COMP 790-125: Goals for today

- Neural Nets
- XOR learning
- Convolution and pooling
Sigmoid

Recall sigmoid function

\[ \sigma(z) = \frac{1}{1 + \exp\{-z\}} \]

We used this function when we discussed logistic regression

\[ p(y = 1|\mathbf{x}) = \sigma(b + \mathbf{w}^T\mathbf{x}) \]
Logistic regression is weak

Logistic regression is a classifier with a linear decision boundary \((wx + b > 0)\) and hence a “weak” learner.

For example, assume that we are trying to learn the following concept

\[
c(x) = \begin{cases} 
1, & 0 \leq x \leq 1 \\
-1, & \text{otherwise}
\end{cases}
\]
Logistic regression is weak

Regardless how many examples of \((x, y)\) pairs we are given logistic regression will never be able to learn this concept.

However, using a derived set of features

\[
x^1(x) = \begin{cases} 
1, & x > 1 \\
0, & \text{otherwise}
\end{cases}
\]

\[
x^2(x) = \begin{cases} 
1, & x < 0 \\
0, & \text{otherwise}
\end{cases}
\]

the problem is trivial even for logistic regression.
Deeper architectures

Logistic regression is shallow – it builds separating hyperplane linearly from features.

Ability to induce more complex features requires deeper architecture (decision trees for example).

Neural networks and deep belief networks and deep boltzmann machines are such alternatives.
Sigmoidal unit

A bit different story than with graphical models.
Nodes/units correspond to function evaluations – neurons responding to their inputs.
If inputs are strong enough, the neuron is activated, and it fires.

\[ y = \sigma(b + \mathbf{w}^T \mathbf{x}) \]
Different activation functions

Sigmoid is just one of possible activation functions. Another, much more commonly used, activation function is Rectified Linear Unit (ReLU):

\[ f(z) = \max(0, z) \]
There is no need to treat bias term $b$ separately. We can simply introduce a node that is always 1 and let it serve as input to all other units.

$$y = f(w^T x)$$
Neural network with a single hidden layer and a single output

\[ h_i = f(w_{1i}^T x) \]
\[ y_1 = f(w_{21}^T h) \]
Neural network with a single hidden layer and multiple outputs

\[ h_i = f(w_{1i}^T x) \]
\[ y_i = f(w_{2i}^T h) \]
Deeper architecture

output layer

hidden layers

input layer

\[ y_1 \quad y_2 \quad \cdots \quad y_n \]

\[ h_{l1} \quad h_{l2} \quad \cdots \quad h_{lm_l} \]

\[ h_{11} \quad h_{12} \quad \cdots \quad h_{1m_1} \]

\[ x_1 \quad x_2 \quad \cdots \quad x_p \]

\( n \) output nodes

\( m_l \) hidden nodes

\( m_1 \) hidden nodes

\( p \) input variables
A simple neural net for solving XOR problem

Correct prediction of Exclusive-OR output cannot be accomplished using shallow models.

<table>
<thead>
<tr>
<th>$x_1$</th>
<th>$x_2$</th>
<th>$x_1 \text{ XOR } x_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Forward propagation in XOR network

We will assume that the activation function $g$ is ReLU, $g(z) = \max(0, z)$,

$$h_i = g(x^T W_{:,i} + c_i)$$

and output is predicted as a linear combination of the hidden variables

$$y = w^T h + b$$

Note that we compute $h$ first and then use $h$ to compute $y$. 
Full computation in XOR network

\[ y = \mathbf{w}^T \max(\mathbf{XW} + \mathbf{c}, 0) + b \]

Let our input data be

\[
\mathbf{X} = \begin{bmatrix}
0 & 0 \\
0 & 1 \\
1 & 0 \\
1 & 1 \\
\end{bmatrix}
\]

and parameters

\[
\mathbf{W} = \begin{bmatrix}
1 & 1 \\
1 & 1 \\
\end{bmatrix}, \quad \mathbf{c} = \begin{bmatrix}
0 \\
-1 \\
\end{bmatrix}, \quad \mathbf{w} = \begin{bmatrix}
1 \\
-2 \\
\end{bmatrix}, \quad b = 0
\]

