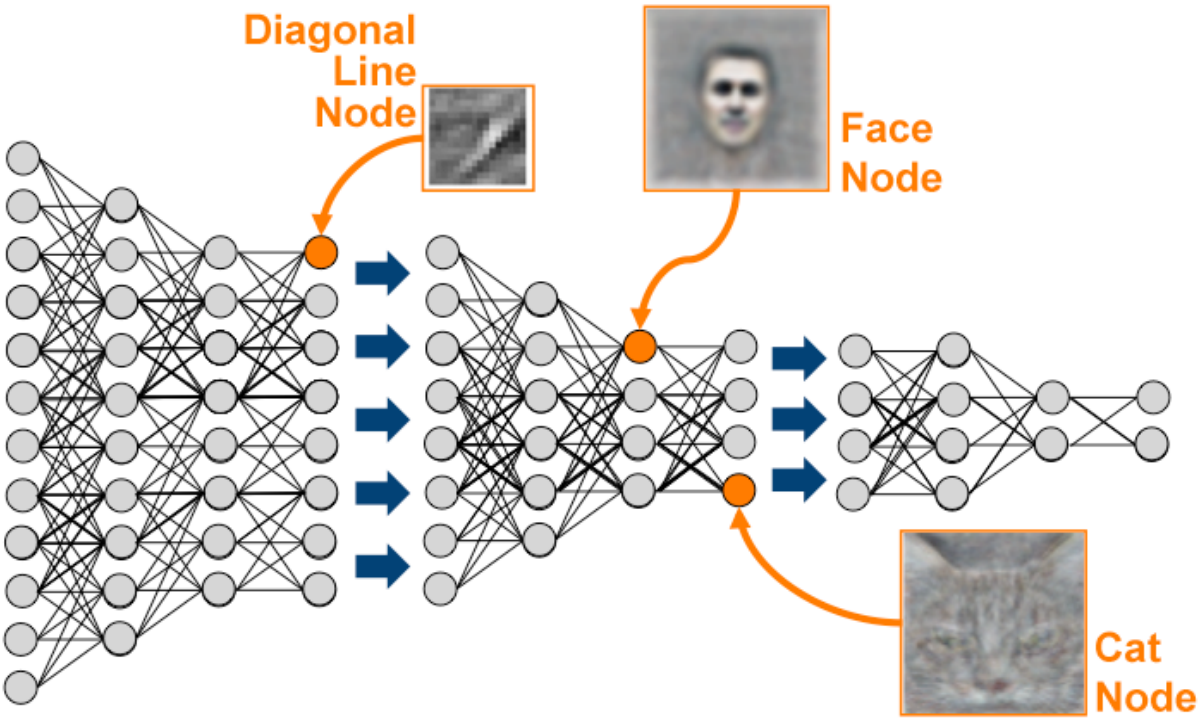
Alex Lenail and Sunjay Bhatia
Team name: “Shen” after the Chinese truth seeing god Erlang Shen

Deep learning is an emerging approach at machine learning which re-implements and extends neural networks, an older category of algorithms that have only recently been found to shine particularly brightly in the context of rapidly growing amounts of computational power and data. Deep learning is a massively parallelizable process, so much so that it is traditionally implemented on Graphical Processing Units (GPUs), though it doesn’t need to be. Deep learning algorithms have especially been exploited in two application spaces in industry: image and language recognition. These are particularly apt problem spaces for neural networks because they perform feature selection and feature representation independently of human input, which is to say, they learn to understand language better than we can describe it to them, in much the same way a baby might learn language as a result of being repeatedly exposed to it.

In the context of images, the feature transformations can be visualized, as below:



For the purposes of image recognition, the features extracted are initially lines and shapes (this is usually done with sparse coding) which together represent the image (in much the same way human brains initially extract lines and contrast from an image). Then, the new representation of the input is once again analyzed, developing increasingly complex representations of the input, until true recognition is achieved.

Deep learning is exciting, for a couple of reasons. First, it's new, which means that we don't actually know quite what it can accomplish for us. Secondly, it bears the promise of actually unveiling something about genuine intelligence which other learning algorithms do not. Due to its biologically motivated origin, although the hardware is dramatically different than an actual brain, it has accidentally mimicked the solutions arrived at by the human brain in a variety of application spaces.

