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 Genetics, Geisel School of Medicine, Dartmouth College*
[3]*Department of Computer Science, Dartmouth College,Hanover, NH 03755, USA*

(Dated: Last Updated: Monday 24th November, 2014, 09:10)

The circadian clock regulates many critical processes, notably including metabolism and cancer suppression. In order to improve both our understanding of the clock and drug discovery and delivery for the many disease processes it interacts with the ability to accurately classify genes as circadianly regulated based on expression data is critical. To improve on existing classification of circadian gene expression we employed an artificial neural network and compared it's performance to the current standard algorithm JTK_{CYCLE}. We employ a feed forward network, and examine the architectural constraints to optmize the ability of the network to classify circadian experession. We discuss the theoretical construction of an artificial neural network (ANN) to better understand design choices made in this work. Preliminary results are found and possible modifications to the network are discussed.

CIRCADIAN RHYTHMS

Circadian Clocks are an adaptive response to the evolutionary necessity of anticipating day night cycles. They occur widely in higher eukaryotes and sparsely in fungi and cyanobacteria. Because of the widespread and longstanding evolutionary history of organisms specializing their activity to a specific phase of the day night cycle, circadian clocks, where present, have been found to regulate almost every biological process imaginable. Notable among these are the cell cycle, DNA damage repair and metabolism, processes central to the majority of human disease [1]. Perturbation of the clock upsets these processes as well and has been shown to contribute to disease [2]. One of the most important recent trends in circadian research has been the interaction of the circadian cycle with disease treatment, specifically the timed delivery of medicine [3]. Additionally the circadian clock's modulation of such a wide variety of gene and protein expression along with the diverse processes employed to achieve that modulation provides an excellent opportunity for the study of gene regulation at every stage from transcription to degradation [4, 5].

The current state of the art in the identification of Circadian gene expression is the JTK_{CYCLE} algorithm (JTK) [6]. JTK uses a nonparametric test for monotonic orderings in the data over all possible period lengths and phases and for high ($\tilde{2}$hr) resolution data it provides high sensitivity, specificity and computational efficiency. JTK and earlier approaches including curve fitting [7], autocorrelation [8], Fourier analysis [**?** **?**] and Fisher's G Test [**?**] all fall short however as the resolution of data decreases. Due to technical challenges in the maintenance and synchronization of cells as well as sequencing costs, dataset resolution is currently a major limiting factor in the identification of circadian genes. Therefore an improved computational technique for identifying circadian genes in low temporal resolution data could rapidly provide knew knowledge from publicly available pre-existing datasets

ARTIFICIAL NEURAL NETWORKS

In this section we go over the theoretical background of how to build an ANN, develop the formalism behind the backpropagation 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 propagation* 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:

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

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 backpropagation 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, 1. It is nothing more than allowing an input signal propagate through the network, and eventually pop out at the end.

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:

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

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:

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

if we define $\phi = \theta' \circ \theta^{-1}$, the gradient of the error can then be written:

$$\begin{aligned}\frac{dE}{dw_{ij}^{(L)}} =& \sum_{k=1}^{N_L} \frac{\partial E}{\partial x_k^{(L)}} \frac{dx_k^{(l)}}{dw_{ij}^{(l)}} \\ =& \frac{\partial E}{\partial x_i^{(L)}} \phi(x_i^{(L)}) x_j^{(L-1)}\end{aligned} \tag{6}$$

We can generalize this for the gradient w.r.t. other layer weights by defining:

$$\delta_i^{(l)} = \begin{cases} \phi(x_i^{(l)}) \frac{\partial E}{\partial x_i^{(l)}} & l = L \\ \phi(x_i^{(l)}) \sum_j \delta_j^{(l+1)} w_{ji}^{(l+1)} & l < L \end{cases} \tag{7}$$

This is known as the δ-rule, and is used to write the learning rate in its compact form:

$$\delta w_{ij}^{(l)} = -\eta \left(\frac{dR}{dw_{ij}^{(l)}} + \delta_i^{(l)} x_j^{(l-1)} \right) \frac{dE}{dw_{ij}^{(L)}} = \sum_{k=0}^{N_L} \tag{8}$$

