Questions

- How should we recognize the contents of an image?
  - What about a noisy image?
  - What about an occluded image?
  - What about an image in shadows?
- Is the symbol hypothesis the way to go?

We Want

- Classification/Pattern Recognition/Memory
- Recall
- Prediction
- Robustness
- Generalization
- Abstraction
- Highly Parallel?
- Learning: supervised and unsupervised

Neural Networks?

"Conventional training methods for multilayer perceptrons ("backprop" nets) can be interpreted in statistical terms as variations on maximum likelihood estimation. The idea is to find a single set of weights for the network that maximize the fit to the training data, perhaps modified by some sort of weight penalty to prevent overfitting."

– Radford Neal

Applications of neural networks

- Alvinn (the neural network that learns to drive a van from camera inputs).
- NETtalk: a network that learns to pronounce English text.
- Recognizing hand-written zip codes.
- Lots of applications in financial time series analysis.

NETtalk (Sejnowski & Rosenberg, 1987)

- The task is to learn to pronounce English text from examples.
- Training data is 1024 words from a side-by-side English/phoneme source.
- Input: 7 consecutive characters from written text presented in a moving window that scans text.
- Output: phoneme code giving the pronunciation of the letter at the center of the input window.
- Network topology: 7x29 inputs (26 chars + punctuation marks), 80 hidden units and 26 output units (phoneme code). Sigmoid units in hidden and output layer.

NETtalk (contd.)

- Training protocol: 95% accuracy on training set after 50 epochs of training by full gradient descent. 78% accuracy on a set-aside test set.
- Comparison against Dectalk (a rule based expert system): Dectalk performs better; it represents a decade of analysis by linguists. NETtalk learns from examples alone and was constructed with little knowledge of the task.
Neural nets are biologically inspired, but don’t operate like real neurons!

The McCulloch-Pitts neuron (1943)

Various functions are used for thresholds:
- Sign and a Step functions
- Logistic or sigmoid function: s-shaped and continuous
- Hyperbolic tangent (tanh): s-shaped, but symmetric about the origin (continuous)

In 1943, McCollough and Pitts showed that a synchronous assembly of such neurons is a universal computing machine. That is, any Boolean function can be implemented with threshold (step function) units.

Implementing AND

\[ o(x_1, x_2) = \begin{cases} 1 & \text{if } -2 + x_1 + x_2 \geq 0 \\ -1 & \text{otherwise} \end{cases} \]

Implementing OR

\[ o(x_1, x_2) = \begin{cases} 1 & \text{if } -1 + x_1 + x_2 \geq 0 \\ -1 & \text{otherwise} \end{cases} \]
Implementing NOT

\[ o(x_1) = \begin{cases} 1 & \text{if } 0.5 - x_1 > 0 \\ 0 & \text{otherwise} \end{cases} \]

Implementing more complex Boolean functions

\[ W = 0.5 \]

Non-biological?

- Artificial Neural Nets don’t account for:
  - Time delays in the system
  - Firing frequency and synchronization

And Then There Were...

- Perceptrons (1958)

Feed-forward Networks

- Perceptrons are examples of a feed-forward network
  - They are a directed graph
  - There are no cycles in the graph
  - There is no internal state other than the weights

How it works

- The value of an output is: \[ O_i = \text{step} \sum_j W_{ij} f_j \]
- Learning occurs when the actual output is modified to be more similar to the desired output
### The Original Learning Rule

- The original perceptron learning rule:
  1. If output neuron $O_i$ incorrectly produced a 1, then we need to decrease $W$ for $i$. For each incoming synapse to $O_i$, set $W_{ij} = W_{ij} - I_j$
  2. If output neuron $O_i$ incorrectly produced a 0, then we need to increase $W$ for $i$. For each incoming synapse to $O_i$, set $W_{ij} = W_{ij} + I_j$
  3. If output neuron $O_i$ got the right value, then make no changes to its weights.

### Better Learning Rule

- The learning rule was a special case of:
  $$\Delta W_{ij} = \alpha (C_i - O_i) I_j$$
  - $C_i$ is the correct output value for neuron $i$.
  - The alpha value is the learning rate (so we don’t overshoot and oscillate around the value).

### The Overall Training Procedure

1. Pick an input/expected output vector pair at random.
2. Present the input vector pair to the input neurons.
3. Read the network's output vector.
4. Using the learning rule, modify each weight according to the difference between output vector and the expected output vector.
5. Go to #1.

### What Can’t Perceptron’s Do?

- **XOR**
  - Distinguish on the basis of connectivity such figures as:
    - One Connected Spiral Region
    - Two Separate Connected Spiral Regions

### Spiral Regions

![Spiral Regions](image)

### Linearly Separable

- Perceptrons can only learn linearly separable functions

![Linearly Separable](image)
There is no place where the xor problem can be linearly separated!

