These slides were assembled by Byron Boots, with only minor modifications from Eric Eaton’s slides and grateful acknowledgement to the many others who made their course materials freely available online. Feel free to reuse or adapt these slides for your own academic purposes, provided that you include proper attribution.
Neural Networks

• Origins: Algorithms that try to mimic the brain.
• Very widely used in 80s and early 90s; popularity diminished in late 90s.
• Recent resurgence: State-of-the-art technique for many applications
• Artificial neural networks are not nearly as complex or intricate as the actual brain structure
Neural networks

- Neural networks are made up of **nodes** or **units**, connected by **links**
- Each link has an associated **weight** and **activation level**
- Each node has an **input function** (typically summing over weighted inputs), an **activation function**, and an **output**
Neuron Model: Logistic Unit

"bias unit"

\[ h_\theta(x) = g(\theta^T x) \]

\[ = \frac{1}{1 + e^{-\theta^T x}} \]

Sigmoid (logistic) activation function:

\[ g(z) = \frac{1}{1 + e^{-z}} \]
Neural Network

$\mathbf{x}_0$, $\mathbf{x}_1$, $\mathbf{x}_2$, $\mathbf{x}_3$

Layer 1: (Input Layer)
Layer 2: (Hidden Layer)
Layer 3: (Output Layer)

$h_\theta(\mathbf{x}) = \mathbf{a}_3^{(2)}$

Bias units
Feed-Forward Process

• Input layer units are set by some exterior function (think of these as sensors), which causes their output links to be activated at the specified level.

• Working forward through the network, the input function of each unit is applied to compute the input value:
  – Usually this is just the weighted sum of the activation on the links feeding into this node.

• The activation function transforms this input function into a final value:
  – Typically this is a nonlinear function, often a sigmoid function corresponding to the “threshold” of that node.
Neural Network

\[ a_i^{(j)} = \text{“activation” of unit } i \text{ in layer } j \]

\[ \Theta^{(j)} = \text{weight matrix controlling function mapping from layer } j \text{ to layer } j + 1 \]

\[
\begin{align*}
    a_1^{(2)} &= g(\Theta_{10}^{(1)} x_0 + \Theta_{11}^{(1)} x_1 + \Theta_{12}^{(1)} x_2 + \Theta_{13}^{(1)} x_3) \\
    a_2^{(2)} &= g(\Theta_{20}^{(1)} x_0 + \Theta_{21}^{(1)} x_1 + \Theta_{22}^{(1)} x_2 + \Theta_{23}^{(1)} x_3) \\
    a_3^{(2)} &= g(\Theta_{30}^{(1)} x_0 + \Theta_{31}^{(1)} x_1 + \Theta_{32}^{(1)} x_2 + \Theta_{33}^{(1)} x_3) \\
    h_{\Theta}(x) &= a_1^{(3)} = g(\Theta_{10}^{(2)} a_0^{(2)} + \Theta_{11}^{(2)} a_1^{(2)} + \Theta_{12}^{(2)} a_2^{(2)} + \Theta_{13}^{(2)} a_3^{(2)})
\end{align*}
\]
Vectorization

\[
a^{(2)}_1 = g \left( \Theta^{(1)}_{10} x_0 + \Theta^{(1)}_{11} x_1 + \Theta^{(1)}_{12} x_2 + \Theta^{(1)}_{13} x_3 \right) = g \left( z^{(2)}_1 \right) \\
a^{(2)}_2 = g \left( \Theta^{(1)}_{20} x_0 + \Theta^{(1)}_{21} x_1 + \Theta^{(1)}_{22} x_2 + \Theta^{(1)}_{23} x_3 \right) = g \left( z^{(2)}_2 \right) \\
a^{(2)}_3 = g \left( \Theta^{(1)}_{30} x_0 + \Theta^{(1)}_{31} x_1 + \Theta^{(1)}_{32} x_2 + \Theta^{(1)}_{33} x_3 \right) = g \left( z^{(2)}_3 \right) \\
h_\Theta(x) = g \left( \Theta^{(2)}_{10} a^{(2)}_0 + \Theta^{(2)}_{11} a^{(2)}_1 + \Theta^{(2)}_{12} a^{(2)}_2 + \Theta^{(2)}_{13} a^{(2)}_3 \right) = g \left( z^{(3)}_1 \right)
\]