Take a datum $(\vec{\mathbf{x}}^{(0)}, \vec{\mathbf{y}})$ and process the input to get an output $d = (\vec{\mathbf{x}}^{(L)}, \vec{\mathbf{y}})$. Then determine the cost of this processed output and the weights of the network via:

$$C = C((\prod_l \theta \circ \mathbf{W}^{(l)})(\vec{\mathbf{x}}^{(0)}), \vec{\mathbf{y}}, \mathbf{W}) \tag{9}$$

This is all we need to define the back propagation algorithm, 2.

To go any farther we need to discuss the possible choices for the functions that define our network: the activation, error, and regularizer. We will now discuss each of these in turn.

Function Choices

Activation Function

The only constraints that we have on our choice is that the function be both differentiable and invertible. Typical choices are therefore any smooth strictly monotonic function. We should also specify the domain of the inputs. The list below has been normalized to the range $(0, 1)$, and we have included the backprop functions for completeness:

- Sigmoid (blue)

$$\begin{aligned} \theta(x) =& \frac{1}{1+e^{-x}} \\ \phi(x) =& x(1-x) \end{aligned} \tag{10}$$

- Hyperbolic Tangent (magenta)

$$\begin{aligned} \theta(x) =& \frac{1}{2}(1+\tanh x) \\ \phi(x) =& 2x(1-x) \end{aligned} \tag{11}$$

- Arctangent (gold)

$$\begin{aligned} \theta(x) =& \frac{1}{2}(1+\frac{2}{\pi}\arctan x) \\ \phi(x) =& \frac{1}{\pi}\cos^2\frac{\pi}{2}(2x-1) \end{aligned} \tag{12}$$

The activations are plotted in the left of figure . The qualitative differences are mostly with the arctan function, which requires incredibly high absolute value inputs to generate output close to 0 or 1. The corresponding backprop functions are plotted in the right of figure .

The main difference between the different activation functions is how they back propagate the error. Notice that the hyperbolic tangent function backprops twice as much error as the sigmoid. The arctan function has a change in concavity relative to the other two. This means that it pushes values close to 0 and 1 at a much slower rate than it does indeterminate ones. This is reflected in the activation function itself, note how the arctan function approaches the asymptotic values much more slowly than either the sigmoid or the hyperbolic tangent.

Error Function

The error function determines how comparisons are made between the output of the neural network and the labels. We need a function that grows with the discrepancy between these them. The most common choices are:

- Mean Square Error

$$E(\vec{\mathbf{x}}, \vec{\mathbf{y}}) = \frac{1}{2}||\vec{\mathbf{x}} - \vec{\mathbf{y}}||_2^2 \tag{13}$$

$$\frac{\partial E}{\partial x_i} = x_i - y_i \tag{14}$$

- Cross Entropy

$$E(\vec{\mathbf{x}}, \vec{\mathbf{y}}) = \sum_i^{N_0} y_i \log\sigma(x_i) + (1-y_i)\log(1-\sigma(x_i)) \tag{15}$$

$$\frac{\partial E}{\partial x_i} = -\, y_i \frac{\sigma'(x_i)}{\sigma(x_i)} \tag{16}$$

Cross entropy requires passing the output through an extra soft-max layer, ensuring that the total output is normalized in the L_1 norm. Soft max is accomplished via:

$$\sigma(x_i) = \frac{e^{-\beta x_i}}{Z}$$

where β is an inverse temperature. In the 0-temperature limit ($\beta \to \infty$), the soft max layer becomes a winner takes all layer. the denominator is the partition function:

$$Z = \sum_{j=1}^{N_L} e^{-\beta x_j}$$

In this case the gradient becomes:

$$\frac{\partial E}{\partial x_i} = \beta(\sigma_i(x_i) - y_i) \tag{17}$$

You can see that the form of the gradient is similar in both the MSE and cross-entropy, in that they are both defined by the difference between the output of the network and the label. Error minima are achieved when the label mathces the output. Cross entropy has the extra parameter of inverse temperature which can be used to fine tune learning rates.

