Biologically plausible deep learning

Should airplanes flap their wings?

O'Connor, P.

Publication date
2020

Document Version
Final published version

License
Other

Link to publication

Citation for published version (APA):
O'Connor, P. (2020). *Biologically plausible deep learning: Should airplanes flap their wings?*.

Biologically Plausible Deep Learning

Should Airplanes flap their Wings?

Peter O’Connor

Biologically Plausible Deep Learning: Should Airplanes flap their Wings?

ACADEMISCH PROEFSCHRIFT
ter verkrijging van de graad van doctor
aan de Universiteit van Amsterdam
op gezag van de Rector Magnificus
prof. dr. ir. K.I.J. Maex
ten overstaan van een door het College voor Promoties ingestelde
commissie, in het openbaar te verdedigen
op woensdag 23 september 2020, te 14:00 uur
door Peter Edmund O'Connor
geboren te Mississauga

Promotiecommissie

Promotor:	Prof. Dr Max Welling, Universiteit van Amsterdam
Copromotor:	Dr. Efstratios Gavves, Universiteit van Amsterdam
Overige leden:	Prof. dr. Cees Snoek, Universiteit van Amsterdam
	Prof. dr. Cyriel Pennartz, Universiteit van Amsterdam
	Dr. Herke van Hoof, Universiteit van Amsterdam
	Prof. dr. Sander Bohte, Universiteit van Amsterdam
	Prof. dr. Pieter Roelfsema, Netherlands Inst. for Neuroscience
	Dr. Michael Pfeiffer, Bosch Research

Faculteit der Natuurwetenschappen, Wiskunde en Informatica

This research was supported by Qualcomm and the Netherlands Organisation for Scientific Research. Cover Photo by Sribas Santra.

Abstract

Deep neural networks follow a pattern of connectivity that was loosely inspired by neurobiology. The existence of a layered architecture, with deeper neurons representing increasingly abstract features, was known from neuroscience long before it was used in machine learning. However, when one looks beyond superficial similarities, deep networks appear to be very different than their biological counterparts.

First and foremost, there is the manner in which they are trained. Deep networks are almost universally trained with stochastic gradient descent, where gradients are computed using backpropagation. Backpropagation requires that neurons are able to emit two types of signal - a forward *activation* and a backward *gradient*. Biological neurons send signals down a one-way signalling pathway called an *axon*, and appear to lack any mechanism for backpropagating gradients.

Secondly, there is the means of communication. Backpropagation requires that neurons communicate continuous-valued signals between each other, whereas biological neurons communicate with a stream of all-or-nothing impulses called *spikes*.

Third, there is the domain in which networks are used. Deep networks are typically fed with independent and identically distributed samples of data, whereas biological networks learn online from a single, unceasing, temporally-correlated data stream.

In this thesis, we examine how how we can effectively train neural networks while obeying the biological constraints. This is not only of academic interest. The brain, which by any estimate does vastly more computation than any existing computer, uses only about 20W of power - less than a light bulb. Understanding how it works may help us to build more efficient computing hardware.

This thesis includes the work of four published papers, which address the following questions, respectively:

- How can we exploit temporal redundancy in data for more efficient inference?
- How can we exploit temporal redundancy in data for more efficient training?
- How can we train a feedforward network without backpropagation?
- How can we achieve gradient descent when neurons are confined only to emit quantized signals, and cannot send signals backwards?

The results from this work help us to see what a truly brain-like machine learning architecture may look like.

Samenvatting (Summary in Dutch)

Diepe neurale netwerken volgen een patroon van connectiviteit geïnspireerd door neurobiologie. Het bestaan van een gelaagde architectuur, met diepere neuronen die steeds abstractere kenmerken vertegenwoordigen, was bekend uit de neurowetenschappen voordat het werd gebruikt in machine learning. Maar afgezien van deze oppervlakkige overeenkomsten zien diepe netwerken er heel anders uit dan hun biologische tegenhangers.

Allereerst is er de manier waarop ze zijn getraind. Diepe netwerken zijn bijna universeel getraind met stochastische gradiëntafdaling, waarbij gradiënten worden berekend met backpropagation. Backpropagation vereist dat neuronen twee soorten signalen kunnen uitzenden: een forward *activering* en een backward *gradiënt*. Biologische neuronen sturen signalen via een eenrichtingssignaleringspad - *het axon*. Ze lijken een mechanisme te missen voor het terugpropageren van gradiënten.

Ten tweede is er het communicatiemiddel. Backpropagation vereist dat neuronen onderling communiceren met analoge signalen. Maar biologische neuronen communiceren met een stroom van alles-of-niets elektrische pulsen die worden *spikes* genoemd.

Ten derde is er het domein waarin netwerken worden gebruikt. Diepe netwerken worden doorgaans gevoed met onafhankelijke en identiek gedistribueerde steekproeven van gegevens, terwijl biologische netwerken online leren van een enkele, onophoudelijke, tijdelijk gecorreleerde gegevensstroom.

In dit proefschrift onderzoeken we hoe we neurale netwerken effectief kunnen trainen zonder biologische beperkingen te schenden. Dit is niet alleen van academisch belang. De hersenen verbruiken slechts 20W aan vermogen - minder dan een gloeilamp. Als we begrijpen hoe het werkt, kunnen we efficiëntere computerhardware bouwen. Misschien.

Dit proefschrift omvat het werk van vier gepubliceerde artikelen. Deze artikelen behandelen de volgende vragen:

- Hoe kunnen we tijdelijke redundantie in gegevens benutten voor een efficiëntere gevolgtrekking?
- Hoe kunnen we tijdelijke redundantie in gegevens benutten voor efficiëntere training?
- Hoe kunnen we een feedforward-netwerk trainen zonder backpropagation?
- Hoe kunnen we gradiëntafdaling bereiken wanneer neuronen beperkt zijn tot het uitzenden van gekwantiseerde signalen en geen signalen achteruit kunnen sturen?

De resultaten van dit werk helpen ons te zien hoe een echt hersenachtige architectuur voor machine learning eruit kan zien.

Contents

1. Introduction

It is an interesting case of convergent evolution that the models achieving state-of-the art performance in machine learning bear some resemblance to the neural networks that comprise our brains. These models, collectively referred to as *Deep Neural Networks*, have in the last 10 years been overtaking more traditional machine learning models in tasks such as image recognition and synthesis, machine translation, audio transcription and synthesis, and reinforcement-learning tasks.

Both biological and artificial neural networks are “deep”, in the sense that they consist of multiple layers of representation building increasingly abstract features on top of each other. Both are made up of neurons which receive input signals from other neurons, weigh each of them by some scalar *synaptic weight* and aggregate them into a neuron *potential* which is is then related by some nonlinear *activation* signal to the output of that neuron. Both biological networks and *convolutional networks* - the most commonly used architecture for computer-vision tasks - share the property of local-connectivity: Neurons respond to stimuli only in certain regions of the visual field, and primarily connect to other neurons in those same regions. Both types of networks learn their synaptic weights by accumulating small adjustments in response to incoming data.

However, there are some seemingly fundamental differences between biological and artificial neural networks. Artificial neural networks are almost always trained with *backpropagation* - the application of reverse-mode automatic differentiation to compute the effect that a neuron’s parameters have on the loss function. Backpropagation requires that neurons have a "bi-directional" signalling mechanism, wherein an error-gradient is propagated in a direction opposite to the activation. Biological neurons lack a secondary signalling mechanism of this type - they transmit signals down strictly one-way pathways called *axons.* Instead, they have “feedback” connections, wherein high-level neurons send signals down their axons to lower level neurons. How these feedback connections could communicate information analogous to an error gradient is unclear. Additionally, backpropagation requires that each neuron’s output be a real number which is a differentiable function of the input. Biological neurons absolutely do not have this property - they communicate with a stream of all-or-nothing impulses called *spikes.*

The manner in which biological networks learn is also very different from that of their counterparts in deep learning. While deep networks are typically trained for millions of iterations off of random sub-samples of a dataset, biological neurons must learn off of a single temporal stream of data that comes in through the sensors. There are no distinct ‘data points’ but a only a stream of impulses from the sensory organs that indicate some update to the state of the world.

The similarities of artificial neural networks to their biological counterparts, and their significant contribution to machine learning in the last decade, would seem to hint that machine learning may provide us with a valuable perspective on what the brain is doing.

Conversely, understanding biological neurons may help us find new ways to apply machine learning to real-world situations. The brain is an asynchronous machine: It receive an asynchronous stream of sensory inputs, it consists of interconnected modules which appear to function without a central clock, and these modules produce another stream of impulses to the muscles. Unlike almost all machine learning systems, there is no global *update step*, wherein a sequence of modules pass information to each other in an orderly manner in order to produces an output. Rather, it is best understood as a continuous-time dynamical system. This makes the brain a very natural system for controlling a robot - another continuous-time dynamical system - which is not surprising, because that is what it evolved to do. Most of the work on understanding artificial neural networks as continuous time-dynamical systems comes from the computational neuroscience literature. The question of how to train such a system to do anything useful has not yet been answered.

In the remainder of this introduction, we will begin by examining the similarities and differences of biological and artificial neural networks in Sections 1.1 to 1.5. In Section 1.6, we will discuss why with think it is worth exploring these differences, and in Section 1.7 we will introduce the remainder of this thesis.

1.1. An Artificial Neural Network

The "neurons" used in deep learning are nonlinear continuous functions mapping a vector input to a scalar output. They generally have the form:

$$s_j = h\left(\sum_{i \in \text{inputs(j)}} s_i w_{ij} + b_j\right) \tag{1.1}$$

Where $s_j \in \mathbb{R}$ is the activation of a neuron, s_i is an input (either the input data $s_i := x_i : i \in \mathcal{I}$, or the activations of another neuron), inputs(j) is the set of neurons directly feeding in to neuron j, $h : \mathbb{R} \mapsto \mathbb{R}$ is some scalar, differentiable nonlinearity such as $\tanh(x)$ or $\max(0, x)$, $w_{ij} \in \mathbb{R}$ the strength of a *synaptic weight* controlling how strongly neuron i influences neuron j, b_j is a scalar bias. Because the bias b_j can be considered an additional weight connected to an input whose value is always 1, we often omit b_j throughout the rest of this thesis for notational brevity.

The nonlinearity h allows a network of these neuron with at least one hidden layer of arbitrary width to approximate any function with arbitrary precision [Hornik et al., 1989]. A network of linear neurons can only implement a linear function.

In a typical deep network, such neurons are composed as a *directed acyclic graph.* In other words, neurons are ordered such that neuron i feeds its activation into neuron $j : j > i$, but **not** the other way around. This stands in stark contrast to biological neurons, where reciprocal connections are common. Given an input vector $x \in \mathbb{R}^D$, some set of output activations $s_{\mathcal{O}} := f(x) \in \mathbb{R}^{|\mathcal{O}|}$ are computed by recursively applying Equation 1.1 until all nodes in the graph are computed. The output neuron activations are then compared to a target y using some *loss* function ℓ to compute a scalar loss:

$$\mathcal{L} = \ell(s_{\mathcal{O}}, y) = \ell(f(x), y) \in \mathbb{R}^+ \tag{1.2}$$

The parameters of the networks are then trained with *gradient descent*, with gradients computed through *backpropagation* (also known as *reverse mode automatic differentiation* - see Section 2.5). For training the parameters of a network of these neurons, a crucial characteristic is that neurons be differentiable. In *backpropagation*, neurons pass their loss-derivatives backwards through the directed acyclic graph in a backward pass:

$$\frac{\partial \mathcal{L}}{\partial w_{ij}} = \frac{\partial \mathcal{L}}{\partial s_j}\frac{\partial s_j}{\partial w_{ij}} \qquad \text{Compute Parameter Gradients} \tag{1.3}$$

$$\frac{\partial \mathcal{L}}{\partial s_i} = \sum_{j \in \text{outputs}(i)} \frac{\partial \mathcal{L}}{\partial s_j}\frac{\partial s_j}{\partial s_i} \qquad \text{Compute Input Gradients} \tag{1.4}$$

Where w_{ij} refers to the weighting that neuron j applies applies to it's $i'th$ input (the *synaptic weight*), and outputs(i) is the set of neurons that neuron i directly connects to. Parameters θ (comprising the w and b of every neuron in the network) are then updated by some gradient descent rule - the simplest of which is to take a small step down the loss gradient:

$$\theta_i = \theta_i - \eta \frac{\partial \mathcal{L}}{\partial \theta_i} \tag{1.5}$$

Where $\theta_i \in \mathbb{R}$ is a parameter (the weight or bias of some neuron in the network) $\eta \in \mathbb{R}^+$ is a learning rate, defining how large a step to take. $\frac{\partial \mathcal{L}}{\partial \theta_i}$ is the derivative of the loss with respect to the parameter value, as computed by backpropagation.

Typically such networks are trained on a fixed dataset $\mathcal{D} = \{(x_n, y_n) : n \in [1..N]\}$. Such datasets can be large, so computing the loss gradient $\frac{\partial \mathcal{L}}{\partial \theta} = \sum_{n=1}^{N} \ell(f(x_n), y_n)$ using the entire dataset on each iteration would result in very slow training. Fortunately, minimizing the loss does not require that we step in the *exact* direction of the gradient at each iteration, but only that on average, we move in the direction of the gradient.

We can thus compute our updates much faster by only sampling a small *minibatch* $\mathcal{D}_{mini} \sim \mathcal{D}$ of data points selected randomly from the dataset, on each iteration, and using the approximate gradients $\frac{\partial \mathcal{L}}{\partial \theta}(\mathcal{D}_{mini}) \approx \frac{\partial \mathcal{L}}{\partial \theta}(\mathcal{D})$ in our update. This is known as *stochastic gradient descent.*

1.1.1. Time in ANNs

It is worth noting how *time* is handled in artificial neural networks. The deep learning literature almost always handles time as a discrete quantity, with steps corresponding to regular time intervals. This allows us to still think of a network as a directed acyclic graph, with edges connecting to both higher layers and future time-steps. In this setting, the network can compute gradients with a forward/backward pass in time in the same manner as it computes a forward/backward pass through layers of the network. This is known as **Backpropagation Through Time**.

There are two basic ways of designing networks to handle temporal data: recurrent networks and temporal convolutional networks - analogous the concepts of infinite Impulse Response (IIR) and Finite Impulse Response (FIR) filters, from signal processing:

- **Recurrent Networks (RNNs)**: The network has the form $s_t = f_\theta(s_{t-1}, x_t)$, where $x_t \in \mathbb{R}^{D_{in}}$, $s_t \in \mathbb{R}^{D_{hid}}$ are the state and input at time t, respectively, and s_0 is generally initialized to a vector of zeros. A directed acyclic graph is formed by recursively computing $[s_1 = f_\theta(s_0, x_1), s_2 = f_\theta(s_1, x_2), ...s_t = f_\theta(s_{t-1}, x_t)]$, finding some loss: $\mathcal{L} = \sum_t \ell(s_t, y_t)$, and then using backpropagation to compute the derivatives with respect to θ. When the sequence x is an infinite stream, a *truncation window* T is typically used, so that derivatives are only computed using data from steps $t - T$ to t. This is known as *Truncated Backpropagation Through Time*

- **Temporal Convolutional Networks (TCNs)**: The network has the form $s_t = f_\theta(x_{t-T}...x_t)$, and is composed of a series of stacked layers, where within each layer, weights are shared across time. Neurons in TCNs have the form $s_t = h\left(\sum_{\tau=1}^{T} x_{(t-\tau\Delta T)} w_\tau + b\right)$, where x is the neuron's input (for the first layer, the input is the data. For later layers, it is the output of the previous layer), $\Delta T \in \mathbb{N}$ is the temporal sub-sampling factor, which is 1 for the first layer and gets larger for higher layers. TCNs have become popular for sequence modelling in recent years, since Oord et al. [2016] demonstrated the good performance of their *WaveNet* TCN for predicting and generating audio data.

1.1.2. Energy-Based Models

Feedforward networks are not the only type of artificial neural network. Another class is known as "energy-based models". In these models, an energy function is defined over the parameters and states of the network, and the network dynamics are designed to as to minimize this energy function. Such models are especially interesting as models of biological computation, because they consist of recurrently connected (non-feedforward) networks, and do not necessarily require backpropagation to learn.

The classic energy-based model is the Hopfield Network [Hopfield, 1982].

$$E(s,(w,b)) = -\sum_{ij} w_{ij} s_i s_j - \sum_i b_i s_i \tag{1.6}$$

Where $s_i \in \{-1,+1\}$ is the state of neuron i and $w_{ij} \in \mathbb{R}, b_i \in \mathbb{R}$ are the parameters of the network, and the weight matrix is constrained to be symmetric and zero along the diagonal ($w_{ij} = w_{ji}.w_{ii} = 0$). The constraints allow us to easily define an update rule which minimizes the energy function. Inference in a Hopfield network corresponds to minimizing the energy conditioned on a set of inputs. In other words, a subset of neurons $\{s_i : i \in \text{Input}\}$ are clamped to an input value, and the states of the remaining neurons are adjusted to minimize the energy conditioned on these. This is achieved by updating neurons in a round-robin fashion where $s_i = \arg\min_{s'_i \in \{-1,+1\}} E(s^{s_i = s'_i}, (w,b))$. In the above formulation, this simply resolves to $s_i = (1 \text{ if } \sum_{j \neq i} w_{ij} s_j + b_i > 0 \text{ otherwise } -1)$.

In the classic Hopfield model, "memories" are stored into the network by adjusting the parameters to lower the energy of desired memories. For each example we'd like to memorize, we clamp all units to observed data and adjust the parameters to lower the energy of this state: $\Delta w_{ij} = s_i s_j$, $\Delta b_i = s_i$. The lowered energy of this state means that later, when partial data $\{x_i : i \in \text{Observed Inputs}\}$ is presented, we can clamp the observed nodes and iteratively energy-minimize the remaining nodes as described above to recover the "memory" $s^* = \arg\min_{s|s_{obs}=\text{ObservedData}} E(s,(w,b))$.

The Hopfield Model has been extended into continuous form [Hopfield, 1984], and to a stochastic model known as a Boltzmann Machine [Ackley et al., 1985], which replaces the notion of finding an energy minimum with approaching a stationary distribution. Boltzmann machines will be discussed more in Section 2.1.

Energy based models are rarely used in the modern literature. One major reason for this is that unlike in a feed-forward network, inference is an optimization process. Because training involves repeated inference in a loop, training energy based networks generally consists of running loops within loops, which is slow.

1.1.3. Weight Sharing

In Deep Networks, it is often useful to constrain parameters from different neurons to have the same value. This "weight sharing" is used in a variety of different ways in a variety of different architectures:

- **Spatial weight-sharing in conv-nets**. Convolutional layers use weight sharing to achieve approximate translation-equivariance (i.e. a shift in the input results in a shift in the output). Here, neurons are arranged in a 2D grid, and neurons at different positions in the grid have the same weights, but apply them to different regions of the input image.

- **Temporal weight-sharing in recurrent networks**. In Backpropagation Through Time, a recurrent network is implemented as a function $s_t = f_\theta(s_{t-1}, x_t)$, applied repeatedly for each time-step t. The parameters θ are thus used repeatedly in the execution of the network. When training such a network, we see this as a form of weight-sharing - the network is represented as single graph “unrolled” in time, and the parameters θ are considered to be *shared* between time-steps.

- **Forward-backward weight-sharing in backpropagation**. One way to view backprop is as a computation performed by two neural networks: A *forward* network, where neurons compute their activations as $z_j^l = \sum_i w_{ij} a_i^{l-1}; a_j^l = h(z_j^l)$, and a *backwards* network, which computes the error signal $e_j^l = h'(z_i^l) \sum_k e_k^{l+1} w_{jl}^{l+1}$. Here, each weight w_{ij}^l is shared by both the forward and backward network.

- **Symmetric Weights in Energy-Based Models**. In Section 1.1.2, we discussed a class of Artificial Neural networks called *Energy-Based Models*. Such models usually require that connections between neurons be symmetrical (i.e. $w_{ij} = w_{ji}$) in order for there to be a well-defined energy function for the network. Without symmetric connections, the network does not necessarily converge to a single point (or stationary distribution for stochastic networks), and may experience limit-cycle or chaotic dynamics.

1.1.4. Stochasticity

For regularization and probabilistic prediction, it can be advantageous to build a network out of stochastic neurons. This can be done by either by stochastically rounding activations in the forward pass, and ignoring the stochasticity when computing the gradient, as in *Dropout* [Srivastava et al., 2014], or by treating the noise as a separate input variable $\epsilon \sim \mathcal{N}(1, 1)$ which modulates the output $s' = s\epsilon$, as in [Kingma et al., 2015]. The neuron then remains differentiable, as the noise is treated as an external input. This is known as the *Reparameterization Trick*.

1.2. A Biological Neural Network

Biological neurons are complicated systems, and in this work we abstract away most of their detail. In this thesis we will spend little time on discussing the mechanics of *real* biological neurons and instead focus on how one could build a learning system out of abstract neurons which share the core characteristics of biological neurons. This section provides a very brief introduction to biological neurons.

1.2.1. Dynamical Neurons

Biological neurons are best described as continuous-time dynamical systems. Such systems have the form:

$$\dot{s} = f_\theta(s, x) \tag{1.7}$$

Where $\dot{s} := \frac{ds}{dt}$ is the temporal rate of change of signal s. We can describe a simple neuron as a dynamical system using same symbols as we did in Equation 1.1:

$$\dot{s} = h\left(\sum_i x_i w_i + b\right) - s \tag{1.8}$$

Where s and x vary in time continuously. Note that unlike our previous concept of a neuron in Equation 1.1, this neuron has *state* - its output $s(t)$ is no longer just a function of its instantaneous input $x(t)$. Note also that in the *steady state* (i.e. when the state of the neuron stops changing: $\dot{s} = 0$), the relation between x and s is identical to that of the non-dynamical neuron in Equation 1.1.

1.3. Anatomy of a Neuron

Biological neurons store their internal “state” in the form of an electrical potential across the membrane of the neuron. The “membrane potential” is the difference in voltage between the inside and the outside of the cell.

Biological neurons collect input signals from other neurons via connections called *synapses*. The input signals flow up a tree of wires called called *dendrites* to the cell body, where they contribute to the membrane potential. If this membrane potential is pushed past a threshold, a *spike* is generated (see following section) and transmitted down the *axon*. The axon connects to the dendrites of other neurons via more synapses. Thus the *spike* can change the membrane potential of other neurons via these synapses. Figure 1.1 illustrates the anatomy of a biological neuron.

1.3.1. Spiking

Most neurons in the brains of most animals do not directly communicate with real-valued signals. Instead, they generate *action potentials*, also known as *spikes*. When one looks at the voltage trace of a neuron in an awake animal, one sees something like Figure 1.2:

In the absence of stimulation, the membrane potential will be maintained at a *resting potential* of around -70mV. As neurons receive input (in the form of electrical current from

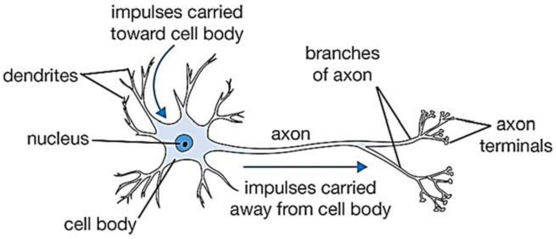

impulses carried toward cell body
dendrites
branches of axon
nucleus
axon
axon terminals
impulses carried away from cell body
cell body

Figure 1.1.: An illustration of a biological neuron [Mijwel, 2017]. In reality the dendritic tree and the axon are much larger, relative to the cell body, than indicated in the illustration, and can receive inputs from thousands of neurons. The cell body can range from $4\mu m$ to $100\mu m$ across. The axon length is extremely variable and can range from under a millimeter in locally connected cortical neurons to over a meter in motor neurons. Dendrites are connected via *synapses* to the axon-terminals of other neurons.

synapses - discussed in the following section), the membrane potential is incremented and decremented, all the while gradually decaying back to the resting potential. If the membrane potential is pushed passed a threshold voltage of -55mV, a spike is generated. The spike consists of a sharp rise (depolarization) in membrane potential beyond the threshold, up to +40mV, followed by a sharp fall (hyper-polarization) down below the resting potential. Figure 1.3 illustrates this process.

Once the spike is generated in the cell body, it propagates down the *axon* - a single cable coming out of the neuron with an active mechanism for propagating voltage spikes. Crucially, the axon is a one-way channel - it transmits spikes from the cell body to the synapses of other neurons, and does not send signals in the reverse direction.

1.4. Modelling a Neuron

The simplest model of a spiking neuron is known as the "Integrate-and-Fire" model. Here, the duration of a spike is ignored, and the spike is treated as a delta function $\delta(t-t_{spike}) := (\infty \text{ if } t = t_{spike} \text{ otherwise } 0)$. We start with a neuron which follows continuous dynamics $\dot{s} = -\frac{1}{\tau_{mem}}(s - s_{rest})$ - meaning that the neuron's membrane potential decays towards a resting potential s_{rest} and will remain there unless disturbed. Solving the dynamical system given initial conditions $s(0) = s_0$, we get $s(t) = (s_0 - s_{rest})e^{-t/\tau_{mem}} + s_{rest}$.

Disturbance comes in two forms - input spikes and output spikes. When the neuron receives a spike from the $i'th$ input neuron, its potential is simply incremented by some amount proportional to the synaptic weight w_i. The effect of this is to simply add a *spike response* $w_i\kappa(t - t_{spike})$, where $\kappa(t) = (e^{-t/\tau_{mem}} \text{ if } t > t_{spike} \text{ otherwise } 0)$ to the neuron's dynamics.

After a spike occurs, the membrane potential is reset to s_{reset}. Equivalently, we can say

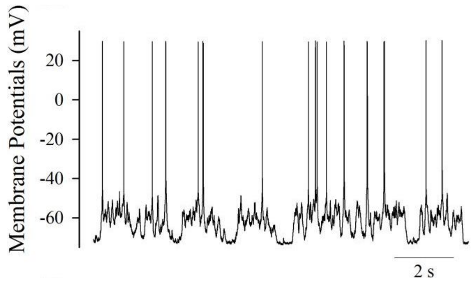


Figure 1.2.: A voltage trace recorded from a cortical neuron in-vivo shows typical spiking behaviour [Ge et al., 2011]. The trace shows the sub-threshold dynamics as the membrane potential is pushed around by stimuli while decaying back to the resting potential. Whenever the membrane potential crosses a threshold, an action potential (spike) is generated. The spikes are transmitted to downstream neurons.

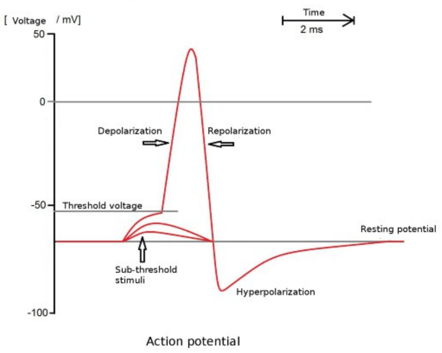


Figure 1.3.: An illustration of a single action potential [Weil, 2017] - the line traces the neuron's membrane potential. An external stimulation brings the membrane potential past a threshold of -55mV, at which point the *action potential* mechanism springs into effect - causing the voltage to suddenly rise then drop to form a spike.

that a potential $s_{thresh} - s_{reset}$ is subtracted off the membrane potential. An output spike acts in the same way on the membrane-potential as an input spike with a weight of $-(s_{thresh} - s_{reset})$. We can now write out the full temporal dynamics for an integrate-and-fire neuron:

$$s(t) = s_{rest} + \overbrace{\sum_n w_{i_n} \kappa^{(t-t_n)/\tau_{mem}}}^{\text{Input Spike Contribution}} - \overbrace{\sum_m (s_{thresh} - s_{reset}) \kappa^{(t-t_m)/\tau_{mem}}}^{\text{Output Spike Contribution}} \tag{1.9}$$

Where:
w_{i_n} is the weight of the input synapse i, from which the $n'th$ input spike comes
$\kappa(t) = (e^{-t/t_{mem}}$ if $t > t_{spike}$ otherwise $0)$ is the spike-response kernel
τ_{mem} is the membrane time constant.

To simplify this model as much as possible, we can set $s_{rest} = s_{reset} = 0$ and $s_{thresh} = 1$, to get:

$$s(t) = \overbrace{\sum_n w_{i_n} \kappa^{(t-t_n)/\tau_{mem}}}^{\text{Input Contribution}} - \overbrace{\sum_m \kappa^{(t-t_m)/\tau_{mem}}}^{\text{Spiking Contribution}} \tag{1.10}$$

The Integrate-and-Fire model is as far as we will go into the modelling of biological neurons. There has been a great deal of research in computational neuroscience into more biologically realistic versions of this model - but for our purposes, the integrate-and-fire model captures the fundamental behaviours:

- Neurons are dynamical systems.
- Neurons communicate by sending “spikes” to one another.
- The only information attached to a spike is its arrival time - spikes carry no further information such as amplitude.

With this simplification, we **ignore** many of the complexities of real biological neurons:

- Biological neurons are either excitatory or inhibitory. When an excitatory neuron fires, it *increases* the membrane potential of its downstream neurons. When an inhibitory neuron fires, it *decreases* it. In other words, biological neurons have the constraint that all outgoing synapses from a given neuron have the same sign.
- Dendrites can have much more complex behaviour than the passive transmission of current from synapses to cell body. They can have nonlinearities, they can gate each other's inputs, and even have spike-like signals of their own - in both the forwards and backwards direction. Thus a single biological neuron may in fact be comparable in function to a small network of our abstract neurons.
- Not all synapses have the form of the axon-dendrite synapse described above. In some brain regions, including the olfactory bulb and the retina, neighbouring neurons form “electrical” or “dendrodendritic” synapses - direct electrical connections between the dendrites of different neurons.
- Biological neurons come in a wide variety of different forms and structures, which presumably makes them perform different functions.
- Global release of neurotransmitters affects the function of neurons.
- Synchronized firing of many nearby neurons forms Local Field Potentials - localized fluctuations in extracellular voltage. When present over a large brain region they can even be measured outside the skull, and are known as Delta, Theta, Beta, Alpha, or Gamma Waves depending on their frequency. These may not just be caused by neural firing but may affect neuron behaviour themselves, possibly acting as a sort of clock for synchronizing neurons.

1.4.1. Spike Timing Dependent Plasticity

Biological neurons have been observed to update their synaptic weights according to the difference in timing between pre-synaptic and post-synaptic spikes. This learning rule is known as *Spike Timing Dependent Plasticity* (STDP) Markram et al. [2012]. Figure 1.4 shows this learning rule. STDP is typically modelled with the Equation 1.11.

$$\Delta w = \begin{cases} -a_- e^{\Delta t/\tau_-} & \text{if } \Delta t < 0 \\ a_+ e^{-\Delta t/\tau_+} & \text{otherwise} \end{cases} \tag{1.11}$$

Where $a_- \in \mathbb{R}^+$ and $a_+ \in \mathbb{R}^+$ are the strings of the anti-causal/causal components, $\Delta t = t_{post} - t_{pre}$ and is the difference in timing between pre- and post- synaptic spikes, and τ_-, τ_+ are the time-constants of the anti-causal/causal time-constants.

This rule says that if a pre-synaptic spike comes *before* a post synaptic spike (meaning it could have caused it - right-side of Figure 1.4) the synapse increases in strength, and if it comes *after* the postsynaptic (meaning it could not have caused it - left-side of Figure 1.4), the synapse decreases in strength.

Up until recently it has not not been clear what STDP means from a machine-learning perspective. One of the more promising interpretations was proposed by Bengio et al. [2015b], where the authors show that, assuming the a network's dynamics (the evolution of states of neurons) are moving towards minimizing an objective function, applying the STDP rule on this network will adjust the weights to also minimize that objective. In Chapter 6 we expand on this idea, and show that STDP can be interpreted as dynamics-based learning in the context of a network of neurons that communicate with Predictive Coding.

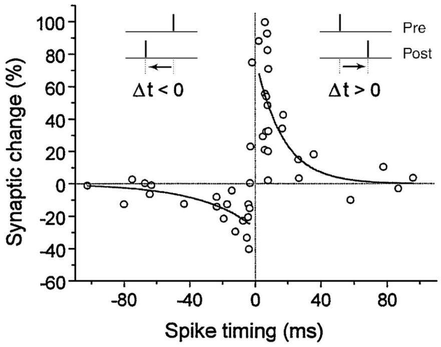


Figure 1.4.: The empirically observed STDP learning rule (figure from Bi and Wang [2002]). When the presynaptic spike comes before the postsynaptic (right half), the synapse strengthens, and when it comes after (left half), the synapse weakens.

1.5. The Gaps

If we compare the artificial neural networks described in Section 1.1 with the biological networks described in Section 1.2, we notice that there are some fundamental gaps in the behaviour of these types of networks. We enumerate them here, and will refer back to them throughout the rest of this work. In Chapter 2, we will discuss how other authors have tackled these points, and in Chapters 3 to 8, we will discuss work that we have done to address these differences.

Gap 1: No Backprop

Neurons used in deep learning emit two types of signals - an activation on the forward pass, and a gradient on the backward pass. Biological neurons send signals down a one-way signalling pathway called an *axon*. They appear to lack any secondary signalling mechanism for sending gradient backwards. That is not to say that information cannot propagate backwards. Biological networks are full of reciprocal connections - where higher-layer neurons send spikes back to their lower-layer inputs. But it is not clear how this information could be used to train the lower layers make themselves more useful to the higher layers.

Backpropagation Through Time (BPTT), the principle method by which recurrent networks are trained, is even more biologically implausible: It would require neurons to build up a buffer of their past activations as they receive data, then at fixed intervals, iterate backwards through this buffer to backpropagate the error signal. No apparent mechanism exists to allow for this behaviour, and so it is not clear at all how biological networks could be learning.

We should note that there is a biological mechanism known as *Neural Backpropagation* [Stuart and Sakmann, 1994], wherein it has been observed that a spike is not only transmitted forward direction down an axon, but also echoes backwards through the dendrites of the spiking neuron, to the input synapses. Critically though, the buck stops there. Unlike in the backpropagation of error signals in artificial networks, there is no mechanism for this signal to hop back across synapses, and down the axons of presynaptic neurons. Thus Neural Backpropagation can only be used as a backwards signalling mechanism within a neuron - not over a network. We will return to the topic Neural Backpropagation at the end of Chapter 6.

Gap 2: Spiking

Neurons in deep learning have continuous, differentiable activation functions. This is necessary in order to propagate useful gradients back through the network. Biological neurons are best understood as dynamical systems which communicate through streams of all-or-nothing “spikes”, as discussed in Section 1.2.

There are two aspects to "spiking" that make it different from the continuous, differentiable activation functions used in the rest of deep learning. First is the **quantized** aspect - neurons do not produce real-valued signals but produce a discretized "spike or no-spike" output. While biological spikes do have an "amplitude" which can be measured in mV, it tends to be roughly the same for all spikes and does not appear to be used to convey information between neurons. Secondly, there is the **temporal** aspect - neurons do not simply output a value representing some function of their current inputs, but produce a series of spikes in time, where the presence or absence of a spike is some function of the recent input to a neuron.

Gap 3: Nonstationary Data

Artificial neural networks are almost always trained with stochastic gradient descent, which involves repeatedly feeding the network small batches of Independent and Identically Distributed (IID) data samples. When the data involves temporal sequences, these samples typically involve "snippets" of sequence short enough that the model consuming them can train off the entire sequence using backpropagation-through time without running into memory-constraints. Biological networks live in a very different domain. They receive a single stream of nonstationary data - neighbouring inputs in time tend to be highly correlated. In most machine learning applications, the experimenter controls the order in which data is fed to the model during training. In nature, the model must cope with data in whatever order it arrives.

Gap 4: Asynchronous Processing

Training artificial neural networks involves a "forward pass", in which the nodes of network are computed in order of their dependency, followed by a "backward pass" where gradients are computed in the reverse order. A physical implementation of this would require global synchronization - where neurons recompute their states when their upstream neurons have completed computation and then hold their states until the gradient propagates back in the backward phase. This "global synchronization" appears not to exist in the brain. It would involve "executing" neurons in stages and asking them to hold onto their state while they wait for a backward pass to arrive. Biological neurons appear, by contrast, to be constantly processing input from both the forward (bottom-up) and backward (top-down) directions. None of the precise control circuitry that would be required to do coordinated forward and backward passes appears to exist. We discuss this in more detail in Section 1.6.2.

Gap 5: No Shared Parameters

A key constraint of biological networks that is ignored in almost all deep networks is that parameters of a neuron belong to that neuron alone. Whereas artificial neural networks share parameters between neurons in a number of ways (see Section 1.1.3), there is no mechanism whereby the weights associated with synapses can be shared across synapses.

1.6. Why care about biological neurons?

There are more reasons to care about how the biological neural networks might work than pure academic curiosity. We believe that two primary reasons are *handling of asynchronous data* (Section 1.6.1), and *energy-efficient computation* (Section 1.6.2).

1.6.1. Asynchronous Data

Suppose, as an ambitious deep learning researcher, you want to try your hand at robotics. Perhaps you want to learn a model of the robot's sensory signals, to see if you can learn high-level percepts like "a door handle that I can pull" from raw image, joint angle, and tactile data. Perhaps you are so ambitious that you would like to learn to generate the motor signals commanding a hand to pull this handle when the moment arrives. Your first step is to take your deep neural network and plug it into the sensors.

You are immediately confronted by an awkward problem of data formats. Instead of the nice time-stepped series of observations $< x_1, ..., x_n >$ that you're familiar with from datasets and simulations, where each x_n summarizes the full sensory state of the robot at a particular time-step, you are confronted by a series of partial observations from various sensors that come in at irregular time intervals.

The gyroscope sends a series of time-stamped 6-dimensional linear-angular acceleration measurements $< (x_1^{gyro}, t_1), ..., (x_n^{gyro}, t_n) >$ around 1,000 times/second, on average, while the joint-angle sensors may send 20-dimensional measurements $< (x_1^{joint}, t_1), ..., (x_n^{joint}, t_n) >$ around 100 times per second, and the camera may send a 1,000,000-dimensional image $< (x_1^{im}, t_1), ..., (x_n^{im}, t_n) >$ around 30 times/s. This situation is illustrated in Figure 1.5. You quickly find that this seemingly lowly question of data-formats is actually not so trivial.

Updating the entire network for each input event is impractically expensive (either in terms of run-time or energy use). Besides, it seems wasteful - the sensory context barely changes two consecutive gyroscope readings - why should we recompute it from scratch every time a new input arrives? An alternative would be to bin events into temporal windows and feed them in batches to our network. However, this sacrifices our ability to respond with low latency to important gyro signals - we have to wait until the next bin is processed to respond to a gyro signal.

The problem is that the current paradigm in deep learning involves doing an update on an entire network in one go - something that is not practical when the data arrives sequentially in small parts. What we want is a system where we could feed every bit of data into the network as it arrives, with the computational cost of the update proportional to the amount of new information in that data. In other words, we want to calculate how our new data affects the sensory state, without having to recompute the entire sensory context from scratch with each newly arriving bit of data. What we are missing is a principled way to only propagate changes when the changes are worth propagating.

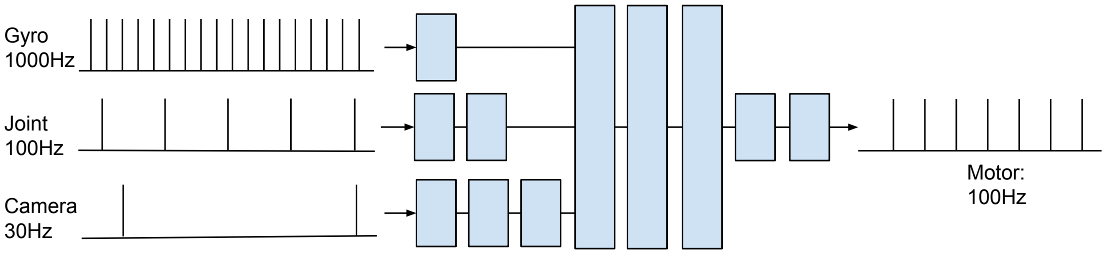


Figure 1.5.: Robots must integrate sensory signals from multiple sources into a global picture of the world. Globally updating the network with each Gyro signal would be impractically expensive, but binning signals over time and updating with the slowest sensor would sacrifice our ability to respond with low latency.

The data that the brain sees is an even more asynchronous than the data in our example above. Not only does each sensor send data asynchronously, but even within a single sensory stream, the data is comprised of a stream of asynchronous events (i.e. spikes). And unlike our robot's sensory signals, where each event comes tagged with some new data (like the current vector of linear and angular accelerations), the only information accompanying a spike is the time at which it arrives and the identity of the sensory neuron from which it came. The underlying signals must be inferred from the pattern in which spikes from different neurons arrive.

There are a few examples of artificial sensors that mimic this "event-based" paradigm of biological sensors. One is the Dynamic Vision Sensor [Lichtsteiner et al., 2008]. This is an event-based video camera, which, instead of sending a sequence of frames x_n at regular intervals of, say, 33ms, produces a sequence of events $(t_n, ij_n, \text{Polarity}_n)$, indicating the times when a pixel has brightened or darkened significantly (With ij_n indicating the (row,column) of the pixel activated in the $n'th$ event and $\text{Polarity}_n \in \{+1, -1\}$ indicating brightening or darkening). The event-rate varies with how much is changing in the scene; the sensor may produce thousands of events per second.

A spiking neural network is an extension of event-based signalling to communication within the network. In a standard deep network, an entire layer is updated in one step, and passes on its vector of activations to the next layer. In a spiking networks, neurons see a stream of input events (t_n, i_n), arriving at times t_n from input neurons i_n, and produce their own streams of events in response.

What we are missing is an idea of how to train such a network to do something useful.

1.6.2. Energy

As long as it is more advantageous to do more computation, people will keep building and running more powerful computers until they run into some limitation that makes it not worth it to build and run more powerful computers anymore.

Some 4 million years ago, a group of apes discovered that they were better off walking around on the ground than climbing in the trees, thus freeing up a perfectly good pair of hands. There are a lot of things that one can do with perfectly good hands, and one of the main limitations on the number of these things is the size of one's imagination. Suddenly there was an advantage to be gained by having a bigger imagination. Over the following years, brain size grew by a factor of 3. It is not clear from the data (see Figure 1.6) if this trend has stopped, but Henneberg [1988] suggest that the trend towards larger brains may have stopped and reversed around 20000 years ago, and that we've lost about a tennis ball's worth of grey matter since. It is not clear why this might be - theories range from the lower cognitive demands of civilized life to evolutionary discoveries in more efficient wiring. What seems fairly uncontroversial is that a significant downward pressure on brain size is energy consumption. Brains are not cheap organs to maintain. Despite comprising only 2% of human body mass, the brain consumes about 20% of the body's power - about 20W out of 100W - [Ling, 2001]. It may be the case that brains stopped growing because additional intelligence, while always a plus, stopped being enough of a plus to be worth the extra energy needed to maintain the larger brain to support it.

The same rules apply to artificial computation. Ultimately, it must provide enough economic value to pay the power bills. When looking for inspiration on energy-efficient machine learning, it may be worthwhile to examine the 20W supercomputer that lives in our heads - as evolution has had both plenty of incentive and plenty of time to optimize this machine.

While the brain is extremely power-hungry as far as human organs are concerned, it is extraordinarily power-lean for a computer. Compared to the human brain's 20W and 10^{11} neurons, a Titan X GPU running real-time object detection with YOLO [Redmon et al., 2016], a network of around 10^{7} much simpler neurons, consumes 250W.

There is no established way to measure the amount of computation that the brain does. Impacts [2015] analyze the issue and observe that, in large parallel supercomputers, the bottleneck to computation is not the aggregated number-crunching ability of all the processors in the machine, but rate at which these processors can communicate information internally within the machine. The reasoning is that as you scale up a

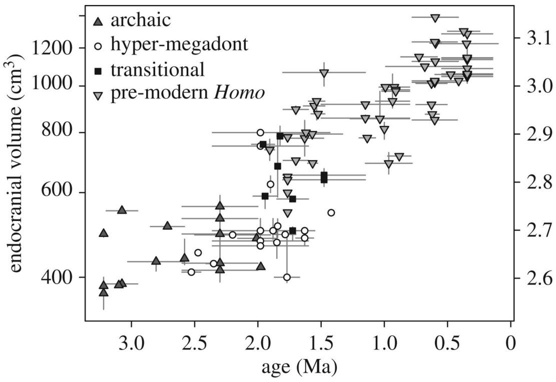


Figure 1.6.: The growth in estimated brain volume of human ancestors over the last 3 million years [Du et al., 2018].

computer, the number of processors scales linearly but the number of connections - and thus the aggregated communication between processors - must scale super-linearly if the machine is to remain a coherent whole. They suggest that Traversed Edges Per Second (TEPS) - a measure of internal information dissipation between processors used in supercomputers - may be the natural way to measure the computational capacity of the brain. Conveniently, it is possible to estimate the brains TEPS-performance by looking at the rate at which neurons send spikes to one another, how many neurons there are, and how many connected neighbours neurons tend to have. By the authors' estimate, the brain performs somewhere within the wide range of 1.8×10^{13} to 6.4×10^{14} TEPS, which would put its efficiency somewhere in the (admittedly wide) range of 9×10^{11} to 3.2×10^{13} TEPS/W.

The leader in non-wet efficient computation (as of 2015), in terms of TEPS/W, is the room-sized IBM Blue Gene/Q, which does 1.5×10^{13} TEPS - on par with the lower end of estimates for the human brain - but consumes some 3.95×10^{6} W, bringing it in at 3.8×10^{6} TEPS/W. So if Impacts [2015] are to be believed, the brain is somewhere within the range of 237,000 to 8,241,000 times more efficient than the most efficient supercomputer.

That hints that perhaps, at least for doing the type of computation that the brain does, we could make a lot more headway by improving algorithms and computational hardware than we could by firing up new power plants.

1.7. How to read this thesis

Grandiose introductions almost always lead to disappointing conclusions, and this thesis is no exception. We do not, to our satisfaction or to the satisfaction of any reasonable reader, fully solve any of the "Gaps" between our understanding of the brain and our understanding of artificial neural networks, outlined in Section 1.5. What we do do is explore ideas that we hope will provide fertile ground for further development, which will lead to future closure of said gaps.

In Chapter 2 we will review several past works that have contributed to closing the theoretical gap between these two types of networks. Chapters 3, 4, 5, 7, and 8 are papers that we have written. Chapter 3 is unpublished, and Chapters 4, 5, 7 and 8 were published in ICLR 2017, ICLR 2018, ICLR 2019, and AISTATS 2019 respectively. Each of these chapters begins with a short summary which ties the work into this thesis. In each summary, we attempt to convey the core ideas of the paper, and its link to this thesis, as fully as possible in under two pages. **For chapters 3, 4, 5, 7, and 8, readers wishing to make the best use of their 20W are encouraged to focus attention on the summary at the beginning of each chapter, and may feel free read the remainder of the chapter lightly or not at all, without missing out on anything critical.** The remainder of each chapter - the paper - is intended to be read as a stand-alone work which explores the ideas in detail.

Then there is Chapter 6, our most recent, and unpublished, work. It is the 6'th Chapter because it flows nicely from the work in Chapter 5. This is probably the most important chapter, because it explains how the STDP curve in Figure 1.4 arises.

Finally, in Section 9, we will discuss what we believe are the next steps are in this field.

2. Related Works

In this section re review past works in the machine learning literature which shed light on what biological networks may be doing. We will also cover other works that are more directly tied to our papers in the later chapters.

2.1. Boltzmann Machines

Boltzmann Machines [Ackley et al., 1985] are generative models based on stochastic, energy-based neural networks. They are the first way that people proposed to use a neural network to learn a probability distribution over data. Compared to more modern methods of learning probability distributions with neural network, they have quite a few biologically plausible features.

These models address a few points of biological implausibility in that they are trained without backpropagation (Gap 1: No Backprop), and they consist of units which communicate binary signals (Gap 2: Spiking). However, they do not meet the other criteria: They need to be trained on on IID data (Gap 3: Nonstationary Data), they rely on globally coordinated "positive" and "negative" training phases (Gap 4: Asynchronous Processing) and they require that the network have symmetric weight matrices (Gap 5: No Shared Parameters). Here we will briefly describe what a Boltzmann machine is and list extensions to it.

Like a Hopfield network (Hopfield [1982], see Section 1.1.2), a Boltzmann machines defines an energy over a network of neurons:

$$E(s,(w,b)) = -\sum_{ij} w_{ij} s_i s_j - \sum_i b_i s_i \tag{2.1}$$

Where s is the vector of states, $(w,b) := \theta$ are the synaptic-weights and biases, respectively. States are constrained to $s_i \in \{0,1\}$ and w is constrained by $w_{ij} = w_{ji} \in \mathbb{R}$, $w_{ii} = 0$. This energy function is used to define a probability distribution over states:

$$p_\theta(s) = \frac{e^{-E(s,(w,b))}}{\sum_{s'} e^{-E(s',(w,b))}} \tag{2.2}$$

Where $\sum_{s'}$ denotes a sum over all possible states of s. For a small network of 100 neurons with 2 possible states $s_i \in \{0, 1\}$, this sum would involve 2^{100} terms. Because computing this is wildly impractical, we reply on estimating this distribution though sampling. From Equations 2.1 and 2.2, we can derive the probability distribution of a single unit given all the rest as a sigmoidal function of the inputs:

$$p_\theta(s_i = 1|s_{\setminus i}) = \text{sigm}\left(\sum_j s_i w_{ij} + b_i\right) \tag{2.3}$$

Where $\text{sigm}(x) := (1 + e^{-x})^{-1}$. If we stochastically update units one-by-one (i.e. Gibbs sample) with $s_i \sim \text{Bernoulli}(p_\theta(s_i = 1|s_{\setminus i}))$, and observe the state of the network for a long time, we can show that the network is actually drawing samples from $p_\theta(s)$ as defined above.

In a Boltzmann machine with *hidden* units, neurons are divided up to disjoint subsets of visible units s_{vis} and hidden units s_{hid}. When training a generative model, the objective is to learn a model where the distribution over the states of the visible units $p_\theta(s_{\text{vis}})$ matches the data distribution $p_{data}(x)$. The Boltzmann Machine learning rule is derived minimizing the Kullback–Leibler Divergence between the data distribution and the model distribution:

$$KL(p_{data}||p_\theta) = \sum_{s'_{\text{vis}}} p_{data}(s'_{\text{vis}}) \log \frac{p_{data}(s'_{\text{vis}})}{p_\theta(s'_{\text{vis}})} \tag{2.4}$$

From here we can derive a simple update rule:

$$\frac{\partial KL(p_{data}||p_\theta)}{\partial \theta} = \overbrace{\mathbb{E}_{s'_{\text{vis}} \sim p_{data}, s'_{\text{hid}} \sim p_\theta(s_{\text{hid}}|s'_{\text{vis}})}\left[\frac{\partial E(s'_{\text{vis}}, s'_{\text{hid}})]}{\partial \theta}\right]}^{\text{positive phase}} - \overbrace{\mathbb{E}_{s'_{\text{vis}}, s'_{\text{hid}} \sim p_\theta(s'_{\text{vis}}, s'_{\text{hid}})}\left[\frac{\partial E(s'_{\text{vis}}, s'_{\text{hid}})]}{\partial \theta}\right]}^{\text{negative phase}} \tag{2.5}$$

Where in the *positive phase*, the visible units are clamped to the data the hidden states are sampled conditioned on these clamped visible units , and in the *negative phase*, the both the hidden and visible states are sampled conditioned on the model alone. For the energy function in Equation 2.1, this results in the very simple weight update rule:

$$\Delta w_{ij} \propto \overbrace{s_i^+ s_j^+}^{\text{positive phase}} - \overbrace{s_i^- s_j^-}^{\text{negative phase}} \tag{2.6}$$

Where s^+ and s^- are samples from the positive and negative phase, respectively. This learning rule is very appealing from a biological perspective, because, in the positive phase, it corresponds the *Hebbian* "fire-together, wire-together" rule that constituted the first observed learning rule in biological neurons by Hebb [2005]. The negative phase corresponds to an *anti-Hebbian* “fire together, wire-apart” rule, which has also been observed biologically Lamsa et al. [2007] (though it appears to be less widespread). Note that the synaptic update only depends on the activations of the pre- and post- synaptic neurons, as opposed to the activations of distant neurons at the top of the network, as is the case when training with backpropagation.

The problem with the above learning rule is that is can be very difficult to obtain these positive and negative samples, especially those from the negative phase. The energy landscape may be very multi-modal. The standard approach to acquiring samples - by the repeated application of 2.3 in a round-robin fashion, corresponds to Gibbs Sampling, a form of Markov-Chain Monte Carlo. When the energy landscape is complex, this may take many thousands of iterations to “burn-in” so that samples accurately represent the probability distribution that they’re trying to estimate. This means that each iteration of training requires many iterations of settling, making the training of Boltzmann machines impractical in all but the smallest of problems.

