Non-Linear Motor Control by Local Learning in Spiking Neural Networks

Aditya Gilra [1 2] **Wulfram Gerstner** [1]

Abstract

Learning weights in a spiking neural network with hidden neurons, using local, stable and online rules, to control non-linear body dynamics is an open problem. Here, we employ a supervised scheme, Feedback-based Online Local Learning Of Weights (FOLLOW), to train a heterogeneous network of spiking neurons with hidden layers, to control a two-link arm so as to reproduce a desired state trajectory. We show that the network learns an inverse model of the non-linear dynamics, i.e. it infers from state trajectory as input to the network, the continuous-time command that produced the trajectory. Connection weights are adjusted via a local plasticity rule that involves pre-synaptic firing and post-synaptic feedback of the error in the inferred command. We propose a network architecture, termed differential feedforward, and show that it gives a lower test error than other feedforward and recurrent architectures. We demonstrate the performance of the inverse model to control a two-link arm along a desired trajectory.

1. Introduction

Motor control requires building internal models of the muscles-body system (Conant & Ashby, 1970; Pouget & Snyder, 2000; Wolpert & Ghahramani, 2000; Lalazar & Vaadia, 2008). The brain possibly uses random movements during pre-natal (Khazipov et al., 2004) and post-natal development (Petersson et al., 2003) termed motor babbling (Meltzoff & Moore, 1997; Petersson et al., 2003) to learn internal models of the muscles-body dynamical system (Lalazar & Vaadia, 2008; Wong et al., 2012; Sarlegna & Sainburg, 2009; Dadarlat et al., 2015). Forward models use neural motor commands to predict body movement, while inverse models take a desired trajectory and generate the neural motor commands that would produce it. Here we focus on the inverse model for motor control: given a desired state trajectory $\vec{x}^D(t)$ for the dynamical system formed by the muscles and body, networks of spiking neurons in the brain must learn to produce the time-dependent neural control input that activates the muscles to produce the desired movement. Abstracting the muscles-body dynamics as

$$d\vec{x}/dt = \vec{f}(\vec{x}, \vec{u}), \tag{1}$$

we require the network to generate control input $\vec{u}(t)$, given desired state trajectory $\vec{x}^D(t)$, that makes the state evolve as $\vec{x}(t) \approx \vec{x}^D(t)$. Our aim is to train a network of spiking neurons with hidden layers, via a local synaptic plasticity rule, to learn the inverse model of the muscle-body dynamics.

In networks of continous-valued or spiking neurons, training the weights of hidden neurons with a local learning rule is considered difficult due to the credit assignment problem (Bengio et al., 1994; Hochreiter et al., 2001; Abbott et al., 2016). Supervised learning in neural networks, including for motor control, has typically been accomplished by backpropagation of error (Rumelhart et al., 1986; Williams & Zipser, 1989) which is non-local, reservoir computing (Jaeger, 2001; Maass et al., 2002) which trains only output weights, FORCE learning (Sussillo & Abbott, 2009) which requires weight changes faster than the requisite dynamics, and adaptive control theory based schemes (Narendra & Parthasarathy, 1990; Sanner & Slotine, 1992; MacNeil & Eliasmith, 2011; Bourdoukan & Denève, 2015; DeWolf et al., 2016) which use non-local rules or if local learn only linear systems. Recently, local learning schemes have emerged for learning forward models of non-linear dynamical systems (Gilra & Gerstner, 2017; Alemi et al., 2017).

We employ the scheme called Feedback-based Online Local Learning Of Weights (FOLLOW) (Gilra & Gerstner, 2017), which draws upon adaptive control theory (Morse, 1980; Narendra et al., 1980; Narendra & Annaswamy, 1989; Ioannou & Sun, 2012) and function and dynamics approximation theory (Funahashi, 1989; Hornik et al., 1989; Girosi & Poggio, 1990; Eliasmith & Anderson, 2004). FOLLOW

[1]School of Computer and Communication Sciences, and Brain-Mind Institute, School of Life Sciences, École Polytechnique Fédérale de Lausanne, 1015 Lausanne EPFL, Switzerland. [2]Now at: Institute for Genetics, University of Bonn, Kirschallee 1, 53115 Bonn, Germany. Correspondence to: Aditya Gilra <agilra@uni-bonn.de>.

Proceedings of the 35^{th} International Conference on Machine Learning, Stockholm, Sweden, PMLR 80, 2018.

is a synaptically local and provably stable and convergent alternative for training the feedforward and recurrent weights of hidden spiking neurons in a network, that was employed to predict non-linear dynamics (Gilra & Gerstner, 2017). Gilra & Gerstner (2017) employed the FOLLOW scheme to enable a recurrent network to learn a forward model of arm dynamics, i.e. to predict arm state given neural control input. Here, we aim at solving the full motor control problem, and also learn the inverse model to infer control input given an arm trajectory. Learning either the forward or the inverse model requires solving an 'inverse problem' i.e. we must infer the model, summarized in the weights of the network, given the data namely the input $\vec{u}(t)$ and output $\vec{x}(t)$ of the dynamical system. However, the roles of input and output are reversed, and those of delays are unique, compared to the forward model. We therefore propose a novel differential feedforward architecture for learning the inverse model, rather than the recurrent architecture used for the forward model. Finally, we embed the inverse model in a closed loop to make the arm replicate a desired trajectory.

