

With this infrastructure in place, the mapping from Nengo Operators to TensorFlow Ops is relatively straightforward. Every Operator implementation follows the same basic structure:

1. Get the current value of all the input Signals from the SignalDict
2. Apply TensorFlow ops that implement the desired computation
3. Use the SignalDict to write the results back to any output Signals

Thus we can create a small computational subgraph consisting of reads, transformations, and writes that will implement each Nengo Operator. The subgraphs for different Operators are connected via the Signals (represented as Variables) that they read and write. So as we iterate through all the Nengo Operators and add them into the TensorFlow graph, we gradually build up a complete graph of interconnected Ops that will implement a single timestep of a Nengo simulation.

The final step is to embed this single timestep within a framework that will simulate the model over time. For this we can use TensorFlow's tf.while_loop, which is a way to represent a loop using TensorFlow's declarative programming style. Generally speaking this will meet all of our needs, although some bookkeeping is needed to make sure that computations from different timesteps do not overlap incorrectly. The only concern is that tf.while_loop adds a certain amount of overhead to every iteration, which can slow down the simulation. Thus NengoDL has an option to unroll the simulation loop by explicitly building multiple timesteps into the TensorFlow computation graph. Essentially we go through the same process as above to build a single timestep, then repeat that n times so that we end up with n implementations of every Nengo Operator (all connected together in the correct order thanks to the SignalDict). We then embed that whole thing within a tf.while_loop, so that every iteration will execute n timesteps. This results in a more complicated TensorFlow graph, which increases the build time and memory usage, but can significantly improve the simulation speed. This functionality is accessed through the unroll_simulation parameter of nengo_dl.Simulator, where nengo_dl.Simulator(net, unroll_simulation=10) indicates that we should unroll the simulation as above with $n = 10$.

Graph Optimizations

Naively implementing the above process results in a functional simulator, but a slow one. The core problem is that every time an Op is executed TensorFlow has to launch a kernel, and there is a certain amount of associated overhead (especially when launching kernels on the GPU). If we have many small kernel launches, any benefits of the underlying accelerator will be lost in that overhead. So when building an efficient neural simulator in TensorFlow it is important that we try to combine operations as much as possible, so that we end up with fewer, larger kernels. For example, imagine we have 10 ElementwiseInc operations, each reading two signals and multiplying them together. Implemented individually, this would be 20 reads, 10 multiplies, and 10 writes. It would be much better to combine those operations together into one large op that would do two reads, one multiply, and one write. NengoDL automatically applies a number of these kinds of optimizations to the Operator graph in order to improve performance. All of these optimizations are transparent to the end user, neither requiring their input nor modifying the model's output (other than making it faster).

[2] Note that although we are using Variables for all the Signals, not all Signals are trainable; we still only optimize the Signals corresponding to trainable parameters of the model (e.g., connection weights and biases).

Merging

The first step is to merge `Signals`, so that we can read and write larger chunks of data. We do this by concatenating them along the first dimension, e.g. combining two 10×5 arrays into one 20×5 array. Note that this requires that the array shapes match on all dimensions beyond the first (i.e., we could not merge a 10×5 with a 10×6 array). The arrays also need to have the same type (e.g., integer versus float) and other TensorFlow meta information, such as trainability. Thus we will still end up with various different base arrays, but a much smaller number than we started with.

To track these new data structures NengoDL defines a new object called a `TensorSignal`, which stores a reference to a base array and some indices. We then translate every `Signal` into a `TensorSignal`, which indicates where the data for that `Signal` is stored and which elements in that base array (indexed along the first axis)

