Dynamic Working Memory in Recurrent Neural Networks

Alexander Atanasov

Research Advisor: John Murray

Physics 471

Fall Term, 2016

Abstract

Recurrent neural networks (RNNs) are physically-motivated models of biological neuronal circuits that can perform elementary computations. The relationship between the structure of a network's connectivity and its resulting dynamics is difficult to determine in all but the simplest of cases. In the past, we have studied the dynamics arising from RNNs that are trained to perform simple discriminatory tasks with one input neuron feeding in information. Generalizing this to discrimination of multiple input neurons requires increasing computational power and flexibility in the way the models are formulated. We present a new package, KerasCog for modeling biological neural networks that allows for increased computational speed, parallelization compatibility, and ease of use for a general audience of researchers in computational neuroscience. We present two examples of networks built using KerasCog that discriminate based on input.

Contents

1 Introduction to Neural Networks

1.1 Feedforward and Recurrent Networks

We begin with an exposition to the field of ANNs, adopted from the previous semester's final review. The study of artificial neural networks (ANNs) has existed already for several decades and finds applications in fields ranging from image detection to natural language processing. Beyond these applications, ANNs provide a looking glass into the ways that neuronal systems in the brain can perform elementary computations.

Within the machine learning community, neural networks are built by feeding data into an initial layer of input neurons, connected to a series of "hidden" layers before returning an output from the output layer [1], as below.

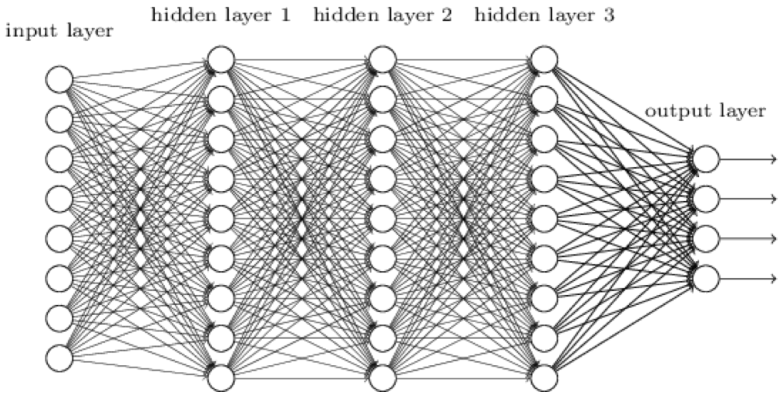

Figure 1.1: A Feedforward Neural Network

These are multilayer systems, with several levels of neuron families and no connectivity among the same level, but very high connectivity between successive levels. The inputs of neurons at level i depend only on the inputs for the neurons at level $i-1$, and their firing will only effect neurons at level $i+1$.

A neuron will have a firing rate r_i depending on whether it is receiving enough current to pass some threshold. This current comes from all the neurons going into it. That is, for each neuron i, a neuron j going into i contributes an amount of current $W_{ij}r_j$, where W_{ij} is the *weight of connectivity* between i and j. This can be negative, in which case we would have that neuron j *inhibits* neuron i. The firing rate of neuron i is then some positive function of the neuron's current $r_i = [x_i]_+$ (e.g. a sigmoid or a delayed linear function with some threshold) [2].

There is some output function z_ℓ^* for each output neuron i, that the neural network must be trained to produce. That is, the goal is to minimize $\mathcal{L} = ||z_\ell^* - z_\ell||$. How do

we minimize this error? We see how we'd have to adjust the current immediately going in to the output layer by adjusting those weights so that the error function decreases the most rapidly. But the current in each penultimate node is dependent on the layer before *that*, so we must see how to adjust those weights as well in order to minimize the total error. The total error change is then expressed (through the chain rule) in terms of all the weight changes from the final to the initial layer. This is the principle of backpropagation.

Once we have the changes in error expressed in terms of the changes of weights, we find the direction which minimizes the error the fastest. This is the method of steepest gradient descent (which will be explained more deeply in the following sections).