This problem was somewhat alleviated by *Restricted Boltzmann Machines* (RBMs) [Smolensky, 1986], where the connectivity of the Boltzmann Machine is constrained such that there are no direct connections between hidden units. This makes it very easy to sample in the positive phase, because hidden units become conditionally independent of one another, no iterative Gibbs-sampling is needed - all hidden units can be sampled in parallel. Hinton [2002] proposed a method called “Contrastive Divergence”, which greatly improved sampling efficiency of the negative phase. In Contrastive Divergence, the *negative-phase* samples, rather than being drawn from the model, which takes a very long time, are initialized at the *positive-phase* and run through the model for just a few iterations, which should make them *closer* to the model-distribution than those of the positive phase, causing the gradient to be biased but still point in the right direction. Hinton et al. [2006] further improved on this idea by showing that by sequentially stacking RBMs (by training one layer, freezing the weights, then training another on top, and so on), the resulting *Deep Belief Network* was guaranteed to improve the likelihood of the data under the model with each additional layer.

RBMs gradually fell out of favour in deep learning. Their two uses were to (A) pre-train the weights in deep feedforward networks before fine-tuning with backpropagation, and (B) as generative models. Use (A) became obsolete when Glorot et al. [2011] showed that once you use a rectified-linear activation function $h(x) = max(0, x)$ for your hidden units in a feedforward network, and initialize your weights with the right magnitude [Glorot and Bengio, 2010], there is no need for pre-trained parameters. Use (B) became obsolete when Kingma and Welling [2013] introduced the *Variational Autoencoder*, a generative model that was much easier (both faster and with fewer hyperparameters) to train and sample from than a Deep-Belief Network.

2.2. SpikeProp

Bohte et al. [2000] proposed an interesting variation on backpropagation to train a spiking network which they called *SpikeProp*. This work addresses the problem of how to train a spiking network, and so addresses Gap 2: Spiking. However it is still a form of backpropagation, and does not address the remaining gaps discussed in Section 1.5. In SpikeProp, the authors address the problem of how to train a spiking neural network to produce a particular sequence of input spikes and desired output spikes:

$$\left\langle t_j^i : i \in \text{InputNeurons}, t \in \mathbb{R}^+ \right\rangle$$
$$\left\langle t_j^d : j \in \text{OutputNeurons}, t \in \mathbb{R}^+ \right\rangle$$

This formulation is quite different from the classic vector-in, vector-out setup of most machine learning problems, so for most current datasets of interest, the data must be remapped to this *temporal sequence* form. Note that in this formulation, each neuron is allowed to fire once. The task is to optimize the firing time given the input sequence to match the output sequence. Here the authors propose to do this by minimizing the loss $\mathcal{L} = \sum_{j \in \text{OutputNeurons}} (t_j^d - t_j)^2$ between the actual and desired spike times, using neurons with an activation:

$$s_j(t) = \sum_{i \in \text{Inputs}} \sum_{k}^{K} w_{ij}^k \kappa(t - t_i - d_k)$$
$$t_j = \arg\min_t s_j(t) > \theta$$

Where $s_j(t)$ is the real-valued activation of neuron j, $\kappa(\tau)$ is a causal spike response kernel, for example $\kappa(\tau) = (\tau e^{-\tau}$ if $\tau > 0$ otherwise $0)$, each synapse has K associated delays d_k, and a weight w_{ij}^k is associated with each delay from each synapse. A neuron fires its spike t_j when its activation $s_j(t)$ first crosses a threshold θ.

The main insight of SpikeProp is that one can find the gradients $\frac{\partial \mathcal{L}}{\partial w_{ij}^k}$ by observing that, for a small window around the time of a spike t_j, the spike time varies linearly with the parameter w_{ij}^t. The authors use this observation to derive an update rule wherein for each output spike, a backward pass is performed and parameters are adjusted to bring that spike closer in time to the target spike. However, SpikeProp remains biologically implausible in a number of ways: It still relies on a secondary signalling mechanism which propagates a gradient backwards across synapses; it assumes neurons only spike once; it assumes coordinated *training iteration* rather than a constantly changing stream of inputs; and it requires that a neuron is already spiking in the vicinity of the target signal to learn.

2.3. Target Propagation

Target propagation is an alternative approach to credit assignment from backpropagation. Instead of passing back a *gradient* (the direction that a neuron's activation must move in order to minimize the loss), the network passes back a *target* (a desired activation for each hidden neuron, such that if it had had that activation on the forward pass, the loss would be lower). Thus it addresses Gap 1: No Backprop. The catch here is that we no longer have a guarantee that our updates will minimize the loss - we just hope they will be in the right direction. As the network gets very deep this may no longer be true. An advantage of target propagation is that it does not assume that the network is a continuous, differentiable function, so it can in principle work in neural networks with discrete activation functions, partially addressing Gap 2: Spiking. Furthermore, the top-down weights of the target-propagating network need not be symmetric with the feedforward network, addressing Gap 5: No Shared Parameters. Target propagation, however, is not a full solution to biologically plausible learning. It still assumes IID data - not addressing Gap 3: Nonstationary Data; and assumes a synchronized forward/backward pass - not addressing Gap 4: Asynchronous Processing.

The trick to making target-propagation work well is in figuring out how to assign targets to hidden neurons that actually lead to the minimization of the final loss. Lee et al. [2015] proposed an efficient way of doing target propagation called *Difference Target Propagation*. Suppose one has a deep network with layer-functions $z_l = f_{\theta_l}(z_{l-1})$, starting from an input layer $z_0 = x$. Suppose the supervision comes in the form of a *target* activation for the final layer: $\hat{z}_L$. If each layer were invertible, then we could find a target for the second last layer: $\hat{z}_{L-1} = f_{\theta_L}^{-1}(\hat{z}_L)$, such that had z_{L-1} been a little bit closer to $\hat{z}_{L-1}$, the loss $\|z_L - \hat{z}_L\|$ would have been lower. Parameters θ_{L-1} can then be adjusted to minimize $\|z_{L-1} - \hat{z}_{L-1}\|$. Since layers are not, in general, invertible, target propagation relies on training an approximate inverse $\hat{z}_{l-1} := g_{\phi_l}(\hat{z}_l) \approx f_{\theta_l}^{-1}(\hat{z}_l)$. Since f^{-1} may not be realizable by g (f may not even be injective), this *target-propagation* is imperfect. *Difference Target Propagation* applies a linear correction term, to calculate targets as $\hat{z}_{l-1} = z_{l-1} + g(\hat{z}_l) - g(z_l)$. The authors show that this greatly reduces the error arising from the imperfect inverse and leads to a clear improvement in target propagation.

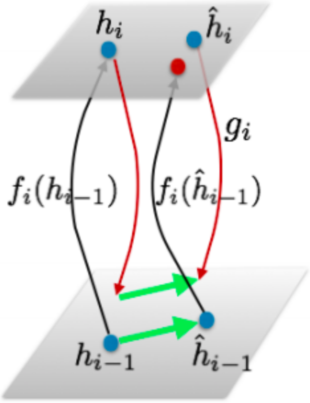


Figure 2.1.: A visual explanation from Lee et al. [2015] of how targets are propagated down from a higher to a lower layer in Difference Target propagation. (Note that their notation uses h_i to represent the activation of the i'th layer as opposed to the z_l convention used in this thesis.)

Target Propagation is more biologically plausible than backpropagation, because neurons only need emit *one kind* of signal - an activation. It does not rely on a secondary signalling mechanism (unobserved in biology) wherein neurons send loss-derivatives backwards through multiple layers of synapses.

2.4. Neural ODE's

Backpropagation is a way to assign credit when the computation takes the form of a directed acyclic graph (where nodes correspond to differentiable functions and edges correspond to variables). Because of this, artificial neural networks have always been awkward in handling continuous time. The common approach has been to discretize time into steps, and assign gradients with Backpropagation-Through time. Temporal-discretization throws away potentially useful information, forcing a trade-off between efficiency (large bins, much information loss) and accuracy (small bins with little information loss).

In *Neural Ordinary Differential Equations*, Chen et al. [2018] propose a novel learning algorithm which shows how continuous-time neural networks could be trained. The authors started with the observation that the standard form of a layer-update for ResNet [He et al., 2016] takes the form $z_l := z_{l-1} + f(z_{l-1})$. Replace the layer-index with a time-step index and this looks like an Euler-discretization $\frac{z_t - z_{t-1}}{\Delta t} \approx f(z_t)$ of a continuous-time dynamical system $\frac{\partial z}{\partial t} = f(z)$ with a step-size $\Delta t = 1$. Euler discretization is known to be the simplest and often worst method of discretizing a dynamical system. The authors propose instead taking the equivalent dynamical system and passing it to a black-box ODE solver (which uses more complex methods which more closely simulate the continuous-time dynamics). They further show how one can design an augmented dynamical system $\frac{\partial a}{\partial t} = g(a)$ which computes the gradients in a backwards pass. This can likewise be passed to a black-box ODE-solver to come up with close approximations.

The idea is interesting to biologically plausible learning because it shows how one could actually train a network that lives in continuous-time in a principled way.

Neural ODE's is an interesting idea with many potential applications. But it is likely not the answer to our question of how biological networks learn. The reason is that it still requires doing a forward and backward pass through time (thus not addressing Gap 1: No Backprop). It is difficult to see how a brain could run a dynamical system of neurons, compute a loss, then run another "backwards" system with the same parameters in reverse to assign credit. Moreover, such a system is inherently episodic - there has to be some "end" point in time at which the loss is computed and the backward dynamics start running (thus not addressing Gap 4: Asynchronous Processing). It's difficult to see how this could be used in online learning, where input arrives continuously.

2.5. Unbiased Online Recurrent Optimization

We have already discussed the fact that backpropagation-through time (BPTT) on a recurrent network $z_t = f_\theta(z_{t-1}, x_t)$ is both inefficient and biologically implausible. The inefficiency is because we must, for every learning iteration, backpropagate through T time-steps, where T is the temporal horizon over which we would like to learn temporal dependencies. The biological implausibility is because in order to do this, we need to keep a buffer of T past-activations for every neuron, whereas biological neurons appear to have no buffering mechanism to keep track of past-states. We would also need to transmit signals backwards though axons, as in normal backpropagation.

There are two domains in which we can consider doing BPTT: (A) The *episodic* domain, where we are presented with short sequences of inputs and must make some prediction at the end of each sequence, and (B) The *online* domain, where the input arrives as a single, infinite stream and we must learn to predict the next element of the stream given the entire past (it may not be known in advance how much of the past is relevant). In case (A) we simply execute a forward pass and then a backward pass on the sequence. In case (B) we must choose a time-horizon of T-steps over which we want to backpropagate. In this case, we have a tricky decision to make: In one limit, we could chose infinite T, to get a correct gradient at the cost of slower and slower computation; As time goes on we have to backpropagate through more steps to get to the beginning of the growing sequence. In the other limit, we can chose a short T, and accept that our gradients are biased, so that we are unable to learn temporal dependencies longer than T-steps. If we want to update our parameters at every time-step, learning requires T-times as much computation as inference, because at every step $z_t = f_\theta(z_{t-1}, x_t)$ we must backpropagate through T steps to find our T-step approximation to the loss-gradient $\frac{\partial \mathcal{L}(z_t)}{\partial \theta}$.

Unbiased Online Recurrent Optimization (UORO) [Tallec and Ollivier, 2017] proposes an imperfect solution to this conundrum. The authors start from the observation that there already exists a way to compute the gradients of the parameters of neural networks with infinite time-horizon without doing backprop-through-time. It's called Real Time Recurrent Learning (RTRL) [Williams and Zipser, 1989] and it's extremely expensive.

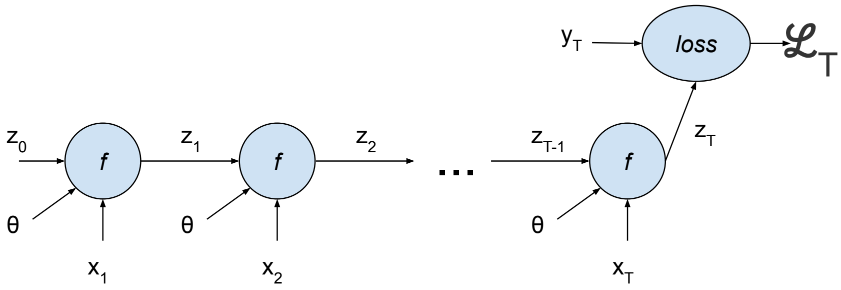


Figure 2.2.: An illustration of the computational graph of a recurrent network.

A recurrent network has the form $z_t = f_\theta(x_t, z_{t-1})$ where z_t is the hidden activation at time t and x_t is the input. Suppose we get a loss $\mathcal{L}_T(z_T, y_T)$ at some time-step T and want to figure out how to update parameters θ to optimize it. θ affects the loss through many paths (one per time-step) so its gradient will be the sum of many components:

$$\frac{d\mathcal{L}_T}{d\theta} = \frac{d\mathcal{L}_T}{dz_T}\frac{dz_T}{d\theta} = \frac{d\mathcal{L}_T}{dz_T}\sum_{t=1}^{T}\frac{\partial z_T}{\partial z_t}\left.\frac{\partial z_t}{\partial \theta}\right|_{z_{t-1}} \tag{2.7}$$

Where the notation $\frac{\partial z_t}{\partial \theta}|_{z_{t-1}}$ means "partial derivative of z_t with respect to θ, with z_{t-1} held constant". We can compute this gradient by Backpropagation-through time (BPTT) - i.e. using the observation that $\frac{\partial \mathcal{L}_T}{\partial z_{t-1}} = \frac{\partial \mathcal{L}_T}{\partial z_t}\frac{\partial z_t}{\partial z_{t-1}}$. The problem, of course, is that as T gets larger, so does the time required to compute this gradient.

Observe the gradient $\frac{\partial \mathcal{L}_T}{\partial z_t}$ can be decomposed as $\frac{\partial \mathcal{L}_T}{\partial z_t} = \frac{\partial \mathcal{L}_T}{\partial z_T}\frac{\partial z_T}{\partial z_{T-1}}...\frac{\partial z_{t+1}}{\partial z_t}$. BPTT corresponds to crunching these terms left-to-right - backwards through time. But we could also crunch then right-to-left, and this is the idea behind RTRL. In RTRL, we apply the update

$$\frac{dz_t}{d\theta} = \frac{\partial z_t}{\partial z_{t-1}}\frac{dz_{t-1}}{d\theta} + \left.\frac{\partial z_t}{\partial \theta}\right|_{z_{t-1}} \tag{2.8}$$

The problem here is that this involves the multiplication of a $|z| \times |z|$ and a $|z| \times |\theta|$ matrix, which is a hugely expensive operation. In a fully connected simple recurrent network where $z_t = h(z_{t-1} \cdot w_{zz} + x_t \cdot w_{xz})$, the cost of the forward pass is $\mathcal{O}(|z| \times |z|)$, while the RTRL gradient update in Equation 2.8 costs $\mathcal{O}(|z| \times |z| \times |\theta|)$. Since the network is fully connected, $|\theta| = |z| \times |z| + |z| \times |x|$, so for a small network of 100 neurons and 100 inputs, the RTRL gradient update would cost $|\theta| = 20000$ times more than the normal forward pass!

UORO is a trick for computing an unbiased approximation to $\frac{\partial z_t}{\partial \theta}$, by using random projections to compress the gradient information from a $|z| \times |\theta|$ matrix to an outer product of a $|z|$ and a $|\theta|$ vector: $\frac{dz_t}{d\theta} \approx \tilde{z}_t \otimes \tilde{\theta}_t$. The update of this approximation at each step only costs $\mathcal{O}(|z_t| \times |z_{t-1}|)$ - the same as a step of the forward pass. However, this comes at a cost: The approximation, while unbiased, may have a large variance, and this variance slows down learning to the point where it appears to be impractical for implementation on real problems. UORO partially addresses Gap 1: No Backprop by not requiring backprop-through time. However it still requires the neurons emit a secondary gradient signal - it just propagates this in the forward direction. It can also run on nonstationary data, addressing Gap 3: Nonstationary Data, and, because it does not require running a backward pass through time, it is may help to address how we could train an asynchronous system, addressing Gap 4: Asynchronous Processing. The idea is interesting and possibly could be extended into a biologically plausible learning rule for training neural networks, but at present we cannot see how.

We tried and failed to improve upon UORO by replacing the random projections with deterministic pseudorandom projections which should converge to the mean more quickly. Our approach was not well theoretically founded, and did not work empirically, so we abandoned it.

Recently, Cooijmans and Martens [2019] have explored other methods of reducing the variance of the gradient-approximation made by UORO, and drawn a connection to REINFORCE. However, there has yet to be a convincing demonstration of good empirical results from this method. The topic of online learning using approximate forward-mode differentiation remains an open and interesting one for the future.

2.6. Synthetic Gradients

Jaderberg et al. [2016] pointed out a problem with the standard backprop-trained deep networks that they called “locking”. Suppose some layer in a deep network computes its output given the state of the previous layer: $z_l = f_{\theta_l}(z_{l-1})$. In order to update that layer's parameters θ_l, we must first compute the rest of the activations in the network $z_{l+1}, ..., z_L$, then find the loss $\mathcal{L} = \ell(z_L, y)$, then sequentially compute the layer-gradients with backprop $\frac{\partial \mathcal{L}}{\partial z_L}, ..., \frac{\partial \mathcal{L}}{\partial z_l}$ before finally computing the parameter gradient $\frac{\partial \mathcal{L}}{\partial \theta_l} = \frac{\partial \mathcal{L}}{\partial z_l}\frac{\partial z_l}{\partial \theta_l}$. Until that gradient arrives, the neurons at layer l must remember the state of their inputs so that they can compute $\frac{\partial \mathcal{L}}{\partial \theta_l}$. This requirement can become onerous when there is a long chain of connections between z_l and the final layer z_L. This problem is especially evident two settings:

1. **Online Streaming Data**. As discussed previously in Section 2.5, when the data

comes in the form of an infinite temporal stream, it becomes intractable to assign credit to parameters. We would have to wait until the end of time to compute the loss which should be used to update a parameter.

2. **Distributed Systems**, where one "shared" network feeds data into several "client" networks running asynchronously. In order to update properly, the shared network would be "locked" until it received the backpropagating signal from every client network. It thus would be constrained to the speed of the slowest client.

The approach that Jaderberg et al. [2016] use to solve this is to train an auxiliary model to predict the backpropagating gradient: $\frac{\partial \mathcal{L}}{\partial z_l} \approx \widehat{\frac{\partial \mathcal{L}}{\partial z_l}} := g_{\phi_l}(z_l)$, and use this "synthetic" predicted gradient to update parameters. This of course just pushes the problem back to how to train the gradient-prediction parameters ϕ_l - it could be trained to minimize $\|\widehat{\frac{\partial \mathcal{L}}{\partial z_l}} - \frac{\partial \mathcal{L}}{\partial z_l}\|$, but then we need to wait for the true $\frac{\partial \mathcal{L}}{\partial z_l}$, which is exactly what we were trying to avoid. Instead, we could 'bootstrap' from the next layer's gradient approximation, and minimize the gradient-prediction error given the next layer's prediction of the gradient $\|\widehat{\frac{\partial \mathcal{L}}{\partial z_l}} - \widehat{\frac{\partial \mathcal{L}}{\partial z_{l+1}}} \frac{\partial z_{l+1}}{\partial z_l}\|$, until finally the second-last layer's parameters ϕ_{L-1} learn to minimize the true-gradient prediction error: $\|\widehat{\frac{\partial \mathcal{L}}{\partial z_{L-1}}} - \frac{\partial \mathcal{L}}{\partial z_L} \frac{\partial z_L}{\partial z_{L-1}}\|$.

Unlike in Unbiased Online Recurrent Optimization (see Section 2.5), this procedure introduces a bias in training. It is unclear how much this matters in practice on real-life problems.

2.7. Equilibrium Propagation

Equilibrium Propagation [Scellier and Bengio, 2017] is an algorithm for gradient-based training of deep networks, wherein neurons may only emit one kind of signal - an activation - and do not have the capacity to send a secondary gradient signal backwards. It thus addressed the problem of how to do gradient-based training of a deep network without backprop (Gap 1: No Backprop).

The algorithm can be used to train an Energy-Based Model [Hopfield, 1984] for classification. The network performs inference by iteratively converging to a fixed-point, conditioned on the input data, and taking the state of the output neurons at the fixed point to be the prediction of the network. The network's dynamics are defined by an energy function over neuron states s and parameters $\theta = (w, b)$:

$$E_\theta(s, x) = \frac{1}{2}\sum_{i \in \mathcal{S}} s_i^2 - \sum_{i \in S} b_i \rho(s_i) - \sum_{j \in \mathcal{S}, i \in \alpha_j \cap \mathcal{S}} w_{ij}\rho(s_i)\rho(s_j) - \sum_{j \in \mathcal{S}, i \in \alpha_j \cap \mathcal{I}} x_i w_{ij} \rho(s_j) \quad (2.9)$$

Where $\mathcal{I}$ is the set of input neuron indices, $\mathcal{S}$ is the set of non-input neuron indices; $s \in \mathbb{R}^{|\mathcal{S}|}$ is the vector of neuron states; where $\alpha_j \subset \{\mathcal{I} \cup \mathcal{S}\}$ is the set of neurons connected to neuron j; x denotes the input vector; and ρ is some nonlinearity; w is a weight matrix with a symmetric constraint: $w_{ij} = w_{ji}$, and entries only defined for $\{w_{ij} : i \in \alpha_j\}$ The state-dynamics for non-input neurons, derived from Equation 2.9, are:

$$\frac{\partial s_j}{\partial t} = -\frac{\partial E_\theta(s,x)}{\partial s_j} = -s_j + \rho'(s_j)\left(b_j + \sum_{j\in\mathcal{S}, i\in\alpha_j\cap\mathcal{S}} w_{ij}\rho(s_i) + \sum_{j\in\mathcal{S}, i\in\alpha_j\cap\mathcal{I}} w_{ij}x_i\right) \forall j \in \mathcal{S} \tag{2.10}$$

The network is trained using a two-phase procedure, with a negative and then a positive phase, as illustrated in Figure 2.3. In the negative phase, the network is allowed to settle to an energy minimum $s^- := \arg\min_s E_\theta(s,x)$ conditioned on a minibatch of input data x. In the positive phase, a target y is introduced, and the energy function is augmented to “perturb” the fixed-point of the state towards the target with a “clamping factor” β: $E_\theta^\beta(s,x,y) = E_\theta(s,x) + \beta C(s_\mathcal{O}, y)$, where β is a small scalar and $C(s_\mathcal{O}, y)$ is a loss defined between the output neurons in the network and the target y (we use $C(s_\mathcal{O}, y) = \|s_\mathcal{O} - y\|_2^2$). The network is allowed to settle to the perturbed state $s^+ := \arg\min_s E^\beta(s,x,y)$.

Finally, the parameters of the network are learned based on a contrastive loss between the negative-phase and positive-phase energy, which can be shown to be proportional to the gradient of the output loss $\frac{\partial C(s_\mathcal{O}^-, y)}{\partial\theta}$ in the limit of $\beta \to 0$:

$$\Delta\theta = -\frac{\eta}{\beta}\left(\frac{\partial E_\theta^\beta(s^+,x,y)}{\partial\theta} - \frac{\partial E_\theta(s^-,x)}{\partial\theta}\right) \propto -\frac{\partial C(s_\mathcal{O}^-, y)}{\partial\theta} \tag{2.11}$$

Where η is some learning rate; $\mathcal{O} \subseteq \mathcal{S}$ is the subset of output neurons. This results in a local learning rule, where parameter changes only depend on the activities of the pre- and post-synaptic neurons:

$$\Delta w_{ij} = \frac{\eta}{\beta}\left(\rho(s_i^+)\rho(s_j^+) - \rho(s_i^-)\rho(s_j^-)\right) \tag{2.12}$$

$$\Delta b_i = \frac{\eta}{\beta}\left(\rho(s_i^+) - \rho(s_i^-)\right) \tag{2.13}$$

Where x_i will be substituted for $\rho(s_i)$ when i is an input unit. Intuitively, the algorithm works by adjusting θ to pull $s^- := \arg\min_s E_\theta(s,x)$ closer to $s^+ : \arg\min_s E_\theta^\beta(s,x,y)$ so that the network will gradually learn to naturally minimize the output loss $C(s_\mathcal{O}^-, y)$ associated with the energy-minimum s^-.

Note that the synaptic learning rule is purely local, and that neurons only emit one kind of signal, so the network is indeed doing gradient-based learning without backprop, addressing Gap 1: No Backprop. However, the algorithm depends on detecting small changes to the fixed-point of a network, and thus cannot not use quantization, failing Gap 2: Spiking (a matter we address in Chapter 8). Further, it requires that input be held static while the network converges, thus failing Gap 3: Nonstationary Data and Gap 4: Asynchronous Processing, and it assumes symmetric weights, failing Gap 5: No Shared Parameters[1].

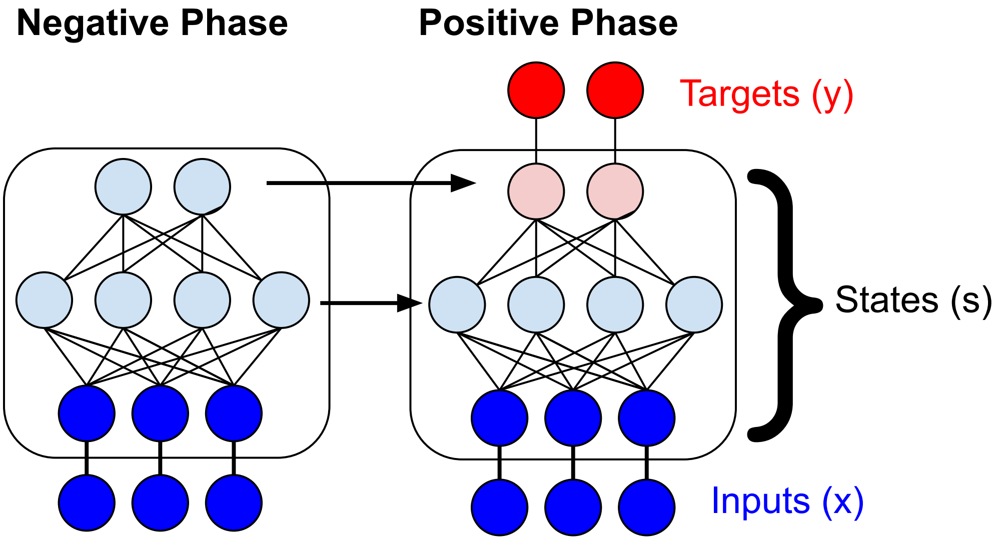


Figure 2.3.: An illustration of the two phases of Equilibrium Propagation. **Left:** In the negative phase, the input is presented and the network is allowed to settle to an energy minimum. **Right:** In the positive phase, the network starts at the steady state of the negative phase, and is perturbed by pulling the output units weakly towards the target, causing the entire network to settle to a new steady state. The parameters are then updated based on the contrast between these two steady states, according to Equation 2.12

.

2.7.1. Relation to Dynamics-Based Learning and STDP

One idea, inspired by Xie and Seung [2000] and Hinton [2012], of how biological neurons might be learning is that they substitute the dynamics for the gradient in their learning rule:

$$\frac{\partial w_{ij}}{\partial t} \propto \rho(s_i)\frac{\partial \rho(s_j)}{\partial t} \tag{2.14}$$

[1] However as described in Scellier et al. [2018], the symmetry requirement can be relaxed without significantly impacting performance. In this case there is no energy function, but one simply defines the network in terms of the state dynamics. The network nevertheless seems to learn to settle to fixed-points rather than falling into limit cycles or chaotic dynamics. The reason for this phenomenon is not well understood.

This implies that if the dynamics are moving in such a way as to minimize a desired loss, the weight update should then also minimize that loss. Bengio et al. [2015b] show that this should correlate to the STDP learning rule observed in biology - a connection that we discuss in detail in Chapter 6.

As Scellier et al. [2018] pointed out, Equilibrium propagation implicitly applies this dynamics-based learning rule during the positive phase. During the positive phase, the states have the average trajectory $\overline{\frac{\partial s}{\partial t}} := \frac{s^+ - s^-}{\Delta T}$. Now, when the difference between the states of the two phases is small (i.e. in the limit of $\beta \to 0$), we can see the relation between the dynamics-based learning of Equation 2.14 and the Equilibrium Propagation update of Equation 2.12:

$$\Delta w_{ij}^{rate} \propto \int_{t=T^-}^{T^+} \rho(s_i) \frac{\partial \rho(s_j)}{\partial t} \approx \rho(s_i^-)(\rho(s_j^+) - \rho(s_j^-)) \approx \rho(s_i^+)\rho(s_j^+) - \rho(s_i^-)\rho(s_j^-) \quad (2.15)$$

Thus we can say that the contrastive update rule of Equilibrium Propagation is approximately equivalent to applying dynamics-based learning just during the course of the positive phase of training.

This is of course a somewhat artificial scenario. It is not particularly biologically plausible that a network would be allowed to settle to a fixed point, then perturbed very slightly by a target and allowed to settle again, with learning switched on just during that second settling phase (Gap 4: Asynchronous Processing). Clearly, the story is not complete. However the underlying notion - That the network's dynamics can take the place of a back-propagated gradient in assigning credit to network parameters - is intriguing and merits further investigation.

3. Deep Spiking Networks

In our first work, we aim to address Gap 2: Spiking, and gap Gap 4: Asynchronous Processing, by constructing a network of spiking neurons and demonstrating that it can be trained using an approximate form of backpropagation. Our goal in this work is to train a multi-layer network for classification in a setting where neurons are not thought of as functions, but as *processes* which consume and produce *events* (i.e. spikes) in sequence. In our formulation, neurons are implemented in an *event-based* manner: Neurons update their state upon receiving an event. Each received event may or may not trigger the neuron to fire an event of its own. Under this formulation, the amount of computation does not simply scale with the size of the network, as it would in a regular deep net, but with the number of events communicated between neurons. Our network is *Asynchronous* in the sense that there is no globally coordinated forward and backward pass. The forward and backward passes run in parallel, with neurons sending out spikes once their activations cross the threshold. Our goal in this work is simply to show that we can train a deep network out of such building blocks.

Our system first converts an input vector $v \in \mathbb{R}^D$ into a "signed spike" sequence $\langle (i_n, s_n) : n \in \mathbb{N}, i_n \in [1..D_{\text{input}}], s_n \in \{-1, +1\} \rangle$ to be processed by the network. For the conversion, we consider each element of the vector to be a "firing rate" which determines the rate of change of potential ϕ_i, so that $\frac{\partial \phi_i}{\partial t} = v_i$. When $|\phi_i|$ crosses some threshold, arbitrarily defined at $\frac{1}{2}$, the spike $(i_n, (s_n := \text{sign}(\phi_i)))$ is appended to the input sequence, and the potential ϕ_i is decremented by s_n. Note that we introduce a notion of time here only to explain how a vector is converted to a spike-sequence; time has no role in the update rules of the network.

The network that consumes this spike sequence consists of a *forward* and *backward* network. In the forward network, first-layer neurons j accumulate input spikes one-by-one into their potentials: $\phi_j \leftarrow \phi_j + w_{i_n j} s_n$. Unlike the input, these neurons have only a positive threshold and emit only positive spikes - making them event-based versions of Rectified-Linear neurons. If a neuron j is pushed past the threshold ($\phi_j > \frac{1}{2}$), the potential ϕ_j is decremented by 1 and the spike is routed to the next layer to be processed in a similar manner *before* any further input spikes are processed.

At the top, error is transformed into a sequence in a similar manner to the input, and fed into the backward network. Backward neurons work similarly to the forward, except that they (since they must communicate gradients) emit both "positive" and "negative" spikes (which is not biologically plausible). and to implement backprop correctly, they must be "gated" such that they are disabled when their corresponding forward-units have received net-negative input.

Every time a forward neuron j or a backward neuron k spikes, the parameter ϕ_{jk} is updated. Thus many update to a single parameter can occur during a single training iteration.

While our creation does not show any dramatic performance improvements as compared to a traditional deep network, it does have some interesting characteristics that separate it from more traditional deep network implementations, and have potential to be beneficial:

1. **Early Guessing** - from the time of the first spike of the penultimate layer, we already have an early guess as to what the final activation of the last layer is going to be. This spike may come well before all the input neurons have spiked. Our network, rather than computing its function in one go, gradually converges to a solution as more and more spikes are propagated through the network.

2. **Many Updates Per Iteration** - In deep learning, stochastic gradient descent tends to outperform batch gradient descent because it is better to make many approximate-but low-cost parameter updates than one big and accurate one. Similarly, in our configuration we can update each parameter every time a spike occurs on a backward neuron. Thus even within a single training iteration, there may be many updates to the value of a parameter.

This work does not address the other “gaps” between biological and artificial neural networks. Our algorithm is simply an approximation to backpropagation, so it fails to pass the Gap 1: No Backprop test. Although inputs are processed as a sequential stream, the input rates are fixed and the network is allowed to run for a set amount of time as it processes the input, so it cannot be said that we are learning on a temporal data stream, and therefore we do not pass Gap 3: Nonstationary Data. Furthermore, because we are approximately implementing backprop, connections used in the backward pass are the same as those used for the forward pass, which is a form of weight-tying (Gap 5: No Shared Parameters).

3.1. Abstract

We introduce an algorithm to do backpropagation on a spiking network. Our network is "spiking" in the sense that our neurons accumulate their activation into a potential over time, and only send out a signal (a “spike”) when this potential crosses a threshold and the neuron is reset. Neurons only update their states when receiving signals from other neurons. Total computation of the network thus scales with the number of spikes caused by an input rather than network size. We show that the spiking Multi-Layer Perceptron behaves identically, during both prediction and training, to a conventional deep network of rectified-linear units, in the limiting case where we run the spiking network for a long time. We apply this architecture to a conventional classification problem (MNIST) and

achieve performance very close to that of a conventional Multi-Layer Perceptron with the same architecture. Our network is a natural architecture for learning based on streaming event-based data, and is a stepping stone towards using spiking neural networks to learn efficiently on streaming data.

3.2. Introduction

In recent years the success of Deep Learning has proven that a lot of problems in machine-learning can be successfully attacked by applying backpropagation to learn multiple layers of representation. Most of the recent breakthroughs have been achieved through purely supervised learning.

In the standard application of a deep network to a supervised-learning task, we feed some input vector through multiple hidden layers to produce a prediction, which is in turn compared to some target value to find a scalar cost. Parameters of the network are then updated in proportion to their derivatives with respect to that cost. This approach requires that all modules within the network be differentiable. If they are not, no gradient can flow through them, and backpropagation will not work.

An alternative class of artificial neural networks are Spiking Neural Networks. These networks, inspired by biology, consist of neurons that have some persistent "potential" which we refer to as ϕ, and alter each-others' potentials by sending "spikes" to one another. When unit i sends a spike, it increments the potential of each downstream unit j in proportion to the synaptic weight $W_{i,j}$ connecting the units. If this increment brings unit j's potential past some threshold, unit j sends a spike to its downstream units, triggering the same computation in the next layer. Such systems therefore have the interesting property that the amount of computation done depends on the contents of the data, since a neuron may be tuned to produce more spikes in response to some pattern of inputs than another.

In our flavour of spiking networks, a single forward-pass is decomposed into a series of small computations provide successively closer approximations to the true output. This is a useful feature for real time, low-latency applications, as in robotics, where we may want to act on data quickly, before it is fully processed. If an input spike, on average, causes one spike in each downstream layer of the network, the average number of additions required per input-spike will be $\mathcal{O}(\sum_{l=1}^{L} N_l)$, where N_l is the number of units in the layer l. Compare this to a standard network, where the basic messaging entity is a vector. When a vector arrives at the input, full forward pass will require $\mathcal{O}(\sum_{l=1}^{L}(N_{l-1} \cdot N_l))$ multiply-adds, and will yield no "preview" of the network output.

Spiking networks are well-adapted to handle data from event-based sensors, such as the Dynamic Vision Sensor (a.k.a. Silicon Retina, a vision sensor) Lichtsteiner et al. [2008] and the Silicon Cochlea (an audio sensor) Chan et al. [2007]. Instead of sending out samples at a regular rate, as most sensors do, these sensors asynchronously output events when there is a change in the input. They can thus react with very low latency to sensory events, and produce very sparse data. These events could be directly fed into our spiking network (whereas they would have to be binned over time and turned into a vector to be used with a conventional deep network).

In this paper, we formulate a deep spiking network whose function is equivalent to a deep network of Rectified Linear (ReLU) units. We then introduce a spiking version of backpropagation to train this network. Compared to a traditional deep network, our Deep Spiking Network has the following advantageous properties:

1. Early Guessing. Our network can make an "early guess" about the class associated with a stream of input events, before all the data has been presented to the network.
2. No multiplications. Our training procedure consists only of addition, comparison, and indexing operations, which potentially makes it very amenable to efficient hardware implementation.
3. Data-dependent computation. The amount of computation that our network does is a function of the data, rather than the network size. This is especially useful given that our network tends to learn sparse representations.

The remainder of this paper is structured as follows: In Section 3.3 we discuss past work in combining spiking neural networks and deep learning. In 3.4 we describe a Spiking Multi-Layer Perceptron. In 3.5 we show experimental results demonstrating that our network behaves similarly to a conventional deep network in a classification setting. In 3.6 we discuss the implications of this research and our next steps.

3.3. Related Work

There has been little work on combining the fields of Deep Learning and Spiking neural networks. The main reason for this is that there is not an obvious way to backpropagate an error signal through a spiking network, since output is a stream of discrete events, rather than smoothly differentiable functions of the input. Bohte et al. [2000] proposes a spiking deep learning algorithm - but it involves simulating a dynamical system, is specific to learning temporal spike patterns, and has not yet been applied at any scale. Buesing et al. [2011] shows how a somewhat biologically plausible spiking network can be interpreted as an MCMC sampler of a high-dimensional probability distribution. Diehl et al. [2015] does classification on MNIST with a deep event-based network, but training is done with a regular deep network which is then converted to the spiking domain. A similar approach was used by Hunsberger and Eliasmith [2015] - they came up with a continuous unit which smoothly approximated the the firing rate of a spiking neuron, and did backpropagation on that, then transferred the learned parameters to a spiking network. Neftci et al. [2013] came up with an event-based version of the contrastive-divergence algorithm, which can be used to train a Restricted Boltzmann Machine, but it was never applied in a Deep-Belief Net to learn multiple layers of representation. O'Connor et al. [2013] did create an event-based spiking Deep Belief Net and fed it inputs from event-based sensors, but the network was trained offline in a vector-based system before being converted to run as a spiking network.

Spiking isn't the only form of discretization. Courbariaux et al. [2015] achieved impressive results by devising a scheme for sending back an approximate error gradient in a deep neural network using only low-precision (discrete) values, and additionally found that the discretization served as a good regularizer. Our approach (and spiking approaches in general) differ from this in that they sequentially compute the inputs over time, so that it is not necessary to have finished processing all the information in a given input to make a prediction.

3.4. Methods

In Sections 3.4.1 to3.4.3 we describe the components used in our model. In Section 3.4.5 we will use these components to put together a Spiking Multi-Layer Perceptron.

3.4.1. Spiking Vector Quantization

The neurons in the input layer of our network use an algorithm that we refer to as Spiking Vector Quantization (Algorithm 1) to generate "signed spikes" - that is, spikes with an associated positive or negative value. Given a real vector: $\vec{v}$, representing the input to an array of neurons, and some number of time-steps T, the algorithm generates a series of N signed-spikes: $\langle (i_n, s_n) : i_n \in [1..len(\vec{v})], s_n \in \{\pm 1\}, n \in [1..N] \rangle$, where N is the total number of spikes generated from running for T steps, i_n is the index of the neuron from which the n'th spike fires (note that zero or more spikes can fire from a neuron within one time step), $s_n \in \{\pm 1\}$ is the sign of the n'th spike.

In Algorithm 1, we maintain an internal vector of "neuron potentials" $\vec{\phi}$. Every time we emit a spike from neuron i we subtract s_i from the potential ϕ_i until $\vec{\phi}$ is in the interval bounded by $(-\frac{1}{2}, \frac{1}{2})^{len(\vec{v})}$. We can show that as we run the algorithm for a longer time (as $T \to \infty$), we observe the following limit:

$$\lim_{T \to \infty} : \vec{v} = \frac{1}{T} \sum_{n=1}^{N} \vec{e_{i_n}} s_n \tag{3.1}$$

Where $\vec{e_{i_n}}$ is an one-hot encoded vector with index i_n set to 1. The proof is in the supplementary material.

Our algorithm is simply doing a discrete-time, bidirectional version of Delta-Sigma modulation - in which we encode floating point elements of our vector $\vec{v}$ as a stream of signed events. We can see this as doing a sort of "deterministic sampling" or "herding"

Algorithm 1 Spiking Vector Quantization

```
Input: $\vec{v} \in \mathbb{R}^d$, $T \in \mathbb{N}$
Internal: $\vec{\phi} \in \mathbb{R}^d \leftarrow \vec{0}$
for t ∈ 1...$T$ do
    $\vec{\phi} \leftarrow \vec{\phi} + \vec{v}$
    while $max(|\vec{\phi}|) > \frac{1}{2}$ do
        $i \leftarrow argmax(|\vec{\phi}|)$
        $s \leftarrow sign(\phi_i)$
        $\vec{\phi_i} \leftarrow \vec{\phi_i} - s$
        FireSpike(source = i, sign = s)
    end while
end for
```

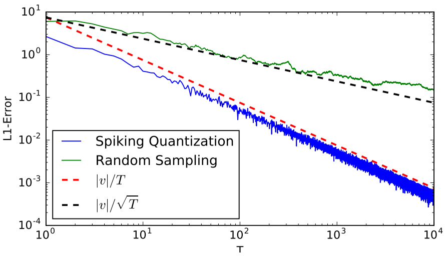


Figure 3.1.: Variable-Spike Quantization shows $1/T$ convergence, while ordinary sampling converges at a rate of $1/\sqrt{T}$. Note both x and y axes are log-scaled.

Welling [2009] of the vector v. Figure 3.1 shows how the cumulative vector from our stream of events approaches the true value of v at a rate of $1/T$. We can compare this to another approach in which we stochastically sample spikes from the vector $\vec{v}$ with probabilities proportional to the magnitude of elements of $\vec{v}$, (see the "Stochastic Sampling" section of the supplementary material), which has a convergence of $1/\sqrt{T}$.

3.4.2. Spiking Stream Quantization

A small modification to the above method allows us to turn a stream of vectors into a stream of signed-spikes.

If instead of a fixed vector $\vec{v}$ we take a stream of vectors $v_{stream} = \{\vec{v_1}, ...\vec{v_T}\}$, we can modify the quantization algorithm to increment $\vec{\phi}$ by $\vec{v_t}$ on timestep t. This modifies Equation 3.1 to:

$$\lim_{T \to \infty} : \frac{1}{T} \sum_{t=1}^{T} \vec{v_t} = \frac{1}{T} \sum_{n=1}^{N} \vec{e_{i_n}} s_n \tag{3.2}$$

So we end up approximating the running mean of v_{stream}. See "Spiking Stream Quantization" in the supplementary material for full algorithm and explanation. When we apply this to implement a neural network in Section 3.4.5, this stream of vectors will be the rows of the weight matrix indexed by the incoming spikes.

3.4.3. Rectifying Spiking Stream Quantization

We can add a slight tweak to our Spiking Stream Quantization algorithm to create a spiking version of a rectified-linear (ReLU) unit. To do this, we only fire events on positive threshold-crossings, resulting in Algorithm 2.

Algorithm 2 Rectified Spiking Stream Quantization

```
Input: $\vec{v_t} \in \mathbb{R}^d$, $t \in [1..T]$
Internal: $\vec{\phi} \in \mathbb{R}^d \leftarrow \vec{0}$
for t ∈ 1...$T$ do
    $\vec{\phi} \leftarrow \vec{\phi} + \vec{v_t}$
    while $max(\vec{\phi}) > \frac{1}{2}$ do
        $i \leftarrow argmax(\vec{\phi})$
        $\vec{\phi_i} \leftarrow \vec{\phi_i} - 1$
        FireSpike(source = i, sign = +1)
    end while
end for
```

We can show that if we draw spikes from a stream of vectors in the manner described in Algorithm 2, and sum up our spikes, we approach the behaviour of a ReLU layer:

$$\lim_{T\to\infty} : \max\left(0, \frac{1}{T}\sum_{t=1}^{T}\vec{v_t}\right) = \frac{1}{T}\sum_{n=1}^{N}\vec{e_{i_n}} \tag{3.3}$$

See “Rectified Stream Quantization” in the supplementary material for a more detailed explanation.

3.4.4. Incremental Dot-Product

Thus far, we’ve shown that our quantization method transforms a vector into a stream of events. Here we will show that this can be used to incrementally approximate the dot product of a vector and a matrix. Suppose we define a vector $\vec{u} \leftarrow \vec{v} \cdot W$, Where W is a matrix of parameters. Given a vector $\vec{v}$, and using Equation 3.1, we see that we can approximate the dot product with a sequence of additions:

$$\vec{u} = \vec{v} \cdot W \approx \frac{1}{T}(\sum_{n=1}^{N}\vec{e}_{i_n}s_n) \cdot W = \frac{1}{T}(\sum_{n=1}^{N}s_n\vec{e}_{i_n} \cdot W) = \frac{1}{T}(\sum_{n=1}^{N}s_n\vec{W}_{i_n,\cdot}) \tag{3.4}$$

Where $W_{i,\cdot}$ is the i’th row of matrix W.

3.4.5. Forward Pass of a Neural Network

Using the parts we've described so far, Algorithm 3 describes the forward pass of a neural network. The InputLayer procedure demonstrates how Spike Vector Quantization, shown in Algorithm 1 transforms the vector into a stream of events. The HiddenLayer procedure shows how we can combine the Incremental Dot-Product (Equation 3.4) and Rectifying Spiking Stream Quantization (Equation 3.3) to approximate the a fully-connected ReLU layer of a neural network. The Figure in the "MLP Convergence" section of the supplimentary material shows that our spiking network, if run for a long time, exactly approaches the function of the ReLU network.

Algorithm 3 Pseudocode for a forward pass in a network with one hidden layer

```
function FORWARDPASS($\vec{x} \in \mathbb{R}^{d_{in}}$, $T \in \mathbb{N}$)
    Variable: $\vec{u} \in \mathbb{R}^{d_{out}} \leftarrow \vec{0}$
    for t $\in$ 1...$T$ do
        InputLayer($\vec{x}$)
    end for
    return $\vec{u}/T$
    procedure INPUTLAYER($\vec{v} \in \mathbb{R}^{d_{in}}$)
        Internal: $\vec{\phi} \in \mathbb{R}^{d_{in}}$
        $\vec{\phi} \Leftarrow \vec{\phi} + \vec{v}$
        while $max(|\vec{\phi}|) > \frac{1}{2}$ do
            $i \leftarrow argmax(|\vec{\phi}|)$
            s $= sign(\phi_i)$
            $\phi_i \leftarrow \phi_i - s$
            HiddenLayer(i, s)
        end while
    end procedure
    procedure HIDDENLAYER($i \in [1..d_{in}]$, $s \in [-1, +1]$)
        Internal: $\vec{\phi} \in \mathbb{R}^{d_{hid}}$, $W \in \mathbb{R}^{d_{in} \times d_{hid}}$
        $\vec{\phi} \Leftarrow \vec{\phi} + s \cdot W_{i,\cdot}$
        while $max(\vec{\phi}) > \frac{1}{2}$ do
            $i \leftarrow argmax(\vec{\phi})$
            $\phi_i \leftarrow \phi_i - 1$
            OutputLayer(i)
        end while
    end procedure
    procedure OUTPUTLAYER($i \in [1..d_{hid}]$)
        Internal: $W \in \mathbb{R}^{d_{hid} \times d_{out}}$
        Global: $\vec{u} \leftarrow \vec{u} + W_{i,\cdot}$                ▷ changes $\vec{u}$
    end procedure
end function
```

3.4.6. Backward Pass

In the backwards pass we propagate error spikes backwards, in the same manner as we propagated the signal forwards, so that the error spikes approach the true gradients of the ReLU network as $T \to \infty$. Pseudocode explaining the procedure is provided in the "Training Iteration" Section of the supplementary material, and a diagram explaining the flow of signals is in the "Network Diagram" section.

A ReLU unit has the function and derivative:

$$\begin{aligned} f(x) &= [x > 0] \cdot x \\ f'(x) &= [x > 0] \end{aligned} \tag{3.5}$$

Where:
$[x > 0]$ denotes a step function (1 if $x > 0$ otherwise 0).

In the spiking domain, we express this simply by blocking error spikes on units for which the cumulative sum of inputs into that unit is below 0 (see the "filter" modules in the "Network Diagram" section of the supplementary material).

The signed-spikes that represent the backpropagating error gradient at a given layer are used to index columns of that layer's weight matrix, and negate them if the sign of the spike is negative. The resulting vector is then quantized, and the resulting spikes are sent back to previous layers.

One problem with the scheme described so far is that, when errors are small, it is possible that the error-quantizing neurons never accumulate enough potential to send a spike before the training iteration is over. If this is the case, we will never be able to learn when error gradients are sufficiently small. Indeed, when initial weights are too low, and therefore the initial magnitude of the backpropagated error signal is too small, the network does not learn at all. This is not a problem in traditional deep networks, because no matter how small the magnitude, some error signal will always get through (unless all hidden units are in their inactive regime) and the network will learn to increase the size of its weights. We found that a surprisingly effective solution to this problem is to simply not reset the ϕ of our error quantizers between training iterations. This way, after some burn-in period, the quantizer's ϕ starts each new training iteration at some random point in the interval $[-\frac{1}{2}, \frac{1}{2}]$, and the unit always has a chance to spike.

A further issue that comes up when designing the backward pass is the order in which we process events. Since an event can move a ReLU unit out of its active range, which blocks the transmission of itself or future events on the backward pass, we need to think about the order in which we processing these events. The topic of event-routing is explained in the "Event Routing" section of the supplementary material.

3.4.7. Weight Updates

We can collect spike statistics and generates weight updates. There are two methods by which we can update the weights. These are as follows:

Stochastic Gradient Descent The most obvious method of training is to approximate stochastic gradient descent. In this case, we accumulate two spike-count vectors, $\vec{c}_{in}$ and $\vec{c}_{error}$ and take their outer product at the end of a training iteration to compute the weight update:

$$\Delta W \leftarrow -\frac{\eta}{T} \cdot \vec{c_{in}} \otimes \vec{c}_{error} \approx -\frac{\eta}{T}\frac{\partial \mathcal{L}}{\partial W} \tag{3.6}$$

Fractional Stochastic Gradient Descent (FSGD) We can also try some thing new. Our spiking network introduces a new feature: if a data point is decomposed as a stream of events, we can do parameter updates even before a single data point has been observed. If we do updates whenever an error event comes back, we update each weight based on only the input data that has been seen *so far*. This is described by the rule:

$$\Delta W_{:,i} \leftarrow -\frac{\eta}{T} \cdot s \cdot \vec{c_{in}} \tag{3.7}$$

Where $\vec{c_{in}}$ is an integer vector of counted input spikes, $\Delta W_{:,i}$ is the change to the $i'th$ column of the weight matrix, $s \in \{-1, 1\}$ is the sign of the error event, and i is the index of the unit that produced that error event, and T is the number of time-steps per training iteration. Early input events will contribute to more weight updates than those seen near the end of a training iteration. Experimentally (see Section 3.5.2, we see that this works quite well. It may be that the additional influence given to early inputs causes the network to learn to make better predictions earlier on, compensating for the approximation caused by finite-runtime of the network.

3.4.8. Training

We chose to train the network with one sample at a time, although in principle it is possible to do minibatch training. We select a number of time steps T, to run the network for each iteration of training. At the beginning of a round of training, we reset the state of the forward-neurons (all $\vec{\phi}$'s and the state of the running sum modules), and leave the state of the error-quantizing neurons (as described in 3.4.6). On each time step t, we feed the input vector to the input quantizer, and propagate the resulting spikes through the network. We then propagate an error spike back from the unit corresponding to the correct class label, and update the parameters by one of the two methods described in 3.4.7. See the “Network Diagram” section of the supplementary material to get an idea of the layout of all the modules.

3.5. Experiments

3.5.1. Simple Regression

We first test our network as a simple regressor, (with no hidden layers) on a binarized version of the newsgroups-20 dataset, where we do a 2-way classification between the electronics and medical newsgroups based word-count vectors. We split the dataset with a 7-1 training-test ratio (as in Crammer et al. [2009]) but do not do cross-validation. Table 3.1 shows that it works.