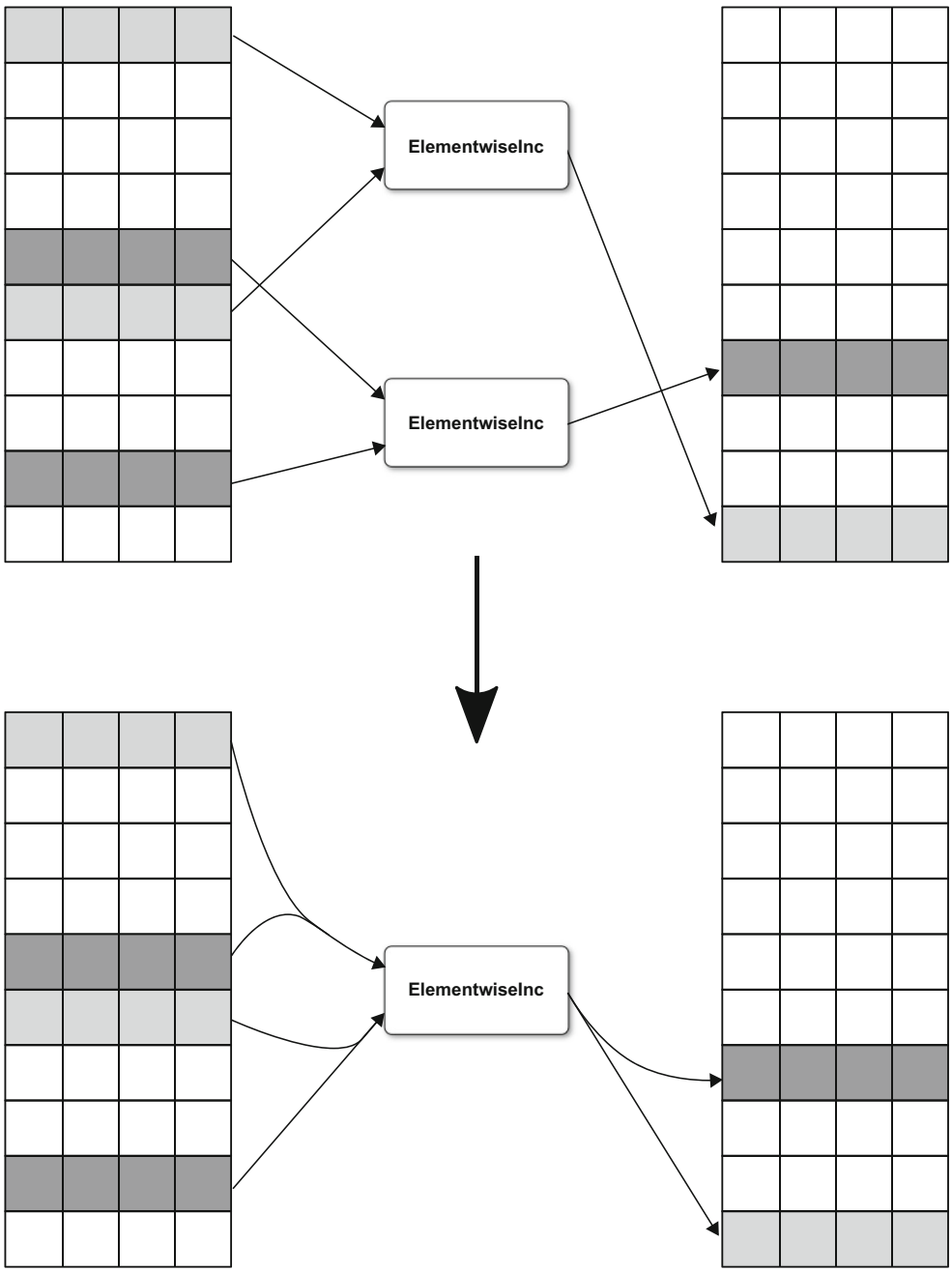

Fig. 3 Illustration of operator merging. Signals have been merged into combined base arrays. We begin with two `ElementwiseInc` operators that each read two input signals (subsets within those base arrays), multiply them together, and write the result to some output signal. To merge the operators we combine the reads, do a single multiply, and write the combined result

contain the data for that `Signal`. So whereas before an `Operator` would read/write to some `Signal`, instead it will read/write to some subset of the base array, as specified by the corresponding `TensorSignal`. Merging multiple reads into one is then as easy as combining their indices (as long as all the reads have the same base array).

The next step is to merge the operations themselves (e.g., combining the ten multiplies into one). Generally speaking, two operations are mergeable if each of their inputs and outputs are mergeable (have the same base array). For example in the `ElementwiseInc` case, once we are able to read each input as one large chunk, we can do a single `tf.multiply` to multiply them all together at once (Fig. 3). There are some additional caveats when merging more complex operators, which depend on the details of those operators, but we will not go into that here.

The other main concern for merging operators is that we cannot merge two operators if the input to one depends on the output of the other. This would introduce a circular dependency if we tried to compute those two operations simultaneously. Fortunately, Nengo already organizes all the `Operators` into a dependency graph, in order to schedule their execution (e.g., so that reads and writes to a `Signal` happen in the correct order). So we can use that graph to determine whether or not two operators depend on each other, and therefore whether or not they are mergeable.

Planning

When optimizing operator merging we also need to consider the order in which (groups of) `Operators` are executed, because the execution order can affect which operators can be merged. We can see an example of this in Fig. 4. At first two `Copy` operators and one `DotInc` operator are available to be scheduled (as they have no incoming dependencies). One might be tempted to schedule the two `Copy` operators first (blue circle), as that allows us to combine the two `Copy` ops into one. However, if we schedule the `DotInc` operator first then the third `Copy` operator will be freed of its dependency, and we will be able to merge all three `Copy` operators (green circle). This is a simple example, but the problem rapidly becomes much more complex, and efficient merging becomes more important, as the number of operators increases. Thus the goal of the planning process is to take an (unordered) list of `Operators`, and try to find an order of execution that best promotes the efficient merging of operators.

NengoDL includes a number of different planning methods, such as a greedy algorithm that simply selects the largest available group of operators to be scheduled next, or a method based on analyzing the transitive closure of the dependency graph (with some heuristic prioritization), inspired by Gosmann and Eliasmith (2017). However, the method that we found to provide the best tradeoff between plan quality and optimization time, in general, is a bounded breadth-first tree search. That is, we search through all possible execution plans up to some length n, and then schedule the group of operators corresponding to the first step in the best plan we find ("best" defined as the plan that schedules the most total operators in n steps). We then repeat this process on the remaining operators, until all operators have been scheduled. For $n = 1$ this corresponds to the greedy algorithm, and for $n = \infty$ we find the optimal plan (the plan with the shortest number of steps). A reasonable value for n depends on the complexity of the

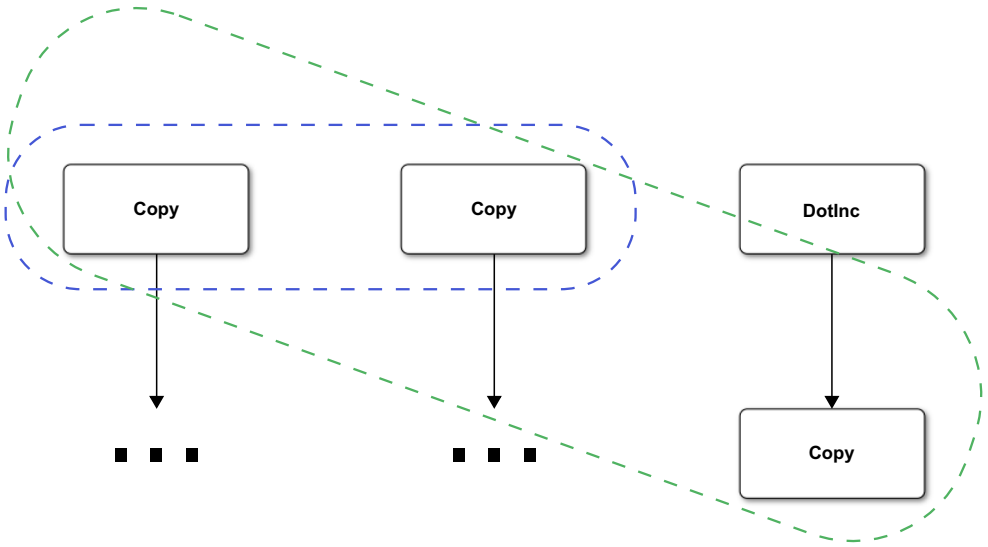

Fig. 4 Example of operator execution order planning. Arrows indicate signal read/write dependencies. By scheduling the `DotInc` operator first, we are able to more efficiently schedule the three `Copy` operators as a single group

model and the available computational budget; however, we find that $n \approx 3$ works well in practice.

