Neural Networks

Prof. Balaraman Ravindran

ID5055: Foundations of Machine Learning

Biology of a Neuron

Biology of a Neuron

Neural Function

- Brain function (thoughts) occurs as the result of the firing of **neurons.**
- Neurons connect to each other through synapses, which propagate **action potential** (*electrical impulses*) by releasing neurotransmitters
  - Synapses can be **excitatory** (potential-increasing) or **inhibitory** (potential-decreasing), and have varying *activation thresholds*
  - Learning occurs as a result of the synapses' plasticity:

    They exhibit long-term changes in connection strength
- There are about 10^{11} neurons and about 10^{14} synapses in the human brain!

Comparison of Computing Power

INFORMATION CIRCA 2012	Computer	Human Brain
Computation Units	10-core Xeon: 10^9 Gates	10^{11} Neurons
Storage Units	10^9 bits RAM, 10^{12} bits disk	10^{11} neurons, 10^{14} synapses
Cycle time	10^{-9} sec	10^{-3} sec
Bandwidth	10^9 bits/sec	10^{14} bits/sec

- Computers are way faster than neurons…
- But there are a lot more neurons than we can reasonably model in modern digital computers, and they all fire in parallel
- The brain is effectively a billion times faster.
- “Neurons that fire together, wire together” — Donald Hebb.

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

Layered feed-forward network

- 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

Neural Model

The Perceptron

$$\mathbf{x} = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} \qquad \theta = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \theta_2 \\ \theta_3 \end{bmatrix}$$

$$h_\theta(\mathbf{x}) = \begin{cases} 1 & \textit{if } \sum_i \theta_i x_i \geq T \\ 0 & \textit{else} \end{cases}$$

Neural Model

The Perceptron

$$\mathbf{x} = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} \theta = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \theta_2 \\ \theta_3 \end{bmatrix}$$

$$h_\theta(\mathbf{x}) = \begin{cases} 1 & \textit{if } \sum_i \theta_i x_i \geq T \\ 0 & \textit{else} \end{cases}$$

- A Threshold Unit
  - "Fires" if the weighted sum of inputs exceeds a threshold (T)
  - Electrical Engineers may refer to this as a *Threshold Gate*

Neural Model

The “Soft” Perceptron (aka Logistic Unit)

$$\mathbf{x} = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} \qquad \theta = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \theta_2 \\ \theta_3 \end{bmatrix}$$

$$h_\theta(\mathbf{x}) = g(\theta^T \mathbf{x}) = \frac{1}{1 + e^{-\theta^T \mathbf{x}}}$$

Sigmoid activation function: $g(z) = \dfrac{1}{1 + e^{-z}}$

Neural Model

Logistic Unit

$$\mathbf{x} = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} \quad \theta = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \theta_2 \\ \theta_3 \end{bmatrix}$$

$$\rightarrow h_\theta(\mathbf{x}) = g(\theta^T \mathbf{x}) = \frac{1}{1 + e^{-\theta^T \mathbf{x}}}$$

Sigmoid activation function: $g(z) = \dfrac{1}{1 + e^{-z}}$

Feed Forward Process

- Input layer units are set by some external function (Eg: Real-world data)
- In the feed-forward process through the network, the input function of each unit is utilised to compute the input value.
  - Usually this is the <u>weighted sum of the activations</u> on the links feeding into this node.
- The activation function transforms this input function into a final value.
  - Typically this is a nonlinear function (sigmoid), corresponding to the “threshold” of that node.

Neural Network

$$a_1^{(2)} = g(\theta_{10}^{(1)}x_0 + \theta_{11}^{(1)}x_1 + \theta_{12}^{(1)}x_2 + \theta_{13}^{(1)}x_3)$$
$$a_2^{(2)} = g(\theta_{20}^{(1)}x_0 + \theta_{21}^{(1)}x_1 + \theta_{22}^{(1)}x_2 + \theta_{23}^{(1)}x_3)$$
$$a_3^{(2)} = g(\theta_{30}^{(1)}x_0 + \theta_{31}^{(1)}x_1 + \theta_{32}^{(1)}x_2 + \theta_{33}^{(1)}x_3)$$

$$h_\theta(x) = a_1^{(3)} = g(\theta_{10}^{(2)}a_0^{(2)} + \theta_{11}^{(2)}a_1^{(2)} + \theta_{12}^{(2)}a_2^{(2)} + \theta_{13}^{(2)}a_3^{(2)})$$

$a_i^{(j)}$ = "activation" of unit i in layer j

$\theta^{(j)}$ = weight matrix controlling function mapping from layer j to layer $j+1$