Network	% **Test / Training Error**
1 Layer NN	2.278 / 0.127
Spiking Regressor	2.278 / 0.82
SVM	4.82 / 0

Table 3.1.: Scores on 20 newsgroups, 2-way classification between 'med' and 'electronic' newsgroups. We see that, somewhat surprisingly, our approach outperforms the SVM. This is probably because, being trained through SGD and tested at the end of each epoch, our classifier had more recently learned on samples at the end of the training set, which are closer in distribution to the test set than those at the beginning.

3.5.2. Comparison to ReLU Network on MNIST

We ran both the spiking network and the equivalent ReLU network on MNIST, using an architecture with 2 fully-connected hidden layers, each consisting of 300 units. Refer to the "Hyperparameters" section of the Supplimentary Material for a full description of hyperparameters.

Table 3.2 shows the results of our experiment, after 50 epochs of training. We find that the conventional ReLU network outperforms our spiking network, but only marginally. In order to determine how much of that difference was due to the fact that the Spiking network has a discrete forward pass, we mapped the learned parameters from the ReLU network onto the Spiking network (spiking with ReLU Weights"), and used the Spiking network to classify . The performance of the spiking network improved nearly to that of the ReLU network , indicating that the difference was not just due to the discretization of the forward pass but also due to the parameters learned in training. We also did the inverse (ReLU with Spiking-FSGD-trained weights) - map the parameters of the trained Spiking Net onto the ReLU net, and found that the performance became very similar to that of the original Spiking (FSGD-Trained) Network. This tells us that most of the difference in score is due to the approximations in training, rather than the forward pass. Interestingly, our Spiking-FSGD approach outperforms the Spiking-SGD - it seems that

Network	**% Test / Training Error**
Spiking SGD:	3.6 / 2.484
Spiking FSGD:	2.07 / 0.37
Vector ReLU MLP	1.63 / 0.426
Spiking with ReLU Weights	1.66 / 0.426
ReLU with Spiking FSGD weights	2.03 / 0.34

Table 3.2.: Scores of various implementations on MNIST after 50 epochs of training on a network with hidden layers of sizes [300, 300]. "Spiking SDG" and "Spiking FSGD" are the spiking network trained with Stochastic Gradient Descent and Fractional Stochastic Gradient descent, respectively, as described in Section 3.4.7. "Vector ReLU MLP" is the score of a conventional MLP with ReLU units and the same architecture and training scheme. "Spiking with ReLU Weights" is the score if we set the parameters of the Spiking network to the already-trained parameters of the ReLU network, then use the Spiking Network to classify MNIST. "ReLU with Spiking weights" is the inverse - we take the parameters trained in the spiking FSGD network and map them to the ReLU net.

by putting more emphasis on early events, we compensate for the finite runtime of the Spiking Network. Figure 3.2 shows the learning curves over the first 20-epochs of training. We see that the gap between training and test performance is much smaller in our Spiking network than in the ReLU network, and speculate that this may be to the regularization effect of the spiking. To confirm this, we would have to show that on a larger network, our regularization actually helps to prevent overfitting.

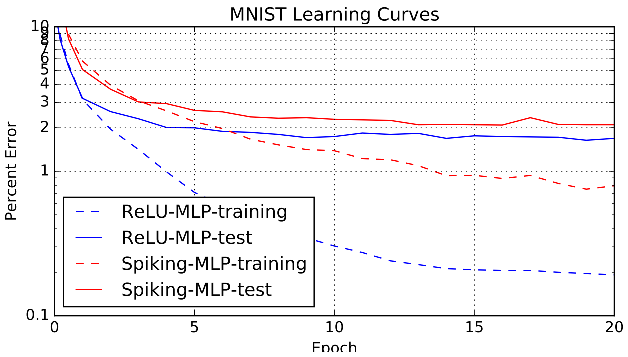


Figure 3.2.: The Learning curves of the ReLU network (blue) and the Spiking network (red). Solid lines indicate test error and dashed lines indicate training error.

3.5.3. Early Guessing

We evaluated the "early guess" hypothesis from Section 3.2 using MNIST. The hypothesis was that our spiking network should be able to make computational cheap “early guesses" about the class of the input, before actually seeing all the data. A related hypothesis was under the “Fractional” update scheme discussed in Section 3.4.7, our networks should learn to make early guesses more effectively than networks trained under regular Stochastic Gradient Descent, because early input events contribute to more weight updates than later ones. Figure 3.3 shows the results of this experiment. We find, unfortunately, that our first hypothesis does not hold. The early guesses we get with the spiking network cost more than a single (sparse) forward pass of the input vector would. The second hypothesis, however, is supported by the right-side of Figure 5. Our networks trained with Fractional Stochastic Gradient Descent make better early guesses than those trained on regular SGD.

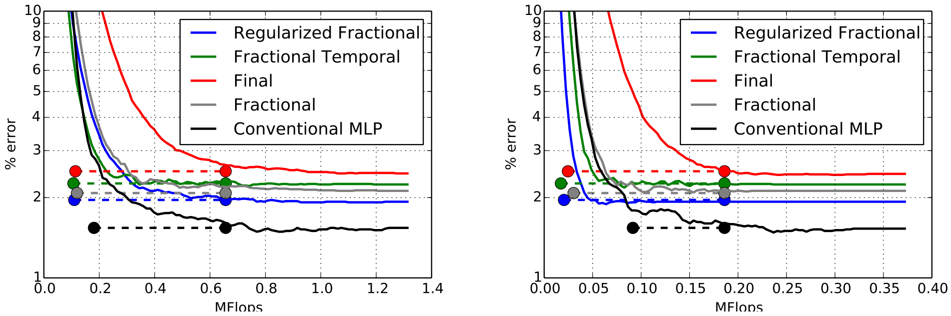


Figure 3.3.: Left: We compare the total computation (x-axis, in MegaFlops) required to achieve a given score (y-axis, in percent error), between differently trained predictors. Each curve represents a differently trained spiking network (Regularized Fraction was trained with a regularization term, Fractional Temporal was trained with higher learning rate for early events). The black line is the convergence curve of a spiking network with parameters learned in a conventional ReLU net of the same architecture (784-300-300-10). The dots show the computation time and performance of a conventional ReLU network, with the right dot indicating the cost of a full feedforward pass, and the left indicating the cost when one removes units with 0-activation when computing the cost of a matrix multiplication. We see that, when considering the full network, our spiking approach does give a good early guess compared to a naively implemented deep net, but not after considering sparsity. Right: However, when only considering layers after the input layer (whose sparsity we do not control), we can see that there is an advantage to our spiking training scheme: In the low-flop range our spiking nets have lower error. The networks that were trained as spiking networks are better at making early guesses than the conventionally trained network.

3.6. Discussion

We implemented a Spiking Multi-Layer Perceptron and showed that our network behaves very similarly to a conventional MLP with rectified-linear units. However, our model has some advantages over a regular MLP, most of which have yet to be explored in full. Our network needs neither multiplication nor floating-point numbers to work. If we use Fractional Stochastic Gradient Descent, and scale all parameters in the network (initial weights, thresholds, and the learning rate) by the inverse of the learning rate, the only operations used are integer addition, indexing, and comparison. This makes our system very amenable to efficient hardware implementation.

The Spiking MLP brings us one step closer to making a connection between the types of neural networks we observe in biology and the type we use in deep learning. Like biological neurons, our units maintain an internal potential, and only communicate when this potential crosses some firing threshold. We believe that the main value of this approach is that it is a stepping stone towards a new type of deep learning. The way that deep learning is done now takes no advantage of the huge temporal redundancy in natural data. In the future we would like to adapt the methods developed here to work with nonstationary data. Such a network could pass spikes to keep the output distribution in "sync" with an ever-changing input distribution. This property - efficiently keeping an online track of latent variables in the environment, could bring deep learning into the world of robotics.

4. Sigma-Delta Quantized Networks

Our second work, *Sigma Delta Quantized Networks*, deals with one aspect of Gap 3: Nonstationary Data: In many situations, we may want to do inference on non-IID data. One example is frame-by-frame classification of video data with a Convolutional neural network. Video data tends to be highly *non-stationary* - neighbouring frames tend to be highly correlated, and as frame rate is increased, this inter-frame correlation increases too. Intuitively, it seems that "spiking" is a sensible approach to such cases. A spiking neuron accumulates input until some threshold is crossed, and only communicates (sends a spike) at that time. If we can use spikes to communicate "changes" to the input, we can potentially design a system which does deep inference on video for much less computation than it costs to run inference on each frame independently. In this work we are not concerned with *training* the network - but merely looking for a more efficient way to do inference with a pre-trained network.

The core idea is to only send *quantized changes* in activation between neurons. The amount of information communicated between neurons should thus scale with the rate of change of the input data. A perfectly static input (e.g. the unmoving view of a CCTV camera upon an empty store), should trigger no new computation within the network. On a slowly varying input, neurons will communicate with sparse integer signals at every time-step.

Our system receives a time-varying input vector $x(t) \in \mathbb{R}^{D_{\text{input}}}$ and transforms it into a sparse integer vector $q_0(t) \in \mathbb{Z}^{D_{\text{input}}}$ by taking the quantized temporal-difference between frames: $q_0(t) := \text{round}(k_0 x(t)) - \text{round}(k_0 x(t-1))$, where $k_0 \in \mathbb{R}^+$ is the scale-factor applied to the input layer (layer 0) - affecting the coarseness of quantization. The first layer then receives this signal via weight matrix w and temporally integrates it to get the activation of layer 1: $z_1(t) := z_1(t-1) + w_1 \cdot q_0(t)/k_0$, which is passed though a nonlinearity h before its temporal-change is quantized as $q_1(t) := round(k_1 h(z_1(t))) - round(k_1 h(z_1(t-1)))$. We repeat this for each layer until the output z_L is computed. The remainder of the paper concerns how to optimize the coarseness coefficients $k_{[0..L-1]}$ to achieve an optimal trade-off between performance (in terms of faithfulness to the function of a non-quantized network) and computation (in terms of the number of arithmetic operations required to update the state).

Because this work not involve any training of the quantized network (only the coarseness parameters k) it does not even address Gap 1: No Backprop. Our quantization could be seen as a form of spiking - addressing Gap 2: Spiking - and it actually brings up a point that inspires our next paper - that spikes may be used to convey *changes* to the state of a neuron, rather than simply being a quantized or stochastic representation of the current state. Since we process data in a feedforward-only and time-stepped manner, we do not really address Gap 4: Asynchronous Processing, although our approach could be extended to continuous-time asynchronous events if the input data arrived in that format. The advantage of our approach is that as our frame-rate increases (as our units of time become smaller), the average amount of computation over time should be bounded, so long as the signal $x(t)$ is smooth in time.

4.1. Abstract

Deep neural networks can be obscenely wasteful. When processing video, a convolutional network expends a fixed amount of computation for each frame with no regard to the similarity between neighbouring frames. As a result, it ends up repeatedly doing very similar computations. To put an end to such waste, we introduce Sigma-Delta networks. With each new input, each layer in this network sends a discretized form of its *change* in activation to the next layer. Thus the amount of computation that the network does scales with the amount of change in the input and layer activations, rather than the size of the network. We introduce an optimization method for converting any pre-trained deep network into an optimally efficient Sigma-Delta network, and show that our algorithm, if run on the appropriate hardware, could cut at least an order of magnitude from the computational cost of processing video data.

4.2. Introduction

For most deep-learning architectures, the amount of computation required to process a sample of input data is independent of the contents of that data.

Natural data tends to contain a great deal of spatial and temporal redundancy. Researchers have taken advantage of such redundancy to design encoding schemes, like jpeg and mpeg, which introduce small compromises to image fidelity in exchange for substantial savings in the amount of memory required to store images and videos.

In neuroscience, it seems clear that that some kind of sparse spatio-temporal coding is going on. Koch et al. [2006] estimate that the human retina transmits 8.75Mbps, which is about the same as compressed 1080p video at 30FPS.

Thus it seems natural to think that perhaps we should be doing this in deep learning. In this paper, we propose a neural network where neurons only communicate discretized changes in their activations to one another. The computational cost of running such a network would be proportional to the amount of change in the input. Neurons send signals when the change in their input accumulates past some threshold, at which point they send a discrete "spike" notifying downstream neurons of the change. Such a system has at least two advantages over the conventional way of doing things.

1. When extracting features from temporally redundant data, it is much more efficient to communicate the changes in activation than it is to re-process each frame.
2. When receiving data asynchronously from different sources (e.g. sensors, or nodes in a distributed network) at different rates, it no longer makes sense to have a global network update. We could recompute the network with every new input, reusing the stale inputs from the other sources, but this requires doing a great deal of repeated computation for only small differences in input data. We could keep a history of all inputs and update the network periodically, but then we lose the ability to respond immediately to new inputs. Our approach gets around this ugly tradeoff by allowing for efficient approximate updates of the network given a partial update to the input data. The computational cost of the update is proportional to the effect that the new information has on the network's state.

4.3. Related Work

This work originated in the study of spiking neural networks, but treads into the territory of discretizing neural nets. The most closely related work is that of Zambrano and Bohte [2016]. In this work, the authors describe an Adaptive Sigma-Delta modulation method, in which neurons communicate analog signals to one another by means of a "spike-encoding" mechanism, where a temporal signal is encoded into a sequence of weighted spikes and then approximately decoded as a sum of temporally-shifted exponential kernels. The authors create a scheme for being parsimonious with spikes by allowing adaptive scaling of thresholds, at the cost of sending spikes with real values attached to them, rather than the classic "all or nothing" spikes. Their work references a slightly earlier work by Yoon [2016] which reframes common neural models as forms of Asynchronous Sigma-Delta modulation. In a concurrent work, Lee et al. [2016] implement backpropagation in a similar system (but without adaptive threshold scaling), and demonstrate the best-yet performance on MNIST for networks trained with spiking models. This work postdates Diehl et al. [2015], which proposes a scheme for normalizing neuron activations so that a spiking neural network can be optimized for fast classification.

Our model contrasts with all of the above in that it is time-agnostic. Although we refer to sending "temporal differences" between neurons, our neurons have no concept of time - their is no "leak" in neuron potential, and our neurons' behaviour only depends on the order of the inputs. Our work also separates the concepts of nonlinearity and discretization, uses units that communicate differences rather than absolute signal values, and explicitly minimizes an objective function corresponding to computational cost.

Coming from another corner, Courbariaux et al. [2016] describe a scheme for binarizing networks with the aim of achieving reductions in the amount of computation and memory required to run neural nets. They introduce a number of tricks for training binarized neural networks - a normally difficult task due to the lack of gradient information. Esser et al. [2016] use a similar binarization scheme to efficiently implement a spiking neural network on the IBM TrueNorth chip. Ardakani et al. [2015] take another approach - to approximate real-valued operations of a neural net with a sequence of stochastic integer operations, and show how these can lead to cheaper computation.

These discretization approaches differ from ours in that they do not aim to take advantage of temporal redundancy in data, but rather aim to find ways of saving computation by learning in a low-precision regime. Ideas from these works could be combined with the ideas presented in this paper.

The idea of sending quantized temporal differences has been applied to make event-based sensors, such as the Dynamic-Vision Sensor [Lichtsteiner et al., 2008], which quantize changes in analog pixel-voltages and send out pixel-change events asynchronously. The model we propose in this paper could be used to efficiently process the outputs of such sensors.

Finally, our previous work, [O'Connor and Welling, 2016a] develops a method for doing backpropagation with the same type of time-agnostic spiking neurons we use here. In this paper, we do not aim to train the network from scratch, but instead focus on how we can compute efficiently by sending temporal differences between neurons.

4.4. The Sigma-Delta Network

In this Section, we describe how we start with a traditional deep neural network and apply two modifications - temporal-difference communication and rounding - to create the Sigma-Delta network. To explain the network, we follow the Figure 4.1 from top to bottom, starting with a standard deep network and progressing to our Sigma-Delta network. Here, we will think of the forward pass of a neural network as composition of subfunctions: $f(x) = (f_L \circ ... \circ f_2 \circ f_1)(x)$.

4.4.1. Temporal Difference Network

We now define "temporal difference" (Δ_T) and "temporal integration" (Σ_T) modules as follows:

Algorithm 4 Temporal Difference (Δ_T):

```
Internal: $\vec{x}_{last} \in \mathbb{R}^d \leftarrow \vec{0}$
Input: $\vec{x} \in \mathbb{R}^d$
$\vec{y} \leftarrow \vec{x} - \vec{x}_{last}$
$\vec{x}_{last} \leftarrow \vec{x}$
Return: $\vec{y} \in \mathbb{R}^d$
```

Algorithm 5 Temporal Integration (Σ_T):

```
Internal: $\vec{y} \in \mathbb{R}^d \leftarrow \vec{0}$
Input: $\vec{x} \in \mathbb{R}^d$
$\vec{y} \leftarrow \vec{y} + \vec{x}$
Return: $\vec{y} \in \mathbb{R}^d$
```

So that when presented with a sequence of inputs $x_1, ...x_t$, $\Delta_T(x_t) = x_t - x_{t-1}|_{x_0=0}$, and $\Sigma_T(x_t) = \sum_{\tau=1}^{t} x_\tau$. It should be noted that when we refer to "temporal differences", we refer not to the change in the signal over time, but in the change between two inputs presented sequentially. The output of our network only depends on the value and order of inputs, not on the temporal spacing between them. This distinction only matters when dealing with asynchronous inputs such as the Dynamic Vision Sensor, [Lichtsteiner et al., 2008], which are not considered in this paper.

Since $\Sigma_T(\Delta_T(x_t)) = x_t$, we can insert $\Sigma_T \circ \Delta_T$ pairs into the network without affecting the function. So we can re-express our network function as: $f(x) = (f_L \circ \Sigma_T \circ \Delta_T \circ ... \circ f_2 \circ \Sigma_T \circ \Delta_T \circ f_1 \circ \Sigma_T \circ \Delta_T)(x)$.

Now suppose our network consists of alternating linear functions $w(x)$, and nonlinear functions $h(x)$, so that $f(x) = (h_L \circ w_L ... \circ h_2 \circ w_2 \circ h_1 \circ w_1)(x)$. As before, we can harmlessly insert our $\Sigma_T \circ \Delta_T$ pairs into the network. But this time, note that for a linear function $w(x)$, the operations (Σ_T, w, Δ_T) all commute with one another. That is:

$$\Delta_T(w(\Sigma_T(x))) = w(\Delta_T(\Sigma_T(x))) = w(x) \tag{4.1}$$

Therefore we can replace all instances of $\Delta_T \circ w \circ \Sigma_T$ with w, yielding $f(x) = (h_L \circ \Sigma_T \circ w_L \circ ... \circ \Delta_T \circ h_2 \circ \Sigma_T \circ w_2 \circ \Delta_T \circ h_1 \circ \Sigma_T \circ w_1 \circ \Delta_T)(x)$, which corresponds to the network shown in Figure 4.1 B. For now this is completely pointless, since we do not change the network function at all, but it will come in handy in the next section, where we discretize the output of the Δ_T modules.

4.4.2. Discretizating the Deltas

When dealing with data that is naturally spatiotemporally redundant, like most video, we expect the output of the Δ_T modules to be a vector with mostly low values, with some peaks corresponding to temporal transitions at certain input positions. We expect the data to have this property not only at the input layer, but even more so at higher layers, which encode higher level features (edges, object parts, class labels), which we would expect to vary more slowly over time than pixel values. If we discretize this "peaky" vector, we end up with a sparse vector of integers, which can then be used to cheaply communicate the approximate change in state of a layer to its downstream layer(s).

A sensible approach is to apply rounding before the temporal-difference operation - i.e. round the activation values and then send the temporal differences of these rounded values. It is then easy to show that the network's function will remain identical to that of the rounding network:

$$\Sigma_T(w(\Delta_T(round(x)))) = w(\Sigma_T(\Delta_T(round(x)))) = w(round(x)) \tag{4.2}$$

It's worth noting that this is equivalent to doing discrete-time Sigma-Delta modulation to quantize the temporal differences - this connection is explained in Appendix B.1.

It follows from this result that our Sigma-Delta network depicted in Figure 4.1 D computes an identical function to that of the rounding network in Figure 4.1 C. In other words, the output y_t of the Sigma-Delta network is solely dependent on the parameters of the network and the current input x_t, and not on any of the previous inputs $x_1..x_{t-1}$. The amount of computation required for the update, however, depends on x_{t-1}. Specifically, if x_t is similar to x_{t-1}, the Sigma-Delta network should require less computation to perform an update than the Rounding Network.

4.4.3. Sparse Dot Product

Most of the computation in Deep Neural networks is consumed doing matrix multiplications and convolutions. The architecture we propose saves computation by translating the input to these operations into an integer array with a small L1 norm.

With sparse, low-magnitude integer input, we can compute the vector-matrix dot product efficiently by decomposing it into a sequence of vector additions. We can see this by decomposing the vector $\vec{x} \in \mathbb{I}^{d_{in}}$ into a set of indices $\langle (i_n, s_n) : i \in [1..len(\vec{x})], s \in \pm 1, n = [1..N] \rangle$, such that: $\vec{x} = \sum_{n=1}^{N} s_n \vec{e}_{i_n}$, where e_{i_n} is a one-hot vector with element i_n hot, and $N = |\vec{x}|_{L1}$ is the total L1 magnitude of the vector. We can then compute the dot-product as a series of additions, as shown in Equation 4.3.

$$\begin{aligned} u &= \vec{x} \cdot W : W \in \mathbb{R}^{d_{in} \times d_{out}} \\ &= \left(\sum_{n=1}^{N} s_n \vec{e}_{i_n} \right) \cdot W = \sum_{n=1}^{N} \vec{s}_n e_{i_n} \cdot W = \sum_{n=1}^{N} s_n \cdot W_{i_n,\cdot} \end{aligned} \tag{4.3}$$

Computing the dot product this way takes $N \cdot d_{out}$ additions. A normal dense dot-product, by comparison, takes $d_{in} \cdot d_{out}$ multiplications and $(d_{in} - 1) \cdot d_{out}$ additions.

This is where the energy savings come in. Horowitz [2014] estimates that on current 45nm silicon process, a 32-bit floating-point multiplication costs 3.7pJ, vs 0.9pJ for floating-point addition. With integer or fixed-point arithmetic, the difference is even more pronounced, with 3.1pJ for multiplication vs 0.1pJ for addition. This of course ignores the larger cost of processing instructions and moving memory, but gives us an idea of how these operations might compare on optimized hardware. So provided we can approximate the forward pass of a network to a satisfactory degree of precision without doing many more operations than the original network, we can potentially compute much more efficiently.

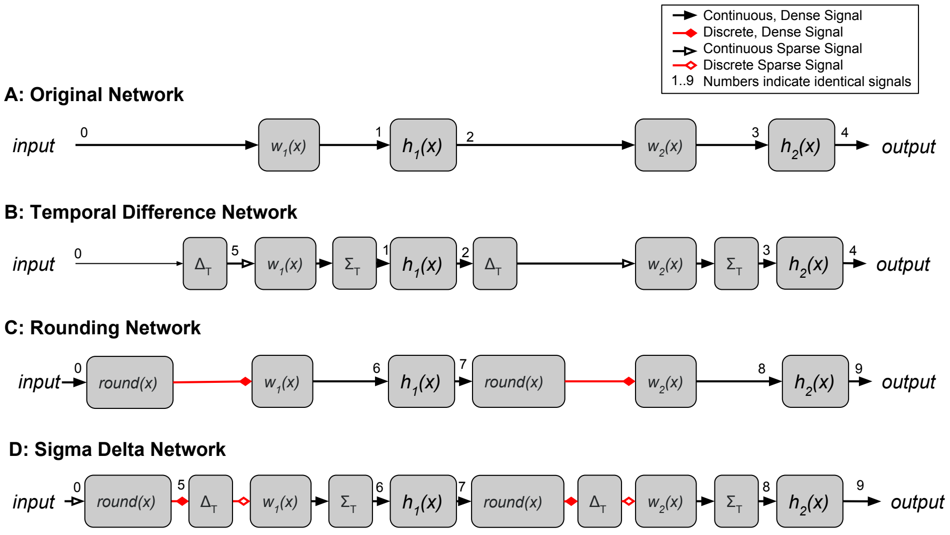


Figure 4.1.: **A**: An ordinary deep network, which consists of an alternating sequence of linear operations $w_i(x)$, and nonlinear transforms $h_i(x)$. **B**: The Temporal-Difference Network, described in Section 4.4.1, computes the exact same function as network A, but communicates differences in activation between layers. **C**: An approximation of network A where activations are rounded before being sent to the next layer. **D**: The Sigma-Delta Network combines the modifications of B and C. Functionally, it is identical to the Rounding Network, but it can compute forward passes more cheaply when input data is temporally redundant.

4.4.4. Putting it all together

Figure 4.1 visually summarizes the four types of network we have described. Inserting the temporal sum and difference modules discussed in Section 4.4.1 leads to the Temporal Difference Network, which is functionally identical to the Original Network. Discretizing the output of the temporal difference modules, as discussed in Section 4.4.2, leads to the Sigma-Delta network. The Sigma-Delta Network is functionally equivalent to the Rounding network, except that it requires less computation per forward pass if it is fed with temporally redundant data.

4.5. Optimizing an Existing Network

In this work, we do not aim to train a quantized networks from scratch, as we did in O'Connor and Welling [2016a]. Rather, we will take existing pretrained networks and optimize them as Sigma-Delta networks. In in our situation, we have two competing objectives: Error (with respect to a non-quantized forward pass), and Computation: the number of additions performed in a forward pass.

4.5.1. Rescaling our Neurons

We can control the trade-off between these objectives by changing the scale of our discretization. We can thus extend our rounding function by adding a scale $k \in \mathbb{R}^+$:

$$round(\vec{x}, k) \equiv round(\vec{x} \cdot k)/k \tag{4.4}$$

This scale can either be layerwise or unitwise (in which case we have a vector of scales per layer). Higher k values will lead to higher precision, but also more computation, for the reason mentioned in Section 4.4.2. Note that the final division-by-k is equivalent to scaling the following weight matrix by $\frac{1}{k}$,. So in practice, our network functions become:

$$f_{round}(x) = \left(h_L \circ \frac{w_L}{k_L} \circ round \circ \cdot k_L \circ ... \circ h_1 \circ \frac{w_1}{k_1} \circ round \circ \cdot k_1 \right)(x) \tag{4.5}$$

$$f_{\Sigma\Delta}(x) = \left(h_L \circ \Sigma_T \circ \frac{w_L}{k_L} \circ round \circ \cdot k_L \circ \Delta_T \circ ... \circ h_1 \circ \Sigma_T \circ \frac{w_1}{k_1} \circ round \circ \cdot k_1 \circ \Delta_T \right)(x) \tag{4.6}$$

For the Rounding Network and the Sigma-Delta Network, respectively. By adjusting these scales k_l, we can affect the tradeoff between computation and error. Note that if we use ReLU activation functions, parameters k_l can simply be baked into the parameters of the network (see Appendix B.3.)

4.5.2. The Art of Compromise

In this section, we aim to find the optimal trade-offs between Error and Computation for the Rounding Network (Network C in Figure 4.1). We define our loss as follows:

$$\mathcal{L}_{error} = \mathcal{D}(f_{round}(x), f_{true}(x)) \tag{4.7}$$

$$\mathcal{L}_{comp} = \sum_{l=1}^{L-1} |s_l|_{L1} d_{l+1} \tag{4.8}$$

$$\mathcal{L}_{total} = \mathcal{L}_{error} + \lambda \mathcal{L}_{comp} \tag{4.9}$$

Where $\mathcal{D}(a, b)$ is some scalar distance function (We use KL-divergence for softmax output layers and L2-norm otherwise), $f_{round}(x)$ is the output of the Rounding Network, $f_{true}(x)$ is the output of the Original Network. $\mathcal{L}_{comp}$ is the computational loss, defined as the total number of additions required in a forward pass. Each layer performs $|s_l|_{L1} d_{l+1}$ additions, where s_l is the discrete output of the l'th layer, d_{l+1} is the dimensionality of the $(l+1)$'th layer. Finally λ is the tradeoff parameter balancing the importance of the two losses.

We aim to use this loss function to optimize our layer-scales, k_l to find an optimal tradeoff between accuracy and computation, given the tradeoff parameter λ.

4.5.3. Differentiating the Undifferentiable

We run into an obvious problem: $y = round(k \cdot x)$ is not differentiable with respect to our scale, k or our input, x. We get around this by using a similar method to Courbariaux et al. [2016], who in turn borrowed it from a lecture by Hinton [2012]. That is, on the backward pass, when computing the gradient with respect to the error $\frac{\partial \mathcal{L}_{error}}{\partial k_l}$, we simply pass the gradient through the rounding function in layers $[l+1, .., .L]$, i.e. we say $\frac{\partial}{\partial x} round(x) \approx 1$.

When computing the gradient with respect to the computational cost, $\frac{\partial \mathcal{L}_{comp}}{\partial k_l}$, we again just pass the gradient through all rounding operations in the backward pass for layers $[l+1, .., .L]$. We found instabilities in training when using the computational loss of higher layers: $\mathcal{L}_{comp,l'} : l' \in [l+1, ..., L]$, to update the scale of layer l. Since we don't expect this term to have much effect anyway, we choose to only use the gradient of the computational cost in layer l when updating scale k_l, i.e., we approximate: $\frac{\partial \mathcal{L}_{comp}}{\partial k_l} \approx \frac{\partial \mathcal{L}_{comp,l}}{\partial k_l}$.

Our scale parameters also must remain in the positive range, and stay well away from zero, where they can cause instability due to the division-by-k (see Equation 4.5). To handle this, we parametrize our scales in log-space, as $\kappa_l = log(k_l)$. Our scale-parameter update rule becomes:

$$\Delta \kappa_l = -\eta \left(\frac{\partial \mathcal{L}_{error}}{\partial \kappa_l} \bigg|_{pass:[l+1..L]} + \lambda \frac{\partial}{\partial \kappa_l} |\vec{s_l}|_{L1} d_{l+1} \bigg|_{pass:l} \right) \tag{4.10}$$

Where $\vec{s_l}$ is the rounded signal from layer l, d_{l+1} is the "fan-out" (equivalent to the dimension of layer $l+1$ in a fully-connected network), and $pass : [l+1..L]$ indicates that, on the backward pass, we simply pass the gradient through the rounding functions on layers $[l+1..L]$.

4.6. Experiments

4.6.1. Toy Problem: A random network

We start with a very simple toy problem to verify our method. We initialize a 2-layer (100-100-100) ReLU network with random weights using the initialization scheme proposed in Glorot and Bengio [2010], then scaled the weights by $\left(\frac{1}{2}, 8, \frac{1}{4}\right)$. The weight-rescaling does not affect the function of the network but makes it very ill-adapted for discretization (the first layer will be represented too coarsely, causing error; the second too finely, causing wasted computation). We create random input data, and use it to optimize the layer scales according to Equation 4.10. We verify, by comparing to a large collection of randomly drawn rescalings, that by tuning lambda we land on different places of the Pareto frontier balancing error and computation. Figure 4.2 shows that this is indeed the case. In this experiment, error and computation are evaluated just on the Rounding network - we test the Sigma-Delta network in the next experiment, which includes temporal data.

4.6.2. Temporal-MNIST

In order to evaluate our network's ability to save computation on temporal data, we create a dataset that we call "Temporal-MNIST". This is just a reshuffling of the standard MNIST dataset so that similar frames tend to be nearby, giving the impression of a temporal sequence (see Appendix B.4 for details). The columns of Figure 4.3 show eight snippets from the Temporal-MNIST dataset.

We started our experiment with a conventional ReLU network with layer sizes [784-200-200-10] pre-trained on MNIST to a test-accuracy of 97.9%. We then apply the same scale-optimization procedure for the Rounding Network used in the previous experiment to find the optimal rescalings under a range of values for λ. This time, we test the learned scale parameters on both the Rounding Network and the Sigma-Delta network. We do not attempt to directly optimize the scales with respect to the amount of computation in the Sigma-Delta network - we assume that the result should be similar to that for the rounding network, but verifying this is the topic of future work.

The results of this experiment can be seen in Figure 4.4. We see that our discretized networks (Rounding and Sigma-Delta) converge to the error of the original network with fewer computations than are required for a forward pass of the original neural network. Note that the errors of the rounding and Sigma-Delta networks are identical. This is a

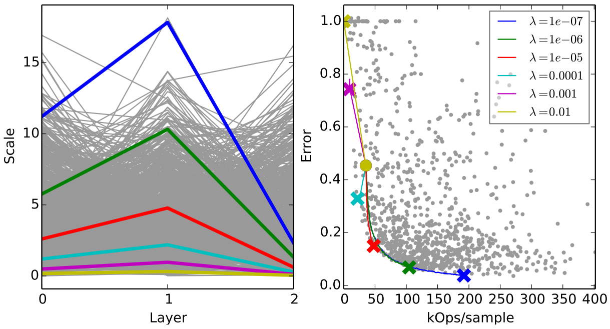


Figure 4.2.: The Results of the "Random Network" experiment described in Section 4.6.1. **Left**: A plot of the layerwise scales. Grey lines show randomly sampled scales, and coloured lines show optimal scales for different values of λ. **Right**: Gray dots show the error-scale tradeoffs of network instantiations using the (gray) randomly sampled rescalings on the left. Coloured lines show the optimization trajectory under different values of λ, starting with the initial state (●), and ending with ×.

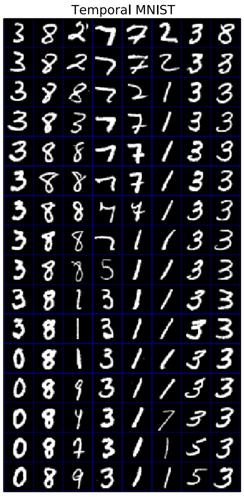


Figure 4.3.: Some samples from the Temporal-MNIST dataset. Each column shows a snippet of adjacent frames.

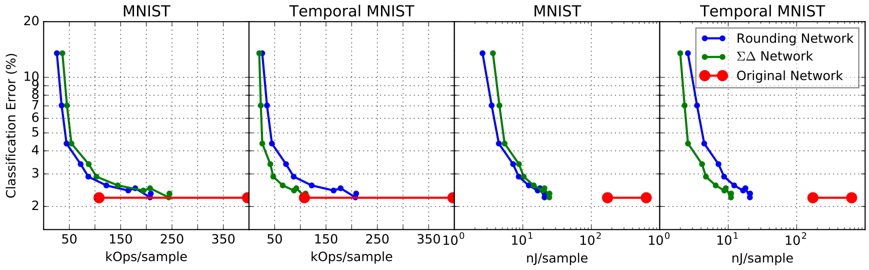


Figure 4.4.: A visualization of our error-computation tradeoff curve for MNIST and our Temporal-mnist dataset. **Plot 1**: Each point on the line for the Rounding (blue) and Sigma-Delta (green) network correspond to the performance of the network for a different value of the error-computation tradeoff parameter λ, ranging from $\lambda = 10^{-10}$ (in the high-computation, low-error regime) to $\lambda = 10^{-5}$ (in the low-computation, high-error regime). The red line indicates the performance of the original, non-discretized network. The red dot on the right indicates the number of flops required for a full forward pass when doing dense multiplication, and the dot on the left indicates the number of flops when factoring in layer sparsity. Note that for the Rounding and Sigma-Delta networks, we count the number of additions, and for the original network we count the numbers of multiplications and additions (as per Section 4.4.3). **Plot 2**: The same, but on the Temporal-MNIST dataset. We see that the Sigma-Delta network uses less computation thanks to the temporal redundancy in the data. **Plots 3 and 4**: Half of the original network's Ops were multiplies, which are more computational costly than the additions of the Rounding and Sigma-Delta networks. In these plots the x-axis is rescaled according to the energy use calculations of Horowitz [2014], assuming the weights and parameters of the network are implemented with 32-bit integer arithmetic. Numbers are in Appendix B.5.

consequence of their equivalence, described in Section 4.4.2. Note also that the errors for all networks are identical between the MNIST and Temporal-MNIST datasets, since for all networks, the prediction function is independent of the order in which inputs are processed. We see that as expected, our Sigma-Delta network does fewer computations than the rounding network on the Temporal-MNIST dataset for the same error, because the update-mechanism of this network takes advantage of the temporal redundancy in the data.

4.6.3. A Deep Convolutional Network on Video

Our final experiment is a preliminary exploration into how Sigma Delta networks could perform on natural video data. We start with "VGG 19" - a 19 layer convolutional network, trained to recognise the 1000 ImageNet categories. The network was trained and made public by Simonyan and Zisserman [2014]. We take selected videos from the ILSVRC 2015 dataset [Russakovsky et al., 2015], and apply the rescaling method from Section 4.5.1 to adjust the scales on a per-layer basis. We initially had some difficulty in optimizing the scale parameters of network to a stable point. The network would either fail to reduce computation when it could afford to, or reduce it to the point where the network's function was so corrupted that error gradients would be meaningless, causing computation loss to win out and activations to drop to zero. A simple solution was to replace the rounding operation in training with addition of uniform random noise $\epsilon \sim U(-\frac{1}{2}, \frac{1}{2})$. This seemed to prevent the network from pushing itself into a regime where all activations become zero. More work is need to understand why the addition of noise is necessary here. Figure 4.5 shows some preliminary results, which indicate that for video data we can get about 4-10x savings in the amount of computation required, in exchange for a modest loss in computational accuracy.

4.7. Discussion

We have introduced Sigma-Delta Networks, which give us a new way compute the forward pass of a deep neural network. In Sigma-Delta Networks, neurons communicate not by telling other neurons about their current level of activation, but about their change in activation. By discretizing these changes, we end up with very sparse communication between layers. The more similar two consecutive inputs (x_t, x_{t+1}) are, the less computation is required to update the network. We show that, while the Sigma-Delta Network's internal state at time-step t depends on past inputs $x_1..x_{t-1}$, the output y_t only depends on the current input x_t. We show that there is a tradeoff between the accuracy of this network (with respect to the function of a traditional deep net with the same parameters), and the amount of computation required. Finally, we propose a method to jointly optimize error and computation, given a tradeoff parameter λ that indicates how much accuracy we are willing to sacrifice in exchange for fewer computations. We demonstrate that this method substantially reduces the number of computations required to run a deep network on natural, temporally redundant data. However, we observe in our final experiment (Figure 4.5, bottom) that our assumption that higher-level features would be more temporally stable - and thus require less computation in our Sigma-Delta net - was not true. We suspect that if we were to train the network from scratch on temporal data, we may learn more temporally stable "slow" features, but this is a topic of future work.

A huge amount of data (eg. video, audio) comes in the form of temporal sequences, and there is an increasingly obvious need to be able to process this data efficiently. There is much to be gained by only doing processing when necessary, based on the contents of the data, and we provide one method for doing that. Further work is needed to determine whether this method would be of use on modern computing hardware, namely GPUs. The problem is that these devices are designed for large, fixed-size array operations, and tend not to be good at taking advantage of sparsity in the data, which requires many

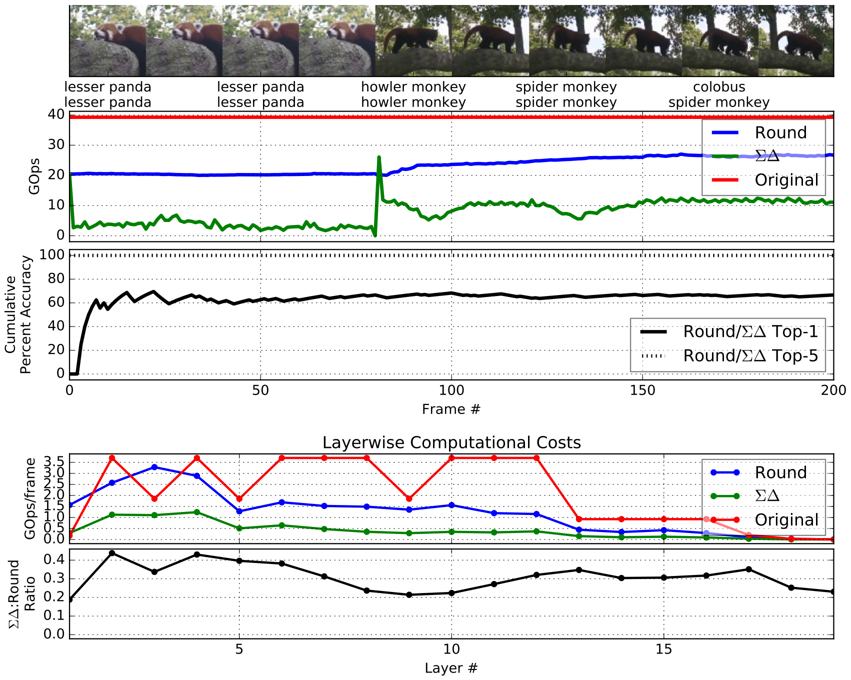


Figure 4.5.: A comparison of the Original VGG Net with the Rounding and Sigma-Delta Networks using the same parameters, after scale-optimization. **Top**: Frames taken from two videos from the ILSVRC2015 dataset. The two videos, with 201 frames in total, were spliced together. The first has a static background, and the second has more motion. Below every second image is the label generated for that image by VGGnet and the Sigma-Delta network (which is functionally equivalent to the Rounding Network, though numerical errors can lead to small changes, not shown here). Scale parameters were trained on separate videos. **Second Plot**: A comparison of the computational cost per-frame. The original VGG network has a fixed cost. The Sigma-Delta network has a cost that varies with the amount of action in the video. The spike in computation occurs at the point where the videos are spliced together. We can see that the Sigma-Delta network does more computation for the second video, in which there is more movement. During the first video it performs about 11 times less computation than the Original Network, during the second, about 4 times less. The difference would be more pronounced if we were to count energy use, as we did in Figure 4.4. **Third Plot**: A plot of the cumulative mean error (over frames) of the Sigma-Delta/Rounding networks, as compared to the Original VGGnet. Most of the time, it gets the same result (Top-1) out of 1000 possible categories. On almost every frame, the guess of the Sigma-Delta network is one of the top-5 guesses of the original VGGNet. **Fourth Plot**: A breakdown of how much of the computational cost of each network comes from each layer. **Fifth Plot**: The layer-wise ratio of the computational cost of the Sigma-Delta net to the rounding net. We had expected (and hoped) this ratio to become very low in the upper layers, as the high-level features should not change much between frames. However this was not the case (as the ratio remains between 0.2 and 0.4 across all layers). It appears therefore that our assumption - that higher level features would be more temporally stable - is untrue. Appendix B.6 shows that this is a property of the pretrained network, not our quantization scheme.

random memory accesses to parameters. Fortunately, other devices such as the the IBM TrueNorth [Cassidy et al., 2013] are being designed which keep memory close to processing, and such handle sparse data (and random memory access) much more efficiently.

This work opens up an interesting door. In asynchronous, distributed neural networks, a node may receive input from many different nodes asynchronously. Recomputing the function of the network every time a new input signals arrives may be prohibitively expensive. Our scheme deals with this by making the computational cost of an update proportional to the amount of change in the input. The next obvious step is to extend this approach to communicating changes in gradients, which may be helpful in setting up distributed, asynchronous schemes for training neural networks.

Code for our experiments can be found at: `https://github.com/petered/sigma-delta/`

Acknowledgments

This work was supported by Qualcomm, who we'd also like to thank for discussing their past work in the field with us. We'd also like to thank fellow lab members, especially Changyong Oh and Matthias Reisser, for fruitful discussions contributing to this work.

5. Temporally Efficient Deep Learning with Spikes

Our next paper further explores the idea that spikes may be primarily used to signal a *change* of state of a neuron, rather than being a representation of the state itself. The paper asks how one could train a network where inter-neuron communication is primarily used to convey *change* of state - and thus exploit the temporal redundancy of data to do more efficient training. Our training scheme is based on backpropagation, thus does not address Gap 1: No Backprop or Gap 4: Asynchronous Processing. In this work, we deal with how to quantize communication between neurons to reduce inter-neuron communication (and therefore potentially energy expenditure) when the data is temporally redundant - thus addressing Gap 2: Spiking and Gap 3: Nonstationary Data.

In Chapter 4 we introduced the notion of having neurons communicate the *quantized temporal change* of their state, rather then their full state, with the goal minimizing the amount of communication required to update the state of a neural network in response to time-varying input data.

The problem becomes more complicated when we want to *train* a network on temporal data - not just do inference. This is because when training, both the weights and activations are changing with time. Suppose a neuron with time-varying state $s(t) \in \mathbb{R}$ takes input from a vector of neurons with time-varying states $x(t) \in \mathbb{R}^D$ and dots it with a weight vector $w \in \mathbb{R}^D$. Ignoring quantization for the time being, these states could be conveyed by sending the *temporal difference* of the inputs $x(t) - x(t-1)$, at each step and reconstructing it on the other end by temporal summation: $s(t) := \sum_{\tau=1}^{t}(x(\tau) - x(\tau-1)) \cdot w|_{x(0)=0} = x(t) \cdot w$. Now, suppose that the weight matrix is also time-varying. We now have $s(t) := \sum_{\tau=1}^{t}(x(\tau) - x(\tau-1)) \cdot w(\tau)|_{x(0)=0}$. Unless $w(\tau)$ is static, i.e. $w(\tau) = w(1) \forall \tau$, we generally have $s(t) \neq x(t) \cdot w(t)$. In fact, our error, $\|s(t) - x(t) \cdot w(t)\|$ will accumulate over time. Still, it seems that if the data is temporally redundant, and the weights are changing slowly in time, then we should be able to exploit that redundancy to reduce inter-neuron communication.

The solution we propose is a compromise: send a linear combination of the rate of change and the value of the signal: $a(t) = k_p x(t) + k_d(x(t) - x(t-1))$. We can reconstruct this on the other end as an exponential-moving average: $s(t) = \frac{a(t) \cdot w(t) + k_d s(t-1)}{k_p + k_d} \approx x(t) \cdot w(t)$. As before we have $s(t) \neq x(t) \cdot w(t)$ unless the weights are fixed. However, now, the error will not accumulate without bound - it will be small when weights are changing more slowly or when k_d is small. When we add quantization back in by quantizing $a(t)$

before the weight multiplication, we find that the k_p, k_d parameters give us a trade-off between accuracy (in terms of faithfulness to the unquantized function of the network) and computation (in terms of number of arithmetic operations required to update).

In the review process, a reviewer rightly pointed out that what we are doing is actually an instance of an idea that is widespread in the signal processing literature: If we re-arrange the equation for our encoding scheme to $a(t) = (k_p + k_d)\left(x(t) - \frac{k_d}{k_p+k_d}x(t-1)\right)$ we see that this is an instance of *Linear Predictive Coding.* The idea of predictive coding is to subtract off the predictable component of a signal before transmitting it, then reconstructing it on the other end. In this case, we have an extremely simple scheme where our "prediction" of $x(t)$ is $\frac{k_d}{k_p+k_d}x_{t-1}$. When $\frac{k_d}{k_p+k_d}$ is close to 1, this encodes our belief that the signal will not change much between time-steps. In order for the encoding/decoding scheme to be stable (for our reconstructions not to accumulate error over time) we must have $\frac{k_d}{k_p+k_d} < 1$. When we add quantization after encoding, it becomes known as *lossy predictive coding.*

In the paper, we use this scheme to encode, quantize and decode communication between layers of a network. We train this network using backpropagation on temporal data, with both the forward and backward passes quantizing their communication according to this scheme. We demonstrate that when we train this network on temporal data using this encoding/decoding scheme, we can get a modest decrease in inter-neuron communication with no noticeable change to learning performance.

What is interesting in this paper is that by pursuing the objective of more efficient training and inference on temporal data, we end up with an architecture with characteristics reminiscent of biological neurons. In our decoding scheme, neurons take an exponential running average of their inputs (reminiscent of Equation 1.9 in the Introduction, which describes a basic model of a biological neuron). Another side effect of this scheme is that our learning rule becomes a form of Spike-Timing dependent plasticity - a rule observed in neuroscience wherein the change in weight on a synapse is a function of the relative timing of the spikes from the presynaptic neuron and postsynaptic neuron (see Section 1.4.1) - albeit of a different (symmetrical) form than the classic version observed in neuroscience. We expand on this connection in Chapter 6.

5.1. Abstract

The vast majority of natural sensory data is temporally redundant. For instance, video frames or audio samples which are sampled at nearby points in time tend to have similar values. Typically, deep learning algorithms take no advantage of this redundancy to reduce computations. This can be an obscene waste of energy. We present a variant on backpropagation for neural networks in which computation scales with the rate of change of the data - not the rate at which we process the data. We do this by implementing a form of Predictive Coding wherein neurons communicate a combination of their state, and

their temporal change in state, and quantize this signal using Sigma-Delta modulation. Intriguingly, this simple communication rule give rise to units that resemble biologically-inspired leaky integrate-and-fire neurons, and to a spike-timing-dependent weight-update similar to Spike-Timing Dependent Plasticity (STDP), a synaptic learning rule observed in the brain. We demonstrate that on MNIST, on a temporal variant of MNIST, and on Youtube-BB, a dataset with videos in the wild, our algorithm performs about as well as a standard deep network trained with backpropagation, despite only communicating discrete values between layers.

5.2. Introduction

Currently, most algorithms used in Machine Learning work under the assumption that data points are independent and identically distributed, as this assumption provides good statistical guarantees for convergence. This is very different from the way data enters our brains. Our eyes receive a single, never-ending stream of temporally correlated data. We get to use this data once, and then it's gone. Moreover, most sensors produce sequential, temporally redundant streams of data. This can be both a blessing and a curse. From a statistical learning point of view this redundancy may lead to biased estimators when used to train models which assume independent and identically distributed input data. However, the temporal redundancy also implies that intuitively not all computations are necessary.

Online Learning is the study of how to learn in this domain - where data becomes available in sequential order and is given to the model only once. Given the enormous amount of sequential data, mainly videos, that are being produced nowadays, it seems desirable to develop learning systems that simply consume data on-the-fly as it is being generated, rather than collect it into datasets for offline-training. There is, however a problem of efficiency, which we hope to illustrate with two examples:

1. *CCTV feeds.* CCTV Cameras collect an enormous amount of data from mostly-static scenes. The amount of new information in a frame, given the previous frame, tends to be low, *i.e.* the data tends to be temporally redundant. If we want to train a model from of this data (for example a pedestrian detector), we need to process a large amount of mostly-static frames. If the frame rate doubles, so does the amount of computation. Intuitively, it feels that this should not be necessary. It would be nice to still be able to use all this data, but have the amount of computation scale with the amount of new information in each frame, not just the number of frames and dimensions of the data.

2. *Robot perception.* Robots have no choice but to learn online - their future input data (*e.g.* camera frames) are dependent on their previous predictions (*i.e.* motor actions). Not only does their data come in nonstationary temporal streams, but it typically comes from several sensors running at different rates. The camera

> may produce 1MB images at 30 frames/s, while the gyroscope might produce 1-byte readings at 1000 frames/s. It is not obvious, using current methods in deep learning, how we can integrate asynchronous sensory signals into a unified, trainable, latent representation, without undergoing the inefficient process of recomputing the function of the network every time a new signal arrives.

These examples point to the need for a training method where the amount of computation required to update the model scales with the amount of new information in the data, and not just the dimensionality of the data.

There has been a lot of work on increasing the computational efficiency of neural networks by quantizing neural weights or activations (see Section 5.5), but comparatively little work on exploiting redundancies in the data to reduce the amount of computation. O'Connor and Welling [2016b], set out to exploit the temporal redundancy in video by having neurons only send their quantized *changes* in activation to downstream neurons, and having the downstream neurons integrate these changes over time. This approach (take the temporal difference, multiply by weights, temporally integrate) works for efficiently approximating the function of the network, but fails for training. The reason for this failure is that when the weights are functions of time, we no longer reconstruct the correct activation for the next layer. In other words, given a sequence of inputs $x_0...x_t$ with $x_0 = 0$ and weights $w_1...w_t$: $\sum_{\tau=1}^{t}(x_\tau - x_{\tau-1}) \cdot w_\tau \neq x_t \cdot w_t$ unless $w_t = w_0 \forall t$. Figure 5.2 describes the problem visually.

In this paper, we correct for this problem by encoding a mixture of two components of the layers activation x_t: the *proportional* component $k_p x_t$, and the *derivative* component $k_d(x_t - x_{t-1})$. When we invert this encoding scheme, we get a decoding scheme which corresponds to taking an exponentially decaying temporal average of past inputs. Interestingly, the resulting neurons begin to resemble models of biological spiking neurons, whose membrane potentials can approximately be modeled as an exponentially decaying temporal average of past inputs.

In this work, we present a scheme wherein the temporal redundancy of input data is used to reduce the computation required to train a neural network. We demonstrate this on the MNIST and Youtube-BB datasets. To our knowledge we are the first to create a neural network training algorithm which uses less computation as data becomes more temporally redundant.

5.3. Methods

We propose a coding scheme where neurons can represent their activations as a temporally sparse series of impulses. The impulses from a given neuron encode a combination of the value and the rate of change of the neuron's activation.