New neural networks from the '80's used an input, output, and 1+ hidden layers.

New Architectures Emerged

There is no place where the xor problem can be linearly separated!

New Architectures Emerged

Minsky and Papert’s description of neural nets with hidden units:

- GAMBA PERCEPTRON: A number of linear threshold systems have their outputs connected to the inputs of a linear threshold system. Thus we have a linear threshold function of many linear threshold functions.

What Killed the Field of Neural Nets…

Minsky and Papert then stated:
- “Virtually nothing is known about the computational capabilities of this latter kind of machine. We believe that it can do little more than can a low order perceptron. (This, in turn, would mean, roughly, that although they could recognize (sp) some relations between the points of a picture, they could not handle relations between such relations to any significant extent.) That we cannot understand mathematically the Gamba perceptron very well is, we feel, symptomatic of the early state of development of elementary computational theories.”

Multiple Layers

A 2-layer network can learn any continuous function.
A three layer neural network can learn any function.
We will deal with 2-layer networks. But...
- How do we train them?

Notation and Output

Variables we’ll use
- The connection to an output neuron \( O_i \) from hidden layer neuron \( H_j \) is represented as \( W_{ij} \)

\[ \begin{align*}
O_i &= \sigma \left( \sum_{j} W_{ij} H_j \right) \\
H_j &= \sigma \left( \sum_{k} W_{ik} O_k \right) \\
\sigma(u) &= \frac{1}{1+e^{-u}}
\end{align*} \]
Initial Weights

- Perceptron: 0
- NN with Hidden Layer: small random values

Training

- The rule we use for modifying the weights is known as the delta rule, because it changes each weight according to how much say it had in the final outcome (the delta, or partial derivative of the output with respect to the weight).

\[
\Delta W_i = \alpha (C_i - O_i) O_i (1 - O_i) H_j \\
\Delta V_j = \alpha \sum (C_i - O_i) O_i (1 - O_i) W_i H_j (1 - H_j) H_i
\]

What This Means

\[
\Delta W_i = \alpha (C_i - O_i) O_i (1 - O_i) H_j \\
\Delta V_j = \alpha \sum (C_i - O_i) O_i (1 - O_i) W_i H_j (1 - H_j) H_i
\]

- The rule is applied to all weights at the same time.
- The alpha value is the learning rate
- Notice the similarity between the learning rule for W and that of the perceptron

\[
\Delta W_i = \alpha (C_i - O_i) H_j
\]

Error is Distributed

- Each hidden node is responsible for some fraction of the error in each of the output nodes. This fraction equals the strength of the connection (weight) between the hidden node and the output node.

The Training Procedure

- Same as the perceptron, but since it operates on continuous values, it will never get the exact output
- We just want the output to converge to a correct output within some limit.
- Backpropagation won't always converge: it may get caught in a suboptimal solution, because it is a greedy algorithm

The Learning Rate

- If it's too high, the we won't converge, but will overshoot
- If it's low we'll take forever to learn
- It's normal to start out with a higher learning rate and lower it slowly to help with convergence
Momentum

- Sometimes we multiply $w_i$ by a momentum factor $\alpha$. This allows us to use a high learning rate, but prevent the oscillatory behavior that can sometimes result from a high learning rate.

$$w_i \leftarrow w_i + \alpha \delta x_i$$

Multiply by momentum $\alpha$.

The Hidden Layer

- If you provide too many nodes in the hidden layer, it will learn every input at one node.
- If you provide too few, then it can’t model the data.
- When you’re between the two extreme’s the network will generalize to a certain extent.

More on backpropagation

- Performs gradient descent over the entire network weight vector.
- Will find a local, not necessarily global error minimum.
- Minimizes error over training set; need to guard against overfitting just as with decision tree learning.
- Training takes thousands of iterations (epochs) — slow!

When neural nets are appropriate for learning problems

- Instances are represented by attribute-value pairs.
  - Pre-processing required: Continuous input values to be scaled in [0-1] range, and discrete values need to converted to Boolean features.
- Training examples are noisy.
- Long training times are acceptable.
- Human understandability of learned function is unimportant.
  - However, there is work on converting NNs to rules.

Network topology

- Designing network topology is an art.
- We can learn the network topology using genetic algorithms. But using GAs is very cpu-intensive. An alternative that people use is hill-climbing.

