Outline:

- Neural network overview mostly taken from:

- Papers:

- Some Analysis

- Future paths
This a basic multilayer perceptron (or backpropagation) network. Feedforward networks do not form cycles and the outputs can be computed in a single forward pass. The inputs $x_i$ are used to model the outputs ($y$).

Here $u_{ij}$ is the weight of the connection from the input unit $i$ to hidden unit $j$ and $v_{jk}$ is the weight of the connection from hidden unit $j$ to the output $k$.

$A$ and $B$ are the biases of the hidden and output units.

In this example hyperbolic tangent is being used, but other functions could be used, but if it is linear, the hidden layer would not be needed and one could just directly connect the inputs and outputs.

$$f_k(x) = b_k + \sum_j v_{jk} h_j(x)$$

$$h_j(x) = \tanh(a_j + \sum_i u_{ij} x_i)$$
Basics continued

Several people (Cybenko 1989, Funahashi 1989, Hornik, Stinchcombe, and White 1989) have shown that a multilayer perceptron network with one hidden layer can approximate any function defined on a compact domain arbitrarily closely, if sufficient numbers of hidden units are used. Nevertheless, more complex network architectures may have advantages, and are commonly used.

Neural networks are commonly used for either fitting data or classification. Then one minimizes the squared error or an error function on the training set or maximizes the likelihood.

\[ P(y \mid x) = \prod_k \frac{1}{\sqrt{2\pi \sigma_k}} \exp\left(-\frac{(f_k(x) - y_k)^2}{2\sigma_k^2}\right) \]

\[ P(y = k \mid x) = \exp(f_k(x)) / \sum_{k'} \exp(f_{k'}(x)) \]
Since there is a theorem that says with enough hidden nodes only one layer is needed, we still need to find the number of hidden nodes.

Then some method is needed to try and avoid over fitting by having too many hidden nodes. (DIC, BIC or some sort of weight decaying likelihood)
Main Objective

- The main objective of Bayesian neural networks is to be able to fit new “test” points.

\[
P(y^{(n+1)} | x^{(n+1)}, (x^{(1)}, y^{(1)}), \ldots, (x^{(n)}, y^{(n)}))
\]

\[
= \int P(y^{(n+1)} | x^{(n+1)}, \theta) P(\theta | (x^{(1)}, y^{(1)}), \ldots, (x^{(n)}, y^{(n)})) d\theta
\]

- In Forwardfeed networks where the input is not models the likelihood is:

\[
L(\theta | (x^{(1)}, y^{(1)}), \ldots, (x^{(n)}, y^{(n)})) = \prod_{i=1}^{n} P(y^{(i)} | x^{(i)}, \theta)
\]
This paper was comparing Boltman Machines to neural networks.
I implied from this paper that Boltman Machines were the predecessor to neural networks.
What is a Boltzmann Machine?

- It is some fixed number of 2 valued units linked by symmetric connections. The 2 values are either 0 and 1 or -1 and 1.
- The energy is $E(\vec{s}) = -\beta \sum_{j<i} s_is_jw_{ij}$ and then the probability is $P(\vec{S} = \vec{s}) = \exp(-E(\vec{s}))/Z$ with $Z = \sum_{\vec{s}} \exp(-E(\vec{s}))$ which makes low energy states more probable.
- The value of each state is picked via $P(S_i = x \mid S_j = s_j : j \neq i) = \sigma(x^* \sum_{j \neq i} s_jw_{ij})$ which takes into account every other state. $\sigma(t)$ is defined as $1/(1 + \exp(-t))$. 
Boltman Machine?

- There is also a negative and a positive phase.
- The positive phase increases weights and the negative phase decreases weights. One runs this method until the weights are in equilibrium.
- The positive phase follows the gradient and the negative phase uses the unconditional distribution for $\overline{S}$.
- The negative phase makes it slower and more sensitive to statistical errors.
Results:

![Graphs showing the performance of Boltzmann and Sigmoid FF networks with different values of \( \epsilon = \frac{1}{3} \) and \( \epsilon = 1 \).]