The human brain rarely involves networks like the one above [2]. Firstly, there are no dynamic elements to these networks. The inputs u_k are fixed, independent of time, and so are the outputs z_ℓ. Biologically, we expect both input and output to be time-varying currents $u_k(t)$ and $z_\ell(t)$

The networks above are called "feedforward", but they are not recurrent. The picture below is a section of a neuronal network from a mouse brain [2].

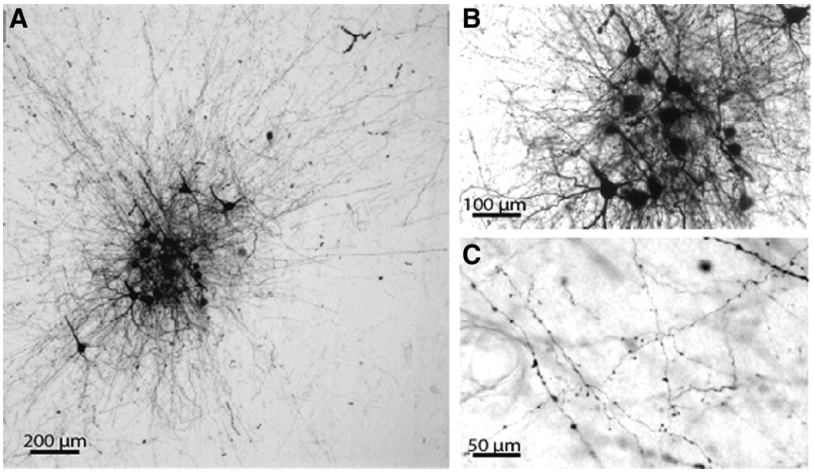

Figure 1.2: A Recurrent Neural Network

Note that in this picture, the networks do not fit together in separable layers but instead seem to bunch up together in a highly connected set of nodes. Moreover, the graph that represents the connectivity of such a neural network may involve multiple cycles, so that expressing it in terms of disjoint layers is impossible. Recurrent neural networks (RNNs) consist of connectivity among all neurons in the network without separating the neuronal connections into distinct sectors as in the first picture.

In certain parts of the brain, there can be a mix of feedforward networks and RNNs, where the network consists of sparsely connected sectors of RNNs, each RNN feeding forward into the next. There is biological evidence for this phenomenon [2], but in this paper we will focus on only RNNs that are densely connected.

A reasonable model for neural networks is given in terms of relating the current in

a given recurrent neuron x_i to a linear inhomogeneous differential equation. This equation would represent exponential decay of activity were it not for the inhomogeneous terms representing the inputs from the other spiking networks.

$$\tau \frac{dx_i}{dt} = -x_i + \sum_{j=1}^{N_{\text{rec}}} W_{ij}^{\text{rec}} r_j + \sum_{k=1}^{N_{in}} W_{ik}^{\text{in}} u_k + \sqrt{2\tau\sigma^2}\xi_i \tag{1.1}$$

$$r_i = [x_i]_+ \tag{1.2}$$

$$z_\ell = \sum_{i=1}^{N} W_{\ell i}^{\text{out}} r_i \tag{1.3}$$

The term u_i is the input while z_i is the output. The last term, ξ_i is Gaussian noise, which is both biologically and computationally significant. The output function $z_\ell(t)$ is obtained from an output weight matrix $W_{\ell i}^{\text{out}}$ acting on the recurrent network.

1.2 Training Neural Networks: Backpropagation and SGD

To train a neural network means to appropriately adjust the weight matrices $W_{ij}^{\text{rec}}, W_{ij}^{\text{in}}, W_{ij}^{\text{out}}$ so that for the given input $u_k(t)$ we get an output $z_\ell(t)$ that is as close as possible to the desired $z_\ell^*(t)$. Since firing rates recursively depend on previous neurons at past time-steps [4], we will have to employ backpropagation as in the feedforward case.

This time it is different: a neuron's own fired current at a given time t may end up influencing it at a future time t' [5]. This forms feedback loops for each individual neuron. Regardless, to minimize the error function $\mathcal{L}(W_{ij})$, known also as the **objective function** or the **loss**, with respect to the weights, we employ the method of stochastic gradient descent. This method is a variation on the more simple method of gradient descent: given a point in the weight space, calculate the gradient of the error function (it points orthogonal to the level curves of the error) and move against the gradient (so as to go in the direction of greatest decrease) [5].

