Lecture 10 – Neural Networks

Machine Learning
Queens College
Last Time

• Perceptrons
  – Perceptron Loss
  – Training Perceptrons using Gradient Descent
Today

- Multilayer Neural Networks
  - Feed Forward
  - Error Back-Propagation
Recall: The Neuron Metaphor

• Neurons
  – accept information from multiple inputs,
  – transmit information to other neurons.
• Multiply inputs by weights along edges
• Apply some function to the set of inputs at each node
Types of Neurons

Linear Neuron

Perceptron

Logistic Neuron

Potentially more. Require a convex loss function for gradient descent training.
Perceptron Limits

- Minksy and Papert (1969) wrote a book analyzing the perceptron and demonstrating many functions it could not learn, such as XOR.
- These results discouraged further research on neural nets; and symbolic AI became the dominate paradigm.
Concept Perceptron Cannot Learn

- Cannot learn exclusive-or, or parity function in general.
Multilayer Networks

- A typical multi-layer network consists of an input, hidden and output layer, each fully connected to the next, with activation feeding forward.
- The output from one layer is the input to the next
- Each Layer has its own sets of weights
Multi-Layer Networks

- The weights determine the function computed. Given an arbitrary number of hidden units, any boolean function can be computed with a single hidden layer.
Linear Regression Neural Networks

• What happens when we arrange **linear neurons** in a multilayer network?
Nothing special happens.
  – The product of two linear transformations is itself a linear transformation.
Neural Networks

• We want to introduce non-linearities to the network.
  – Non-linearities allow a network to identify complex regions in space
Linear Separability

- 1-layer cannot handle XOR
- More layers can handle more complicated spaces – but require more parameters
- Each node splits the feature space with a hyperplane
- If the second layer is AND a 2-layer network can represent any convex hull.
Training Multi-layer Neural Networks

• Multi-layer networks can represent arbitrary functions
• But an effective learning algorithm for such networks was thought to be difficult
Gradient Descent

• Define objective to minimize error:

\[ E(W) = \frac{1}{2} \sum_{d \in D} \sum_{k \in K} (t_{kd} - o_{kd})^2 \]

where \( D \) is the set of training examples, \( K \) is the set of output units, \( t_{kd} \) and \( o_{kd} \) are, respectively, the teacher and current output for unit \( k \) for example \( d \).

• Learning rule to change weights to minimize error is:

\[ \Delta w_{ji} = -\eta \frac{\partial E}{\partial w_{ji}} \]
Derivative of Sigmoid Function

- The derivative of a sigmoid unit with respect to an input is:

\[ o_j = g(in_j) \quad \frac{\partial o_j}{\partial in_j} = o_j(1 - o_j) \]
Error Backpropagation

Update rule for $w_{jk}$:

$$E_{mk} = \frac{1}{2} (t_k - o_k)^2$$

$$\frac{\partial E_{mk}}{\partial w_{jk}} = \frac{\sqrt{2} (t_k - o_k)}{2w_{jk}} g'(\sum_j w_{jk} o_j)$$

Update rule:

$$w_{jk} = w_{jk} - \eta (- (t_k - o_k) g'(\sum_j w_{jk} o_j) o_j)$$

$$= w_{jk} + \eta (t_k - o_k) g'(\sum_j w_{jk} o_j) o_j$$
Error Backpropagation

Update rule for $w_{ij}$:

$$Z_{mk} = \frac{1}{2}(t_k - o_k)^2$$

$$2Z_{mk} \frac{\partial Z_{mk}}{\partial w_{ij}} = -(t_k - o_k) \frac{\partial o_k}{\partial w_{ij}}$$

$$= -(t_k - o_k) g'(i_k) \frac{\partial g'(o_j w_{jk})}{\partial w_{ij}}$$

$$= -(t_k - o_k) g'(i_k) w_{jk} \frac{\partial g'(\sum_i w_{ij} i_i)}{\partial w_{ij}}$$

$$= -(t_k - o_k) g'(i_k) w_{jk} g'(i_j) \frac{\partial (\sum_i w_{ij} i_i)}{\partial w_{ij}}$$

$$= -(t_k - o_k) g'(i_k) w_{jk} g'(i_j) \frac{\partial w_{ij}}{\partial w_{ij}}$$

$$= -(t_k - o_k) g'(i_k) w_{jk} g'(i_j) w_{ij}$$

$$w_j = w_{ij} + \eta (t_k - o_k) g'(o_j) w_{jk} g'(i_j) i_i$$
Backpropagation Learning Rule

• For each data point, each weight changed by:

\[ \Delta w_{ij} = \eta \delta_j o_i \]

\[ \delta_j = o_j (1-o_j)(t_j-o_j) \quad \text{if } j \text{ is an output unit} \]

\[ \delta_j = o_j (1-o_j) \sum_k \delta_k w_{jk} \quad \text{if } j \text{ is a hidden unit} \]

where \( \eta \) is a constant called the learning rate

\( \delta_j \) is the error measure for unit \( j \)
Error Backpropagation

• First calculate error of output units and use this to change the top layer of weights.