<table>
<thead>
<tr>
<th>Type of network</th>
<th>Time/pass</th>
<th>Best ( \epsilon )</th>
<th>Values of (-L/N)</th>
<th>Error rates</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boltzmann machine (0/1 units)</td>
<td>0.39 s</td>
<td>( \frac{1}{3} )</td>
<td>8.30 8.22 8.23</td>
<td>35% 30% 33%</td>
</tr>
<tr>
<td>Boltzmann machine ((-1/+ 1 units))</td>
<td>0.58 s</td>
<td>( \frac{1}{2} )</td>
<td>7.94 8.25 8.10</td>
<td>10% 37% 25%</td>
</tr>
<tr>
<td>Sigmoid FF network (0/1 units)</td>
<td>0.30 s</td>
<td>4</td>
<td>7.76 7.82 7.77</td>
<td>10% 10% 17%</td>
</tr>
<tr>
<td>Sigmoid FF network ((-1/+1 units))</td>
<td>0.35 s</td>
<td>2</td>
<td>7.72 7.74 7.74</td>
<td>17% 17% 16%</td>
</tr>
</tbody>
</table>

Table 2: Best performances after 200 learning passes (three runs each).

<table>
<thead>
<tr>
<th>Type of network</th>
<th>Passes</th>
<th>( \epsilon )</th>
<th>Values of (-L/N)</th>
<th>Error rates (exact ~ simulated)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boltzmann machine (0/1 units)</td>
<td>1600</td>
<td>( \frac{1}{4} )</td>
<td>7.98 7.94 7.94</td>
<td>20% 20% 21% ~ 22% 21% 21%</td>
</tr>
<tr>
<td>Boltzmann machine ((-1/+1 units))</td>
<td>1000</td>
<td>( \frac{1}{4} )</td>
<td>7.80 7.84 7.77</td>
<td>19% 17% 18% ~ 20% 18% 18%</td>
</tr>
<tr>
<td>Sigmoid FF network (0/1 units)</td>
<td>800</td>
<td>4</td>
<td>7.67 7.72 7.68</td>
<td>16% 19% 16% ~ 20% 20% 18%</td>
</tr>
<tr>
<td>Sigmoid FF network ((-1/+1 units))</td>
<td>800</td>
<td>2</td>
<td>7.64 7.66 7.70</td>
<td>15% 17% 18% ~ 18% 18% 20%</td>
</tr>
</tbody>
</table>
This paper was trying to motivate the hybrid Monte Carlo method to solve neural networks.

In this paper they use a weight decay method to decide the number of hidden modes.

\[ L'(w) = -\frac{|w|^2}{2\omega^2} - \sum_{i=1}^{n} \frac{|y_i - f(x_i, w)|^2}{2\sigma^2} \]

This paper also used annealing a lot. Which is a technique to explore a space to try and find optimum solutions.
If one assumes things to be Gaussian then one can get a Gaussian probability, but this paper is trying to avoid that. Which means Gibbs steps are out.

For a Monte Carlo method the options proposed in the paper are to either change one weight at a time or change all the weights simultaneously. The paper says that terms have a strong interaction so one weight at a time does not make much sense and is expensive to calculate. Also if one tries to change all the weights at once one must take very small steps or it will just be rejected.
Hybrid Monte Carlo method

Hybrid method uses gradient information to pick candidate directions with higher probability of being accepted. This eliminates much of the random walk aspect of the metropolis algorithm, but speeds the exploration of the space.
Results:

Based on informal experiments, a value of $L = 1000$ for the number of steps in the Hybrid Monte Carlo method was selected. Good results were sometimes obtained with 200 iterations, so this was selected as the duration for the later experiments. The plain Metropolis algorithm was run for 200,000 iterations, which results in the same number of function evaluations (and hence roughly the same computation time, approximately twenty minutes on a machine rated at about 25 MIPS).

The paper said that the plain Metropolis algorithm did much worse and had not converged in this time.