More formally, we say that a network is defined by a set of parameters $\boldsymbol{\theta}^i$ (in this case the weights of the neuronal connections) and we want to minimize the objective function:

$$\mathcal{L}(\boldsymbol{\theta}) = \sum_{t,\ell} [z_\ell(t) - z_\ell^*(t)]^2. \tag{1.4}$$

At each step in training, the parameters $\boldsymbol{\theta}^i$ are adjusted from the previous step by calculating the gradient $\nabla\mathcal{L}$ with respect to total derivatives of each of the $\boldsymbol{\theta}^i$ and applying:

$$\boldsymbol{\theta}^i = \boldsymbol{\theta}^{i-1} + \delta\boldsymbol{\theta}^{i-1},\ \delta\boldsymbol{\theta}^i = -\eta \frac{d}{d\boldsymbol{\theta}^i}(\mathcal{L}) \tag{1.5}$$

this total derivative can be calculated accurately through back-propogation through time[1].

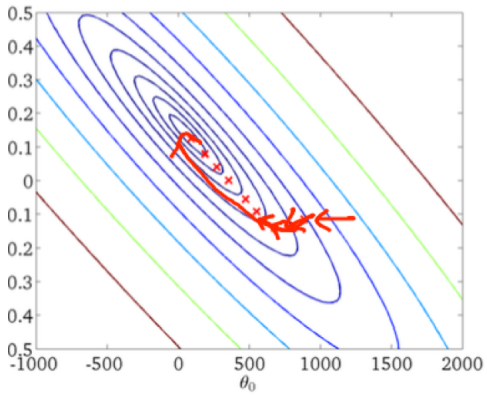

Figure 1.3: The Method of Gradient Descent in 2D

We continue this process iteratively to converge on a minimum. Note however that this minimum may only be a local minimum and thus not be good enough to sufficiently minimize the error. The stochastic element is meant to avoid this. By adding an element of randomness, we are able to avoid long-term convergence to a local but not global minimum.

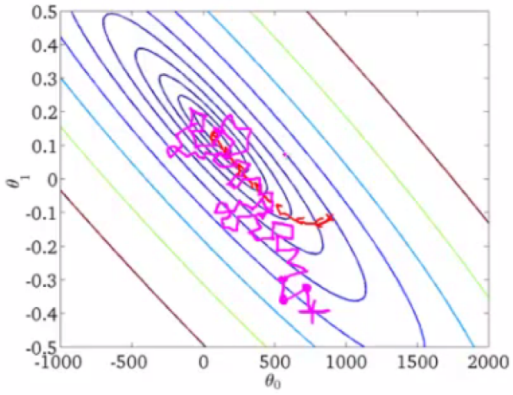

Figure 1.4: The Method of Stochastic Gradient Descent in 2D

The idea is that if we are trapped in only a local minimum, with a better one nearby, this randomness can "bump" us out, into the lower minimum.

[1]Note that these derivatives are not numerical, but rather rely on symbolic differentiation, pre-implemented on the built-in functions of a given machine learning library

1.3 Adaptive Moment Estimation: "Adam"

The method of stochastic gradient descent can be greatly improved by using so-called "adaptive" methods to predict, using past data, what direction we should change our parameters in order to get maximal decrease in our objectie function.

SGD training has difficulty with "ravines", which are places where there is much higher gradient fluctuation along one dimension than in others [3]:

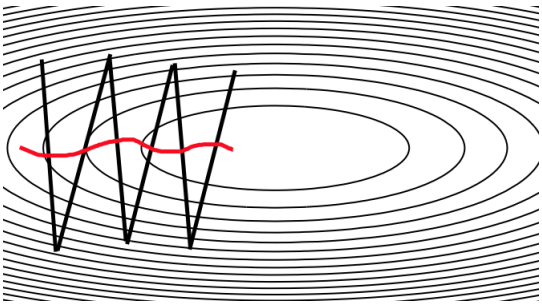

Figure 1.5: Illustration of the adaptive method, Adam, in red against the naive SGD approach in black

By keeping track of the gradient mean and standard deviation in past timesteps, the adaptive moment estimation method, **Adam**, makes an informed guess about the path being traced through weight space that minimizes the loss. This is done by taking an exponentially weighted sum of the past gradient means and gradient variances.