While our algorithm is designed to work efficiently with *temporal data*, we do not aim to learn *temporal sequences* in this work. We aim to efficiently approximate a function $y_t = f(x_t)$, where the current target y_t is solely a function of the current input x_t, and not previous inputs $x_0 ... x_{t-1}$. The temporal redundancy between neighbouring inputs x_{t-1}, x_t will however be used to make our approximate computation of this function more efficient.

5.3.1. Preliminary

Throughout this paper we will use the notation $(f_3 \circ f_2 \circ f_1)(x) = f_3(f_2(f_1(x)))$ to denote function composition. We slightly abuse the notion of functions by allowing them to have an internal state which persists between calls. For example, we define the Δ function in Equation 5.1 as being the difference between the inputs in two consecutive calls (where persistent variable x_{last} is initialized to 0). The Σ function, defined in Equation 5.2, returns a running sum of the inputs over calls. So we can write, for example, that when our composition of functions $(\Sigma \circ \Delta)$ is called with a sequence of input variables $x_1 ... x_t$, then $(\Sigma \circ \Delta)(x_t) = x_t$, because $(x_1 - x_0) + (x_2 - x_1) + ... + (x_t - x_{t-1})|_{x_0=0} = x_t$.

In general, when we write $y_t = f(x_t)$, where f is a function with persistent state, it will be implied that we have previously called $f(x_\tau)$ for $\tau \in [1, .., t-1]$ in sequence. Variable definitions that are used later will be highlighted in blue. While all terms are defined in the paper, we encourage the reader to refer to Appendix C.1 for a complete collection of definitions and notations.

5.3.2. Position-Derivative (PD) Encoding

Suppose a neuron has time-varying activation $x_1 .. x_t$. Taking inspiration from Proportional-Integral-Derivative (PID) controllers, we can "encode" this activation at each time step as a combination of its current activation and change in activation as $a_t \triangleq enc(x_t) = k_p x_t + k_d(x_t - x_{t-1})$, (see Equation 5.4). The parameters k_p and k_d determine what portion of our encoding represents the value of the activation and the rate of change of that value, respectively. In Section 5.5, we discuss how this is a form of *Predictive Coding* and in Appendix C.5, we discuss the effect our choices for these parameters have on the network.

$$\begin{aligned} &\Delta : x \mapsto y; \quad \text{Persistent: } x_{last} \leftarrow 0 \\ &\quad y \leftarrow x - x_{last} \\ &\quad x_{last} \leftarrow x \end{aligned} \tag{5.1}$$

$$\begin{aligned} &\Sigma : x \mapsto y; \quad \text{Persistent: } y \leftarrow 0 \\ &\quad y \leftarrow y + x \end{aligned} \tag{5.2}$$

$$\begin{aligned} &Q : x \mapsto y; \quad \text{Persistent: } \phi \leftarrow 0 \\ &\quad \phi' \leftarrow \phi + x \\ &\quad y \leftarrow round(\phi') \\ &\quad \phi \leftarrow \phi' - y \end{aligned} \tag{5.3}$$

$$\begin{aligned} &enc : x \mapsto y; \quad \text{Persistent: } x_{last} \leftarrow 0 \\ &\quad y \leftarrow k_p x + k_d(x - x_{last}) \\ &\quad x_{last} \leftarrow x \end{aligned} \tag{5.4}$$

$$\begin{aligned} &dec : x \mapsto y; \quad \text{Persistent: } y \leftarrow 0 \\ &\quad y \leftarrow \frac{x + k_d y}{k_p + k_d} \end{aligned} \tag{5.5}$$

To derive our decoding formula, we can

simply solve for x_t as $x_t = \frac{a_t + k_d x_{t-1}}{k_p + k_d}$ (Equation 5.5), such that $(dec \circ enc)(x_t) = x_t$. Notice that Equation 5.5 corresponds to decaying the previous decoder state by some constant $k_d/(k_p + k_d)$ followed by adding the input $a_t/(k_p + k_d)$. We can expand this recursively to see that this corresponds to a temporal convolution $a * \kappa$ where κ is a causal exponential kernel $\kappa_\tau = \left\{ \frac{1}{k_p+k_d} \left(\frac{k_d}{k_d+k_p} \right)^\tau \text{ if } \tau \geq 0 \text{ otherwise } 0 \right\}$.

5.3.3. Quantization

Our motivation for the aforementioned encoding scheme is that we want a sparse signal which can be quantized into a low bitrate discrete signal. This will later be used to reduce computation. We can quantize our signal a_t into a sparse, integer signal $s_t \triangleq Q(a_t)$, where the quantizer Q is defined in Equation 5.3. Equation 5.3 implements a form of Sigma-Delta modulation, a method widely used in signal processing to approximately communicate signals at low bit rates [Candy and Temes, 1962]. We can show that $Q(x_t) = (\Delta \circ R \circ \Sigma)(x_t)$ (see Supplementary Material Section C.3), where $\Delta \circ R \circ \Sigma$ indicates applying a temporal summation, a rounding, and a temporal difference, in series. If x_t is temporally redundant and we set k_p to be small, then $|a_t| \ll 1 \forall t$, and we can expect s_t to consist of mostly zeros with a few 1's and -1's.

We can now approximately reconstruct our original signal x_t as $\hat{x}_t \triangleq dec(s_t)$ by applying our decoder, as defined in Equation 5.5. As our coefficients k_p, k_d become larger, our reconstructed signal $\hat{x}_t$ should become closer to the original signal x_t. We illustrate examples of encoded signals and their reconstructions for different k_p, k_d in Figure 5.1.

Special cases

We can compactly write the entire reconstruction function as $\hat{x} = (dec \circ \Delta \circ R \circ \Sigma \circ enc)(x_t)$.

$k_p = 0$: When $k_p = 0$, we get $dec(x_t) = (k_d^{-1} \circ \Sigma)(x_t)$ and $enc(x_t) = (k_d \circ \Delta)(x_t)$, so our reconstruction reduces to $\hat{x} = (k_d^{-1} \circ \Sigma \circ \Delta \circ R \circ \Sigma \circ k_d \circ \Delta)(x_t)$. Because $\Sigma \circ k_d \circ \Delta$ all commute with one another, we can simplify this to $\hat{x}_t = (k_d^{-1} \circ R \circ k_d)(x_t)$. so our decoded signal is $\hat{x}_t = round(x_t \cdot k_d)/k_d$, with no dependence on x_{t-1}. This is visible in the bottom row of Figure 5.1. This was the encoding scheme used in O'Connor and Welling [2016b].

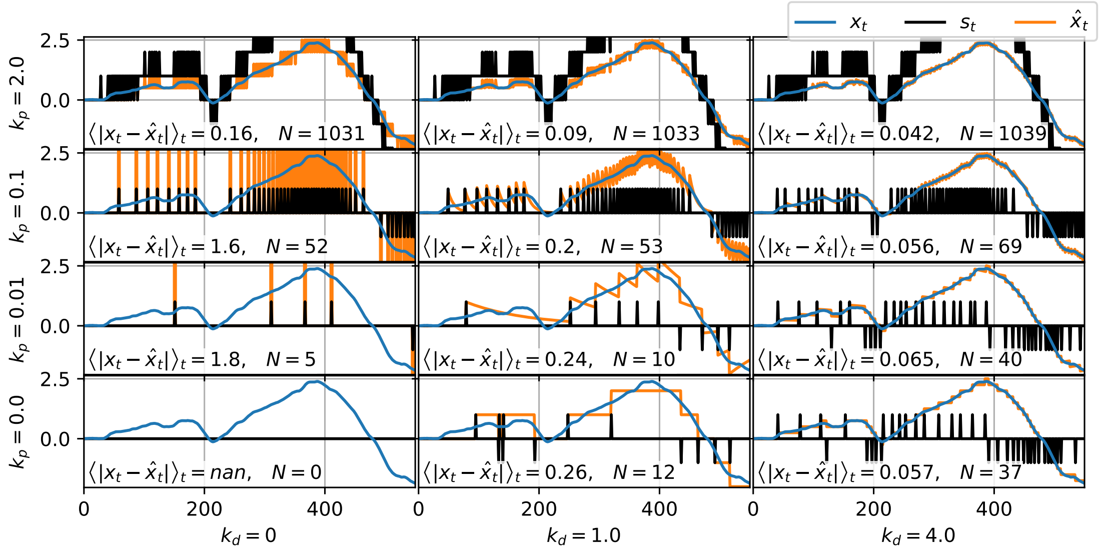


Figure 5.1.: An example signal x_t (blue), encoded with k_p varying across rows and k_d varying across columns. s_t (black) is the quantized signal produced by the successive application of encoding (Equation 5.4) and quantization (Equation 5.3, where N indicates the total number of spikes. $\hat{x}_t$ (orange) is the reconstruction of x_t produced by applying Equation 5.5 to s_t. One might, after a careful look at this figure, ask why we bother with the proportional (k_p) term at all? Figure 2 anticipates this question and answers it visually.

$k_d = 0$: In this case, $dec(x_t) = k_p^{-1} x_t$ and $enc(x_t) = k_p x_t$ so our encoding-decoding process becomes $\hat{x} = (k_p^{-1} \circ \Delta \circ R \circ \Sigma \circ k_p)(x_t)$. Neither our encoder nor our decoder have any memory, and we take no advantage of temporal redundancy.

5.3.4. Sparse Communication Between Layers

The purpose of our encoding scheme is to reduce computation by sparsifying communication between layers of a neural network. Our approach is to approximate the matrix-product as a series of additions, where the number of additions is inversely proportional to the sparsity of the input data. Suppose we are trying to compute the pre-nonlinearity activation of the first hidden layer, $z_t \in \mathbb{R}^{d_{out}}$, given the input activation, $x_t \in \mathbb{R}^{d_{in}}$. We approximate z_t as:

$$z_t \triangleq x_t \cdot w_t \approx \hat{x}_t \cdot w_t \triangleq dec(Q(enc(x_t))) \cdot w_t \triangleq dec(s_t) \cdot w_t \approx dec(s_t \cdot w_t) \triangleq \hat{z}_t \\ \text{where: } x_t, \hat{x}_t \in \mathbb{R}^{d_{in}}; s_t \in \mathbb{Z}^{d_{in}}; w \in \mathbb{R}^{d_{in} \times d_{out}}; z_t, \hat{z}_t \in \mathbb{R}^{d_{out}} \tag{5.7}$$

The first approximation comes from the quantization (Q) of the encoded signal, and the

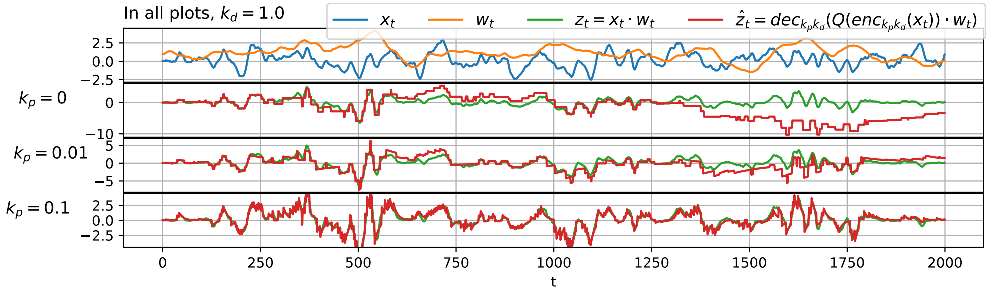


Figure 5.2.: The problem with *only* sending changes in activation (i.e. $k_p = 0$) is that during training, weights change over time. **Top:** we generate random signals for a single scalar activation x_t and scalar weight w_t. **Row 2:** We efficiently approximate z_t by taking the temporal difference, multiplying by w_t then temporally integrating, to produce $\hat{z}_t$, as described in Section 5.3.4. As the weight w_t changes over time, our estimate $\hat{z}$ diverges from the correct value. **Rows 3, 4:** Introducing k_p allows us to bring our reconstruction back in line with the correct signal.

second from the fact that the weights change over time, as explained in Figure 5.2. The effects of these approximations are further explored in Appendix C.5.1.

Computing z_t takes $d_{in} \cdot d_{out}$ multiplications and $(d_{in} - 1) \cdot d_{out}$ additions. The cost of computing $\hat{z}_t$, on the other hand, depends on the contents of s_t. If the data is temporally redundant, $s_t \in \mathbb{Z}^{d_{in}}$ should be sparse, with total magnitude $N \triangleq \sum_i |s_{t,i}|$. s_t can be decomposed into a sum of one-hot vectors $s_t = \sum_{n=1}^{N} sign(s_{t,i_n}) \cdot \gamma_{i_n} : i_n \in [1..d_{i_n}]$ where γ_{i_n} is a length-d_{in} one-hot vector with element $(\gamma_{i_n})_{i_n} = 1$. The matrix product $s_t \cdot w$ can then be decomposed into a series of row additions:

$$s_t \cdot w = \left(\sum_{n=1}^{N} \text{sign}(s_{t,i_n}) \cdot \gamma_{i_n} \right) \cdot w = \sum_{n=1}^{N} \text{sign}(s_{t,i_n}) \gamma_{i_n} \cdot w = \sum_{n=1}^{N} \text{sign}(s_{t,i_n}) \cdot w_{i_n,\cdot} \tag{5.8}$$

If we include the encoding, quantization, and decoding operations, our matrix product takes a total of $2d_{in} + 2d_{out}$ multiplications, and $\sum_n |s_{t,n}| \cdot d_{out} + 3d_{in} + d_{out}$ additions. Assuming the $\sum_n |s_{t,n}| \cdot d_{out}$ term dominates, we can say that the relative cost of computing $\hat{z}_t$ vs z_t is:

$$\frac{cost(\hat{z})}{cost(z)} \approx \frac{\sum_n |s_{t,n}| \cdot cost(add)}{d_{in} \cdot (cost(add) + cost(mult))} \tag{5.9}$$

5.3.5. A Neural Network

We can implement this encoding scheme on every layer of a neural network. Given a standard neural net f_{nn} consisting of alternating linear $(\cdot w_l)$ and nonlinear (h_l) operations, our network function f_{pdnn} can then be written as:

$$f_{nn}(x) = (h_L \circ \cdot w_L \circ ... \circ h_1 \circ \cdot w_1)(x) \tag{5.10}$$

$$f_{pdnn}(x) = (h_L \circ dec_L \circ w_L \circ Q_L \circ enc_L \circ ... \circ h_1 \circ dec_1 \circ \cdot w_1 \circ Q_1 \circ enc_1)(x) \tag{5.11}$$

We can use the same approach to approximately calculate our gradients to use in training. If we define our layer activations as $\hat{z}_l \triangleq (dec \circ \cdot w_l \circ Q \circ enc)(x)$ if $l = 1$ otherwise $(dec \circ \cdot w_l \circ Q \circ enc)(\hat{z}_{l-1})$, and $\mathcal{L} \triangleq \ell(f_{pdnn}(x), y)$, where ℓ is some loss function and y is a target, we can backpropagate the approximate gradients as:

$$\widehat{\frac{\partial \mathcal{L}}{\partial \hat{z}_l}} = \begin{cases} \frac{\partial \mathcal{L}}{\partial z_L} & \text{if } l = L \\ \left(\odot h_l'(\hat{z}_l) \circ dec_l^{back} \circ \cdot w_{l+1}^T \circ Q_{l+1}^{back} \circ enc_{l+1}^{back}\right) \left(\widehat{\frac{\partial \mathcal{L}}{\partial \hat{z}_{l+1}}}\right) & \text{otherwise} \end{cases} \tag{5.12}$$

This can be implemented by either executing a (sparse) forward and backward pass at each time-step, or in an "event-based" manner, where the quantizers fire "events" whenever incoming events push their activations past a threshold, and these events are in turn sent to downstream neurons. For ease of implementation, we opt for the former in our code. Note that unlike in regular backprop, computing these forward and backward passes results in changes to the internal state of the *enc*, *dec*, and Q components.

5.3.6. Parameter Updates

There is no use in having an efficient backward pass if the parameter updates are not also efficient. In a normal neural network trained with backpropagation and simple stochastic gradient descent, the parameter update for weight matrix w has the form $w \leftarrow w - \eta \frac{\partial \mathcal{L}}{\partial w}$ where η is the learning rate. If w connects layer $l-1$ to layer l, we can write $\frac{\partial \mathcal{L}}{\partial w} = x_t \otimes e_t$ where $x_t \triangleq h_{l-1}(z_{l-1,t}) \in \mathbb{R}^{d_{in}}$ is the presynaptic (layer $l-1$) activation, $e_t \triangleq \frac{\partial \mathcal{L}}{\partial z_{l,t}} \in \mathbb{R}^{d_{out}}$ is the postsynaptic (layer l) backpropagating gradient and $\otimes$ is the outer product. So we require $d_{in} \cdot d_{out}$ multiplications to update the parameters for each sample.

We want a more efficient way to compute this product, which takes advantage of the sparsity of our encoded signals to reduce computation. We can start by applying our encoding-quantizing-decoding scheme to our input and error signals as $\bar{x}_t \triangleq (Q \circ enc)(x_t) \in \mathbb{Z}^{d_{in}}$ and $\bar{e}_t \triangleq (Q \circ enc)(e_t) \in \mathbb{Z}^{d_{out}}$, and approximate our true update as $\widehat{\frac{\partial \mathcal{L}}{\partial w}}_{recon,t} \triangleq \hat{x}_t \otimes \hat{e}_t$ where $\hat{x}_t \triangleq dec(\bar{x}_t)$ and $\hat{e}_t \triangleq dec(\bar{e}_t)$. This does not do any good by itself, because the vectors involved in the outer product, $\hat{x}_t$ and $\hat{e}_t$, are still not sparse. However, we can

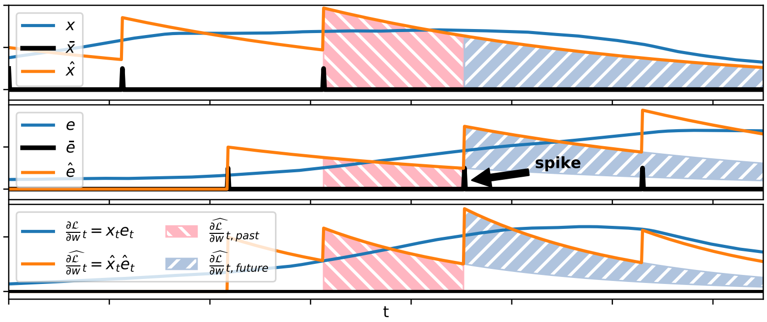


Figure 5.3.: A visualization of our efficient update schemes from Section 5.3.6. **Top**: A scalar signal representing a presynaptic neuron activation $x_t = h_{l-1}(z_l - 1)$, its quantized version, $\bar{x}_t = (Q \circ enc)(x_t)$, and its reconstruction $\hat{x}_t = dec(\bar{x}_t)$. **Middle**: Another signal, representing the postsynaptic gradient of the error $e = \frac{\partial \mathcal{L}}{\partial z_l}$, along with its quantized ($\bar{e}$) and reconstructed ($\hat{e}$) variants. **Bottom**: The true weight gradient $\frac{\partial \mathcal{L}}{\partial w_t}$ and the reconstruction gradient $\widehat{\frac{\partial \mathcal{L}}{\partial w_t}}$. At the time of the spike in $\bar{e}_t$, we have two schemes for efficiently computing the weight gradient that will be used to increment weight (see Section 5.3.6). The *past* scheme computes the area under $\hat{x} \cdot \hat{e}$ since the last spike, and the *future* scheme computes the total future additional area due to the current spike.

exactly compute the sum of this value over time using one of two sparse update schemes - *past updates* and *future updates* - which are depicted in Figure 5.3. We give the formula for the Past and Future update rules in Appendix C.4, but summarize them here:

Past Updates: For a given synapse $w_{i,j}$, if either the presynaptic neuron spikes ($\bar{x}_{t_i} \neq 0$) or the postsynaptic neuron spikes ($\bar{e}_{t_i} \neq 0$), we increment the $w_{i,j}$ by the total area under $\hat{x}_{\tau,i}\hat{e}_{\tau,j}$ since the last spike. We can do this efficiently because between the current time and the time of the previous spike, $\hat{x}_{\tau,i}\hat{e}_{\tau,j}$ is a geometric sequence. Given a known initial value u, final value v, and decay rate r, a geometric sequence sums to $\frac{u-v}{1-r}$. The area calculated is shown in pink on the bottom row of Figure 5.3, and one algorithm to calculate it is in Equation C.6 in Appendix C.4.

Future Updates: Another approach is to calculate the Present Value of the future area under the integral from the current spike. This is depicted in the blue-gray area in Figure 5.3, and the formula is in Equation C.7 in Appendix C.4.

Finally, because the magnitude of our gradient varies greatly over training, we create a scheme for adaptively tuning our k_p, k_d parameters to match the running average of the magnitude of the data. This is described in detail in Appendix C.5.

5.3.7. Relation to STDP

An extremely attentive reader might have noted that Equation C.7 has the form of an online implementation of Spike-Timing Dependent Plasticity (STDP). STDP [Markram et al., 2012] emerged from neuroscience, where it was observed that synaptic weight changes appeared to be functions of the relative timing of pre- and post-synaptic spikes. The empirically observed function usually has the double-exponential form seen on the rightmost plot of Figure 5.4.

Using the quantized input signal $\bar{x}$ and error signal $\bar{e}$, and their reconstructions $\hat{x}_t$ and $\hat{e}_t$ as defined in the last section, we define a causal convolutional kernel $\kappa_t = \left\{k_\beta \left(k_\alpha\right)^t \text{ if } t \geq 0 \text{ otherwise } 0\right\}$, where $k_\alpha \triangleq = \frac{k_d}{k_p+k_d}$, $k_\beta \triangleq \frac{1}{k_p+k_d}$. We can then define a "cross-correlation kernel" $g_t = \{\kappa_t \text{ if } t \geq 0 \text{ otherwise } \kappa_{-t}\} = k_\beta (k_\alpha)^{|t|} : t \in \mathbb{Z}$ which defines the magnitude of a parameter update as a function of the difference in timing between pre-synaptic spikes from the forward pass and post-synaptic spikes from the backward pass. The middle plot of Figure 5.4 is a plot of g. We define our STDP update rule as:

$$\frac{\widehat{\partial \mathcal{L}}}{\partial w_{t,STDP}} = \left(\sum_{\tau=-\infty}^{\infty} \bar{x}_{t-\tau} g_\tau \right) \otimes \bar{e}_t \tag{5.13}$$

We note that while our version of STDP has the same double-exponential form as the classic STDP rule observed in neuroscience [Markram et al., 2012], our "presynaptic" spikes come from the forward pass while our "postsynaptic" spikes come from the *backwards* pass. STDP is not normally used to as a learning rule networks trained by backpropagation, so the notion of forward and backward pass with a spike-timing-based learning rule are new. Moreover, unlike in classic STDP, we do not have the property that sign of the weight change depends on whether the presynaptic spike preceded the postsynaptic spike.

In Section C.4 in the supplementary material we show experimentally that while Equations $\frac{\widehat{\partial \mathcal{L}}}{\partial w_{recon}}$, $\frac{\widehat{\partial \mathcal{L}}}{\partial w_{past}}$, $\frac{\widehat{\partial \mathcal{L}}}{\partial w_{future}}$, $\frac{\widehat{\partial \mathcal{L}}}{\partial w_{stdp}}$ may all result in different updates at different times, the rules are equivalent in that for a given set of pre/post-synaptic spikes $\bar{x}, \bar{e}$, the cumulative sum of their updates over time converges exactly.

5.4. Experiments

5.4.1. Temporal MNIST

To evaluate our network's ability to learn, we train it on the standard MNIST dataset, as well as a variant we created called "Temporal MNIST". Temporal MNIST is simply a reshuffling of the MNIST dataset so that so that similar inputs (in terms of L2-pixel distance), are put together. Figure 5.6 shows several snippets of consecutive frames in

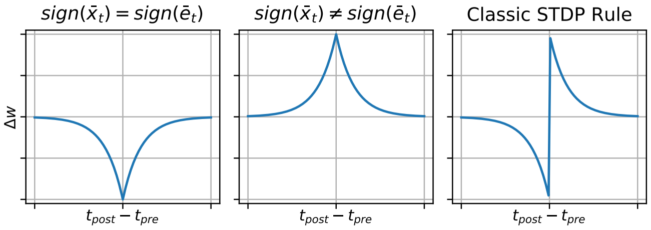


Figure 5.4.: **Left**: Our STDP rule, when both the input and error spikes have the same sign. **Middle**: Our STDP rule, when the input and error spikes have opposite signs. **Right**: The classic STDP rule Markram et al. [2012], where the weight update is positive when a presynaptic spike preceeds a postsynaptic spike, and negative otherwise.

the temporal MNIST dataset. We compare our Proportional-Derivative Net against a conventional Multi-Layer Perceptron with the same architecture (one hidden layer of 200 ReLU hidden units and a softmax output). The results are shown in Figure 5.5. Somewhat surprisingly, our predictor slightly outperformed the MLP, getting 98.36% on the test set vs 98.25% for the MLP. We assume this improvement is due to the regularizing effect of the quantization. On Temporal MNIST, our network was able to converge with less computation than it required for MNIST (it used $32 \cdot 10^{12}$ operations for MNIST vs $15 \cdot 10^{12}$ for Temporal MNIST), but ended up with a slightly worse test score when compared with the MLP (the PDNN achieved 97.99% vs 98.28% for the MLP). The slightly higher performance of the MLP on Temporal MNIST may be explained by the fact that the gradients on Temporal MNIST tend to be correlated across time-steps, so weights will tend to move in a single direction for a number of steps, which will interfere with the PDNN's ability to accurately track layer activations (see Figure 5.2). Appendix C.6 contains a table of results with varying hyperparameters.

5.4.2. YouTube Video Dataset

Next, we want to simulate the setting of CCTV cameras, discussed in Section 5.2, where we have a lot of data with only a small amount of new information per frame. In the absence of large enough public CCTV video datasets, we investigate the surrogate task of frame-based object classification on wild YouTube videos from the large, recently released Youtube-BB dataset Real et al. [2017]. Our subset consists of 358 Training Videos and 89 Test videos with 758,033 frames in total. Each video is labeled with an object in one of 24 categories.

We start from a VGG19 network [Simonyan and Zisserman, 2014]: a 19-layer convolutional network pre-trained on imagenet. We replace the top three layer with three of our own randomly initialized layers, and train the network both as a spiking network, and as a regular network with backpropagation. While training the entire spiking network end-to-end works, we choose to only train the top layers, in order to speed up our training

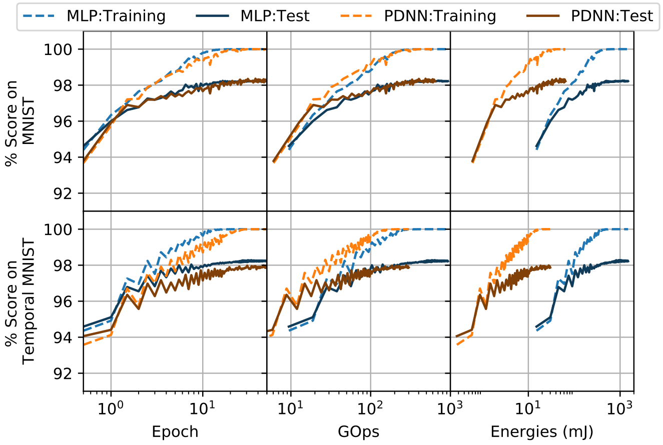


Figure 5.5.: Top Row: Results on MNIST. Bottom Row: Results on Temporal MNIST. Left Column: the training and test scores as a function of epoch. Middle: We now put the number of computational operations on the x-axis. We see that as a result our PDNN shifts to the left. Right: Because our network computes primarily with additions rather than multiplications. When we multiply our operation counts with the estimates of Horowitz [2014] for the computational costs of arithmetic operations (0.1pJ for 32-bit fixed-point addition vs 3.2pJ for multiplication), we can see that our algorithm would be at an advantage on any hardware where arithmetic operations were the computational bottleneck.

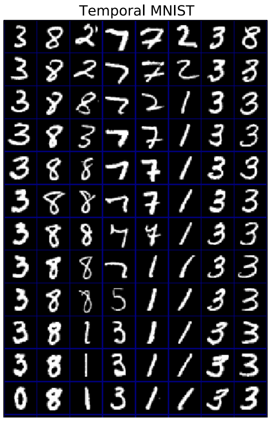


Figure 5.6.: Some samples from the Temporal-MNIST dataset. Each column shows a snippet of adjacent frames.

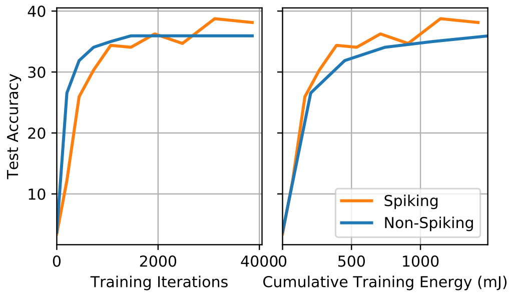


Figure 5.7.: Left: Learning Curves on the Youtube Dataset. Right: Learning curves with respect to computational energy using the conversion of Horowitz [2014]. The spiking network slightly outperforms the non-spiking baseline - we suspect that this is because the added noise of spiking acts as a regularizer.

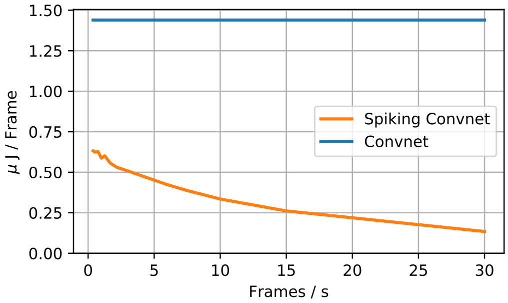


Figure 5.8.: We simulate different frame-rates by selecting every n'th frame. This plot shows our network's mean computation over several snippets of video, at varying frame rates. As our frame rate increases, the computation per-frame of our spiking network goes down, while with a normal network, it remains fixed.

time.

We compare our training scores and computation between a spiking and non-spiking implementation. The learning curves in Figure 5.7 show that our spiking network performs comparably to a non-spiking network, and Figure 5.8 shows how the computation per frame of our spiking network decreases as we increase the frame rate (i.e. as the input data becomes more temporally redundant). Because our spiking network uses only additions, while a regular deep network does multiply-adds, we use the estimated energy-costs per op of Horowitz [2014] to compare computations to a single scale, which estimates the amount of energy required to do multiplies and adds in fixed-point arithmetic.

5.5. Related Work

Noise-Shaping is a quantization technique that aims to increase the fidelity of signal reconstructions, per unit of bandwidth of the encoded signal, by quantizing the signal in such a way that the quantization noise is pushed into a higher frequency band which is later filtered out upon decoding. Sigma-Delta (also known as Delta-Sigma) quantization is a form of noise-shaping. Shin [2001] first suggested that biological neurons may be performing a form of noise shaping, and Yoon [2017] found standard spiking neuron models actually implement a form of Sigma-Delta modulation.

The encoding/decoding scheme we use in this paper can be seen as a form of Predictive Coding. Predictive coding is a lossless compression technique wherein the predictable parts of a signal are subtracted away so that just the unpredictable parts are transmitted. The idea that biological neurons may be doing some form of predictive coding was first proposed by Srinivasan et al. [1982]. In a predictive-coding neuron (unlike neurons commonly used in Deep Learning), there is a distinction between the signal that a neuron *represents* and the signal that it *transmits*. The neurons we use in this paper can be seen as implementing a simple form of predictive coding where the "prediction" is that the neuron maintains a decayed form of its previous signal - i.e. that $pred(x_t) \triangleq \frac{k_d}{k_p+k_d} x_{t-1}$ (See Appendix C.2 for detail). Chklovskii and Soudry [2012] suggest that the biological spiking mechanism may be thought of as consisting of a sigma-delta modulator within a predictive-coding circuit.

To our knowledge, none of the aforementioned work has yet been used in the context of deep learning.

There has been sparse but interesting work on merging the notions of spiking neural networks and deep learning. Diehl et al. [2015] found a way to efficiently map a trained neural network onto a spiking network. Lee et al. [2016] devised a method for training integrate-and-fire spiking neurons with backpropagation - though their neurons did not send a temporal difference of their activations. O'Connor and Welling [2016a] created a method for training event-based neural networks - but their method took no advantage of temporal redundancy in the data. Binas et al. [2016] and [O'Connor and Welling, 2016b] both took the approach of sending quantized temporal changes to reduce computation on temporally redundant data, but their schemes could not be used to train a neural network. Bohte et al. [2000] showed how one could apply backpropagation for training spiking neural networks, but it was not obvious how to apply the method to non-spiking data. Zambrano and Bohte [2016] developed a spiking network with an adaptive scale of quantization (which bears some resemblance to our tuning scheme described in Appendix C.5), and show that the spiking mechanism is a form of Sigma-Delta modulation, which we also use here. Courbariaux et al. [2015] showed that neural networks could be trained with binary weights and activations (we just quantize activations). Bengio et al. [2015a] found a connection between the classic STDP rule (Figure 5.4, right) and optimizing a dynamical neural network, although the way they arrived at an STDP-like rule was quite different from ours (they frame STDP as a way to minimze an objective based on the rate of change of the real-valued state of the network, whereas we use it approximately compute gradients based on spike-encodings of layer activations).

5.6. Discussion

We set out with the objective of reducing the computation in deep networks by taking advantage of temporal redundancy in data. We described a simple rule (Equation 5.4) for sparsifying the communication between layers of a neural network by having our neurons communicate a combination of their temporal change in activation, and the current value of their activation. We show that it follows from this scheme that neurons should behave as leaky integrators (Equation 5.5). When we quantize our neural activations with Sigma-Delta modulation, a common quantization scheme in signal processing, we get something resembling a leaky integrate-and-fire neuron. We derive efficient update rules for the weights of our network, and show these to be related to STDP - a learning rule first observed in neuroscience. Finally, we train our network, verify that it does indeed compute more efficiently on temporal data, and show that it performs about as well as a traditional deep network of the same architecture, but with significantly reduced computation. Finally, we show that our network can train on real video data.

The efficiency of our approach hinges on the temporal redundancy of our input data and neural activations. There is an interesting synergy here with the concept of slow-features [Wiskott, 1999]. Slow-Feature learning aims to discover latent objects that persist over time. If the hidden units were to specifically learn to respond to slowly-varying features of the input, the layers in a spiking implementation of such a network would have to communicate less often. In such a network, the tasks of feature-learning and reducing inter-layer communication may be one and the same.

Code is available at github.com/petered/pdnn.

Acknowledgments

This work was supported by Qualcomm, who we'd also like to thank for sharing their past work with us.

6. STDP is just Predictive Coding and Dynamics-Based learning

6.1. Abstract

Spike-Timing-Dependent Plasticity (STDP) is a synaptic learning rule observed in neuroscience, wherein the updates to a synaptic weight are a function of the relative timing of two spikes on the pre- and post-synaptic neuron. When pre precedes post, the weight is strengthened, when post precedes pre, the weight is weakened.

Bengio et al. [2015b] suggested that this rule would correlate to "dynamics-based learning", where a network's dynamics takes the place of the gradient in communicating how parameters should be updated.

Here, we show that if neurons communicate with a simple form of lossy predictive coding, that relationship becomes exact. We exactly reproduce the classic STDP learning rule when we apply dynamics-based learning on a network where neurons quantize their communication using lossy linear predictive coding.

6.2. Introduction

The previous chapter hinted at something interesting: Trying to efficiently compute weight updates in our network, when neurons communicated quantized values using a form of lossy predictive-coding, yielded an STDP-like double exponential kernel (see Figure 5.4). Unlike regular STDP, our kernel was symmetric, and our spikes were signed (could have a value of -1 or +1).

Coming from a completely different direction, Bengio et al. [2015a] (building on previous work by Xie and Seung [2000]), proposed how STDP may be a way of implementing a dynamics-based (as opposed to gradient-based) learning rule with spiking neurons. They show why STDP should produce results that correlate with dynamic-based learning, and demonstrate these empirically, but do not show an exact equivalence.

6.3. Background

6.3.1. Dynamics-Based Learning

In an ordinary deep network trained with backpropagation, the gradient update rule for a synapse connecting neuron i to neuron j is:

$$\Delta w_{ij} \propto -\frac{\partial \mathcal{L}}{\partial w_{ij}} = -h_i \frac{\partial \mathcal{L}}{\partial z_j} \tag{6.1}$$

Where $z_j := \sum_i w_{ij} h_i$ is the pre-nonlinearity activation; $h_i := h(z_i)$ is a post nonlinearity activation with $h(\cdot)$ being an elementwise nonlinearity such as sigmoid or ReLU; $\frac{\partial \mathcal{L}}{\partial z_j}$ is the gradient of the loss w.r.t. the activation of neuron j as computed by backpropagation.

As we have discussed in Chapter 1, this is not biologically plausible, because biological networks have no *secondary signalling mechanism* to backpropagate gradients. Hinton [2007] suggested that in biology, dynamics might take the place of negative-loss-gradients, and that the *secondary signal* is actually the temporal derivative of the activation. In other words, the negative gradient $-\frac{\partial \mathcal{L}}{\partial z_j}$ is replaced by the post-synaptic rate of change $\dot{z}_j := \frac{\partial z_j}{\partial t}$, and the synaptic learning rule becomes:

$$\Delta w_{ij}^{t_1:t_2} \propto \int_{t=t_1}^{t_2} h_i(t)\dot{z}_j(t) \approx \frac{1}{2}\sum_{t=t_1}^{t_2} h_i(t)\big(z_j(t+1) - z_j(t-1)\big) \tag{6.2}$$

Where $\Delta w_{ij}^{t_1:t_2}$ is the change in the strength of synapse ij between times t_1 and t_2. The implication is that if the dynamics of the network are moving in a direction that minimizes some loss, the parameter update will then also minimize that loss. The tricky part, not explored here but investigated in Scellier et al. [2018], is how to make those dynamics move in a direction that minimize the desired loss.

For ease of understanding, in the remainder of this chapter we shall use the discrete-time form of Equation 6.2, with the assumption that our chosen time-units (say ms for real neurons) are small enough to capture the dynamics in the signals.

6.3.2. Spike-Timing-Dependent Plasticity (STDP)

STDP is an observation from neuroscience Markram and Sakmann [1995] that the update to a synaptic weight of a spiking biological neuron appears to be a function of the relative timing of pre- and post-synaptic spikes. Specifically, if the pre-synaptic spike comes first, the weight strengthens, and if the post-synaptic spike comes first, it weakens. This rule is visualized in Figure 6.1, and can be modeled as anti-symmetrical double-exponential curve:

$$\Delta w_{ij} \propto \kappa^{STDP}(\Delta t) = \begin{cases} -a_- e^{\Delta t/\tau_-} & \text{if } \Delta t < 0 \\ 0 & \text{if } \Delta t = 0 \\ a_+ e^{-\Delta t/\tau_+} & \text{if } \Delta t > 0 \end{cases} \tag{6.3}$$

Where $a_- \in \mathbb{R}^+$ and $a_+ \in \mathbb{R}^+$ are the strengths of the anti-causal/causal components, $\Delta t = t_{post} - t_{pre}$ is the difference in timing between pre- and post- synaptic spikes, and τ_-, τ_+ are the anti-causal/causal time-constants.

Note that STDP can be expressed compactly as a convolution and an scalar-product. Suppose $s_i(t) \in \{0,1\}$ and $s_j(t) \in \{0,1\}$ are streams of spiking signals from a presynapic neuron i and postsynaptic neuron j. For every spike of postsynaptic neuron j at time t, we add up the weight changes from every spike of presynaptic neuron i at time τ :

$$\begin{aligned} \Delta w_{ij} &\propto \sum_{t:s_j(t)=1} \sum_{\tau:s_i(\tau)=1} \kappa^{STDP}(t-\tau) = \sum_t s_j(t) \sum_\tau s_i(\tau)\kappa^{STDP}(t-\tau) \\ &= s_j \cdot (\kappa^{STDP} * s_i) \in \mathbb{R} \end{aligned} \tag{6.4}$$

Where $(a * b)(t) := \sum_\tau a(\tau)b(t-\tau)$ denotes convolution and $a \cdot b := \sum_t a(t)b(t)$ denotes the scalar-product over time.

6.3.3. Linking dynamics-based learning to STDP

Xie and Seung [2000] were the first to observe a connection between the idea of an dynamics-based learning rule and STDP. Beginning with the STDP kernel, they show empirically that following an STDP-based learning rule will result in weight changes that look like those resulting from dynamics-based learning.

Bengio et al. [2015a] arrived to this same conclusion the other way around - they show that if one tries to implement dynamics-based learning with rate-coded spiking neurons (i.e. neurons that simply represent their value with the frequency of outgoing spikes), they empirically recover the STDP kernel in Figure 6.1. They also explain an intuition, illustrated in Figure 6.2 - as to why dynamics-based learning should produce similar results to STDP.

The bottom line, as flatly stated in the title of Bengio et al. [2015a]: *STDP as presynaptic activity times rate of change of postsynaptic activity*, is that STDP corresponds to an update rule of $\Delta w_{ij}^{STDP} \approx \int_t h_i \dot{h}_j$, where the post-nonlinearity activation $h_i := h(z_i)$ is interpreted as a "firing rate". Note that this does not quite correspond to Equation 6.2 because the post-synaptic activity z_j is replaced the post-nonlinearity activity h_j.

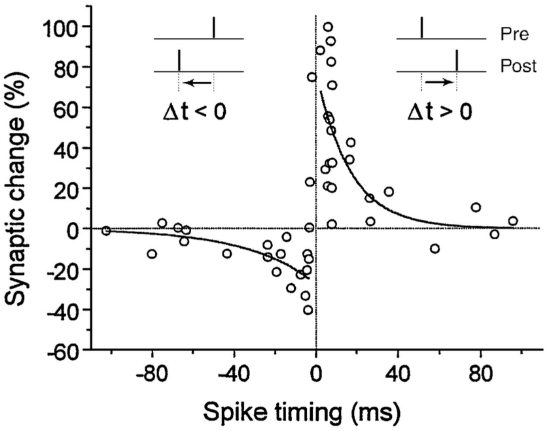


Figure 6.1.: The STDP learning rule (figure from Bi and Wang [2002]), showing the Synaptic change Δw_{ij} on the y-axis, as a function of the relative timing between the pre and post-synaptic spikes $\Delta t = t_{post} - t_{pre}$ on the x-axis. The anti-symmetrical double-exponential curve very roughly models data from neural recordings (points)

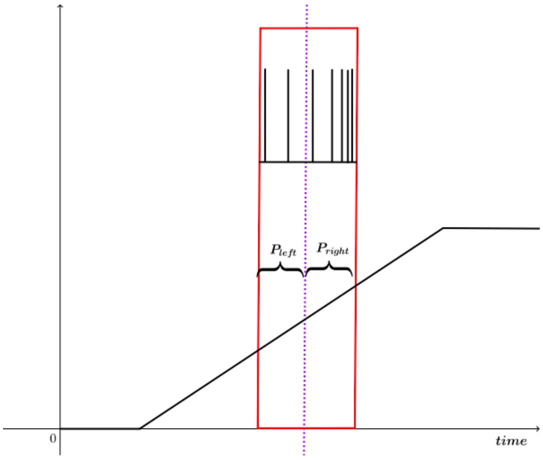


Figure 6.2.: An illustration from Bengio et al. [2015b] of why the Dynamics-based learning (Equation 6.2) correlates to STDP (Equation 6.3). The y-axis indicates the postsynaptic firing rate t_j, as a function of time, and the dotted vertical line indicates the time t_{pre} of a pre-synaptic spike. When, as in this figure, $\dot{h}_j(t) > 0$, we can see that a post-synaptic spike is more likely to occur in some window *after* the pre-synaptic spike than in the same window *before* it - causing a positive weight change in the STDP equation.

However, under the weak assumption that nonlinearity is monotonic, the updates should point in the same direction.

Both of these results are empirical. In this work we will show a mathematical connection between the anti-symmetrical double-exponential STDP kernel of Equation 6.3 and the Dynamics-Based learning in Equation 6.2. Specifically, we show that *Linear Predictive Coding* - a method for efficiently encoding signals from the signal processing literature Itakura [1972], is the link needed to connect them exactly, and that firing rate does not exactly correspond to “activity” $h(t)$, but to an encoded version of this activity $h(t) - \alpha h(t-1)$.

6.3.4. Predictive Coding

In Chapter 5 we described a network wherein neurons communicate their values (and backpropagate their gradients) using lossy predictive coding - a method for quantized communication of time-series data. The central idea of Predictive Coding is to subtract off the predictable parts of a signal before transmitting it so that the limited communication bandwidth is used to communicate the unpredictable components. The predictable component can then be reconstructed on the receiving end of the channel. Here we will describe the simplest form of predictive coding - first order linear predictive coding. [1]

Suppose we have a time-varying signal $x(t)$. We will assume discrete-time here, though everything here can be extended to continuous time. In first-order linear lossy predictive coding, we encode a bounded signal $x(t) \in [0, 1]$ into a bitstream $\overset{Q}{x}(t) \in \{0, 1\}$ as follows (also illustrated in Figure 6.3):

$$\begin{aligned}
\overset{E}{x}(t) &= x(t) - \overset{P}{x}(t-1) && \text{Subtract off predictable component} \\
\overset{Q}{x}(t) &= \text{Quantize}(\overset{E}{x}(t)) && \text{Quantize the encoded signal into a bitstream} \\
\overset{R}{x}(t) &= \overset{Q}{x}(t) + \overset{P}{x}(t-1) && \text{Reconstruct the signal} \\
\overset{P}{x}(t) &= \text{Predict}(\overset{R}{x}(t)) = \alpha\overset{R}{x}(t) && \text{Make a prediction of the next input}
\end{aligned} \tag{6.5}$$

Where $\overset{E}{x}(t)$ is the encoded signal, $\overset{Q}{x}(t) \in \{0, 1\}$ is the binary quantized signal, $\overset{R}{x}(t)$ is the reconstructed signal, and $\overset{P}{x}(t-1)$ is the prediction of the signal at time t based on the data available at time $t-1$. $\alpha \in [0, 1)$ is the predictive coding coefficient. Intuitively, when α is close to 1, we spend most of our bandwidth encoding *changes* to the signal, and when it is 0, we simply encode the current state of the signal. If α were > 1, our reconstructions would explode. This is called *First Order* Linear Predictive Coding because it is a special case of Linear Predictive Coding (LPC) Itakura [1972]. In $T'th$ order LPC, the prediction coefficients α form a vector and the prediction has the form $\overset{P}{x}(t) = \sum_{\tau=0}^{T-1} \alpha_\tau \overset{R}{x}(t-\tau)$. Note that in practice, to accurately represent a signal, $\overset{E}{x}(t)$ must be mostly in the range of $[0, 1]$, otherwise distortions will occur. This means $x(t)$ must be scaled by an appropriate constant before encoding and after decoding. For simplicity, we will assume here that this scaling of the signal has already been done as a preprocessing step.

[1] Note that this is different from the type of Predictive Coding discussed in the well-known work by Rao and Ballard [1999] where they hypothesize that in the brain, higher layers of neurons predict the activations of lower layers, and lower layers send up the difference between the top-down predictions and the bottom-up activations. It is also different from the *Contrastive Predictive Coding* of Oord et al. [2018], where a network learns a representation by predicting the output of the network at neighbouring time-steps. Here by contrast, prediction occurs *within the neuron*, takes a much simpler form, and has the function of signal-compression, not representation learning.

How the quantization of $\overset{E}{x}(t)$ into $\overset{Q}{x}(t)$ is done is not important for the results presented in this Chapter. Different methods of quantization are best for different objectives:

- **Thresholding**: Minimizes error at any given time but may have systematic biases over time:
$$\overset{Q}{x}(t) = \left[\overset{E}{x}(t) > \frac{1}{2}\right]$$
- **Random**: Each sample is an unbiased estimator of the encoded value. Will have higher error on average than thresholding but errors will cancel over time:
$$\overset{Q}{x}(t) = \left[\overset{E}{x}(t) > \mathcal{U}(0,1)\right]$$
- **Sigma Delta Modulation**: Maintains balance of total input and total output over time. Averaged error over time cancels faster than for Random sampling.
$$\begin{aligned}\overset{Q}{x}(t) &= \left[\phi(t-1) + \overset{E}{x}(t) > \frac{1}{2}\right]\Bigg|_{\phi(0):=0}\\ \phi(t) &= \phi(t-1) + \overset{E}{x}(t) - \overset{Q}{x}(t)\end{aligned}$$

Note that in the process of encoding the signal in Equation 6.5, we already computed $\overset{R}{x}(t)$, the reconstruction of $x(t)$. The decoder simply involves reusing the portion of the encoder that computes $\overset{R}{x}(t)$:

$$\begin{aligned}\overset{R}{x}(t) &= \overset{Q}{x}(t) + \overset{P}{x}(t-1)\\ \overset{P}{x}(t) &= \text{Predict}(\overset{R}{x}(t)) = \alpha\overset{R}{x}(t)\end{aligned} \tag{6.6}$$

Note that the Equation 6.6 can be unrolled so that it corresponds to a convolution of signal $\overset{Q}{x}$ with an infinite exponential kernel $\kappa_t^{pred} = (\alpha^{(t)}$ if $t \geq 0$ otherwise $0)$, where $\cdot^{(t)}$ denotes exponentiation by t. Written this way, we have:

$$\overset{R}{x} = \overset{Q}{x} * \kappa^{Pred} = \overset{Q}{x} * \left\{\begin{matrix}0 & \text{if } t < 0\\ \alpha^{(t)} & \text{otherwise}\end{matrix}\right\} \tag{6.7}$$

Computing $\overset{R}{x}$ as a convolution would of course be computationally wasteful - as it costs $\mathcal{O}(|\overset{Q}{x}|\min(|\overset{Q}{x}|, |\kappa^{pred}|))$ as opposed to the $\mathcal{O}(|\overset{Q}{x}|)$ of Equation 6.6, but the equivalence will be used in the following section to establish a link to STDP.

In Chapter 5, we presented a neural network where neurons communicate with predictive coding. A key observation here is that since the prediction is linear, computing the

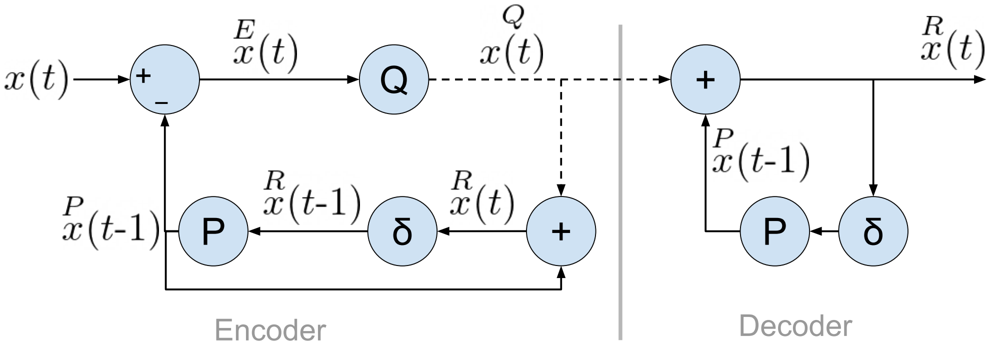


Figure 6.3.: An illustration of the predictive coding circuit. P indicates the prediction $P(y(t)) = \alpha y(t)$. δ indicates a time-delay $\delta(y(t)) = y(t-1)$, Q indicates the quantization function.

sum of weighted decoded input spikes is equivalent to computing the *decoded sum of weighted input spikes* - i.e. we only need one decoder per neuron, not one per synapse, allowing for efficient implementation. Equation 6.6 gives us update rule for neuron j: $z_j(t) := \alpha z_j(t-1) + \sum_i w_{ij} \overset{Q}{h}_i(t)$, where $h_i := h(z_i)$ denotes the post-nonlinearity neural activation.

The first two rows of Figure 6.5 show the quantization and reconstruction of example signals using predictive coding.

6.4. Dynamical Learning and Predictive Coding result in STDP

In this section, we will demonstrate that applying Dynamical learning to neurons which communicate with predictive coding yields the classic STDP kernel of Figure 6.1.

6.4.1. Properties of Convolution

The following sections will make use of some properties of convolution ($*$). We will introduce them here. All variables in these equations are time varying signals.

1. **Commutative Property**: $a * b = b * a$
2. **Associative Property**: $(a * b) * c = a * (b * c)$
3. **Linear Property**: $a * (b + c) = a * b + a * c$
4. **Scalar-Product Kernel-Flip**: $a \cdot (b * c) = (a * c^{-}) \cdot b$, Where $d \cdot e := \sum_t d_t e_t$ denotes the scalar-product over time-steps, and $c_t^{-} := c_{-t}$ denotes a flip about $t = 0$.

