

# Learning with Neural Networks

Sidharth Baskaran

June 2021

## Backpropagation Cost Function

- Notation
  - $L$  = total no. of layers in network
  - $s_l$  = no. of units excluding bias in layer  $l$
- Classification types
  - Binary -  $y \in \{0, 1\}$
  - Multi-class -  $y \in \mathbb{R}^K$  where there are  $K$  output units ( $h_{\Theta}(x) \in \mathbb{R}^K : K \geq 3$ )
- Cost function with regularization
  - Let  $(h_{\Theta}(x))_i$  be  $i$ th output

$$J(\Theta) = -\frac{1}{m} \sum_{i=1}^m \sum_{k=1}^K \left[ y_k^{(i)} \log(h_{\Theta}(x))_k + (1 - y_k^{(i)}) \log(1 - (h_{\Theta}(x))_k) \right] + \frac{\lambda}{2m} \sum_{l=1}^{L-1} \sum_{i=1}^{s_l} \sum_{j=1}^{s_{l+1}} (\theta_{j,i}^{(l)})^2$$

## Backpropagation Algorithm

- Need to find  $\min_{\Theta} J(\Theta)$ 
  - Compute  $J(\Theta)$  and  $\frac{\partial}{\partial \Theta_{ij}^{(l)}}$
- Gradient **forward propagation** computation given 1 example  $(x, y)$ 
  - $a^{(1)} = x$
  - $z^{(2)} = \Theta^{(1)} a^{(1)}$
  - $a^{(2)} = g(z^{(2)})$  (add  $a_0^{(2)}$ )
  - etc.
- Intuition - compute  $\delta_j^{(l)}$  is error of node  $j$  in layer  $l$ 
  - If output unit is layer  $i$ ,  $\delta_j^{(i)} = a_j^{(i)} - y_j$ 
    - \*  $a_j^{(4)} = (h_{\Theta}(x))_j$
  - $\delta^{(i)} = (\Theta^{(i)})^T \delta^{(i+1)} \cdot g'(z^{(i)})$ 
    - \* Can be shown that  $g'(z^{(i)}) = a^{(i)} \cdot (1 - a^{(i)})$
    - \* No  $\delta^{(1)}$  term because input features don't have error
- General training set of  $m$  examples  $\{(x^{(1)}, y^{(1)}), \dots, (x^{(m)}, y^{(m)})\}$ 
  - Set  $\Delta_{ij}^{(l)} = 0 \forall l, i, j$  (used to compute the partial derivative of  $J(\Theta)$ )

## Algorithm

For  $i = 1 \rightarrow m$  ( $\leftarrow (x^{(i)}, y^{(i)})$ ):

- Set  $a^{(1)} = x^{(i)}$
- Forward propagation to compute  $a^{(l)}$  for  $l \in \text{range}(2, 3, \dots, L)$

- Use  $y^{(i)}$  to compute  $\delta^{(L)} - y^{(i)}$
- Back propagation to compute  $\delta^{(L-1)}, \dots, \delta^{(2)}$  (no  $\delta^{(1)}$ )
- $\Delta_{ij}^{(l)} := \Delta_{ij}^{(l)} + a_j^{(l)} \delta_i^{(l+1)} \longleftrightarrow \Delta^{(l)} := \Delta^{(l)} + \delta^{(l+1)}(a^{(l)})^T$

Then

$$D_{ij}^{(l)} := \frac{1}{m} (\Delta_{ij}^{(l)} + \lambda \Theta_{ij}^{(l)}) | j \neq 0$$

$$D_{ij}^{(l)} := \frac{1}{m} \Delta_{ij}^{(l)} | j = 0$$

- Can then show that  $\boxed{\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta)} = D_{ij}^{(l)}$
- Use this partial derivative in an algorithm like gradient descent or `fminunc`

## Backpropagation Intuition

- Forward propagation - L to R
  - Inputs from data set fed into 1st layer
  - Sigmoid applied to  $z$  values to get activation values in each layer
  - Weights (arrow/lines)  $\times$  originating activation values (a dot product)  $\rightarrow$  new  $z$ -value to which sigmoid is applied
- Backpropagation - R to L
  - Let  $i$ th training example have a cost  $\text{cost}(i) = y^{(i)} \log h_{\Theta}(x^{(i)}) + (1 - y^{(i)}) \log h_{\Theta}(x^{(i)})$ 
    - \* Measure of how well the network is performing on example  $i$
    - \* Ignore regularization, so  $\lambda = 0$
  - $\delta_j^{(l)}$  is error of cost for  $a_j^{(l)}$
  - Formally,  $\delta_j^{(l)} = \frac{\partial}{\partial z_j^{(l)}} \text{cost}(i)$  for  $j \geq 0$  so **delta values are derivative of the cost function**
    - \* Steeper slope means more incorrect, so is defined as  $\delta$
- Backpropagation in the sense that an error of a unit in previous layer is weighted sum (using  $\Theta$ ) of errors in current layer that are edged to the unit in question
  - Opposite of forward-propagation

