Percieving Circadian Gene Expression through Artificial Neural Networks

Damian Sowinski,[1,*] Alexander Crowell,[2,†] Jack Holland,[3,‡] and Rawan Al Ghofaili[3,§]

[1]*Center for Cosmic Origins*
Department of Physics and Astronomy, Dartmouth College
[2]*Department of Biology, Dartmouth College*
[3]*Department of Computer Science, Dartmouth College,Hanover, NH 03755, USA*

(Dated: Last Updated: Sunday 9th November, 2014, 14:51)

Science is done!

INTRODUCTION

ARTIFICIAL NEURAL NETWORKS

In this section we go over the theoretical background of how to build an ANN, develop the formalism behind the backpropogation algorithm, and discuss possible candidates for tunable hyperparameters/functions used. We will develop both the feed forward and back prop algorithms for completeness.

ANNs are a supervised machine learning algorithm used for classification problems. Given some labeled set of data $\mathcal{D} = \{(\vec{\mathbf{x}}, \vec{\mathbf{y}})_i\}_{i=1}^{D}$, where to each multidimensional datum $\vec{\mathbf{x}}$ there is an associated multidimensional label $\vec{\mathbf{y}}$, the ANN learns from $\mathcal{D}$ how to place labels on new data. Processing a datum by the network is accomplished via the *feed forward* algorithm, while training is done through the *back propogation* algorithm.

The basic building block of an ANN is the perceptron. N signals go into the perceptron, are weighted, summed, and then put through some activation function to create an output signal:

$$\text{output} = \theta(\sum_{i=0}^{N} w_i \times \text{input}_i) \tag{1}$$

The form of the **activation function**, θ is typcally chosen to be one that gives a value of 1 for arguments larger than some threshhold, and 0 otherwise. The backpropogation algorithm requires this function to be both differentiable and invertible, so typical choices include the logistic, arctan, and hyperbolic tangent functions. Note also that the sum begins at 0, leading to $N+1$ inputs. The 0^{th} input is always taken to be 1, and its corresponding weight is known as the **bias**.

We can stack perceptrons into layers, and in turn stack those together so that the inputs of the l^{th} layer are the outputs coming from the $(l-1)^{th}$ layer. This is the basic form of an ANN. We need only specify the architecture of the ANN (number of layers/{Input dimensionality,number of perceptrons in each layer}), which will be $L/\{N_0, N_1, \ldots, N_L\}$. Note that N_0 represents the dimensionality of the input signal, and N_L is the dimensionality of the output layer. Between each pair of layers there is a corresponsing weight matrix, $\mathbf{w}^{(l)} : \mathbb{R}^{N_{l-1}} \to \mathbb{R}^{N_l}$, the components of which are $w_{ij}^{(l)}$, where l is the target layer, i is the index of the *to* perceptron in the target layer, and j is the *from* perceptron in the domain layer. We denote the set of all these matrices $\mathbf{W}$, and refer to this set as the **architecture** of our ANN. With these definitions in hand, we can write the output of the i^{th} perceptron in the l^{th} layer in compact form:

$$x_i^{(l)} = \theta(\sum_{j=0}^{N_{l-1}} w_{ij}^{(l)} x_j^{(l-1)}) \tag{2}$$

where once again,, the $w_{i0}^{(l)}$'s represent biases.

With the architecture in place, we can now easily state the feed forward algorithm. It is nothing more than allowing an input signal propagate through the network, and eventually pop out at the end.

Algorithm 1 Feed Forward

procedure FeedForward($\vec{\mathbf{x}}^{(0)}$,$\mathbf{W}$)

$L \leftarrow$ number of layers in $\mathbf{W}$
$N_l \leftarrow$ number of perceptrons in $\mathbf{w}^{(l)} \in \mathbf{W}$
$N_0 \leftarrow \dim(\text{domain}((\vec{\mathbf{x}}^{(0)})$

for $l = 1, 2, \ldots, L$
  $x_0^{(l-1)} \leftarrow 1$
  for $i = 1, 2, \ldots, N_l$
    $x_i^{(l)} \leftarrow \theta(\sum_{j=0}^{N_{l-1}} w_{ij}^{(l)} x_j^{(l-1)})$ ▷ and store
  end
end

return $\mathbf{X}= (\vec{\mathbf{x}}^{(0)}, \vec{\mathbf{x}}^{(1)}, \ldots, \vec{\mathbf{x}}^{(L)})$

end procedure

This is all fine and dandy, but what we want is to be able to train the network based on the known labels. We need to compare the output of the network with the known label, define an error based on that comparison, and then alter the weights in the network so as to decrease that error. Furthermore, when the weights are changed, we do not want them to take on any values (one can imagine one weight becoming so high that it dominates the flow of signals in the ANN), so we must also regularize the way in which the error function is causing

the weights to change. These statements can be made more precise.