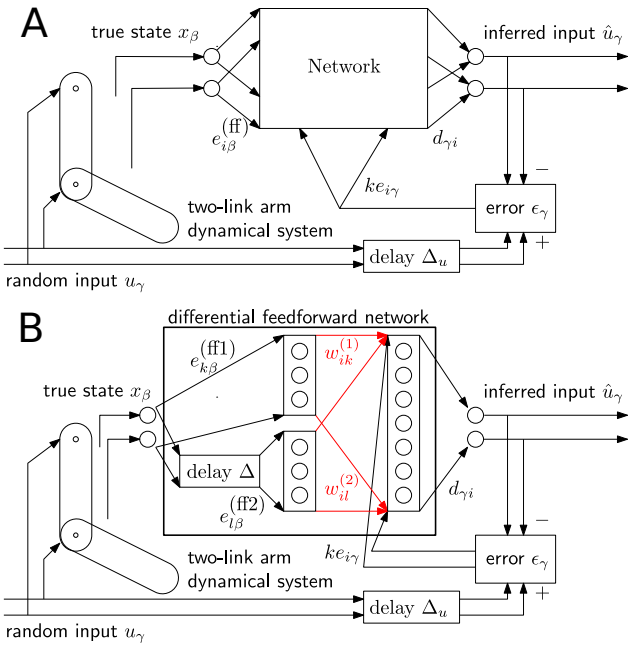

Figure 1. **Network configuration for learning the inverse model. A**. During learning, random motor commands u_γ cause arbitrary movements of the arm (motor babbling). The observed 4-dimensional state x_β of the arm (from visual and proprioceptive feedback) is given as input to the network via fixed random weights $e^{(\mathrm{ff})}_{i\beta}$. The inferred 2-dimensional motor command $\hat{u}_\gamma$ is linearly decoded from the filtered output spike trains of the network via fixed weights $d_{\gamma i}$. A copy of the actual motor command (input to the arm) is used, after a delay of Δ_u, to compute the error in the inferred motor command, i.e. error $\epsilon_\gamma(t) = u_\gamma(t - \Delta_u) - \hat{u}_\gamma(t)$. This error is fed back into the network with fixed random encoding weights $ke_{i\gamma}$. **B.** Learning inverse model with a differential feedforward network. The learning paradigm and configuration are as in panel A. The input is sent undelayed to a set of neurons ff1 via encoding weights $e^{(\mathrm{ff1})}_{k\beta}$ and delayed by Δ to another set ff2 via encoding weights $e^{(\mathrm{ff2})}_{l\beta}$. These two sets feed into the next layer with plastic weights $w^{(1)}_{ik}$ and $w^{(2)}_{il}$ respectively (red). The fedback error and the filtered pre-synaptic spike train are used to update these feedforward weights (red). In both panels, twin lines in the connection arrows denote multi-dimensional signals, but their number is not representative of the dimensionality.

2. Network Architecture and FOLLOW Scheme to Learn Inverse Model

We use a network of leaky-integrate-and-fire (LIF) neurons with different biases to learn the inverse model of an arm (Fig. 1). The arm, adapted from (Li, 2006), is modelled as a two-link pendulum moving in a vertical plane under gravity, with friction at the joints, 0° as the equilibrium downwards position, and soft bounds on motion beyond $\pm 90^\circ$. As shown in Figure 1A, random 2-dimensional torque input $u_\gamma(t), \gamma = 1, 2$ is provided to the arm to generate random state trajectories $\vec{x}(t)$ analogous to motor babbling. The 4-dimensional state of the arm (2 joint angles and 2 joint velocities), denoted $x_\beta, \beta = 1, ..., 4$, is fed as input to the network via fixed, random weights. The network must learn to infer $\vec{u}(t)$ that generated the arm state trajectory $\vec{x}(t)$.

We chose the network in Figure 1B, termed the differential feedforward network, from a variety of network architectures (see section 4), to learn the inverse model. We have $K = L = 3000$ neurons each, in the two sets in the first hidden layer, indexed by $k = 1 \ldots K$, and $l = 1 \ldots L$, followed by $N = 5000$ neurons in the next hidden layer indexed by $i = 1 \ldots N$. The state vector $\vec{x}$ is fed to the two differential feedforward sets, undelayed to one and delayed to the other by an interval Δ, via fixed random weights $e^{(\mathrm{ff1})}_{k\beta}$ and $e^{(\mathrm{ff2})}_{l\beta}$ respectively. The current into a neuron k in the undelayed set is given by

$$J_k = \sum_\beta e^{\mathrm{ff1}}_{k\beta} x_\beta(t) + b_k, \tag{2}$$

where b_k is a fixed, random neuron-specific bias. Similarly, the current into a neuron l in the delayed set is given by

$$J_l = \sum_\beta e^{\mathrm{ff2}}_{l\beta} x_\beta(t - \Delta) + b_l, \tag{3}$$

where b_l is also a fixed, random neuron-specific bias.

The output $\vec{\hat{u}}(t)$ of the network is a linearly weighted sum of filtered spike trains $S_i(t)$ of the second hidden layer

neurons:

$$\hat{u}_\gamma(t) = \sum_i d_{\gamma i} \int_{-\infty}^{t} S_i(s)\kappa(t-s)ds \equiv \sum_i d_{\gamma i}(S_i * \kappa)(t), \quad (4)$$

with readout weights $d_{\gamma i}$ and filtering kernel $\kappa(t)$ a decaying exponential with time constant $\tau_s = 20$ ms. This readout $\vec{\hat{u}}(t)$ of the network is compared to a time-delayed version of the command input, with delay Δ_u ms, to causally infer the past command. Borrowing from adaptive control theory (Narendra & Annaswamy, 1989; Ioannou & Sun, 2012), the error $\epsilon_\gamma \equiv u_\gamma(t-\Delta_u) - \hat{u}_\gamma(t)$ is fed back as neural currents to the second hidden layer neurons with fixed random feedback weights $e_{i\gamma}$ multiplied by a gain $k = 10$. The total current into neuron i in the second hidden layer is a sum of the two feedforward currents and the error current:

$$J_i = \sum_k w^{(1)}_{ik}(S^{\text{ff1}}_k * \kappa)(t) + \sum_l w^{(2)}_{il}(S^{\text{ff2}}_l * \kappa)(t) + \sum_\gamma k e_{i\gamma}(\epsilon_\gamma * \kappa)(t) + b_i, \quad (5)$$

where b_i is a neuron-specific bias, and S^{ff1}_k and S^{ff2}_l are the spike trains of the two sets of neurons in the first hidden layer.

The trick in the FOLLOW scheme is to pre-learn the readout weights $d_{\gamma i}$ to be an auto-encoder with respect to error feedback weights $e_{i\gamma}$. Learning the auto-encoder could be accomplished by existing local learning schemes (Burbank, 2015; Urbanczik & Senn, 2014), and so the auto-encoder was pre-learned algorithmically here. Due to this auto-encoder, the error feedback with high gain k acts as a negative feedback that serves to make the network output $\vec{\hat{u}}(t)$ *follow* the true command torque $\vec{u}(t-\Delta)$, even before the hidden weights are learned. The system dynamics and command torque are required to vary slower than the synaptic timescale. The learning rule is (Gilra & Gerstner, 2017):

$$\frac{dw_{ij}}{dt} = \eta\,(I^\epsilon_i * \kappa^\epsilon)(S_j * \kappa)(t), \quad (6)$$

where $I^\epsilon_i \equiv k\sum_\alpha e_{i\alpha}\epsilon_\alpha$ is the error current injected into each neuron, κ^ϵ is a decaying exponential filter of time constant 200 ms and S_j is the pre-synaptic spike train. Therefore, the learning rule uses only quantities that are locally available at the site of the synapse.