Regularizer

One major problem with the ANN is that the values of the weights are unbounded. They can grow indiscriminantly based on pathological training data. To overcome

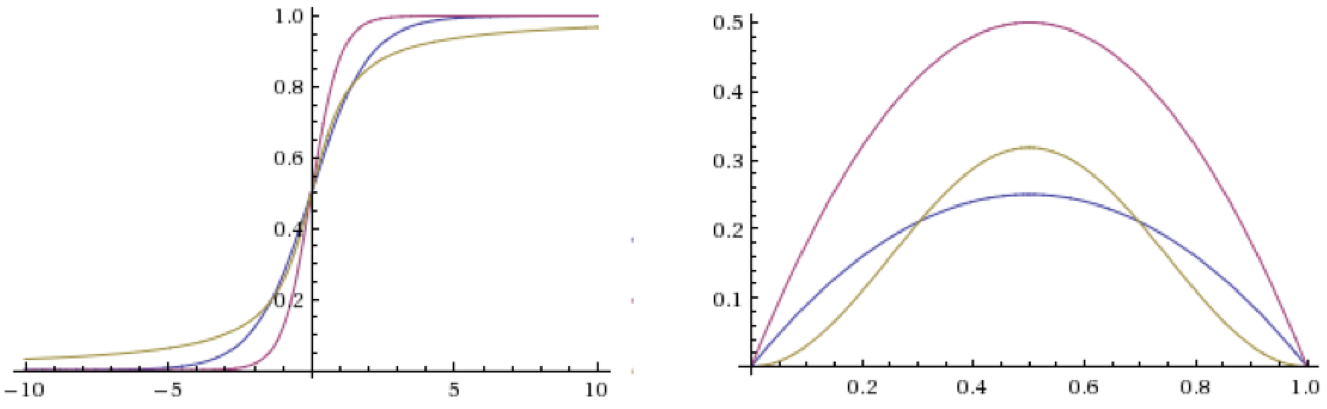

FIG. 1: On the left we have the activation functions plotted, and the back propagated functions on the right. The sigmoid, hyperbolic tangent, and arctangent are blue, magenta, and gold, respectively. Not that the main difference

this, regularizers are employed to create bounds and correlations between the weight vectors. Two types of regularizers are typically implemented, one which penalizes excessive weight, and another which enforces sparsity of connections, as in [9].

- Weight Penalization

$$R(\mathbf{W}) = \sum_l \sum_{i,j} |w_{ij}^{(l)}|^2 \tag{18}$$

- Sparsity Enforcement

$$R(\mathbf{W}) = \sum_l \sum_{i,j} S(|w_{ij}^{(l)}|) \tag{19}$$

One can think of the first regulator as being a multi-dimenional parabaloid, with the center of the 'bowl' at the origin, where all the weights vanish. As we move up the bowl, there is an associated cost with increasing the values of the weights too high.

For the latter, the function S is typically chosen from amongst $S(x) \in \{x, e^{-x^2}, \log(1+x)\}$. There are other possibilities, but the key here is that they are engineered in a very special way. If you imagine the hypersphere centered at the origin that has constant L_2 norm in the weights, then the regularizer minimizes at the intersections of that hypersphere and the axes in weight space. This means that networks that activate many weights in each layer will have a greater cost associated with them than those that activate fewer weights. In essence, it picks out amongst all the networks with the same variance those which have the least number of activated weights: the sparse ones.