Sorting

Another important optimization concern is the order in which the `Signals` are arranged in memory. Recall that `Signals` are combined into large TensorFlow `Variables`, and when we read from a `Signal` we are reading from some subset of indices within that `Variable`. However, it is faster to read from a contiguous, in-order block of indices (e.g. 5, 6, 7, 8), rather than an arbitrary set of indices (e.g., 5, 10, 12, 20, or 5, 8, 7, 6). So we want to try to arrange the `Signals` within the `Variables` such that `Signals` that are read by the same group of `Operators` are adjacent and in the same order as the `Operators` (Fig. 5).

The challenge is that we have many different groups of `Operators`, reading from possibly overlapping sets of `Signals`, such that reordering signals with respect to one group of `Operators` may break the contiguity with respect to a different set of `Operators`. We also need to consider the order of the `Operators` within a group; we can rearrange the `Operators` to promote efficient reads, rather than reordering the `Signals`. For example, if the `Signals` are arranged in the order 4, 1, 2, 3, but we move the first `Operator` in the group (which reads from the `Signal` with index 4) to the end, then this becomes an in-order, efficient read. The reason why we might want to rearrange `Operators`, rather than just changing the order of the `Signals`, is that the 4, 1, 2, 3 order may be an efficient order for a different group of `Operators` reading from an overlapping set of `Signals`. Yet another issue is that a single group of `Operators` can be reading from multiple blocks of `Signals`, meaning that if we change the order of the `Operators` we change the order of the reads within all of those `Signal` blocks (possibly changing some other block that used to be in-order to now be out-of-order).

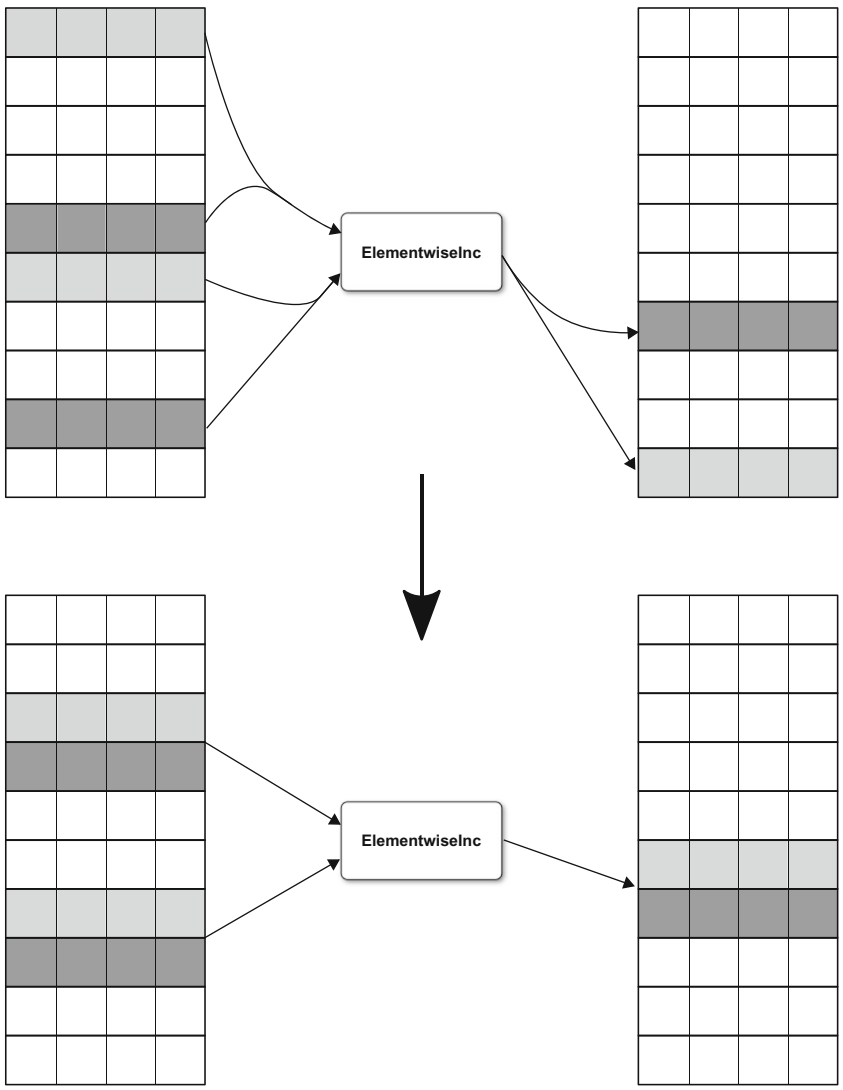

Fig. 5 Illustration of signal sorting (continuing the example from Fig. 3). By rearranging the signals into ordered, contiguous blocks we can increase the efficiency of the read operations

We end up with a complex constraint satisfaction problem, where we are trying to find the `Signal`/`Operator` ordering that will result in the best possible read performance. A perfect solution, where every read is a contiguous block, is usually not possible, nor is there an efficient algorithm for finding an optimal solution (that we know of). We arrived at a solution that uses some heuristic prioritization and an iterative settling procedure to try to find an ordering that works well in practice.

The first step is to sort the `Signals` into contiguous blocks, without worrying about order. The `Operators` are already arranged into groups by the planning process described above, so we know all the `Signals` that will be read by each group of `Operators` (which we will call a read block). We can then group all the `Signals` according to which set of read blocks they participate in, which we will call a meta-block. If all the read blocks were non-overlapping, then every meta-block would contain a single read block; however, this is rarely the case. Since the order of `Signals` within a meta-block does not matter (yet), we can reformulate the problem as sorting the meta-blocks into an order that ensures the underlying read blocks are as contiguous as possible. Again, an ideal sorting is usually not possible, so in general we prioritize the contiguity of larger blocks (as they are the more expensive read operations).