We believe that in order to understand a thing, truly and entirely, you must take it apart and put it back together again. We'd like to implement a fairly naive parallelized neural network in erlang or go, and test it against standard UCI datasets. We'd also like to experiment with how it might work against standard image recognition datasets, such as ImageNet.

Thankfully, a number of open source resources exist detailing the topic (some of which can be found in the References). That means it shouldn't be too difficult to develop an understanding of these techniques, and implement them. But re-implementing an existing algorithm isn't meaningful enough a piece of work to turn in as a final project, nor is it very groundbreaking given that although this is still a young algorithm, many open source repositories exist with functional source code. So the final piece of this project will be to deploy our algorithm against a dataset we haven't seen deep learning leveraged against. We don't necessarily expect impressive results, but we hope we might uncover something interesting.

Deliverables:

Minimum: A functional neural network we implement which operates successfully on the UCI datasets, and has output to some other dataset in which deep learning hasn't been applied.

Maximum: A functional neural network implementation which operates successfully on a variety of datasets, including UCI, ImageNet, and some other dataset(s) on which deep learning hasn't been applied (to our knowledge).

First Step: Functional Algorithm on UCI datasets.

Biggest potential problem: If we go with Erlang, it might not lend itself to heavily mathy code. If we go with Go, we will need to learn a sizeable part of the language in order to implement a deep learning algorithm. The implementation work also isn't too parallelizable, but with careful consideration and design of good abstractions the work should be divisible.

References:

http://deeplearning.stanford.edu/tutorial/
http://deeplearning.net/ especially http://deeplearning.net/datasets/
http://www.iro.umontreal.ca/~bengioy/dlbook/
http://image-net.org/
http://archive.ics.uci.edu/ml/

Design Proposal

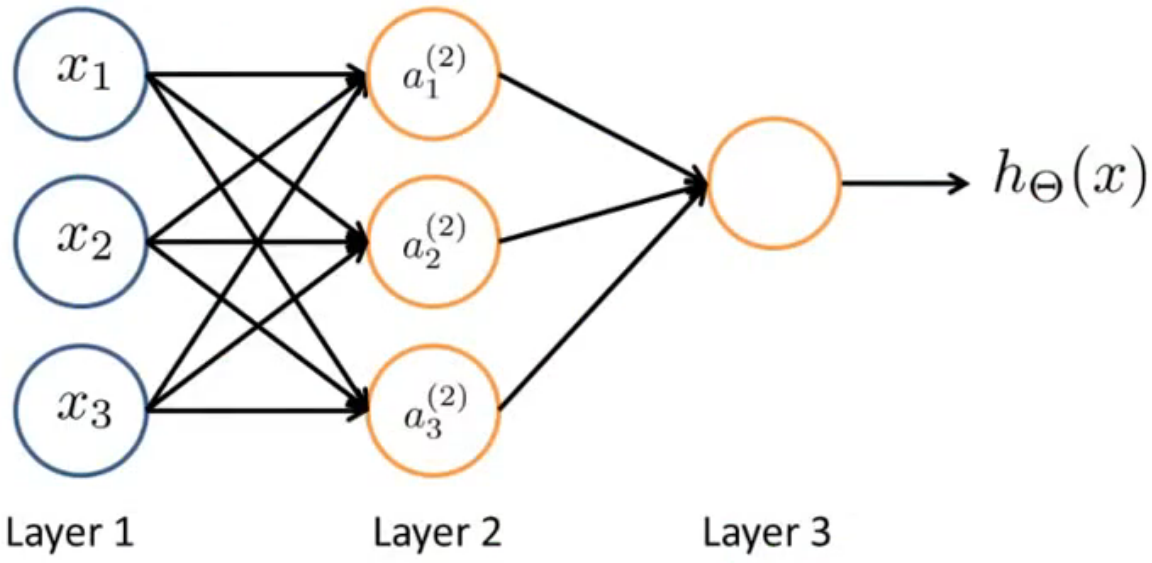
Technologies

In the interest of simplicity, we've elected to move forward with Erlang as our language of choice. There are a variety of reasons we arrived at this conclusion but the most important one was that we care deeply about the deep learning and not so much for the message-passing language upon which it is implemented, and since we already know some erlang and it hides many of the implementation details from us, especially with regards to concurrency, this was the natural choice. More importantly still, we have a really cool team name that relies on our use of Erlang. Although Go would have yielded advantages for the mathematical aspects of our project, notably the matrix manipulations inherent in this algorithm, we felt that learning a new language and concurrency model weren't worth it.