5. **Online Trick**: This is actually a result of combining the **Linear Property** and the **Scalar Product Kernel-Flip**: $(a * b) \cdot c = (a * b^{right}) \cdot c + (c * b^{-left}) \cdot a$, where $b_t^{-left} := \begin{cases} 0 & \text{if } t \leq 0 \\ b_{-t} & \text{otherwise} \end{cases}$, $b_t^{right} := \begin{cases} 0 & \text{if } t < 0 \\ b_t & \text{otherwise} \end{cases}$ indicate the flipped-left and right sides of the signal b (note that $b = b^{left} + b^{right}$). This is useful when you want to compute $(a * b) \cdot c$ in an online setting - where the signals come in as a temporal stream and you want to efficiently update $(a * b) \cdot c$ at each new time-step. We can use the Online Trick to convert STDP from the *Kernel Form* in Equation 6.3 to an *Online Form*:

$$\begin{aligned} \Delta w_{ij}^{STDP} &:= (\overset{Q}{h}_i * \kappa^{STDP}) \cdot \overset{Q}{h}_j \\ &= (\overset{Q}{h}_i * \kappa^{STDP,right}) \cdot \overset{Q}{h}_j + (\overset{Q}{h}_j * \kappa^{STDP,-left}) \cdot \overset{Q}{h}_i \quad \text{Online Trick} \end{aligned} \tag{6.8}$$

6.4.2. Approximate Gradient Descent yields a Symmetric STDP Kernel

In Chapter 5, we encoded and quantized both our forward-pass activations h and gradients g as bitstreams $\overset{Q}{h}$ and $\overset{Q}{g}$ for communication between layers. We then showed that applying the gradient-based weight update (Equation 6.1) to the reconstructions $\overset{R}{h}$ and $\overset{R}{g}$ from these bitstreams yields a symmetric form of STDP.

$$\begin{aligned} \Delta w_{ij}^{ApproxGD} \propto -\sum_t \overset{R}{h}_i(t) \overset{R}{g}_j(t) = -\overset{R}{h}_i \cdot \overset{R}{g}_j &= -(\overset{Q}{h}_i * \kappa^{Pred}) \cdot (\overset{Q}{g}_j * \kappa^{Pred}) && \text{Equations 6.1 \& 6.7} \\ &= -(\overset{Q}{h}_i * \kappa^{Pred} * \kappa^{-Pred}) \cdot \overset{Q}{g}_j && \text{Scalar Product Kernel Flip} \\ &:= -(\overset{Q}{h}_i * \kappa^{SymSTDP}) \cdot \overset{Q}{g}_j && \text{Kernel Form} \\ &= -(\overset{Q}{h}_i * \kappa^{SymSTDP,right}) \cdot \overset{Q}{g}_k - (\overset{Q}{g}_j * \kappa^{SymSTDP,-left}) \cdot \overset{Q}{h}_i && \text{Online Form} \end{aligned} \tag{6.9}$$

We can find the form of the kernel by plugging in the formula for κ^{Pred} from subsection 6.3.4

Predictive Coding:

$$\begin{aligned}\kappa_t^{SymSTDP} &: = (\kappa^{Pred} * \kappa^{-Pred})_t := \sum_\tau \begin{Bmatrix} 0 & \text{if } t-\tau<0 \\ \alpha^{(t-\tau)} & \text{otherwise} \end{Bmatrix} \begin{Bmatrix} \alpha^{(-\tau)} & \text{if } \tau \leq 0 \\ 0 & \text{otherwise} \end{Bmatrix} \\ &= \begin{cases} \alpha^{(-t)}/(1-\alpha^2) & \text{if } t<0 \\ \alpha^{(t)}/(1-\alpha^2) & \text{otherwise} \end{cases}\end{aligned} \tag{6.10}$$

Thus we showed that our approximate form of gradient descent was equivalent to an STDP-like learning rule (except with a symmetric kernel and potentially signed spikes). $\kappa^{SymSTDP}$ is plotted in the second row of Figure 6.4.

6.4.3. The Link to Dynamics-Based Learning

Finally, we are ready to demonstrate the connection to dynamics-based learning. Suppose we, following the ideas presented in subsection 6.3.1 Dynamics-Based Learning, simply replace the reconstructed gradient $\overset{R}{g}_j$ with a discrete-time approximation of the dynamics $\dot{h}_j(t) := \frac{\partial}{\partial t} h(z_j)(t) \approx \frac{1}{2}(\overset{R}{h}_j(t+1) - \overset{R}{h}_j(t-1)) := \overset{\Delta}{h}_j(t)$. Note that this corresponds to a convolution with a kernel $\kappa_t^{\Delta} := (\frac{1}{2} \text{ if } t=-1; -\frac{1}{2} \text{ if } t=1; 0 \text{ otherwise})$. Let us redo the calculation in Equation 6.9 with these approximate dynamics plugged in in place of the gradient:

$$\begin{aligned}\Delta w_{ij} &\propto \sum_t \overset{R}{h}_i(t) \overset{\Delta}{h}_j(t) := \overset{R}{h}_i \cdot \overset{\Delta}{h}_j := \overset{R}{h}_i \cdot (\overset{R}{h}_j * \kappa^{\Delta}) \\ &= (\overset{Q}{h}_i * \kappa^{Pred}) \cdot (\overset{Q}{h}_j * \kappa^{Pred} * \kappa^{\Delta}) \\ &= (\overset{Q}{h}_i * \kappa^{Pred} * \kappa^{-Pred} * \kappa^{-\Delta}) \cdot \overset{Q}{h}_j && \text{Scalar Product Kernel Flip} \\ &:= (\overset{Q}{h}_i * \kappa^{DynSTDP}) \cdot \overset{Q}{h}_j && \text{STDP Form}\end{aligned} \tag{6.11}$$

The above equation gives us the kernel $\kappa^{DynSTDP} := \kappa^{Pred} * \kappa^{-Pred} * \kappa^{-\Delta}$. Recall from Section 6.3.4 that $\kappa_t^{Pred} = (\alpha^{(t)} \text{ if } t \geq 0 \text{ otherwise } 0)$, and the definitions $\kappa_t^{-Pred} := \kappa_{-t}^{Pred}$ and $\kappa_t^{-\Delta} := (-\frac{1}{2} \text{ if } t=-1; \frac{1}{2} \text{ if } t=1; 0 \text{ otherwise})$. Convolving these three kernels, and using the result from Equation 6.10, yields:

$$\begin{aligned}
\kappa_t^{DynSTDP} &:= \left(\kappa^{Pred} * \kappa^{-Pred} * \kappa^{-\Delta}\right)_t := \left(\kappa^{SymSTDP} * \kappa^{-\Delta}\right)_t \\
&= \sum_\tau \begin{Bmatrix} \alpha^{-(t-\tau)}/(1-\alpha^2) & \text{if } t-\tau < 0 \\ \alpha^{(t-\tau)}/(1-\alpha^2) & \text{otherwise} \end{Bmatrix} \begin{Bmatrix} -1/2 & \text{if } \tau = -1 \\ 1/2 & \text{if } \tau = 1 \\ 0 & \text{otherwise} \end{Bmatrix} \\
&= \frac{1}{2\alpha} \begin{cases} -\alpha^{(-t)} & \text{if } t < 0 \\ 0 & \text{if } t = 0 \\ \alpha^{(t)} & \text{if } t > 0 \end{cases}
\end{aligned} \tag{6.12}$$

Note that this takes the same form as the original STDP kernel in Equation 6.3, with $a_i = a_+ = \frac{1}{2\alpha}$ and $\tau_- = \tau_+ = \frac{1}{\log \alpha^{-1}} \in (0, \infty)$. The fourth row of Figure 6.4 plots $\kappa^{DynSTDP}$ and the fourth row of figure Figure 6.5 shows the resulting weight updates on simulated data, as compared to unquantized dynamics-based learning.

6.5. Discussion

What we have here is an exact explanation for the anti-symmetrical double-exponential curve of STDP. It implements an approximation to dynamics-based learning. The approximation arises from the fact that the synapse only has access to the times of the pre- and post- synaptic spikes generated by predictive coding of their activations.

Biologically, it is clear that the only way a synapse ij can access the state of neuron i is via its spikes. As discussed in Chapter 1, axons are very long compared to the cell-bodies of neurons, and the primary purpose of the spiking mechanism is probably to enable long-distance signal transmission in a setting where analog voltages cannot be easily transmitted. Dendrites, which connect synapse ij to postsynaptic neuron j, tend to be much shorter than axons, but still may be long enough to have very non-uniform voltage. Thus synapse ij generally does not have direct access to the voltage in the body of cell j either. How then could the synapse even know when postsynaptic neuron j spikes? The answer appears to be a phenomenon known as *Neural Backpropagation* - wherein when neuron j produces a spike, an "echo" of this spike is also transmitted back throught the dendrites to the input synapses [Stuart and Sakmann, 1994]. This has long been believed to be the mechanism by which STDP occurs [Waters and Helmchen, 2004]. Unlike the Backpropagation of machine learning - neural backpropagation does not hop back over synapses, but stays within a cell.

Thus we show here how three reasonable hypotheses about how biological neurons communicate and learn neurons are actually connected: (1) Neurons perform dynamics-based learning, (2) Synapses update their weights as a function of pre- and post- synaptic spike

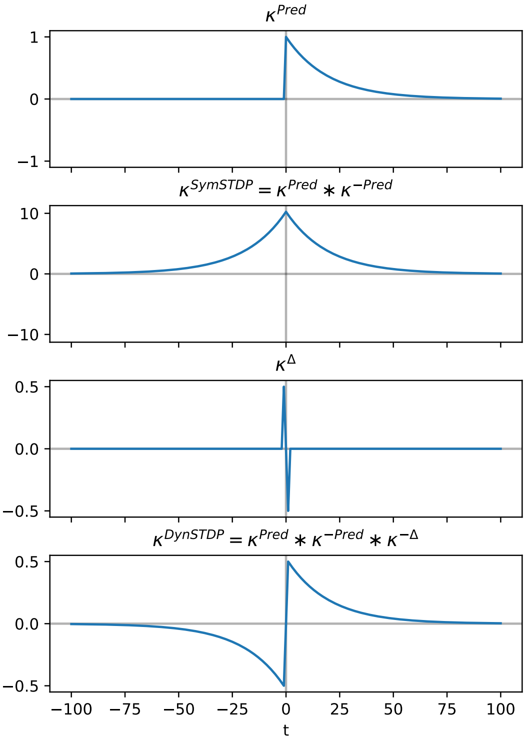


Figure 6.4.: The kernels discussed in this chapter. **Top:** The Predictive-Decoding kernel reconstructs a signal from it's quantized representation: $\overset{R}{x} = \overset{Q}{x} * \kappa^{Pred}$. **Second Row:** The Symmetric STDP kernel implements approximate gradient descent between two predictive-coded neurons. **Third Row:** The Δ-Kernel, when convolved with a signal, yields a discrete-time approximation to the temporal-derivative: $\dot{x}(t) \approx (x * \kappa^{\Delta})_t$. **Bottom Row:** The STDP kernel is can be obtained from the Predictive-Decoding Kernel and the Δ-Kernel.

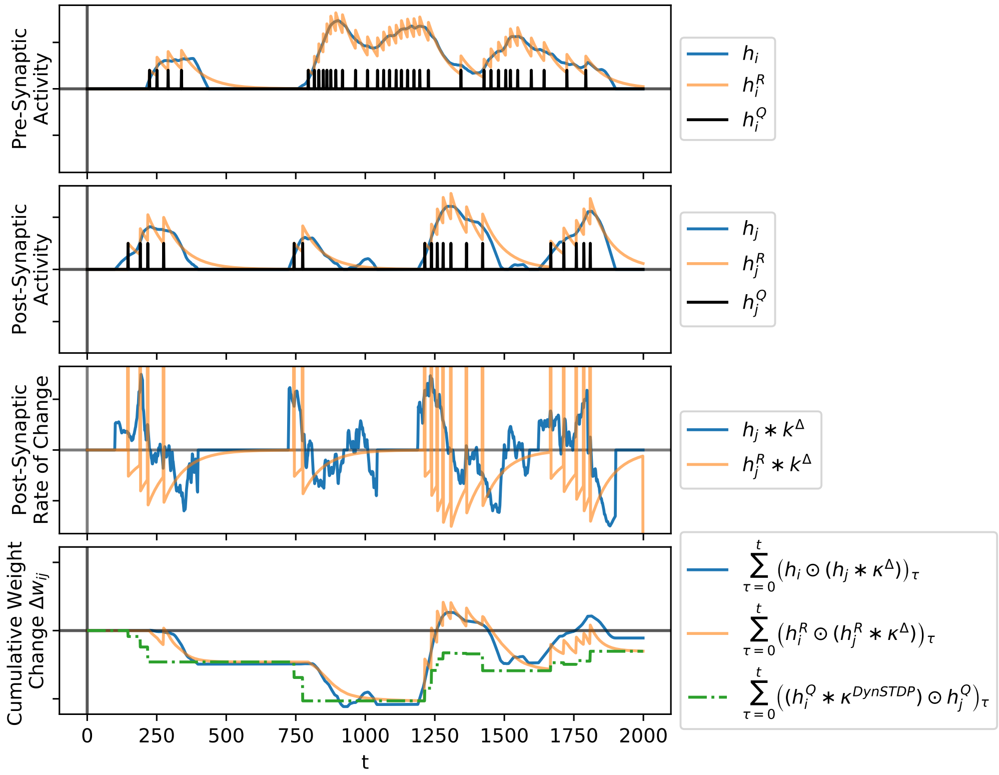


Figure 6.5.: Dynamics of the weight change Δw_{ij} in response to randomly-generated signals from presynaptic neuron i and postsynaptic neuron j. **Top:** Activation $\boldsymbol{h_i}$ of the presynaptic neuron, alongside the quantized (spiking) representation $\overset{Q}{\boldsymbol{h_i}}$, and the reconstruction of the activation from this representation $\overset{R}{\boldsymbol{h_i}}$. **Second Row:** The same for postsynaptic neuron j. **Third Row:** The rate of change of the postsynaptic activity $\boldsymbol{h_j * \kappa^{\Delta}} \approx \dot{\boldsymbol{h_j}}$, and the rate of change of *reconstructed* post-synaptic activity: $\overset{R}{\boldsymbol{h_j}} \boldsymbol{* \kappa^{\Delta}}$. **Bottom:** The cumulative weight change $\Delta w_{ij}(t)$ resulting from dynamics-based updates of Equation 6.2, the approximate dynamics-based updates from the first line of Equation 6.11, based on the reconstructions from the pre- and post-synaptic spikes, and the STDP-updates from Equation 6.4, using the kernel in Equation 6.12. Note that the approximate dynamics-based updates and the STDP-updates arrive at the exact same value of Δw_{ij} in the end. This is not a coincidence - it is a consequence of the equivalence expressed in Equation 6.11. Note also that despite the approximation arising from quantizing and decoding the signal, the approximate-dynamics weight-trajectory closely matches the one obtained from unquantized dynamics-based learning.

times, according to STDP. (3) Neurons use predictive coding to efficiently communicate their signals. We show that (3) shows how (2) is actually a result of (1).

7. Initialized Equilibrium Propagation

In Chapter 3 and ref 5, we devised schemes for training feedforward spiking neural networks. Both of these schemes were basically quantized versions of backpropagation. Since one basic property of biological networks is that they do not use backprop (Gap 1: No Backprop) we began looking into ideas for how one could train a feedforward network in a more biologically plausible way.

In Section 2.7, we introduced Equilibrium Propagation [Scellier and Bengio, 2017] - an algorithm for training an energy-based network to minimize a supervised objective function. Unlike backprop and some other non-backprop training schemes, such as Difference Target Propagation [Lee et al., 2015] and Feedback Alignment Lillicrap et al. [2014], the neurons used in Equilibrium Propagation produce only an activation, and do not need to produce a secondary error signal to learn. Neurons are considered dynamical systems whose rate of change is a function of the state and the inputs. Without a need for neurons to hold their state until an error signal returns, Equilibrium Prop is a step towards training asynchronous neural networks, and resolving Gap 4: Asynchronous Processing. What's missing is that it does not work on dynamic *inputs* - the algorithm assumes that inputs are held fixed while the network settles to a steady state.

Instead of backpropagatating a loss-gradient, Equilbrium Propagation computes the gradient by first presenting the input data x, letting the network settle to an energy-minimizing state s^-, then presenting the target y, which slightly perturbs output units in the network so that they come closer to the target. The perturbation of the target neurons also affects neurons in the rest of the network, leading neurons to settle at a new energy-minimizing state s^+. This is illustrated in the right two panes of Figure 7.1.

One impracticality with Equilibrium Propagation is that inference requires a settling process. The network is defined by the dynamics: $\frac{\partial s}{\partial t} = -\frac{\partial E(s,\theta,x)}{\partial s}$, where $E(s, \theta, x)$ is some energy function over the state of the network s, the synaptic weights θ, and inputs x, and inference is done by presenting an input x and letting the network settle to an energy minimum s^-. The values of the output units $s^-_{\mathcal{O}}$ are then taken to be the prediction of the network. The time it takes to settle scales poorly with the depth of the network, making Equilibrium Propagation impractically slow.

Our proposed modification is to initialize the settling process with a feedforward network. This is illustrated in the left pane of Figure 7.1. A feedforward network aims to approximate s^- with as a function $s^f = f_\phi(x)$. s^f can then be used to initialize the settling process. The parameters ϕ of the feedforward network are trained to give a better prediction of

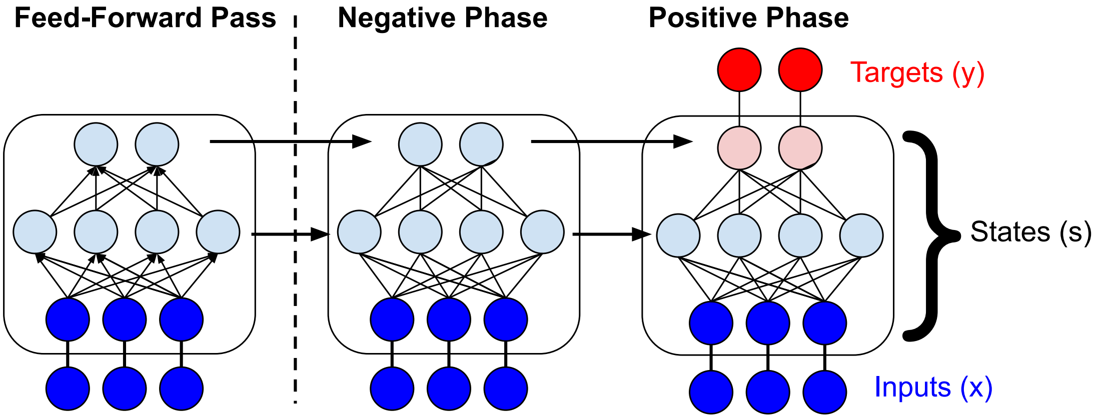


Figure 7.1.: The **middle** and **right** panels illustrate Equilibrium propagation: The network first settles to a minimum conditioned on an input x, arriving at a state s^-. Then, the network is perturbed by the target, which pushes it towards a new state s^+. The parameters are then updated to bring s^- closer to s^+, so that the network better predicts the target the next time around. The **left** pane illustrates our modification. We initialized the settling state with a feedforward network. This network learns to approximate the settling of s^- in one forward pass. After training, the feedforward initializing network can be used directly for inference.

the energy-minimizing state. Specifically, the parameters ϕ_i of each feedforward neuron i learn to minimize a local objective $\|s_i^f - s_i^-\|$. This feedforward network need not just be used for initialization. Since it is being trained to approximate the energy-minimizing state, and the energy-minimizing state is the prediction of the network, we can simply use our feedforward net as the inference network in the end. Thus we end up with a backprop-free, approximately gradient-based way to train a feedforward network.

7.1. Abstract

Deep neural networks are almost universally trained with reverse-mode automatic differentiation (a.k.a. backpropagation). Biological networks, on the other hand, appear to lack any mechanism for sending gradients back to their input neurons, and thus cannot be learning in this way. In response to this, Scellier and Bengio [2017] proposed *Equilibrium Propagation* - a method for gradient-based training of neural networks which uses only local learning rules and, crucially, does not rely on neurons having a mechanism for back-propagating an error gradient. Equilibrium propagation, however, has a major practical limitation: inference involves doing an iterative optimization of neural activations to find a fixed-point, and the number of steps required to closely approximate this fixed point scales poorly with the depth of the network. In response to this problem, we propose

Initialized Equilibrium Propagation, which trains a feedforward network to initialize the iterative inference procedure for Equilibrium propagation. This feed-forward network learns to approximate the state of the fixed-point using a local learning rule. After training, we can simply use this initializing network for inference, resulting in a learned feedforward network. Our experiments show that this network appears to work as well or better than the original version of Equilibrium propagation while requiring fewer steps to converge. This shows how we might go about training deep networks without using backpropagation.

7.2. Introduction

Deep neural networks are almost always trained with gradient descent, and gradients are almost always computed with backpropagation. For those interested in understanding the working of the brain in the context of machine learning, it is therefore distressing that biological neurons appear not to send signals backwards.

Biological neurons communicate by sending a sequence of pulses to downstream neurons along a one-way signaling pathway called an "axon". If neurons were doing backpropagation, one would expect a secondary signalling pathway wherein gradient signals travel backwards along axons. This appears not to exist, so it seems that biological neurons cannot be doing backpropagation.

Moreover, backpropagation may not be the ideal learning algorithm for efficient implementation in hardware, because it involves buffering activations for each layer until an error gradient returns. This requirement becomes especially onerous when we wish to backpropagate through many steps of time, or through many layers of depth. For these reasons, researchers are looking into other means of neural *credit assignment* - mechanisms for generating useful learning signals without doing backpropagation.

Recently, Scellier and Bengio [2017] proposed a novel algorithm called *Equilibrium Propagation*, which enables the computation of parameter gradients in a deep neural network without backpropagation. Equilibrium Propagation defines a neural network as a dynamical system, whose dynamics follow the negative-gradient of an energy function. The "prediction" of this network is the fixed-point of the dynamics - the point at which the system settles to a local minimum energy given the input, and ceases to change. Because of this inference scheme, Equilibrium Propagation is impractically slow for large networks - the network has to iteratively converge to a fixed point at every training iteration.

In this work, we take inspiration from Hinton et al. [2015] and distill knowledge from a slow, energy based *equilibrating network* into a fast *feedforward network* by training the feedforward network to predict the fixed-points of the equilibrating network with a local loss. At the end of training, we can then discard the equilibrating network

[0] Code available at https://github.com/QUVA-Lab/init-eqprop

and simply use our feedforward network for test-time inference. We thus have a way to train a feedforward network without backpropagation. The resulting architecture loosely resembles a Conditional Generative Adversarial Network [Mirza and Osindero, 2014], where the feedforward network produces a network state which is evaluated by the energy-based equilibrating network.

To aid the reader, this paper contains a glossary of symbols in Appendix D.1.

7.3. Methods

7.3.1. Background: Equilibrium Propagation

Equilibrium propagation is an algorithm for training an energy-based network on a supervised task. We refer the reader to Section 2.7 for the details of how this algorithm works.

Because training involves many rounds of inference, and each round of inference involves an iterative settling process, training a network with Equilibrium Propagation can be impractically slow. In their experiments, Scellier and Bengio [2017] indicate that the number of settling steps required scales super-linearly with the number of layers. This points to an obvious need for a fast inference scheme.

7.3.2. Adding an Initialization Network

We propose training a feedforward network $f_\phi(x) \rightarrow s^f \in \mathbb{R}^{|\mathcal{S}|}$ to predict the fixed-point of the equilibrating network. This allows the feedforward network to achieve two things: First, it initializes the state of the equilibrating network, so that the settling process starts in the right regime. Second, the feedforward network can be used to perform inference at test-time, since it learns to approximate the minimal-energy state of the equilibrating network, which corresponds to the prediction. $f_\phi(x)$ is defined as follows:

$$\begin{aligned} f_\phi(x) &:= (s_j^f : j \in \mathcal{S}) && \in \mathbb{R}^{|\mathcal{S}|} \\ s_j^f &:= \rho\left(\left(\sum_{i \in \alpha_j^f \cap \mathcal{S}} \omega_{ij} s_i^f\right) + \left(\sum_{i \in \alpha_j^f \cap \mathcal{I}} \omega_{ij} x_i\right) + c_j\right) && \in \mathbb{R} \end{aligned} \tag{7.1}$$

Where $\alpha_j^f = (i : (i \in \alpha_j) \wedge (i < j))$ is the set of *feedforward* connections to neuron j (which is a subset of α_j - the full set of connections to neuron j from the equilibrium network from Equation 2.9); $\phi = (\omega, c)$ is the set of parameters of the feedforward network. This feedforward network produces the initial state of the negative phase of equilibrium propagation network, given the input data - i.e., instead of starting at a zero-state, the equilibrium-propagation network is initialized in a state $s^f := f_\phi(x)$. We train the parameters ϕ to approximate the minimal energy state s^- of the equilibrating network [1]. In other words, we seek:

$$\phi^* := \arg\min_\phi \mathcal{L}(s^f, s^-) \tag{7.2}$$

$$\mathcal{L}(s^f, s^-) := \sum_{i \in \mathcal{S}} \mathcal{L}_i(s_i^f, s_i^-) := \sum_{i \in \mathcal{S}} (s_i^f - s_i^-)^2 \tag{7.3}$$

The derivative of the forward parameters of the i'th neuron, $\phi_i = (\omega_{\alpha_i,i}, c_i)$, can be expanded as:

$$\frac{\partial \mathcal{L}}{\partial \phi_i} := \sum_{j \in \mathcal{S}} \frac{\partial \mathcal{L}_j(s_j^f, s_j^-)}{\partial \phi_i} = \overbrace{\frac{\partial \mathcal{L}_i}{\partial s_i^f} \frac{\partial s_i^f}{\partial \phi_i}}^{\text{local}} + \overbrace{\sum_{j>i} \frac{\partial \mathcal{L}_j}{\partial s_j^f} \frac{\partial s_j^f}{\partial s_i^f} \frac{\partial s_i^f}{\partial \phi_i}}^{\text{distant}} \tag{7.4}$$

The *distant* term is problematic, because computing $\frac{\partial s_j^f}{\partial s_i^f}$ would require backpropagation, and the entire purpose of this exercise is to train a neural network without backpropagation. However, we find that only optimizing the local term $\frac{\partial \mathcal{L}_i}{\partial \phi_i}$ does not noticeably harm performance. In Section 7.3.4 we go into more detail on why it appears to be sufficient to minimize local losses.

Over the course of training, parameters ϕ will learn until our feedforward network is a good predictor of the minimal-energy state of the equilibrating network. This feedforward network can then be used to do inference: we simply take the state of the output neurons to be our prediction of the target data. The full training procedure is outlined in Algorithm 6. At the end of training, inference can be done either by taking the output activations from the forward pass of the inference network f_ϕ (Algorithm 7), or by initializing with a forward pass and then iteratively minimizing the energy (Algorithm 8). Experiments in Section 7.4 indicate that the forward pass performs just as well as the full energy minimization.

[1] We could also minimize the distance with s^+, but found experimentally that this actually works slightly worse than s^-. We believe that this is because equilibrium propagation depends on s^- being very close to a true minimum of the energy function, and so initializing the negative phase to $s^f \approx s^-$ will lead to better gradient computations than when we initialize the negative phase to $s^f \approx x^+$

Algorithm 6 Training

```
Input: Dataset $(x, y)$, Step Size $\epsilon$, Learning Rate $\eta$, Network Architecture $\alpha$, Number of negative-phase steps $T^-$, Number of positive-phase steps $T^+$
$\phi \leftarrow$ InitializeFeedforwardParameters$(\alpha)$
$\theta \leftarrow$ InitializeEquilibriumParameters$(\alpha)$
while not converged do
    $x_m, y_m \rightarrow$ SampleMinibatch$(x, y)$
    $s \leftarrow s^f \leftarrow f_\phi(x_m)$
    for $t \in 1..T^-$ do # Neg. Phase
        $s \leftarrow s - \epsilon \frac{\partial E_\theta(s, x_m)}{\partial s}$
    end for
    $s^- \leftarrow s$
    for $t \in 1..T^+$ do # Pos. Phase
        $s \leftarrow s - \epsilon \frac{\partial E_\theta^\beta(s, x_m, y_m)}{\partial s}$
    end for
    $s^+ \leftarrow s$
    $\theta \leftarrow \theta - \frac{\eta}{\beta}\left(\frac{\partial E_\theta(s^+, x)}{\partial \theta} - \frac{\partial E_\theta(s^-, x)}{\partial \theta}\right)$
    $\phi_i \leftarrow \phi_i - \eta \frac{\partial \mathcal{L}_i(s_i^f, s_i^-)}{\partial \phi_i} \forall i$
end while
Return: $\phi, \theta$ # Parameters
```

Algorithm 7 Feedforward Inference

```
Input: Input Data $x$, Inference Parameters $\phi$
$s \leftarrow f_\phi(x)$
return $(s_i : i \in \mathcal{O})$ # Output unit states
```

Algorithm 8 Iterative Inference

```
Input: Input Data $x$, Initialization Parameters $\phi$, Equilibriating Parameters $\theta$, Number of Negative Steps $T^-$
$s \leftarrow f_\phi(x)$
for $t \in 1..T^-$ do # Neg. Phase
    $s \leftarrow s - \epsilon \frac{\partial E_\theta(s, x_m)}{\partial s}$
end for
return $(s_i : i \in \mathcal{O})$ # Output unit states
```

7.3.3. Including the forward states in the energy function

The fixed point s^- of the equilibrating network is a nonlinear function of x, whose value is computed by iterative bottom-up and top-down inference using all of the parameters θ. The initial state s^f, by contrast, is generated in a single forward pass, meaning that the function relating s_j^f to its direct inputs $s_{\alpha_j}^f \in \mathbb{R}^{|\alpha_j|}$ is constrained to the form of Equation 7.1. Because of this, the computation resulting in s^- may be more *flexible* than that of the forward pass, so it is possible for the equilibrating network to create targets that are not *achievable* by the neurons in the feedforward network. This is similar to the notion of an "amortization gap" in variational inference, and we discuss this connection more in Section 7.5.2.

Neurons in the feedforward network simply learn a linear mapping from the previous layer's activations to the targets provided by the equilibrating network. In order to encourage the equilibrating network to stay in the regime that is reachable by the forward network, we can add a loss encouraging the fixed points to stay in the regime of the forward pass.

$$E_\theta^\lambda(s, x) = E_\theta(s, x) + \lambda \sum_{j \in \mathcal{S}} (s_j^f - s_j)^2 \tag{7.5}$$

Where λ is a hyperparameter which brings the fixed-points of the equilibrating network closer to the states of the forward pass, and encourages the network to optimize the energy landscape in the region reachable by the forward network. Of course this may reduce the effective capacity of the equilibrating network, but if our goal is only to train the feedforward network, this does not matter. This trick has a secondary benefit: It allows faster convergence in the negative phase by pulling the minimum of $E_\theta^\lambda(s, x)$ closer to the feedforward prediction, so we can learn with fewer convergence steps. It can however, cause instabilities when set too high. We investigate the effect of different values of λ with experiments in Appendix D.4.

7.3.4. Why the local loss is sufficient: Gradient Alignment

In Equation 7.4 we decompose the loss-gradient of parameters ϕ into a local and a global component. Empirically (see Figures 7.2, 7.4), we find that using the local loss and simply ignoring the global loss led to equally good convergence. To understand why this is the case, let use consider a problem where we learn the mapping from an input x to a set of neuron-wise targets: s^*. Assume these targets are generated by some (unknown) set of ideal parameters ϕ^*, so that $s^* = f_{\phi^*}(x)$. To illustrate, we consider a two layer network with $\phi = (w_1, w_2)$ and $\phi^* = (w_1^*, w_2^*)$:

$$\begin{array}{lll} s_1 = \rho(x w_1) & s_1^* = \rho(x w_1^*) & \mathcal{L}_1 = \|s_1 - s_1^*\|_2^2 \\ s_2 = \rho(s_1 w_2) & s_2^* = \rho(s_1^* w_2^*) & \mathcal{L}_2 = \|s_2 - s_2^*\|_2^2 \end{array} \tag{7.6}$$

It may come as a surprise that when ϕ is in the neighbourhood of the ideal parameters ϕ^*, the cosine similarity between the *local* and *distant* gradients: $S\left(\frac{\partial \mathcal{L}_1}{\partial w_1}, \frac{\partial \mathcal{L}_2}{\partial w_1}\right)$ is almost always positive, i.e. the local and distant gradients tend to be aligned. This is a pleasant surprise because it means the local loss will tend to guide us in the right direction. The reason becomes apparent when we define $\Delta w := w - w^*$, and write out the expression for

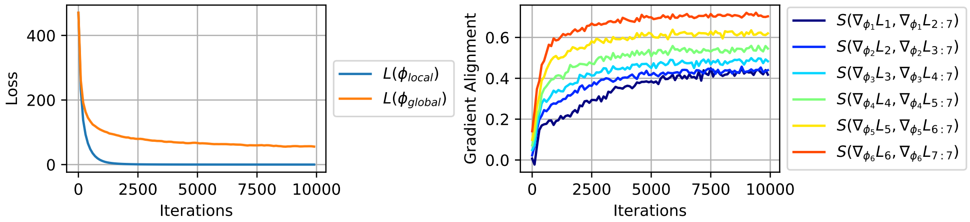


Figure 7.2.: We train a 6-layer network with parameters ϕ to predict layerwise targets generated by another network with random parameters ϕ^*. **Left:** We compare the convergence of the global loss of two training runs starting from the same initial conditions and identical (untuned) hyperparamters: A network with parameters ϕ_{local} trained using only local losses and a network with parameters ϕ_{global} trained directly on the global loss. We note that the *locally trained* network converges significantly faster, suggesting that optimization is easier in the absence of the "confusing" distant-gradient signals from the not-yet-converged higher layers. **Right:** We plot the cosine-similarity of local and distant components of the gradient of ϕ_{local} as training progresses. We see that as we approach convergence (as $\phi_{local} \to \phi^*$), the local and distant gradients tend to align.

the gradient in the limit of $\Delta w \to 0$ (see Appendix D.2 for derivation)

$$\begin{aligned}
\frac{\partial \mathcal{L}_1}{\partial w_1} &\underset{\Delta w \to 0}{=} x^T(x\Delta w_1 \odot \rho'(xw_1) \odot \rho'(xw_1)) \\
\frac{\partial \mathcal{L}_2}{\partial w_1} &\underset{\Delta w \to 0}{=} \overbrace{x^T\Big(x\Delta w_1 \odot \rho'(xw_1) w_2 \odot \rho'(s_1 w_2)^2 w_2^T \odot \rho'(xw_1)\Big)}^{G_1} \\
&\quad \overbrace{x^T\Big(s_1 \Delta w_2 \odot \rho'(s_1 w_2)^2 w_2^T \odot \rho'(xw_1)\Big)}^{G_2}
\end{aligned} \tag{7.7}$$

When the term $w_2 \odot \rho'(s_1 \cdot w_2)^2 \cdot w_2^T$ is proportional to an identity matrix, we can see that $\frac{\partial \mathcal{L}_1}{\partial w_1}$ and G_1 are perfectly aligned. This will be the case when w_2 is orthogonal and layer 2 has a linear activation. However, even for randomly sampled parameters and a nonlinear activation, $w_2 \odot \rho'(s_1 \cdot w_2)^2 \cdot w_2^T$ tends to have a strong diagonal component and the terms thus tend to be positively aligned. Figure 7.2 demonstrates that this gradient-alignment tends to *increase* as then network trains to approximate a set of targets (i.e. as $\phi \to \phi^*$). Note that the alignment of the *local* loss-gradient with the *global* loss-gradient is at least as high as with the *distant* loss-gradient, because $\nabla_\phi \mathcal{L}_{global} = \nabla_\phi \mathcal{L}_{local} + \nabla_\phi \mathcal{L}_{distant}$ and $\mathcal{S}(\nabla_\phi \mathcal{L}_{distant}, \nabla_\phi \mathcal{L}_{local}) \leq S(\nabla_\phi \mathcal{L}_{distant} + \nabla_\phi \mathcal{L}_{local}, \nabla_\phi \mathcal{L}_{local}) \quad \forall \nabla_\phi \mathcal{L}_{local}, \nabla_\phi \mathcal{L}_{distant}$.

This explains the empirical observation in Figures 7.2 and 7.4 that optimizing the local, as opposed to the global, loss for the feedforward network does not appear to slow down convergence: Later layers do not have to "wait" for earlier layers to converge before they themselves converge - earlier layers optimize the loss of later layers right from the beginning of training. As shown in Figure 7.2, it may in fact speed up convergence since each layer's optimizer is solving a simpler problem (albeit with changing input representations for layers > 1).

When local targets s^- are provided by the equilibrating network, it is not in general true that there exists some ϕ^* such that $s^- = s^*$. In our experiments, we observed that this did not prevent the forward network from learning to classify just as well as the equilibrating network. However, this may not hold for more complex datasets. As mentioned in Section 7.3.3, this could be resolved in future work with a scheme for annealing λ up to infinity while maintaining stable training.

7.4. Experiments

We base our experiments off of those of Scellier and Bengio [2017]: We use the hard sigmoid $\rho(x) = max(0, min(1, x))$ as our nonlinearity. We clip the state of s_i to the range (0, 1) because, since $\rho'(x) = 0 : x < 0 \vee x > 1$, if the system in Equation 2.10 were run in continuous time, it should never reach states outside this range. Borrowing a trick from Scellier and Bengio [2017] to avoid instability problems arising from incomplete negative-phase convergence, we randomly sample $\beta \sim \mathcal{U}(\{-\beta_{base}, +\beta_{base}\})$, where β_{base} is a small positive number, for each minibatch and use this for both the positive phase and for multiplying the learning rate in Equation 2.11 (for simplicity, this is not shown in Algorithm 6). [2] . Unlike Scellier and Bengio [2017], we do not use the trick of caching and reusing converged states for each data point between epochs. In order to avoid "dead gradient" zones, we modify the activation function of our feedforward network (described in Equation 7.1) to $\rho^{mod}(x) = \rho(x) + 0.01x$, where the 0.01 "leak" is added to prevent the feed-forward neurons from getting stuck due to zero-gradients in the saturated regions. We use $\lambda = 0.1$ as the regularizing parameter from Equation 7.5, having scanned for values in Appendix D.4.

7.4.1. MNIST

We verify that the our algorithm works on the MNIST dataset. The learning curves can be seen in Figure 7.3. We find, somewhat surprisingly, that the forward pass of our network performs almost indistinguishably from the performance of the negative-phase of Equilibrium Propagation. This encouraging result shows that this approach for training a feedforward network without backprop does indeed work. We also see from from the

[2] When β is negative, the positive-state s^+ is pushed *away* from the targets, but gradients still point in the correct direction because the learning rate is scaled by $-1/\beta$. This trick avoids an instability when, due to incomplete negative-phase convergence, the network continues approaching the true minimum of $E(s, x)$ in the positive phase, and thus on every iteration contues to push *down* the energy of this "true" negative minimum

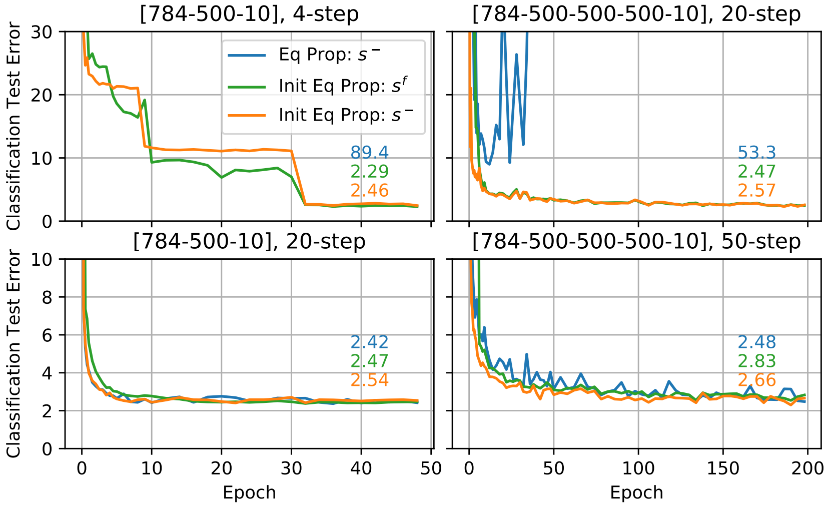


Figure 7.3.: Learning Curves on MNIST comparing the performance of Equilibrium Propagation (Eq Prop: s^-), the Forward-Pass in Initialized Equilibrium Propagation (Fwd Eq Prop: s^f) (Algorithm 7) and the Negative Phase in Initialized Equilibrium Propagation (Fwd Eq Prop: s^-) (Algorithm 8) Numbers indicate error at the final test. **Left Column:** A shallow network with a single hidden layer of 500 units. **Right Column:** A deeper network with 3 layers of [500, 500, 500] hidden units. **Top Row:** Training with a small-number of negative-phase steps (4 for the shallow network, 20 for the deeper) shows that feedfoward initialization makes training more stable by providing a good starting point for the negative phase optimization. The *Eq Prop* s^- lines on the upper plots are shortened because we terminate training when the network fails to converge. **Bottom Row:** Training with more negative-phase steps shows that when the baseline Equilibrium Propagation network is given sufficient time to converge, it performs comparably with our feedforward network (Note that the y-axis scale differs from the top).

top-two panels of Figure 7.3 that our approach can stabilize Equilibrium-Prop learning when we run the network for fewer steps than are needed for full convergence. By initializing the negative phase in a close-to-optimal regime, the network is able to learn when the number of steps is so low that plain Equilibrium Propagation cannot converge. Moreover we note that as the number of steps is enough for convergence, there is not much advantage to using more negative-phase iterations - the longer negative phase does not improve our error.

In Figure 7.4 we demonstrate that using only local losses to update the feedforward network comes with no apparent disadvantage. In line with our results from Section 7.3.4, we see that local loss gradients become aligned with the loss gradients from higher layers, explaining why it appears to be sufficient to only use the local gradients.

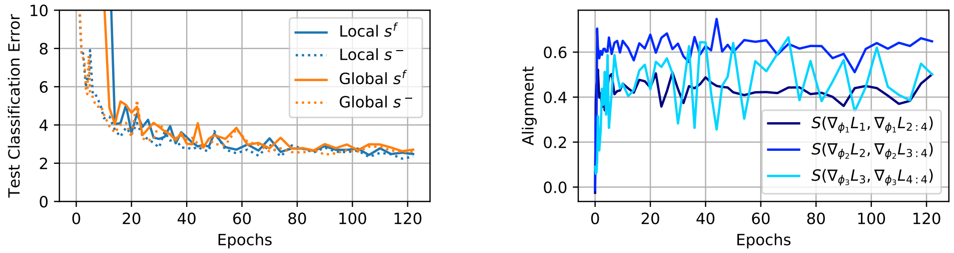


Figure 7.4.: Test scores and gradient alignment on [784-500-500-500-10] network trained on MNIST **Left:** We compare the performance of Initialized Equilibrium Propagation when the feedforward network is trained using only local losses vs the global loss (i.e. using backpropagation). s^f denotes the forward pass and s^- denotes the state at the end of the negative phase. Note that we observe no disadvantage when we only use local losses. **Right:** We observe the same effect as for our toy problem (see Figure 7.2). Early on in training, the local error gradients tend to align with gradients coming from higher layers.

7.5. Related Work

The most closely related work to ours is by Bengio et al. [2016]. There, the authors examine the idea of initializing an iterative settling process with a forward pass. They propose using the parameters of the Equilibriating network to do a forward pass, and describe the conditions under which this provides a good approximation of the energy-minimizing state. Their conclusion is that this criterion is met when consecutive layers of the energy-based model form a good autoencoder. Their model differs from ours in that the parameters of the forward model are tied to the parameters of the energy-based model. The effects of this assumption are unclear, and the authors do not demonstrate a training algorithm using this idea.

Our work was loosely inspired by Hinton et al. [2015], who proposed “distilling” the knowledge of a large neural network or ensemble into a smaller network which is designed to run efficiently at inference time. In this work, we distill knowledge from a slow, equilibrating network in to a fast feedforward network.

7.5.1. Relation to Adversarial Learning

Several authors [Kim and Bengio, 2016], [Finn et al., 2016], [Zhai et al., 2016] have pointed out the connection between Energy Based Models and Generative Adversarial Networks (GANs). In these works, a feedforward *generator* network proposes synthetic samples to be evaluated by an energy-based *discriminator*, which learns to push down the energy of real samples and push up the energy of synthetic ones. In these models, both the *generator/sample proposer* and the *discriminator/energy-based-model* are deep feedforward networks trained with backpropagation.

In our approach, we have a similar scenario. The inference network f_ϕ can be thought of as a conditional generator which produces a network state s^f given a randomly sampled input datum x: $s^f = f_\phi(x)$. Parameters ϕ are trained to approximate the minimal-energy states of the energy function: $\min_\phi \| f_\phi(x) - \arg\min_s E_\theta(s,x)\|$. However, in our model, the Energy-Based network $E_\theta(s,x)$ does not directly evaluate the energy of the generated data s^f, but of the minimizing state $s^- = \arg\min_s E_\theta(s,x)$ which is produced by performing T^- energy-minimization steps on s^f (see Algorithm 6). Like a discriminator, the energy-based model parameters θ learn based on a contrastive loss which pushes up the energy of the "synthetic" network state s^- while pushing down the energy of the "real" state s^+.

7.5.2. Relation to Amortized Variational Inference

In variational inference, we aim to estimate a posterior distribution $p(z|x)$ over a latent variable z given data x, using an *approximate posterior* $q(z)$. Algorithms such as Expectation Maximization [Dempster et al., 1977] iteratively update a set of posterior parameters μ per-data point, so that $z_n \sim q(z|\mu_n)$. In *amortized inference*, we instead learn a global set of parameters ϕ which can map a sample x_n to a posterior estimate $z_n \sim q_\phi(z|x_n)$. Dayan et al. [1995] proposed using a "recognition' network" as this amortized predictor, and Kingma and Welling [2013] showed that you can train this recognition network efficiently using the reparameterization trick. However, this comes at the expense of an "amortization gap" [Cremer et al., 2018] - where the posterior estimate suffers due to the sharing of posterior estimation parameters across data samples. Several recent works [Marino et al., 2018], [Li et al., 2017], [Kim et al., 2018], have proposed various versions of a "teacher-student" framework, in which an amortized network $q_\theta(z|x)$ provides an initial guess for the posterior, which is then refined by a slow, non-amortized network which refines $q(z)$ in several steps into a better posterior estimate. The "student" then learns to refine its posterior estimate using the final result of the iterative inference. In the context of training Deep Boltzmann Machines, Salakhutdinov and Larochelle [2010] trained a feedforward network with backpropagation to initialize variational parameters which are then optimized to estimate the posterior over latent variables.

Initialized Equilibrium Propagation is a zero-temperature analog of amortized variational inference. In the zero-temperature limit, the mean-field updates of variational inference reduce to coordinate ascent on the variational parameters. The function of the amortized student network $q_\phi(z|x)$ is then analogous to the function of our initializing network $f_\phi(x)$, and the negative phase corresponds to the iterative optimization of varational parameters from the starting point provided by $f_\phi(x)$.

7.5.3. Relation to other work in Local Credit-Assignment

Another interesting approach to shortening the inference phase in Equilibrium propagation was proposed by Kohan et al. [2018]. The authors propose a model that is *almost* a feedforward network, except that the output layer projects back to the input layer. The negative phase consists of making several feedforward passes through the network, reprojecting the output back to the input with each pass. Although the resulting inference model is not a feedforward network, the authors claim that this approach allows them to dramatically shorten convergence time of the negative phase.

There is also a notable similarity between Initialized Equilibrium Propagation and Method of Auxiliary Coordinates [Carreira-Perpinan and Wang, 2014]. In that paper, the authors propose a scheme for optimizing a layered feedforward network which consists of alternating optimization of the neural activations (which can be parallelized across samples) and parameters (which can be parallelized across layers). In order to ensure that the layer activations z_k remain close to what a feedforward network can compute, the objective includes a layerwise cost $\frac{\mu}{2}\|z_k - f_k(z_{k-1})\|^2$, where z_k is layer k's activation, f_k is layer k's function, and μ is a the strength of the layerwise cost (as they anneal $\mu \to \infty$ this cost becomes a constraint). This is identical in form and function to our $\lambda \sum_{j \in \mathcal{S}} (s^f - s^-)^2$ term in Equation 7.5. However, they differ from our method in that their neurons backpropagate their gradients back to input neurons (albeit only across one layer). Taylor et al. [2016] do something similar with using the Alternating Direction Method of Multipliers (ADMM), where Lagrange multipliers enforce the "layer matching" constraints exactly. Both methods, unlike Equilibrium Prop, are full-batch methods.

More broadly, other approaches to backprop-free credit assignment have been tried. Difference-Target propagation [Lee et al., 2015] proposes a mechanism to send back targets to each layer, such that locally optimizing targets also optimizes the true objective. Feedback-Alignment [Lillicrap et al., 2014] shows that, surprisingly, it is possible to train while using random weights for the backwards pass in backpropagation, because the forward pass parameters tends to "align" to the backwards-pass parameters so that the pseudogradients tend to be within 90° of the true gradients. A similar phenomenon was observed in Equilibrium Propagation by Scellier et al. [2018], who showed that when one removed the constraint of symmetric weight in Equilibrium propagation, the weights would evolve towards symmetry through training. Finally, Jaderberg et al. [2016] used a very different approach - rather than create local targets, each layer predicts its own "pseudogradient". The gradient prediction parameters are then trained either by the true gradients (which no longer need to arrive before a parameter update takes place) or by backpropagated versions of pseudogradients from higher layers.

7.6. Discussion

In this paper we describe how to use a recurrent, energy-based model to provide layerwise targets with which to train a feedforward network without backpropagation. This work helps us understand how the brain might be training fast inference networks. In this view, neurons in the inference network learn to predict local targets, which correspond to the minimal energy states, which are found by the iterative settling of a separate, recurrently connected *equilibrating network*.

More immediately perhaps, this could lead towards efficient analog neural network designs in hardware. As pointed out by Scellier and Bengio [2017], it is much easier to design an analog circuit to minimize some (possibly unknown) energy function than it is to design a feedforward circuit and a parallel backwards circuit which exactly computes its gradients. However it is very undesirable for the function of a network to depend on peculiarities of a particular piece of analog hardware, because then the network cannot be easily replicated. We could imagine using a hybrid circuit to train a digital, copy-able feedforward network, which is updated by gradients computed in the analog hardware. Without the constraint of having to backpropagate through the feedforward network, designs could be simplified, for example to do away with the need for differentiable activation functions or to use feedforward architectures which would otherwise suffer from vanishing/exploding gradient effects.

8. Spiking Equilibrium Propagation

One problem with Equilibrium Propagation (see Section 2.7), from a biological plausibility perspective, is that neurons communicate real-valued activations to one another - i.e. it does not pass Gap 2: Spiking. The algorithm depends on the network settling to an energy-minimizing state $s^* = \arg\min_s E_\theta(s, x)$, where s are the states of neurons in the network and x is some input. The network does this by following dynamics $\frac{\partial s}{\partial t} = -\frac{\partial E_\theta(s,x)}{\partial s} = \rho'(s)(w_{xs}x + w_{ss}s + b) - s$, where ρ is a nonlinear activation function, $w_{xs}, w_{ss}, b = \theta$ are the weight matrices and biases. When the dynamics is approximated in discrete time with an Euler update rule, we get $s_{t+1} = s_t + \epsilon\left(\rho'(s_t)(w_{xs}x + w_{ss}s_t + b) - s_t\right)$

In this work, we address the question of how Equilibrium Propagation could work if neurons faced with the same sort of communication bottleneck that biological neurons have. We impose the constraint that, at each time step, a neuron may communicate a 1 or a 0 to the other neurons in the network. The question is how to design the dynamics such that the network efficiently settles to the minimum of an energy function.

The most efficient way to encode a bounded real value $v \in [0, 1)$ with a stream of bits is to bisect the range at each iteration, transmitting a 1 if the value is below a bisection point and a zero otherwise. For example the value 0.4257 would be sequentially be encoded as [0, 1, 1, 0, 1], and reconstructed with increasing precision at [0.25, 0.375, 0.4375, 0.40625, 0.421875]. In this scheme, each additional bit communicates an increasingly fine *increment* to the estimate of the value being transmitted.

However, what do we do when the underlying value is changing at the same time we are transmitting it? This is the case for the neurons in the energy-based network used in Equilibrium Propagation, where neurons states change as the network settles. Intuitively, we still want neurons to use their bits to communicate their states with increasingly high resolution as the network converges. However our encoding/decoding scheme has to account for the fact that the underlying value being encoded will change as encoding is taking place. Our solution is not, as one might hope, a clean, theoretically grounded communication scheme, but an ad-hoc combination of predictive coding, sigma-delta modulation, and adaptive-step-sized averaging.