Algorithm 1: Meta-block sorting algorithm.

initialize list of all meta-blocks M
initialize sorted list S (empty)
set active read block a to be the largest read block
set active meta-block c to be any meta-block containing a
while M *is not empty* **do**
  $X \leftarrow \{m \in M \mid a \in m\}$
  if $X = \emptyset$ **then**
    $X \leftarrow M$
    $a \leftarrow$ largest read block in c
  $Y \leftarrow \{x \in X \mid c \subseteq x\}$
  if $Y = \emptyset$ **then**
    $Y \leftarrow X$
  $Z \leftarrow \{y \in Y \mid |y \oplus c| = \min_{n \in Y} |n \oplus c|\}$
  while $—Z— > 1$ **do**
    $m \leftarrow$ the next largest read block in c
    $Z \leftarrow \{z \in Z \mid m \in z\}$
  remove the remaining $z \in Z$ from M and append it to S
  $c \leftarrow z$

Our meta-block sorting algorithm is shown in Algorithm 1. Broadly speaking, this algorithm tries to find the next meta-block that best matches the last meta block we selected. Matching is determined by narrowing down the set of remaining meta-blocks according to increasingly strict criteria: 1) any meta-blocks that contain the active read block (so that we know that at least the active block will end up being contiguous); 2) any meta-blocks that contain all the elements in the last meta-block; 3) the meta-blocks with minimal Hamming distance to the last meta-block; and 4) the meta-block that contains the largest read blocks in the last meta-block.

After the meta-block sorting process, the `Signals` are arranged into (semi-) contiguous blocks of indices within the base `Variables`. We then want to sort the `Operators` and `Signals` within each meta-block so that the indices are in increasing order. Recall that because our `Signal` blocks overlap, and because a group of `Operators` can read from multiple `Signal` blocks, there is unlikely to be an ordering that satisfies all the constraints. Again we prioritize larger read blocks.

Algorithm 2: Signal/Operator sorting algorithm.

sort the list of read blocks B by increasing order of size
for $i = 1 \rightarrow n$ **do**
  for $b \in B$ **do**
    $O \leftarrow$ the set of `Operators` associated with b
    sort O to match the order of the `Signals` in b
    $C \leftarrow$ the set of read blocks associated with O
    for $c \in C$ **do**
      sort the signals in c to match the order of O
  if the order of the `Signals`/`Operators` did not change, terminate early

Algorithm 2 cycles between two steps: 1) sort the `Operators` to match the order of a given `Signal` block b, and 2) sort all the `Signal` blocks read by that group of `Operators` to match the new order of the `Operators` (this sorting is restricted such that it cannot change the meta-block order). The basic idea is that after step 1, we know that b will be contiguous and in-order (assuming that the meta-block sorting algorithm was able to make b contiguous). However, imagine that our `Operator` group also reads from another block c. Rearranging the order of the `Operators` may have put c out of order, so we fix that in step 2.

Note, however, that there may be some other group of `Operators` that also reads from b or c (or some overlapping set). Thus the sorting we just performed might have broken an earlier ordering we established for that other group of `Operators`. That is why we iterate over the read blocks in increasing order of size; we know that later sorting will only break the ordering of earlier, and therefore smaller, blocks. However, it is possible that after the `Signals` are reordered by a larger block (step 2), the `Operators` in a smaller block could be reordered to match that new `Signal` order (step 1). That is why we perform multiple passes over

the read blocks, to allow the smaller blocks to settle into orderings that are consistent with the larger blocks.

Simplification

Another optimization we perform is to simplify the `Operator` graph by checking for certain special case combinations of `Operators`. For example, we can change $y \mathrel{+}= x * 1$ to $y \mathrel{+}= x$ in order to save a multiplication, or if there is a `Copy` operation that moves data from x to y, but the value of x never changes, we can change that to a `Reset` operator that directly sets the value of y to that constant value (saving a read). These optimizations do not have a large impact relative to the merging and sorting, but they are also relatively simple and quick to perform.

Results

There are two areas we will focus on in the results: the simulation speed of NengoDL, and some practical demonstrations of using NengoDL to construct and optimize a neural model. The code needed to reproduce any of the results presented here is available at https://github.com/nengo/nengo-dl/tree/master/docs/whitepaper.

Simulation Speed

We compare the simulation speed of NengoDL to the default Nengo simulator (which is CPU only) as well as NengoOCL (a simulator that runs on the GPU using custom OpenCL kernels). All results are collected using an Intel Xeon E5-1650 3.5GHz CPU and an Nvidia GeForce GTX Titan X GPU (in the case of NengoDL and NengoOCL).

Figure 6 shows the relative speed of the simulators on four different benchmark models. The purpose of the models is not particularly important; they were simply chosen to showcase a range of different models with varying complexities:

- **integrator**: A single ensemble of recurrently connected neurons (a common component used to implement a memory system in Nengo)
- **cconv**: A network implementing the circular convolution of two input vectors (commonly used in Nengo Semantic Pointer Architecture models)
- **basal_ganglia**: A model of basal ganglia circuitry, commonly used to perform action selection
- **pes**: An ensemble of neurons with output weights that are updated as the simulation runs, using the Prescribed Error Sensitivity learning rule (MacNeil and Eliasmith 2011)