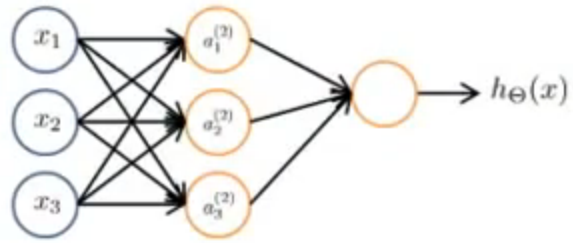
With regards to exploring the potential of implementing these algorithms on GPUs, which we were considering earlier, it seems that there are very few implemented bindings between major GPU SDKs and Erlang (though a couple unstable ones under active development). Though we'd like to keep this option open, for the meantime we are focusing on implementing a neural network in Erlang for CPUs.

Algorithm Design: A brief tutorial on Neural Networks

Neural Networks are often explained with the visual aid of a graph of columns of vertices and connecting edges, which happen to map well to the two major constituent parts of the algorithm: neurons and synapses. Each 'neuron', which is drawn as a node and will be implemented as a process, takes input (messages) from it's 'dendrites' or 'input wires' and outputs the computed sigmoid function of those inputs, passing that along to those next neurons in the network as 'output'.



This leads us to the ‘forward propagation’ algorithm, which maps input to output, which is how predictions are made on test sets but also a core piece of the ‘back propagation’ algorithm which is used to train the model (and which is also described further down).



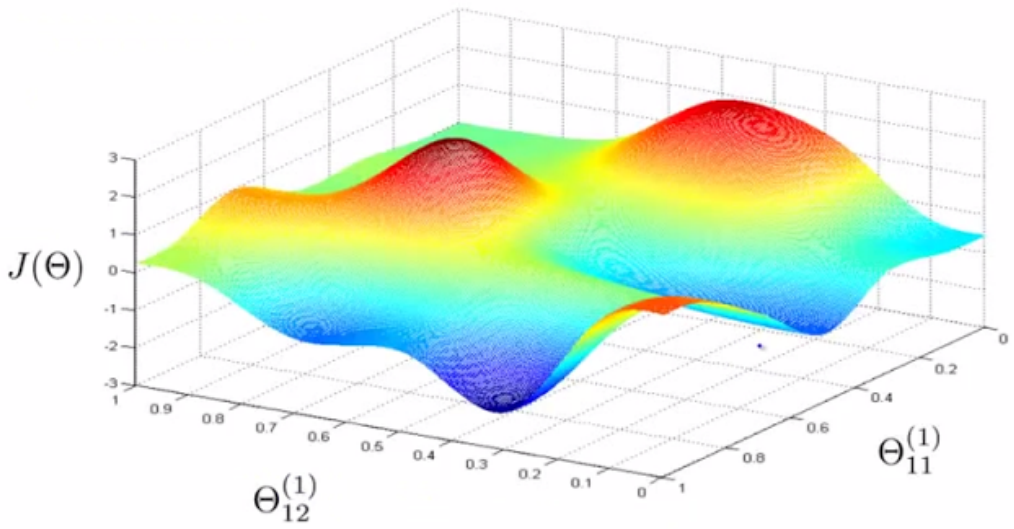
$$a_1^{(2)} = g(\Theta_{10}^{(1)} x_0 + \Theta_{11}^{(1)} x_1 + \Theta_{12}^{(1)} x_2 + \Theta_{13}^{(1)} x_3)$$
$$a_2^{(2)} = g(\Theta_{20}^{(1)} x_0 + \Theta_{21}^{(1)} x_1 + \Theta_{22}^{(1)} x_2 + \Theta_{23}^{(1)} x_3)$$
$$a_3^{(2)} = g(\Theta_{30}^{(1)} x_0 + \Theta_{31}^{(1)} x_1 + \Theta_{32}^{(1)} x_2 + \Theta_{33}^{(1)} x_3)$$

$$h_\Theta(x) = g(\Theta_{10}^{(2)} a_0^{(2)} + \Theta_{11}^{(2)} a_1^{(2)} + \Theta_{12}^{(2)} a_2^{(2)} + \Theta_{13}^{(2)} a_3^{(2)})$$