This FOLLOW learning rule is applied on the plastic weights (in red in Fig. 1B), while the current state of the arm serves as input to the network. Due to the error feedback loop, the network output follows the time-delayed command input torque. FOLLOW learning has been shown, using a Lyapunov approach and under reasonable approximations, to be uniformly stable, with the squared error in the learned output tending asymptotically to zero (Gilra & Gerstner, 2017). Details and parameters for generating the random input, the arm dynamics and the fixed random network parameters are as in (Gilra & Gerstner, 2017), but the learning rule has so far not been tested on the inverse model.

After the training phase, we expect that even if we remove the error feedback loop, the network will still be able to infer the command input given the current trajectory.

3. Learning the Inverse Model: Inferring Control Command Given Arm Trajectory

At the start of learning, all trainable weights were initialized to zero. The state trajectory was fed as input to the network, while the network had to learn to produce the command (delayed by Δ_u) that caused the trajectory.

In Figure 2, we show the different stages in FOLLOW learning for the differential feedforward network of Figure 1B. Before learning, without feedback, the network output remained zero. With feedback on, even before the weights have been learned, the network output approximately followed the desired output i.e. the time-delayed reference command, but a small error remained. With feedback on, the network was trained with the FOLLOW learning rule, which relies on this error fedback into the neurons (Equation (6)). At the end of 10,000 s of FOLLOW learning on the feedforward weights, at learning rate 2×10^{-4}, the error had plateaued, and we froze the weights and removed the feedback, in order to test if the network had learned the inverse model. We also tested the network without feedback on a more structured task trajectory, in addition to random motor babbling. The network output inferred the command given the state trajectory, without feedback, with a mean squared error of 0.00417 ± 0.00096 (over 4 s each of 5 different instantiations of the network, learning and test protocols), indicating that the network had learned the inverse model.

We used a delay Δ_u in supplying the target commands, since the command in the past that caused the current state input needs to be inferred. A second parameter is the differential delay Δ between the undelayed and delayed sets of neurons in the first layer. We swept the delays Δ_u and Δ over typical network time scales, shown in Figure 3A, and found that $\Delta_u \approx 50$ ms and $\Delta \approx 50$ ms gave the lowest test error. This Δ_u is consistent with the delay due to two synaptic filtering time constants of 20 ms each from the input state to the inferred command. Thus, for all remaining simulations including Figure 2, we used $\Delta = 50$ ms and $\Delta_u = 50$ ms. On increasing the number of neurons in the network, the performance of the inverse model improved

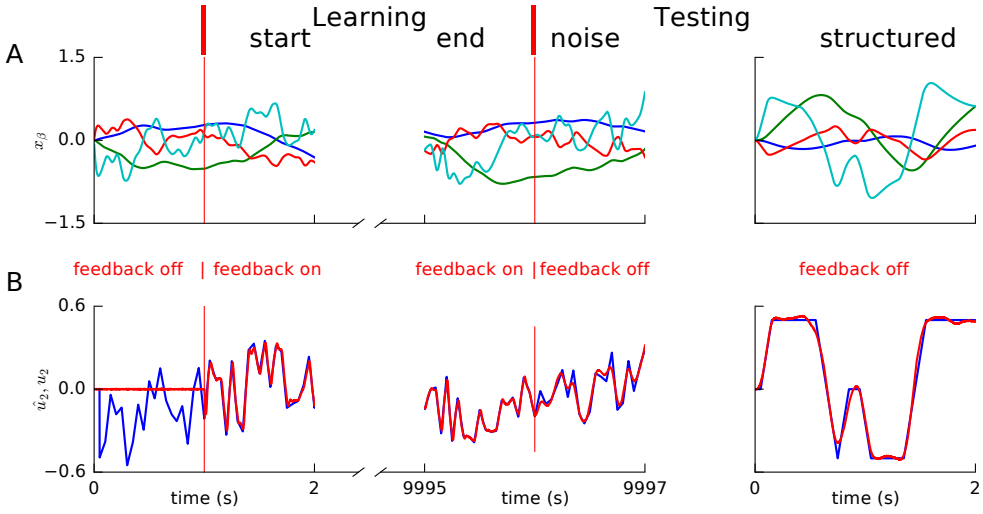

Figure 2. **Stages during FOLLOW learning of inverse model using the differential feedforward network (Fig. 1B). A,B**. The vertical red lines divide the figure into three time stages: before learning without feedback on the left, during learning with feedback in the middle, and during testing without feedback on the right. **A**. The input to the network $\vec{x}(t)$ namely the 4-dimensional state trajectories are shown in 4 different colours. **B**. A component of the reference torque $u_2(t-\Delta)$ to the arm (in blue) is shown along with the network readout $\hat{u_2}(t)$ (in red). Before learning without feedback (on the left), the network output (red) doesn't infer the command torque (blue). During learning with feedback (middle), the network output *follows* the command torque due to the negative feedback (Fig. 1), while the error, which decreases over time, is used to train the hidden weights. After learning (right), i.e. freezing weights and removing feedback, the network infers the command torque even without feedback, indicating that it has learned the inverse model. Test performance measures and parameter sweeps are shown in Figure 3.

