Neural Networks

These slides were assembled by Eric Eaton, with grateful acknowledgement of 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. Please send comments and corrections to Eric.
Neural Networks

• Origins: Algorithms that try to mimic the brain
• 40s and 50s: Hebbian learning and Perceptron
• Perceptrons book in 1969 and the XOR problem
• 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
Single Node

“bias unit”

\[ x_0 = 1 \]

\[ x_1, x_2, x_3 \]

\[ \theta_0, \theta_1, \theta_2, \theta_3 \]

\[ \sum \int h_\theta(x) = g(\theta^Tx) \]

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

Sigmoid (logistic) activation function:

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

Based on slide by Andrew Ng
Neural Network

bias units

Layer 1 (Input Layer)

Layer 2 (Hidden Layer)

Layer 3 (Output Layer)

$h_\theta(x)$
Neural networks Terminology

- 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.
Feed-Forward Process

• Input layer units are set by external data, 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
\( a_i^{(j)} \) = “activation” of unit \( i \) in layer \( j \)

\( \Theta^{(j)} \) = weight matrix stores parameters from layer \( j \) to layer \( j + 1 \)

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

If network has \( s_j \) units in layer \( j \) and \( s_{j+1} \) units in layer \( j+1 \), then \( \Theta^{(j)} \) has dimension \( s_{j+1} \times (s_j+1) \).

\[
\Theta^{(1)} \in \mathbb{R}^{3 \times 4} \quad \Theta^{(2)} \in \mathbb{R}^{1 \times 4}
\]
Vectorization

\[ a_1^{(2)} = g \left( \Theta_{10}^{(1)} x_0 + \Theta_{11}^{(1)} x_1 + \Theta_{12}^{(1)} x_2 + \Theta_{13}^{(1)} x_3 \right) = g \left( z_1^{(2)} \right) \]

\[ a_2^{(2)} = g \left( \Theta_{20}^{(1)} x_0 + \Theta_{21}^{(1)} x_1 + \Theta_{22}^{(1)} x_2 + \Theta_{23}^{(1)} x_3 \right) = g \left( z_2^{(2)} \right) \]

\[ a_3^{(2)} = g \left( \Theta_{30}^{(1)} x_0 + \Theta_{31}^{(1)} x_1 + \Theta_{32}^{(1)} x_2 + \Theta_{33}^{(1)} x_3 \right) = g \left( z_3^{(2)} \right) \]

\[ h_\Theta(x) = g \left( \Theta_{10}^{(2)} a_0^{(2)} + \Theta_{11}^{(2)} a_1^{(2)} + \Theta_{12}^{(2)} a_2^{(2)} + \Theta_{13}^{(2)} a_3^{(2)} \right) = g \left( z_1^{(3)} \right) \]

Feed-Forward Steps:

\[ z^{(2)} = \Theta^{(1)} x \]

\[ a^{(2)} = g(z^{(2)}) \]

Add \[ a_0^{(2)} = 1 \]

\[ z^{(3)} = \Theta^{(2)} a^{(2)} \]

\[ h_\Theta(x) = a^{(3)} = g(z^{(3)}) \]
Other Network Architectures

$L$ denotes the number of layers

$s \in \mathbb{N}^+^L$ 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:

\[
\begin{align*}
\hat{h}_\Theta(x) &\approx \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} & \text{when pedestrian} \\
\hat{h}_\Theta(x) &\approx \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} & \text{when car} \\
\hat{h}_\Theta(x) &\approx \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} & \text{when motorcycle} \\
\hat{h}_\Theta(x) &\approx \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} & \text{when truck}
\end{align*}
\]
Multiple Output Units: One-vs-Rest

We want:

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

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

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

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

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

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

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

**Binary classification**

\[ y = 0 \text{ or } 1 \]

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

**Multi-class classification** \( (K \text{ classes}) \)

\[ y \in \mathbb{R}^K \]

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

---

Given:

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

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

\[ s_0 = d \] \( (# \text{ features}) \)
Understanding Representations
Representing Boolean Functions

**Simple example: AND**

\[ x_1, x_2 \in \{0, 1\} \]

\[ y = x_1 \text{ AND } x_2 \]

\[ h_\theta(x) = g(-30 + 20x_1 + 20x_2) \]

<table>
<thead>
<tr>
<th>( x_1 )</th>
<th>( x_2 )</th>
<th>( h_\theta(x) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>( g(-30) \approx 0 )</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>( g(-10) \approx 0 )</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>( g(-10) \approx 0 )</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>( g(10) \approx 1 )</td>
</tr>
</tbody>
</table>

Based on slide and example by Andrew Ng
Representing Boolean Functions

\[ h_\theta(x) = 1 + e^T x \]

**AND**

- \( x_1 \): +20
- \( x_2 \): +20
- +1

**OR**

- \( x_1 \): +20
- \( x_2 \): +20
- +1

**NOT**

- \( x_1 \): -20
- +1

**(NOT \( x_1 \)) AND (NOT \( x_2 \))**

- \( x_1 \): -20
- \( x_2 \): -20
- +1
Combining Representations to Create Non-Linear Functions