## Implementation note - unrolling parameters

- Process of unrolling parameters from matrices  $\rightarrow$  vectors

Optimization routine:

```
function [jval, gradient] = costFunction(theta)
...
optTheta = fminunc(@costFunction, initialTheta, options)
```

Neural network with ex.  $L = 4$ :  $\Theta^{(1)}, \Theta^{(2)}, \Theta^{(3)} \rightarrow$  matrices `Theta1`, `Theta2`, `Theta3`  $D^{(1)}, D^{(2)}, D^{(3)} \rightarrow$  matrices `D1`, `D2`, `D3`

Unroll into vectors:

```
thetaVec = [ Theta1(:); Theta2(:); Theta3(:) ];
gradientVec = [ D1(:); D2(:); D3(:) ];
```

Getting back to matrices:

```
Theta1 = reshape(thetaVec(1:110), 10, 11);
Theta2 = reshape(thetaVec(111:220), 10, 11);
Theta3 = reshape(thetaVec(221:231), 1, 11);
```

Can then do

```
function [jval, gradientVec] = costFunction(thetaVec);
```

Use unrolling and forward/back propagation to get  $D^{(i)}$  and  $J(\Theta)$

- Example NN
  - $s_1 = 10, s_2 = 10, s_3 = 1$
  - $\Theta^{(1)} \in \mathbb{R}^{10 \times 11}, \Theta^{(2)} \in \mathbb{R}^{10 \times 11}, \Theta^{(3)} \in \mathbb{R}^{1 \times 11}$
  - $D^{(1)} \in \mathbb{R}^{10 \times 11}, D^{(2)} \in \mathbb{R}^{10 \times 11}, D^{(3)} \in \mathbb{R}^{1 \times 11}$

## Gradient Checking

- Allows to check implementation of backprop
- Numerical estimation of gradients
  - $\frac{d}{d\theta} J(\theta) \approx \frac{J(\theta+\epsilon) - J(\theta-\epsilon)}{2\epsilon}$  where  $\epsilon = 10^{-4}$
  - Implement this as some `gradApprox` in Octave
- If  $\theta = [\theta_1, \dots, \theta_n] \in \mathbb{R}^n$ , then the  $n$ th partial derivative is  $\frac{\partial}{\partial \theta_n} J(\theta) \approx \frac{J(\theta_1, \dots, \theta_n + \epsilon) - J(\theta_1, \dots, \theta_n - \epsilon)}{2\epsilon}$

Algorithm:

```
for i = 1:n,  
    % setting theta +/- epsilon  
    thetaPlus = theta;  
    thetaPlus(i) = thetaPlus(i) + EPSILON;  
    thetaMinus = theta;  
    thetaMinus(i) = thetaMinus(i) - EPSILON;  
    % grad approximation  
    gradApprox(i) = (J(thetaPlus) - J(thetaMinus))/(2 * EPSILON)  
end  
% example check  
gradApprox - DVec <= 10^(-4) % precision desired
```

- Implementation notes
  - Implement backprop to compute `DVec` (unrolled  $D$  matrices)
  - Implement `gradApprox` through algorithm and check with gradient
  - Turn off checking and use backprop for learning
  - Make sure to disable gradient check  $\rightarrow$  slows down code significantly due to numerical gradient computations on every optimization iteration - **reason for using backprop**

## Random Initialization

- Pick initial value for  $\Theta$ 
  - Can not initialize to 0, makes activations, errors, and partials equivalent from 1st to 2nd layer
  - All hidden features compute same function of input  $\rightarrow$  redundant
- Initialize each  $\Theta_{ij}^{(l)}$  to random value  $\in [-\epsilon, \epsilon]$
- Values should be close to 0 but not 0
- Need to break symmetry

```
Theta1 = rand(10,11) * (2 * INIT_EPSILON) - INIT_EPSILON;  
Theta2 = rand(1,11) * (2 * INIT_EPSILON) - INIT_EPSILON;
```

## Overall Implementation Process

- Network architecture - connectivity between neurons

- No. of input units - dimension of features  $x^{(i)}$
- No. of output units - number of classes
  - \*  $y \in \mathbb{R}^K$  where there are  $K$  classes
- Default - 1 hidden layer and if  $\geq 1$ , equal number of units in each layer
  - \* More is better but balance with cost function
- Training neural network steps
  - Randomly initialize weights
  - Implement forward propagation to get  $h_{\Theta}(x^{(i)})$  for any  $x^{(i)}$
  - Compute  $J(\Theta)$
  - Implement backprop to get partial derivatives of  $J(\Theta)$
  - Use gradient checking to compare backpropagation partial with numerical estimate of  $\nabla J(\Theta)$ , then disable check
  - Use optimization routine to minimize  $J(\Theta)$

`for i = 1:m,`

`Perform forward propagation and backpropagation using example (x(i),y(i))`

`(Get activations a(l) and delta terms d(l) for l = 2,...,L`

- For NN  $\rightarrow J(\Theta)$  is **non-convex** so has local extrema
  - Ideally want  $h_{\Theta}(x^{(i)}) \approx y^{(i)}$