The main contribution of this work, then, is not to propose a clean solution, but to bring attention to the problem. Energy-based models propose that inference is the minimization of a global energy function - i.e, the process of finding $s^* = \arg\min_s E_\theta(s, x)$, where x is some input and s is the states of all neurons in the network. When neurons are restricted in their communication, how can the network most efficiently converge to find s^*?

8.1. Abstract

Backpropagation is almost universally used to train artificial neural networks. However, there are several reasons that backpropagation could not be plausibly implemented by biological neurons. Among these are the facts that (1) biological neurons appear to lack any mechanism for sending gradients backwards across synapses, and (2) biological "spiking" neurons emit binary signals, whereas back-propagation requires that neurons communicate continuous values between one another. Recently Scellier and Bengio [2017], demonstrated an alternative to backpropagation, called Equilibrium Propagation, wherein gradients are implicitly computed by the dynamics of the neural network, so that neurons do not need an internal mechanism for backpropagation of gradients. This provides an interesting solution to problem (1). In this paper, we address problem (2) by proposing a way in which Equilibrium Propagation can be implemented with neurons which are constrained to just communicate binary values at each time step. We show that with appropriate step-size annealing, we can converge to the same fixed-point as a real-valued neural network, and that with predictive coding, we can make this convergence much faster. We demonstrate that the resulting model can be used to train a spiking neural network using the update scheme from Equilibrium propagation.

8.2. Introduction

The human brain, a network of around 10^{11} neurons, consumes around 20W [Ling, 2001]. For comparison, a Titan X GPU running real-time object detection with YOLO [Redmon et al., 2016], a network of around 10^7 neurons, consumes 250W. In the quest for more efficient hardware for deep learning, biology is a not a bad place to start looking.

The "neurons" used in deep learning are so-named because of their loose correspondence to biological neurons. There are however, a number of fundamental differences between the types of neurons used in deep learning and those we observe in biology [Crick, 1989]. Among them are :

1. **Biological Neurons do not do Backpropagation**: Neurons used in deep learning emit two types of signals - an activation on the forward pass, and a gradient on the backward pass. Biological neurons send signals down a one-way signalling pathway called an *axon*. They appear to lack any secondary signalling mechanism for sending gradient backwards.

2. **Biological Neurons communicate with Spikes**: Neurons in deep learning have continuous, differentiable activation functions. This is necessary in order to propagate useful gradients back through the network. Biological neurons are best understood as dynamical systems, which output streams of all-or-nothing signals called "spikes", which are some function of recent inputs to the neuron.

These two characteristics pose a conundrum to those looking to reconcile theories in machine-learning with how the brain might be reasonably expected to operate. The recent successes in deep learning have been based on achieving gradient-descent by propagating error-gradients backwards through a network. But it is not clear at all how biological neurons could achieve this.

To address the "no biological backprop" problem, Scellier and Bengio [2017] proposed Equilibrium Propagation. This showed how one may propagate gradients through a deep network in a setting where neurons only produce one type of signal - the forward activation. The authors use a continuous Hopfield network [Hopfield, 1984] - a symmetrically-weighted neural network whose dynamics are defined according to the gradient of an energy function ($\frac{\partial s}{\partial t} \propto -\frac{\partial E}{\partial s}$, where s is the state of the neurons). Learning is based on allowing the network to converge to a fixed-point conditioned on the input data, then perturbing the output units towards the target, letting the network settle again, and then updating parameters to minimize a contrastive loss between the original fixed-point state and the perturbed fixed-point. Their work showed a semi-plausible mechanism by which biological neural networks (or artificial networks implemented as analog circuits) may be able to achieve gradient descent.

The original formulation of Equilibrium Propagation, however, still assumes continuous-valued units. In this paper, we constrain neurons to emit binary-valued signals, and look at how neurons can efficiently convey their real-valued activations to other neurons despite this bottleneck. Specifically, we show how a network of neurons can efficiently minimize an energy function when neurons are "spiking" - i.e. constrained to only communicate binary values at each time-step.

This line of research may be of interest for designing the next generation of neural network hardware. A continuous-dynamical system can be implemented with an analog circuits, but electrical issues such as capacitance, inductance, and cross-talk make it difficult to faithfully transmit analog values over a circuit. Digital signals, by comparison, can be transmitted with ease. The brain appears to use a hybrid approach, with neurons having analog internal dynamics but communicating with one another using digital "spikes".

8.3. Background

8.3.1. A Neural Network as a Dynamical System

Suppose we have a network of recurrently connected neurons with symmetric weights ($w_{ij} = w_{ji}$). This is known as a continuous Hopfield Network. Hopfield [1984] proposed an energy-function for such a network, which can be defined as:

$$E(s) = \frac{1}{2}\sum_{u} s_i^2 - \sum_{i \neq j} w_{ij}\rho(s_i)\rho(s_j) - \sum_{i} b_i \rho(s_i) \tag{8.1}$$

Where s_i is the activation of neuron i, w_{ij} and b_i are model parameters, and ρ is a nonlinearity. Scellier and Bengio [2017] use a hard sigmoid function: $\rho(s) = [s]_0^1$, where $[\cdot]_a^b$ indicates that values outside the range of a and b are clipped to these limits. Given this energy function, we can define the temporal dynamics that minimize this energy with respect to activations:

$$\frac{\partial s_j}{\partial t} = -\frac{\partial E(s_j)}{\partial s_j} = -s_j + \rho'(s_j)\left(\sum_i w_{ij}\rho(s_i) + b_j\right) \tag{8.2}$$

Where $\rho'(s_j)$ is the derivative of the activation function about s_j. For implementation in discrete time, this can be expressed as a difference-equation (this is known as the *Forward Euler Method*):

$$s_j^t = \left[(1-\epsilon)s_j^{t-1} + \epsilon\rho'(s_j^{t-1})\left(\sum_i w_{ij}\rho(s_i^{t-1}) + b_j\right)\right]_0^1 \tag{8.3}$$

Where $[\cdot]_0^1$ indicates clipping to range $[0, 1]$ and $\epsilon \in (0, 1)$ can be seen either as the size of the time-step or as the learning-rate of the activations. This update will converge to the optimum for a sufficiently small ϵ (e.g. $\epsilon = \frac{1}{2}$).

8.3.2. Equilibrium Propagation

We refer the reader to Section 2.7, where we introduce Equilibrium Propagation.

8.4. Binary Communication

Suppose we now operate under the constraint that neurons can only output binary values at each time-step. Our objective is to optimally converge to the same fixed-points as the continuous-valued dynamical system, under the constraint of binary communication between neurons. In other words, we constrain our neurons to obey the interface:

$$q_j^t, s_j^t, z_j^t = f(q_{\backslash j}^{t-1}, s_j^{t-1}, w_{\backslash j,j}, b_j, z_j^{t-1}) \tag{8.4}$$

Where $q_j^t \in \{0, 1\}$ is the binary output of neuron j, $q_{\backslash j} \in \{0, 1\}^D$ are the binary signals of other neurons in the network, $s_j^t \in \mathbb{R}$ is the *external state* associated with a neuron, $w_{\backslash j,j}, b_j$ are the parameters associated with neuron j, and z_j is the *internal state* of encoders and decoders which we will discuss in the following section. Note that the only values that are communicated between neurons are the binary q_j's. Our goal is to design our neurons so that despite being limited by binary communication, the states s in our network to converge to the same fixed point as they would when following the real-valued dynamics of Equation 8.5.

We propose to design our neurons as follows:

$$\begin{aligned} u_j^t &= \sum_i w_{ij} q_i^{t-1} \\ v_j^t, z_{dec,j}^t &= dec(u_j^t, z_{dec,j}^{t-1}) \\ \epsilon_j^t, z_{anneal,j}^t &= anneal(\epsilon_j^{t-1}, v_j^t, z_{anneal,j}^{t-1}) \\ s_j^t &= [(1-\epsilon_j^t) s_j^{t-1} + \epsilon_j^t \rho'(s_j^{t-1}) \left(v_j^t + b_j\right)]_0^1 \\ q_j^t, z_{enc,j}^t &= enc(\rho(s_j^t), z_{enc,j}^{t-1}) \end{aligned} \tag{8.5}$$

Where *enc* and *dec* are functions for encoding and decoding signals between neurons, *anneal* is a function of updating the step size ϵ, and the form of internal state variables $z_j = (z_{dec,j}, z_{anneal,j}, z_{enc,j})$ will be defined in the following sections.

In this work we show how various definitions of *enc*, *dec* and *anneal* affect the convergence of our discrete dynamics to the true minimum of the energy (Equation 8.1). In the following sections we propose a quantization method that allows our neurons to efficiently settle towards this fixed point.

8.4.1. Stochastic Approximation

One approach we could take is to look at this as a Stochastic Approximation problem from the perspective of each neuron. The task of Stochastic Approximation is to keep an online estimate $\hat{\theta}^t$ of a time-varying parameter θ^t from a stream of noisy samples $x^t = \theta^t + \zeta^t$, where ζ^t is some unbiased noise. When θ^t is not constant in time, we stay the input is *nonstationary*.

Robbins and Monro [1951] showed that if the nonstationarity is transient (θ^t converges to a final value over time), we can sequentially average out the noisy samples to form estimates:

$$\hat{\theta}^t = (1-\epsilon^t)\hat{\theta}^{t-1} + \epsilon^t x^t \tag{8.6}$$

If we anneal the step-size (or learning rate) ϵ^t in such a way that $\sum_{t=0}^{\infty} \epsilon^t = \infty$ and $\sum_{t=0}^{\infty} (\epsilon^t)^2 < \infty$, then our estimator eventually converges to the true parameter values ($\lim_{t\to\infty} \hat{\theta}^t = \theta^t$). For stationary problems, when $\theta^t = \theta^0 : \forall t$, the optimal annealing schedule is $\epsilon^t = \frac{1}{t}$, which corresponds to a simple moving average. For *nonstationary* signals (e.g. the activations in our network, which undergo some transient dynamics before settling), we can converge faster by forgetting early samples, so that the average is not corrupted by stale values. There are a number of ways to do this [George and Powell, 2006]. A simple one is to schedule the step-size as:

$$\epsilon^t = \frac{\epsilon^0}{(t)^\eta} \tag{8.7}$$

With the exponent $\eta \in (\frac{1}{2}, 1)$. This guarantees that as $t \to \infty$, the inputs at $t = 0$ diminish to have zero weight relative to the most recent inputs, but the average still smooths over an ever-growing number of samples.

8.4.2. A Naive Approach: Stochastic Rounding and the Robinson-Munroe Annealing

In our case, the "true" parameter θ corresponds to the total *pressure* exerted on neuron j by the rest of the network: $\rho'(s_j^{t-1}) \left(\sum_i w_{ij}\rho(s_i^{t-1}) + b_j\right)$ (from Equation 8.3). The noise arises from trying to represent real signals with a temporal stream of bits. The non-stationarity arises from the fact that the rest of the network has not yet settled to the fixed point. Note that our estimate itself affects future inputs: Neurons are connected recurrently in a network and the estimator in neuron i affects the estimator in neuron j which in turn affects the estimators in neuron i.

Suppose each input neuron i in Equation 8.5 stochastically outputs bits $q_i^t \sim \text{Bernoulli}(\rho(s_i))$, where $\rho(s_i) \in (0, 1)$ is the neuron's activation. Since q_i^t is an unbiased estimator of $\rho(s_i)$, a neuron j receiving this signal should eventually average it out, along with all its other inputs, to achieve a correct estimate of $\rho'(s_j^{t-1}) \left(\sum_i w_{ij}\rho(s_i^{t-1}) + b_j\right)$, provided that its input neurons do indeed converge to the correct fixed point s_i^{-}. A simple communication scheme can then be described (with reference to the variables in Equation 8.5) as:

$$q^t = \text{Bern}(\rho(s^t)) \qquad \text{Stochastic Encoder} \tag{8.8}$$

$$v^t = u^t \qquad \text{Identity Decoder} \tag{8.9}$$

$$\epsilon^t = \frac{1}{(t)^\eta} \qquad \text{Annealer} \tag{8.10}$$

8.4.3. Better Averaging with Adaptive Step Sizes

In choosing the step-size for stochastic optimization, we face a trade-off. Small step sizes allow us to average out noise, but also cause our current estimates to include outdated values of the time-varying parameter θ^t. It can therefore be advantageous to adaptively adjust our step size according to our estimate of how nonstationary θ is. George and Powell [2006] review several existing step-size adaptation algorithms and propose one of their own called Optimal Step-Size Adaptation (OSA) which, similarly to a Kalman filter, adjusts its step-size according to the ratio of the estimated *drift* in the underlying parameter and the *noise* in the measurement. OSA is "optimal" in the sense that it optimally estimates the parameter θ if the drift and noise are known. Since they are not - OSA also estimates these quantities, and bases the step-size on these estimates. OSA has only a single parameter, $\bar{\nu}$, which is the target learning rate for estimating the drift and noise. The full algorithm is included in Appendix E.1.

8.4.4. Better Encoding with Sigma-Delta Modulation

There are more efficient ways to communicate a time-varying real value than to send random bits centered around that value. A simple method from signal processing for encoding time-varying signals is Sigma-Delta modulation [Candy and Temes, 1962]. Suppose we have a time-varying input signal $x_1, ...x_t$ where $x_\tau \in (0,1) \forall \tau$. We then quantize x_t into q_t according to:

$$\begin{aligned} \phi' &= \phi^{t-1} + x^t \\ q^t &= \left[\phi' > \frac{1}{2} \right] \quad \text{Sigma Delta Encoder} \\ \phi^t &= \phi' - q^t \end{aligned} \tag{8.11}$$

Where $[a > b]$ evaluates to 1 if $a > b$ and 0 otherwise. By expanding Equation 8.11 recursively, we can verify that if $x^t \in (0,1)$ and $\phi_0 = 0$, the mean quantization error is bounded: $\frac{1}{T} \left| \sum_{t=0}^{T} (x^t - q^t) \right| \leq \frac{1}{2T}$. So we have $\mathcal{O}(1/T)$ convergence, compared to the $\mathcal{O}(1/\sqrt{T})$ convergence that we would get from averaging out a stochastic estimator.

Note that this corresponds to an "integrate-and-fire" quantization - inputs are added to a "potential" ϕ, and once that potential crosses a threshold a "spike" $(q^{(t)} = 1)$ is sent out, and subtracted from the potential. Sigma-Delta modulation has previously been used as a model of the neural spiking mechanism: [Yoon, 2016], Zambrano and Bohte [2016], O'Connor et al. [2017].

8.4.5. Better Bit-Economy with Predictive Coding

When the signal is time-varying, it seems like a poor use of bandwidth to simply communicate a stream of bits that averages out to the current signal value. Instead, we can use an encoding scheme wherein neurons primarily send temporal *changes* in the signal value to downstream neurons, and downstream neurons integrate these changes. This is an instance of *Predictive Coding*, a widely used concept in the Signal Processing literature. Predictive Coding has in the past been proposed as a possible mechanism in neural communication. [Srinivasan et al., 1982], [Shin, 2001], [Tewksbury and Hallock, 1978], [Bharioke and Chklovskii, 2015].

Lossy Predictive Coding is a method for efficiently encoding a real-valued signal as a bitstream, and decoding it again on the other end of a communication channel. At each time-step, a *predictor* attempts to predict the current signal from past signal values, and the prediction is subtracted from the signal before quantization. On the receiving end, the same predictor is used to reconstruct the signal from the stream of bits. In the case where the predictor is a linear function of past inputs, we can exploit the commutativity of the weight-multiplication and decoding operations [O'Connor et al., 2017] to sandwich a weight matrix between the encoders and decoders. Here, we formulate an extremely simple predictor $\text{Pred}(x^{t-1}, ...x^0) = (1-\lambda)x^{t-1}$ where $\lambda \in (0,1)$. We write our encoder (with reference to the variables in Equation 8.5) as:

$$\begin{aligned} a^t &= \frac{1}{\lambda}\big(\rho(s^t) - (1-\lambda)\rho(s^{t-1})\big) \quad \text{Predictive Encoder} \\ q^t &= Q(a^t) \end{aligned} \tag{8.12}$$

Where Q is some (possibly stateful) quantization procedure, such Sigma-Delta modulation (Equation 8.11) or Stochastic Rounding (Equation 8.8). On the decoding side, we sum up the weighted quantized inputs and invert the encoding function:

$$\begin{aligned} u_j^t &= \sum_i w_{ij} q_i^{t-1} \\ v_j^t &= (1-\lambda)v_j^{t-1} + \lambda u_j^t \quad \text{Predictive Decoder} \end{aligned} \tag{8.13}$$

When λ is close to 0, we have a system that only sends *changes* in state, and accumulates these change in a running sum. When λ is 1, we recover the case with no predictive coding.

8.4.6. Lambda-Annealing

As was the case with ϵ, it is also possible to anneal the prediction-factor λ. Intuitively, we would like to start the convergence process with a very short memory (λ close to 1), primarily using bits to communicate the rapidly changing current state. Later, as we approach a fixed point, we would like to lengthen the memory (λ close to 0) and use our bits to communicate increasingly fine increments to the state.

8.4.7. The Resulting Model

Combining Sigma-Delta encoding from Equation 8.11 with the predictive encoder/decoder of Equations 8.12 and 8.13 by plugging them all into Equation 8.5 results in a biologically-plausible model that applies double-exponential smoothing to inputs and produces output

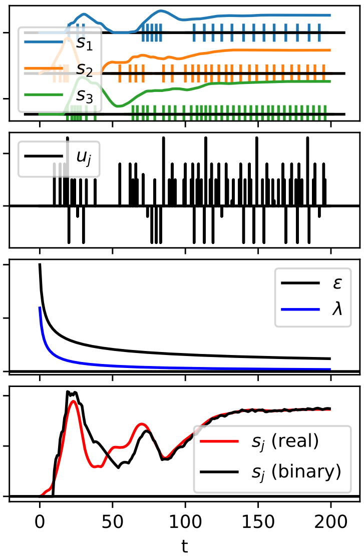


Figure 8.1.: An illustration of the evolution of a neuron in response to converging inputs. **Top:** The values of three input neurons as they converge towards a fixed point. Tick marks indicate the times where the encoders of those neurons output a 1. **Row 2:** The total weighted input from the input neurons to a post-synaptic neuron. **Row 3:** the step size ϵ and predictive-coding parameter λ as they anneal. **Bottom:** A comparison of the value of the post-synaptic neuron under the continuous dynamics (Equation 8.3, red curve) and our binary dynamics (Equation 8.14, black curve).

spikes with an integrate-and-fire mechanism:

$$\begin{aligned}
v_j^t &= (1-\lambda^t)v_j^{t-1} + \lambda^t \sum_i w_{ij} q_i^{t-1} \\
s_j^t &= [(1-\epsilon^t)s_j^{t-1} + \epsilon^t \rho'(s_j^{t-1})\left(v_j^t + b_j\right)]_0^1 \\
a^t &= \frac{1}{\lambda^t}\left(\rho(s^t) - (1-\lambda^t)\rho(s^{t-1})\right) \qquad (8.14) \\
q_j^t &= [\phi_j^{t-1} + a_j^t > \frac{1}{2}] \\
\phi_j^t &= \phi_j^{t-1} + a_j^t - q_j^t
\end{aligned}$$

Figure 8.1 shows some example dynamics from our model.

8.5. Experiments

We explore several combinations of the hyperparameters ϵ^t, λ^t, introduced in Section 8.4. First in Section 8.5.1, we compare the rate at which these various hyperparameter settings converge to the fixed point for randomly initialized networks. Then in Section 8.5.2 we apply the more promising settings to train a neural network on the MNIST dataset.

8.5.1. Convergence

To understand how our encoding/decoding parameters affect the rate of convergence, we use a randomly initialized network with 3 layers of [500-500-10] units, where the first is considered the "input" layer and is clamped to a random input vector ($s_{in} = x$). We simulate the two-phase learning of Equilibrium Propagation by running the network for a fixed number of steps (corresponding to the negative phase), then adding a perturbation to the output layer, and allowing the network to settle again (corresponding to the positive phase). We compare scheduled and adaptive (OSA) step-size annealing with and without predictive coding. For each annealing scheme we compare, we take the optimal hyper-parameters as found by a Gaussian Process optimizer which attempts to minimize the error after 250 steps of convergence. We find that the best convergence is obtained by combining OSA step-size annealing with a predictive coding (with parameter λ). The results can be seen in Figure 8.2.

8.5.2. Equilibrium Propagation on MNIST

We applied our quantization methods to train our binary-valued network on the MNIST dataset using Equilibrium Propagation. We compare to our implementation of continuous-valued Equilibrium-Propagation by Scellier and Bengio [2017] for a network with [784-500-10] units in each layer. Unlike the author's implementation, we did not use the trick of keeping persistent activations per training sample between epochs. This trick would have improved the performance of both the continuous and binary network but would not be useful in drawing conclusion about the performance of the binary network relative to the continuous one.

In order to reach similar performance to the continuous-valued network, we had to extend the positive and negative phases to (100, 50) steps respectively, compared to the (20, 4) steps used in real-valued equilibrium propagation. We found no significant difference between the networks trained with OSA and with annealing schedules (after finding the optimal annealing-parameters for each model with a Gaussian-Process parameter search - see Appendix E.3). In the remainder of this section, we report the results for the OSA model with a constant $\lambda = 0.275$ predictive-coding coefficient. Our binary-network performed similarly on the test set to the continuous-valued-network (see Figure 8.3),

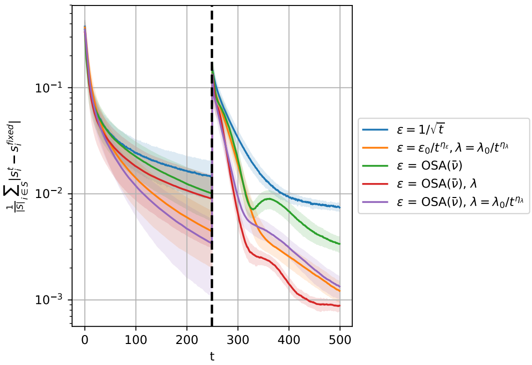


Figure 8.2.: A network is presented with a constant input at $t = 0$, and allowed to settle. Then at $t = 100$ the output layer is perturbed, and the network is allowed to settle to the new fixed point. The y axis indicates the log-mean-error between the true fixed-point s^{fixed}, and the current state of s of the quantized network. s^{fixed} is calculated by running a continuous-valued network (with the same parameters and input) to convergence. Note that s^{fixed} is different for $t < 100$ and $t \geq 100$ due to the external perturbation at $t = 100$. Shaded regions indicate the standard deviation over 20 runs with randomly initialized networks.

achieving a (2.37% test / 0.15% training) error, compared to the continuous network's (2.51% test / 0.25% training) error.

We also ran our model on a deeper network with 3 hidden layers [784-500-500-500-10], and found that our network slightly underperformed the continuous-valued network, achieving (3.65% test/, 3.02% training) error vs (2.42% test/ 0.27 % training) error for the continuous-valued network. The discrepancy is likely due to the quantized network needing longer negative/positive convergence phases. The learning curves and details on the paremetrizations of these experiments can be found in Appendix E.3.

8.6. Discussion and Related Work

Much of the work in the stochastic approximation literature is about how to adapt step-sizes according to the statistics of the incoming sample stream [Chau and Fu, 2015], [George and Powell, 2006]. It is unclear whether we can transfer adaptive step-size algorithms to the similar task of choosing optimal predictive coding coefficients (λ). The difficulty is that if the encoder of neuron i has a different predicitve coefficient λ than the decoder of neuron j, the signal will not be correctly transmitted, but the binary-communication constraint prohibits us from directly transmitting predictive coding coefficients between neurons. It may be possible to define an adaptive predictive coding

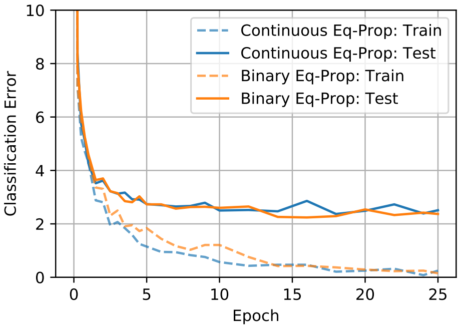


Figure 8.3.: Learning curves on the MNIST dataset, on an equilibrium propagation network with one hidden layer (with OSA step-size adaptation and a fixed $\lambda = 0.275$ predictive coding parameter. **Blue Curves**: Training/Test set Learning curves of our implementation of "Continuous-Valued" Equilibrium propagation by Scellier and Bengio [2017]. **Orange Curve:** Scores from our Binary-Values implementation with the best-performing quantization hyperparameters from Figure 8.2.

rule where the predictive coefficients of different neurons tend to converge so that the network reaches the correct fixed point in the end, but this needs more work. [Bharioke and Chklovskii, 2015] suggested that including a nonlinearity in the predictor can have the effect of rapid online adaptation of prediction coefficients. Adaptive predictive coding could allow us to perform efficient inference on nonstationary data. We can imagine constructing a network where neurons use their bits to communicate fine changes in state when the network is near a fixed point but then adapts itself to take larger, coarser steps when it sees the input has started changing rapidly (e.g. during a *saccade*).

Mesnard et al. [2016] also implemented Equilibrium Propagation with spiking neurons. Their model was primarily built to mimic the leaky-integrate-and-fire dynamics of biological neurons. There was no annealing and their neurons do not converge to a fixed point when presented with a constant input. They demonstrate that the model can train on a toy dataset, but it is not clear if this approach would scale to a more standard machine learning task.

There are still several obstacles to doing truly biologically plausible deep learning. One subtle issue is that we rely on "capturing" a negative state s^- and a positive state s^+ in order to do the parameter update (Equation 2.12). This requires holding onto two states at once - something which biological neurons seem unlikely to do. If we instead take the approach of using the rate of change of the postsynaptic neuron at the beginning of the positive phase, as suggested in Bengio et al. [2015b], Scellier and Bengio [2017], (i.e. $\Delta w_{ij} \propto \rho(s_i)\frac{\partial s_j}{\partial t}$) we have the problem that we get very noisy updates due to the quantization.

Another lingering biological-implausibility that is not specific to our algorithm is that we still use symmetric weights. Learning with symmetric weights implies an additional communication channel to synchronize synapse w_{ij} of neuron j to synapse w_{ji} of neuron i. However recent work by [Scellier et al., 2018] and Lillicrap et al. [2014] seems to suggest that this may not be a problem, because weights tend to align themselves to be approximately symmetrical through learning anyway. Another obstacle is that biological networks do not appear to have distinct forward/backward passes, or negative/positive phases (except perhaps on the very slow timescale of the sleep/wake cycle). It is still not clear how one could do learning in a setting where new data is continuously coming in and we do not have the luxury of pausing our sensory input while we wait for our brains to do a forward/backward pass or settle to an energy minimum. Even if we could, we face the problem of Catastrophic Forgetting, wherein deep networks tend to forget old training data when presented with a nonstationary stream of training samples (though Kirkpatrick et al. [2017] have done some work addressing this). Finally the question of how to learn temporal sequences without doing backpropagation-through-time remains an open one, though recent works such as Ollivier et al. [2015] and Tallec and Ollivier [2017] have begun to address this.

8.7. Conclusion

We demonstrate that we can train a network with Equilibrium Propagation even when neurons are constrained to only communicate binary values. To achieve efficient communication between neurons, we use ideas from Stochastic Approximation and Predictive Coding.

We believe that this work is relevant to designing of the next generation of neural computing hardware. In modern computers, most of the energy cost is not spent on computation (in terms of adds and multiples), but in *moving data around* Horowitz [2014]. It seems likely that future neural computing hardware will consist of neurons implemented as physical circuits, with computation co-located with memory, so that parameter values never need to be moved. The main energetic bottleneck will then be the *communication* between neurons. In the brain, Attwell and Laughlin [2001] estimate that 81% of metabolic energy is spent on sending signals between neurons. It makes sense then, that neurons should have evolved to use the minimum number of spikes to communicate what they need to communicate. If the brain is doing something similar to Equilibrium Propagation, then neurons compute by collectively trying to find the fixed point of a dynamical system. Our work addresses the question of how we can efficiently find this fixed point when there is a communication bottleneck between neurons.

Code is available at `https://github.com/quva-lab/spiking-eqprop`

9. Discussion

It should be clear at this point that we are still a few breakthroughs away from having a learning system which shares the basic characteristics of biological learning. Here we will review the "Gaps" between biological and machine learning systems that we discussed in Section 1.5, and review our contribution to closing them, while being forthright about the shortcomings of our own approaches.

9.1. The Gaps

Gap 1: No Backprop

Rather than invent our own way out of backprop, we stood on the shoulders of giants and built on Equilibrium Propagation [Scellier and Bengio, 2017] as a method for training a neural network without backprop. In *Initialized Equilibrium Propagation* (Chapter 7), we showed how the we can use Equilibrium Propagation to provide (approximate) gradients to train a feedforward network. In *Spiking Equilibrium Propagation* (Chapter 8), we showed how this idea could work even when communication between neurons is bottlenecked.

The problem is that Equilibrium Propagation is not a useful or complete answer to backprop-free training. Each iteration of training involves an iterative settling process, and the time required for this settling process to converge gets longer as the network gets larger. Our approach of initializing settling with a feedforward network, in *Initialized Equilibrium Propagation*, only partially alleviated the slow-convergence problem. Especially as the network got deeper, it still required many iterations of convergence per training iteration to provide suitable gradients for training the feedforward network.

Perhaps the solution is to give up on the idea that biological networks even try to minimize a global loss by gradient descent. Rather, it is possible that each layer of neurons greedily optimizes some local objective, without explicitly optimizing any well defined loss function. The idea of greedily training layers on local objectives goes back to Hinton et al. [2006]. Recently, Löwe et al. [2019] showed that for some image and speaker identification tasks, a greedy, unsupervised local training scheme on the lower layers can substantially help supervised tasks that are learned on higher-layer representations.

Intuitively, it makes sense that there should be some top-down feedback in feature learning. For an animal in the wild, only a small subset of the sensory world is relevant to survival, and it makes sense that low-level feature extractors should be tuned to look for things that matter in the environment. However, this top-down feedback need not be internal to the network. If the higher layers control the movements of the animal's eyes, it may simply be that the feedback comes in the form of where the animal chooses to look - lower layers then adjust themselves to learn a feature representation of whatever the higher layers make them look at.

However, neuroanatomy suggests that in-network top-down feedback does play an important role in perception. A review by Chen et al. [2009] suggests that top-down connections in the brain may serve primarily to modulate the activations of bottom-up connections. One interpretation of this [Rao and Ballard, 1999] is that top-down connections attempt to predict the future states of lower layers, and it is these prediction errors that are transmitted up. Such a scheme turns the backpropagation-trained network on its head. Higher layers predict the future activations of lower layers via feedback connections, and feedfoward connections transmit the prediction *errors* back up.

It thus remains unclear how, or even if, biological networks do assign credit to low-level parameters for their contribution to solving high-level objectives. The circuitry is there - there are ample feedback connections to communicate error information back to lower layers - but it is still unclear of how a low level synapse could know its contribution to solving a high level objective. Equilibrium Propagation, and its extensions [Scellier et al., 2018] provides an enticing framework for how this may happen, but it makes too many biologically unrealistic, not to mention computationally inefficient, assumptions (see following sections) to be a solution on its own.

Gap 2: Spiking

One conclusion of this thesis is that it seems likely that the brain uses some form of intra-neuron predictive coding[1] to communicate values between neurons. Predictive coding makes sense from a signal-processing standpoint because it aims to make maximally efficient use of spikes to transmit the unpredictable components of a signal - allowing the boring and predictable components to be reconstructed on the other end.

In Chapter 4 we proposed communicating *change* of state between neurons to reduce inter-neuron communication when processing temporal data. In Chapter 5 we proposed that the transmitted signal should really be a combination of the current value of the signal and the rate of change, in order for training to work - and showed that this was an instance of predictive coding. In Chapter 6 we showed that predictive coding, when combined with dynamics-based learning, actually explains the Spike-Timing-Dependent Plasticity (STDP) learning rule first observed in experimental neuroscience [Markram and Sakmann, 1995]. In Chapter 7 we did not address the issue at all, and in Chapter 8 we combined this scheme with a dynamical model of neurons and experimented with

[1] We say *intra-neuron* to contrast with the *network-level* forms of predictive coding proposed by Rao and Ballard [1999] and Oord et al. [2018], which the brain may also be doing for all we know.

dynamically adapting the encoding and decoding coefficients to allow a dynamical network to optimally converge to the correct fixed-point.

Given that we were perusing the goal of reducing inter-neuron communication, it is interesting that the resulting model shares a lot of features in common with biological neurons.

The neuron-model we arrived at in Chapter 8 was:

$$\begin{aligned}
v_j^t &= (1-\lambda)v_j^{t-1} + \lambda \sum_i w_{ij} q_i^{t-1} \\
s_j^t &= [(1-\epsilon)s_j^{t-1} + \epsilon\rho'({s_j}^{t-1})\left(v_j^t + b_j\right)]_0^1 \\
a^t &= \frac{1}{\lambda}\left(\rho(s_j^t) - (1-\lambda)\rho(s_j^{t-1})\right) \\
q_j^t &= [\phi_j^{t-1} + a_j^t > \frac{1}{2}] \\
\phi_j^t &= \phi_j^{t-1} + a_j^t - q_j^t
\end{aligned} \tag{9.1}$$

Converted to a continuous time dynamical system [2], this becomes:

$$\dot{v_j} = \lambda\left(-v_j + \sum_i w_{ij} q_i\right) \tag{9.2}$$

$$\dot{s_j} = \epsilon\Big(-s_j + \rho'(s_j)(v_j + b_j)\Big) \tag{9.3}$$

$$\dot{a_j} = \frac{1}{\lambda}\dot{\rho}(s_j) - \rho(s_j) \tag{9.4}$$

$$q_j = \delta_{\phi_j > \frac{1}{2}} \tag{9.5}$$

$$\dot{\phi_j} = a_j - q_j \tag{9.6}$$

What's interesting here is that we came to this neuron model by (1) trying to train a network without backprop and (2) trying to design an efficient model by bottlenecking communication between neurons. Yet without explicitly designing for it, our model has several characteristics of biological neurons.

1. **Double-Exponential smoothing:** Equations 9.2 and 9.3 implement double-exponential temporal smoothing on their inputs. In a biological neuron, the first smoothing on v can be seen as the effect of the input-spike on the input-current to cell, and the second, on s, can be seen as the effect of the leaky integration of the membrane potential.

[2] The $[\cdot]_0^1$ term disappears when $\rho(x) = [x]_0^1$, because it is already impossible for s_j to be driven out of the range $[0, 1]$

2. **Integrate-and-Fire Spiking:** The spiking mechanism in Equations 9.5 and 9.6 involves integrating a potential until a it crosses a threshold and then resetting. This quantization scheme is known as *Sigma-Delta Modulation*, and we use it because it allows downstream neurons to estimate the value of s_j faster than if we were to use another scheme like a Poisson process for conveying a time-varying real-value in bits.

In summary, we propose that the spiking code arises from neurons trying to communicate real-values to one-another with as few spikes (and thus as little energy expenditure) as possible. Under our model, neurons communicate using a combination of predictive-coding and sigma-delta modulation to convey their values to one another.

Gap 3: Nonstationary Data

In Chapters 4 and 5 we dealt with one aspect of the Non-IID data problem. We propose how, when the data arrives as a temporally redundant (and therefore non-IID) time-series, we can avoid the redundant computation that comes with passing every frame though the network independently. Instead we propose a quantized communication scheme wherein neurons primarily send their state *change* to other neurons.

There are other aspects to the IID data problem that we have not addressed here:

- **Learning without Catastrophic forgetting**. When the data distribution shifts over training, a network trained on this data experiences a phenomenon known as *Catastrophic Forgetting*, wherein old data are completely forgotten as new data come in. Several authors, [Nguyen et al., 2017], [Kirkpatrick et al., 2017], have proposed how this problem can be mitigated by selectively making some parameters of the network less *learnable* once they have become useful for one task.

- **Credit assignment through time**. Non-IID data implies that target variables y_t are not independent of past inputs $x_0..x_{t-1}$. For infinite streams of data, back-propagating back to the beginning of the sequence at each step is infeasible, and truncating the number of time-steps introduces a bias. This problem is tackled by Tallec and Ollivier [2017], where they propose a scheme for approximate credit-assignment in this “infinite non-IID data stream” setting.

Gap 4: Asynchronous Processing

We did not discover an “asynchonous” way to train neural networks. However, as we worked on this problem, it became clear that there are a few aspects to the notion of an “asynchonous” network:

- **No Locking:** Jaderberg et al. [2016] first described the problem of “locking” in neural networks - where a neuron that has just completed its forward pass is “locked”

- in the sense that it must remember the state of its inputs - until it receives a backpropagated gradient. Neurons in Equilibrium propagation do not have a locking problem (they may forget their past input once they've computed their output), but they achieve this at the cost of introducing a new type of non-asynchronicity:

- **No loops within loops:** Equilibrium Propagation requires an iterative convergence on each round of training, during which the input data must be held constant. In a network running in real-time, this is an unreasonable constraint.
- **Communication does not scale with frame-rate:** As we discussed in Section 1.6.1, real-world data often does not come in complete frames describing the entire state of the world, but may often come from several sensors operating at different rates, each describing an update to the world-state. Any model that needs to do a full update with each new datum is impractical in such a setting. The models we describe in Chapters 4: Sigma-Delta Quantized Networks and 5: Temporally Efficient Deep Learning with Spikes have the property that the rate of communication within the network need not scale with the frame-rate of the input data. However, there remains a problem with training. *Sigma-Delta Quantized networks* are only used for inference, not training, and the model in 5 is not asynchronous in the sense that it relies on backprop, and thus suffers from the *locking* sort of non-asynchronicity.

We are still missing a way to train networks in a truly asynchronous way. The primary reason for this is that it is unclear how to assign credit effectively in such a system. At present, all known forms of gradient-based credit assignment introduce some form of "locking" - wherein one component of the network must hold its state until a feedback signal arrives allowing it to update. This *locking* can either be on the level of the neuron (as in regular backprop) or on the level of the *network* (as in Equilibrium Prop).

Gap 5: No Shared Parameters

In this thesis we do not attack the problem of how we can train without shared parameters. However, this may not matter. Backprop requires that the network doing the backward pass use the same parameters as the forward pass. Equilibrium propagation, upon which we build our last two works, does not necessarily require this. As shown empirically by Scellier et al. [2018], when one simply removes the symmetric-weight constraint that $w_{ij} = w_{ji}$ (and therefore removes the existence of an energy-function), the algorithm still works. All that is required is that the network-dynamics to a fixed point. Symmetric weights guarantee this condition, but are not necessarily required.

9.2. Conclusion

There has been a lot of talk about *biologically plausible*, learning systems, and how they can help make machine learning systems more scalable, energy efficient, and applicable to raw, real-world data. But obviously not all characteristics of biological systems are necessary or desirable. To use a common analogy, airplanes do not flap their wings. Nor are they feathered. Engineers isolated the key principle that allows birds to fly - the airfoil, and proceeded to design aircraft from first principles rather than mimicry. Human flight would probably have gotten off to a much rocker start had engineers insisted on basing early aircraft designs on a wholesale replication of avian anatomy.

Likewise, we need to identify the core characteristics that make biological networks work, and ignore the rest. Thus, we would like to conclude by creating a checklist of the characteristics of biological systems that we believe are desirable to replicate in machine learning systems.

1. **Efficient Credit Assignment [ECA]** - Neurons update their parameters in the direction of the gradient, and learning is at least on the same order of efficiency as gradient-descent.

2. **Efficient Credit Assignment through Time [ECAT]** - Neurons can update their parameters to optimize future losses without keeping a buffer of their states and backpropagating through these.

3. **Non-Locking Neurons [NLN]** - Neurons simply process their inputs and produce outputs, they do not need to wait for an error signal to return to update before forgetting their input signals.

4. **Efficient Inference [EI]** - Inference can be performed on new data efficiently, i.e. without requiring a minimization process or scanning through past data for comparison.

5. **Asynchronous Data Processing [ADP]** - The network can receive sequential data in parts (e.g. the Camera/Gyro example from Section 1.6), and update its state without entirely recomputing a forward pass from scratch.

6. **Quantized Communication [QC]** - The components in the network communicate sparse, discrete signals.

7. **Continual Learning [CL]** - The network can learn from a non-IID data stream without catastrophic forgetting.

Table 9.1 matches these characteristics to our algorithms and a selection of related works discussed in this thesis.

At present, and to our knowledge, there is no learning algorithm that meets all of these criteria. It seems plausible, given progress on multiple fronts, that one will come along in the next few years. The discovery of a “brain-like” learning algorithm will not mean that the singularity has arrived and that we have entered into the era of artificial general intelligence - after all, fruit fly brains would probably fill all of the checkboxes in the above table. It will however, make it much easier to apply to machine learning to natural,

Algorithm	**ECA**	**ECAT**	**NLN**	**EI**	**ADP**	**QC**	**CL**
BackProp	✓			✓			
Equilbrium Prop Scellier and Bengio [2017]	✓^{1}		✓^{4}				
Difference Target Prop Lee et al. [2015]	✓^{1}			✓			
Synthetic Gradients Jaderberg et al. [2016]	✓^{1}		✓	✓			
Binary Weights and Activations Courbariaux et al. [2016]	✓^{1}			✓		✓	
Unbiased Online Recurrent Optimization Tallec and Ollivier [2017]	✓	✓^{2}	✓^{3}	✓			
Variational Continual Learning Nguyen et al. [2017]	✓			✓			✓
Deep Spiking Networks Chapter 3	✓^{1}		✓^{4}		✓	✓	
Sigma Delta Quantized Networks Chapter 4	✓^{1}			✓	✓	✓	
Temporally Efficient Deep Learning with Spikes Chapter 5	✓^{1}			✓	✓	✓	
Initialized Equilibrium Propagation Chapter 7	✓^{1}		✓^{4}	✓			
Spiking Equilibrium Propagation Chapter 8	✓^{1}		✓^{4}			✓	
The Brain	?^{5}	?^{5}	✓	✓^{6}	✓	✓	✓

Table 9.1.: 1. While these algorithms do have efficient credit assignment, they all make use of some approximation which may be detrimental. Specifically, the algorithm may work in small-scale networks but not generalize well at scale.
2. UORO's method of credit-assignment through time involves making stochastic approximations to the gradients. So the efficiency of estimating the gradient comes at the cost of added noise.
3. UORO has neurons that do not lock for the purposes of backpropagating gradients *back-through-time*, but they still face the locking-with-depth that ordinary networks do.
4. While these methods all have Non-Locking Neurons (which do not need to hold their state while waiting for a gradient to propagate back), they do require that network settles to a fixed point at every training iteration, which is a network-level form of "locking".
5. It is obviously difficult to compare biological networks to artificial networks on machine learning tasks. Adult brains come with a huge amount of supervised pre-training built in, giving them an unfair advantage, and infant brains are uncooperative in experiments, so we can not make clear comparisons to artificial networks about the efficiency of credit assignment.
6. Studies on primates [Hung et al., 2005] suggest the existence of a fast feedforward inference circuit in the brain, based on the fact that it is possibly to infer from neural recordings which object the eyes have seen after a short (as little as 12.5ms) delay, which would only leave time for a few feedforward synaptic transmissions.

asynchronous, nonstationary data, and it will make it easier to design scalable, modular, energy-efficient hardware on which to run neural networks.

10. Acknowledgements

I would like to thank Max Welling for supervising me throughout my PhD. Our weekly meetings were always a pleasant and insightful exploration of ideas, and his parsimonious thinking and insistence on clarity were very valuable. I was frequently amazed at Max's ability to save the state of his brain at the end of each meeting and reload it the following week as if no time had passed at all, as well as his willingness to entertain the notion that even the most half-baked of ideas thrown at his whiteboard just may have a grain of insight buried deep within - if only it could be extracted.

I would also like to thank my co-supervisor Efstratios Gavves for his spontaneity and stream-of-consciousness-style thinking which was the defacto format of our research meetings. It was always a pleasure to create and refine ideas with him - whether work-related, boat-related, or somewhere in between.

And of course, the QUVA lab, with whom I shared an office, AMLAB, with whom I shared a floor. Naming each of you independently would just be list-making, and naming only a few would be too exclusive, so I will name none of you at all. Just know that without the daily banter, humour, debates, frisbee, and occasional substantive discussions, I would have gone insane much more quickly.

I would like to thank Qualcomm for following me half way across the globe to fund my PhD, and the Dutch taxpayer for their willingness to fund open-ended research in the hope of attracting talent and industry to their beautiful country - I believe it pays off on average.

Finally, I thank my family for creating me and tolerating my absence, the friends I met along the way, and the Fietskliniek for adding purpose and structure to the dark Dutch winters.

11. Bibliography

David H Ackley, Geoffrey E Hinton, and Terrence J Sejnowski. A learning algorithm for boltzmann machines. *Cognitive science*, 9(1):147–169, 1985. 1.1.2, 2.1

Arash Ardakani, François Leduc-Primeau, Naoya Onizawa, Takahiro Hanyu, and Warren J Gross. Vlsi implementation of deep neural network using integral stochastic computing. *arXiv preprint arXiv:1509.08972*, 2015. 4.3

David Attwell and Simon B Laughlin. An energy budget for signaling in the grey matter of the brain. *Journal of Cerebral Blood Flow & Metabolism*, 21(10):1133–1145, 2001. 8.7

Yoshua Bengio, Thomas Mesnard, Asja Fischer, Saizheng Zhang, and Yuhai Wu. An objective function for stdp. *arXiv preprint arXiv:1509.05936*, 2015a. 5.5, 6.2, 6.3.3

Yoshua Bengio, Thomas Mesnard, Asja Fischer, Saizheng Zhang, and Yuhuai Wu. Stdp as presynaptic activity times rate of change of postsynaptic activity. *arXiv preprint arXiv:1509.05936*, 2015b. 1.4.1, 2.7.1, 6.1, 6.2, 8.6

Yoshua Bengio, Benjamin Scellier, Olexa Bilaniuk, Joao Sacramento, and Walter Senn. Feedforward initialization for fast inference of deep generative networks is biologically plausible. *arXiv preprint arXiv:1606.01651*, 2016. 7.5

Arjun Bharioke and Dmitri B Chklovskii. Automatic adaptation to fast input changes in a time-invariant neural circuit. *PLoS computational biology*, 11(8):e1004315, 2015. 8.4.5, 8.6, C.2

Guo-Qiang Bi and Huai-Xing Wang. Temporal asymmetry in spike timing-dependent synaptic plasticity. *Physiology & behavior*, 77(4-5):551–555, 2002. 1.4, 6.1

Jonathan Binas, Giacomo Indiveri, and Michael Pfeiffer. Deep counter networks for asynchronous event-based processing. *CoRR*, abs/1611.00710, 2016. URL http://arxiv.org/abs/1611.00710. 5.5

Sander M Bohte, Joost N Kok, and Johannes A La Poutré. Spikeprop: backpropagation for networks of spiking neurons. In *ESANN*, pages 419–424, 2000. 2.2, 3.3, 5.5

Lars Buesing, Johannes Bill, Bernhard Nessler, and Wolfgang Maass. Neural dynamics as sampling: a model for stochastic computation in recurrent networks of spiking neurons. *PLoS Comput Biol*, 7(11):e1002211, 2011. 3.3

James C Candy and Gabor C Temes. *Oversampling delta-sigma data converters: theory, design, and simulation.* University of Texas Press, 1962. 5.3.3, 8.4.4

Miguel Carreira-Perpinan and Weiran Wang. Distributed optimization of deeply nested systems. In *Artificial Intelligence and Statistics*, pages 10–19, 2014. 7.5.3

Andrew S Cassidy, Paul Merolla, John V Arthur, Steve K Esser, Bryan Jackson, Rodrigo Alvarez-Icaza, Pallab Datta, Jun Sawada, Theodore M Wong, Vitaly Feldman, et al. Cognitive computing building block: A versatile and efficient digital neuron model for neurosynaptic cores. In *Neural Networks (IJCNN), The 2013 International Joint Conference on*, pages 1–10. IEEE, 2013. 4.7

Vincent Chan, Shih-Chii Liu, and André Van Schaik. Aer ear: A matched silicon cochlea pair with address event representation interface. *Circuits and Systems I: Regular Papers, IEEE Transactions on*, 54(1):48–59, 2007. 3.2

Marie Chau and Michael C Fu. An overview of stochastic approximation. In *Handbook of Simulation Optimization*, pages 149–178. Springer, 2015. 8.6

Chun-Chuan Chen, RN Henson, Klaas E Stephan, James M Kilner, and Karl J Friston. Forward and backward connections in the brain: a dcm study of functional asymmetries. *Neuroimage*, 45(2):453–462, 2009. 9.1

Tian Qi Chen, Yulia Rubanova, Jesse Bettencourt, and David K Duvenaud. Neural ordinary differential equations. In *Advances in neural information processing systems*, pages 6571–6583, 2018. 2.4

Dmitri B Chklovskii and Daniel Soudry. Neuronal spike generation mechanism as an oversampling, noise-shaping a-to-d converter. In *Advances in Neural Information Processing Systems*, pages 503–511, 2012. 5.5

Tim Cooijmans and James Martens. On the variance of unbiased online recurrent optimization. *arXiv preprint arXiv:1902.02405*, 2019. 2.5

Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. Binaryconnect: Training deep neural networks with binary weights during propagations. *CoRR*, abs/1511.00363, 2015. URL http://arxiv.org/abs/1511.00363. 3.3, 5.5

Matthieu Courbariaux, Itay Hubara, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Binarized neural networks: Training deep neural networks with weights and activations constrained to+ 1 or-1. *arXiv preprint arXiv:1602.02830*, 2016. 4.3, 4.5.3, **??**

Koby Crammer, Alex Kulesza, and Mark Dredze. Adaptive regularization of weight vectors. In *Advances in neural information processing systems*, pages 414–422, 2009. 3.5.1

Chris Cremer, Xuechen Li, and David Duvenaud. Inference suboptimality in variational autoencoders. *arXiv preprint arXiv:1801.03558*, 2018. 7.5.2

Francis Crick. The recent excitement about neural networks. *Nature*, 337(6203):129–132, 1989. 8.2

Peter Dayan, Geoffrey E Hinton, Radford M Neal, and Richard S Zemel. The helmholtz machine. *Neural computation*, 7(5):889–904, 1995. 7.5.2

Arthur P Dempster, Nan M Laird, and Donald B Rubin. Maximum likelihood from incomplete data via the em algorithm. *Journal of the royal statistical society. Series B (methodological)*, pages 1–38, 1977. 7.5.2

Peter U Diehl, Daniel Neil, Jonathan Binas, Matthew Cook, Shih-Chii Liu, and Michael Pfeiffer. Fast-classifying, high-accuracy spiking deep networks through weight and threshold balancing. In *2015 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8. IEEE, 2015. 3.3, 4.3, 5.5

Andrew Du, Andrew M Zipkin, Kevin G Hatala, Elizabeth Renner, Jennifer L Baker, Serena Bianchi, Kallista H Bernal, and Bernard A Wood. Pattern and process in hominin brain size evolution are scale-dependent. *Proceedings of the Royal Society B: Biological Sciences*, 285(1873):20172738, 2018. 1.6

Steven K Esser, Paul A Merolla, John V Arthur, Andrew S Cassidy, Rathinakumar Appuswamy, Alexander Andreopoulos, David J Berg, Jeffrey L McKinstry, Timothy Melano, Davis R Barch, et al. Convolutional networks for fast, energy-efficient neuromorphic computing. *arXiv preprint arXiv:1603.08270*, 2016. 4.3

Xiequan Fan, Ion Grama, and Quansheng Liu. Hoeffding's inequality for supermartingales. *Stochastic Processes and their Applications*, 122(10):3545–3559, 2012. A.1.4

Chelsea Finn, Paul Christiano, Pieter Abbeel, and Sergey Levine. A connection between generative adversarial networks, inverse reinforcement learning, and energy-based models. *arXiv preprint arXiv:1611.03852*, 2016. 7.5.1

Rongjing Ge, Hao Qian, and Jin-Hui Wang. Physiological synaptic signals initiate sequential spikes at soma of cortical pyramidal neurons. *Molecular brain*, 4(1):19, 2011. 1.2

Abraham P George and Warren B Powell. Adaptive stepsizes for recursive estimation with applications in approximate dynamic programming. *Machine learning*, 65(1):167–198, 2006. 8.4.1, 8.4.3, 8.6, E.1

Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Aistats*, volume 9, pages 249–256, 2010. 2.1, 4.6.1

Xavier Glorot, Antoine Bordes, and Yoshua Bengio. Deep sparse rectifier neural networks. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, pages 315–323, 2011. 2.1