Initialize an ANN with randomized weights. Take a datum $(\vec{\mathbf{x}}^{(0)}, \vec{\mathbf{y}})$ and have the ANN process the input to get an output $(\vec{\mathbf{x}}^{(L)}, \vec{\mathbf{y}})$. We define the cost of the function as an explicit function of the output, the **error**, and an explicit function of the weights, the **regularizer**:

$$C(\vec{\mathbf{x}}^{(L)}, \vec{\mathbf{y}}, \mathbf{W}) = E(\vec{\mathbf{x}}^{(L)}, \vec{\mathbf{y}}) + R(\mathbf{W}) \tag{3}$$

This factorization of the arguments is a bit deceptive, since the output is an implicit function of the weights of the network. This fact will be crucial as we start exploring weight-space and will need to take gradients of the cost with respect to the weights. Our goal will be to decrease the cost of our ANN at each iteration by changing the weights of the network. Over the course of many epochs, the network will learn the patterns in the data, and we will have minimized the cost. So, how do we choose the change in weights to achieve this at each iteration?

We want:

$$C(\vec{\mathbf{x}}^{(L)}, \vec{\mathbf{y}}, \mathbf{W} + \delta\mathbf{W}) \leq C(\vec{\mathbf{x}}^{(L)}, \vec{\mathbf{y}}, \mathbf{W})$$

Expanding the RHS to first order in $\delta\mathbf{W}$ we have:

$$\nabla C \cdot \delta\mathbf{W} \leq 0$$

where the gradient is taken with respect to the weights. We can easilly satisfy this equation if we choose:

$$\begin{aligned} \delta\mathbf{W} &= -\eta \nabla C \\ &\Downarrow \\ \delta w_{ij}^{(l)} &= -\eta \frac{dC}{dw_{ij}^{(l)}} \end{aligned} \tag{4}$$

This is our **learning rule**. The hyperparameter η is called the **learning rate**. Taking the gradient of the regularizer is simple, since it is an explicit function of the weights. To take the gradient of the error, however, we will have to employ some mathematical trickery. We have to use the fact that the output depends on the weights, making the error an implicit function of the weights.

First off we will break up the reweighing of the ANN layer by layer. So we will start at the output layer, and move backward. To proceed we need one result on how to differentiate an output neuron with respect to weight:

$$\begin{aligned} \frac{dx_n^{(l)}}{dw_{ij}^{(l)}} =& \frac{d}{dw_{ij}^{(l)}} \theta\Big(\sum_{k=0}^{N_{l-1}} w_{nk}^{(l)} x_k^{(l-1)}\Big) \\ =& \theta' \circ \theta^{-1}(x_n^{(l)}) \delta_{ni} x_j^{(l-1)} \end{aligned} \tag{5}$$

The gradient of the error can then be written:

$$\frac{dE}{dw_{ij}^{(L)}} = \sum_{k=0}^{N_L} \tag{6}$$

Algorithm 2 Back Propogation

procedure BACKPROP($\mathbf{X}$, $\vec{\mathbf{y}}$, $\mathbf{W}$)

$L \leftarrow$ number of layers in $\mathbf{W}$
$N_l \leftarrow$ dimensionality of each $\vec{\mathbf{x}}^{(l)} \in \mathbf{X}$

for $l = L, L-1, \ldots, , 2, 1$
  for $i = 1, 2, \cdots, N_l$
    if $l = L$
      $\delta_i^{(l)} \leftarrow \phi(x_i^{(l)}) \nabla_i E$
    else
      $\delta_i^{(l)} \leftarrow \phi(x_i^{(l)}) \sum_{j=0}^{N_{l+1}} \delta_j^{(l+1)} w_{ji}^{(l+1)}$
    end
    for $j = 1, 2, \ldots, N_{l-1}$
      $w_{ij}^{(l)} \leftarrow w_{ij}^{(l)} - \eta \left(\nabla_{ij}^{(l)} R + \delta_i^{(l)} x_j^{(l-1)} \right)$
    end
  end
end

return $\mathbf{W}$

end procedure

Algorithm 3 Train Network

procedure TRAIN($\mathcal{D}$, $\mathbf{W}$, numEpochs)

for $i = 1, 2, \ldots,$ numEpochs
  randomize($\mathcal{D}$)
  for $d \in \mathcal{D}$
    $\mathbf{X} \leftarrow$ FeedForward$(d_1, \mathbf{W})$
    $\mathbf{W} \leftarrow$ BackProp$(\mathbf{X}, d_2, \mathbf{W})$
  end
end

return $\mathbf{W}$

end procedure

Algorithm 4 Cross Validation

procedure CROSSVAL($\mathcal{D}$, $\mathbf{W}$, numEpochs, N)

randomize($\mathcal{D}$)
partition $\mathcal{D} = \mathcal{D}^1 \sqcup \mathcal{D}^2 \sqcup \cdots \sqcup \mathcal{D}^N$
for $i = 1, 2, \ldots, N$
  $\mathbf{W} \leftarrow$ Train$(\mathcal{D} \backslash \mathcal{D}^i, \mathbf{W},$ numEpochs$)$
  MSE$_i \leftarrow \|\mathcal{D}_2^i -$ FeedForward$(\mathcal{D}_1^i, \mathbf{W})\|_2$
end
CVerror $\leftarrow$ mean$(\{MSE_i\})$

return CVerror

end procedure

* Electronic address: Damian.Sowinski.GR@dartmouth.edu

† Electronic address: Alexander.M.Crowell.GR@dartmouth.edu

‡ Electronic address: Jack.E.Holland.GR@dartmouth.edu

§ Electronic address: Rawan.al.Ghofaili.GR@dartmouth.edu