Feed-Forward Steps:

\[
\begin{align*}
z^{(2)} &= \Theta^{(1)} x \\
a^{(2)} &= g(z^{(2)}) \\
\text{Add } a^{(2)}_0 &= 1 \\
z^{(3)} &= \Theta^{(2)} a^{(2)} \\
h_\Theta(x) &= a^{(3)} = g(z^{(3)})
\end{align*}
\]
Other Network Architectures

$L$ denotes the number of layers

$s = [3, 3, 2, 1]$ contains the numbers of nodes at each layer

- Not counting bias units
- Typically, $s_0 = d$ (# input features) and $s_{L-1} = K$ (# classes)
Multiple Output Units: One-vs-Rest

We want:

\[ h_\Theta(x) \approx \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} \]
when pedestrian

\[ h_\Theta(x) \approx \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} \]
when car

\[ h_\Theta(x) \approx \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} \]
when motorcycle

\[ h_\Theta(x) \approx \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \]
when truck

\[ h_\Theta(x) \in \mathbb{R}^K \]
Multiple Output Units: One-vs-Rest

We want:

\[ h_\Theta(x) \approx \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} \quad h_\Theta(x) \approx \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} \quad h_\Theta(x) \approx \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} \quad h_\Theta(x) \approx \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \]

when pedestrian, when car, when motorcycle, when truck

- Given \{ (x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n) \}

- Must convert labels to 1-of-\( K \) representation

\[ y_i = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} \text{ when motorcycle, } y_i = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} \text{ when car, etc.} \]
Neural Network Classification

**Given:**

\[ \{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\} \]

\( s \in \mathbb{N}^L \) contains # nodes at each layer

- \( s_0 = d \) (# features)

**Binary classification**

\( y = 0 \) or \( 1 \)

1 output unit \( (s_{L-1} = 1) \)

**Multi-class classification** \((K\) classes\)

\( y \in \mathbb{R}^K \) e.g. \[
\begin{bmatrix}
1 \\
0 \\
0
\end{bmatrix},
\begin{bmatrix}
0 \\
1 \\
0
\end{bmatrix},
\begin{bmatrix}
0 \\
0 \\
1
\end{bmatrix},
\begin{bmatrix}
0 \\
0 \\
0
\end{bmatrix}
\]

pedestrian car motorcycle truck

\( K \) output units \( (s_{L-1} = K) \)

---

Slide by Andrew Ng
Neural Network Learning
Perceptron Learning Rule

\[ \theta \leftarrow \theta + \alpha (y - h(x)) x \]

Equivalent to the intuitive rules:

– If output is correct, don’t change the weights
– If output is low \((h(x) = 0, y = 1)\), increment weights for all the inputs which are 1
– If output is high \((h(x) = 1, y = 0)\), decrement weights for all inputs which are 1

Perceptron Convergence Theorem:

• If there is a set of weights that is consistent with the training data (i.e., the data is linearly separable), the perceptron learning algorithm will converge [Minicksy & Papert, 1969]
Batch Perceptron

Given training data \( \{ (x^{(i)}, y^{(i)}) \}_{i=1}^{n} \)
Let \( \theta \leftarrow [0, 0, \ldots, 0] \)
Repeat:

1. Let \( \Delta \leftarrow [0, 0, \ldots, 0] \)
2. for \( i = 1 \ldots n \), do
3. if \( y^{(i)} x^{(i)} \theta \leq 0 \) \// prediction for \( i^{th} \) instance is incorrect
4. \( \Delta \leftarrow \Delta + y^{(i)} x^{(i)} \)
5. \( \Delta \leftarrow \Delta / n \) \// compute average update
6. \( \theta \leftarrow \theta + \alpha \Delta \)

Until \( \|\Delta\|_2 < \epsilon \)

- Simplest case: \( \alpha = 1 \) and don’t normalize, yields the fixed increment perceptron
- Each increment of outer loop is called an \textit{epoch}
Learning in NNs: Backpropagation

• Similar to the perceptron learning algorithm, we cycle through our examples
  – If the output of the network is correct, no changes are made
  – If there is an error, weights are adjusted to reduce the error

• The trick is to assess the blame for the error and divide it among the contributing weights
Cost Function

Logistic Regression:

\[
J(\theta) = -\frac{1}{n} \sum_{i=1}^{n} [y_i \log h_\theta(x_i) + (1 - y_i) \log (1 - h_\theta(x_i))] + \frac{\lambda}{2n} \sum_{j=1}^{d} \theta_j^2
\]