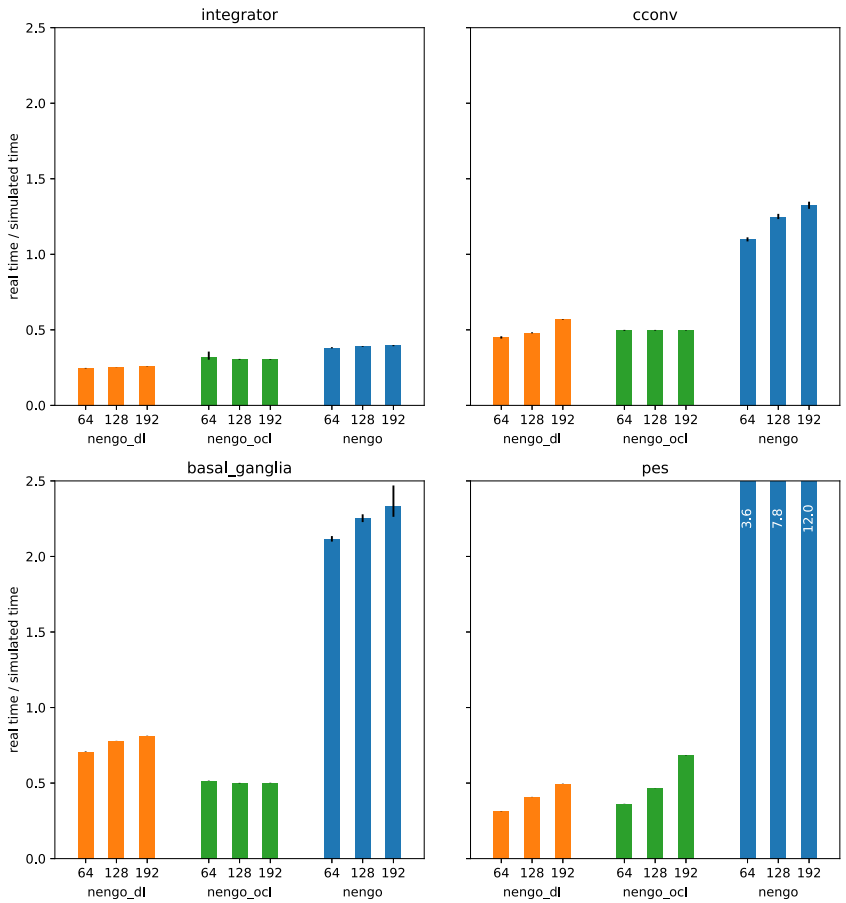

Fig. 6 Comparing simulation speed of NengoDL versus NengoOCL versus Nengo on various benchmark models. Error bars indicate 95% confidence intervals on the mean over 5 runs. We show scaling with respect to the represented dimensionality (64, 128, 192)

In order to get a better picture how the different backends compare, we show how the performance scales as we change a parameter of these model, the represented dimensionality. This increases the complexity of the model in a number of ways; for example, it increases the number of neuron ensembles, the dimensionality of the signals being transmitted throughout the model, and the number of parameters (through the size of encoder/decoder matrices).

Overall we can see that the GPU-based simulators (NengoDL and NengoOCL) offer significant performance improvements, with NengoOCL or NengoDL offering the best performance on different benchmarks.

That being said, we can see an important advantage of NengoDL in Fig. 7. In this case we are running the same benchmarks, but we are running each model ten times. With Nengo and NengoOCL this involves serially running the model ten times in a row, which, unsurprisingly, takes about ten times as long. However, NengoDL allows models to be run with batched inputs, so we can simulate the model once with ten different inputs in parallel. This scales much better as we increase the batch size, thanks to the parallelism of the computations. Thus if a modeller wants to test their model with a range of different inputs, NengoDL will probably offer the best performance.