AND

\[ h_\theta(x) = 1 + e^{T_x} \]

\[ h_\theta(x) = 1 + e^{T_x} \]

\[ h_\theta(x) = 1 + e^{T_x} \]

\[ h_\theta(x) = 1 + e^{T_x} \]

(NOT \( x_1 \)) AND (NOT \( x_2 \))

OR

NOT XOR

I or III

Based on example by Andrew Ng
Layering Representations

| 79 65 87 44 18 |
| 07 33 24 84 51 |
| 63 29 13 32 62 |
| 13 71 56 52 42 |
| 09 27 58 95 45 |
| 46 65 02 13 69 |
| 85 18 97 87 36 |
| 10 28 30 51 15 |
| 67 82 53 97 00 |
| 79 39 85 72 98 |

20 × 20 pixel images

\( d = 400 \) 10 classes

Each image is “unrolled” into a vector \( \mathbf{x} \) of pixel intensities
Layering Representations

Input Layer

Hidden Layer

Output Layer

x₁
x₂
x₃
x₄
x₅
xₐ

“0”
“1”
“9”
Neural Network Learning
Perceptron Learning Rule

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

Intuitive rule:
- 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 [Minksy & Papert, 1969]
Batch Perceptron

Given training data \( \{(x^{(i)}, y^{(i)})\}_{i=1}^{n} \)

Let \( \theta \leftarrow [0, 0, \ldots, 0] \)

Repeat:

Let \( \Delta \leftarrow [0, 0, \ldots, 0] \)

for \( i = 1 \ldots n \), do

if \( y^{(i)} x^{(i)} \theta \leq 0 \) \hspace{1cm} // prediction for \( i^{th} \) instance is incorrect

\[ \Delta \leftarrow \Delta + y^{(i)} x^{(i)} \]

\[ \Delta \leftarrow \Delta / n \] \hspace{1cm} // compute average update

\[ \theta \leftarrow \theta + \alpha \Delta \]

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 epoch
Learning in NN: 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

• We are just performing (stochastic) gradient descent!
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:

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

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

\[k^{th} \text{ class: true, predicted not } k^{th} \text{ class: true, predicted}\]
Optimizing the Neural Network

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

\[ + \frac{\lambda}{2n} \sum_{l=1}^{L-1} \sum_{i=1}^{s_{l-1}} \sum_{j=1}^{s_l} (\Theta^{(l)}_{ji})^2 \]

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

Unlike before, \( J(\Theta) \) is not convex, so GD on a neural net yields a local optimum

\[ \frac{\partial}{\partial \Theta^{(l)}_{ij}} J(\Theta) = a^{(l)}_j \delta^{(l+1)}_i \]

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

Based on slide by Andrew Ng
Forward Propagation

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

Forward Propagation

• \(a^{(1)} = x\)
• \(z^{(2)} = \Theta^{(1)}a^{(1)}\)
• \(a^{(2)} = g(z^{(2)}) \quad \text{[add } a_0^{(2)}]\)
• \(z^{(3)} = \Theta^{(2)}a^{(2)}\)
• \(a^{(3)} = g(z^{(3)}) \quad \text{[add } a_0^{(3)}]\)
• \(z^{(4)} = \Theta^{(3)}a^{(3)}\)
• \(a^{(4)} = h_\Theta(x) = g(z^{(4)})\)
\[ \delta_j^{(l)} = \text{“error” of node } j \text{ in layer } l \]

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
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 \( \text{cost}(x_i) = y_i \log h_\Theta(x_i) + (1 - y_i) \log(1 - h_\Theta(x_i)) \)
\[ \delta_j^{(l)} = \text{"error" of node } j \text{ in layer } l \]

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
Backpropagation: Gradient Computation

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

(#layers $L = 4$)

Backpropagation

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

Element-wise product. *

Based on slide by Andrew Ng
Backpropagation

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

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 $\Delta_{ij}^{(l)} = \Delta_{ij}^{(l)} + a_j^{(l)} \delta_i^{(l+1)}$

Compute avg regularized gradient $D_{ij}^{(l)} = \begin{cases} \frac{1}{n} \Delta_{ij}^{(l)} + \lambda \Theta_{ij}^{(l)} & \text{if } j \neq 0 \\ \frac{1}{n} \Delta_{ij}^{(l)} & \text{otherwise} \end{cases}$

$D^{(l)}$ 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 \( \Delta_{ij}^{(l)} = 0 \quad \forall l, i, j \) (Used to accumulate gradient)

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

---

Backpropagation

Based on slide by Andrew Ng
Several Practical Tricks

Initialization

- Problem is highly non-convex, and heuristics exist to start training (at the least, randomize initial weights)

Optimization tricks

- Momentum-based methods
- Decaying step size
- Dropout to avoid co-adaptation / overfitting

Minibatch

- Use more than a single point to estimate gradient
Neural Networks vs Deep Learning?

DNN are big neural networks

• Depth: often ~5 layers (but some have 20+)
  – Typically not fully connected!

• Width: hidden nodes per layer in the thousands

• Parameters: millions to billions

Algorithms / Computing

• New algorithms (pre-training, layer-wise training, dropout, etc.)

• Heavy computing requirements (GPUs are essential)