Specifically, given a calculated gradient g_t at time t the estimate for the first moment (mean), m and second moment (mean of the square), ν for the correct direction to move at timestep t is taken as

$$\begin{aligned} m_t &= \beta_1 m_{t-1} + (1-\beta_1) g_t \\ \nu_t &= \beta_2 \nu_{t-1} + (1-\beta_2) g_t^2 \end{aligned} \tag{1.6}$$

and β_1, β_2 are taken as parameters, less than unity, that determine the decay rate of the past gradients taken into account in these weighted averages. By keeping track of this information, we can increase the training rate, often orders of magnitude [3].

2 The KerasCog Package

2.1 Introduction

Over the spring semester, a great deal of the computational work done was through the *pycog* package developed by Francis Song et al. [5]. This package employed methods of SGD and the *Theano* machine learning library to build biologically-realistic neural networks and train them to produce results seen in data from cognitive tasks.

Multi-input discrimination tasks are particularly difficult to train networks on, and require an increase in computational power given either by a more powerful descent method (such as Adam over SGD) or access to parallelization (i.e. using graphic-processing units to speed up matrix manipulations in training). Unfortunately, *pycog* has not implemented any more advanced method of training past SGD, and is unlikely to be updated in the future. Moreover, the *Theano* library's function used in back-propagation by *pycog*, known as the **scan**, meant to back-propagate for multiple steps backwards in time, is not GPU-compatible.

Rather than modifying *pycog* and rewriting the descent method to be compatible with GPUs and Adam, we chose to work from the bottom up and implement a package of our own using the *Keras* machine-learning package.

Keras is a "wrapper" package, meaning that it is a higher-level package that already has implemented various networks, interfaces, and training methods for machine learning purposes. Keras uses either *Theano* or Google's *TensorFlow* to perform the matrix manipulations involved in training networks, and can swap out one for the other easily. Switching over to such a package, rather than starting from scratch using *Theano* or *Tensorflow* has several advantages:

1. We can directly appeal to Keras' RNN classes without having to build them from scratch,
2. We can use a much wider variety of descent methods and criteria when training, including Adam.
3. All methods in Keras are tested to be GPU-compatible.
4. Using Keras greatly decreases the amount of code necessary to build a research package for studying cognitive processes, as most of the low-level work is implemented within the package and done in the "background"

The latest release of KerasCog can be found and downloaded at

https://github.com/ABAtanasov/KerasCog.

2.2 Enforcing Biological Constraints in Keras RNNs

The RNN class in Keras involves a time-dependent input fed into a dense recurrent network to produce a time dependent output. The equations of motion for a single neuron are

$$x(t + \Delta t) = \sum_{j}^{N_{rec}} W_{ij} r_j(t) + \sum_{k=1}^{N_{in}} W_{ik}^{in} u_k(t) \tag{2.1}$$

moreover the weights W_{ij} for neuron j are not constrained to all have the same sign in general, unlike in biology where **Dale's Law** states that a neuron must be either inhibitory (negative weights) or excitatory (positive weights) with a roughly 80/20 ratio of excitatory to inhibitory neurons.

This type of network is more optimal for machine learning than the biologically-motivated network described previously. Our modifications to the Keras RNN class involved four changes:

1. Input Noise
2. Recurrent Noise
3. Neuron Decay in Time
4. Dale's Law

The input noise was the easiest to implement, by directly adding Gaussian noise to the input neurons' signal $u_i(t)$ as it was fed in to the network. Recurrent noise is implemented by connecting a "gaussian noise network" to be fed in to the RNN together with the input, supplying the randomness at each timestep.

The neuron decay was more challenging, and required us to build a new **LeakyRecurrent** class, deriving all the methods from the Keras **Recurrent** class, but changing the step in time so as to implement the same equations of motion as in Equations (1.1)-(1.3). The timestep function is a particularly sensitive one to build, as it will be symbolically differentiated by Keras' backpropagation methods. For this reason, we could not have implemented noise by naively adding it at each time-step, because such a function has no symbolic derivative.