We’ll compute a forward propagation on the board.
Neural networks – computing a prediction

We can define a bit of notation to simplify discussion

- $a_{j,i}$, unit $i$’s input to unit $j$
- $w_{j,i}$, weight of unit $i$’s input to unit $j$
- $q_j = \sum_i w_{j,i} a_{j,i}$ net input to unit $j$
- $o_j$ output of unit $j$

For input layer $o_j = x_j$, for the rest $o_j = f(q_j)$. For output layers

- continuous $y_j$ then $\hat{y}_j = q_j$
- binary $y_j$ then $\hat{y}_j = \frac{1}{1+\exp(-q_j)}$ (sigmoid)
- categorical $y_j$ then $\hat{y}_j = \frac{\exp(q_j)}{\sum_k \exp q_k}$ (softmax)
Neural networks – computing a prediction

In this case

\[ q_j = \sum_i w_{j,i} a_{j,i} = w_{j,r}o_r + w_{j,s}o_s + w_{j,t}o_t \]

\[ o_j = f(q_j) = f(w_{j,r}o_r + w_{j,s}o_s + w_{j,t}o_t) \]
The forward propagation computes output of neural network by iterating through layers and computing:

\[ q_j = \sum_i w_{j,i} a_{j,i} \]

\[ o_j = f(q_j) \]

and for the output \( \hat{y}_j = q_j \) or \( \hat{y}_j = \sigma(q_j) \).
Neural networks – error

Given set of training pairs \((x^t, y^t), t = 1, \ldots, T\), for each input \(x^t\) forward propagation computes \(\hat{y}^t\).

To measure training error

- **continuous outputs** \(E = \sum_t E_t = \sum_t (y_t - \hat{y}_t)^2\)
- **binary outputs** \(E = \sum_t E_t = -\sum_t y_t \log \hat{y}_t\)

A learning procedure finds weights that minimize this error.
Neural network training

The error functions used in neural network training are usually smooth.

By far the most popular way of training a neural network is gradient descent:

$$w^{(k+1)} = w^{(k)} - \eta \nabla E$$

where $\eta$ is a reasonably small value, learning rate.
Computing gradient of neural network error

*Back-propagation* is an efficient algorithm for computing gradients of neural network errors.

It is a recursive application of chain rule.

\[
E = \sum_t E_t
\]

\[
\frac{\partial E}{\partial w_{j,i}} = \sum_t \frac{\partial E_t}{\partial w_{j,i}}
\]

Since the error is additive across training examples so are the derivatives.

\[
\frac{\partial E_t}{\partial w_{j,i}} = \frac{\partial E_t}{\partial q_j} \frac{\partial q_j}{\partial w_{j,i}} = \frac{\partial E_t}{\partial q_j} a_{j,i}
\]
Derivatives of output unit’s weights

\[
\frac{\partial E_t}{\partial w_{j,i}} = \frac{\partial E_t}{\partial q_j} a_{j,i}
\]

We will assume squared error \( E_t = (\hat{y}_t - y_t)^2 \).

If we assume that output of unit \( k \), \( o_k = \sigma(q_k) \) then

\[
\frac{\partial E_t}{\partial q_k} = \frac{\partial E_t}{\partial o_k} \frac{\partial o_k}{\partial q_k} = \frac{\partial E_t}{\partial o_k} o_k(1 - o_k)
\]

Under assumption of squared error

\[
\frac{\partial E_t}{\partial o_k} = \frac{\partial}{\partial o_k} (o_k - y_t)^2 = 2(o_k - y^t)
\]

Finally we can compute

\[
\frac{\partial E_t}{\partial w_{k,i}} = \frac{\partial E_t}{\partial q_j} a_{j,i} = \frac{\partial E_t}{\partial o_k} o_k(1 - o_k) a_{j,i} = 2(o_k - y^t) o_k(1 - o_k) a_{k,i}
\]
Derivatives of output unit’s weights

\[
\frac{\partial E_t}{\partial w_{k,i}} = 2(o_k - y^t)o_k(1 - o_k)a_{k,i}
\]

To compute these derivatives we need results of forward propagation starting with inputs $x^t$, in particular $o_k$ and $a_{k,i}$. 
Derivatives of hidden unit’s weights

Situation here is a little more complicated since outputs of hidden layer units serve as inputs to other units.