This table is for the hybrid case. For each spot on the table they did 9 tests and the failures are what is being recorded (So the bigger the square the worse it did). The $v$ is the step size and the $\mathcal{E}$ is variable that affects convergence speed.

They concluded that the hybrid Monte Carlo method was better and that annealing helped.
This is work on non-forwardfeed neural networks. This is an attempt to do something along the line of time series with neural networks.

They say this can be thought of as a forwardfeed network with time delayed inputs.
Model:

\[ p(y_i|x_{i-1}, \ldots, x_i; \beta, w) \]

\[ = \int_{-\infty}^{\infty} p(y_i|x_{i-1}, \ldots, x_1; s_{i-1}; \beta, w) \cdot p(s_{i-1}|x_{i-1}, \ldots, x_1; \beta, w) ds_{i-1} \]

\[ p(y_i|x_{i-1}, \ldots, x_1; s_{i-1}; \beta, w) = \sqrt{\frac{\beta}{2\pi}} \cdot \exp\left\{ -\frac{\beta}{2} \cdot \left[ y_i - g_w(x_{i-1}, \ldots, x_{i-q}, s_{i-1}) \right]^2 \right\} \]

\[ p(D_n|\beta, w) = p(x_n, \ldots, x_1) \left(\frac{\beta}{2\pi}\right)^{(n-q)/2} \]

\[ \times \exp\left\{ -\frac{\beta}{2} \sum_{i=q+1}^{n} \left[ y_i - g_w(x_{i-1}, \ldots, x_{i-q}, s_{i-1}) \right]^2 \right\} \]

\[ P(c_i = \text{class}_k|x_i, \ldots, x_{i-q+1}, s_i; w) = \frac{\exp[g^k_w(x_i, \ldots, x_{i-q+1}, s_i)]}{\sum_z \exp[g^l_w(x_i, \ldots, x_{i-q+1}, s_i)]} \]

\[ P(c_l = \text{class}_k|D_{l-1}, x_l) = \int_0^1 P(c_l = \text{class}_k|x, w; D_{l-1}, x_l)p(x, w|D_n)dx\,dw, \quad \forall l > n \]

\[ p(D_n|w) = p(x_n, \ldots, x_1) \prod_{i=q}^{n} \exp[g^k_w(x_i, \ldots, x_{i-q+1}, s_i)] \]

\[ \prod_{j} \sum_{i=q}^{n} \exp[g^j_w(x_i, \ldots, x_{i-q+1}, s_i)] \]
They also give a scheme of generating a prior. In which they generate a synthetic data set with their prior knowledge and then use that data set to generate their prior.

I believe this is their method to avoid improper priors.

\[ p(\Theta|D_0) = \frac{p(D_0|\Theta) \cdot p(\Theta)}{p(D_0)} \]
One Xi for each Yi and with clear separation. I used one node.

\[ f_k(x) = b_k + \sum_j v_{jk} h_j(x) \]

\[ h_j(x) = \tanh \left( u_j + \sum_i u_{ij} x_i \right) \]
There is still only one Xi per Yi, but now there are 5 groups. The picture below is with one node. I tried it again with 2 nodes but as one would expect it showed almost no improvement.

$$f_k(x) = b_j + \sum_j v_{jk} h_j(x)$$

$$h_j(x) = \tanh(a_j + \sum_i u_{ij}x_i)$$
For this case I made an X matrix with 5 columns for each Y with 6 groups.

Below is the data in with the first 2 X columns plotted.
For one hidden node one has to fit 18 weights and for each hidden node after that one has to fit 12 more weights. One can see that using BIC or something else is going to restrict the number of hidden nodes.
Going from here

Well I would have to get this harder model working.

The first paper needed Bayesian stuff which the second paper solved. The second paper needed to be able to do more time series like stuff and the third paper solved that.

From here one would want to do more test cases to see how much effect the time series method has to the non–time series method when the data has the relation and does not. Also comparing speeds would be good.