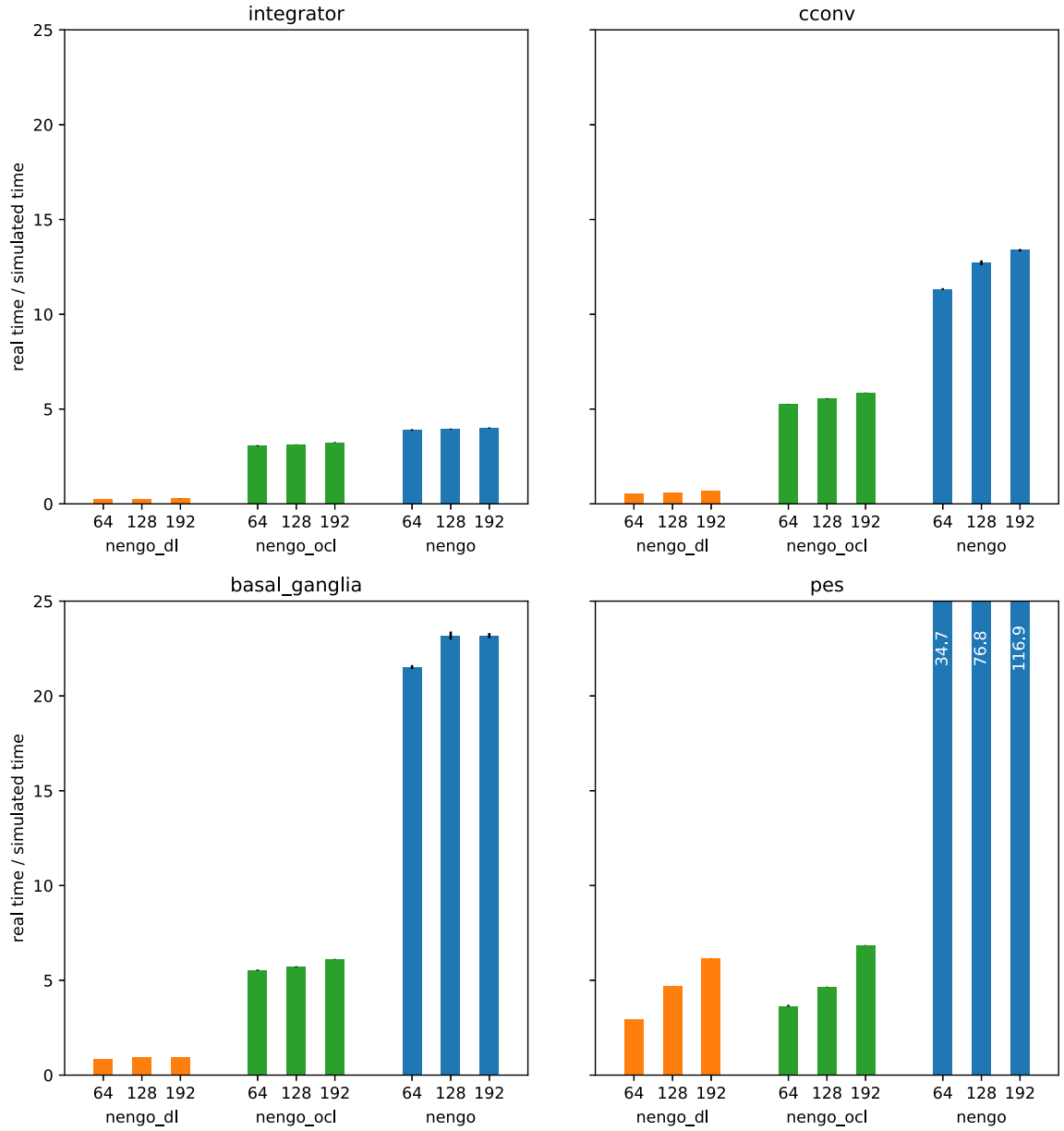

Fig. 7 Comparing simulation speed of NengoDL versus NengoOCL versus Nengo on various benchmark models with 10 batched inputs. We show scaling with respect to the represented dimensionality (64, 128, 192). Note that we do not get the NengoDL batching benefits on the PES benchmark, because that network applies an online learning rule to the weights (meaning that we need a separate weight matrix for each batch element)

Finally, it is interesting to explore the effect of the various graph optimization steps described in Section "Graph Optimizations". Figure 8 shows the speed of NengoDL when simulating the Spaun model (an updated version of Eliasmith et al. (2012), available at https://github.com/xchoo/spaun2.0) with 128-dimensional vectors, consisting of 1.2 million neurons split amongst 21k ensembles with 91k connections. Spaun was chosen because the complexity of this model provides a good stress test for the graph optimization methods. We can see that each type of optimization provides incremental improvements to the simulation speed. Note that in the "planning" case we are comparing the tree planner to the greedy planner (see Section "Planning"), rather than the presence and absence of planning. That is, in all cases we are performing operator merging. If we do not perform any merging then the simulation is extremely slow (after one hour the simulation had still not finished initializing the TensorFlow graph).

Model Examples

Simulation speed is an important aspect of NengoDL, but equally important are the novel features NengoDL provides that are not available in any other Nengo simulator. Specifically, NengoDL includes the ability to: a) insert Tensor Flow components, such as convolutional layers, into a Nengo model; b) convert rate-based deep learning networks into spiking versions; and c) optimize the parameters of a Nengo model using deep learning training methods. In this section we will present some basic examples illustrating these features and the advantages they provide.

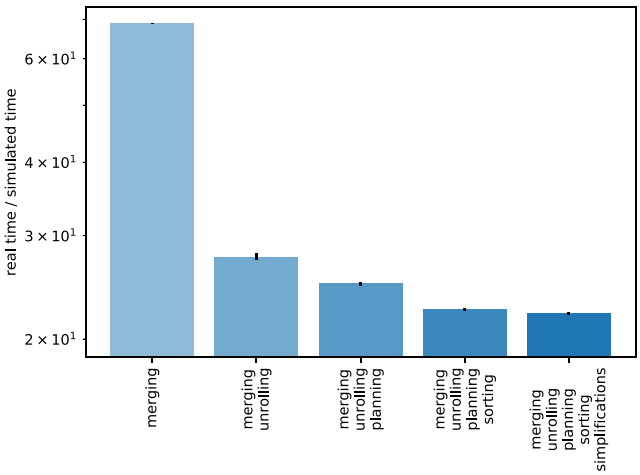

Fig. 8 Impact of the various NengoDL graph optimization methods on the simulation speed of the Spaun model. Note that the speed is being plotted on a logarithmic scale. **Merging**: multiple operators of the same type are combined into a single, larger operator of that type (using a greedy planner). Without this step the simulation speed is extremely slow, so we include it in all cases. **Unrolling**: the simulation loop is unrolled within the TensorFlow computation graph. **Planning**: A more advanced planning algorithm is used (the tree planner) to promote better operator merging. **Sorting**: Signals are sorted to promote more efficient reads. **Simplifications**: Unnecessary `Operations` are removed from the Nengo build graph. More details on all the optimization methods can be found in Section "Graph Optimizations"

Spiking MNIST

In this model we use the `TensorNode/tensor_layer` syntax to create a simple convolutional network in Nengo, consisting of three convolutional layers, two average pooling layers, and a dense linear readout. We use the Leaky Integrate and Fire neuron model, which has both a rate and spike-based implementation. As described in Section "Training a Model", we use NengoDL to automatically swap between the differentiable rate implementation during training and the spiking model during testing/inference. We also take advantage of NengoDL's ability to smoothly combine TensorFlow and Nengo models; we use `TensorNodes` to implement convolutional and pooling layers using TensorFlow, combined with standard Nengo `Ensembles` to implement the neural nonlinearities and `Connections` to link layers together.

We train the model on the deep learning "hello world" task, MNIST digit classification (the model receives an image of a hand-written digit as input and must classify that digit 0–9). This kind of vision system has been integrated with cognitive models in, e.g., Eliasmith et al. (2012), where the model used an MNIST vision system combined with working memory, inductive reasoning, and motor control capabilities to perform a range of different cognitive tasks. However, in that case the vision system was trained separately using a standard deep learning package, and then imported into Nengo. Here we show that using the new features of NengoDL we can directly build and train these deep learning style networks within the Nengo framework, making it much simpler to construct integrated, hybrid cognitive model as in Eliasmith et al. (2012).

After training, the model achieves 99.05–99.09% classification accuracy (95% confidence intervals), which is the performance we would expect for MNIST. However, one of the important features of Nengo, which is retained in NengoDL, is the ability to smoothly switch between rate and spiking neuron models. After training the model using the rate-based implementation of LIF neurons, we can then run the model using spiking LIF neurons (using the same trained weights). This spiking version of the network achieves 98.41–98.87% classification accuracy, only a small decrease from the rate version. Spiking deep learning is an interesting and active research field (Hunsberger and Eliasmith 2016; Lee et al. 2016), and one which NengoDL is naturally situated to support.