Lastly, Dale's law was implemented by adding a non-trainable matrix

$$D = \text{diag}[(1, 1, \ldots, 1, -1, \ldots, -1)] \tag{2.2}$$

with an 80/20 ratio of excitatory to inhibitory neurons. The equations of motion then replace W_{ij}^{rec} by $D|W_{ij}^{\text{rec}}|$ so that the effective weights are consistently positive or negative for each recurrent neuron.

3 Discrimination of Inputs

Our long-term goal is to use KerasCog to train neural networks for tasks of input discrimination, in order to perform analysis to determine how working memory is stored in these dynamical systems.

We can make the pulses be different frequency, or we can make them come from different neurons. These two tasks, while seemingly simple in structure, lead to widely different dynamics and training behaviour. In the past semester, we heavily studied the structure of frequency discrimination, but were unable to train a very large variety of networks to perform input discrimination-based memory tasks. Here, we use the increased computational power of KerasCog to train networks to perform such input discrimination. The first example we give is the FlipFlop task:

3.1 The FlipFlop Network

We have two input neurons A and B, one output neuron C, and a time series consisting of n "turns". At each turn, we have a firing period followed by a "quiet" period. During the firing period one of the two input neurons will fire. If neuron A has fired, the output neuron will stay close to zero, with no firing activity. If neuron B fires, the output neuron fires with activity close to 1. Below is an implementation of this task using the Keras built-in RNN

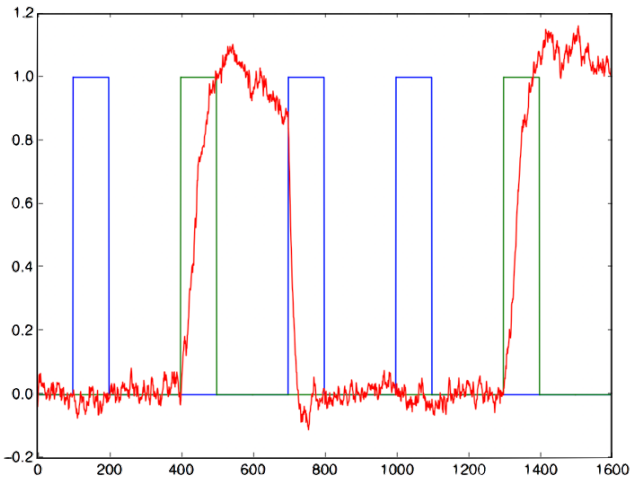

Figure 3.1: A Keras neural network

Upon implementing leak in this built-in network, we can see it manifest itself in the output neuron's activity.

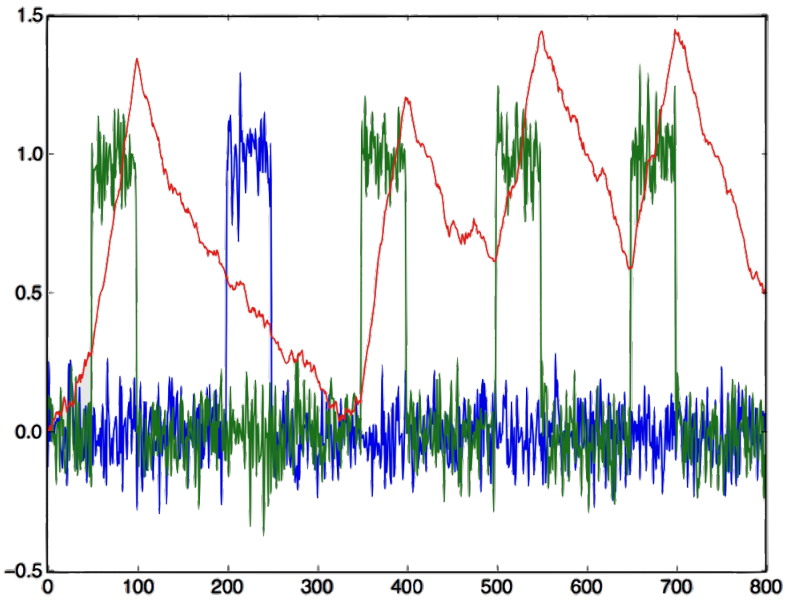

Figure 3.2: Keras neural network with Dale's law, leak, and noise implemented

The above figure is generated from the KerasCog class of **LeakRecurrent** neural networks.