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016. 2.4

Donald Olding Hebb. *The organization of behavior: A neuropsychological theory*. Psychology Press, 2005. 2.1

Maciej Henneberg. Decrease of human skull size in the holocene. *Human biology*, pages 395–405, 1988. 1.6.2

Geoffrey Hinton. How to do backpropagation in a brain. In *Invited talk at the NIPS'2007 deep learning workshop*, volume 656, 2007. 6.3.1

Geoffrey Hinton. Neural networks for machine learning. coursera, video lectures. 2012. 2.7.1, 4.5.3

Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531*, 2015. 7.2, 7.5

Geoffrey E Hinton. Training products of experts by minimizing contrastive divergence. *Neural computation*, 14(8):1771–1800, 2002. 2.1

Geoffrey E Hinton, Simon Osindero, and Yee-Whye Teh. A fast learning algorithm for deep belief nets. *Neural computation*, 18(7):1527–1554, 2006. 2.1, 9.1

John J Hopfield. Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the national academy of sciences*, 79(8):2554–2558, 1982. 1.1.2, 2.1

John J Hopfield. Neurons with graded response have collective computational properties like those of two-state neurons. *Proceedings of the national academy of sciences*, 81(10): 3088–3092, 1984. 1.1.2, 2.7, 8.2, 8.3.1

Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5):359–366, 1989. 1.1

Mark Horowitz. 1.1 computing's energy problem (and what we can do about it). In *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, pages 10–14. IEEE, 2014. 4.4.3, 4.4, 5.5, 5.7, 5.4.2, 8.7, B.1, C.6

Chou P Hung, Gabriel Kreiman, Tomaso Poggio, and James J DiCarlo. Fast readout of object identity from macaque inferior temporal cortex. *Science*, 310(5749):863–866, 2005. 9.1

Eric Hunsberger and Chris Eliasmith. Spiking deep networks with lif neurons. *arXiv preprint arXiv:1510.08829*, 2015. 3.3

AI Impacts. Brain performance in teps, 2015. URL `https://aiimpacts.org/brain-performance-in-teps/`. 1.6.2

F Itakura. Speech analysis and synthesis systems based on statistical method. *Doctor of Engineering Dissertation, Dept. of Engineering, Nagoya University, Japan*, 51:562–574, 1972. 6.3.3, 6.3.4

Max Jaderberg, Wojciech Marian Czarnecki, Simon Osindero, Oriol Vinyals, Alex Graves, David Silver, and Koray Kavukcuoglu. Decoupled neural interfaces using synthetic gradients. *arXiv preprint arXiv:1608.05343*, 2016. 2.6, 2.6, 7.5.3, 9.1, **??**

Taesup Kim and Yoshua Bengio. Deep directed generative models with energy-based probability estimation. *arXiv preprint arXiv:1606.03439*, 2016. 7.5.1

Yoon Kim, Sam Wiseman, Andrew C Miller, David Sontag, and Alexander M Rush. Semi-amortized variational autoencoders. *arXiv preprint arXiv:1802.02550*, 2018. 7.5.2

Diederik P Kingma and Max Welling. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*, 2013. 2.1, 7.5.2

Durk P Kingma, Tim Salimans, and Max Welling. Variational dropout and the local reparameterization trick. In *Advances in Neural Information Processing Systems*, pages 2575–2583, 2015. 1.1.4

James Kirkpatrick, Razvan Pascanu, Neil Rabinowitz, Joel Veness, Guillaume Desjardins, Andrei A Rusu, Kieran Milan, John Quan, Tiago Ramalho, Agnieszka Grabska-Barwinska, et al. Overcoming catastrophic forgetting in neural networks. *Proceedings of the national academy of sciences*, page 201611835, 2017. 8.6, 9.1

Kristin Koch, Judith McLean, Ronen Segev, Michael A Freed, Michael J Berry, Vijay Balasubramanian, and Peter Sterling. How much the eye tells the brain. *Current Biology*, 16(14):1428–1434, 2006. 4.2

Adam A Kohan, Edward A Rietman, and Hava T Siegelmann. Error forward-propagation: Reusing feedforward connections to propagate errors in deep learning. *arXiv preprint arXiv:1808.03357*, 2018. 7.5.3

Karri P Lamsa, Joost H Heeroma, Peter Somogyi, Dmitri A Rusakov, and Dimitri M Kullmann. Anti-hebbian long-term potentiation in the hippocampal feedback inhibitory circuit. *Science*, 315(5816):1262–1266, 2007. 2.1

Dong-Hyun Lee, Saizheng Zhang, Asja Fischer, and Yoshua Bengio. Difference target propagation. In *Machine Learning and Knowledge Discovery in Databases*, pages 498–515. Springer, 2015. 2.3, 2.1, 7, 7.5.3, **??**

Jun Haeng Lee, Tobi Delbruck, and Michael Pfeiffer. Training deep spiking neural networks using backpropagation. *arXiv preprint arXiv:1608.08782*, 2016. 4.3, 5.5

Yingzhen Li, Richard E Turner, and Qiang Liu. Approximate inference with amortised mcmc. *arXiv preprint arXiv:1702.08343*, 2017. 7.5.2

Patrick Lichtsteiner, Christoph Posch, and Tobi Delbruck. A 128× 128 120 db 15 μs latency asynchronous temporal contrast vision sensor. *Solid-State Circuits, IEEE Journal of*, 43(2):566–576, 2008. 1.6.1, 3.2, 4.3, 4.4.1

Timothy P Lillicrap, Daniel Cownden, Douglas B Tweed, and Colin J Akerman. Random feedback weights support learning in deep neural networks. *arXiv preprint arXiv:1411.0247*, 2014. 7, 7.5.3, 8.6

Jacqueline Ling. The power of a human brain. https://hypertextbook.com/facts/2001/JacquelineLing.shtml, 2001. Accessed: 2018-12-10. 1.6.2, 8.2

Sindy Löwe, Peter O'Connor, and Bastiaan S Veeling. Greedy infomax for biologically plausible self-supervised representation learning. *arXiv preprint arXiv:1905.11786*, 2019. 9.1

Joseph Marino, Yisong Yue, and Stephan Mandt. Iterative amortized inference. *arXiv preprint arXiv:1807.09356*, 2018. 7.5.2

H Markram and B Sakmann. Action potentials propagating back into dendrites trigger changes in efficacy of single-axon synapses between layer v pyramidal neurons. In *Soc. Neurosci. Abstr*, volume 21, page 2007, 1995. 6.3.2, 9.1

Henry Markram, Wulfram Gerstner, and Per Jesper Sjöström. Spike-timing-dependent plasticity: a comprehensive overview. *Frontiers in synaptic neuroscience*, 4, 2012. 1.4.1, 5.3.7, 5.3.7, 5.4

Thomas Mesnard, Wulfram Gerstner, and Johanni Brea. Towards deep learning with spiking neurons in energy based models with contrastive hebbian plasticity. *arXiv preprint arXiv:1612.03214*, 2016. 8.6

Maad Mijwel. Pattern recognition and neural networks, 01 2017. 1.1

Mehdi Mirza and Simon Osindero. Conditional generative adversarial nets. *arXiv preprint arXiv:1411.1784*, 2014. 7.2

Emre Neftci, Srinjoy Das, Bruno Pedroni, Kenneth Kreutz-Delgado, and Gert Cauwenberghs. Event-driven contrastive divergence for spiking neuromorphic systems. *Frontiers in neuroscience*, 7, 2013. 3.3

Cuong V Nguyen, Yingzhen Li, Thang D Bui, and Richard E Turner. Variational continual learning. *arXiv preprint arXiv:1710.10628*, 2017. 9.1, **??**

Peter O'Connor and Max Welling. Deep spiking networks. *arXiv preprint arXiv:1602.08323*, 2016a. 4.3, 4.5, 5.5, B.1

Peter O'Connor and Max Welling. Sigma delta quantized networks. *arXiv preprint arXiv:1611.02024*, 2016b. 5.2, 5.3.3, 5.5

Peter O'Connor, Daniel Neil, Shih-Chii Liu, Tobi Delbruck, and Michael Pfeiffer. Real-time classification and sensor fusion with a spiking deep belief network. *Frontiers in neuroscience*, 7, 2013. 3.3

Peter O'Connor, Efstratios Gavves, and Max Welling. Temporally efficient deep learning with spikes. *arXiv preprint arXiv:1706.04159*, 2017. 8.4.4, 8.4.5

Yann Ollivier, Corentin Tallec, and Guillaume Charpiat. Training recurrent networks online without backtracking. *arXiv preprint arXiv:1507.07680*, 2015. 8.6

Aaron van den Oord, Sander Dieleman, Heiga Zen, Karen Simonyan, Oriol Vinyals, Alex Graves, Nal Kalchbrenner, Andrew Senior, and Koray Kavukcuoglu. Wavenet: A generative model for raw audio. *arXiv preprint arXiv:1609.03499*, 2016. 1.1.1

Aaron van den Oord, Yazhe Li, and Oriol Vinyals. Representation learning with contrastive predictive coding. *arXiv preprint arXiv:1807.03748*, 2018. 1, 1

Rajesh PN Rao and Dana H Ballard. Predictive coding in the visual cortex: a functional interpretation of some extra-classical receptive-field effects. *Nature neuroscience*, 2(1): 79, 1999. 1, 9.1, 1

Esteban Real, Jonathon Shlens, Stefano Mazzocchi, Xin Pan, and Vincent Vanhoucke. Youtube-boundingboxes: A large high-precision human-annotated data set for object detection in video. *arXiv preprint arXiv:1702.00824*, 2017. 5.4.2

Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 779–788, 2016. 1.6.2, 8.2

Herbert Robbins and Sutton Monro. A stochastic approximation method. *The annals of mathematical statistics*, pages 400–407, 1951. 8.4.1

Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015. doi: 10.1007/s11263-015-0816-y. 4.6.3

Ruslan Salakhutdinov and Hugo Larochelle. Efficient learning of deep boltzmann machines. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 693–700, 2010. 7.5.2

Benjamin Scellier and Yoshua Bengio. Equilibrium propagation: Bridging the gap between energy-based models and backpropagation. *Frontiers in computational neuroscience*, 11:24, 2017. 2.7, 7, 7.1, 7.2, 7.3.1, 7.4, 7.6, 8.1, 8.2, 8.3.1, 8.5.2, 8.3, 8.6, 9.1, **??**, D.4

Benjamin Scellier, Anirudh Goyal, Jonathan Binas, Thomas Mesnard, and Yoshua Bengio. Extending the framework of equilibrium propagation to general dynamics, 2018. URL `https://openreview.net/forum?id=SJTB5GZCb`. 1, 2.7.1, 6.3.1, 7.5.3, 8.6, 9.1, 9.1

Jonghan Shin. Adaptive noise shaping neural spike encoding and decoding. *Neurocomputing*, 38:369–381, 2001. 5.5, 8.4.5

Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014. 4.6.3, 5.4.2, C.8

P Smolensky. Chapter 6: information processing in dynamical systems: foundations of harmony theory. *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, 1, 1986. 2.1

Mandyam V Srinivasan, Simon B Laughlin, and Andreas Dubs. Predictive coding: a fresh view of inhibition in the retina. *Proceedings of the Royal Society of London B: Biological Sciences*, 216(1205):427–459, 1982. 5.5, 8.4.5, C.2

Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1):1929–1958, 2014. 1.1.4

Greg J Stuart and Bert Sakmann. Active propagation of somatic action potentials into neocortical pyramidal cell dendrites. *Nature*, 367(6458):69–72, 1994. 1.5, 6.5

Corentin Tallec and Yann Ollivier. Unbiased online recurrent optimization. *arXiv preprint arXiv:1702.05043*, 2017. 2.5, 8.6, 9.1, **??**

Gavin Taylor, Ryan Burmeister, Zheng Xu, Bharat Singh, Ankit Patel, and Tom Goldstein. Training neural networks without gradients: A scalable admm approach. In *International Conference on Machine Learning*, pages 2722–2731, 2016. 7.5.3

S Tewksbury and RW Hallock. Oversampled, linear predictive and noise-shaping coders of order n$>$ 1. *IEEE Transactions on Circuits and Systems*, 25(7):436–447, 1978. 8.4.5

Jack Waters and Fritjof Helmchen. Boosting of action potential backpropagation by neocortical network activity in vivo. *Journal of Neuroscience*, 24(49):11127–11136, 2004. 6.5

Konstantin Weil. Action potential, 2017. URL https://flexikon.doccheck.com/en/Action_potential. 1.3

Max Welling. Herding dynamical weights to learn. In *Proceedings of the 26th Annual International Conference on Machine Learning*, pages 1121–1128. ACM, 2009. 3.4.1, B.1

Ronald J Williams and David Zipser. A learning algorithm for continually running fully recurrent neural networks. *Neural computation*, 1(2):270–280, 1989. 2.5

Laurenz Wiskott. Learning invariance manifolds. *Neurocomputing*, 26:925–932, 1999. 5.6

Xiaohui Xie and H Sebastian Seung. Spike-based learning rules and stabilization of persistent neural activity. In *Advances in neural information processing systems*, pages 199–208, 2000. 2.7.1, 6.2, 6.3.3

Young C Yoon. Lif and simplified srm neurons encode signals into spikes via a form of asynchronous pulse sigma-delta modulation. 2016. 4.3, 8.4.4

Young C Yoon. Lif and simplified srm neurons encode signals into spikes via a form of asynchronous pulse sigma–delta modulation. *IEEE transactions on neural networks and learning systems*, 28(5):1192–1205, 2017. 5.5

Davide Zambrano and Sander M Bohte. Fast and efficient asynchronous neural computation with adapting spiking neural networks. *arXiv preprint arXiv:1609.02053*, 2016. 4.3, 5.5, 8.4.4

Shuangfei Zhai, Yu Cheng, Rogerio Feris, and Zhongfei Zhang. Generative adversarial networks as variational training of energy based models. *arXiv preprint arXiv:1611.01799*, 2016. 7.5.1

12. List of Publications

The following publications were made in the course of this research. For all of these works, the first author was the primary contributor.

1. Peter O'Connor, Efstratios Gavves, Max Welling. *Sigma Delta Quantized Networks.* Published in ICLR 2017.

2. Peter O'Connor, Efstratios Gavves, Matthias Reisser, Max Welling. *Temporally Efficient Deep Learning with Spikes.* Published in ICLR 2018.

3. Peter O'Connor, Efstratios Gavves, Max Welling. *Initialized Equilibrium Propagation.* Published in ICLR 2019.

4. Peter O'Connor, Efstratios Gavves, Max Welling. *Spiking Equilibrium Propagation.* Published in AISTATS 2019.

13. Appendix

Appendix is available in the online version of this thesis at: github.com/petered/data/blob/master/thesis.pdf or bitly.ws/8Man

A. Deep Spiking Networks

A.1. Algorithms

A.1.1. Proof of convergence of Spike-Vector Quantization

Here we show that if we obtain events $\langle (i_n, s_n) : n \in [1..N] \rangle$ given a vector $\vec{v}$ and a time T from the Spiking-Vector Quantization Algorithm then:

$$\lim_{T \to \infty} \vec{v} = \frac{1}{T} \sum_{n=1}^{N} \vec{e_{i_n}} s_n \tag{A.1}$$

Algorithm 9 Spiking Vector Quantization

```
Input: $\vec{v} \in \mathbb{R}^d$, $T \in \mathbb{N}$
Internal: $\vec{\phi} \in \mathbb{R}^d \leftarrow \vec{0}$
for t $\in$ 1...$T$ do
    $\vec{\phi} \leftarrow \vec{\phi} + \vec{v}$
    while $max(|\vec{\phi}|) > \frac{1}{2}$ do
        $i \leftarrow argmax(|\vec{\phi}|)$
        $s \leftarrow sign(\phi_i)$
        $\vec{\phi_i} \leftarrow \vec{\phi_i} - s$
        FireSpike(source = i, sign = s)
    end while
end for
```

Since $\forall i : -\frac{1}{2} < \phi_i < \frac{1}{2}$ the L1 norm is bounded by:

$$\|\phi_T\|_{L1} = \| \sum_{t=1}^{T} v - \sum_{n=1}^{N} e_{i_n} s_n \|_{L1} < \frac{l(\vec{v})}{2} \tag{A.2}$$

where $l(\vec{v})$ is the number of elements in vector $\vec{v}$. We can take the limit of infinite time, and show that our spikes converge to form an approximation of $\vec{v}$:

$$\begin{aligned} &\lim_{T\to\infty} : \frac{1}{T}\|\phi_T\|_{L1} = \lim_{T\to\infty} : \|\frac{1}{T}\sum_{t=1}^{T} v - \frac{1}{T}\sum_{n=1}^{N} e_{\vec{i_n}} s_n\|_{L1} = 0 \\ &\lim_{T\to\infty} : \vec{v} = \frac{1}{T}\sum_{n=1}^{N} e_{\vec{i_n}} s_n \end{aligned} \tag{A.3}$$

A.1.2. Stochastic Sampling

Algorithm 10 Stochastic Sampling of events from a vector

```
Input: vector $v$, int $T$
$mag \leftarrow sum(abs(\vec{v}))$
$\vec{p} = abs(\vec{v})/mag$
for $t \in [1...T]$ do
    $N = poisson(mag)$
    for $n \in [1...N]$ do
        $i \leftarrow DrawSample(\vec{p})$
        $s \leftarrow sign(v_i)$
        FireSignedSpike(index = i, sign = s)
    end for
end for
```

A.1.3. Spiking Stream Quantization

In our modification to Spiking Vector Quantization, we instead feed in a stream of vectors, as in Algorithm 11.

Algorithm 11 Spiking Stream Quantization

```
Input: $\vec{v_t} \in \mathbb{R}^d$, $t \in [1..T]$
Internal: $\vec{\phi} \in \mathbb{R}^d \leftarrow \vec{0}$
for t $\in$ 1...$T$ do
    $\vec{\phi} \leftarrow \vec{\phi} + \vec{v_t}$
    while $max(|\vec{\phi}|) > \frac{1}{2}$ do
        $i \leftarrow argmax(|\vec{\phi}|)$
        $s \leftarrow sign(\phi_i)$
        $\vec{\phi_i} \leftarrow \vec{\phi_i} - s$
        FireSpike(source = i, sign = s)
    end while
end for
```

If we simply replace the term $\sum_{t=1}^{T} \vec{v}$ in Equation A.3 with $\sum_{t=1}^{T} \vec{v_t}$, and follow the same reasoning, we find that we converge to the running mean of the vector-stream.

$$\begin{aligned}
&\lim_{T\to\infty} : \frac{1}{T}\|\phi_T\|_{L1} = \lim_{T\to\infty} : \|\frac{1}{T}\sum_{t=1}^{T} v_t - \frac{1}{T}\sum_{n=1}^{N} \vec{e_{i_n}} s_n\|_{L1} = 0 \\
&\lim_{T\to\infty} : \frac{1}{T}\sum_{t=1}^{T} \vec{v} = \frac{1}{T}\sum_{n=1}^{N} \vec{e_{i_n}} s_n
\end{aligned} \tag{A.4}$$

A.1.4. Rectified Stream Quantization

We can further make a small modification where we only send positive spikes (so our $\vec{\phi}$ can get unboundedly negative.

Algorithm 12 Rectified Spiking Stream Quantization

```
Input: $\vec{v_t} \in \mathbb{R}^d$, $t \in [1..T]$
Internal: $\vec{\phi} \in \mathbb{R}^d \leftarrow \vec{0}$
for t ∈ 1...$T$ do
    $\vec{\phi} \leftarrow \vec{\phi} + \vec{v_t}$
    while $max(\vec{\phi}) > \frac{1}{2}$ do
        $i \leftarrow argmax(\vec{\phi})$
        $\vec{\phi_i} \leftarrow \vec{\phi_i} - 1$
        FireSpike(source = i, sign = +1)
    end while
end for
```

To see why this construction approximates a ReLU unit, first observe that the total number of spikes emitted can be computed by considering the total cumulative sum $\sum_{t=1}^{T} v_{j,t}$. More precisely:

$$N_{j,T} = \max\big(0, \lfloor \max_{T' \in [1...T]} \big(\sum_{t=1}^{T'} v_{j,t} + \frac{1}{2}\big)\rfloor\big) \tag{A.5}$$

where $N_{j,T}$ indicates the number of spikes emitted from unit j by time T and $\lfloor\cdot\rfloor$ indicates the integer floor of a real number.

Assume the $v_{j,t}$ are IID sampled from some process with mean $E[v_{j,t}] = \mu_j$ and finite standard deviation σ_j. Define $\zeta_{j,t} = v_{j,t} - \mu_j$ which has zero mean and the cumulative sum $\xi_{j,T} = \sum_{t=1}^{T} \zeta_{j,t}$ which is martingale. There are a number of concentration inequalities, such as the Bernstein concentration inequalities Fan et al. [2012] that bound the sum or the maximum of the sequence $\xi_{j,T}$ under various conditions. What is only important for us is the fact that in the limit $T \to \infty$ the sums $\xi_{j,T}$ concentrate to a delta peak at zero in probability and that we can therefore conclude $\sum_{t=1}^{T} v_{j,t} \to T\mu_j$ from which we can also conclude that the maximum, and thus the number of spikes will grow in the same way. From this we finally conclude that $\frac{N_{j,T}}{T} \to \max(0, \mu)$, which is the ReLU non-linearity. Thus the mean spiking rate approaches the ReLU function of the mean input.

A.2. MLP Convergence

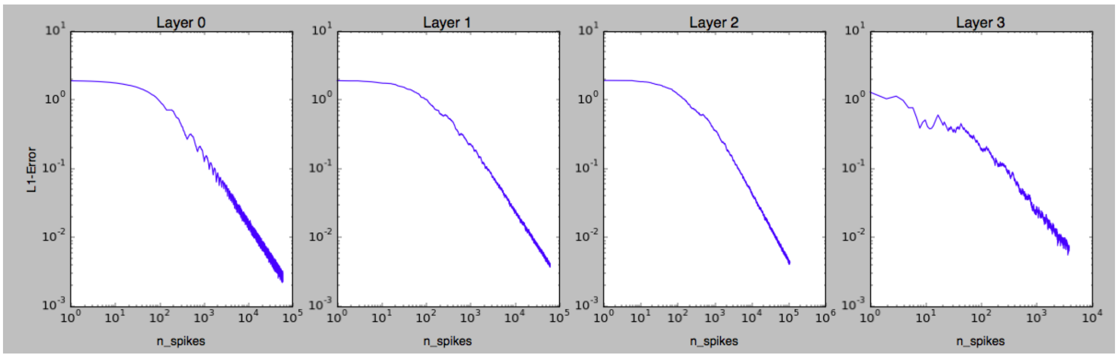


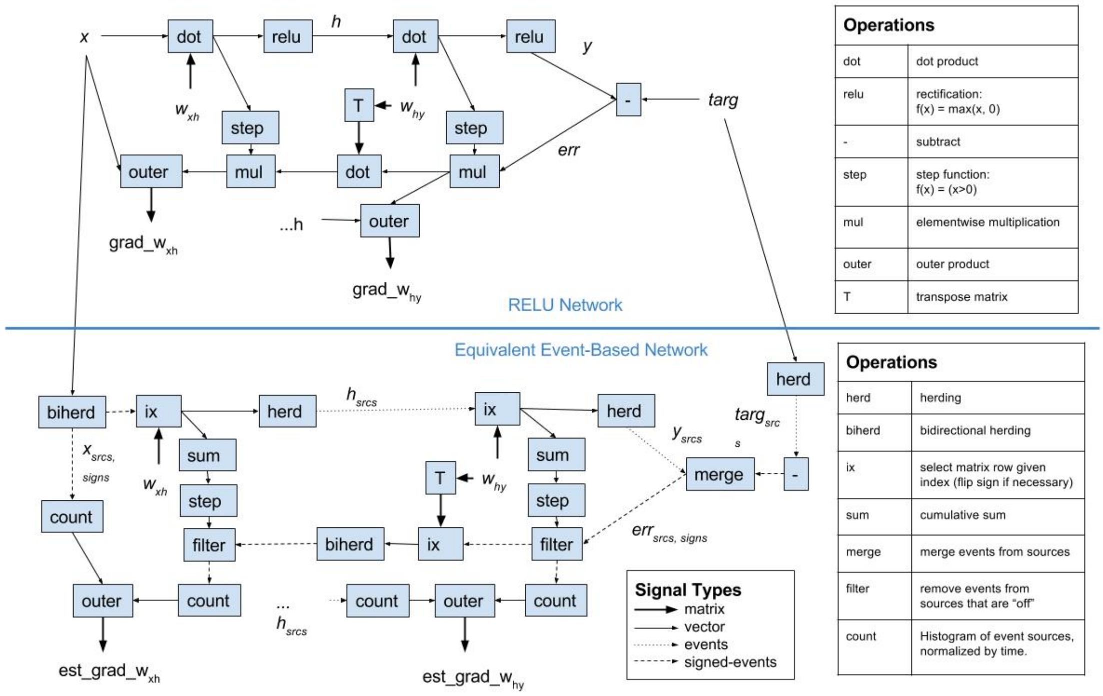
Figure A.1.: A 3-layer MLP (784-500-500-10) MLP with random weights ($\sim \mathcal{N}(0, 0.1)$) is fed with a random input vector, and a forward pass is computed. We compare the response of the ReLU network to the counts of spikes from our spiking network, and see that over all layers, the responses converge as $T \to \infty$. Note both x and y axes are log-scaled.

A.3. A Training Iteration

Algorithm 13 Pseudocode for a single Training Iteration on a network with one hidden layer with Fractional Stochastic Gradient Descent

```
procedure TrainingIteration($\vec{x} \in \mathbb{R}^{d_{in}}$, $\vec{y} \in \mathbb{R}^{d_{out}}$, $T \in \mathbb{N}$, $\eta \in \mathbb{R}$)
    Variables: $\vec{u} \in \mathbb{R}^{d_{out}} \leftarrow \vec{0}$, $W_{hid} \in \mathbb{R}^{d_{in} \times d_{hid}}$, $W_{out} \in \mathbb{R}^{d_{hid} \times d_{out}}$
    for t $\in$ 1...$T$ do
        InputLayer.forward($\vec{x}$)
        OutputLayer.backward($\vec{u} - \vec{y}$)
    end for
    procedure InputLayer($\vec{v} \in \mathbb{R}^{d_{in}}$)
        Internal: $\vec{\phi} \in \mathbb{R}^{d_{in}}$
        $\vec{\phi} \Leftarrow \vec{\phi} + \vec{v}$
        while $max(|\vec{\phi}|) > \frac{1}{2}$ do
            $i \leftarrow argmax(|\vec{\phi}|)$
            s = $sign(\phi_i)$
            $\phi_i \leftarrow \phi_i - s$
            HiddenLayer(i, s)
        end while
    end procedure
     Object: HiddenLayer
        Internal:  $\vec{c_{in}} \in \mathbb{R}^{d_{in}} \leftarrow \vec{0}$, $\vec{c_{out}} \in \mathbb{R}^{d_{hid}} \leftarrow \vec{0}$
        procedure Forward($i \in [1..d_{in}]$, $s \in [-1, +1]$)
            Internal: $\vec{\phi} \in \mathbb{R}^{d_{hid}}$
            $c_{in_i} \leftarrow c_{in_i} + s$
            $\vec{\phi} \Leftarrow \vec{\phi} + s \cdot [W_{hid}]_{i,\cdot}$
            $\vec{c_{out}} \leftarrow \vec{c_{out}} + [W_{hid}]_{i,\cdot}$
            while $max(\vec{\phi}) > \frac{1}{2}$ do
                $i \leftarrow argmax(\vec{\phi})$
                $\phi_i \leftarrow \phi_i - 1$
                OutputLayer(i, +1)
            end while
        end procedure
        procedure Backward($\vec{v} \in \mathbb{R}^{d_{hid}}$)
            Internal: $\vec{\phi} \in \mathbb{R}^{d_{hid}}$
            $\vec{\phi} \Leftarrow \vec{\phi} + \vec{v} \odot [\vec{c_{out}} > 0]$
            while $max(|\vec{\phi}|) > \frac{1}{2}$ do
                $j \leftarrow argmax(|\vec{\phi}|)$
                s = $sign(\phi_i)$
                $\phi_j \leftarrow \phi_j - s$
                $[W_{hid}]_{\cdot,j} \leftarrow [W_{hid}]_{\cdot,j} - \eta \cdot s \cdot \vec{c_{in}}$        ▷ Update to $W_{hid}$
            end while
        end procedure
     Object: OutputLayer
        Internal:  $\vec{c_{in}} \in \mathbb{R}^{d_{hid}} \leftarrow \vec{0}$
        procedure Forward($i \in [1..d_{hid}]$, $s \in [-1, +1]$)
            $c_{in_i} \leftarrow c_{in_i} + s$
            Global: $\vec{u} \leftarrow \vec{u} + s \cdot [W_{out}]_{i,\cdot}$        ▷ Update to $\vec{u}$
        end procedure
        procedure Backward($v \in \mathbb{R}^{d_{out}}$)
            Internal: $\vec{\phi} \in \mathbb{R}^{d_{out}}$
            $\vec{\phi} \Leftarrow \vec{\phi} + \vec{v}$
            while $max(|\vec{\phi}|) > \frac{1}{2}$ do
```

A.4. Network Diagram



Operations	
dot	dot product
relu	rectification: f(x) = max(x, 0)
-	subtract
step	step function: f(x) = (x>0)
mul	elementwise multiplication
outer	outer product
T	transpose matrix

Operations	
herd	herding
biherd	bidirectional herding
ix	select matrix row given index (flip sign if necessary)
sum	cumulative sum
merge	merge events from sources
filter	remove events from sources that are "off"
count	Histogram of event sources, normalized by time.

Figure A.2.: The architecture of the Spiking MLP. On the top, we have a conventional neural network of rectified linear units with one hidden layer. On the bottom, we have the equivalent spiking network.

A.5. Hyperparameters

Our spiking architecture introduced a number of new hyperparameters and settings that are unfamiliar with those used to regular neural networks. We chose to evaluate these empirically by modifying them one-by-one as compared to a baseline.

- Fractional Updates.
  - False (Baseline): We use the standard stochastic-gradient descent method
  - True: We use our new Fractional Stochastic Gradient Descent method - described in section 3.4.7
- Depth-First
  - False (Baseline): Events are propagated "Breadth-first", meaning that, at a given time-step, all events are collected from the output of one module before any of their child-events are processed.

  - True: If an event from module A creates child-events from module B, those are processed immediately, before any more events from module A are processed.
- Smooth Weight Updates
  - False (Baseline): The weight-update modules take in a count of spikes from the previous layer as their input.
  - True: The weight-update modules take the rectified cumulative sum of the pre-quantized vectors from the previous layer - resulting in a smoother estimate of the input.
- Backwards-Quantization:
  - No-Reset-Quantization (Baseline): The backwards quantization modules do not reset their $\vec{\phi}$s with each training iteration.
  - Random: Each element of $\vec{\phi}$ is randomly selection from the interval $[-\frac{1}{2}, \frac{1}{2}]$ at the start of each training iteration.
  - Zero-Reset: The backwards quantizers reset their $\vec{\phi}$s to zero at the start of each training iteration.
- Number of time-steps: How many time steps to run the training procedure for each sample (Baseline is 10).

Since none of these hyperparameters have obvious values, we tested them empirically with a network with layer sizes [784-200-200-10], trained on MNIST. Table A.1 shows the affects of these hyperparameters.

Table A.1.: Percent error on MNIST for various settings. We explore the effects of different network settings by changing one at a time, training on MNIST, and comparing the result to a baseline network. The baseline network has layer-sizes [784, 200, 200, 10], uses regular (non-fractional) stochastic gradient descent, uses Breadth-First (as opposed to Depth-First) event ordering, does not use smooth weight updates, uses the "No-reset" scheme for its backward pass quantizers, and runs for 10 time-steps on each iteration.

Variant	**% Error**
Baseline	3.38
Fractional Updates	3.10
Depth-First Propagation	81.47
Smooth Gradients	2.85
Smooth & Fractional	3.07
Back-Quantization = Zero-Reset	87.87
Back-Quantization = Random	3.15
5 Time Steps	4.41
20 Time Steps	2.65

Most of the Hyperparameter settings appear to make a small difference. A noteable exception is the Zero-Reset rule for our backwards-quantizing units - the network learns almost nothing throughout training. The reason for this is that the initial weights, which were drawn from $\mathcal{N}(0, 0.01)$ are too small to allow any error-spikes to be sent back (the backward-pass quantizers never reach their firing thresholds). As a result, the network fails to learn. We found two ways to deal with this: "Back-Quantization = Random" initializes the $\vec{\phi}$ for the backwards quantizers randomly at the beginning of each round of training. "Back-Quantization = No-Reset" simply does not reset $\vec{\phi}$ in between training iterations. In both cases, the backwards pass quantizers always have some chance at sending a spike, and so the network is able to train. It is also interesting that using Fractional Updates (FSGD) gives us a slight advantage over regular SGD (Baseline). This is quite promising, because it means we have no need for multiplication in our network - As Section 3.4.7 explains, we simply add a column to the weight matrix every time an error spike arrives. We also observe that using the rectified running sum of the pre-quantization vector from the previous layer as our input to the weight-update module (Smooth Gradients) gives us a slight advantage. This is expected, because it is simply a less noisy version of the count of the input spikes that we would use otherwise.

A.6. Event Routing

Since each event can result in a variable number of downstream events, we have to think about the order in which we want to process these events. There are two issues:

1. In situations where one event is sent to multiple modules, we need to ensure that it is being sent to its downstream modules in the right order. In the case of the SMLP, we need to ensure that, for a given input, its child-events reach the filters in the backward pass before its other child-events make their way around and do the backward pass. Otherwise we are not implementing backpropagation correctly.

2. In situations where one event results in multiple child-events, we need to decide in which order to process these child events and their child events. For this, there are two routing schemes that we can use: Breadth-first and depth-first. We will outline those with the example shown in Figure A.3. Here we have a module A that responds to some input event by generating two events: a_1 and a_2. Event a_1 is sent to module B and triggers events b_1 and b_2. Event a_2 is sent and triggers event b_3. Table A.2 shows how a breadth-first vs depth-first router will handle these events.

Breadth-First	Depth-First
$B(a_1)$	$B(a_1)$
$B(a_2)$	$D(b_1)$
$C(a_1)$	$D(b_2)$
$C(a_2)$	$C(a_1)$
$D(b_1)$	$B(a_2)$
$D(b_2)$	$D(b_3)$
$D(b_3)$	$C(a_2)$

Table A.2.: Depth-First and Breadth-First routing differ in their order of event processing. This table shows the order of event processed in each scheme. Refer to A.3.

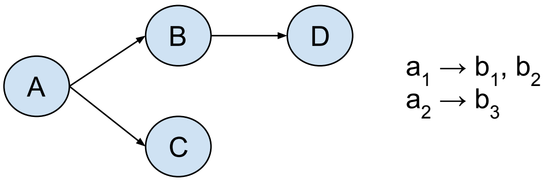


Figure A.3.: A simple graph showing 4 modules. Module A generates an event a_1 that causes two events: b_1 and b_2. These are then distributed to downstream modules. The order in which events are processed depends on the routing scheme.

Experimentally, we found that Breadth-First routing performed better on our MNIST task, but we should keep an open mind on both methods until we understand why.

B. Sigma Delta Quantized Networks

B.1. Delta-Herding Proof

Algorithm 14 Herding

```
**Internal:** $\vec{\phi} \in \mathbb{R}^d \leftarrow \vec{0}$
**Input:** $\vec{x_t} \in \mathbb{R}^d$
$\vec{\phi} \leftarrow \vec{\phi} + \vec{x}_t$
$\vec{s} \leftarrow round(\vec{\phi})$
$\vec{\phi} \leftarrow \vec{\phi} - \vec{s}$
**Return:** $\vec{s} \in \mathbb{I}^d$
```

Algorithm 15 Delta-Herding

```
**Internal:** $\vec{s}_{last} \in \mathbb{I}^d \leftarrow \vec{0}$
**Input:** $\vec{x_t} \in \mathbb{R}^d$
$\vec{s} \leftarrow round(x_t)$
$\Delta\vec{s} \leftarrow \vec{s} - \vec{s}_{last}$
$\vec{s}_{last} \leftarrow \vec{s}$
**Return:** $\Delta\vec{s} \in \mathbb{I}^d$
```

In previous work [O'Connor and Welling, 2016a], we used a quantization scheme which we refer to as *herding* for brevity and because of its relation to the deterministic sampling scheme in [Welling, 2009], but could otherwise be called Discrete-Time Bidirectional Sigma-Delta Modulation. The procedure is described in Algorithm 14. The input is summed into a potential ϕ over time until crossing a quantization threshold (in this case the $\pm\frac{1}{2}$ at which the round function changes value), and then resets.

Here we prove that Algorithm 15 is equivalent to applying Algorithm 14 to the output of a temporal difference modules. i.e. $herd(\Delta_t(x_t)) = \Delta_T(round(x_t))\forall t$.

First start by observing the following equivalence:

$$b = round(a) \Leftrightarrow |a - b| < \frac{1}{2} : b \in \mathbb{I} \tag{B.1}$$

We can apply this to the update rule in Algorithm 14:

$$\begin{aligned} s_t &= round(\phi_{t-1} + x_t) \in \mathbb{I} \\ \phi_t &= (\phi_{t-1} + x_t) - s_t \end{aligned} \tag{B.2}$$

$$\Rightarrow |\phi_t| < \frac{1}{2} \tag{B.3}$$

Now, if we unroll the two Equations B.2 over time, with initial condition $\phi_0 = 0$, we see that.

$$\phi_t = \sum_{\tau=1}^{t} x_\tau - \sum_{\tau=1}^{t} s_\tau : \phi_t \in \mathbb{R}, x_\tau \in \mathbb{R}, s_\tau \in \mathbb{I} \tag{B.4}$$

Using Equations B.3 and B.1, respectively, we can say:

$$|\phi| = \left| \sum_{\tau=1}^{t} x_t - \sum_{\tau=1}^{t} s_\tau \right| < \frac{1}{2} \tag{B.5}$$

$$\Rightarrow \sum_{\tau=1}^{t} s_\tau = round\left(\sum_{\tau=1}^{t} x_\tau \right) \tag{B.6}$$

Which can be rearranged to solve for s_t.

$$s_t = round\left(\sum_{\tau=0}^{t} x_\tau \right) - round\left(\sum_{\tau=0}^{t-1} x_\tau \right) \tag{B.7}$$

Now if we receive inputs from a Δ_T unit: $x_\tau = u_\tau - u_{\tau-1}$ with initial condition $u_0 = 0$, then:

$$\vec{s}_t = round\left(\sum_{\tau=0}^{t} (u_\tau - u_{\tau-1}) \right) - round\left(\sum_{\tau=0}^{t-1} (u_\tau - u_{\tau-1}) \right) \tag{B.8}$$

$$= round\,(u_t) - round\,(u_{t-1}) \tag{B.9}$$

$$= \Delta_T(round(u_t)) \tag{B.10}$$

Leaving us with the Delta-Herding algorithm (Algorithm 15).

Therefore, if we have a linear function $w(x)$, and make use of Equation 4.1, then we can see that the following is true:

$$\Sigma_T(w(herd(\Delta_T(x)))) = \Sigma_T(w(\Delta_T(round(x)))) = w(\Sigma_T(\Delta_T(round(x)))) = w(round(x)) \tag{B.11}$$

B.2. Calculating Flops

When computing the number of operations required for a forward pass, we only account for the matrix-products/convolutions (which form the bulk of computation in done by a neural network), and not hidden layer activations.

We compute the number of operations required for a forward pass of a fully connected network as follows:

For the non-discretized network, the number of flops for a single forward pass of a single data point through the network, the flop count is:

$$nFlops_{dense} = \sum_{l=0}^{L-1} (d_l \cdot d_{l+1} + (d_l - 1) \cdot d_{l+1} + d_{l+1}) = 2 \sum_{l=0}^{L-1} d_l \cdot d_{l+1} \tag{B.12}$$

Where d_l is the dimensionality of layer l (with $l = 0$ indicating the input layer). The first term counts the number of multiplications, the second the number of additions for reducing the dot-product, and the third the addition of the bias.

It can be argued that this is an unfair way to count the number of computations done by the non-discretized network because of the sparsity of the input layer (due to the zero-background of datasets like MNIST) and the hidden layers (due to ReLU units). Thus we also compute the number of operations for the non-discretized network when factoring in sparsity. The equation is:

$$\begin{aligned} nFlops_{sparse} &= \sum_{l=0}^{L-1} \left(\sum_{i=0}^{N_l} ([a_l]_i \neq 0) \cdot d_{l+1} + \left(\sum_{i=0}^{N_l} ([a_l]_i \neq 0) - 1 \right) \cdot d_{l+1} + d_{l+1} \right) \\ &= 2 \sum_{l=0}^{L-1} \sum_{i=0}^{N_l} ([a_l]_i \neq 0) \cdot d_{l+1} \end{aligned} \tag{B.13}$$

Where a_l are the layer activations N_l is the number of units in layer l and $([a_l]_i \neq 0)$ is 1 if unit i in layer l has nonzero activation and 0 otherwise.

For the rounding networks, we count the total absolute value of the discrete activations.

$$nFlops_{Round} = \sum_{l=0}^{L-1} \left(\sum_{i=0}^{N_l} |[s_l]_i| \cdot d_{l+1} + d_{l+1} \right) \tag{B.14}$$

Where s_l is the discrete activations of layer l. This corresponds to the number of operations that would be required for doing a dot product with the "sequential addition" method described in Section 4.4.2.

Finally, the Sigma-Delta network required slightly fewer flops, because the bias only need to be added once (at the beginning), so its cost is amortized.

$$nFlops_{\Sigma\Delta} = \sum_{l=0}^{L-1} \sum_{i=0}^{N_l} |[s_l]_i| \cdot d_{l+1} \tag{B.15}$$

B.3. Baking the scales into the parameters

In Section 4.5.1, we mention that we can "bake the scales into the parameters" for ReLU networks. Here we explain that statement.

Suppose you have a function

$$f(x) = k_2 \cdot h\left(x \cdot \frac{w}{k_1}\right)$$

If our nonlinearity h is homogeneous (i.e. $k \cdot h(x) = h(k \cdot x)$), as is the case for $relu(x) = max(0, x)$, we can collapse the scales k into the parameters:

$$\begin{aligned} f(x) &= k_2 \cdot relu(x \cdot w/k_1 + b) && \text{(B.16)} \\ &= relu\left(x \cdot w \cdot k_2/k_1 + k_2 \cdot b\right) && \text{(B.17)} \end{aligned}$$

So that after training scales, for a given network, we can simply incorporate them into the parameters, as: $w' = w \cdot k_2/k_1$, and $b' = k_2 \cdot b$.

B.4. Temporal MNIST

The Temporal MNIST dataset is a version of MNIST that is reshuffled so that similar frames end up being nearby. We generate this by iterating through the dataset, keeping a fixed-size buffer of candidates for the next frame. On every iteration, we compare all the candidates to the current frame, and select the closest one. The place that this winning candidate occupied in the buffer is then filled by a new sample from the dataset, and the winning candidate becomes the current frame. The process is repeated until we've sorted though all frames in the dataset. Code for generating the dataset can be found at: https://github.com/petered/sigma-delta/blob/master/sigma_delta/temporal_mnist.py

B.5. MNIST Results Table

		Mnist			Temp mnist		
Setting	Net Type	KFlops Test (ds\sp)	Class error (tr\ts)	Int32-Energy (nJ)	KFlops Test (ds\sp)	Class error (tr\ts)	Int32-
==========	==========	==========	==========	==========	==========	==========	====
Unoptimized	Original	397 \ 107	0.024 \ 2.24	636 \ 173	397 \ 107	0.024 \ 2.24	636 \
	Round	44	2.12 \ 4.21	4.42	44	2.12 \ 4.21	4.42
	ΣΔ	53	2.12 \ 4.21	5.32	24	2.12 \ 4.21	2.49
λ=1e-10	Original	397 \ 107	0.024 \ 2.24	636 \ 173	397 \ 107	0.024 \ 2.24	636 \
	Round	209	0.07 \ 2.39	20.9	209	0.07 \ 2.39	20.9
	ΣΔ	245	0.07 \ 2.39	24.6	110	0.07 \ 2.39	11
λ=3.59e-10	Original	397 \ 107	0.024 \ 2.24	636 \ 173	397 \ 107	0.024 \ 2.24	636 \
	Round	206	0.058 \ 2.3	20.7	206	0.058 \ 2.3	20.7
	ΣΔ	243	0.058 \ 2.3	24.3	109	0.058 \ 2.3	11
λ=1.29e-09	Original	397 \ 107	0.024 \ 2.24	636 \ 173	397 \ 107	0.024 \ 2.24	636 \
	Round	178	0.094 \ 2.42	17.8	178	0.094 \ 2.42	17.8
	ΣΔ	207	0.096 \ 2.42	20.7	92	0.094 \ 2.42	9.2
λ=4.64e-09	Original	397 \ 107	0.024 \ 2.24	636 \ 173	397 \ 107	0.024 \ 2.24	636 \
	Round	164	0.084 \ 2.41	16.4	164	0.084 \ 2.41	16.4
	ΣΔ	193	0.082 \ 2.41	19.4	87	0.084 \ 2.41	8.75
λ=1.67e-08	Original	397 \ 107	0.024 \ 2.24	636 \ 173	397 \ 107	0.024 \ 2.24	636 \
	Round	122	0.19 \ 2.55	12.2	122	0.19 \ 2.55	12.2
	ΣΔ	144	0.19 \ 2.55	14.5	65	0.19 \ 2.55	6.58
λ=5.99e-08	Original	397 \ 107	0.024 \ 2.24	636 \ 173	397 \ 107	0.024 \ 2.24	636 \
	Round	86	0.476 \ 2.88	8.66	86	0.476 \ 2.88	8.66
	ΣΔ	102	0.478 \ 2.88	10.3	47	0.476 \ 2.88	4.71
λ=2.15e-07	Original	397 \ 107	0.024 \ 2.24	636 \ 173	397 \ 107	0.024 \ 2.24	636 \
	Round	72	1.17 \ 3.28	7.21	72	1.17 \ 3.28	7.21
	ΣΔ	87	1.18 \ 3.28	8.78	41	1.17 \ 3.28	4.15
λ=7.74e-07	Original	397 \ 107	0.024 \ 2.24	636 \ 173	397 \ 107	0.024 \ 2.24	636 \
	Round	44	2.32 \ 4.26	4.49	44	2.32 \ 4.26	4.49
	ΣΔ	54	2.32 \ 4.27	5.46	26	2.32 \ 4.26	2.61
λ=2.78e-06	Original	397 \ 107	0.024 \ 2.24	636 \ 173	397 \ 107	0.024 \ 2.24	636 \
	Round	34	5.91 \ 7.37	3.49	34	5.91 \ 7.37	3.49
	ΣΔ	45	5.9 \ 7.37	4.53	23	5.9 \ 7.37	2.3
λ=1e-05	Original	397 \ 107	0.024 \ 2.24	636 \ 173	397 \ 107	0.024 \ 2.24	636 \
	Round	24	14.6 \ 14.6	2.5	24	14.6 \ 14.6	2.5
	ΣΔ	35	14.6 \ 14.6	3.58	19	14.6 \ 14.6	1.98

Table B.1.: Results on the MNIST and Temporal-MNIST datasets. MFlops indicates the number of operations done by each network. For the Original Network, the number of Flops is considered when using both (Dense / Sparse) matrix operations. The "Class Error" column shows the classification error on the training / test set respectively. The "Energy" is an estimate of the average energy that would be used by arithmetic operations per sample, if the network were implemented with all integer values. This is based on the estimates of Horowitz [2014]. Again, for the Original Network, the figure is based on the numbers for dense/sparse matrix operations.

B.6. High-Level Feature Stability

We had initially expected that, when a convolutional network is tasked with processing subsequent frames of video, high-level features would change much more slowly than the pixels and low-level features. This would give a computational advantage to our Sigma-Delta networks, whose computational cost scales with the amount of change in the feature representations. To our surprise, this appeared not to be the case. See the final plot of Figure 4.5. To verify that this was a property of the original convolutional network (and not somehow related our discretization scheme), we take the same snippet of video used for Figure 4.5 and measure the inter-frame differences. Figure B.1 shows the results of this small experiment, and confirms that our initial belief - that inter-frame differences should become smaller and smaller at higher layers, was not quite correct.

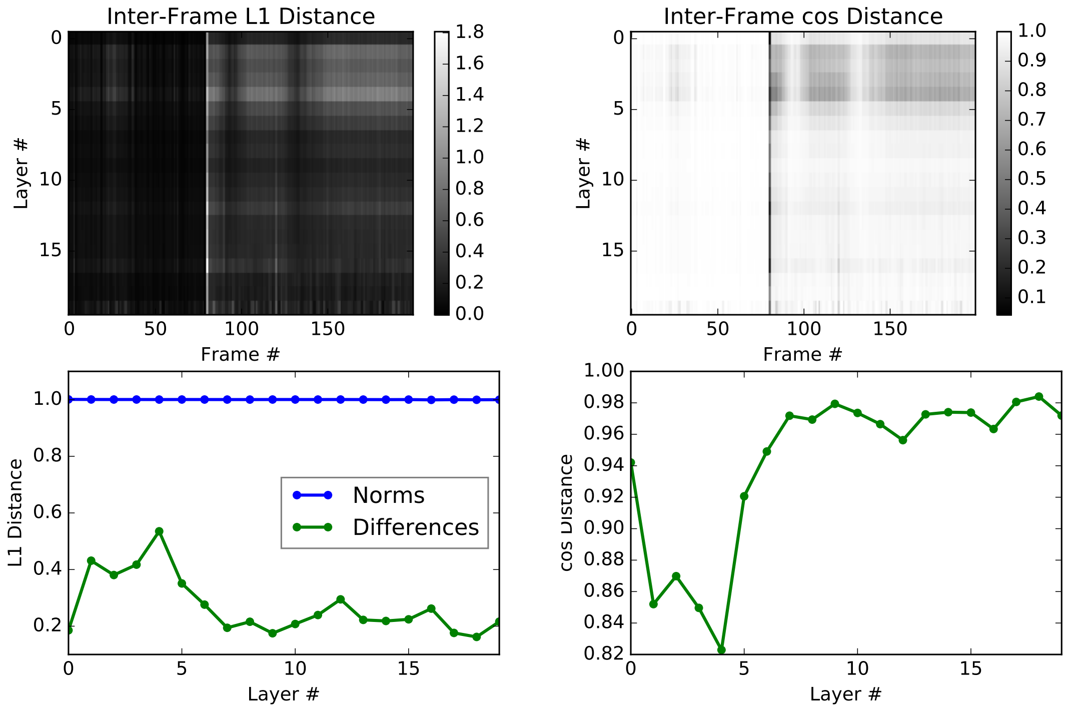


Figure B.1.: **Top-Left**: A heatmap showing the L1-distances between the feature representations (post-nonlinearity) of adjacent frames from the video in Figure 4.5 at different layers (rows) and frames (columns). The input is considered to be layer 0. Feature representations have been L1-normalized per-layer **Bottom Left**: The L1-Norms (which are 1 due to the normalization) and inter-frame L1-Distances for each layer, averaged over frames. **Top and Bottom Right**: The same measurements, with the cosine-similarity metric instead of L1. We note from these plots that the inter-frame difference is not much smaller in higher layers than it is at the pixel level, and that in the lower layers, feature representations of neighbouring frames are significantly more *dissimilar* than they are at the pixel level.

C. Temporally Efficient Deep Learning with Spikes

C.1. Notation

Here we present a legend of notation used throughout this paper. While the paper is intended to be self-contained, the reader may want to consult this list if ever there is any doubt about the meaning of a variable used in the paper. Here we indicate the section in which each symbol is first used.

Section 5.3.1

Δ: A "temporal difference" operator. See Equation 5.1

Σ: A "temporal integration" operator. See Equation 5.2

Q: Sigma-Delta quantization. See Equation 5.3

ϕ: The internal state variable of the quantizer Q.

enc: An "encoding" operation, which takes a signal and encodes it into a combination of the signal's current value and its change in value since the last time step. See Equation 5.4.

dec: A "decoding" operation, which takes an encoding signal and attempts to reconstruct the original signal that it was encoded from. If there was no quantization done on the encoded signal, the reconstruction will be exact, otherwise it will be an approximation. See Equation 5.5.

R: The "rounding" operation, which simply rounds an input to the nearest integer value.

x_t: Used throughout the paper to represent the value of a generic analog input signal at time t. In Sections 5.3.1, 5.3.2, and 5.3.3 it represents a scalar, and thereafter it represents a vector of inputs.

Section 5.3.2

$k_p, k_d \in \mathbb{R}^+$: Positive scalar coefficients used in the encoder and decoder, controlling how the extent to which the encoding is *proportional* to the input (k_p) vs *proportional to the temporal difference* of the input (k_d).

$a_t \triangleq enc(x_t)$: Used to represent the encoded version of x_t.