Memory Storage and Retrieval

In the second example we want to explore the application of the NengoDL training functionality to a more cognitive/neuromorphic style of model, rather than a standard deep learning vision network. We construct a model using Nengo's Semantic Pointer Architecture (SPA), which uses high-dimensional vectors, encoded in neural activity, to represent structured symbolic information. We apply the model to a memory retrieval task: the network is given a sequence of attribute-value pairs as input (e.g. $(colour : red)$, $(shape : circle)$, $(texture : smooth)$) that it must dynamically store in memory using neural activities. At a later point the network is prompted with one of the attributes (e.g., $shape$), and must respond with the corresponding value (e.g., $circle$). Note that we want to perform this task for arbitrary input sequences, so the solution cannot be directly built into the connection weights (i.e., we cannot just train the network to output $circle$ when it gets the cue $shape$). We want the network to learn the abstract functions required to store and retrieve generic items from memory.

The network architecture consists of a circular convolution network (to combine the attribute-value pairs into a linked representation), a recurrently connected ensemble of neurons to implement the memory, and a final circular convolution network to extract the cued attribute from the memory. In this example we use rectified linear neurons. We can construct this model, without NengoDL, by using the least-squares-based optimization methods standard in Nengo. The advantage of these methods is that they are fast and do not rely on gradient descent (and therefore do not require the model to be differentiable). However, these methods can only optimize the output weights of one ensemble/layer at a time. This means that each layer in the above model is optimized independently, and there are many parameters (e.g., input weights, gains, and biases) that are not optimized (they are typically chosen from some random distribution). By using NengoDL we can begin with the standard Nengo optimized model, and then apply the deep learning optimization on top of that. This allows us to jointly optimize across the layers of the model, and fine tune all the parameters in the model as well as the output weights.

We train the model by generating a randomized set of training data, with different sequences of attribute-value pairs and different vector vocabularies. For each of these inputs we can specify what the correct model output would be. We can then use TensorFlow's gradient-descent based optimizers (in this case, RMSProp; Tieleman and Hinton 2012) to optimize all the parameters of the model with respect to those inputs and target outputs. The details of the training hyperparameters can be found in the code at https://github.com/nengo/nengo-dl/tree/master/docs/whitepaper.

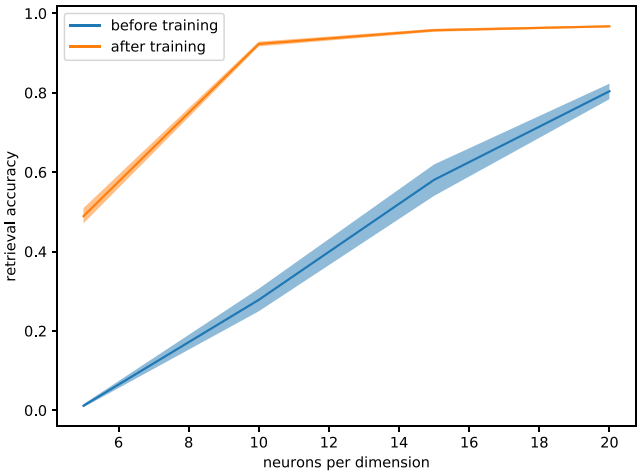

Fig. 9 Retrieval accuracy on the memory task, before and after training is applied. Showing 95% confidence intervals (for different random initializations)

The effect of the training is shown in Fig. 9. This figure shows the retrieval accuracy of the model (on a separate set of randomly generated test data), which is computed by comparing the output of the model (which should be the value of the cued attribute) to all the vectors in the vocabulary (e.g., $colour$, $shape$, red, $circle$, etc.). If the output of the model is most similar to the correct answer, then that is a successful retrieval. We can see that the performance of the model is significantly improved after applying the NengoDL training. In particular we can see the impact of the training for smaller numbers of neurons. This makes sense given the random initialization of many of the neural parameters under the standard Nengo methods. For larger numbers of neurons that random initialization is likely to give a good-enough coverage of the parameter space, but for smaller numbers it is more important that those parameters be fine-tuned for the problem. In other words, one important advantage of the NengoDL optimization features are that they allow us to take better advantage of limited neural resources.

Conclusion

The goal of NengoDL is to provide a tool that unites deep learning and neuromorphic modelling methods. It combines the robust modelling API of Nengo with the speed and optimization methods of TensorFlow. This allows the modeller to build complex cognitive/neuromorphic models, combine them with deep learning elements (such as convolutional layers), simulate them efficiently, and optimize their parameters using modern deep learning training methods.

In this paper we have introduced the features and some interesting implementation aspects of NengoDL. Those interested in learning more or using NengoDL in their own work can find much more information in the online documentation at https://www.nengo.ai/nengo-dl. This includes installation instructions, details on all the novel features of NengoDL and how to access them, as well as examples illustrating various different styles of models. All the source code can be found at https://github.com/nengo/nengo-dl. There is also a forum at https://forum.nengo.ai where users can get help with specific questions. Finally, NengoDL is under active development; feel free to suggest features on the forum or at https://github.com/nengo/nengo-dl/issues so that we can continue to improve this tool for the modelling community.

Information Sharing Statement

NengoDL is available online at https://github.com/nengo/nengo-dl, and all of the code needed to reproduce the results in this paper is available at https://github.com/nengo/nengo-dl/tree/master/docs/whitepaper.

Acknowledgments This work was supported by Applied Brain Research, Inc. and ONR MURI N00014-16-1-2832.

Compliance with Ethical Standards

Conflict of interests DR is an employee/shareholder of Applied Brain Research, Inc., which owns the Nengo software package (including NengoDL). Nengo is free for research/personal/non-commercial use, but ABR charges a license fee for commercial use.

References

Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., Kudlur, M., Levenberg, J., Monga, R., Moore, S., Murray, D.G., Steiner, B., Tucker, P., Vasudevan, V., Warden, P., Wicke, M., Yu, Y., Zheng, X., Brain, G., Osdi, I., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., Kudlur, M., Levenberg, J., Monga, R., Moore, S., Murray, D.G., Steiner, B., Tucker, P., Vasudevan, V., Warden, P., Wicke, M., Yu, Y., Zheng, X. (2016). TensorFlow: a system for large-scale machine learning. In *Proceedings of the 12th USENIX symposium on operating systems design* (pp. 265–283). Savannah, GA, USA.