Equations of motion

Is there an more intuitive way of thinking about what happens to the weight vectors during training? There is if we think of each training cycle as being a pair of moments connected by some tiny temporal interval, Δt. We can then write:

$$\begin{aligned} \delta\mathbf{W} =& \mathbf{W}(t+\Delta t) - \delta\mathbf{W}(t) \\ =& \frac{d\mathbf{W}}{dt}\Delta t + \frac{1}{2}\frac{d^2\mathbf{W}}{dt^2}\Delta t^2 + O[\Delta t^3] \end{aligned} \tag{20}$$

Rewriting the temporal interval as $\Delta t = 2m/\gamma$ and renrmalizing the learning rate as $\eta \to 2m\eta/\gamma^2$, we find the weights obey the differential equation:

$$m\frac{d^2\mathbf{W}}{dt^2} = -\gamma\frac{d\mathbf{W}}{dt} - \eta\nabla C \tag{21}$$

This equation provides the key to understanding the training of the network. Along with the learning rate, η, we also now have two new constants: the inertia m and a viscocity γ. Equation 21 is nothing more than Newton's second law of motion for trajectories of ANNs in weight space!

The left hand side is simply the mass times the acceleration. In computer science parlance the mass/inertia is typically refferred to as the **momentum**. The right hand side of the equation is just the net force acting on an ANN moving through weight space. The first term is a type of viscous friction, causing trajectories to slow down. In common parlance the viscocity, η, is reffered to as **weight decay**. The second term is the gradient of the cost function, which is acting like a potential energy in weight space, generating a force in the direction of steepest descent of the cost.

One can think of an ANN as a particle initialized somewhere in weight space. The ANN then begins to slide down the cost function, which is determined by both the error and the regularization. As it slides down the contours of the cost, it picks up speed, and can roll into and out of local minima, bouncing around as the force on it changes. The viscosity, however, will slowly drain kinetic energy fromt he ANN, decreasing its velocity, and eventually causing it to settle down in some minimum.

Implementing the equation of motion can be implemented by a second order staggered leap-from scheme:

$$w_{i+1} = w_i + \frac{\Delta t}{m} p_i \tag{22}$$

$$p_{i+1/2} = (1 - \frac{\gamma \Delta t}{m}) p_{i-1/2} - \eta \Delta t \nabla C_{i-1/2} \tag{23}$$

which is a simple extension of the first order back propagation algorithm 2 developed earlier.

The nice thing about about thinking about the ANN in terms of an equation of motion is that simple extensions based on classical and relativistic mechanics are easily implementable.

For example, instead of having a linear viscous force acting on the ANN, a more realistic model could use a quadratic viscosity. In another case, rather than using Newton's laws to find the trajectory, it would be fairly straightforward to instead use relativistice equations of motion by incorporating a speed limit for the propagation of information in weight space. Furthermore, the symmetry groups of weight space could have a profound impact on training. For example, an analysis of the subgroups of the Poincare group on weight space could shed light on how to deal with translational symmetries in the input data. These are lines of research that could be fruitful to explore in the future.

DATA

The training dataset employed in this project was derived from Neurospora Crassa, a filamentous fungus which is one of the primary model organisms in the study of circadian clocks due to its genetic tractability, ease of culture, and possession of a semantically similar clock to higher eukaryotes. All samples were synchronized with a light to dark transition and three biological replicates were collected every two hours for 48 hours. These samples were then sequenced and aligned to the neurospora genome using tophat. Feature counting was performed with HTSeq count and normalization performed with the DESeq bioconductor package. The resulting gene counts were then processed using JTK for Circadian classification. The training data is comprised of a combination of the normalized gene counts at each time point and a classifier of circadian, non-circadian or unclassified based on the adjusted P value output by JTK. The cutoffs for these classifiers were determined by examining the P and Q value distributions for all genes along with hierarchically clustered heatmaps for each set.

After processing, our training data consists of 9548 temporal gene expression profiles, each one a $24-$dimensional vector. The data is divided into three classes, $\{\text{Circadian}, \text{Unclassified}, \text{Non-circadian}\}$ with a 611/1213/7724. The unclassified data has some oscillatory behavior, but doesn't have a high enough $q-$value to be considered circadian. Non-circadian data is non-oscillatory. The data is displayed in 2

One of the main problems that needs to be addressed is how to deal with the vast difference in class volumes. Proposals that seem likely to work are data perturbation and data genesis via PCA. The latter would require doing a PCA analysis of each class to determine the eigen genes in each class, and then creating new linear combinations of them with predetermined monotonically decreasing component weights.