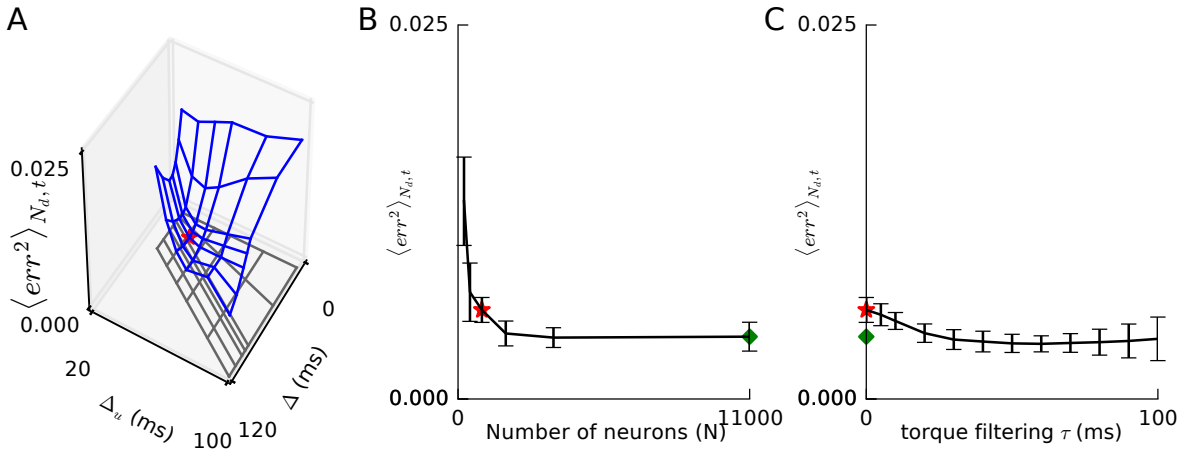

Figure 3. **Mean squared test error versus network parameters.** For different values of the network parameters, we learned the inverse model using the differential feedforward network (Fig. 1B). We plot the mean squared test error without feedback, after approximately 10,000 s of learning, averaged over 4 s and per state dimension N_d. Mean and standard deviations, where marked by error bars, are over five instantiations with different seeds for the network biases, gains and fixed weights, and for the learning and test protocols. **A.** Mean squared test error versus different values of differential delay Δ and causal torque delay Δ_u, with 200 neurons in each of the undelayed and delayed sets of neurons in the first layer and 500 neurons in the next layer. The red star marks the mean squared test error for $\Delta = 50$ ms and $\Delta_u = 50$ ms. Note that the grid of parameters tested (gray grid on bottom surface) is not uniform, with closer sampling near the minimum. **B.** Mean squared error versus number of neurons in the differential feedforward network, fixing $\Delta = 50$ ms and $\Delta_u = 50$ ms. The red star marks the mean squared error for the same parameters as for the red star in panel A, while the green diamond marks it for larger number of neurons as used in Figure 2. **C.** Mean squared error versus increasing values of torque filtering time constant τ, with the same number of neurons as in panel A, fixing $\Delta = 50$ ms and $\Delta_u = 50$ ms. The red star and green diamond correspond to their counterparts in panel B.

(Fig. 3B), as expected. We used 900 neurons when testing other parameters or architectures (Figs. 3, 4, 5), but 11,000 neurons for the final applications (Figs. 2, 7).

In Figure 2B, we observed low-pass filtering of the inferred command compared to the target command during testing. To check its origin, we low-pass filtered the commands for both the arm and the network, with increasing filtering time constants, using a decaying exponential kernel. The mean test error reached a minimum at a filtering time constant of around 50 ms (Fig. 3C) corresponding roughly to the sum of the filtering time constants of our two hidden neuronal layers. In a fully neural system, like the brain, the commands will also be generated by neurons and thus filtered at a synaptic time scale. Still, for generality, we did not use any filtering on the command torque in all our other simulations. With filtering on the command input, performance is expected to improve.

4. Network Architectures

In principle, we could use a recurrent network, as it is Turing complete (Siegelmann & Sontag, 1995), to approximate the inverse model, provided all the input, recurrent and output weights are set correctly. We tried to learn the inverse model with a recurrent network as in Figure 4A. The total current into neuron i in the recurrent layer, having fixed random bias b_i, was a sum of the feedforward, recurrent and error currents:

$$J_i = \sum_\beta e^{\text{ff}}_{i\beta} x_\beta(t) + \sum_j w_{ij}(S_j * \kappa)(t) + \sum_\gamma k e_{i\gamma}(\epsilon_\gamma * \kappa)(t) + b_i. \quad (7)$$

We kept a fixed learning rate on the feedforward weights, but increased the learning rate on the recurrent weights from effectively no learning to a reasonable learning rate 2×10^{-4} that had been suggested for learning the forward model with a recurrent network (Gilra & Gerstner, 2017). As shown in Figure 4, we found that the error increased sharply when the learning rate on the recurrent weights was increased. Near zero learning on the recurrent weights gave the lowest mean squared test error. Even if we included a feedforward layer between the state input and the recurrent network, with learnable feedforward and recurrent weights, as for learning the forward model in (Gilra & Gerstner, 2017), still test performance degraded when learning on the recurrent weights was switched on. Note that a recurrent network in the FOLLOW learning scheme, which has fixed output weights (Fig. 4A), cannot be transformed into the differential feedforward architecture (Fig. 1B). Even if part of the recurrent network can learn to delay or hold an earlier state in memory, it remains connected to the output by fixed weights. However, gated recurrent networks like LSTMs (Hochreiter & Schmidhuber, 1997; Gers et al., 2000) trained via backpropagation through time can learn the inverse model (Rueckert et al., 2017), albeit with a rule that is non-local in time and biologically implausible.