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

$$\theta^{(1)} \in \mathbb{R}^{3\times 4} \quad \theta^{(2)} \in \mathbb{R}^{1\times 4}$$

$$\frac{\partial h_\theta}{\partial \theta_{31}^{(1)}} = \boxed{\frac{\partial h_\theta(x)}{\partial a_3^{(2)}}} \frac{\partial a_3^{(2)}}{\partial \theta_{31}^{(1)}}$$

Representing Boolean Functions

Neural Network

Representing Boolean Functions

Example: AND

$x_1, x_2 \in \{0,1\}$
$y = x_1 \mathbf{AND} x_2$

x_1	x_2	Output
0	0	0
0	1	0
1	0	0
1	1	1

$$h_\theta(\mathbf{x}) = g(-30 + 20x_1 + 20x_2)$$

Logistic / Sigmoid Function

x_1	x_2	$h_\Theta(\mathbf{x})$
0	0	$g(-30) \approx 0$
0	1	$g(-10) \approx 0$
1	0	$g(-10) \approx 0$
1	1	$g(10) \approx 1$

Neural Network

Representing Boolean Functions

Perceptron Learning Rule

(Recap)

$$\theta \leftarrow \theta + \alpha(y - h(\mathbf{x}))\mathbf{x}$$

- 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

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
- The trick is to assess the blame for the error and divide it among the contributing weights.

Cost Function

- Logistic Regression:

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

- Neural Network:

$$h_\theta \in \mathbb{R}^K \qquad (h_\theta(\mathbf{x}))_i = i^{th}\,output$$

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

Cost Function

$h_\theta(x_i)$ $\cdot\theta^{(2)}\cdot a$

- Neural Network:

$$h_\theta \in \mathbb{R}^K \qquad (h_\theta(\mathbf{x}))_i = i^{th}\,output$$

$$J(\theta) = -\frac{1}{n}\sum_{i=1}^{n}\sum_{k=1}^{K}[y_{ik}\log(h_\theta(\mathbf{x_i}))_k + (1-y_{ik})\log(1-(h_\theta(\mathbf{x_i}))_k)] + \frac{\lambda}{2n}\sum_{l=1}^{L-1}\sum_{i=1}^{s_l-1}\sum_{j=1}^{s_l}(\theta_{ji}^{(l)})^2$$

- Solve: $\min_\theta J(\theta)$

$J(\theta)$ is may not be convex

So, GD on a Neural Network yields a local optimum.

Forward Propagation

- Given one labeled training instance (x, y):
- **Forward Propagation:**
  - $\mathbf{a}^{(1)} = \mathbf{x}$
  - $\mathbf{z}^{(2)} = \theta^{(1)}\mathbf{a}^{(1)}$
  - $\mathbf{a}^{(2)} = g(\mathbf{z}^{(2)})$
  - $\mathbf{z}^{(3)} = \theta^{(2)}\mathbf{a}^{(2)}$
  - $\mathbf{a}^{(3)} = g(\mathbf{z}^{(3)})$
  - $\mathbf{z}^{(4)} = \theta^{(3)}\mathbf{a}^{(3)}$
  - $\mathbf{a}^{(4)} = h_\theta(\mathbf{x}) = g(\mathbf{z}^{(4)})$

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.

Backpropagation

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

$$\delta_j^{(l)} = \frac{\partial}{\partial z_j^{(l)}} \text{cost}(\mathbf{x}_i)$$

where, $\text{cost}(\mathbf{x}_i) = y_i \log h_\theta(\mathbf{x}_i)) + (1 - y_i)\log(1 - h_\theta(\mathbf{x}_i))$

Backpropagation

In Regression Setting

- Let,
  - 1st Layer: $\alpha_0, \alpha_1, \alpha_2, \ldots, \alpha_M$
  - 2nd Layer: $\beta_0, \beta_1, \beta_2, \ldots, \beta_K$
- Regression: $R(\theta) = sum_{k=1}^{K} sum_{i=1}^{M} (y_{ik} - f_k(x_i))^2$
- Let, $z_{mi} = \sigma(\alpha_0 + \alpha_m^T x_i)$, where $z_i = (z_{1i}, z_{2i}, \ldots, z_{Mi})$ corresponds to the entire activation of the hidden layer for the ith input.
  - $f_k(x_i) = g_k(\beta_0 + \beta_k^T z_i)$

Backpropagation

In Regression Setting

- Regression: $R(\theta) = sum_{k=1}^{K} sum_{i=1}^{M} (y_{ik} - f_k(x_i))^2$
- Let, $z_{mi} = \sigma(\alpha_0 + \alpha_m^T x_i)$, where $z_i = (z_{1i}, z_{2i}, \ldots, z_{Mi})$ corresponds to the entire activation of the hidden layer for the ith input.
  - $f_k(x_i) = g_k(\beta_0 + \beta_k^T z_i)$
- Taking the derivative of the Error, w.r.t. single output weight:
  - $\dfrac{\partial R_i(\theta)}{\partial \beta_{km}} = -2(y_{ik} - f_k(x_i)) \cdot g_k'(\beta_k^T z_i) \cdot z_{mi}$