The latter, data perturbation, uses two parameters, ϵ_M and ϵ_A, to perturb the data vectors both multiplicatively and additively with random numbers drawn from a uniform distribution on $[-1/2, 1/2]$. The random numbers are arranged in a diagonal matrix, $\mathbf{G}_M$ and a vector $\vec{\mathbf{G}}_A$, the multiplicative and additive perturbations, respectively:

$$\vec{\mathbf{x}}' = (\mathbb{1} + \epsilon_M \mathbf{G}_M) \cdot \vec{\mathbf{x}} + \epsilon_A \vec{\mathbf{G}}_A \tag{24}$$

To ensure that the perturbations are not too large, the analysis described in the first paragraph can then be utilized to determine the q-values of the new data. In this way we can supplement known data with synthetic data and balance the class volumes. We use this method in our experiment to equalize the class volumes. Using both $\epsilon = .2$ we expand the volume of the circadian data by 10, and the unclassified data by 5. Out final data split is 6110/6065/7724.

EXPERIMENT

For the preliminary experiments that have been done we have made the following architectural choices:

- Activation Function - Sigmoid, with a soft max output layer
- Error Function - Cross Entropy
- Regularizer - Weight Penalization
- Momentum - .1
- Weight Decay - .0001
- Learning Rate - .1
- Layer - 24-24-3, 24-48-3, 24-48-24-3, 48-96-3, 24-24-24-24-3, 24-48-24-24-3, 24-48-3-48-3

Initially we goofed around with the weight decay and the momentum. What we found was that as the weight decay gets larger or the momentum gets smaller, the ANN tends to fall into a local minimum and plateau