Current output: $o_j=0.2$

Correct output: $t_j=1.0$

Error $\delta_j = o_j(1-o_j)(t_j-o_j) = 0.2(1-0.2)(1-0.2)=0.128$

Update weights into $j$

$$\Delta w_{ij} = \eta \delta_j o_i$$
Error Backpropagation

• Next calculate error for hidden units based on errors on the output units it feeds into.

\[ \delta_j = o_j (1 - o_j) \sum_k \delta_k w_{jk} \]
Error Backpropagation

- Finally update bottom layer of weights based on errors calculated for hidden units.

$$\delta_j = o_j(1-o_j) \sum_k \delta_k w_{jk}$$

Update weights into j

$$\Delta w_{ij} = \eta \delta_j o_i$$
Backpropagation Training Algorithm
Online version

Create the 3-layer network with H hidden units with full connectivity between layers.
Set weights to small random real values.
Until all training examples produce the correct value (within $\varepsilon$), or mean squared error ceases to decrease, or other termination criteria:

Begin epoch
For each training example, d, do:
  Calculate network output for d’s input values
  Compute error between current output and correct output for d
  Update weights by backpropagating error and using learning rule
End epoch
function BACK-PROP-LEARNING(examples, network) returns a neural network

inputs: examples, a set of examples, each with input vector \( \mathbf{x} \) and output vector \( \mathbf{y} \)

network, a multilayer network with \( L \) layers, weights \( w_{i,j} \), activation function \( g \)

local variables: \( \Delta \), a vector of errors, indexed by network node

repeat

for each weight \( w_{i,j} \) in network do
    \( w_{i,j} \leftarrow \) a small random number

for each example \((\mathbf{x}, \mathbf{y})\) in examples do
    /* Propagate the inputs forward to compute the outputs */
    for each node \( i \) in the input layer do
        \( a_i \leftarrow x_i \)
    for \( \ell = 2 \) to \( L \) do
        for each node \( j \) in layer \( \ell \) do
            \( \text{in}_j \leftarrow \sum_i w_{i,j} a_i \)
            \( a_j \leftarrow g(\text{in}_j) \)
    /* Propagate deltas backward from output layer to input layer */
    for each node \( j \) in the output layer do
        \( \Delta[j] \leftarrow g'(\text{in}_j) \times (y_j - a_j) \)
    for \( \ell = L - 1 \) to 1 do
        for each node \( i \) in layer \( \ell \) do
            \( \Delta[i] \leftarrow g'(\text{in}_i) \sum_j w_{i,j} \Delta[j] \)
    /* Update every weight in network using deltas */
    for each weight \( w_{i,j} \) in network do
        \( w_{i,j} \leftarrow w_{i,j} + \alpha \times a_i \times \Delta[j] \)

until some stopping criterion is satisfied

return network
Backpropagation Training Algorithm
Batch version

Create the 3-layer network with H hidden units with full connectivity between layers.
Set weights to small random real values.
Until all training examples produce the correct value (within \( \varepsilon \)), or mean squared error ceases to decrease, or other termination criteria:

Begin epoch
For each training example, \( d \), do:
  Calculate network output for \( d \)'s input values
  Compute error between current output and correct output for \( d \)
  Accumulate errors for each weight
Update weights by backpropagating error and using learning rule
End epoch
Comments on Training Algorithm

• Not guaranteed to converge to zero training error, may converge to local optima or oscillate indefinitely.
• However, in practice, does converge to low error for many large networks on real data.
• Many epochs (thousands) may be required, hours or days of training for large networks.
• To avoid local-minima problems, run several trials starting with different random weights (*random restarts*).
  – Take results of trial with lowest training set error.
  – Build a committee of results from multiple trials (possibly weighting votes by training set accuracy).
Representational Power

- **Boolean functions**: Any boolean function can be represented by a two-layer network with sufficient hidden units.

- **Continuous functions**: Any bounded continuous function can be approximated with arbitrarily small error by a two-layer network.
  - Sigmoid functions can act as a set of basis functions for composing more complex functions, like sine waves in Fourier analysis.

- **Arbitrary function**: Any function can be approximated to arbitrary accuracy by a three-layer network.
XOR Network

\[
\begin{align*}
X_1 \text{ XOR } X_2 &= (X_1 \text{ AND NOT } X_2) \text{ OR } (X_2 \text{ AND NOT } X_1) \\
\text{Threshold}(Y) &= 2
\end{align*}
\]
Problems with Neural Networks

- Interpretation of Hidden Layers
- Overfitting
Hidden Unit Representations

• What are the hidden layers doing?!

• Trained hidden units can be seen as newly constructed features that make the target concept linearly separable in the transformed space.

• On many real domains, hidden units can be interpreted as representing meaningful features such as vowel detectors or edge detectors, etc..

• However, the hidden layer can also become a distributed representation of the input in which each individual unit is not easily interpretable as a meaningful feature.
Interpretation of Hidden Layers

• The non-linearities in the feature extraction can make interpretation of the hidden layers very difficult.
• This leads to Neural Networks being treated as black boxes.
Overfitting in Neural Networks

• Neural Networks are especially prone to overfitting.

• Recall Perceptron Error
  – Zero error is possible, but so is more extreme overfitting
Bayesian Neural Networks

- Inserting a prior on the weights
  - Similar to L2 Regularization
- Error Backprop then becomes Maximum A Posteriori (MAP) rather than Maximum Likelihood (ML) training

\[ R(\theta) = \frac{1}{N} \sum_{n=0}^{N} L(y_n - f(x_n)) + \lambda ||\theta||^2 \]
Other Neural Networks

• Skip Layer Network
• Recurrent Neural Networks
• Etc.
Next

• Logistic regression