3.2 Results of Training Times

The amount of time required to train a network to perform the FlipFlop task heavily depended on the biological constraints enforced. With few biological constraints enforced (only leak and random noise on the input), the networks trained successfully in a matter of seconds, much in part due to the very fast speedup given by Adam.

Upon enforcing Dale's law, however, the training time became significantly longer, requiring much more time for the network to minimize its loss function. Moreover, there appears to be an asymptote in loss past which it would require a significant of time to reduce the error. This issue is indicative of Dale's law being a significant constraint on the network that makes it more difficult to find solution networks to a given input discrimination task.

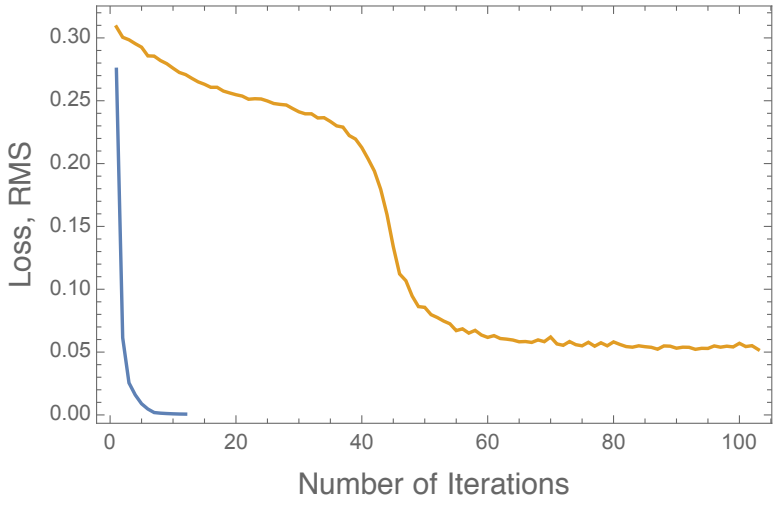

Figure 3.3: Training times for RNNs with Dale's law and recurrent noise enforced (yellow) vs. without (blue)

4 Future Steps

The next steps will involve using Keras' increased computational power together with the Yale HPC clusters to train networks to build longer term memory for input discrimination tasks. We have gotten around the issue of GPU-compatibility encountered last semester.

The remainder of this research will then center on writing up visualization tools for Keras from which the dynamics of the RNNs can be understood. Such visualization tools will range from simply seeing each neuron's activity over the time of the task, and seeing the connectivity matrix of a given network, to more advanced PCA-inspired and spectral methods. Among these methods will be looking at the eigenvalue spectrum of the connectivity matrix, and studying both dynamic and steady-state behaviour in the networks based on the interpretation of these eigenvalues.

The FlipFlop task in particular, lends itself to be analyzed in terms of attractor states of the network output. We can view the network as a point moving through the phase space of activity states, that is influenced by the external input to move into one of two "troughs" or attractor states. Such analysis, combined with the ability to reduce the high-dimensional system to a simple 2-D figure could prove to be a powerful and unique visualization tool in KerasCog for understanding dynamics.

References

[1] Juan C. Cuevas-Tello, Manuel Valenzuela-Rendón, and Juan Arturo Nolazco-Flores. A tutorial on deep neural networks for intelligent systems. *CoRR*, abs/1603.07249, 2016.

[2] Peter Dayan and L. F. Abbott. *Theoretical Neuroscience: Computational and Mathematical Modeling of Neural Systems*. The MIT Press, 2005.

[3] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014.

[4] Jing Li, Ji-hang Cheng, Jing-yuan Shi, and Fei Huang. *Advances in Computer Science and Information Engineering: Volume 2*, chapter Brief Introduction of Back Propagation (BP) Neural Network Algorithm and Its Improvement, pages 553–558. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[5] H. Francis Song, Guangyu R. Yang, and Xiao-Jing Wang. Training excitatory-inhibitory recurrent neural networks for cognitive tasks: A simple and flexible framework. *PLoS Comput Biol*, 12(2):1–30, 02 2016.