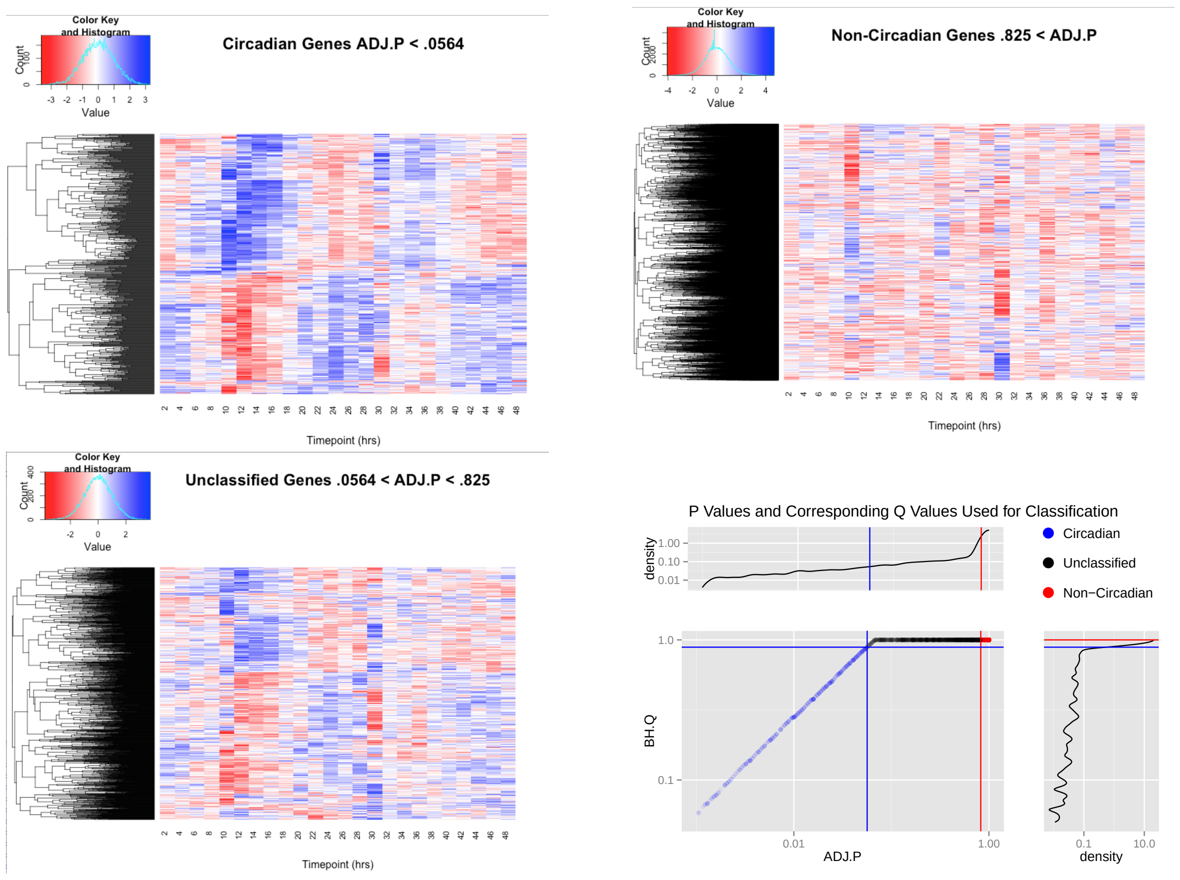

FIG. 2: Hierarchically clustered heatmaps of each class along with a plot of P vs Q for all genes. You can see the oscillatory behavior of the circadian genes oover the course of 48hrs. The unclassified genes have some oscillatory behavior, though not on a 24-hour cycle. Non-cicadian data have no discernable oscillatory behavior. The lower right plot displays the cut-offs used to generate the classes.

at a larger training error. After several runs with high training errors we settled on the above values for both constants.

We trained the network iterating over all the data 5 times (each of which we call an epoch). We use a winner takes all classification on the output, as well as an L_1-norm error. The latter is a useful measure of the confidence we have in our winner takes all classification. The larger the L_1-norm is, the more deviation there is in the output classes, leading us to believe that the winner takes all could be a win by a small margin. We plot these results for a 4 and 5 layer network in figure .

Initially we did not implement a class volume equalization scheme, and found that the non-circadian class overwhelmed the ANN. Errors reflected the discrepency in the class sizes and not the actual performance of the network for all the architectures mentioned above. To fix this we then began using data perturbation to equalize class volumes. Increasing the Circadian and Unclassified classes to have similar volumes to the non-circadian data instantly resulted in better error rates within $1-2$ iterations. We used a 2-1 split of the data into training and test sets, and plot both the training and test winner take all error, as well as the L_1-norm error. We plot the results in figure for the 24-48-3 network.

The training error results in this run are around 2%, and the test error clocks in at around 4%. The confidence in both, as measured by the L_1-norm is about an order of magnitude higher. Thhis is a good error, but there's a little bit of worry about whether or not the class volume utilization has properly generaed new test data. Typically the test error starts to get larger after a certain number of iterations, as the network begins to overlearn hte given training data. We do not see this happening in **??**, though it could be that 100 iterations is not enough for overfitting to start to become relevant. Longer tests need to be performed.

FINAL REMARKS

After trying several different network architectures, we found one that has a pretty decent training and test error. We are a little dubious of our method of class volume