Let $\delta_j = \frac{\partial E_t}{\partial q_j}$

Assume that for each unit $j$ we have a list of units $F_j$ that take output of unit $j$ as input – units $j$ feeds into. For example, a node $j$ in the last hidden layer feeds into an output node.

$$
\delta_j = \frac{\partial E_t}{\partial q_j} = \sum_{i \in F_j} \frac{\partial E_t}{\partial q_i} \frac{\partial q_i}{\partial q_j} = \sum_{i \in F_j} \delta_i \frac{\partial q_i}{\partial q_j}
$$

$$
= \sum_{i \in F_j} \delta_i \frac{\partial q_i}{\partial o_j} \frac{\partial o_j}{\partial q_j}
$$

$$
= \frac{\partial o_j}{\partial q_j} \sum_{i \in F_j} \delta_i w_{i,j}
$$

$$
= o_j(1 - o_j) \sum_{i \in F_j} \delta_i w_{i,j}
$$
Backpropagation

We know how to compute derivative’s of output unit’s weights

\[ \delta_k = \frac{\partial E_t}{\partial q_k} = 2(o_k - y^t)o_k(1 - o_k) \]

Now we can consider the layer \( l \) above the output layer. For each node \( j \) in that layer, \( F_j \) is subset of output nodes.

\[ \delta_j = o_j(1 - o_j) \sum_{i \in F_j} \delta_i w_{i,j} \]

Hence we have all the \( \delta \)'s needed for layer \( l \), and we can proceed in a backward fashion to the top layer.

Once all the \( \delta \)'s are computed we obtain

\[ \frac{\partial E_t}{\partial w_{j,i}} = \delta_j a_{j,i} \]
Backprop algorithm

input : \( w, \text{Train} = \{(x^t, y^t) : t = 1, \ldots, T\} \)
output: Error gradient \( \frac{\partial E}{\partial w} \)

for \( t = 1, 2, \ldots, T \) do

  Forward propagate starting with \( x^t \)
  
  Inputs: \( o_i = x^t_i \)
  
  Hiddens: \( o_j = \sigma(\sum_i w_{ji}o_i) \)

  Propagate deltas backward
  
  Outputs : \( \delta_k = 2(o_k - y^t)o_k(1 - o_k) \)
  
  Hiddens : \( \delta_j = o_j(1 - o_j)\sum_{i \in F_j} \delta_i w_{i,j} \)

  Compute gradients
  
  \( \frac{\partial E_t}{\partial w_{ji}} = \delta_j a_{j,i} \)

end
Convolutional networks

One fairly old, but still popular, approach to structuring neural nets is to use convolutional layers

Neural network architecture for digit recognition.
Understanding convolution
Understanding convolution – example of edge detector

\[ S(i, j) = (I * K)(i, j) = \sum_m \sum_n I(m, n)K(i - m, j - n). \]  \hspace{1cm} (9.4)

\[ S(i, j) = (I * K)(i, j) = \sum_m \sum_n I(i + m, j + n)K(m, n). \]  \hspace{1cm} (9.6)

Image on the left is convolved with kernel

\[ K = \begin{bmatrix} -1 & 1 \end{bmatrix} \]

without flipping (Eq. 9.6)
Another very common building block of deep networks is a pooling layer

$$h_i^k = \max(h_{S_i}^{k-1})$$

where $S_i$ is a subset of units in the previous layer.
COMP 790-125: Goals for today

- Neural Nets
- XOR learning
- Convolution and pooling