Forward propagation is an algorithm which computes the ‘activation’ of each neuron, from left to right. The activation of a neuron is isomorphic to whether or not that neuron can be said to have ‘fired’. The hidden layer **a(2)** has three neurons: a(2)(1), a(2)(2), and a(2)(3). The activation of each is the sigmoid function g of the linear combination of the previous layer (in this case, the input **x**) weighted by the model at that layer, **Θ(2)**. This can conveniently be expressed as a matrix vector product: g (**Θ(2)** * **x**). Forward propagation eventually maps the input through the extracted features in the hidden layers to an output, h_Θ(x) by repeating the above procedure appropriately given the topology of the particular network.

In order to tune our network’s model Θ to improve the accuracy of our mapping of inputs to outputs, we use the back propagation algorithm.

At each node in each layer, we have some amount of error in our model’s learned weights. These errors percolate through to some amount of error in our output layer, which we can observe when training our model. This motivates an algorithm which adequately distributes the ‘wrongness’ of the output to the weights in the model and gently shifts those parameters towards a more optimal solution for each training example.



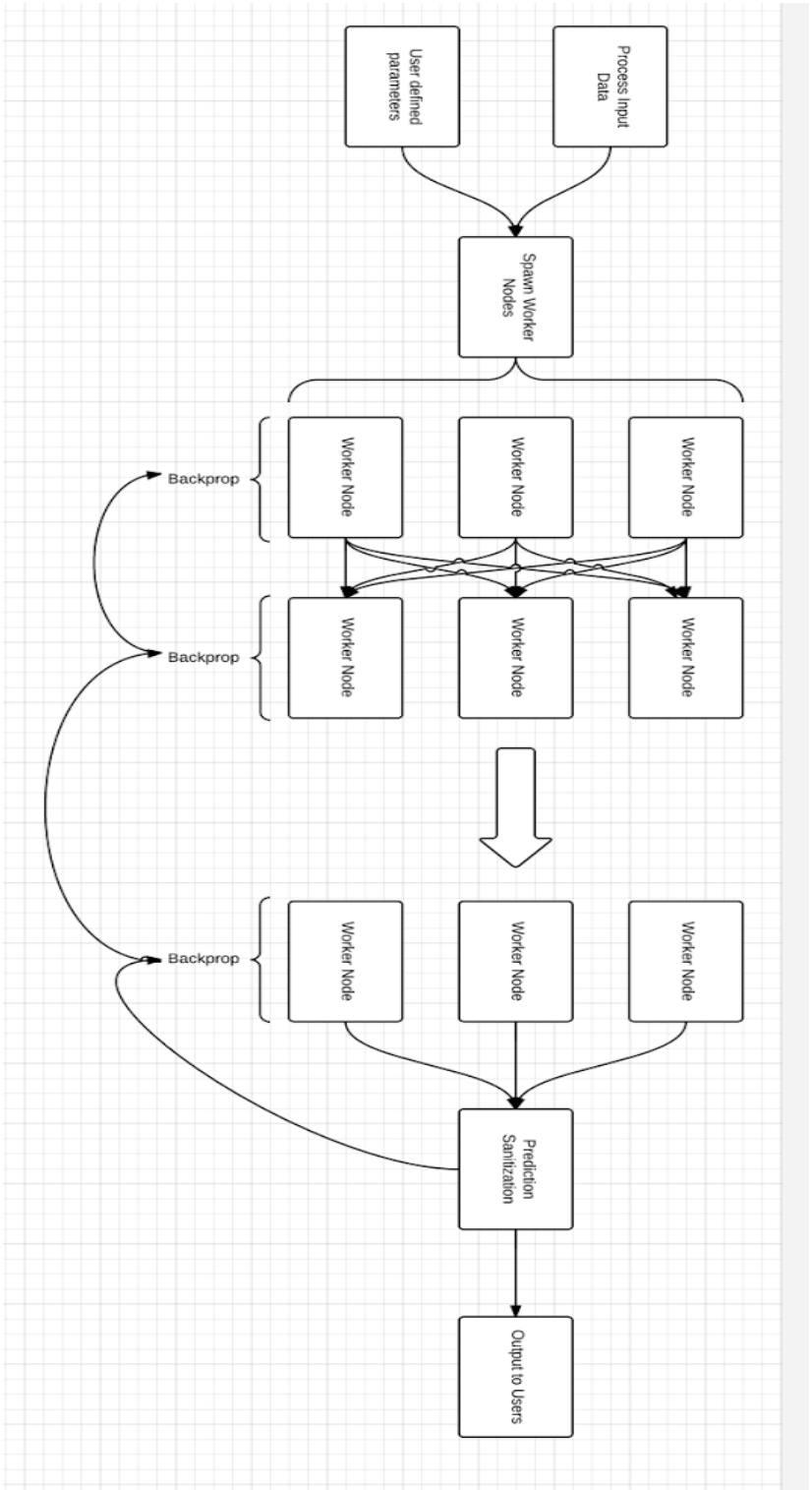
In order to find the error in our network's prediction, we compare our predicted output to the desired output. This where the concept of back propagation comes in, we work backwards in the network and update each node's error based on the next layer's error and the nodes which it propagates to. The gradient of the error is calculated and we use gradient descent to update weights such that the error at that node converges towards a local minimum. The image above shows a multidimensional error function, where the blue valleys represent a more optimal setting of two model parameters with respect to error function J(Θ). With the algorithms random initialization, we start at a random point and using the gradient descent, we attempt to descend into one of these low error valleys (in a hyperdimensional space of model parameters).