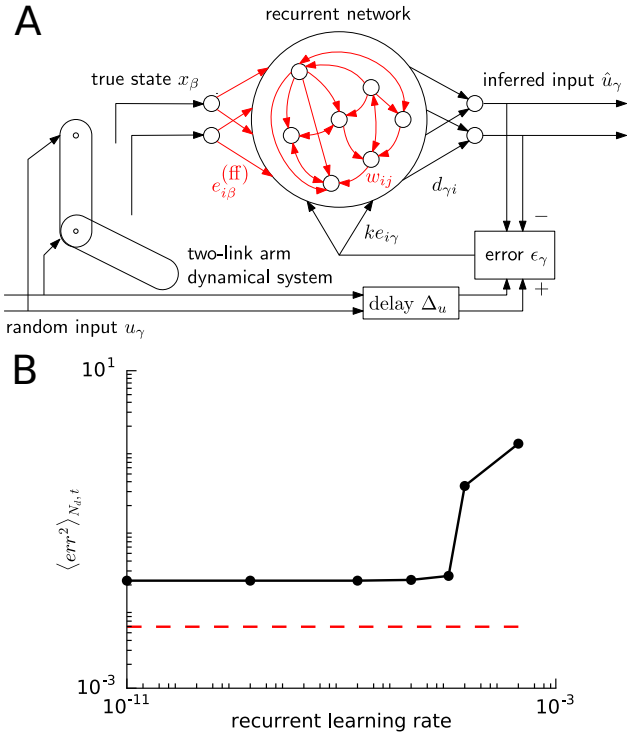

Figure 4. **Recurrent network did not learn the inverse model via FOLLOW learning. A**. The learning configuration is same as Figure 1A, except that the spiking network is recurrently connected. The feedforward weights $e^{\text{ff}}_{i\beta}$ and the recurrent weights w_{ij} (in red) are plastic under the FOLLOW rule. **B**. We trained the recurrent network on the inverse model as in A, with 900 neurons in the recurrent layer and $\Delta_u = 50$ ms, at increasing values of the learning rate (x-axis, log scale) on the recurrent weights, while the feedforward weights were learned at a constant rate. We plot the mean squared test error (y-axis, log scale) without feedback, after approximately 500 s of learning, averaged over 4 s and per state dimension. The red dashed line marks the mean squared test error obtained with the differential feedforward network having 900 neurons (red stars in Fig. 3).

Among different feedforward network architectures, we found that the best performace for a fixed number of neurons was obtained by the differential feedward network (Fig. 5). A variant of the differential feedforward network, with undelayed and delayed state inputs fed to a common set of neurons in the first layer keeping total number of neurons the same, performed similar to our default which had separate undelayed and delayed sets of neurons in the first layer as in Figure 5A. In learning the forward model, the command $\vec{u}$

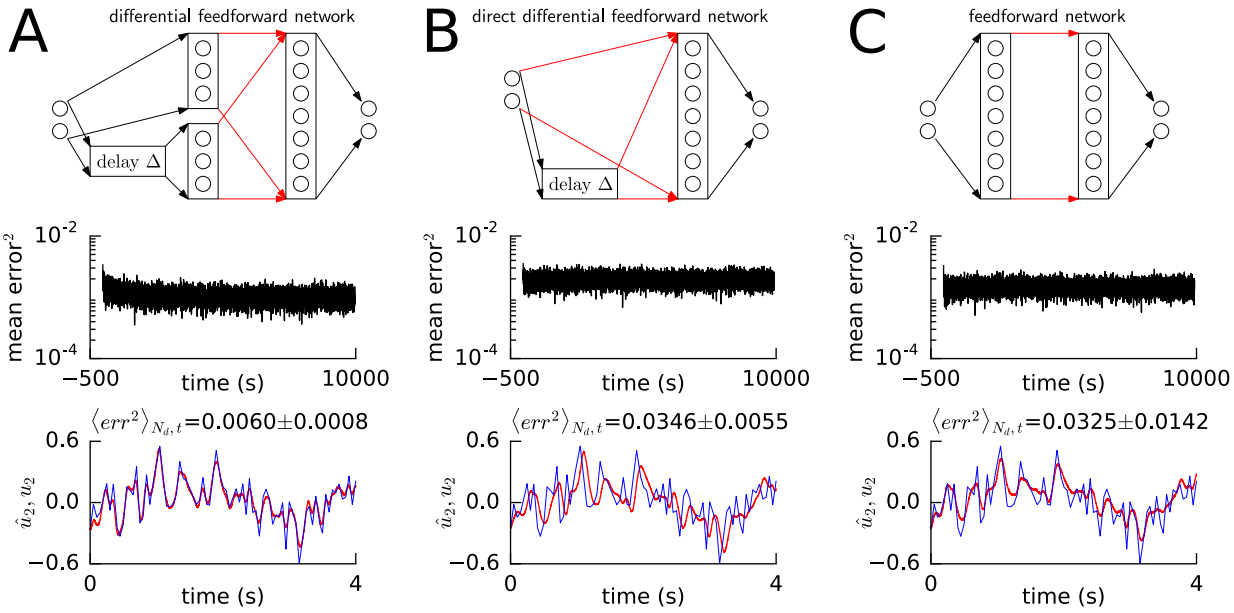

Figure 5. **Differential feedforward performs better than other feedforward architectures. A,B,C**. In the top subpanels, the feedforward network architecture used to learn the inverse model is shown. Weights in red are plastic. Each network has the same number of heterogeneous LIF neurons. Mean squared error during training with feedback is plotted versus time over approximately 10,000s seconds of learning in the middle subpanels. Reference torque component u_2 (blue) and inferred torque component $\hat{u}_2$ (red) given desired state are plotted versus time, during 4 s of testing without feedback. The mean squared test error without feedback per unit time per state dimension is written above the plot (mean and standard deviation are over 5 different instantiations). Mean squared error during test without feedback is typically larger than that towards the end of training, as the latter has corrective negative feedback. **A**. The differential feedforward network is our default network architecture, here with 200 neurons in the undelayed and delayed sets and 500 neurons in the second hidden layer, with $\Delta = 50$ ms and $\Delta_u = 50$ ms. **B**. The direct differential feedforward network doesn't have intermediate undelayed and delayed sets of the first hidden layer. Rather the undelayed and delayed states are directly fed on to a hidden layer having 900 neurons. **C**. The purely feedforward network has 450 neurons in each hidden layer.

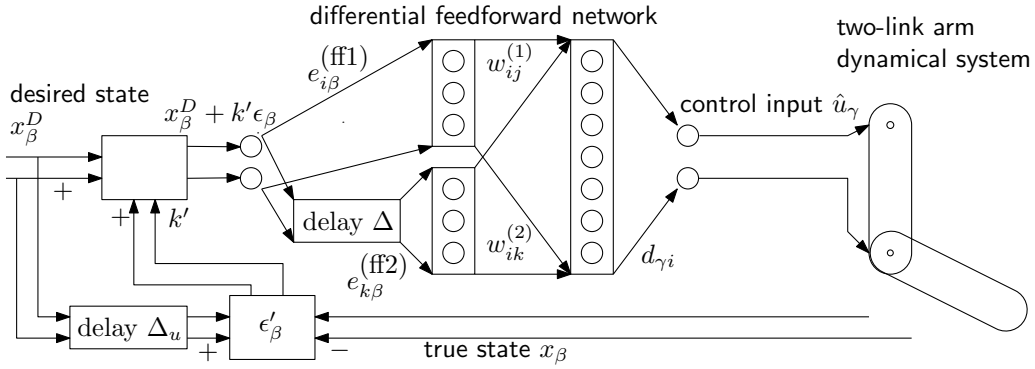

Figure 6. **Schematic for control using inverse model.** The desired state trajectory $\vec{x}^D(t)$ is fed into the differential feedforward network that is already trained as an inverse model. The control input $\hat{u}_\gamma$ is read out from the network. The read out control input torques are delivered to the arm which produces the true state $x_\beta(t)$, which is already close to the desired one if the inverse model performs well. To further improve performance, the true state of the arm is compared with the desired state, delayed by $\Delta_u = 50$ ms (since the inverse model has learned to infer the control input with this delay), and the error $\epsilon'_\beta(t) = \left(x^D_\beta(t - \Delta_u) - x_\beta(t)\right)$ is fed back with gain $k' = 3$. The non-linear feedforward control by the inverse model brings the arm state close to the desired state, after which the linear negative feedback control brings the arm state even closer. Twin lines in the connection arrows denote multi-dimensional signals, but their number is not representative of the dimensionality.