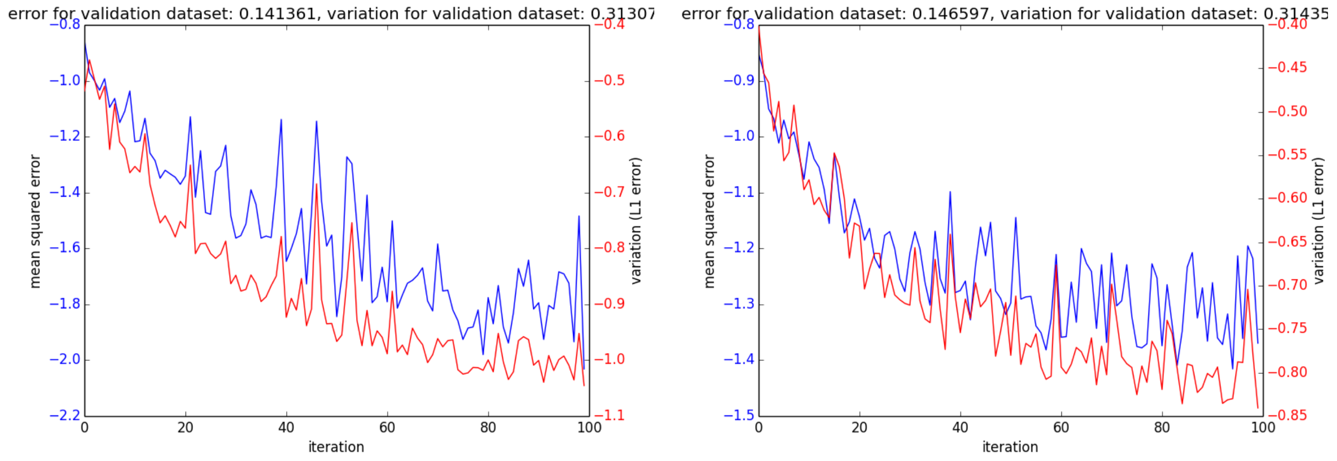

FIG. 3: The error rates during training for two network architectures on a logarithmic scale. Both the L_1 and L_2 errors decrease as more iterations occur. They tend to flatline by 100 iterations, however this could be indicative of overfitting the training data.