Delta Rule Proof

- Each weight contributes some amount to the output.
- We change the weights to approximate the “correct” values more.
- The gradient of the output: the change in the output with respect to the change in weights — the first derivative with respect to the weights.
- The network should go away from the direction of the gradient (we want the difference to get smaller): this is known as gradient descent.
**Error Metric**

- We measure the difference between the output and the desired output with an error metric. Our goal will be to move in the direction that makes this error metric as small as possible (away from the error gradient). The error metric we'll use is a mean squared error:

$$E = \frac{1}{2} \sum_j (C_j - O_j)^2$$

**Greedy Change**

- We will do this greedily with respect to each weight; each weight will be changed a little bit (depending on the alpha value) in the direction that goes down that gradient, that is, that makes the error smaller.
- The gradient with respect to a single weight is the same thing as the partial derivative of the output with respect to that weight. For a weight $W_{ij}$, we want the change to be (expanding out with the chain rule):

$$\Delta W_{ij} = -\alpha \frac{\partial E}{\partial W_{ij}}$$

**First Term**

- Remember, it's negative because we want to move away from the gradient. Using the error function, the first term reduces to:

$$\frac{\partial E}{\partial O_j} = (C_j - O_j)(-1)$$

**Second Term**

- For the second term, we need to take advantage of the nifty derivative of the sigmoid. The derivative of the sigmoid function includes the sigmoid function itself in the answer!

$$\sigma'(u) = \sigma(u)(1 - \sigma(u))du$$

**Some Comments**

- Thus the second term is the derivative of the sigmoid function. Don't forget the chain rule, however! So the $du$ gets expanded as well, and we get:

$$\frac{\partial^2 E}{\partial W_{ij} \partial O_j} = \sigma(\sum_j W_{ij}H_j)(1 - \sigma(\sum_j W_{ij}H_j)) \frac{\partial (\sum_j W_{ij}H_j)}{\partial W_{ij}}$$

$$\frac{\partial^2 E}{\partial W_{ij} \partial W_{ik}} = \sigma(\sum_j W_{ij}H_j)(1 - \sigma(\sum_j W_{ij}H_j))H_k$$

- The summation here deserves a little explaining. Notice that in some places we use $j$ to refer to a specific $j$, but in the sums what we mean is to sum over all $j$. You'll see this happening in the later sums in the proof as well. Sorry 'bout that.
- What this means is that in the partial derivative, over the sum of all $j$, with respect to a specific $j$, all the summed stuff gets thrown out (treated as a constant) except for the specific $j$'s stuff. That's why the delta just reduces to $H$ --- because the derivative of $W_{ij}H_j$ with respect to $W_{ij}$ is simply $H_j$. 
Notice

- Notice the similarity between pieces of this function and the input rule for a neuron $O_i$ way back when. So this is nothing more than:
  \[ \frac{\partial O_i}{\partial W_{ij}} = O_i(1-O_i)H_j \]
- and thus we get:
  \[ \Delta W_{ij} = \alpha (C_i - O_i)O_i(1-O_i)H_j \]

The V Matrix

- The rule for the V matrix is derived similarly; it's just a little longer, with one extra term expanded out in the first chain rule expansion:
  \[ \Delta V_{ij} = -\alpha \frac{\partial E}{\partial V_{ij}} = -\alpha \frac{\partial E}{\partial H_j} \frac{\partial H_j}{\partial V_{ij}} = -\alpha \sum \frac{\partial E}{\partial O_i} \frac{\partial O_i}{\partial H_j} \frac{\partial H_j}{\partial V_{ij}} \]

V Matrix (cont’d)

- The first term in the summation we’ve already figured out before. So let’s start with the second term. Once again, we use the derivative of sigma, and things work out similarly to the way they did before:
  \[ \frac{\partial O_i}{\partial H_j} = \alpha (\sum W_{ij}H_{ij})(1-\alpha (\sum W_{ij}H_{ij})) \frac{\partial (\sum W_{ij}H_{ij})}{\partial H_j} \]
  \[ \frac{\partial O_i}{\partial j} = \alpha (\sum W_{ij}H_{ij})(1-\alpha (\sum W_{ij}H_{ij}))W_{ij} \]
  \[ \frac{\partial O_i}{\partial j} = O_i(1-O_i)W_{ij} \]

V Matrix (cont’d)

- The final term works out, once again, in the same basic way:
  \[ \frac{\partial H_j}{\partial V_{ij}} = \alpha (\sum Y_{ij}X_i - \alpha (\sum Y_{ij}X_i)) \frac{\partial (\sum Y_{ij}X_i)}{\partial V_{ij}} \]
  \[ \frac{\partial H_j}{\partial V_{ij}} = \alpha (\sum Y_{ij}X_i - \alpha (\sum Y_{ij}X_i))Y_i \]
  \[ \frac{\partial H_j}{\partial V_{ij}} = H_j(1-H_j)Y_i \]

Finally

- ...and thus our learning rule for V is:
  \[ \Delta V_{ij} = \alpha \sum (C_i - O_i)O_i(1-O_i)W_{ij}(1-H_j)Y_i \]