must be integrated as $\int f(\vec{x}, \vec{u})dt$ to obtain the state $\vec{x}$, see equation (1). However, for the inverse model, the state $\vec{x}$ has to be differentiated, and the function f inverted, to infer the command $\vec{u}$. Thus, while a recurrent network is required for learning the forward model (Gilra & Gerstner, 2017), the differential feedforward network is best (among the feedforward and recurrent networks tested) for FOLLOW learning of the inverse model.

5. Moving the Arm As Desired Using the Inverse Model

In a control setting, the control input is not known, but only the target trajectory is given. Having learned the inverse model in the differential feedforward network, we froze its weights, and used it to control the two-link arm to reproduce a desired state trajectory. In the open loop control mode, the desired state $x^D_\beta(t)$ was fed to the network, which outputs the requisite control command $\hat{u}_\gamma$. This command generated by the network was fed as torques to the joints of the arm to produce the desired motion. In this open loop control mode, small errors in the generated control command will integrate over time, causing deviations from the desired trajectory.

To ameliorate this, we closed the loop, with a weak feedback of gain $k' = 3$, injecting the difference between the desired state (delayed by $\Delta_u = 50$ ms) and the true state back into the network input (Fig. 6). We emphasize that this feedback loop for the error in the state variables is different from the feedback loop for the error in the control command during learning.

We tested our motor control scheme for drawing a parallelogram and a zigzag on the wall (Fig. 7). In open loop control mode, the command torque generated by the network enabled the arm to draw a pattern that was roughly similar to the desired one (Fig. 7C,D and Supplementary movies 1 and 2). Open loop control mode can be considered similar to drawing with eyes shut, i.e. without any sensory feedback. When we closed the loop with gain $k' = 3$, the pattern was reproduced better (Fig. 7C,D and Supplementary movies 3 and 4).

Without the inverse model, the linear negative feedback loop on the state variables cannot make the arm follow the desired trajectory, even by tuning feedback gains, because of the complicated non-linear dynamics of the two-link arm. In our control scheme, the intuition is that the inverse model learned in the weights of the network produces an open-loop command that brings the arm close enough to the desired state, such that linear negative feedback around this momentary operating point can bring the arm even further closer.

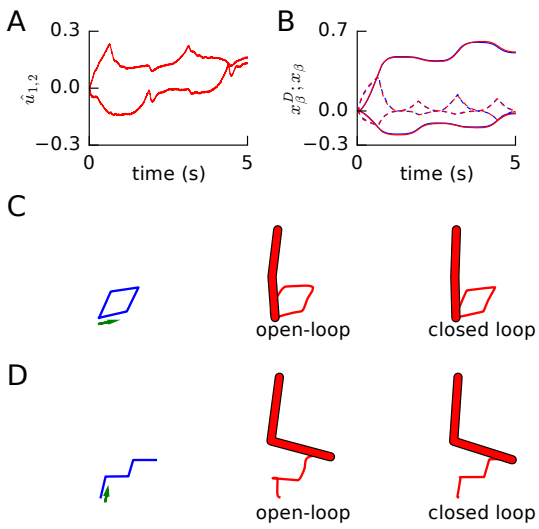

Figure 7. **Control of two-link arm via inverse model. A.** The elbow and shoulder control torques generated by the learned differential feedforward network, under closed loop in the control architecture of Figure 6, in reponse to a desired zigzag drawing trajectory. **B.** The zigzag drawing trajectories for the angles (solid) and angular velocities (dashed) as desired (blue), and as produced by the arm (red) in response to the torques in A, under closed loop control of the inverse model. **C,D.** The network controls the arm to draw a parallelogram (C) and a zigzag (D). On the left in blue, is the desired trajectory, in coordinate space, that the endpoint of the arm is required to trace, starting in the direction of the green arrow. In the middle in red, is the trace produced by the arm under open-loop control, i.e. no feedback of the error in state back to the network. To the right, in red is the trace produced under closed-loop, with feedback (gain $k' = 3$) of the error in state variables.

6. Discussion

We used the synaptically local, online and stable learning scheme, FOLLOW (Gilra & Gerstner, 2017), to train a heterogeneous network of spiking neurons with hidden layers, to infer the continuous-time command needed to drive a non-linear dynamical system, here a two-link arm, to produce a desired trajectory. We found that the differential feedforward network architecture performed best in learning the inverse model, from among a variety of feedforward and recurrent architectures. Under closed-loop, the trained inverse model was able to control a two-link arm. Here, we used only proportional feedback with low gain. A proportional-integral-derivative (PID) feedback should provide even better control of the arm without oscillations and offset.