Backpropagation

In Regression Setting

- Taking the derivative of the Error, w.r.t. single output weight:
  - $$\frac{\partial R_i(\theta)}{\partial \beta_{km}} = -2(y_{ik} - f_k(x_i)) \cdot g_k'(\beta_k^T z_i) \cdot z_{mi}$$
  - $$\frac{\partial R_i(\theta)}{\partial \alpha_{ml}} = \frac{\partial R_i(\theta)}{\partial z_{mi}} \cdot \frac{\partial z_{mi}}{\partial \alpha_{ml}}$$

$$\frac{\partial z_{mi}}{\partial \alpha_{ml}} = \sigma'(\alpha_m^T x_i) \cdot x_{il}$$

$$\frac{\partial R_i(\theta)}{\partial z_{mi}} = \sum_{k=1}^{K} \frac{\partial R_i(\theta)}{\partial f_k} \cdot \frac{\partial f_k}{\partial z_{mi}}$$

Backpropagation

In Regression Setting

- $\dfrac{\partial R_i(\theta)}{\partial \alpha_{ml}} = \dfrac{\partial R_i(\theta)}{\partial z_{mi}} \cdot \dfrac{\partial z_{mi}}{\partial \alpha_{ml}}$
- $\dfrac{\partial R_i(\theta)}{\partial z_{mi}} = \sum_{k=1}^{K} -2(y_{ik} - f_k(x_i)) \cdot g_k'(\beta_k^T z_i)\beta_{km}$
- $\dfrac{\partial R_i(\theta)}{\partial \alpha_{ml}} = \sum_{k=1}^{K} -2(y_{ik} - f_k(x_i)) \cdot g_k'(\beta_k^T z_i)\beta_{km} \cdot \sigma'(\alpha_m^T x_i) \cdot x_{il}$
- $\delta_{ki} = -2(y_{ik} - f_k(x_i)) \cdot g_k'(\beta_k^T z_i)$; $\rho_{mi} = \sigma'(\alpha_m^T x_i) \sum_{k=1}^{K} \beta_{km}\delta_{ki}$
- $\dfrac{\partial R_i}{\partial \beta_{km}} = \delta_{ki} z_{mi}$; $\dfrac{\partial R_i}{\partial \alpha_{ml}} = \rho_{mi} x_{il}$

Backpropagation

In Regression Setting

- $\beta_{km}^{(r+1)} = \beta_{km}^{(r)} - \eta \sum_{i=1}^{N} \delta_{ki} z_{mi}$
- $\alpha_{ml}^{(r+1)} = \alpha_{ml}^{(r)} - \eta \sum_{i=1}^{N} \rho_{mi} x_{il}$

Backpropagation: Gradient Computation

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

- **Backpropagation**
  - $\delta^{(4)} = \mathbf{a}^{(4)} - \mathbf{y}$
  - $\delta^{(3)} = (\theta^{(3)})^T \delta^{(4)} \,.*\, g'(\mathbf{z}^{(3)})$
  - $\delta^{(2)} = (\theta^{(2)})^T \delta^{(3)} \,.*\, g'(\mathbf{z}^{(2)})$
  - (NO $\delta^{(1)}$) .

Element-wise product

$$\frac{\partial}{\partial \theta_{ij}^{(l)}} J(\theta) = a_j^{(l)} \delta_i^{(l+1)}$$

(ignoring λ; i.e. λ = 0)

Backpropagation

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

For each training instance $(\mathbf{x}_i, y_i)$:

Set $\mathbf{a}^{(1)} = \mathbf{x}_i$

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

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

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

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 $\{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)\}$

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 $(\mathbf{x}_i, y_i)$:

Set $\mathbf{a}^{(1)} = \mathbf{x}_i$

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

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

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

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

Putting It All Together

Training a Neural Network

- Randomly initialize weights
- Implement forward propagation to get $h_\theta(\mathbf{x}_i)$ for any instance $\mathbf{x}_i$
- Implement code to compute cost function $J(\theta)$
- Implement backprop to compute partial derivatives $\frac{\partial}{\partial \theta_{jk}^{(l)}} J(\theta)$
- Use gradient checking to compare $\frac{\partial}{\partial \theta_{jk}^{(l)}} J(\theta)$, computed using backpropagation vs. the numerical gradient estimate.
    - Then, disable gradient checking code.
- Use gradient descent with backprop to fit the network.