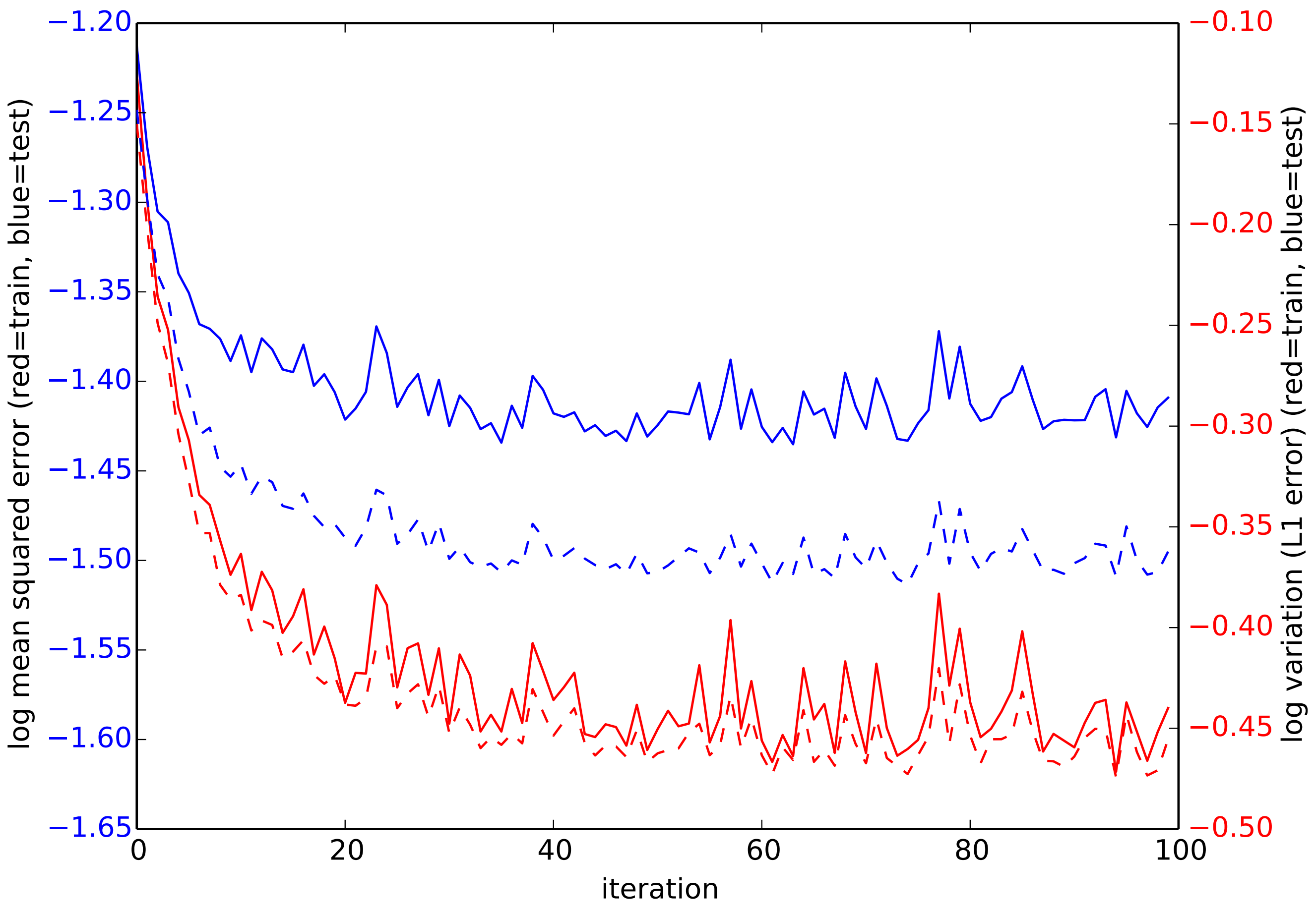

FIG. 4: The winner take all error, and confidence as measured byt he L_1-norm for both the trainging data (red) and the test data (blue). Both begin to plateau after about 40 iterations.

expansion using data perturbation, because we feel that the resulting data perhaps copies the original data too closely. Given more time, we would hope that implementing data genesis via PCA would lead to more varied data in each class.

Fully exploring deeper networks to see how well they do at the classification task is important. Though we tested a few shallow architectures, it would be nice to also get results for much deeper networks (those consisting of 6+ layers). Examining the use of a sparsity regularizer would have been nice too, however there was insufficient time for us to really get to play around with all the possible parameters as we would have liked.

We would also like to direct you to our github page, where we have created a movie of the weights changing over time as we train a network. It's a neat visualization and allows one to see how the weighs slowly settly in on a minimum of the cost function.

ALGORITHMS

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
```

Algorithm 2 Back propagation

```
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
```

BIBLIOGRAPHY

* Electronic address: Damian.Sowinski@dartmouth.edu
† Electronic address: Alexander.M.Crowell@dartmouth.edu
‡ Electronic address: Jack.E.Holland@dartmouth.edu
§ Electronic address: Rawan.al.Ghofaili@dartmouth.edu

[1] Dunlap JC. Molecular bases for circadian clocks. *Cell*, 96(2):271–290, 1999.

[2] Bass J and Takahashi JS. Circadian integration of metabolism and energetics. *Science*, 330(6009):1349–1354, 2010.

[3] Zhang R, Lahens NF, Ballance HI, Hughes ME, and Hogenesch JB. A circadian gene expression atlas in mammals: Implications for biology and medicine. *PNAS*, 111(45):16219–16224, 2014.

[4] Mehra A, Baker CL, Loros JJ, and Dunlap JC. Post-translational modifications in circadian rhythms. *Trends in Biochemical Sciences*, 34(10):483–490, 2009.

[5] Menet JS, Pescatore S, and Rosbash M. Clock:bmal1 is a pioneer-like transcription factor. *Genes & Development*, 28(1):8–13, 2014.

[6] Hughes ME, Hogenesch JB, and Kornacker K. JTK_CYCLE: An efficient nonparametric algorithm for detecting rhythmic components in genome-scale data sets. *Journal of Biological Rhythms*, 25(5):372–380, 2010.

[7] Straume M. Dna microarray time series analysis: automated statistical assessment of circadian rhythms in gene expression patterning. *Meth Enzymol.*

[8] Levine JD, Funes P, Dowse HB, and Hall JC. Signal analysis of behavioral and molecular cycles. *BMC Neuroscience.*

[9] Olshausen BA and Field DJ. Emergence of simple-cell receptive field properties by learning sparse code for natural images *Nature* vol 381, 607-609, June 1996