Bekolay, T., Bergstra, J., Hunsberger, E., DeWolf, T., Stewart, T.C., Rasmussen, D., Choo, X., Voelker, A.R., Eliasmith, C. (2014). Nengo: a Python tool for building large-scale functional brain models. *Frontiers in Neuroinformatics*, *7*(48), 1–13.

Benjamin, B.V., Gao, P., Mcquinn, E., Choudhary, S., Chandrasekaran, A.R., Bussat, J.-M., Alvarez-Icaza, R., Arthur, J.V., Merolla, P.A., Boahen, K. (2014). Neurogrid: a mixed-analog-digital multichip system for large-scale neural simulations. In *Proceedings of the IEEE*, Vol. 102(5).

Bobier, B., Stewart, T.C., Eliasmith, C. (2014). A unifying mechanistic model of selective attention in spiking neurons. *PLos Computational Biology*, 10(6).

Choo, X., & Eliasmith, C. (2010). A spiking neuron model of serial-order recall. In Cattrambone, R., & Ohlsson, S. (Eds.) *Proceedings of the 32nd annual conference of the cognitive science society*. Portland. Cognitive Science Society.

Collobert, R., Kavukcuoglu, K., Farabet, C. (2011). Torch7: a Matlab-like environment for machine learning. In *Biglearn, NIPS workshop* (pp. 1–6).

Davies, M., Srinivasa, N., Lin, T.-H., Chinya, G., Cao, Y., Choday, S.H., Dimou, G., Joshi, P., Imam, N., Jain, S., Liao, Y., Lin, C.-K., Lines, A., Liu, R., Mathaikutty, D., McCoy, S., Paul, A., Tse, J., Venkataramanan, G., Weng, Y.-H., Wild, A., Yang, Y. (2018). Loihi: a neuromorphic manycore processor with on-chip learning. *IEEE Micro*, 38(1).

Davison, A.P., Brüderle, D., Eppler, J., Kremkow, J., Muller, E., Pecevski, D., Perrinet, L., Yger, P. (2009). PyNN: a common interface for neuronal network simulators. *Frontiers in Neuroinformatics*, *2*, 1–10.

DeWolf, T., Stewart, T.C., Slotine, J.-J., Eliasmith, C. (2016). A spiking neural model of adaptive arm control. *Proceedings of the Royal Society: Biological Sciences, 283*(1843).

Eliasmith, C., & Anderson, C. (2003). *Neural engineering: Computation, representation, and dynamics in neurobiological systems*. Cambridge: MIT Press.

Eliasmith, C., Stewart, T.C., Choo, X., Bekolay, T., DeWolf, T., Tang, Y., Rasmussen, D. (2012). A large-scale model of the functioning brain. *Science*, *338*(6111), 1202–1205.

Esser, S.K., Appuswamy, R., Merolla, P.A., Arthur, J.V., Modha, D.S. (2015). Backpropagation for energy-efficient neuromorphic computing. In *Advances in neural information processing systems* (pp. 1–9).

Gewaltig, M.-O., & Diesmann, M. (2007). NEST (NEUral Simulation Tool). *Scholarpedia*, *2*, 1430.

Gosmann, J., & Eliasmith, C. (2017). Automatic optimization of the computation graph in the Nengo neural network simulator. *Frontiers in Neuroinformatics*, *11*, 1–11.

Hines, M.L., & Carnevale, N.T. (1997). The NEURON simulation environment. *Neural Computation*, *9*(6), 1179–1209.

Hunsberger, E., & Eliasmith, C. (2016). Training spiking deep networks for neuromorphic hardware. arXiv:1611.05141 (v1):1–10.

Jia, Y., Shelhamer, E., Donahue, J., Karayev, S., Long, J., Girshick, R., Guadarrama, S., Darrell, T. (2014). Caffe: Convolutional architecture for fast feature embedding. arXiv:1408.5093 (v1).

Kay, K.N. (2017). Principles for models of neural information processing. *NeuroImage*, 1–20.

Khan, M., Lester, D., Plana, L. (2008). SpiNNaker: mapping neural networks onto a massively-parallel chip multiprocessor. In *IEEE joint conference on neural networks* (pp. 2849–2856).

Kriegeskorte, N. (2015). Deep neural networks : a new framework for modeling biological vision and brain information processing. *Annual Review of Vision Science*, *1*, 417–446.

Lee, J.H., Delbruck, T., Pfeiffer, M. (2016). Training deep spiking neural networks using backpropagation. *Frontiers in Neuroscience*, 10.

MacNeil, D., & Eliasmith, C. (2011). Fine-tuning and the stability of recurrent neural networks. *PloS ONE*, *6*(9), e22885.

Rasmussen, D., & Eliasmith, C. (2014). A spiking neural model applied to the study of human performance and cognitive decline on Raven's advanced progressive matrices. *Intelligence*, *42*, 53–82.

Rasmussen, D., Voelker, A., Eliasmith, C. (2017). A neural model of hierarchical reinforcement learning. *PLoS ONE*, *12*(7), 1–39.

Rueckauer, B., Lungu, I.-A., Hu, Y., Pfeiffer, M., Liu, S.-C. (2017). Conversion of continuous-valued deep networks to efficient

event-driven networks for image classification. *Frontiers in Neuroscience*, *11*, 1–12.
Stewart, T.C., Bekolay, T., Eliasmith, C. (2012). Learning to select actions with spiking neurons in the Basal Ganglia. *Frontiers in Decision Neuroscience*, *6*, 2.
Stimberg, M., Goodman, D.F.M., Benichoux, V., Brette, R. (2013). Brian 2 - the second coming : spiking neural network simulation in Python with code generation. In *Twenty second annual computational neuroscience meeting* (pp. 1–2).
Team, T.D. (2016). Theano: a Python framework for fast computation of mathematical expressions. arXiv:1605.02688 (v1):1–19.
Tieleman, T., & Hinton, G.E. (2012). Lecture 6.5-Rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural networks for machine learning*, *4*(2), 26–31.
Yamins, D.L.K., & DiCarlo, J.J. (2016). Using goal-driven deep learning models to understand sensory cortex. *Nature Neuroscience*, 19(3).