Algorithm Design: Implementation Details

The main design decision that we must make is how to represent each node in our network and the interface with which nodes interact. These forward and back propagation algorithms lend themselves well to Erlang's concurrency model and message passing between processes. Nodes in a layer are independent and can operate concurrently when performing matrix operations on our learned weights/error matrices.

The first way we will implement this algorithm is with a naive 'neuron-centric' architecture, in which each neuron will be a process. It will receive messages from neurons and broadcast messages to other neurons once given proper input. This has a couple advantages. First, we will experiment with the idea of a 'distributed neural network' whereby we can have this computation operating on a variety of machines linked together. We expect this will worsen performance, but it proposes an interesting idea and perhaps an avenue for further research. Second, this approach will be simple to put together and extremely intuitive, since neurons

are in fact processes which receive and send messages. This gives us a baseline to work with.
Our first pass at an overall network architecture flow:



An alternate model would be to abstract further and have each layer as a process as opposed to spawning individual nodes at the top level. Each layer would take care of delegation and nodes/neuron processes would simply be computation units that do not interact directly. The layers would be the vessels for message passing and accumulating outputs to pass to the next/previous layer for forward and back propagation.

Each node is simply a computational unit in either design, potentially distributed across multiple computers. Our outward facing interface simply consists of receivers and senders of messages, just as actual neurons receive input from dentrites and pass output with the axon.

Our implementation of this algorithm also hinges on a robust matrix manipulation module for Erlang, which we have looked into and may have found a sort of solution in our preliminary search with: https://github.com/vascokk/NumEr (which could also allow us to expand to GPU use in the future).

Development Plan

So far, we've gotten up to speed on the algorithm which we want to learn: neural networks. We've each studied them using resources online and discussed the finer points of their implementation, as applied to erlang.

In the coming two weeks, the major project is to build and test the naive version, which we expect to complete at least the first part of by this weekend. We anticipate pair coding this piece of the project since it is so central to the remainder of our work.

Thereafter, we'll likely work individually on projects with this baseline which suit our fancy and will likely fall into two categories: generalizing and optimizing the algorithm, and running it against various datasets. I might decide that I'd like to speed up the algorithm using a matrix-math library, while my partner decides that he'd like to test it on a very particular dataset, and builds infrastructure to transform the data into a suitable form for a standard set of interfaces to our implementation.