Previous methods of training neural networks with hidden units incorporate only some of the desirable features of

FOLLOW learning. Backpropagation and variants, backpropagation through time (BPTT) (Rumelhart et al., 1986) and real-time recurrent learning (RTRL) (Williams & Zipser, 1989), are non-local in time or space (Pearlmutter, 1995). Synaptically local learning rules in recurrent spiking neural networks have typically not been demonstrated for learning inverse models and non-linear motor control (MacNeil & Eliasmith, 2011; Bourdoukan & Denève, 2015; Gilra & Gerstner, 2017; Alemi et al., 2017). Reservoir computing methods (Jaeger, 2001; Maass et al., 2002; Legenstein et al., 2003; Maass & Markram, 2004; Jaeger & Haas, 2004; Joshi & Maass, 2005; Legenstein & Maass, 2007; Waegeman et al., 2012) did not initially learn weights within the hidden units, while newer FORCE learning methods that did (Sussillo & Abbott, 2009; 2012; DePasquale et al., 2016; Thalmeier et al., 2016; Nicola & Clopath, 2016), require weights to change faster than the time scale of the dynamics, and require different components of a multi-dimensional error to be fed to different sub-parts of the network. Reward-modulated Hebbian rules have not yet been used for continuous-time control (Legenstein et al., 2010; Hoerzer et al., 2014; Kappel et al., 2017). Several known approaches to neural control of non-linear systems use non-local learning rules (Narendra & Parthasarathy, 1990; Sanner & Slotine, 1992; Slotine & Li, 1987; Slotine & Coetsee, 1986; DeWolf et al., 2016; Zerkaoui et al., 2009; Hennequin et al., 2014; Song et al., 2016; Rueckert et al., 2017) or abstract networks (Berniker & Kording, 2015; Hanuschkin et al., 2013), and are thus further away from biological plausibility than our spiking network with local plasticity suggested here.

Our network approximates the inverse dynamics using the tuning curves of heterogeneous neurons as an overcomplete set of basis functions (Funahashi, 1989; Hornik et al., 1989; Girosi & Poggio, 1990; Eliasmith & Anderson, 2004). As studied in adaptive control theory (Ioannou & Sun, 2012; Narendra & Annaswamy, 1989), the error due to model approximation causes a drift in weights, which can lead to error increasing after some training time. The same literature also suggests ameliorative techniques which include a weight leakage term switched on slowly, whenever a weight crosses a set value (Ioannou & Tsakalis, 1986; Narendra & Annaswamy, 1989), or a dead zone policy of not updating weights if the error falls below a threshold (Slotine & Coetsee, 1986; Ioannou & Sun, 2012). However, we did not need to implement these techniques for learning the inverse model with our networks.

Our scheme should be applicable to any arbitrary fully-observed finite-dimensional dynamical system. Depending on the non-linearity and the coupling amongst the dimensions of the dynamical system, we expect the number of neurons required to scale supra-linearly with the dimensionality. The analysis of the precise scaling is left for further work. The learning rule and architecture can also be used with continuous-valued artificial neural networks, with either the same or potentially even a smaller number of neurons.

Relating our work to neuroscience, forward and inverse models have been linked to the cerebellum and the motor cortex (Kawato, 1999; Sabes, 2000; Golub et al., 2015). The FOLLOW rule predicts that plasticity in feedforward or recurrent synaptic connections is modulated by independent error feedback, reminiscent of plasticity in parallel fibre to Purkinje cell synapses mediated by supervisory feedback via climbing fibres (D'Angelo, 2014), often tuned to account for sensory delays up to 150 ms (Suvrathan et al., 2016). Further, the delay lines of around 50 ms required for learning and control can be created by harnessing smaller synaptic delays in a spiking neural network (Voelker & Eliasmith, 2017). Additional detailed modelling, large-scale recordings and systems-level experiments are required to verify or falsify the learning rule and the predicted architectures for the forward and inverse models.

Our FOLLOW-based learning of motor control can be implemented directly in neuromorphic hardware (Schuman et al., 2017) and incorporated into (neuro-)robotics for motor control, where the spike-based coding provides power efficiency, and locality eases hardware implementation and speeds up computations required for learning.

Interesting directions to explore could be to learn and control more complex dynamical systems like robotic arms in real-time, to implement the same on neuromorphic hardware, to include Dale's law into the FOLLOW scheme, to learn to generate the desired trajectory given a higher-level goal via reinforcement learning, and to extend motor control to a hierarchy of levels as in the brain.

Code for learning the inverse model using FOLLOW and employing it for closed-loop control, is available at https://github.com/adityagilra/FOLLOWControl.

7. Acknowledgements

Financial support was provided by the European Research Council (Multirules, grant agreement no. 268689), the Swiss National Science Foundation (Sinergia, grant agreement no. CRSII2_147636), the European Commission Horizon 2020 Framework Program (H2020) (Human Brain Project, grant agreement no. 720270), and the German Federal Ministry of Education and Research (Bernstein Network – Bernstein Award 2014 to Raoul-Martin Memmesheimer).

References

Abbott, L. F., DePasquale, B., and Memmesheimer, R.-M. *Nature Neuroscience*, 19(3):350–355, 2016.

Alemi, A., Machens, C., Denève, S., and Slotine, J.-J. *arXiv:1705.08026 [q-bio]*, 2017.

Bengio, Y., Simard, P., and Frasconi, P. *IEEE Transactions on Neural Networks*, 5(2):157–166, 1994.

Berniker, M. and Kording, K. P. *Frontiers in Computational Neuroscience*, 9, 2015.

Bourdoukan, R. and Denève, S. In *Advances in Neural Information Processing Systems 28*, pp. 982–990. Curran Associates, Inc., 2015.

Burbank, K. S. *PLoS Computational Biology*, 11(12): e1004566, 2015.

Conant, R. C. and Ashby, W. R. *Intl. J. Systems Science*, pp. 89–97, 1970.

Dadarlat, M. C., O'Doherty, J. E., and Sabes, P. N. *Nature Neuroscience*, 18(1):138–144, 2015.

D'Angelo, E. In Ramnani, N. (ed.), *Progress in Brain Research*, volume 210 of *Cerebellar Learning*, pp. 31–58. Elsevier, 2014.

DePasquale, B., Churchland, M. M., and Abbott, L. F. *arXiv:1601.07620 [q-bio]*, 2016.

DeWolf, T., Stewart, T. C., Slotine, J.-J. E., and Eliasmith, C. *Proc. R. Soc. B*, 283(1843):20162134, 2016.

Eliasmith, C. and Anderson, C. H. MIT Press, 2004.

Funahashi, K.-I. *Neural Networks*, 2(3):183–192, 1989.

Gers, F. A., Schmidhuber, J. A., and Cummins, F. A. *Neural Computation*, 12(10):2451–2471, 2000.

Gilra, A. and Gerstner, W. *eLife*, 6:e28295, 2017.

Girosi, F. and Poggio, T. *Biological Cybernetics*, 63(3): 169–176, 1990.

Golub, M. D., Yu, B. M., and Chase, S. M. *eLife*, 4:e10015, 2015.

Hanuschkin, A., Ganguli, S., and Hahnloser, R. *Frontiers in Neural Circuits*, 7:106, 2013.