Neural Network:

\[
h_\theta \in \mathbb{R}^K \\
(h_\theta(x))_i = i^{th} \text{ output}
\]

\[
J(\Theta) = -\frac{1}{n} \left[ \sum_{i=1}^{n} \sum_{k=1}^{K} y_{ik} \log (h_\Theta(x_i))_k + (1 - y_{ik}) \log \left( 1 - (h_\Theta(x_i))_k \right) \right] \\
+ \frac{\lambda}{2n} \sum_{l=1}^{L-1} \sum_{s_{l-1}=1}^{s_l} \sum_{s_l=1}^{s_l} \left( \Theta^{(l)}_{ji} \right)^2
\]

\(k^{th}\) class: \ true, predicted
\(not\ k^{th}\ class: \ true, predicted\)

Based on slide by Andrew Ng
Optimizing the Neural Network

\[ J(\Theta) = -\frac{1}{n} \left[ \sum_{i=1}^{n} \sum_{k=1}^{K} y_{ik} \log(h_{\Theta}(x_i))_k + (1 - y_{ik}) \log\left(1 - (h_{\Theta}(x_i))_k\right) \right] \]

\[ + \frac{\lambda}{2n} \sum_{l=1}^{L-1} \sum_{s_{l-1}}^{s_l} \sum_{s_l} \left(\Theta_{ji}^{(l)}\right)^2 \]

Solve via: \[ \min_{\Theta} J(\Theta) \]

Need code to compute:
• \( J(\Theta) \)
• \( \frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta) \)

\( J(\Theta) \) is not convex, so GD on a neural net yields a local optimum
• But, tends to work well in practice

Based on slide by Andrew Ng
Forward Propagation

• Given one labeled training instance \((x, y)\):

Forward Propagation

• \(a^{(1)} = x\)
• \(z^{(2)} = \Theta^{(1)}a^{(1)}\)
• \(a^{(2)} = g(z^{(2)})\) [add \(a_0^{(2)}\)]
• \(z^{(3)} = \Theta^{(2)}a^{(2)}\)
• \(a^{(3)} = g(z^{(3)})\) [add \(a_0^{(3)}\)]
• \(z^{(4)} = \Theta^{(3)}a^{(3)}\)
• \(a^{(4)} = h_\Theta(x) = g(z^{(4)})\)
Backpropagation Intuition

• Each hidden node $j$ is “responsible” for some fraction of the error $\delta_j^{(l)}$ in each of the output nodes to which it connects.

• $\delta_j^{(l)}$ is divided according to the strength of the connection between hidden node and the output node.

• Then, the “blame” is propagated back to provide the error values for the hidden layer.
\[ \delta_j^{(l)} = \text{“error” of node } j \text{ in layer } l \]

Formally, \[ \delta_j^{(l)} = \frac{\partial}{\partial z_j^{(l)}} \text{cost}(x_i) \]

where \[ \text{cost}(x_i) = y_i \log h_\Theta(x_i) + (1 - y_i) \log(1 - h_\Theta(x_i)) \]
Backpropagation Intuition

\[ \delta_j^{(l)} = \text{"error" of node } j \text{ in layer } l \]

Formally, \[ \delta_j^{(l)} = \frac{\partial}{\partial z_j^{(l)}} \text{cost}(x_i) \]

where \[ \text{cost}(x_i) = y_i \log h_\Theta(x_i) + (1 - y_i) \log(1 - h_\Theta(x_i)) \]

\[ \delta^{(4)} = a^{(4)} - y \]
Backpropagation Intuition

\[ \delta_j^{(l)} = \text{"error" of node } j \text{ in layer } l \]

Formally, \[
\delta_j^{(l)} = \frac{\partial}{\partial z_j^{(l)}} \text{cost}(x_i)
\]

where \[
\text{cost}(x_i) = y_i \log h_\Theta(x_i) + (1 - y_i) \log(1 - h_\Theta(x_i))
\]

Based on slide by Andrew Ng
Backpropagation Intuition

\[ \delta_j^{(l)} = \text{"error" of node } j \text{ in layer } l \]

Formally, \( \delta_j^{(l)} = \frac{\partial}{\partial z_j^{(l)}} \text{cost}(x_i) \)

where \( \text{cost}(x_i) = y_i \log h_\Theta(x_i) + (1 - y_i) \log(1 - h_\Theta(x_i)) \)