Section 5.3.3

$s_t \triangleq Q(a_t)$: Used to represent the quantized, encoded version of x_t.

$\hat{x}_t \triangleq dec(s_t)$: Used to represent the reconstruction of input x_t,, obtained by encoding, quantizing, and decoding x_t.

Section 5.3.4

$w_t \in \mathbb{R}^{d_{in} \times d_{out}}$ is the value of a weight matrix at time t.

$z_t \triangleq x_t \cdot w_t \in \mathbb{R}^{d_{out}}$ is the value of a pre-nonlinearity hidden layer activation in a non-spiking network at time t.

$\hat{z}_t \triangleq dec(Q(enc(x_t)) \cdot w_t) \in \mathbb{R}^{d_{out}}$ is the value of a pre-nonlinearity hidden layer activation in the spiking network at time t. It is an approximation of z_t.

Section 5.3.5

$(\cdot w_l)$ indicates applying a function which takes the dot-product of the input with the l'th weight matrix: $(\cdot w_l)(x) \triangleq x \cdot w_l$

h_l indicates an elementwise nonlinearity (e.g. a ReLU).

Q_l indicates the quantization step applied at the l'th layer (because quantization has internal state, ϕ, and an associated layer dimension, we use the subscript to distinguish quantizers at different layers.)

dec_l, enc_l are likewise the (stateful) encoding/decoding functions applied before/after layer l.

$\frac{\partial \mathcal{L}}{\partial z_l}$ is the derivative of the loss with respect to the (pre-nonlinearity) activation of layer l.

$(\cdot w_l^T)$ indicates the dot product with the *transpose* of w_l. This is simply backpropagation across a weight matrix: If $u \triangleq x \cdot w_l$, then $\frac{\partial \mathcal{L}}{\partial x} = \frac{\partial \mathcal{L}}{\partial u} \cdot w_l^T$

$\hat{z}_l$ is the approximation to the (pre-nonlinearity) activation to layer l (ie the output of dec_l), computed by the spiking network.

$(\odot h_l'(\hat{z}_l))$ is a function that performs an elementwise multiplication of the input with the derivative of nonlinearity h_l evaluated at $\hat{z}_l$. This is simply backpropagation across a nonlinearity: If $u \triangleq h_l(x)$, then $\frac{\partial \mathcal{L}}{\partial x} = \frac{\partial \mathcal{L}}{\partial u} \odot h_l'(x)$

dec_l^{back}, enc_l^{back}, Q_l^{back} serve the same functions as dec_l, enc_l, Q_l, but for the backward pass.

$\widehat{\frac{\partial \mathcal{L}}{\partial \hat{z}_l}} \in \mathbb{R}^{d_l}$ Is our approximation to the derivative of the loss of our network with respect to $\hat{z}_l$, which is itself an approximation of the activation z_l in a non-spiking implementation of the network.

Section 5.3.6

In the updates section we describe how we calculate the weight gradients in layer l. Because this description holds for any arbitrary layer, we get rid of the layer subscript and use the following notation:

$x_t \triangleq h_{l-1}(z_{l-1,t}) \in \mathbb{R}^{d_{in}}$ here is defined as a shorthand for "the input to layer l".

$e_t \triangleq \widehat{\frac{\partial \mathcal{L}}{\partial \hat{z}_{l,t}}} \in \mathbb{R}^{d_{out}}$ is simply a shorthand for "the approximate backpropagated gradient at layer l"

($\bar{x}_t$ and $\bar{e}_t$) are the encoded and quantized versions of signals (x_t and e_t)

($\hat{x}_t$ and $\hat{e}_t$) are the reconstructed versions of signals (x_t and e_t), taken from the quantized ($\bar{x}_t$ and $\bar{e}_t$)

$\widehat{\frac{\partial \mathcal{L}}{\partial w}}_{recon,t} \triangleq \hat{x}_t \otimes \hat{e}_t \in \mathbb{R}^{d_{in} \times d_{out}}$ is the approximate gradient of weight matrix w, as calculated by taking the outer product of the (input, error) reconstructions, $\hat{x}$, $\hat{e}$.

$\left(\widehat{\frac{\partial \mathcal{L}}{\partial w}}_{past,t}, \widehat{\frac{\partial \mathcal{L}}{\partial w}}_{future,t}\right)$ are the gradient approximations at time t taken using the (past, future) approximation methods, defined in Appendix C.4. They are more efficient to calculate than $\widehat{\frac{\partial \mathcal{L}}{\partial w}}_{recon,t}$ but converge to the same value when averaged over time (i.e. $\lim_{T\to\infty} \frac{1}{T}\sum_t^T \widehat{\frac{\partial \mathcal{L}}{\partial w}}_{future,t} = \frac{1}{T}\sum_t^T \widehat{\frac{\partial \mathcal{L}}{\partial w}}_{recon,t}$ (see Figure C.1).

Section 5.3.7

$\widehat{\frac{\partial \mathcal{L}}{\partial w}}_{stdp,t}$ is the gradient approximation taken using the STDP-type update. It also converges to the same value as $\widehat{\frac{\partial \mathcal{L}}{\partial w}}_{recon,t}$ when averaged out over time.

$k_\alpha \triangleq \frac{k_d}{k_p+k_d} \in [0,1], k_\beta \triangleq \frac{1}{k_p+k_d} \in R^+$: A reparametrization of k_p, k_d in terms of the *memory* in our decoder k_α and the *scaling of our encoded signal* (k_β). This reparametrization is also used when discussing the automatic tuning of k_p, k_d to match the dynamic range of our data in Appendix C.5

C.2. Relation to Predictive Coding

Our encoding/decoding scheme is an instance of predictive coding - an idea imported from the signal processing literature into neuroscience by Srinivasan et al. [1982] wherein the power of a transmitted signal is reduced by subtracting away the predictable component of this signal before transmission, then reconstructing it after (This requires that the encoder and decoder share the same prediction model).

Bharioke and Chklovskii [2015] formulate feedforward predictive coding as follows (with variables names changed to match the conventions of this paper):

$$\begin{aligned} a_t &\triangleq x_t - C_{feedforward}(x_{t-1}, x_{t-2}, ...) && \text{(C.1)} \\ &= x_t - \sum_{\tau=1}^{\infty} \omega_\tau x_{t-\tau} && \text{In the case of Linear Predictive Coding} \quad \text{(C.2)} \end{aligned}$$

Where the reconstruction is done by:

$$x_t = a_t + C_{feedforward}(x_{t-1}, x_{t-2}, ...) \tag{C.3}$$

They go on to define "optimal" liner filter parameters $[w_1, w_2, ...]$ that minimize the average magnitude of a_t in terms of the autocorrelation and signal-to-noise ratio of x.

Our scheme defines:

$$a_t \triangleq k_p x_t + k_d(x_t - x_{t-1}) = (k_p + k_d)\left(x_t - \frac{k_d}{k_p + k_d} x_{t-1}\right) \tag{C.4}$$

So it is identical to feedforward predictive coding with $\omega_\tau = \begin{cases} \frac{k_d}{k_p+k_d} & \text{if } \tau = 1 \\ 0 & \text{otherwise} \end{cases}$ up to a scaling constant of $(k_p + k_d)$. In our case, the function of this additional constant is to determine the coarseness of the quantization.

From this relationship it is clear that this work could be extended to come up with more efficient predictive coding schemes which could further reduce computation by learning the temporal characteristics of the input signal.

C.3. Sigma-Delta Unwrapping

Here we show that $Q = \Delta \circ R \circ \Sigma$, where Q, Δ, R, Σ are defined in Equations 5.3, 5.2, 5.6, 5.1, respectively.

From Equation 5.3 (Q) we can see that

$$\begin{aligned} y_t &\leftarrow round(x_t + \phi_{t-1}) \in \mathbb{Z} \\ \phi_t &\leftarrow \phi_{t-1} + x_t - y_t \in \mathbb{R} \end{aligned}$$

Now we can unroll for y_t and use the fact that if $s \in \mathbb{Z}$ then $round(a+s) = round(a) + s$:

$$\begin{aligned}
y_t &= round(x_t + \phi_{t-1}) \\
&= round(x_t + \phi_{t-2} + x_{t-1} - y_{t-1}) \\
&= round(x_t + x_{t-1} + \phi_{t-2}) - y_{t-1} \\
&= round(x_t + x_{t-1} + \phi_{t-2}) - round(x_{t-1} + \phi_{t-2}) \\
&= \left(round(\sum_{\tau=1}^{t} x_\tau + \cancelto{0}{\phi_0}) - \sum_{\tau=0}^{t-2} y_\tau \right) - \left(round(\sum_{\tau=1}^{t-1} x_\tau + \cancelto{0}{\phi_0}) - \sum_{\tau=0}^{t-2} y_\tau \right) \\
&= round(\sum_{\tau=1}^{t} x_\tau) - round(\sum_{\tau=1}^{t-1} x_\tau)
\end{aligned} \tag{C.5}$$

At which point it is clear that Q is identical to a successive application of a temporal summation, a rounding, and a temporal difference. That is why we say $Q = \Delta \circ R \circ \Sigma$.

C.4. Update Algorithms

In Section 5.3.6, we visually describe what we call the "Past" and "Future" parameters updates. Here we present the algorithms for implementing these schemes.

To simplify our expressions in the update algorithms, we re-parametrize our k_p, k_d coefficients as $k_\alpha \triangleq \frac{k_d}{k_p + k_d}$, $k_\beta \triangleq \frac{1}{k_p + k_d}$.

$$\begin{aligned}
&past : (\bar{x} \in \mathbb{Z}^{d_{in}}, \bar{e} \in \mathbb{Z}^{d_{out}}) \mapsto \widehat{\frac{\partial \mathcal{L}}{\partial w}}_{past} \\
&\quad \text{Persistent: } w, u \in \mathbb{R}^{d_{in} \times d_{out}}, \\
&\quad\quad \hat{x} \leftarrow 0^{d_{in}}, \hat{e} \leftarrow 0^{d_{out}} \\
&\quad i \leftarrow \bar{x} \neq 0, \quad j \leftarrow \bar{e} \neq 0 \\
&\quad \hat{x} \leftarrow k_\alpha \hat{x} \quad , \quad \hat{e} \leftarrow k_\alpha \hat{e} \\
&\quad v \leftarrow \hat{x}_i \otimes \hat{e}_j \in \mathbb{R}^{\sum_{i'}[\bar{x}_{i'} \neq 0] \times \sum_{j'}[\bar{e}_{j'} \neq 0]} \\
&\quad \widehat{\frac{\partial \mathcal{L}}{\partial w}}_{past,i,j} \leftarrow \frac{u_{i,j} - v}{1 - k_\alpha^2} \\
&\quad \hat{x} \leftarrow \hat{x} + k_\beta \bar{x}, \quad \hat{e} \leftarrow \hat{e} + k_\beta \bar{e} \\
&\quad u_{i,j} \leftarrow v
\end{aligned} \tag{C.6}$$

$$\begin{aligned}
&future : (\bar{x} \in \mathbb{Z}^{d_{in}}, \bar{e} \in \mathbb{Z}^{d_{out}}) \mapsto \widehat{\frac{\partial \mathcal{L}}{\partial w}}_{future} \\
&\quad \text{Persistent: } w \in \mathbb{R}^{d_{in} \times d_{out}}, \\
&\quad\quad \hat{x} \leftarrow 0^{d_{in}}, \hat{e} \leftarrow 0^{d_{out}} \\
&\quad \hat{x} \leftarrow k_\alpha \hat{x} \\
&\quad \hat{e} \leftarrow k_\alpha \hat{e} + k_\beta \bar{e} \\
&\quad \widehat{\frac{\partial \mathcal{L}}{\partial w}}_{future} \leftarrow \frac{\bar{x} \otimes \hat{e} + \hat{x} \otimes \bar{e}}{k_\alpha^2 - 1} \\
&\quad \hat{x} \leftarrow \hat{x} + k_\beta \bar{x}
\end{aligned} \tag{C.7}$$

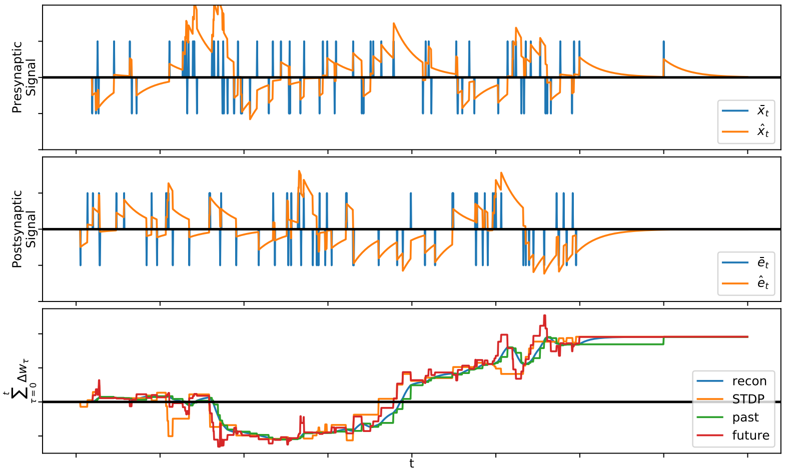


Figure C.1.: In Section 5.3.6 and 5.3.7, we described 4 different update rules ("Reconstruction", "Past", "Future", and "STDP"), and stated that while they do not necessarily produce the same updates at the same times, they produce the same result in the end. Here we demonstrate this empirically. We generate two random spike-trains representing the presynaptic input and the postsynaptic error signal to a single synapse and observe how the weight changes according to the different update rules. **Top**: A randomly generated presynaptic quantized signal $\bar{x}$, along with its reconstruction $\hat{x}$. **Middle**: A randomly generated postsynaptic quantized error signal $\bar{e}$, along with its reconstruction $\hat{e}$. **Bottom**: The cumulative weight update arising from our four updates methods. "*recon*" is just $\sum_{\tau=1}^{t} \hat{x}_\tau \hat{e}_\tau$, "*past*" and "*future*" are described in Section 5.3.6 and "*STDP*" is described in Section 5.3.7. Note that by the end all methods arrive at the same final-weight value.

C.5. Tuning k_p, k_d

C.5.1. Causes of approximation error

Equation 5.7 shows how we make two approximations when approximating $z_t = x_t \cdot w_t$ with $\hat{z}_t = (dec \circ w \circ Q \circ enc)(x_t)$. The first is the "nonstationary weight" approximation, arising from the fact that w changes in time. The second is the "quantization" approximation, arising from the quantization of x. Here we do a small experiment in which we multiply a time-varying scalar signal x_t with a time-varying weight w_t for many different values of k_p, k_d to understand the effects of k_p, k_d on our approximation error. The bottom-middle plot in Figure C.2 shows that we enter a high-reconstruction-error regime (blue on plot) when k_d is small (high quantization error), or when $k_d >> k_p$ (high nonstationary-weight error). The bottom-right plot shows that blindly increasing k_p and k_d leads to representing the signal with many more spikes. Thus we need to tune hyperparameters to find the "sweet spot" where reconstruction error is fairly low but our encoded signal remains fairly sparse, keeping computational costs low.

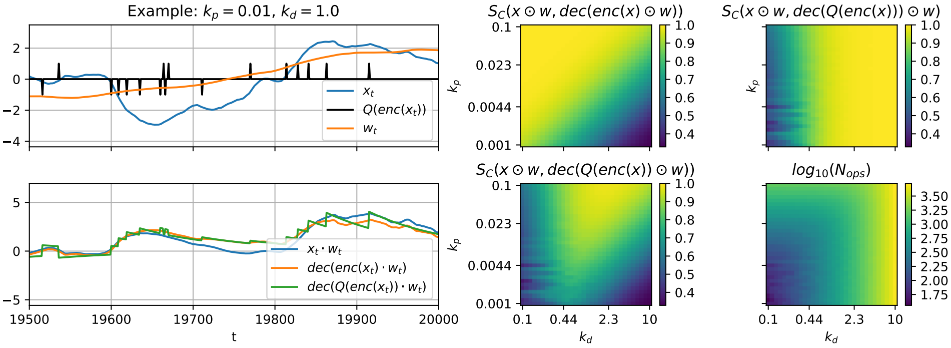


Figure C.2.: **Top Left**: A time varying signal x_t, the quantized signal $Q(enc(x_t))$, and the time-varying "weight" w_t. **Bottom Left**: Compare the true product of these signals $x_t \cdot w_t$ with the $dec(enc(x_t) \cdot w_t)$, which shows the effects of the non-stationary weight approximation, and $dec(Q(enc(x_t)) \cdot w)$ which shows both approximations. **Top Middle**: The Cosine distance between the "true" signal $x \odot w$ and the approximation due to the nonstationary w, scanned over a grid of k_p, k_d values. **Top Right**: The cosine distance between the "true" signal and the approximation due to the quantization of x. **Bottom Middle**: The Cosine Distance between the "true" signal and the full approximation described in Equation 5.7. This shows why we need both k_p and k_d to be nonzero. **Bottom Right**: The Number of spikes in the encoded signal. In a neural network this would correspond to the number of weight-lookups required to compute the next layer's activation: $dec(Q(enc(x)) \odot w)$.

C.5.2. An auto-tuning scheme for k_p, k_d

The smaller the magnitude of a signal, the more severely distorted it is by our quantization-reconstruction scheme. We can see that scaling a signal by K has the same effect on the quantized version of the signal, s_t, as scaling k_p and k_d by K: $s_t = (Q \circ enc_{k_p,k_d})(Kx_t) = Q(k_p K x_t + k_d(Kx_t - Kx_{t-1})) = Q(Kk_p x_t + Kk_d(x_t - x_{t-1})) = (Q \circ enc_{Kk_p,Kk_d})(x_t)$. The fact that the reconstruction quality depends on the signal magnitude presents a problem when training our network, because the error gradients tend to change in magnitude throughout training (they start large, and become smaller as the network learns). To keep our signal within the useful dynamic range of the quantizer, we apply a simple scheme to heuristically adjust k_p and k_d for the forward and backward passes separately, for each layer of the network. Instead of directly setting k_p, k_d as hyperparameters, we fix the ratio $k_\alpha \triangleq \frac{k_d}{k_p+k_d}$, and adapt the scale $k_\beta \triangleq \frac{1}{k_p+k_d}$ to the magnitude of the signal. Our update rule for k_β is:

$$\begin{aligned} \mu_t &= (1-\eta_k)\mu_{t-1} + \eta_k \cdot |x_t|_{L_1} \\ k_\beta &= k_\beta + \eta_k(k_\beta^{rel} \cdot \mu_t - k_\beta) \end{aligned} \tag{C.8}$$

Where η_k is the scale-adaptation learning rate, μ_t is a rolling average of the L_1 magnitude of signal x_t, and k_β^{rel} defines how coarse our quantization should be relative to the signal magnitude (higher means coarser). We can recover k_p, k_d for use in the encoders and decoders as $k_p = (1-k_\alpha)/k_\beta$ and $k_d = k_\alpha/k_\beta$. In our experiments, we choose $\eta_k = 0.001, k_\beta^{rel} = 0.91, k_{alpha} = 0.91$, and initialize $\mu_0 = 1$.

C.6. MNIST Results

Here we show training scores and computation for the PDNN and MLP under different input-orderings (the unordered MNIST vs the ordered Temporal MNIST) and hidden layer depths. We notice no dropoff in performance of the PDNN (as compared to an MLP) with the same architecture as we add hidden layers - indicating that the accumulation of quantization noise over layers appears not to be a problem. For all experiments, the PDNN started with $k_\alpha = 0.5$, and this was increased to $k_\alpha = 0.9$ after 1 epoch (see Appendix C.1 for the meaning of k_α). Note that the numbers for Mean Computation are counting additions for the PDNN, and multiply-adds for the MLP, so they are not directly comparable (a 32-bit multiply, if implemented in fixed point, is 32 times more energetically expensive than an add [Horowitz, 2014])

dataset	hidden_sizes	Network	kOps/sample	Training Score	Test Score
mnist	[200]	PDNN	711000	100	98.34
mnist	[200]	MLP	314000	100	98.3
mnist	[200, 200]	PDNN	1000000	99.82167	98.18
mnist	[200, 200]	MLP	434000	100	98.5
mnist	[200, 200, 200]	PDNN	1300000	99.91	98.16
mnist	[200, 200, 200]	MLP	554000	99.99333	98.39
mnist	[200, 200, 200, 200]	PDNN	1620000	99.96	98.41
mnist	[200, 200, 200, 200]	MLP	674000	99.99167	98.28
temporal_mnist	[200]	PDNN	484000	100	98.39
temporal_mnist	[200]	MLP	314000	100	98.2
temporal_mnist	[200, 200]	PDNN	740000	99.97833	98.27
temporal_mnist	[200, 200]	MLP	434000	100	98.38
temporal_mnist	[200, 200, 200]	PDNN	967000	99.98	98.31
temporal_mnist	[200, 200, 200]	MLP	554000	99.99833	98.45
temporal_mnist	[200, 200, 200, 200]	PDNN	1170000	99.995	98.18
temporal_mnist	[200, 200, 200, 200]	MLP	674000	100	98.53

C.7. Sample frames from the YouTube-BB Dataset

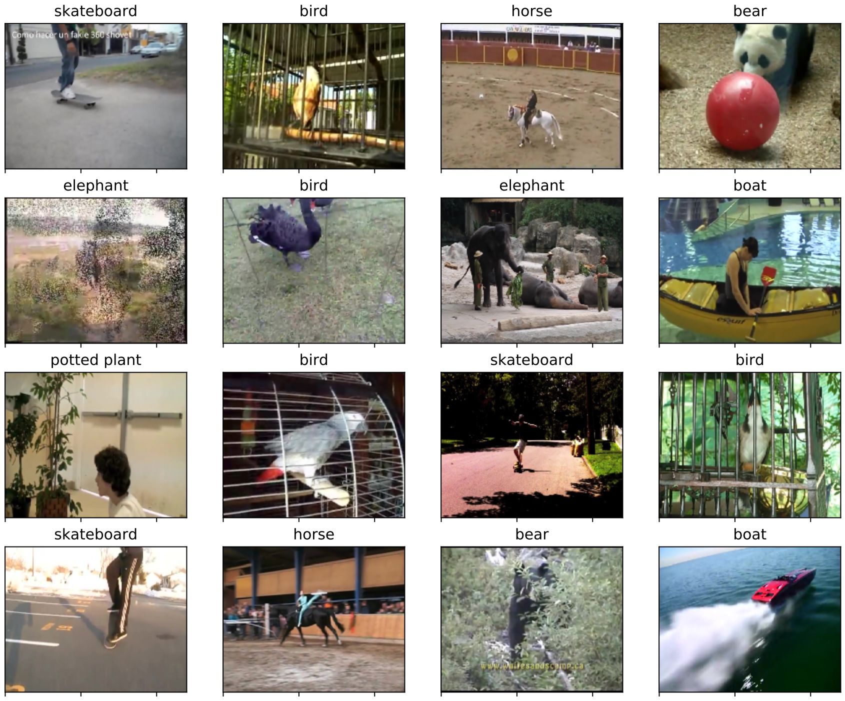


Figure C.3.: 16 Frames from the Youtube-BB dataset. Each video annotated as having one of 24 objects in it. It also comes with annotated bounding-boxes, which we do not use in this study.

C.8. Instability in Neural Network Representations

Figure 5.8 seems to show that computation doesn't quite approach zero as our frame-rate increases, but flat-lines at a certain point. We think this may have to do with the fact that hidden layer activations are not necessarily smoother in time than the input. We demonstrate this by taking 5 video snippets from the Youtube-BB dataset and running them through a (non-spiking) 19 layer VGGNet architectures [Simonyan and Zisserman, 2014], which was pre-trained on ImageNet.

Given these 5 snippets, we measures how much the average relative change in layer activation $\frac{|a_t-a_{t-1}|}{2(|a_t|+|a_{t-1}|)}$ varies as we increase our frame-rate, at various layer-depths. We simulate lower frame rates by skipping every N'th frame of video. (so for example to get a 10FPS frame rate we simply select every 3rd frame of the 30FPS video). For each selected frame rate, and for the given layers, we measure the average inter-frame change at various layers:

$$FPS(n) = 30/n \qquad \text{x-axis} \tag{C.9}$$

$$RelChange(n) = \frac{1}{S}\sum_{s=1}^{S}\sum_{t=1}^{T/n}\frac{|a_{nt} - a_{(n-1)t}|}{2(|a_{nt}| + |a_{(n-1)t}|)} \qquad \text{y-axis} \tag{C.10}$$

Where:
$S = 5$ is the number of video snippets we average over
T is the number of frames in each snippet
a_t is the activation of a layer at time t
n is the number of frames we are skipping over.

This shows something interesting. While our deeper layers do indeed show less relative change in activation over frames than our input/shallow layers, we note that as frame-rate increases, this seems to approach zero *much more slowly* than our input/shallow layers. This is a problem for our method, which relies on temporal smoothness in all layers (especially those hidden layers with large amounts of downstream units) to save computation. It suggests that methods for learning *slow feature detectors* - layers that are trained specifically to look for slowly varying features of the input, may be helpful to us.

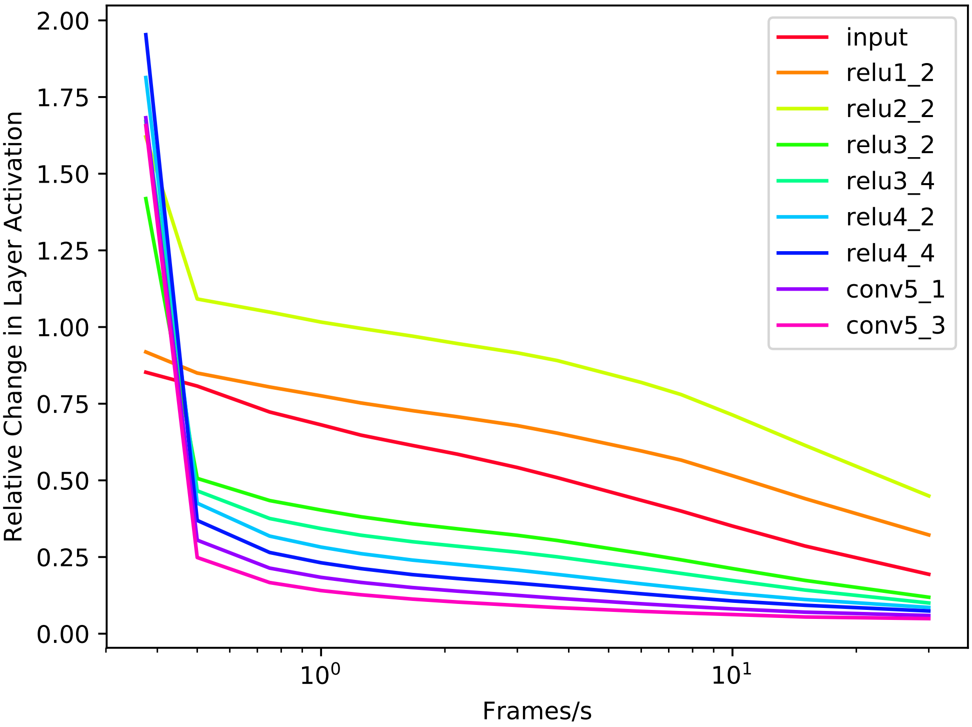


Figure C.4.: The average relative change in layer activation between frames, as frame-rate increases. For increasing network depth (red=shallow, violet=deep)

D. Initialized Equilibrium Propagation

D.1. Glossary

Here we we have a reference of symbols used in the paper, in (Greek, Latin) alphabetical order.

$\alpha_i \subset \{j : j \in \mathcal{S}, j \neq i\}$: The set of neurons in the Equilibrating Network that connect to neuron i

$\alpha_i^f = \{j : j \in \alpha_i, j < i\}$: The set of neurons in the Feedforward Network connected to neuron i.

$\beta \in \mathbb{R}$: The perturbation factor, which modulates how much the *augmented energy* E_θ^β is affected by the output loss.

$\eta \in \mathbb{R}^+$: The learning rate.

θ: The set of parameters (all w_{ij}'s and b_j's, in the Equilibrating network)

ρ: a neuron nonlinearity. In all experiments it is $\rho(x) = max(0, min(1, x))$

$\phi = (\omega, c)$: The set of parameters (all ω_{ij}'s and c_j's), in the feedforward network

ϕ_j: The set of parameters (all $\omega_{\cdot j}$'s and c_j') belonging to a neuron j

(ω, c): The (weights, biases) of the feedforward network. Collectively called ϕ

$C(s_\mathcal{O}, y) \in \mathbb{R}$: The output loss function, defined on the states of the output units.

$E_\theta(s, x) \in \mathbb{R}$: The energy function of the Equilibrating network (Equation 2.9) produces a scalar energy given a set of states s and input x

$E_\theta^\beta(s, x, y) = E_\theta(s, x) + \beta \frac{\partial C(s_\mathcal{O}, y)}{\partial s} \in \mathbb{R}$: The *augmented* energy function of the Equilibrating network, when it has been perturbed by a factor β by target data y

$f_\phi(x) \mapsto s^f$: The *initialization* function: A feedforward network which initializes the state of the Equilibrating network.

$\mathcal{I}$: The set of indices of input neurons.

$\mathcal{L}$: The total loss in the Feedforward network's prediction. Defined in Equation 7.3

$\mathcal{L}_i$: The local loss on the $i'th$ neuron in the feedforward network. Defined in Equation 7.3

$\mathcal{O}$: The set of indices output neurons, a subset of $\mathcal{S}$

$\mathcal{S}$: The set of inidices non-input neurons.

s: The set of neuron states. $s := \{s_i : i \in \mathcal{S}\} \in \mathbb{R}^{\dim(\mathcal{S})}$

$s^- := \arg\min_s E(s,x) \in \mathbb{R}^{\dim(\mathcal{S})}$: The minimizing state of the Energy function.

$s^+ := \arg\min_s E_\theta^\beta(s,x,y) \in \mathbb{R}^{\dim(\mathcal{S})}$ The minimizing state of the augmented energy function.

$s^f := f_\phi(x)z \in \mathbb{R}^{\dim(\mathcal{S})}$: The state output by the feedforward network.

$s_\mathcal{O} \in \mathbb{R}^{\dim(\mathcal{O})}$: the states of the output units

T^-, T^+: Hyperparameters for Equilibrium Prop defining the number of steps of convergence of the negative/positive phase.

w, b: the parameters of the Equilibrating network (collectively called θ)

$x \in \mathbb{R}^{\dim \mathcal{I}}$: The input data

$y \in \mathbb{R}^{\dim \mathcal{O}}$: The target data

D.2. Gradient Alignment

Here to derive the result in Equation 7.7. First, we restate Equation 7.6 substituting $w^* = w + \Delta w$:

$$\begin{aligned} s_1 &= \rho(xw_1) & s_1^* &= \rho(x(w_1 - \Delta w_1)) & \mathcal{L}_1 &= \|s_1 - s_1^*\|_2^2 \\ s_2 &= \rho(s_1 w_2) & s_2^* &= \rho(s_1^*(w_2 - \Delta w_2)) & \mathcal{L}_2 &= \|s_2 - s_2^*\|_2^2 \end{aligned}$$

Where:

$x \in \mathbb{R}^{N \times D_0}$; $s_1, s_1^* \in \mathbb{R}^{N \times D_1}$; $s_2, s_2^* \in \mathbb{R}^{N \times D_2}$; $w_1, \Delta w_1 \in \mathbb{R}^{D_0 \times D_1}$; $w_2, \Delta w_2 \in \mathbb{R}^{D_1 \times D_2}$

Now we will compute the gradient of each of the local losses with respect to Δw_1, in the limit where Δw_1 is small.

$$\begin{aligned} \frac{\partial \mathcal{L}_1}{\partial w_1} &= \frac{\partial \mathcal{L}_1}{\partial s_1} \frac{\partial s_1}{\partial w_1} \\ &= \Big(\big((s_1 - s_1^*) \odot \rho'(xw) \big)^T \cdot x \Big)^T \\ &= \overline{x^T \Big(\big(\rho(xw_1) - \rho(x(w_1 - \Delta w_1)) \big) \odot \rho'(xw_1) \Big)}^{\,g(\Delta w_1)} \\ &\overset{\lim_{\Delta w \to 0}}{=} g(0) + \Delta w_1 \frac{\partial g}{\partial \Delta w_1}(0) \qquad \text{(1st order Taylor Expansion about } \Delta w_1 = 0\text{)} \\ &= x^T \big(\rho(x(w_1 - 0)) - \rho(xw_1)) \rho'(xw_1) \big) + x^T \Big(x \Delta w_1 \odot \rho'(xw_1) \rho'(xw_1) \Big) \\ &= 0 + x^T \Big(x \Delta w_1 \odot \rho'(xw_1)^2 \Big) \end{aligned}$$

$$
\begin{aligned}
\frac{\partial \mathcal{L}_2}{\partial w_1} &= \frac{\partial \mathcal{L}_2}{\partial s_2}\frac{\partial s_2}{\partial s_1}\frac{\partial s_1}{\partial w_1} \\
&= \Big(\big((s_2 - s_2^*) \odot \rho'(s_1 w_2) w_2^T \rho'(x w_1) \big)^T x \Big)^T \\
&= x^T \Big((s_2 - s_2^*) \odot \rho'(s_1 w_2) w_2^T \rho'(x w_1) \Big) \\
&= \overbrace{x^T \Big(\big(\rho(\rho(x w_1) w_2) - \rho(\rho(x(w_1 - \Delta w_1))(w_2 - \Delta_{w_2})) \big) \odot \rho'(\rho(x w_1) w_2) w_2^T \rho'(x w_1) \Big)}^{g(\Delta w_1, \Delta w_2)} \\
&\overset{\lim_{\Delta w \to 0}}{=} g(0,0) + \Delta w_1 \frac{\partial g}{\partial \Delta w_1}(0,0) + \Delta w_2 \frac{\partial g}{\partial \Delta w_2}(0,0) \qquad \text{(1st order Taylor Expansion about } \Delta w_1 = \\
&= 0 + \overbrace{x^T \Big(x \Delta w_1 \odot \rho'(x w_1) w_2 \odot \rho'(s_1 w_2)^2 w_2^T \odot \rho'(x w_1) \Big)}^{G_1} \; \overbrace{+ x^T \Big(s_1 \Delta w_2 \odot \rho'(s_1 w_2)^2 w_2^T \odot \rho'(x w_1) \Big)}^{G_2}
\end{aligned}
$$

D.3. Gradient Alignment at Initialization

Why do we observe gradient alignment even at random initialization? Let us start with the same 2-layer network defined in Appendix D.2

$$\begin{aligned}
\frac{\partial \mathcal{L}_1}{\partial w_1} &= \frac{\partial \mathcal{L}_1}{\partial s_1}\frac{\partial s_1}{\partial w_1} \\
&= \Big(\big((s_1 - s_1^*) \odot \rho'(xw)\big)^T \cdot x\Big)^T \\
&= x^T\Big(\big(\rho(xw_1) - \rho(xw_1^*)\big)\odot\rho'(xw_1)\Big) \\
&= \overbrace{x^T\rho(xw_1)\odot\rho'(xw_1)}^{G_A} - x^T\rho(xw_1^*)\odot\rho'(xw_1)
\end{aligned}$$

$$\begin{aligned}
\frac{\partial \mathcal{L}_2}{\partial w_1} &= \frac{\partial \mathcal{L}_2}{\partial s_2}\frac{\partial s_2}{\partial s_1}\frac{\partial s_1}{\partial w_1} \\
&= \Big(\big((s_2 - s_2^*) \odot \rho'(s_1w_2)w_2^T\rho'(xw_1)\big)^T x\Big)^T \\
&= x^T\Big((s_2 - s_2^*) \odot \rho'(s_1w_2)w_2^T\rho'(xw_1)\Big) \\
&= x^T\Big(\big(\rho(\rho(xw_1)w_2) - \rho(\rho(xw_1^*)w_2^*)\big) \odot \rho'(\rho(xw_1)w_2)w_2^T\rho'(xw_1)\Big) \\
&= \overbrace{x^T\Big(\rho(\rho(xw_1)w_2) \odot \rho'(\rho(xw_1)w_2)w_2^T\rho'(xw_1)\Big)}^{G_B} - \\
&\quad x^T\Big(\rho(\rho(xw_1^*)w_2^*) \odot \rho'(\rho(xw_1)w_2)w_2^T\rho'(xw_1)\Big)
\end{aligned}$$

G_A and G_B tend to be aligned because the terms $\rho(xw_1)$ and $\rho(\rho(xw_1)w_2)\odot\rho'(\rho(xw_1)w_2)w_2^T$ tend to be aligned. Suppose ρ is a piecewise saturating nonlinearity (as we have in this paper) with $\rho(x) = [a \text{ if } (x < a); x \text{ if } (x \in [a, b]); b \text{ otherwise}]$

Then we can define a weight matrix w_2' by filtering rows of w_2 to only include weights projecting to non-saturated neurons: $w_2' = [w_2^{(i)} \forall i : \rho'(\rho(xw_1)w_2^{(i)}) \neq 0]$ Where $w_2^{(i)}$ denotes the $i'th$ row of w_2.

Then our second term can be rewritten as: $\rho(\rho(xw_1)w_2)\odot\rho'(\rho(xw_1)w_2)w_2^T = \rho(xw_1)w_2'w_2'^T$. Given a random matrix w_2', the matrix $w_2'w_2'^T$ will tend to have a strong diagonal component, causing this term to be aligned with $\rho(xw_1)$

D.4. Effect of λ parameter

In Equation 7.5 we introduce a new parameter λ which encourages the state of the equilibrating network to state close to that of the forward pass. Here we perform a sweep of parameter λ to evaluate its effect.

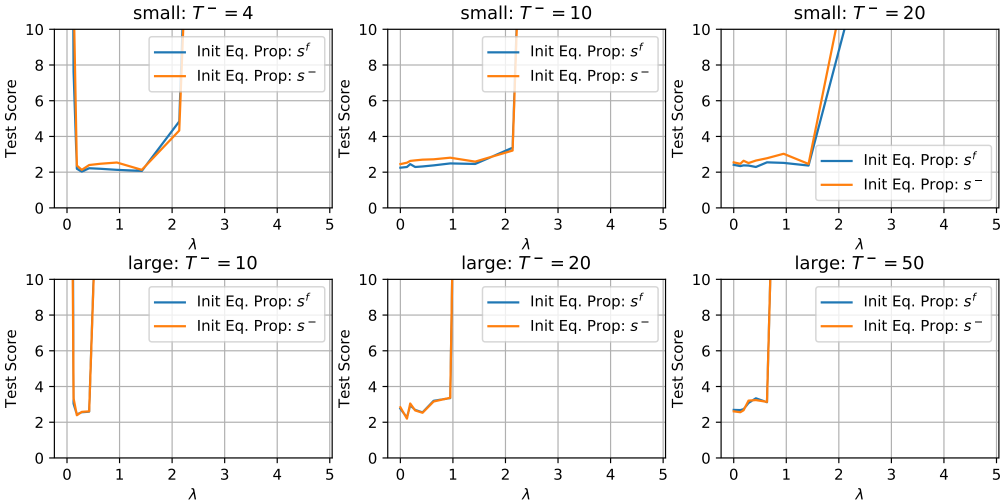


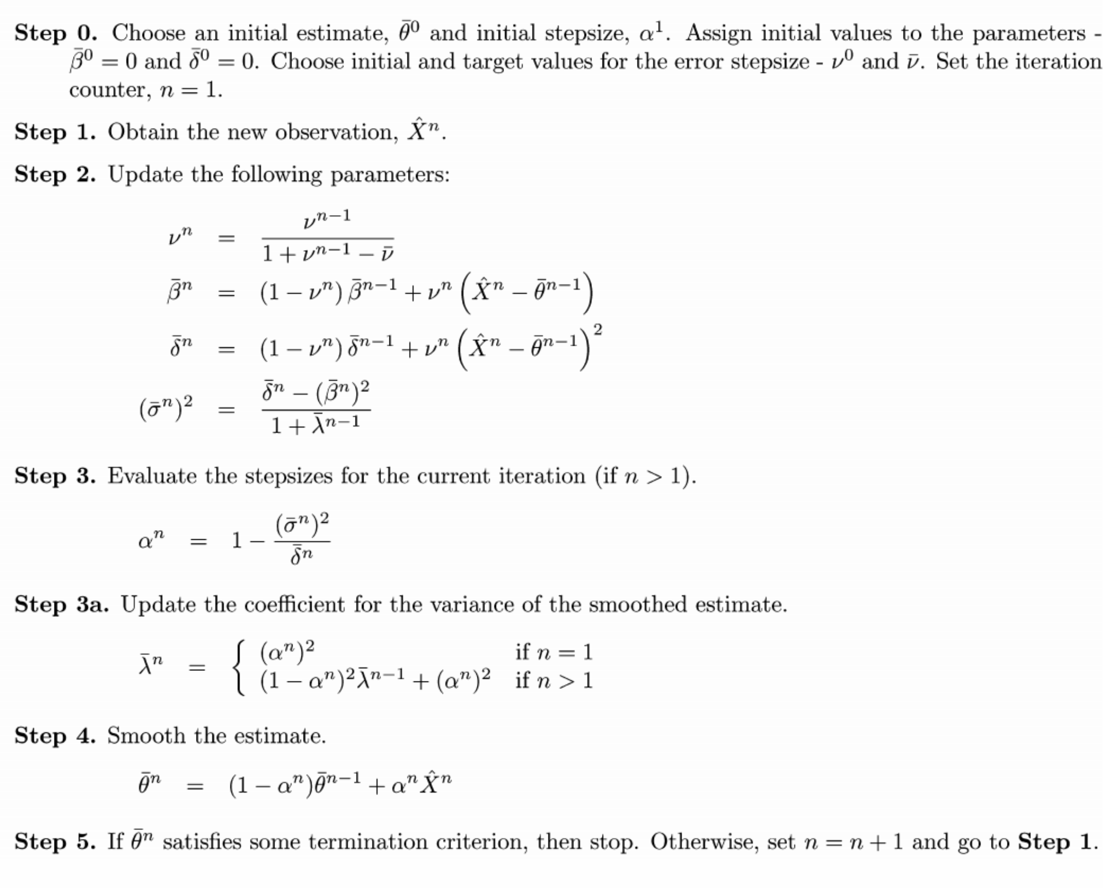
Figure D.1.: Here we scan the λ parameter and plot the final score at the end of training. Each point in each plot corresponds to the final score of a network with parameter λ fixed at the given value throughout training. The top row of plots is a for a small network with one hidden layer of 500 hidden units. The bottom is for a large network with 3 layers of [500, 500, 500] hidden units. Each column is for a different number of steps of negative-phase convergence.

We see in Figure D.1 that when the number of steps of negative-phase convergence is small, introducing λ can allow for more stable training. This makes sense - if the minimizing state of the equilibating network is "pulled" towards the state at the forward pass, it should take fewer steps of iteration to reach this state when initialized at the state of the forward pass. However, we also see that training fails when λ is too high. We believe this is because the simple iterative settling scheme (Euler integration) used in this paper, as well as the original Equilibrium Prop by Scellier and Bengio [2017], can become unstable when optimizing a loss surface with sharp, steep, minima (as are induced with large λ). This could be addressed in later work by either using an adaptive λ term or an adaptive Euler-integration step-size.

E. Spiking Equilibrium Propagation

E.1. Optimal Step-Size Adaptation

George and Powell [2006] propose Optimal Step-Size Adaptation (OSA), an algorithm for optimally estimating the value of a parameter θ given only noisy samples $x^n = \theta^n + \epsilon^n$ (n is a superscript indexing the ordered sequence of samples). The algorithm assumes that ϵ is some zero-mean IID noise and that we generate our estimates $\bar{\theta}^n$ by averaging: $\bar{\theta}^n = (1 - \alpha^n)\bar{\theta}^{n-1} + \alpha^n x^n$. The algorithm is "optimal" in the sense that the step sizes α^n are optimal if the bias $\beta = \theta^n - \mathbb{E}[\bar{\theta}^{n-1}]$ in θ and the magnitude σ of the noise are known. Since they are generally not, OSA also estimates these from the input stream using running averages. The algorithm, copied directly from their paper, is as follows:



Step 0. Choose an initial estimate, $\bar{\theta}^0$ and initial stepsize, α^1. Assign initial values to the parameters - $\bar{\beta}^0 = 0$ and $\bar{\delta}^0 = 0$. Choose initial and target values for the error stepsize - ν^0 and $\bar{\nu}$. Set the iteration counter, $n = 1$.

Step 1. Obtain the new observation, $\hat{X}^n$.

Step 2. Update the following parameters:

$$\begin{aligned}
\nu^n &= \frac{\nu^{n-1}}{1 + \nu^{n-1} - \bar{\nu}} \\
\bar{\beta}^n &= (1 - \nu^n)\,\bar{\beta}^{n-1} + \nu^n \left(\hat{X}^n - \bar{\theta}^{n-1}\right) \\
\bar{\delta}^n &= (1 - \nu^n)\,\bar{\delta}^{n-1} + \nu^n \left(\hat{X}^n - \bar{\theta}^{n-1}\right)^2 \\
(\bar{\sigma}^n)^2 &= \frac{\bar{\delta}^n - (\bar{\beta}^n)^2}{1 + \bar{\lambda}^{n-1}}
\end{aligned}$$

Step 3. Evaluate the stepsizes for the current iteration (if $n > 1$).

$$\alpha^n = 1 - \frac{(\bar{\sigma}^n)^2}{\bar{\delta}^n}$$

Step 3a. Update the coefficient for the variance of the smoothed estimate.

$$\bar{\lambda}^n = \begin{cases} (\alpha^n)^2 & \text{if } n = 1 \\ (1 - \alpha^n)^2 \bar{\lambda}^{n-1} + (\alpha^n)^2 & \text{if } n > 1 \end{cases}$$

Step 4. Smooth the estimate.

$$\bar{\theta}^n = (1 - \alpha^n)\bar{\theta}^{n-1} + \alpha^n \hat{X}^n$$

Step 5. If $\bar{\theta}^n$ satisfies some termination criterion, then stop. Otherwise, set $n = n + 1$ and go to **Step 1**.

E.2. Hyperparameter Search

The parameters used in Figure 8.2 were obtained by running a hyperparameter search which looked for hyperparameters which led to the smallest average error at the end of each phase (i.e. at t=249 and t=499). The search used a Gaussian Process optimizer with 500 iterations. We found that there generally tends to be a large, flat region in parameter space with reasonably "good" parameters, as is evidence by the large regions of purple in Figure E.1.

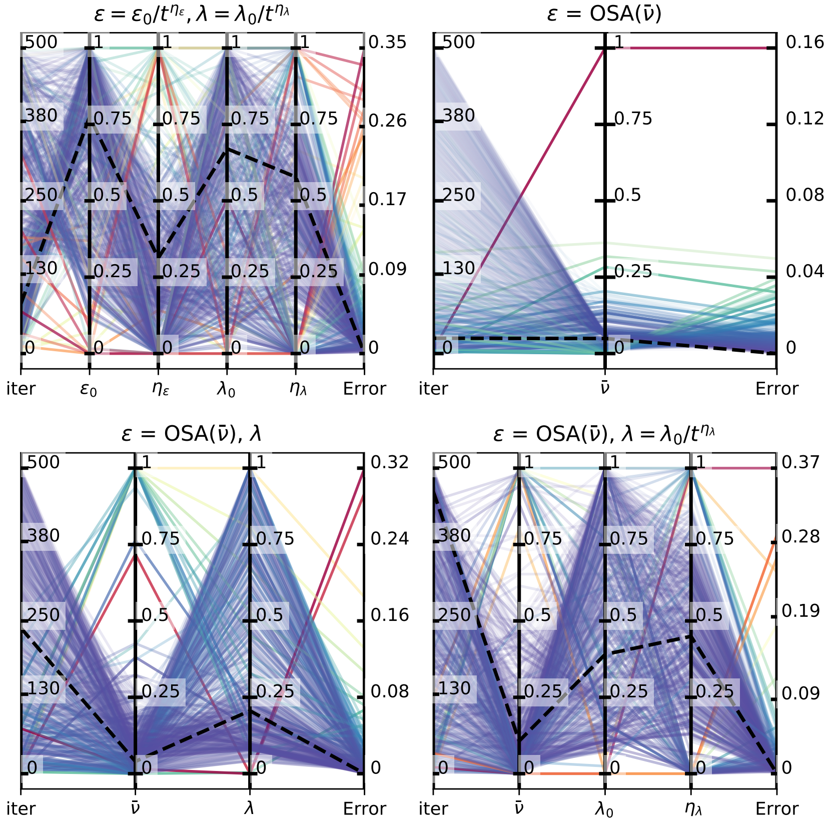


Figure E.1.: Parallel-Coordinates plot visualizing the hyperparameter search for the best-converging parameters. Each plot corresponds to one of the lines in Figure 8.2 (excluding the first, $1/\sqrt{t}$, which has no hyperparameters). In each plot, the leftmost axis represents the iteration in the Gaussian-Process search, the middle axes represent the hyperparameter values, the rightmost axis represents the error resulting from that set of hyperparameters (in this case the mean of the distance from the true fixed point at the end of each phase - see Figure 8.2), and lines are also colour-coded to indicate the error (purple is low and red is high). The black dotted line indicates the final selected set of hyperparameters. Large purple regions in these plots indicate insensitivity to hyperparameter values.

E.3. Details on MNIST Experiments

For the MNIST experiments, we ran the hyperparameter optimization for both scheduled annealing and OSA + predictive coding. We optimized the hyperparameters based on the validation error after 1 epoch of training. Figure E.2 visualizes the hyperparameter search. Because the two performed similarly and the latter had fewer parameters, we chose OSA + predictive coding for the experiment in Figure 8.3.

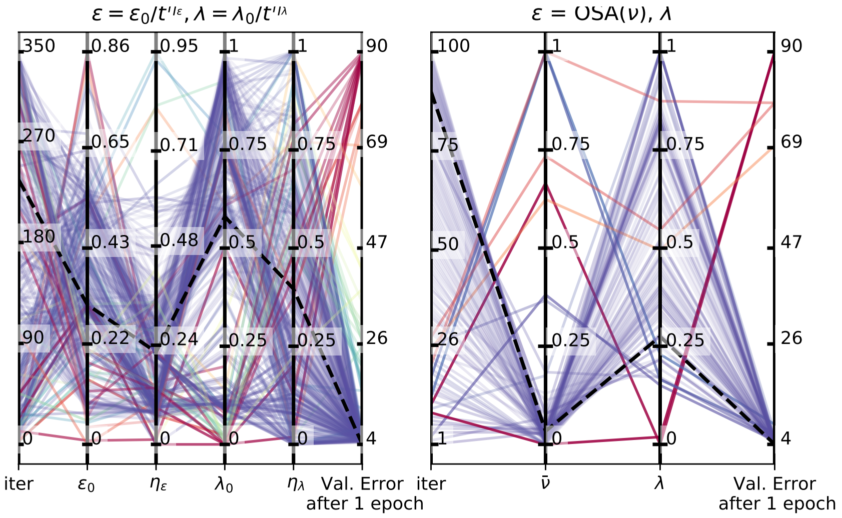
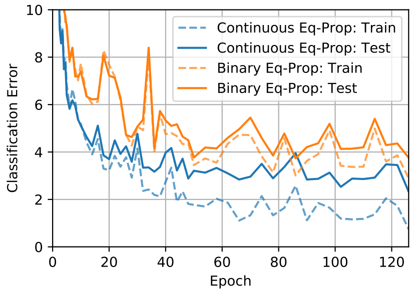


Figure E.2.: Parallel Coordinates plot for the hyperparameter search for the 1-layer MNIST network. See Figure E.1 caption for explanation of the plot. The hyperparameter search aimed to minimize the validation-set error after 1 epoch. **Left**: The search using scheduled step-size / predictive coding. **Right**: The search using OSA and fixed predictive-coding. Because these behaved similarily, we used the OSA + predictive coding (right) which had fewer parameters, with the optimal parameters found here model for Figure 8.3.

We also experimented with a deeper network,with 3 hidden layers of 500 units. The following plot shows the learning curve of Equilibrium Prop. The "binary" network is run with OSA, with parameters $\lambda = 0.771, \bar{\nu} = 0.686$ found in a hyperparameter search.



A final note is that it is possible to get our spiking network to perform on par with Equilibrium Prop *without* increasing the number of convergence steps (T^+ and T^-). The trick is to save a "checkpoint" of the state of the neurons (including the states of all encoders and decoders), at the end of the negative phase (see the "splitstream" parameter in the code). We then allow both the negative phase and the positive phase to proceed independently from this checkpoint for T^+ steps, to achieve the s^- and s^+ used in the update rule in Equation 2.12. This results in a much lower variance gradient estimate because the noise due to the internal states of the encoders/decoders cancels out in the constrastive update. We do not use this in any of our experiments because the notion of saving a state "checkpoint" which you can return to is biologically unrealistic.