Hennequin, G., Vogels, T. P., and Gerstner, W. *Neuron*, 82 (6):1394–1406, 2014.

Hochreiter, S. and Schmidhuber, J. *Neural Computation*, 9 (8):1735–1780, 1997.

Hochreiter, S., Bengio, Y., Frasconi, P., and Schmidhuber, J. Wiley-IEEE Press, 2001.

Hoerzer, G. M., Legenstein, R., and Maass, W. *Cerebral Cortex*, 24(3):677–690, 2014.

Hornik, K., Stinchcombe, M., and White, H. *Neural Networks*, 2(5):359–366, 1989.

Ioannou, P. and Sun, J. Dover Publications, Mineola, New York, 2012.

Ioannou, P. and Tsakalis, K. *IEEE Transactions on Automatic Control*, 31(11):1033–1043, 1986.

Jaeger, H. Technical report, 2001.

Jaeger, H. and Haas, H. *Science*, 304(5667):78–80, 2004.

Joshi, P. and Maass, W. *Neural Computation*, 17(8):1715–1738, 2005.

Kappel, D., Legenstein, R., Habenschuss, S., Hsieh, M., and Maass, W. *arXiv:1704.04238 [cs, q-bio]*, 2017.

Kawato, M. *Current Opinion in Neurobiology*, 9(6):718–727, 1999.

Khazipov, R., Sirota, A., Leinekugel, X., Holmes, G. L., Ben-Ari, Y., and Buzsáki, G. *Nature*, 432(7018):758–761, 2004.

Lalazar, H. and Vaadia, E. *Current Opinion in Neurobiology*, 18(6):573–581, 2008.

Legenstein, R. and Maass, W. *Neural Networks*, 20(3): 323–334, 2007.

Legenstein, R., Markram, H., and Maass, W. *Reviews in the Neurosciences*, 14(1-2):5–19, 2003.

Legenstein, R., Chase, S. M., Schwartz, A. B., and Maass, W. *Journal of Neuroscience*, 30(25):8400–8410, 2010.

Li, W. PhD thesis, University of California, San Diego, 2006.

Maass, W. and Markram, H. *Journal of Computer and System Sciences*, 69(4):593–616, 2004.

Maass, W., Natschläger, T., and Markram, H. *Neural Computation*, 14(11):2531–2560, 2002.

MacNeil, D. and Eliasmith, C. *PLoS ONE*, 6(9):e22885, 2011.

Meltzoff, A. N. and Moore, M. K. *Early development & parenting*, 6(3-4):179–192, 1997.

Morse, A. *IEEE Transactions on Automatic Control*, 25(3): 433–439, June 1980.

Narendra, K., Lin, Y.-H., and Valavani, L. *IEEE Transactions on Automatic Control*, 25(3):440–448, 1980.

Narendra, K. S. and Annaswamy, A. M. Prentice-Hall, Inc, 1989.

Narendra, K. S. and Parthasarathy, K. *IEEE Transactions on Neural Networks*, 1(1):4–27, 1990.

Nicola, W. and Clopath, C. *arXiv:1609.02545 [q-bio]*, 2016.

Pearlmutter, B. A. *IEEE Transactions on Neural Networks*, 6(5):1212–1228, September 1995.

Petersson, P., Waldenström, A., Fåhraeus, C., and Schouenborg, J. *Nature*, 424(6944):72–75, 2003.

Pouget, A. and Snyder, L. H. *Nature Neuroscience*, 3:1192–1198, 2000.

Rueckert, E., Nakatenus, M., Tosatto, S., and Peters, J. In *2017 IEEE-RAS 17th International Conference on Humanoid Robotics (Humanoids)*, pp. 811–816, 2017.

Rumelhart, D. E., Hinton, G. E., and Williams, R. J. In *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Vol 1*, volume 1. MIT Press Cambridge, MA, USA, 1986.

Sabes, P. N. *Current Opinion in Neurobiology*, 10(6):740–746, 2000.

Sanner, R. M. and Slotine, J.-J. E. *IEEE Transactions on Neural Networks*, 3(6):837–863, November 1992.

Sarlegna, F. R. and Sainburg, R. L. *Advances in experimental medicine and biology*, 629:317–335, 2009.

Schuman, C. D., Potok, T. E., Patton, R. M., Birdwell, J. D., Dean, M. E., Rose, G. S., and Plank, J. S. *arXiv:1705.06963 [cs]*, 2017.

Siegelmann, H. T. and Sontag, E. D. *Journal of Computer and System Sciences*, 50(1):132–150, 1995.

Slotine, J.-J. E. and Coetsee, J. A. *International Journal of Control*, 43(6):1631–1651, 1986.

Slotine, J.-J. E. and Li, W. *The International Journal of Robotics Research*, 6(3):49–59, 1987.

Song, H. F., Yang, G. R., and Wang, X.-J. *PLOS Comput Biol*, 12(2):e1004792, February 2016.

Sussillo, D. and Abbott, L. *PLoS ONE*, 7(5):e37372, 2012.

Sussillo, D. and Abbott, L. F. *Neuron*, 63(4):544–557, 2009.

Suvrathan, A., Payne, H. L., and Raymond, J. L. *Neuron*, 92(5):959–967, 2016.

Thalmeier, D., Uhlmann, M., Kappen, H. J., and Memmesheimer, R.-M. *PLOS Comput Biol*, 12(6): e1004895, 2016.

Urbanczik, R. and Senn, W. *Neuron*, 81(3):521–528, 2014.

Voelker, A. R. and Eliasmith, C. *Neural Computation*, pp. 1–41, December 2017.

Waegeman, T., Wyffels, F., and Schrauwen, B. *IEEE Transactions on Neural Networks and Learning Systems*, 23 (10):1637–1648, 2012.

Williams, R. J. and Zipser, D. *Neural Computation*, 1(2): 270–280, 1989.

Wolpert, D. M. and Ghahramani, Z. *Nature Neuroscience*, 3:1212–1217, 2000.

Wong, J. D., Kistemaker, D. A., Chin, A., and Gribble, P. L. *Journal of Neurophysiology*, 108(12):3313–3321, 2012.

Zerkaoui, S., Druaux, F., Leclercq, E., and Lefebvre, D. *Engineering Applications of Artificial Intelligence*, 22 (4–5):702–717, 2009.