Based on slide by Andrew Ng
\[ \delta_j^{(l)} = \text{"error" of node } j \text{ in layer } l \]

Formally,

\[ \delta_j^{(l)} = \frac{\partial}{\partial z_j^{(l)}} \text{cost}(x_i) \]

where \[ \text{cost}(x_i) = y_i \log h_\Theta(x_i) + (1 - y_i) \log(1 - h_\Theta(x_i)) \]
Backpropagation: Gradient Computation

Let $\delta_j^{(l)} = "\text{error}"$ of node $j$ in layer $l$

(#layers $L = 4$)

Backpropagation

• $\delta^{(4)} = a^{(4)} - y$
• $\delta^{(3)} = (\Theta^{(3)})^T \delta^{(4)} .* g'(z^{(3)})$
• $\delta^{(2)} = (\Theta^{(2)})^T \delta^{(3)} .* g'(z^{(2)})$
• (No $\delta^{(1)}$)

\[
\dfrac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta) = a_j^{(l)} \delta_i^{(l+1)}
\]

(ignoring $\lambda$; if $\lambda = 0$)

Based on slide by Andrew Ng
Backpropagation

Set \( \Delta^{(l)}_{ij} = 0 \quad \forall l, i, j \)

For each training instance \((x_i, y_i)\):

Set \( a^{(1)} = x_i \)

Compute \( \{a^{(2)}, \ldots, a^{(L)}\} \) via forward propagation

Compute \( \delta^{(L)} = a^{(L)} - y_i \)

Compute errors \( \{\delta^{(L-1)}, \ldots, \delta^{(2)}\} \)

Compute gradients \( \Delta^{(l)}_{ij} = \Delta^{(l)}_{ij} + a^{(l)}_j \delta^{(l+1)}_i \)

Compute avg regularized gradient \( D^{(l)}_{ij} = \begin{cases} \frac{1}{n} \Delta^{(l)}_{ij} + \lambda \Theta^{(l)}_{ij} & \text{if } j \neq 0 \\ \frac{1}{n} \Delta^{(l)}_{ij} & \text{otherwise} \end{cases} \)

\( D^{(l)} \) is the matrix of partial derivatives of \( J(\Theta) \)
Training a Neural Network via Gradient Descent with Backprop

Given: training set \( \{(x_1, y_1), \ldots, (x_n, y_n)\} \)
Initialize all \( \Theta^{(l)} \) randomly (NOT to 0!)
Loop // each iteration is called an epoch

Set \( \Delta_{ij}^{(l)} = 0 \ \forall l, i, j \)
For each training instance \((x_i, y_i)\):
  Set \( a^{(1)} = x_i \)
  Compute \( \{a^{(2)}, \ldots, a^{(L)}\} \) via forward propagation
  Compute \( \delta^{(L)} = a^{(L)} - y_i \)
  Compute errors \( \{\delta^{(L-1)}, \ldots, \delta^{(2)}\} \)
  Compute gradients \( \Delta_{ij}^{(l)} = \Delta_{ij}^{(l)} + a_j^{(l)} \delta_i^{(l+1)} \)
Compute avg regularized gradient \( D_{ij}^{(l)} = \begin{cases} \frac{1}{n} \Delta_{ij}^{(l)} + \lambda \Theta_{ij}^{(l)} & \text{if } j \neq 0 \\ \frac{1}{n} \Delta_{ij}^{(l)} & \text{otherwise} \end{cases} \)
Update weights via gradient step \( \Theta_{ij}^{(l)} = \Theta_{ij}^{(l)} - \alpha D_{ij}^{(l)} \)
Until weights converge or max #epochs is reached
Backprop Issues

“Backprop is the cockroach of machine learning. It’s ugly, and annoying, but you just can’t get rid of it.”

—Geoff Hinton

Problems:
• black box
• local minima
Putting It All Together
Training a Neural Network

Pick a network architecture (connectivity pattern between nodes)

- # input units = # of features in dataset
- # output units = # classes

Reasonable default: 1 hidden layer
- or if >1 hidden layer, have same # hidden units in every layer (usually the more the better)
Training a Neural Network

1. Randomly initialize weights
2. Implement forward propagation to get $h_\Theta(x_i)$ for any instance $x_i$
3. Implement code to compute cost function $J(\Theta)$
4. Implement backprop to compute partial derivatives
   $$\frac{\partial}{\partial \Theta^{(l)}_{jk}} J(\Theta)$$
5. Use gradient descent with backprop to fit the network